# Distributed Commit

## Курс «Базы данных»

Цесько Вадим Александрович
http://incubos.org
@incubos

Computer Science Center

23 сентября 2013 г.

# Содержание

# Two-Phase Commit Protocol

- Distributed atomic commitment protocol
- `commit` или `abort` (`rollback`)
- Переживает (*некоторые*) временные сбои
- Полагается на логгирование для восстановления
- Восстановление — бОльшая часть логики протокола

# Фазы

- Commit-request (voting)
  - **Координатор** пытается подготовить участников транзакции
  - Каждый участник отвечает Yes или No
- Commit
  - Координатор принимает решение на основе ответов
  - Все сказали Yes $\Rightarrow$ `commit`, No $\Rightarrow$ `abort`
  - Координатор отправляет решение участникам
  - Участники применяют решение

**Предусловие**

Если нет сбоев

# Предположения

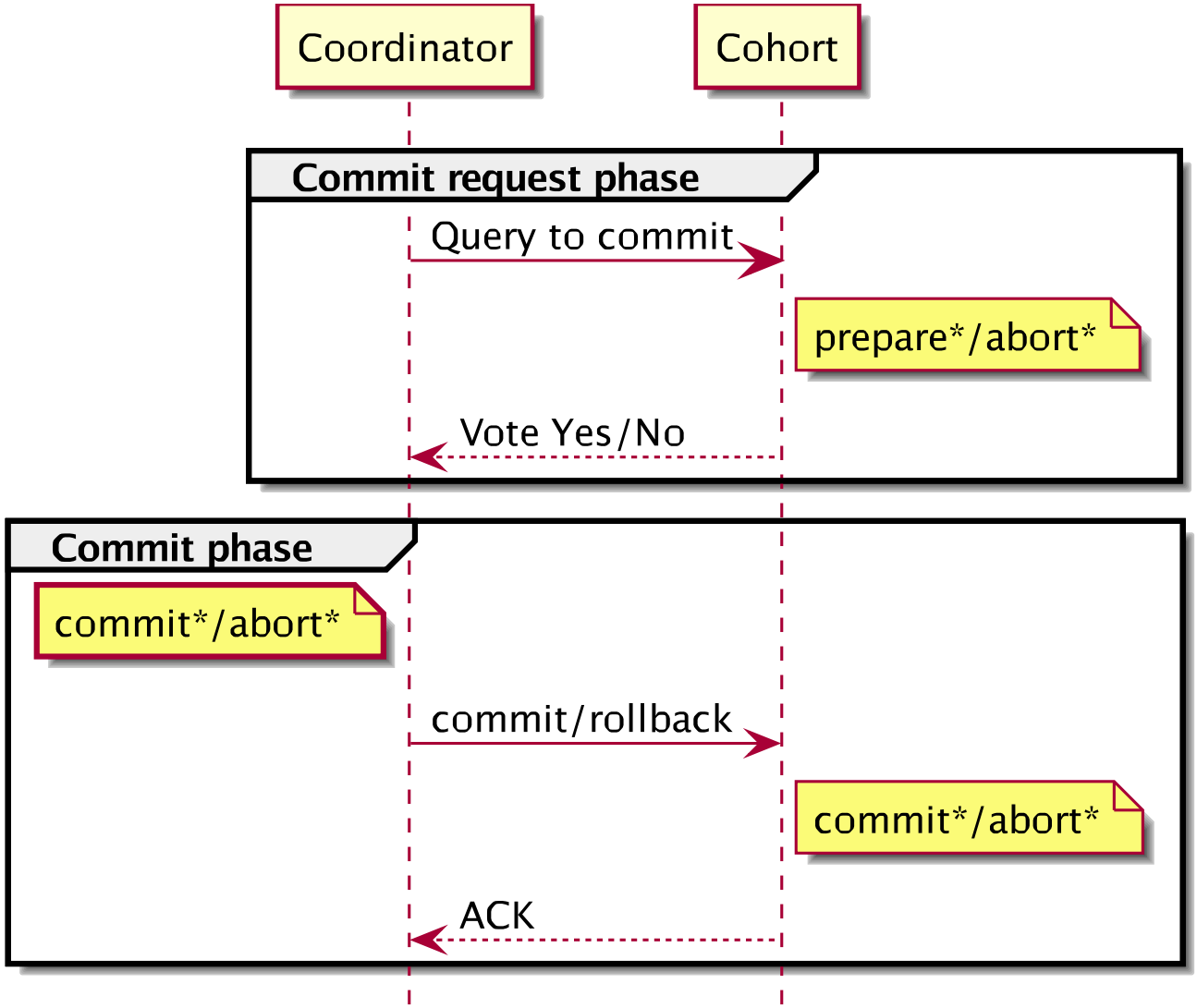
- Выделенный координатор
- Stable storage[1] + write-ahead log (WAL)
- Узлы не умирают навсегда
- Данные в WAL не теряются и не портятся
- Любые два узлы могут общаться

**Внимание**

Если полностью уничтожить узел, можно потерять данные

[1] http://en.wikipedia.org/wiki/Stable_storage

# Диаграмма


Coordinator
Cohort
Commit request phase
Query to commit
prepare*/abort*
Vote Yes/No
Commit phase
commit*/abort*
commit/rollback
commit*/abort*
ACK

# Оптимизации

- **Presumed** `abort`/`commit`: работает при восстановлении, экономим на логгировании, используем статистику и ожидания
- **Tree two-phase commit protocol** (Nested/Recursive 2PC): агрегация в узлах дерева, экономнее используем сеть
- **Dynamic two-phase commit**: идём по дереву в одну сторону, динамический выбор координатора, быстрее освобождаем ресурсы

# Ограничения

## Блокирующийся протокол

Если координатор исчезнет, то некоторые участники могут не закончить транзакцию:

- Участник отправил координатору Yes
- Координатор упал
- Участник ожидает `commit` или `rollback`

# Поведение при сбоях

Пример ситуации:

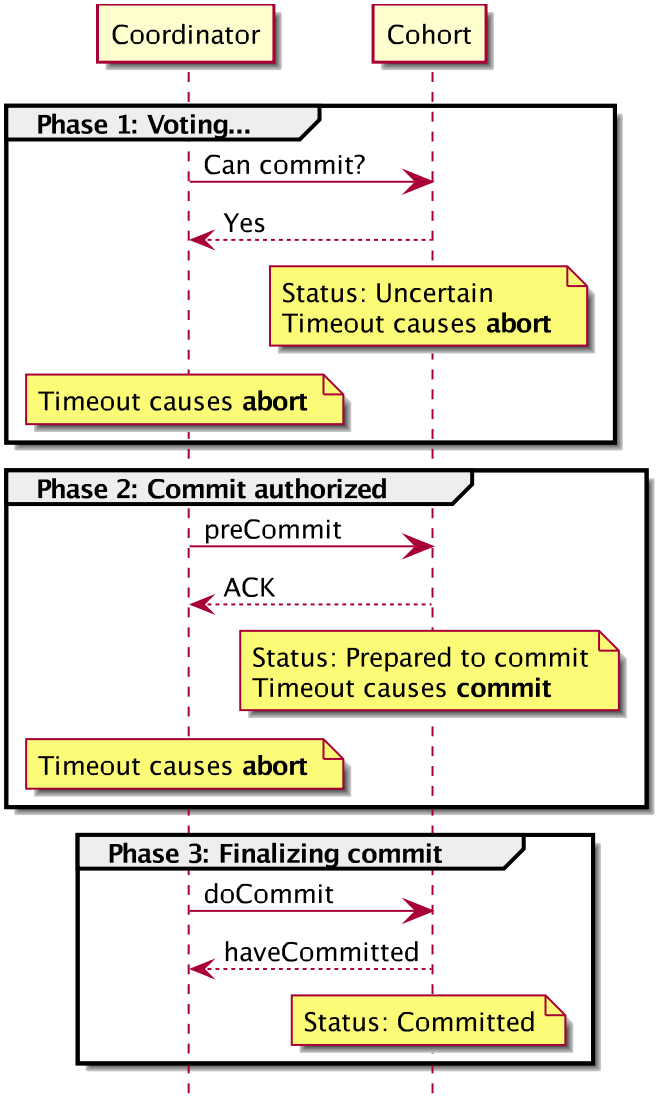
- Реплика выполнила `commit` и упала вместе с координатором
- Система не может восстановить результат транзакции:
  - Только умершие 2 ноды знают точный результат
  - Pessimistic `abort` невозможен — упавшая реплика могла сделать `commit`
  - Pessimistic `commit` невозможен — изначальное решение могло быть `abort`

# Three-Phase Commit Protocol

- Назначение как у 2PC
- **Неблокирующийся** — ограничение сверху на `commit/abort`
- Лучше ведёт себя при сбоях[2]
- 2PC фаза Commit разбивается на 2 фазы — получаем 3PC

[2] `http://the-paper-trail.org/blog/consensus-protocols-three-phase-commit/`

# Диаграмма


Coordinator
Cohort
Phase 1: Voting...
Can commit?
Yes
Status: Uncertain
Timeout causes abort
Timeout causes abort
Phase 2: Commit authorized
preCommit
ACK
Status: Prepared to commit
Timeout causes commit
Timeout causes abort
Phase 3: Finalizing commit
doCommit
haveCommitted
Status: Committed

# Анализ

- Вторая фаза доносит решение до всех участников
- $\Rightarrow$ состояние можно восстановить при сбое реплики
- Phase 3 = 2PC Commit
- При сбое координатора состояние восстанавливается с помощью реплик:
  - Phase 1 (кто-то не получил `Can commit?`) $\Rightarrow$ спокойно делаем `abort`
  - Phase 2 (кто-то получил `preCommit`) $\Rightarrow$ доводим транзакцию до `commit`

# Lamport timestamps

## Цель

Определить порядок событий в распределённой системе[a]

[a] Lamport, L. Time, Clocks, and the Ordering of Events in a Distributed System. 1978

Правила:

- Каждый процесс имеет счётчик
- Инкремент перед каждым событием
- Значение счётчика вместе с каждым сообщением
- При получении сообщения $C_{self} = max(C_{self}, C_{sender})$

# Формализация

- $C(x)$ — время события $x$
- $\forall a \forall b (a \neq b \Rightarrow C(a) \neq C(b))$
- Для различия событий в разных процессах часто добавляют `PID`
- Отношение happens-before: $\rightarrow$
- Clock consistency: $a \rightarrow b \Rightarrow C(a) < C(b)$
- Strong clock consistency: $C(a) < C(b) \Rightarrow a \rightarrow b$ — **только** через Vector Clocks
- Но верно, что $C(a) \geq C(b) \Rightarrow a \not\rightarrow b$

# Анализ

- Невозможно точно синхронизировать время[3]
- Если процессы не общаются, то между их событиями нет отношения порядка
- Обеспечивается лишь **частичный** порядок

[3] Хотя см. `http://research.google.com/archive/spanner.html`

# Vector clocks

## Цель

- Ввести частичный порядок событий в распределённой системе
- Обнаружить нарушения причинно-следственных связей

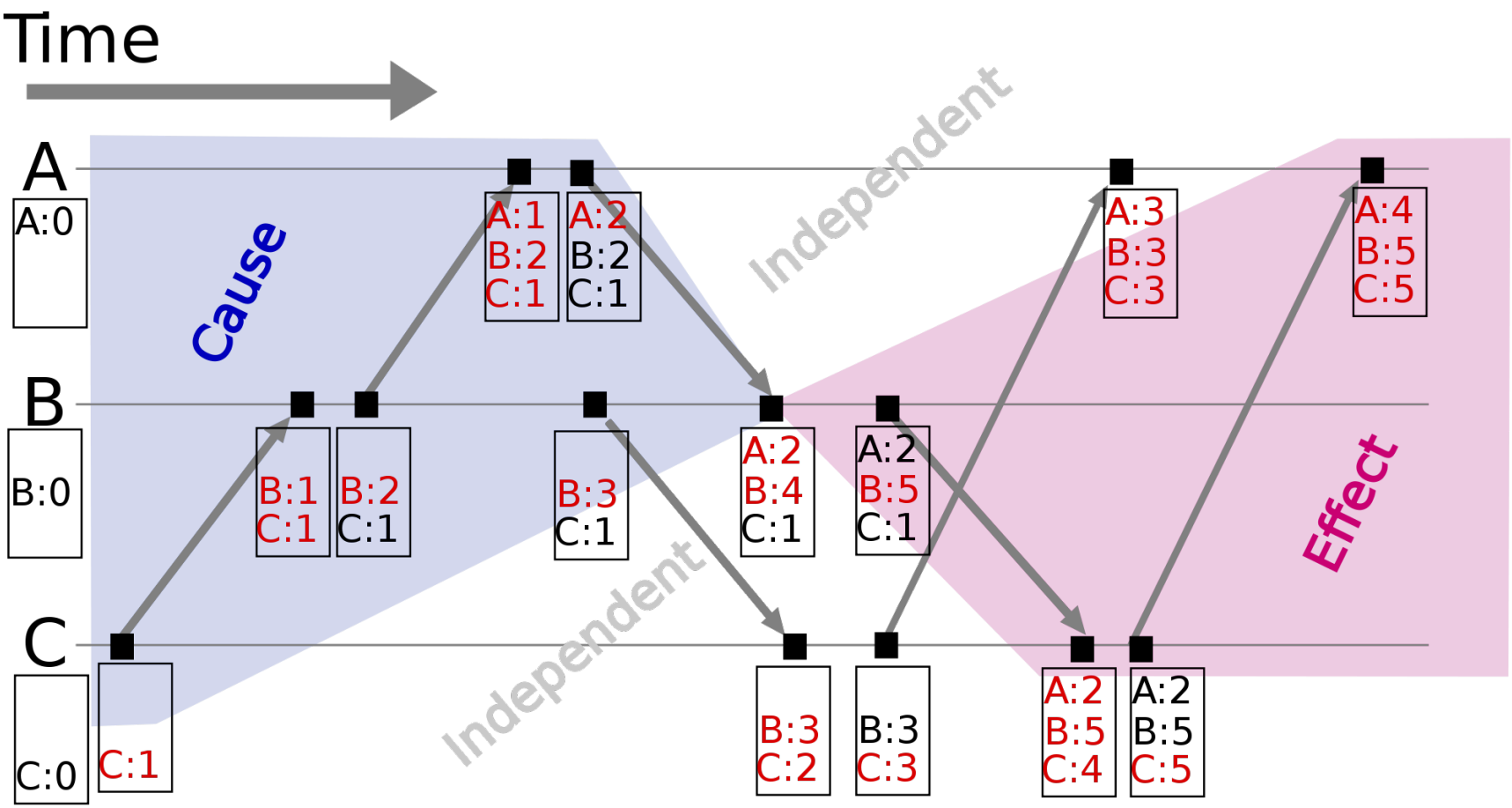
## Определение

Векторные часы в системе с $N$ процессами — это массив $N$ логических часов по одному на процесс

# Правила обновления массива часов

- Изначально все часы выставлены в 0
- При каждом событии процесс инкрементирует **свои** логические часы на 1
- Перед отправкой сообщения процесс:
  - Инкрементирует **свои** логические часы
  - Отправляет весь вектор вместе с сообщением
- При получении сообщения процесс:
  - Инкрементирует **свои** логические часы
  - Обновляет свой вектор, беря покомпонентный максимум с присланным вектором

# Пример


Time
A
B
C
A:0
B:0
C:0
C:1
A:1
B:2
C:1
A:2
B:2
C:1
B:1
C:1
B:2
C:1
B:3
C:1
A:2
B:4
C:1
A:2
B:5
C:1
B:3
C:2
B:3
C:3
A:3
B:3
C:3
A:2
B:5
C:4
A:2
B:5
C:5
A:4
B:5
C:5
Cause
Independent
Independent
Effect

# Формализация

- $VC(x)$ — векторные часы события $x$
- $VC(x)_z$ — компонента векторных часов процесса $z$
- $VC(x) < VC(y) \Leftrightarrow \forall z[VC(x)_z \leq VC(y)_z] \wedge \exists z'[VC(x)_{z'} < VC(y)_{z'}]$
- $x \to y \Leftrightarrow VC(x) < VC(y)$
- $VC(x) < VC(y) \Rightarrow C(x) < C(y)$
- Отношение happens-before антисимметрично и транзитивно

# Мотивация

*There are significant gaps between the description of the Paxos algorithm and the needs of a real-world system... and the final system will be based on an unproven protocol*[4].

## Raft

- Протокол консенсуса для реплицированных автоматов
- Более простой, **понятный** и реализуемый чем Paxos
- Демонстрирует логику рассуждений

[4] Chandra T. D., Griesemer R., Redstone J. *Paxos made live: an engineering perspective*. 2007

# Ссылки

- Replicated State Machines[5]
- Ongaro D., Ousterhout J. *In Search of an Understandable Consensus Algorithm*. 2013.
- Diego Ongaro. *Raft lecture*[6] (Raft user study)
- Diego Ongaro. *Paxos lecture*[7] (Raft user study)
- Множество реализаций[8]

[5] http://en.wikipedia.org/wiki/State_machine_replication

[6] http://www.youtube.com/watch?v=YbZ3zDzDnrw

[7] http://www.youtube.com/watch?v=JEpsBg0AO6o

[8] https://ramcloud.stanford.edu/wiki/display/logcabin/LogCabin

# Replicated State Machines

- Работают на множестве узлов
- Вычисляют идентичные копии одного и того же состояния
- Устойчивы к падению узлов
- Реплицированный лог (последовательность команд)
- Решаемые задачи: выбор лидера, хранилище конфигураций и др.
- Примеры: Chubby (GFS), ZooKeeper in HDFS, ...

# Особенности Raft

- **Strong Leader**
  - Клиенты общаются только с лидером
  - Данные только от лидера к репликам
- **Leader Election**
  - Случайные таймеры
- **Membership changes**
  - Joint consensus
- Формальна описана и доказана безопасность
- Производительность на уровне аналогов

# Алгоритм работы лидера

1. Получить команду от клиента
2. Добавить в локальный лог
3. Доставить команду другим узлам
4. При успехе применить команду к своему автомату
5. Вернуть ответ автомата клиенту

# Свойства протокола

- **Безопасность**: никогда не вернёт некорректный результат
  - non-Byzantine fault tolerance[9]
  - Учитывает ненадёжную сеть
- **Доступность**
  - Пока работают и могут общаться *большинство* узлов
  - Узлы останавливаются и снова подключаются
- **Независимость от абсолютного времени**
- Команда выполняется при ответе от большинства — устойчивость к медленным узлам

[9] http://en.wikipedia.org/wiki/Byzantine_fault_tolerance

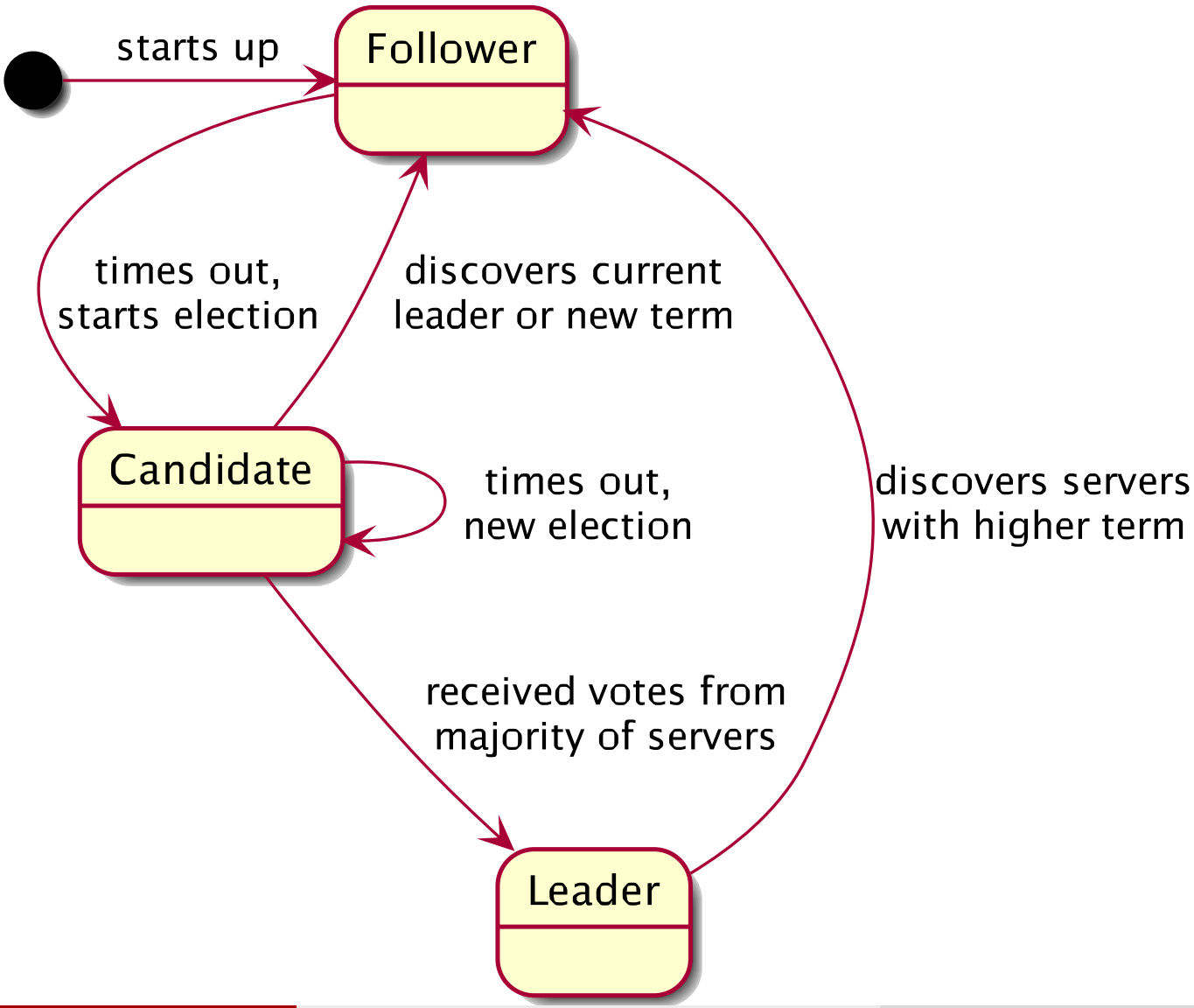
# Подсистемы

- Выбор лидера
- Репликация лога
- Безопасность/корректность

# Кластер

- Несколько узлов (3, 5, ...)
- Узёл в одном из трёх состояний: leader, follower, candidate
- В нормальном режиме: 1 лидер и $n - 1$ последователей
- Последователи пассивны: отвечают на RPC от лидера и кандидатов

# Состояния узла



# Семестры

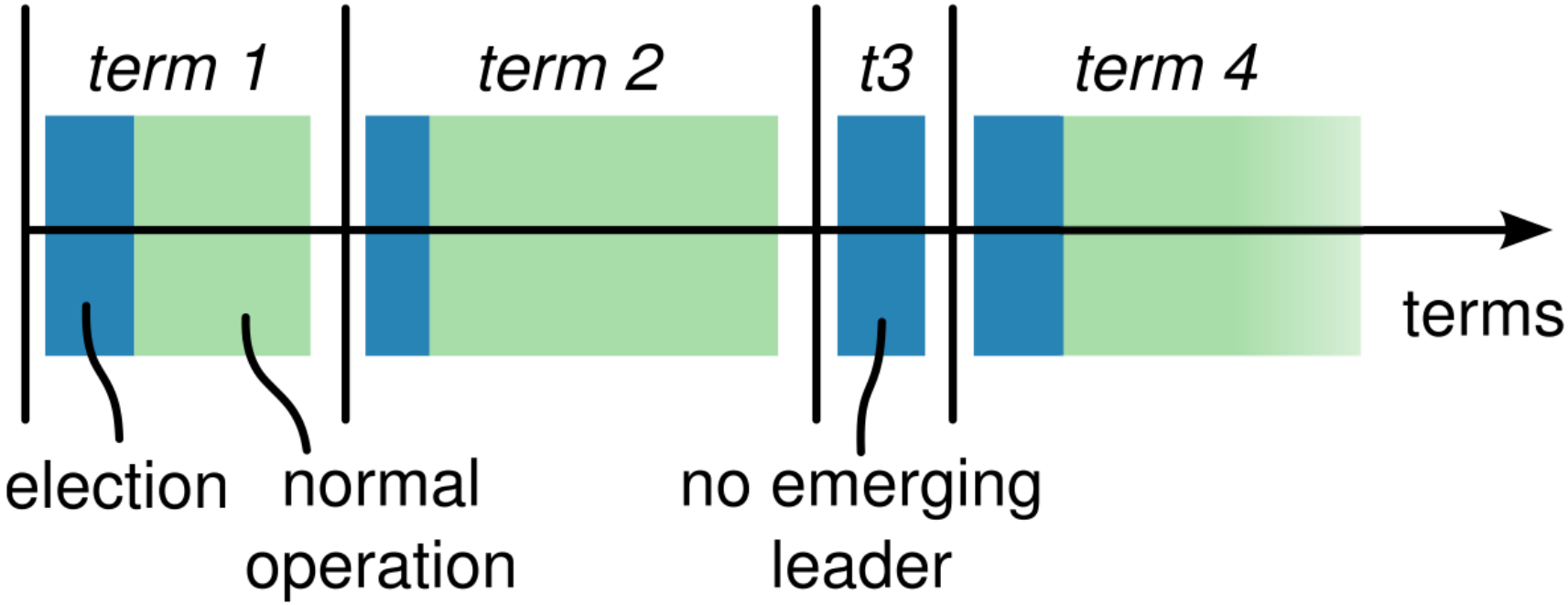
- Время разбивается на семестры (terms) неопределённой длины[10]
- Семестры нумеруются последовательно
- Каждый семестр начинается с выбора лидера
- Если кандидат выигрывает выборы, то он служит лидером до конца семестра
- Если никто не победил на выборах, начинается новый семестр

**Инвариант**

В каждом семестре не больше одного лидера

[10] Аналог логических часов

# Семестры: Пример



# Обнаружение устаревшей информации

- Каждый узел хранит **текущий семестр**
- Текущий семестр каждый раз передаётся в запросах
- Если текущий семестр меньше пришедшего, выбираем пришедший
- Если кандидат или лидер — step down
- Если пришедший семестр меньше текущего, то отвергаем запрос

# RPC

- `RequestVote` — кандидат просит отдать ему голос
- `AppendEntries` — лидер реплицирует команды + heartbeat

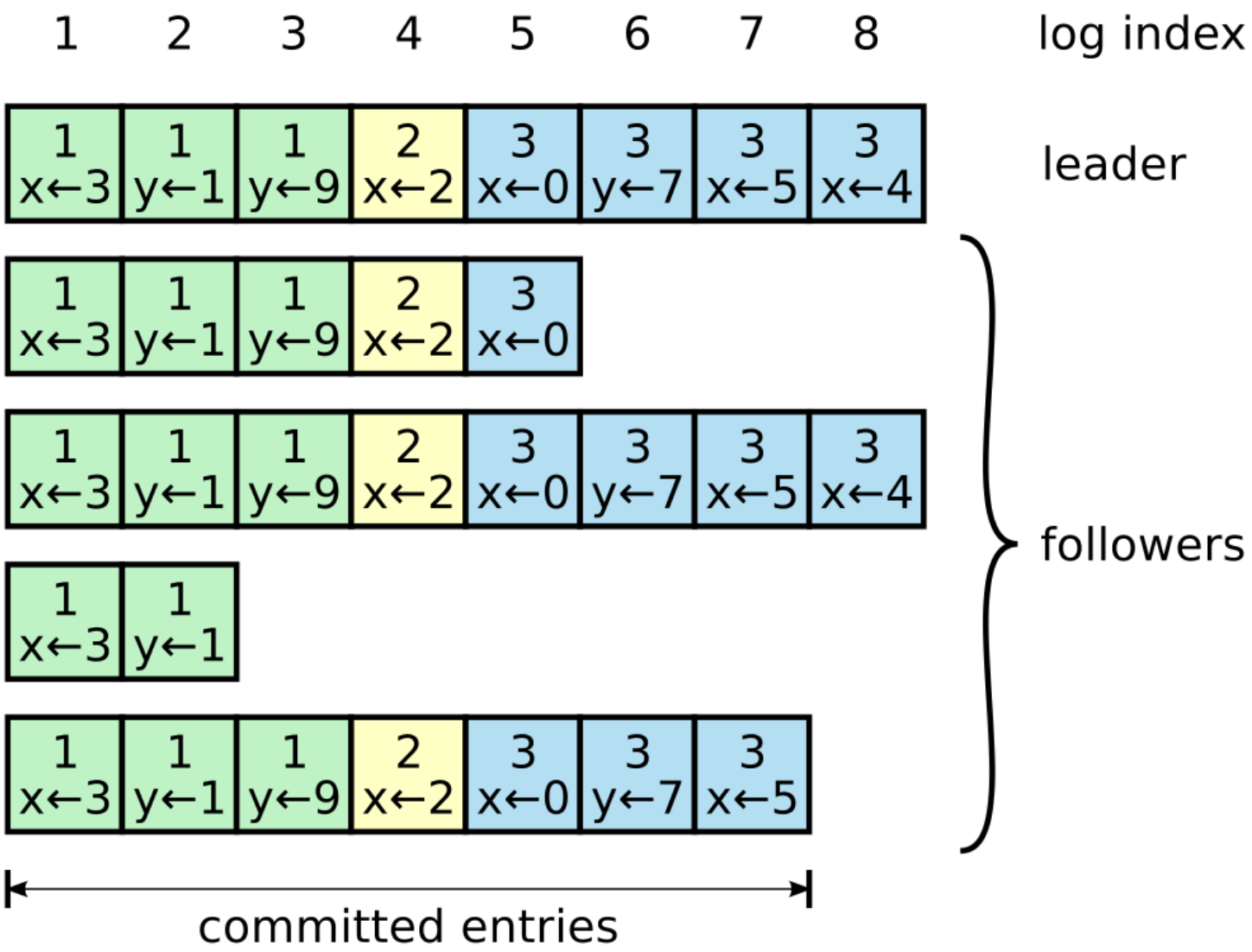
# Условия запуска

- Follower не получает heartbeat в течение `election timeout`
- Follower увеличивает текущий семестр
- Переходит в состояние кандидата
- Параллельно отправляет всем `RequestVote`
- Перезапросы до получения ответа или окончания выборов

# Условия останова

- Follower выиграл выборы (набрал большинство голосов)
  - Каждый узел голосует единожды за семестр (FIFO)
  - Новый лидер начинает рассылать всем heartbeats
- Другой узел стал лидером
  - Кандидат получил `AppendEntries` с семестром $\geq$ текущего
  - Кандидат переходит в состояние follower (step down)
- Наступил таймаут, а победителя всё нет
  - Никто не набрал большинства голосов
  - Кандидат переходит в состояние follower (step down)
  - Random `election timeout`

# Формат лога


1 2 3 4 5 6 7 8
log index
1 x←3 | 1 y←1 | 1 y←9 | 2 x←2 | 3 x←0 | 3 y←7 | 3 x←5 | 3 x←4
leader
1 x←3 | 1 y←1 | 1 y←9 | 2 x←2 | 3 x←0
1 x←3 | 1 y←1 | 1 y←9 | 2 x←2 | 3 x←0 | 3 y←7 | 3 x←5 | 3 x←4
1 x←3 | 1 y←1
1 x←3 | 1 y←1 | 1 y←9 | 2 x←2 | 3 x←0 | 3 y←7 | 3 x←5
followers
committed entries

# Committed Entry

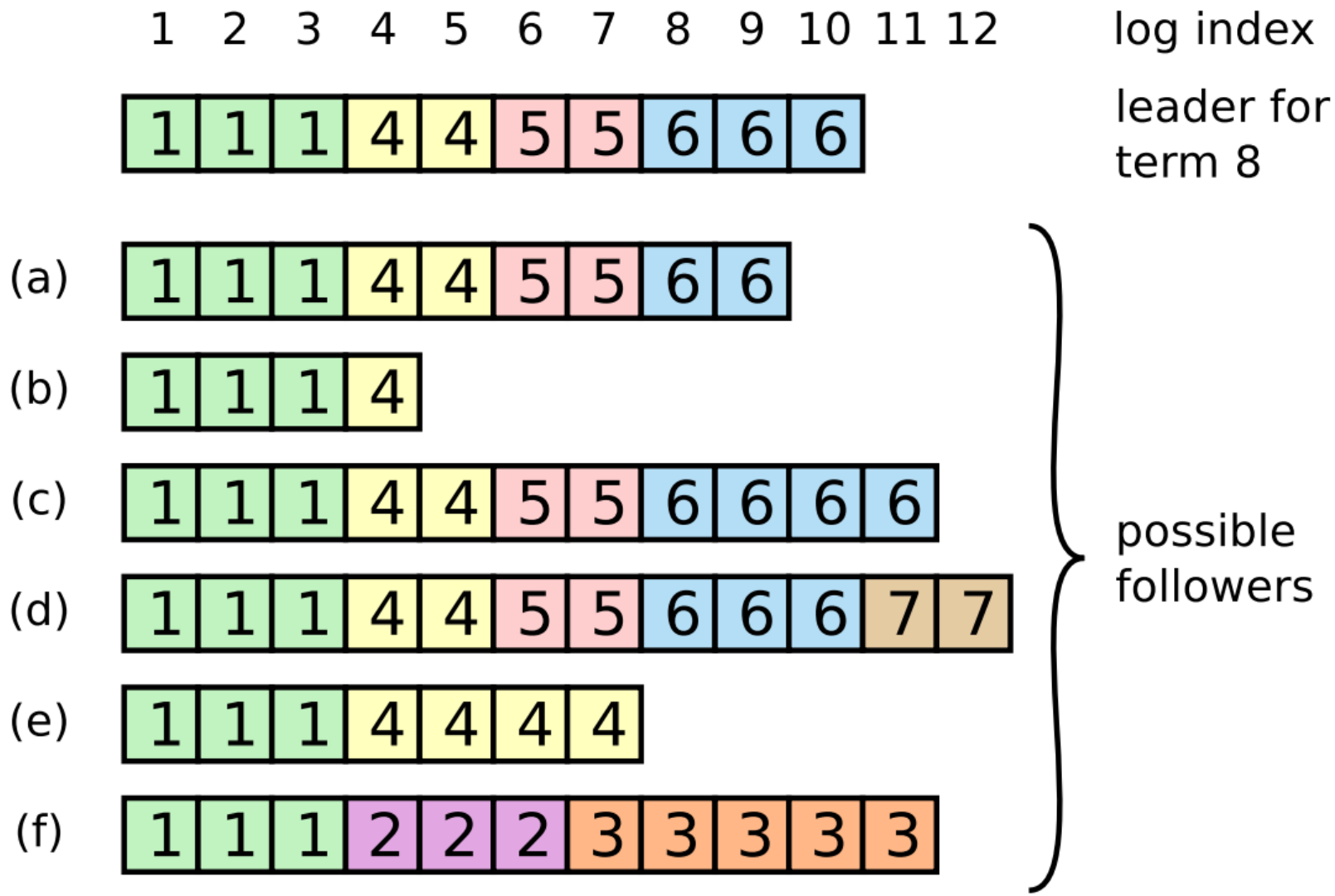
- Гарантируется, что будет выполнена всеми автоматами
- В простейшем случае — если подтверждена запись большинством (см. записи 1-7)
- Лидер пересылает индекс последнего коммита в `AppendEntries` — followers коммитят

# Log Matching Property

Если у двух записей в разных логах совпадают индекс и семестр, то:

- Они содержат одинаковую команду
  - Лидер создаёт не больше одной записи с одним индексом и семестром
  - Записи в логе не перемещаются
- Логи идентичны во всех предшествующих записях
  - Лидер с `AppendEntries` пересылает индекс и семестр предыдущей записи
  - Follower отвергает запрос, если не совпадает
  - Шаг индукции

# Но может быть так



# Пояснения

- a-b — не хватает записей
- c-d — лишние записи
- e-f — оба случая

# Разрешение конфликтов

- Замена конфликтов на followers записями лога лидера
- Лидер помнит `nextIndex` для каждого follower — изначально следующий за последним в логе
- Используется RPC `AppendEntries` Consistency Check
  - Если ошибка, то `nextIndex - 1` и retry
  - Если совпадение, то удаляется хвост и добавляются записи лога лидера
- Автоматическая сходимость логов
- Лидер никогда не удаляет и не перезаписывает записи в собственном логе

# Проблема

- Лидер закоммитил несколько записей
- Последователь был недоступен
- Лидер ушёл с радаров
- Последователь стал новым лидером
- Последователь перезаписал всем логи
- В результате разные автоматы выполнили разный набор команд

# Решение

Расширим протокол:

- Ограничение на узлы, которые могут стать лидером
- Ограничение на записи, которые считаются закоммиченными

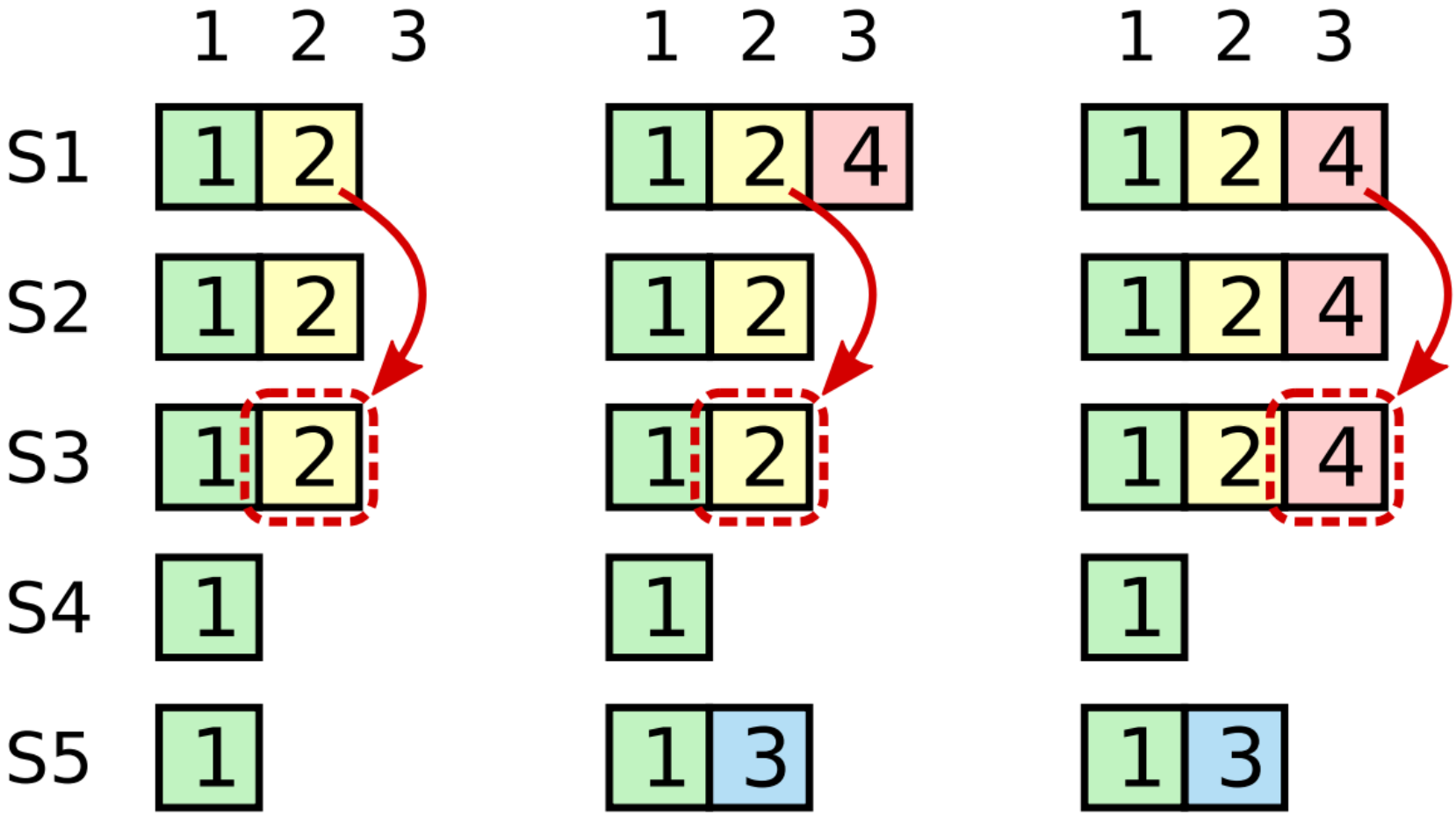
Обеспечиваем:

- Лидер любого семестра содержит все записи, закоммиченые в предыдущих семестрах
- Записи направлены только от лидера к последователям
- Лидеры никогда не перезатирают записи в своём логе

# Ограничение на выбор лидера

- Всё так же нужно собрать голоса большинства
- Но можно стать лидером, только если наш лог не менее свежий, чем у каждого голосующего
- `RequestVote` RPC содержит информацию о последней записи в логе
- Лог новее, если семестр старше или индекс больше (лог длиннее)

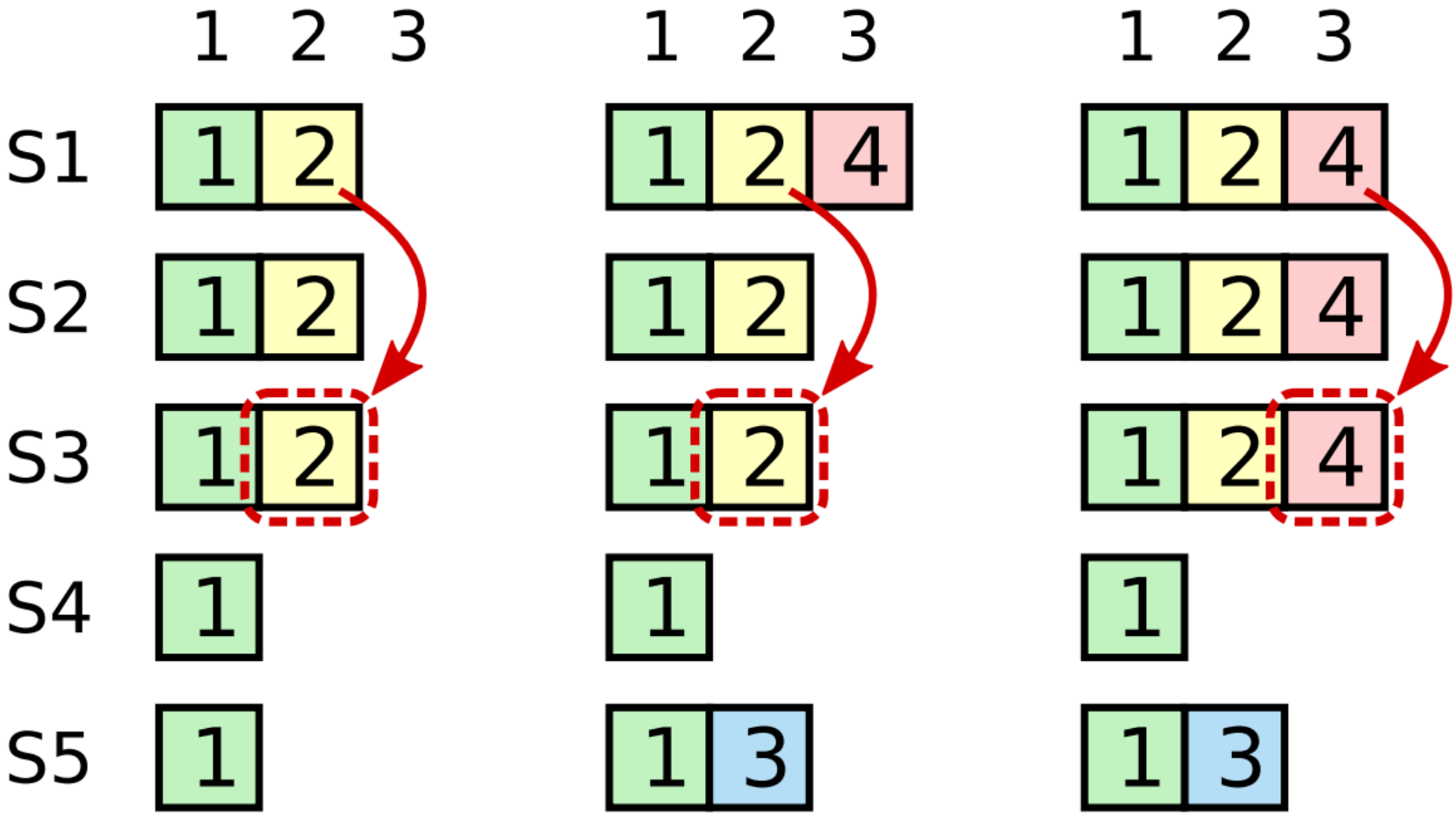
# Коммиты


1 2 3
S1
S2
S3
S4
S5
1 2
1 2
1 2
1
1
1 2 3
S1
S2
S3
S4
S5
1 2 4
1 2
1 2
1
1 3
1 2 3
S1
S2
S3
S4
S5
1 2 4
1 2 4
1 2 4
1
1 3

# Случай 1

- Наиболее популярный
- Лидер реплицирует запись из текущего семестра
- Запись закоммичена, как только подтвердит большинство
- Лидером могут стать только те, у кого есть эта запись

# Коммиты



# Случай 2

- Лидер коммитит запись из предыдущего семестра
- Лидер семестра 2 создал запись по индексу 2, отреплицировал на S1 и S2 и упал
- S5 стал лидером в семестре 3 (собрал голоса у S3 и S4)
- Записал запись в свой лог по индексу 2 и упал
- S1 стал лидером в семестре 4 (голоса от S2 и S3)
- Отреплицировал запись по индексу 2 на S3
- Незакоммичена несмотря на большинство
- S5 может стать лидером (его лог новее чем S2, S3 и S4) и распространить **своё** значение по индексу 2

# Ограничение на коммиты

- Пока новый лидер не закоммитит запись из **текущего** семестра, он считает предыдущие записи незакоммиченными
- См. случай 3 — после этого S5 не может стать лидером
- См. доказательство корректности[11]

[11] Safety proof and formal specification for Raft: `http://raftuserstudy.s3-website-us-west-1.amazonaws.com/proof.pdf`

# Timing and availability

## Timing requirement

$$broadcastTime \ll electionTimeout \ll MTBF$$

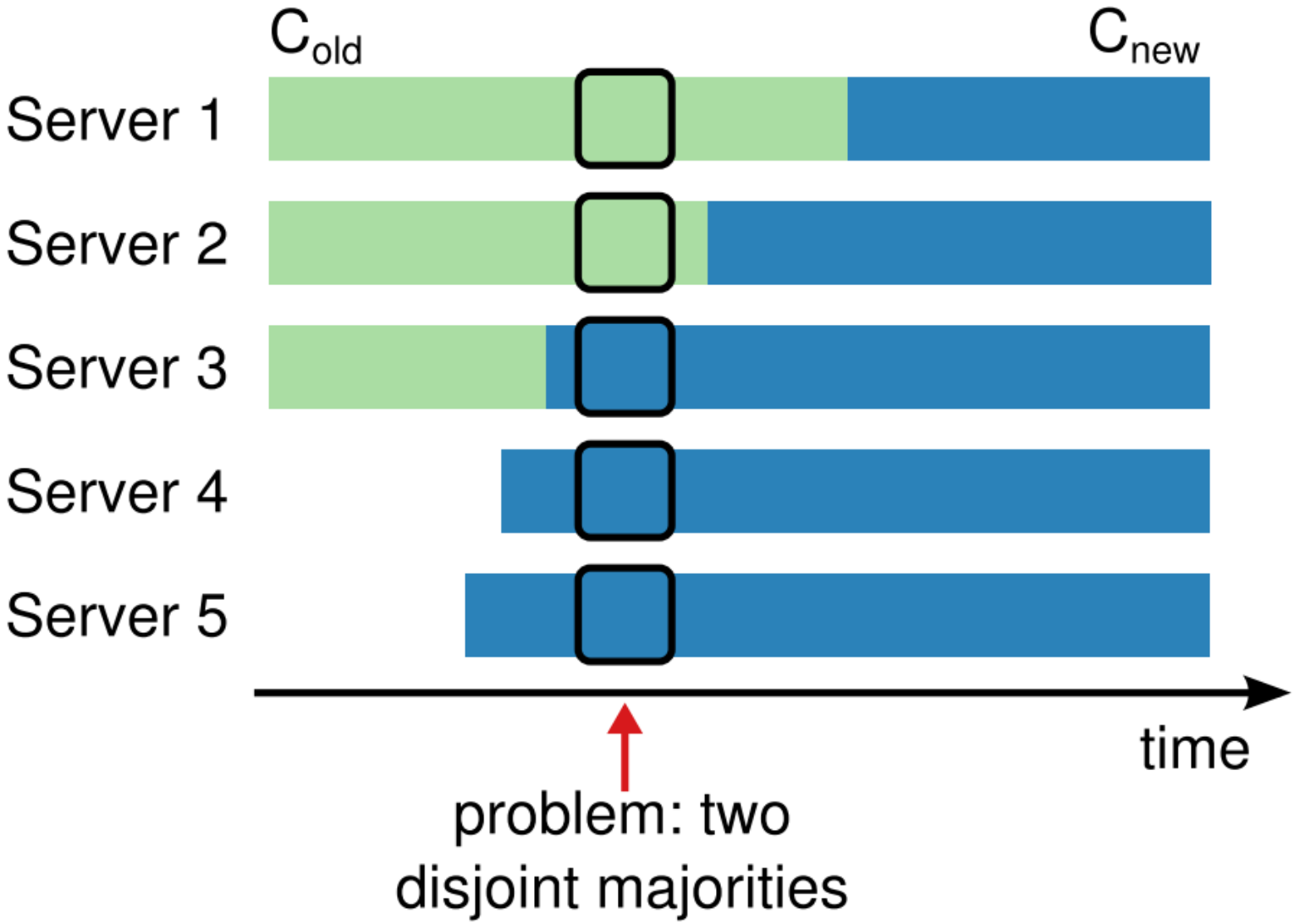
Типичные значения:

- *broadcastTime*: 0.5-20 ms
- *electionTimeout*: 10-500 ms
- *MTBF*: $\ll$ month

# Изменение конфигурации кластера

- Предполагали, что состав узлов статичен
- Но иногда нужно заменять сервера или изменять уровень репликации
- В идеале — без downtime
- И автоматически — исключить человеческий фактор

# Ситуация


$C_{old}$
$C_{new}$
Server 1
Server 2
Server 3
Server 4
Server 5
time
problem: two
disjoint majorities

# Проблема

## Проблема

Возможно одновременное существование двух лидеров для одного семестра в двух подкластерах

Решения:

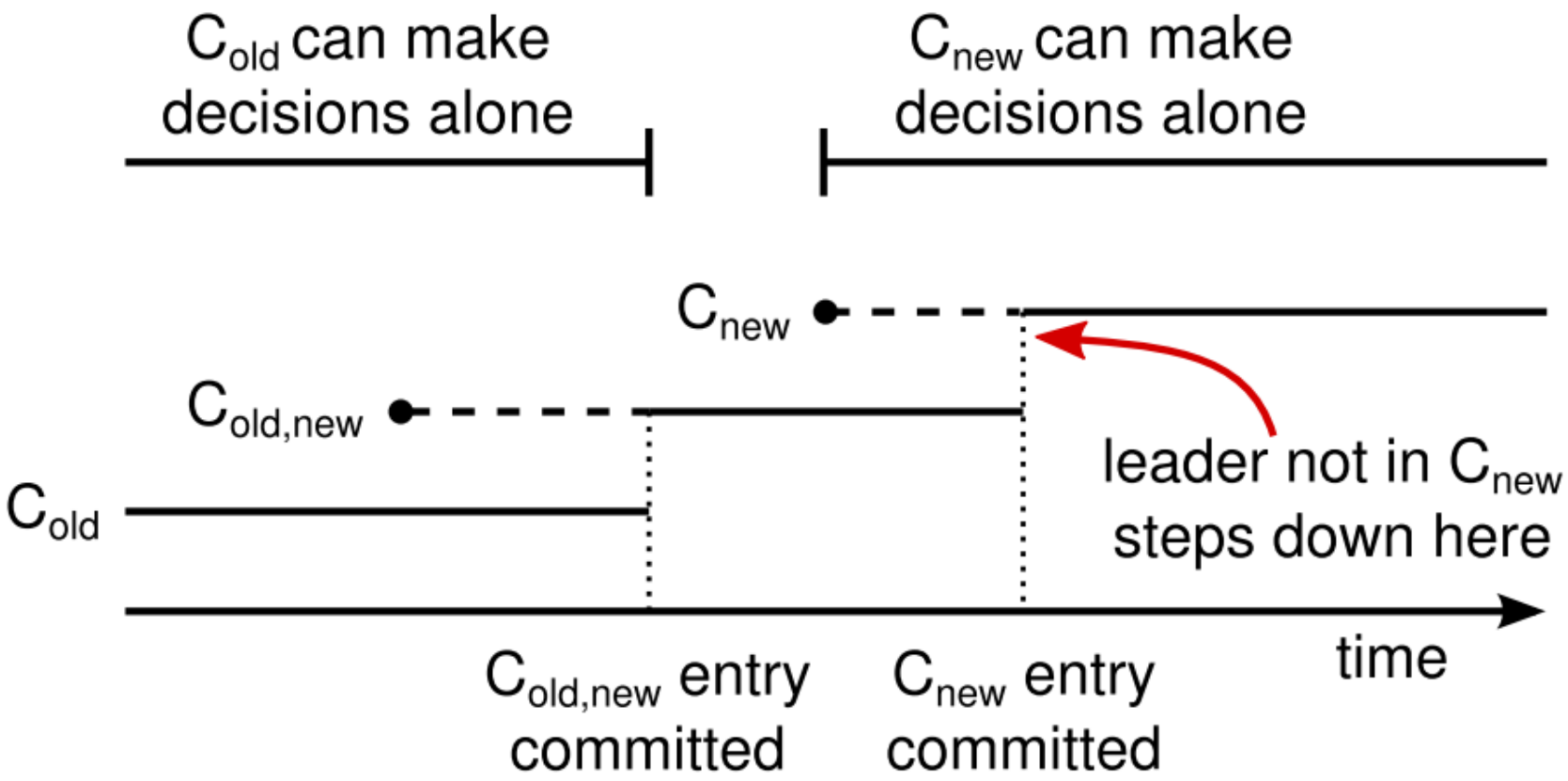
- 2PC: сначала выключаем старую конфигурацию, возможен downtime
- Raft: промежуточная конфигурация (joint consensus)

# Идея

Комбинация двух конфигураций:

- Записи реплицируются серверам в обеих конфигурациях
- Сервер из любой конфигурации может стать лидером
- Нужно собрать большинство в **каждой** конфигурации по-отдельности
- При этом без downtime

# Joint Consensus



# Алгоритм

- Запрос лидеру на смену конфигурации с $C_{old}$ на $C_{new}$
- Сохраняет в лог $C_{old,new}$ и реплицирует
- Каждый сервер сохраняет $C_{old,new}$ в лог и сразу начинает использовать
- Лидер упал — новый лидер из $C_{old,new}$ или $C_{old}$, но не $C_{new}$
- Успешно закоммитили — лидером может стать только $C_{old,new}$
- Лидер сохраняет в лог $C_{new}$ и реплицирует и т. д.
- $C_{old}$ и $C_{new}$ **не могут одновременно** принимать решения

# Особенности

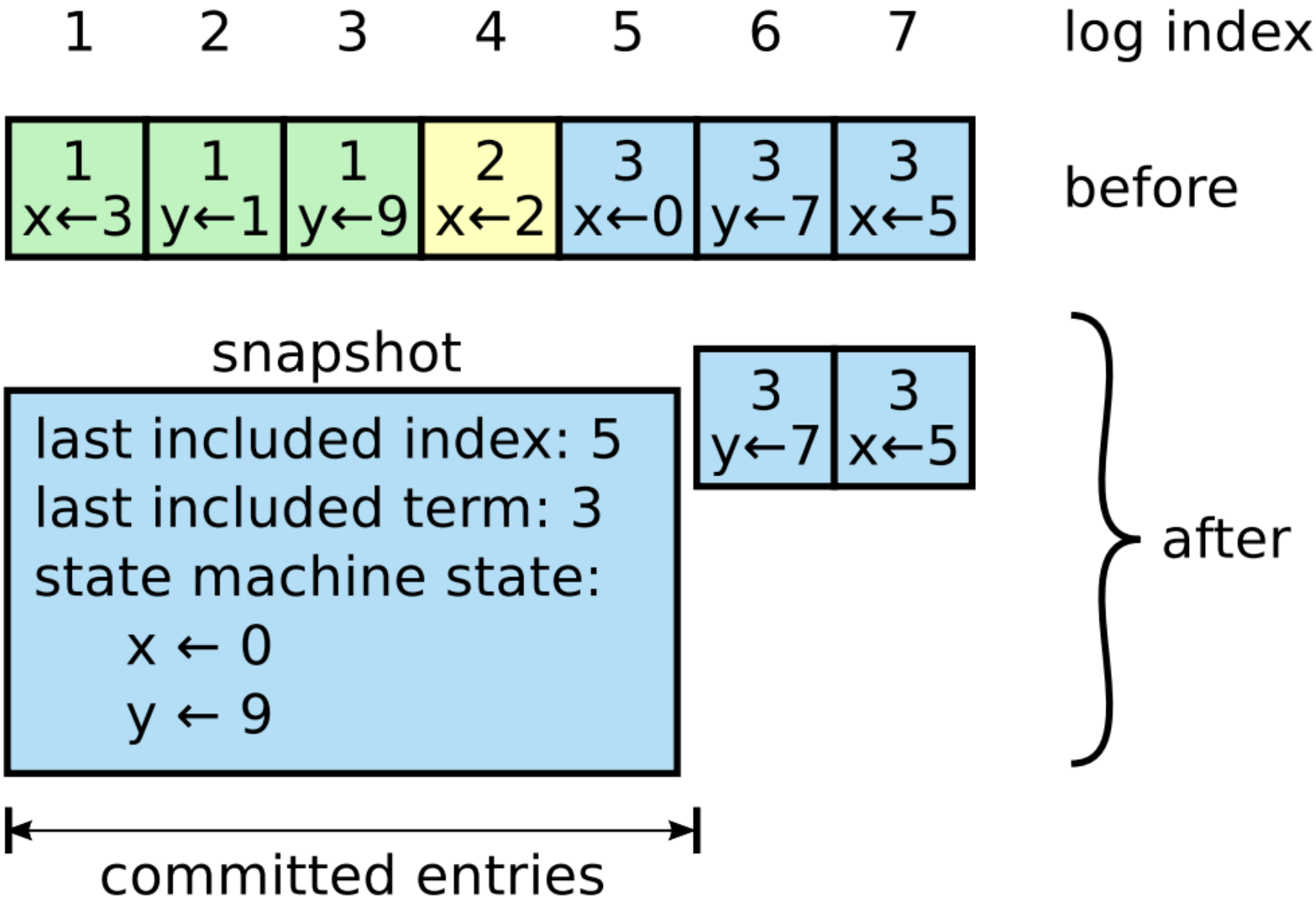
- Если лидер входит в $C_{old}$, но не в $C_{new}$, то должен выйти из кластера (реплицирует, но не входит в большинство)
- Новые сервера могут быть пустыми — вначале добавляем как неголосующих, но реплицируем на них логи
- Удаление серверов, которые не в курсе, что их удаляют, может снизить производительность кластера — инициируют безнадёжные выборы

# Log Compaction

Подходы:

- Очистка логов — перемещаем «живые» записи в голову и очищаем мёртвый хвост
  - Инкрементально и эффективно
  - Определение живых записей может быть сложным
- Слепки — состояние системы периодически сохраняется на stable storage, а лог сбрасывается
  - Менее эффективно (неинкрементально)
  - Просто

# Snapshotting



# Snapshotting: Нюансы

- Нужно решить, когда создавать слепки — например, при достижении логом определённого размера
- Copy-on-write для асинхронной записи слепков + functional data structures/`fork()`

# Проблема

## Команда может применяться несколько раз

- Лидер получил команду, закоммитил и умер, не успев ответить
- Клиент делает перезапрос

## Идемпотентность

Клиент присваивает командам уникальные последовательные номера

# Проблема

## Протухшее чтение

- У лидера есть все закоммиченые изменения
- Но в начале семестра он не знает, кто из них закоммичен
- Вначале он должен закоммитить запись из нового семестра

## Решение

- `no-op` в начале семестра
- Heartbeat с большинством кластера перед ответом на read

# Альтернативное домашнее задание

Проект `akka-raft`:

- Open-source Raft implementation
- Scala
- Akka Extension
- Parameterized by Akka FSM
- Extended with API (`spray`-based)
- Use Case: `etcd`[12] implementation

[12] `https://github.com/coreos/etcd`

# Библиотечка

- Think Distributed: A Distributed Systems Podcast[13]
- Наборы открытых данных для курсовой работы[14]

[13] http://thinkdistributed.io/

[14] Via @pulser: http://www.hackathon.spb.ru/#!datasets/c1miv

# Homework: Feature request 1

- Key-Value API (кто ещё не)
- Данные не влезают в память (>= 4 ГБ)
- Используйте открытые источники данных
- Стресстест в виде shell-скрипта
  - Скачали
  - Распаковали
  - Запустили Storage (`Xmx1G`)
  - Залили в Storage
- Срок реализации до 2013-10-14
- Hints: `mmap()`, дисковый кэш, ...

# Вопросы?

- http://incubos.org/contacts/
- Общие вопросы — в Twitter: @incubos
- Вопросы по лекциям — в комментариях: http://incubos.org/blog/
- Частные вопросы — в почту vadim.tsesko@gmail.com