CAP. Распределённый commit.

«Использование баз данных»

Вадим Цесько
https://incubos.org
@incubos

ТЕХНОПОЛИС

25 апреля 2018 г.

Содержание

1. Remembrance Inc.
2. CAP theorem
3. Транзакции
4. Distributed Commit
5. Happens-before
6. Raft

Идея сервиса

Never forget, even without remembering![a]

[a] http://ksat.me/a-plain-english-introduction-to-cap-theorem/

Ever felt bad that you forget so much? Don't worry. Help is just a phone away!

When you need to remember something, just call 555-55-REMEM and tell us what you need to remember. For eg., call us and let us know of your boss's phone number and forget to remember it. When you need to know it back, call back the same number 555-55-REMEM and we'll tell you what's your boss's phone number.

Charges: only $0.1 per request

Типичный диалог

- *Customer*: Hey, can you store my neighbor's birthday?
- *You*: Sure... When is it?
- *Customer*: 2nd of Jan
- *You*: (write it down against the customer's page in your paper note book) Stored. Call us any time for knowing your neighbor's birthday again!
- *Customer*: Thank you!
- *You*: No problem! We charged your credit card with $0.1

Растёт нагрузка

- Нужен только блокнот и телефон
- Отлично работает
- Сарафанное радио
- Получили финансирование от YCombinator
- Сотни звонков в день

Проблемы подхода

- Клиенты висят в очереди на телефоне и злятся
- Вы заболели и пропустили рабочий день и день оповещений

Масштабируемся

- Берём жену в долю
- У каждого добавочный номер
- Клиенты используют всё тот же 555–55-REMEM
- АТС балансирует клиентов между сотрудниками

Проблема

- *Customer*: Hey!
- *You*: Glad you called “Remembrance Inc!”. What can I do for you?
- *Customer*: Can you tell me when is my flight to New Delhi?
- *You*: Sure... 1 sec, sir. (You look up your notebook. WOW! There is no entry for "flight date"in customer’s page!!!)
- *You*: Sir, I think there is a mistake. You never told us about your flight to Delhi.
- *Customer*: What?! I just called you guys yesterday! (Cuts the call!)

Чиним неконсистентность

- Перед тем как записать что-либо, говорим партнёру
- Таким образом, обновления хранятся у нас обоих
- Когда клиент спрашивает, то вся информация под рукой

Проблемы подхода

- Дольше операции обновления, но операций чтения большинство
- Если кто-то из нас заболеет — проблема **недоступности**

Consistent & Available

Новая версия алгоритма:

- Перед записью говорим партнёру
- Партнёр недоступен — отправляем ему email
- В начале рабочего дня разбираем почту и обновляем блокнот

Partition Tolerance

- Жена обиделась, всё-таки пришла на работу, но решила не сообщать об обновлениях
- Можно реализовать partition tolerance, если решить не принимать запросы, пока не восстановится связность с женой
- Но тогда жертвуем availability

CAP theorem

E. Brewer. *Towards Robust Distributed Systems*, 2000.

Суть

Выберите 2 из 3:

- Consistency
- Availability
- Partition Tolerance

Пояснения

На самом деле[1]:

- Невозможность **полного** обеспечения **C**, **A** и **P**
- **P** in a distributed system is a **MUST HAVE**
- Партиции относительно редки — необязательно жертвовать **C**/**A** при их отсутствии
- Выбор различного соотношения **C**/**A** для разных подсистем, запросов, данных, пользователей и т. д.
- Свойства **непрерывны**, а не бинарны

[1] http://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed

Latency

CAP проявляется во время **таймаута**:

- Отменить операцию = снизить доступность
- Подтвердить операцию = снизить консистентность
- Перезапрос = отложить решение
- Бесконечный перезапрос = выбираем консистентность

Обработка партиций

- Обнаружение партиционирования
  - Не все узлы могут признать партиционирование
  - Можно подбирать таймауты ⇒ false positives
- Ограничение набора возможных операций/снижение консистентности
  - Можно запомнить и «отложить» операцию
  - Yahoo PNUTS: локальный мастер per user + асинхронная репликация ⇒ меньше задержки
  - Facebook[2]: мастер всегда один, запись в удалённый мастер и чтение 20 сек из мастера, затем из локальной реплики
- Восстановление консистентности и устранение ошибок

[2] http://www.facebook.com/note.php?note_id=23844338919&id=9445547199

Картинка

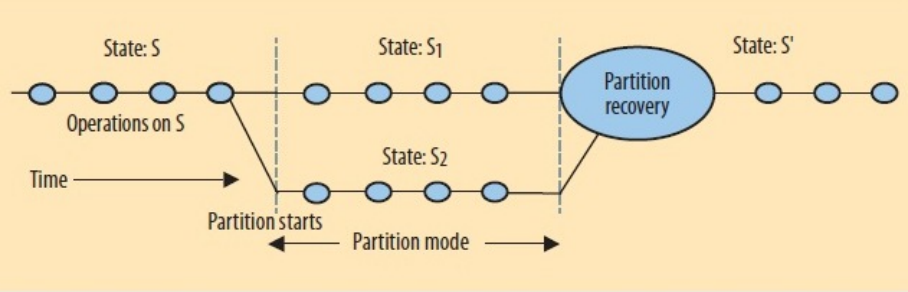


Figure 1. The state starts out consistent and remains so until a partition starts. To stay available, both sides enter partition mode and continue to execute operations, creating concurrent states S_1 and S_2, which are inconsistent. When the partition ends, the truth becomes clear and partition recovery starts. During recovery, the system merges S_1 and S_2 into a consistent state S' and also compensates for any mistakes made during the partition.

Трюки

- Вся система недоступна — есть, например, HTML5 on-client persistent storage
- Различные области консистентности при партициях (primary partition, quorum)
- Шардирование
  - Нет глобальных инвариантов
  - Но возможен прогресс даже при партициях

Восстановление консистентности

- Нужно знать инварианты (не всегда очевидно)
- Почти то же, что параллельные обновления в многопоточном программировании
- Векторные часы для обнаружения конфликтов
- Откат ошибок (желательно иметь лог операций — см. DVCS)
- Commutative Replicated Data Types (CRDTs)[3][4]

[3] M. Shapiro et al. Conflict-Free Replicated Data Types. 2011

[4] M. Shapiro et al. Convergent and Commutative Replicated Data Types. 2011

Пример: ATM

- Инвариант: баланс больше 0
- Выбираем availability
- Связь пропала — лимит на снятие наличных
- Операция коммутативна
- Связь восстановилась — считаем баланс, возможно, штраф за overdraft
- См. Check kiting[5]

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

Use Case: Ebay

Principles for Scaling[6]:

- Partition Everything
  - Vertical, horizontal, no session state
- Asynchrony Everywhere
  - Events, multicast
- Automate Everything
  - Adaptive configuration, machine learning
- Remember Everything Fails
  - Failure detection, rollback, graceful degradation
- Embrace Inconsistency
  - Eventual consistency, no distributed transactions

[6] Randy Shoup at LADIS 2008, `http://www.cs.cornell.edu/projects/ladis2008/materials/eBayScalingOdyssey%20ShoupTravostino.pdf`

Определение

Транзакция

A unit of work performed against a database, and treated in a coherent and reliable way independent of other transactions[a]

[a] http://en.wikipedia.org/wiki/Database_transaction

Две философии:

- ACID[7] — focus on consistency
- BASE — focus on high availability

[7] Name was coined (no surprise) in California in 60's

ACID-транзакции

- **Atomicity** — «либо всё, либо ничего»
  - `commit/rollback`, undo-log, атомарные операции, ...
- **Consistency** — переход из одного «корректного» состояния в другое
  - Индексы, ссылки, ограничения, триггеры, ...
- **Isolation** — иллюзия последовательного выполнения транзакций
  - Уровни изоляции: Read uncommitted / dirty reads, Read committed / non-repeatable read, Repeatable reads / phantom reads, Serializable, MVCC[8], ...
- **Durability** — надёжное долговременное хранение

[8] `http://en.wikipedia.org/wiki/Multiversion_concurrency_control`

Распределённый commit

- Two-phase commit protocol[9]
- Three-phase commit protocol[10]
- Paxos[11]
- Raft[12]

[9] http://en.wikipedia.org/wiki/2PC
[10] http://en.wikipedia.org/wiki/3PC
[11] http://en.wikipedia.org/wiki/Paxos_(computer_science)
[12] https://raft.github.io

ACID in CAP

- **Atomicity**
  - Используется во всех партициях
  - Упрощает восстановление
- **Consistency**
  - Невозможна при партиционировании
  - Запрет некоторых операций
  - Восстановление инвариантов впоследствии
- **Isolation**
  - Можно работать только с одной партицией
  - Либо более слабая корректность, компенсируемая восстановлением
- **Durability**
  - Durable history позволяет исправлять ошибки при восстановлении
  - Иногда ослабляют из-за дороговизны

BASE

Dan Pritchett. BASE: An Acid Alternative[13]. 2008:

- **Basically Available**
- **Soft State**
- **Eventually consistent**

[13] http://queue.acm.org/detail.cfm?id=1394128

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

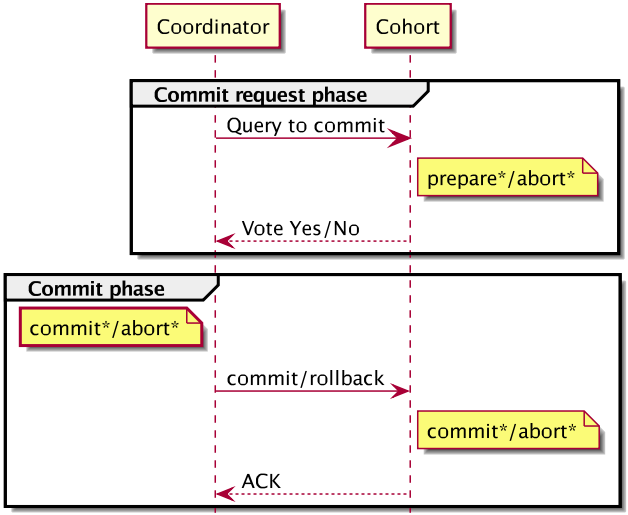
- Выделенный координатор
- Stable storage[14] + write-ahead log (WAL)
- Узлы не умирают навсегда
- Данные в WAL не теряются и не портятся
- Любые два узлы могут общаться

Внимание

Если полностью уничтожить узел, можно потерять данные

[14] 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**: отправляем решение оставшемуся молчуну, 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:

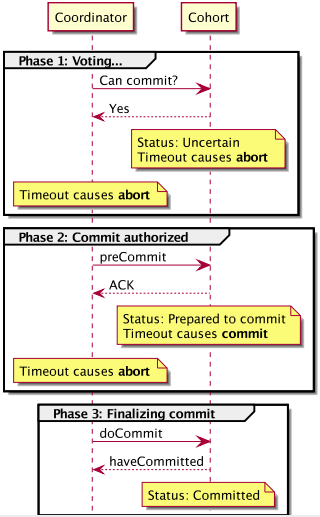
- Реплика выполнила `commit` и упала вместе с координатором
- Система не может восстановить результат транзакции:
  - Только умершие 2 ноды знают точный результат
  - Pessimistic `abort` невозможен — упавшая реплика могла сделать `commit`
  - Pessimistic `commit` невозможен — изначальное решение могло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

Three-Phase Commit Protocol

- Назначение как у 2PC
- **Неблокирующийся** — ограничение по времени на `commit/abort`
- Лучше ведёт себя при сбоях[15]
- 2PC фаза Commit разбивается на 2 фазы — получаем 3PC

[15] `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`

Fail-stop

Неустойчив к network partitioning.

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[16]
- Если процессы не общаются, то между их событиями нет отношения порядка
- Обеспечивается лишь **частичный** порядок

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

Vector clocks

Цель

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

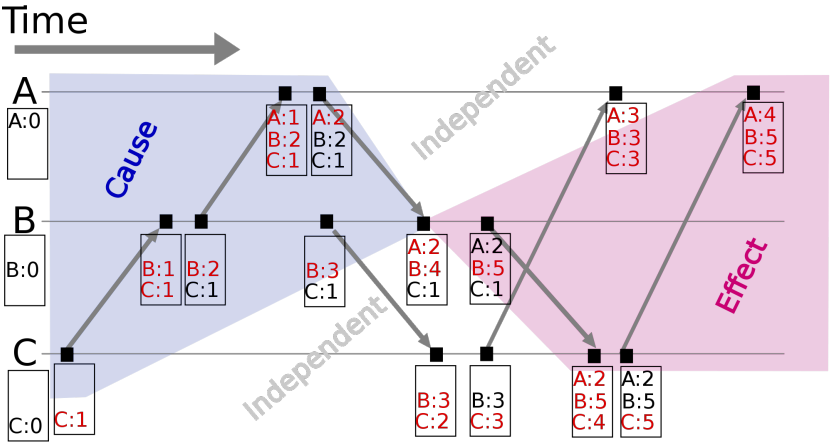
Определение

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
B:1
C:1
B:2
C:1
A:1
B:2
C:1
A:2
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
- $VC(x) < VC(y) \Leftrightarrow \forall z[VC(x)_z \leq VC(y)_z] \wedge \exists z'[VC(x)_{z'} < VC(y)_{z'}]$
- $x \rightarrow y \Leftrightarrow VC(x) < VC(y)$
- $VC(x) < VC(y) \Rightarrow C(x) < C(y)$
- 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[17].

Raft

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

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

Ссылки

- Replicated State Machines[18]
- Ongaro D., Ousterhout J. *In Search of an Understandable Consensus Algorithm*. 2013.
- Diego Ongaro. *Raft lecture*[19] (Raft user study)
- Diego Ongaro. *Paxos lecture*[20] (Raft user study)
- Множество реализаций[21]

[18] http://en.wikipedia.org/wiki/State_machine_replication
[19] http://www.youtube.com/watch?v=YbZ3zDzDnrw
[20] http://www.youtube.com/watch?v=JEpsBg0AO6o
[21] https://raft.github.io/#implementations

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, ZooKeeper, etcd, Consul, ...

Особенности Raft

- **Strong Leader**
  - Клиенты общаются только с лидером
  - Данные только от лидера к репликам
- **Leader Election**
  - Случайные таймеры
- **Membership changes**
  - Joint consensus
- Формально 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[22] fault tolerance[23]
  - 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

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

[23] https://c3.nasa.gov/dashlink/resources/624/

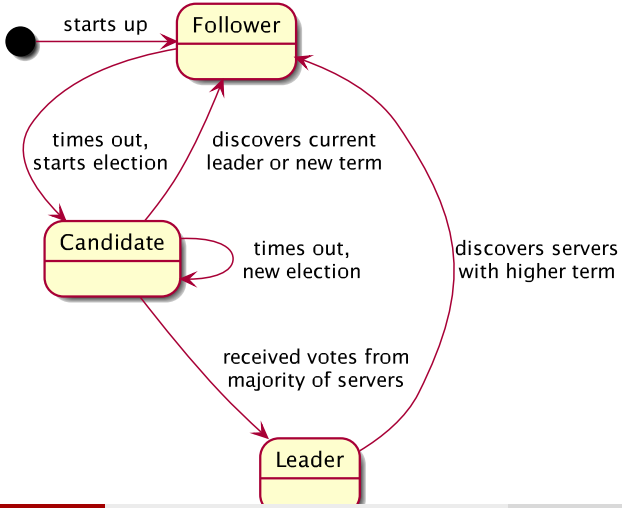
Подсистемы

- 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

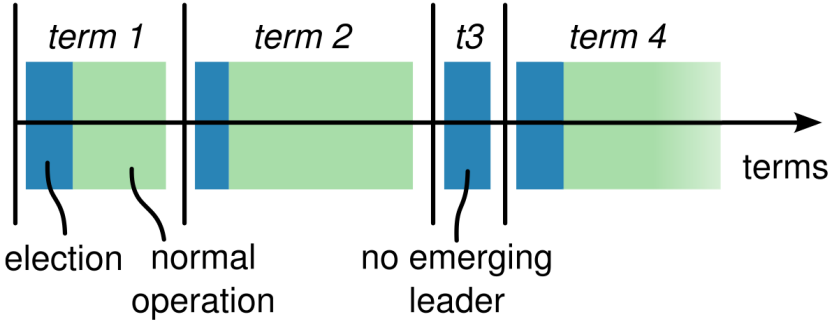
- Время разбивается на семестры (terms) неопределённой длины[24]
- Семестры нумеруются последовательно
- Каждый семестр начинается с выбора лидера
- Если кандидат выигрывает выборы, то он служит лидером до конца семестра
- Если никто не победил на выборах, начинается новый семестр

Инвариант

В каждом семестре не больше одного лидера

[24] 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

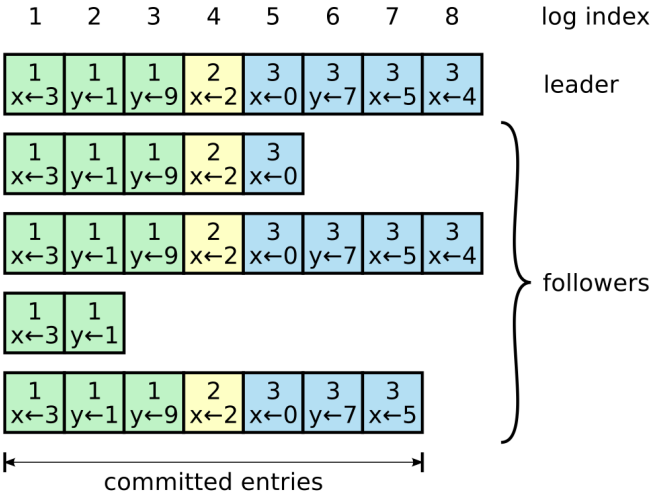
Условия запуска

- 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

	1	2	3	4	5	6	7	8	9	10	11	12	log index
	1	1	1	4	4	5	5	6	6	6			leader for term 8
(a)	1	1	1	4	4	5	5	6	6				possible followers
(b)	1	1	1	4									possible followers
(c)	1	1	1	4	4	5	5	6	6	6	6		possible followers
(d)	1	1	1	4	4	5	5	6	6	6	7	7	possible followers
(e)	1	1	1	4	4	4	4						possible followers
(f)	1	1	1	2	2	2	3	3	3	3	3		possible followers

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 закоммичеными

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

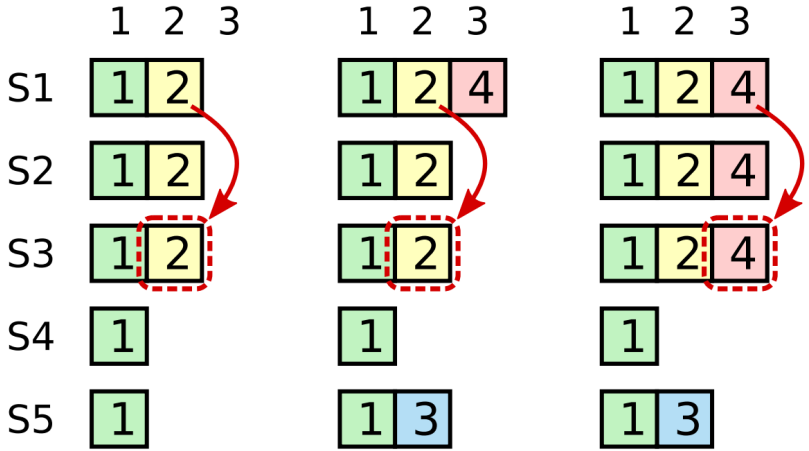
Ограничение на выбор лидера

- 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)

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

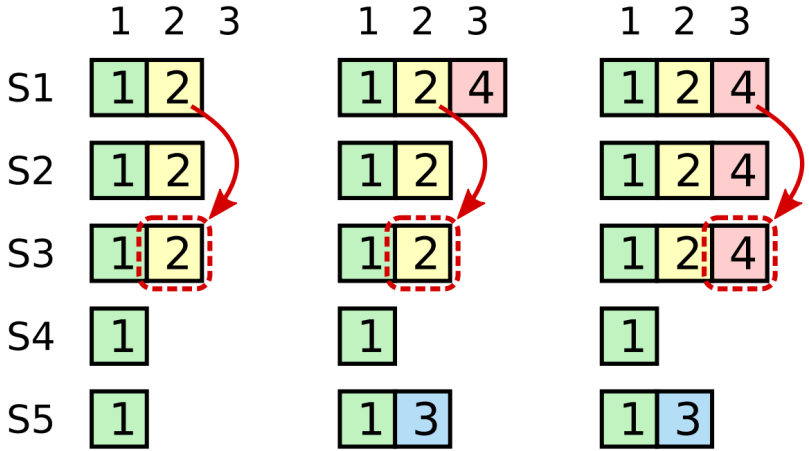
Коммиты



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

Коммиты



Ограничение на коммиты

- Пока новый лидер не закоммитит запись из **текущего** семестра, он считает предыдущие записи незакоммичеными
- См. случай 3 — после этого S5 не может стать лидером
- См. доказательство корректности[25]

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

Timing and availability

Timing requirement

$broadcastTime \ll electionTimeout \ll MTBF$

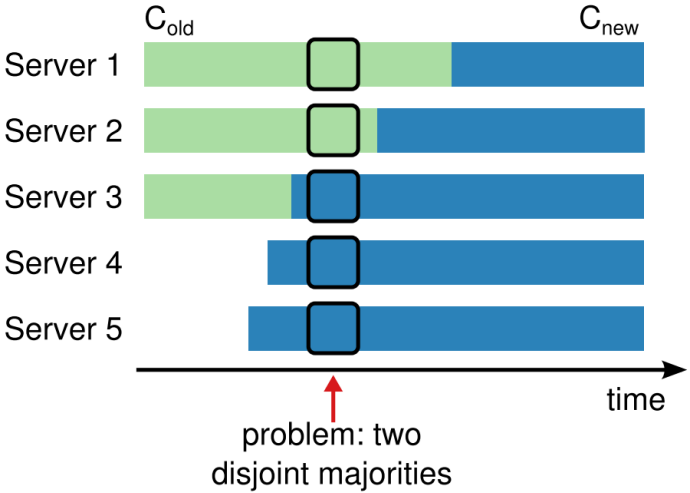
Типичные значения:

- $broadcastTime$: 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:

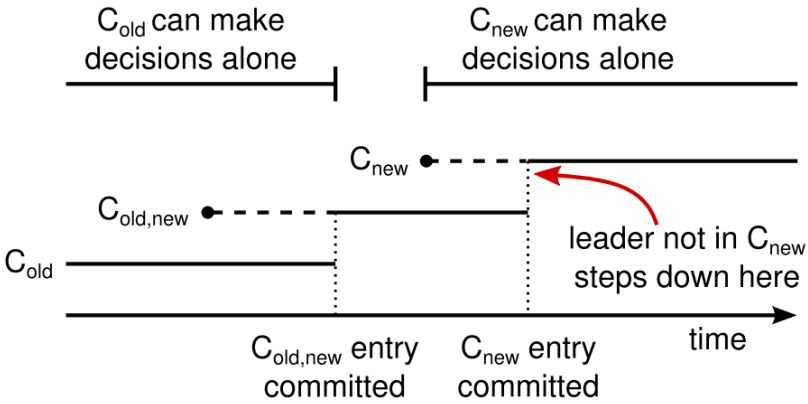
- 2PC: сначала выключаем старую конфигурацию, возможен downtime
- Raft: промежуточная конфигурация (joint consensus)

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

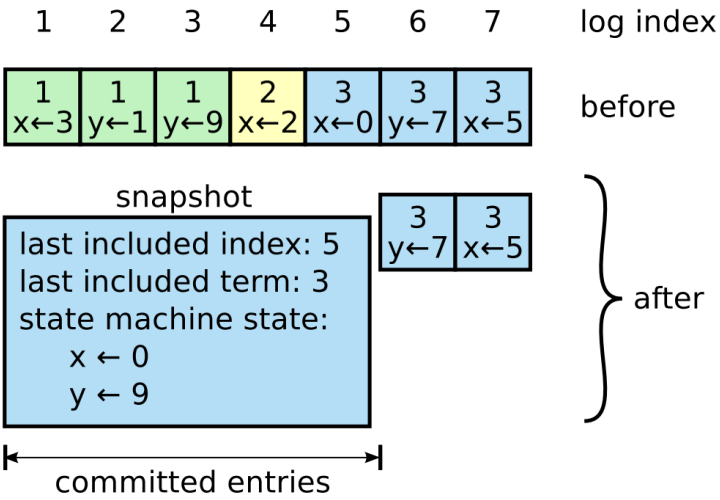
- Если лидер входит в C_{old}, но не в C_{new}, то должен выйти из кластера (реплицирует, но не входит в большинство)
- Новые сервера могут быть пустыми — вначале добавляем как неголосующих, но реплицируем на них логи
- Удаление серверов, которые не в курсе, что их удаляют, может снизить производительность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

Чтение на ночь

Fallacies of Distributed Computing[26][27]:

- The network is reliable
- Latency is zero
- Bandwidth is infinite
- The network is secure
- Topology doesn't change
- There is one administrator
- Transport cost is zero
- The network is homogeneous

[26] http://en.wikipedia.org/wiki/Fallacies_of_Distributed_Computing

[27] Arnon Rotem-Gal-Oz. Fallacies of Distributed Computing Explained, http://www.rgoarchitects.com/Files/fallacies.pdf

Библиотечка

- Google. Distributed Systems and Parallel Computing[28]
- Quora: What are the top startup engineering blogs?[29]

[28] http://research.google.com/pubs/DistributedSystemsandParallelComputing.html
[29] http://www.quora.com/What-are-the-top-startup-engineering-blogs

Вопросы?

- https://polis.mail.ru/curriculum/program/discipline/604/
- https://incubos.org/contacts/
- Не забудьте про ДЗ