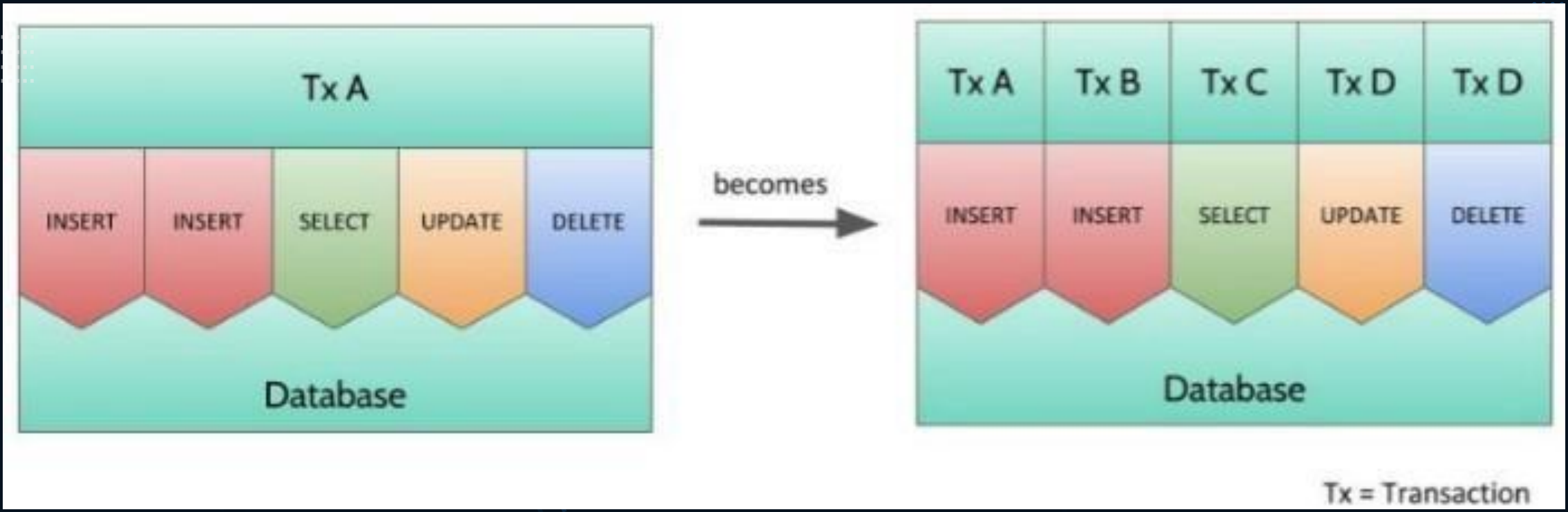
Системный Дизайн

Консистентность данных в распределенных системах



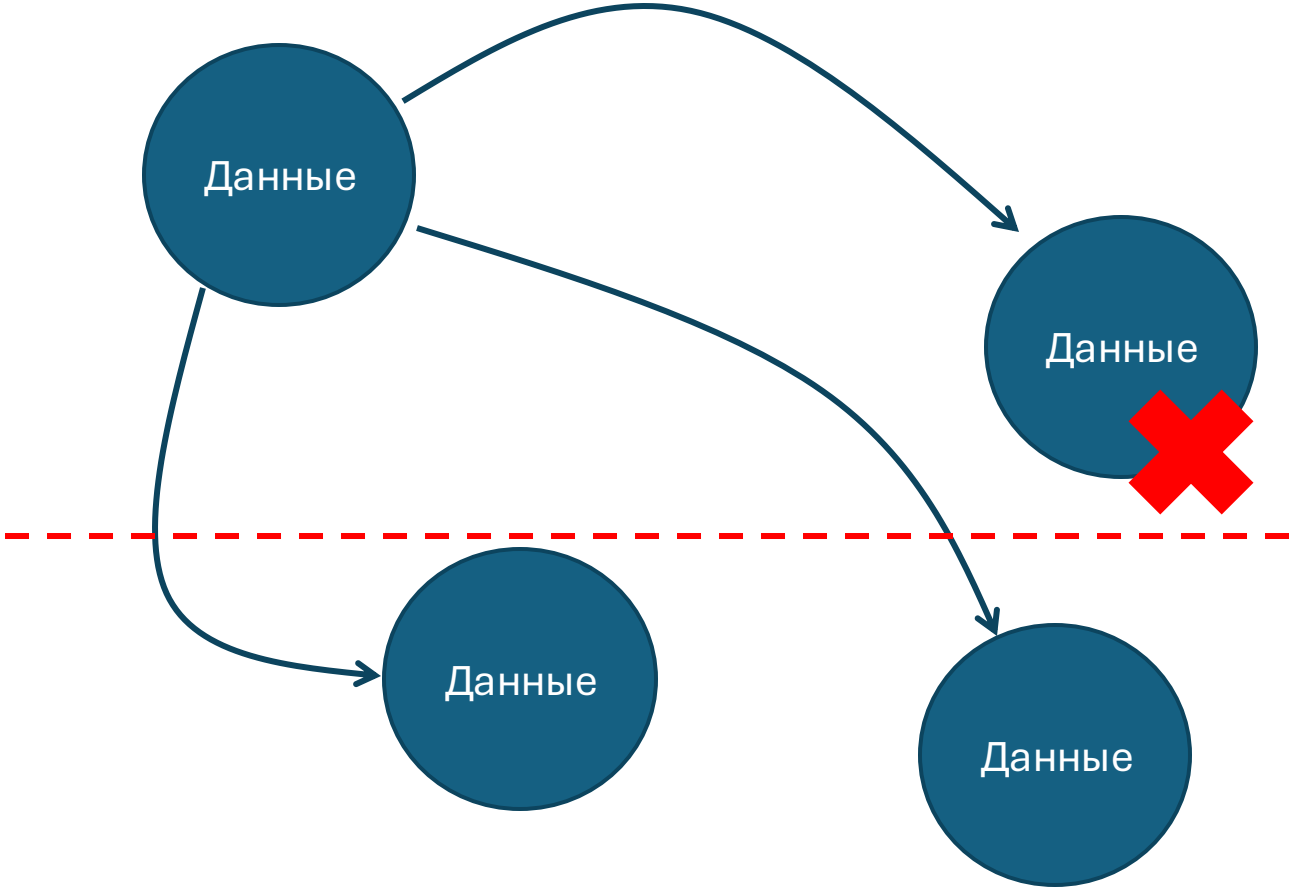
Значимость согласованности

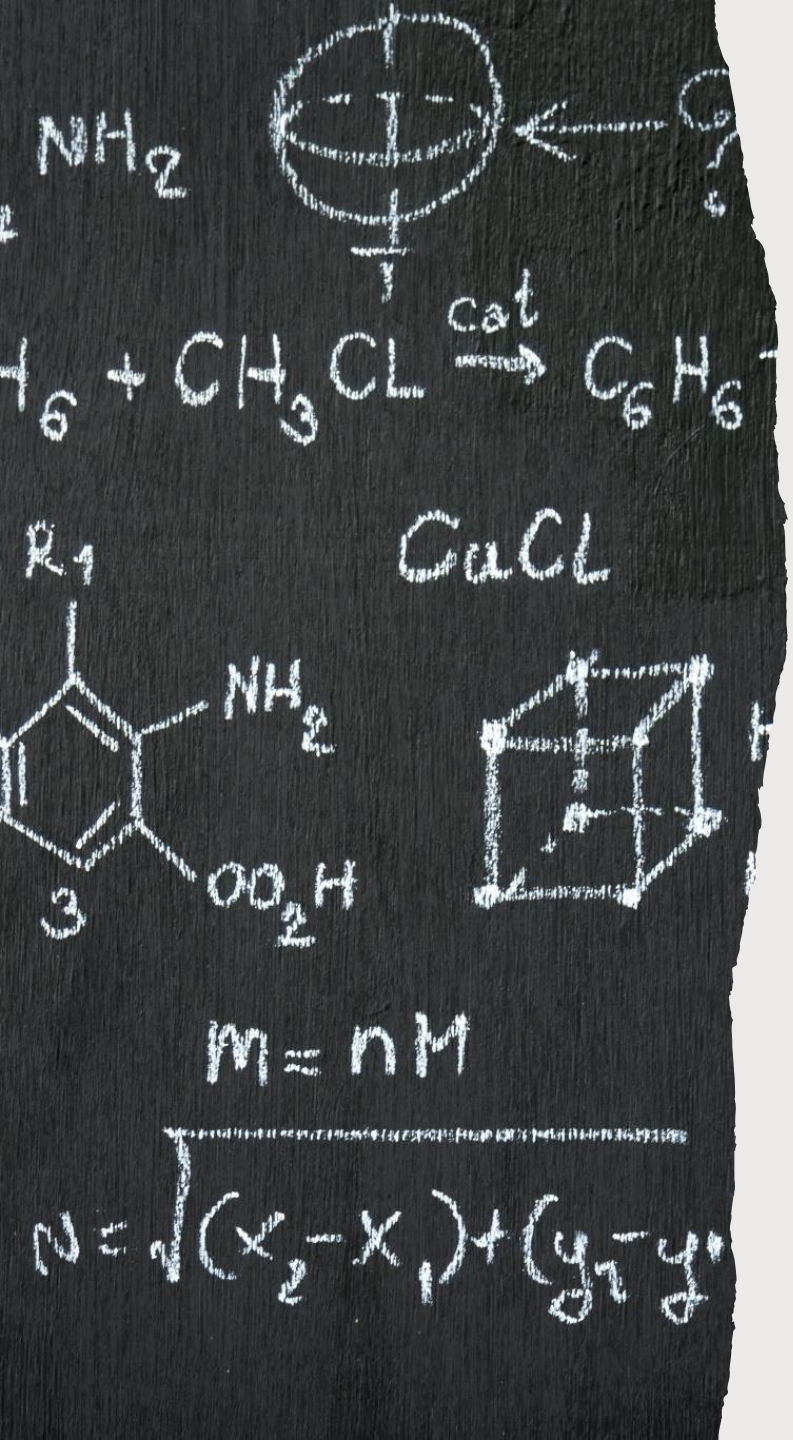
- Распределение данных по различным хранилищам приводит к невозможности одновременно поддерживать доступность и согласованность данных
- Проведение транзакционных операций в микросервисной архитектуре затруднено
- Решение проблемы согласованности – обязательный этап проектирования микросервисов

Согласованность

- Незавершенная (ошибочная) операция не вносит никаких эффектов и не меняет данные
- При конкурентном доступе к данным все операции рассматриваются как атомарные. Нельзя увидеть промежуточный результат операции
- Если мы имеем множество копий данных, то последовательность применения операций на всех копия одна и та же

Данные в распределенных системах



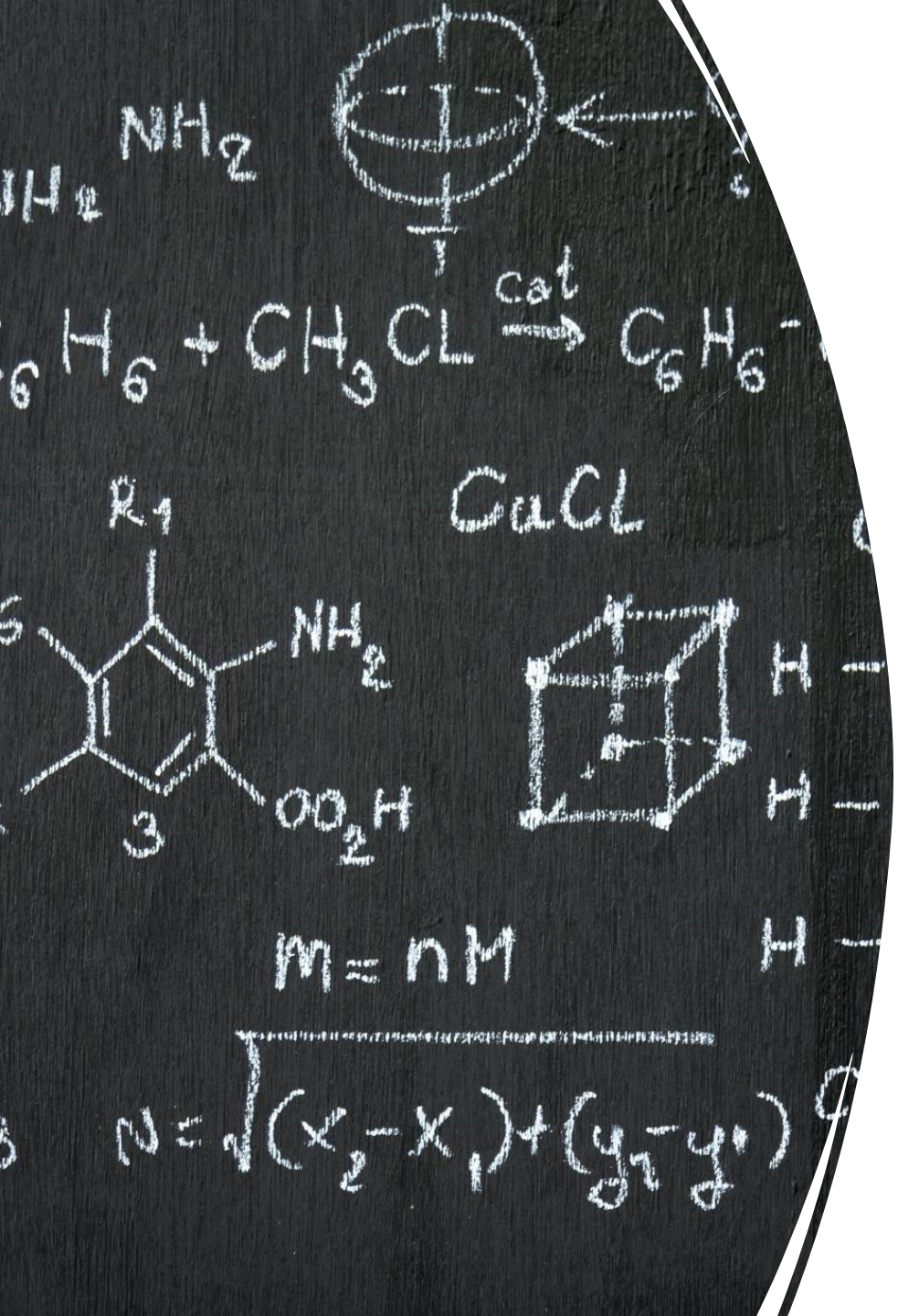


CAP теорема (Брюера)

- **C**onsistency
  каждое чтение данных дает самую актуальную версию данных

- **A**vailability
  каждый узел (не упавший) всегда успешно выполняет запросы на чтение и запись

- **P**artition tolerance
  даже если между узлами нет связи, они продолжают работать независимо друг от друга



CAP теорема гласит

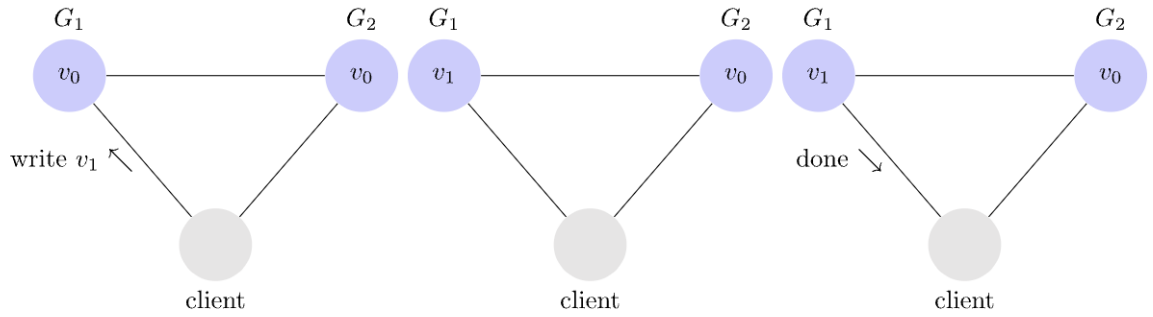
Из 3-х свойств можно одновременно обладать не более, чем двумя

доказательство теоремы

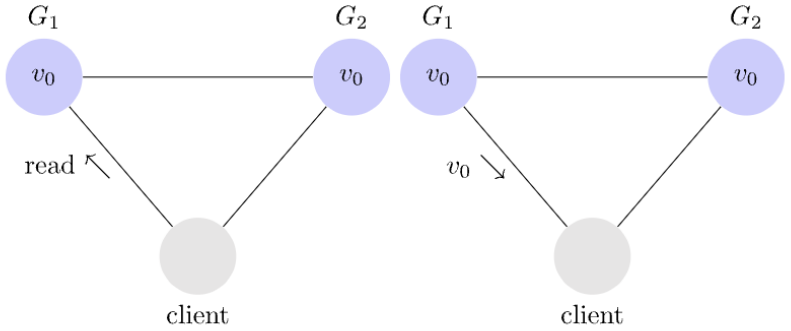
https://mwhittaker.github.io/blog/an_illustrated_proof_of_the_cap_theorem/

Простая распределенная система

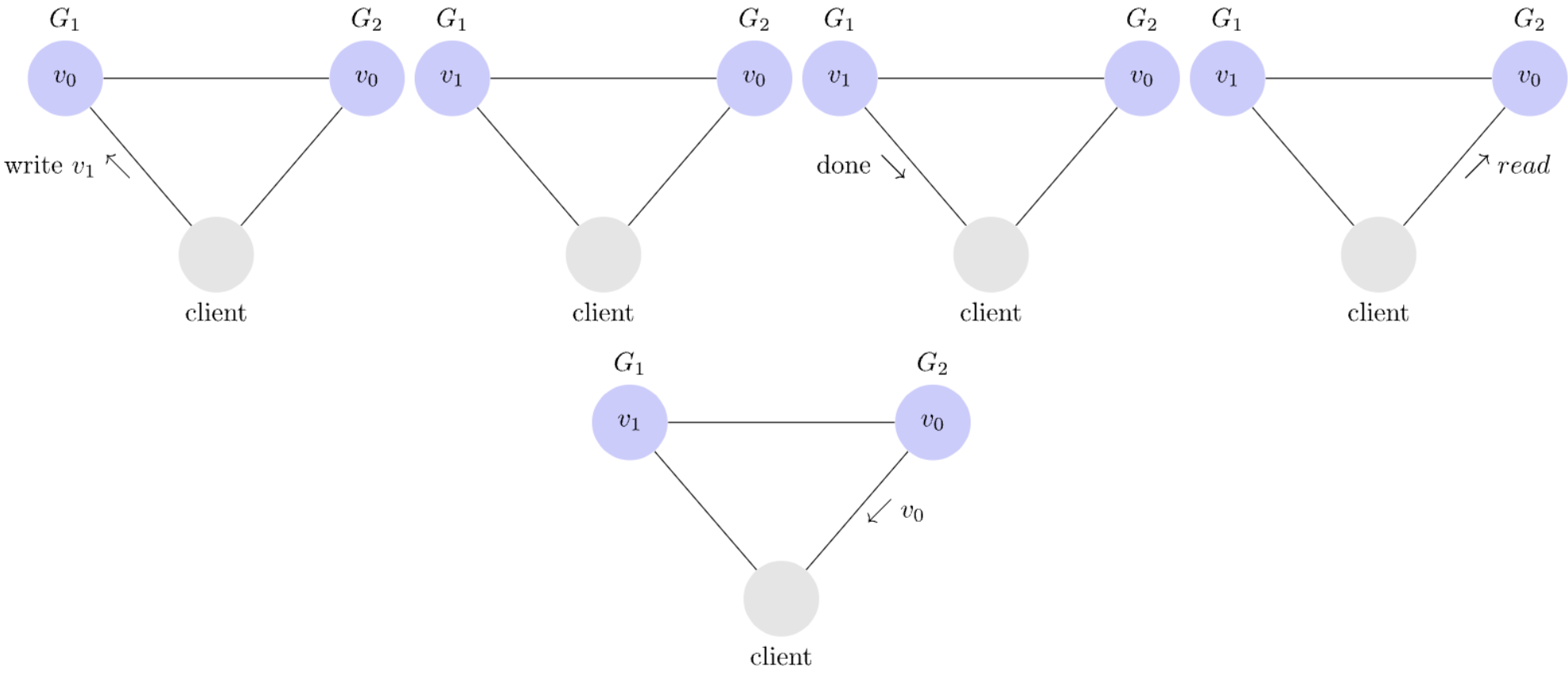
Запись



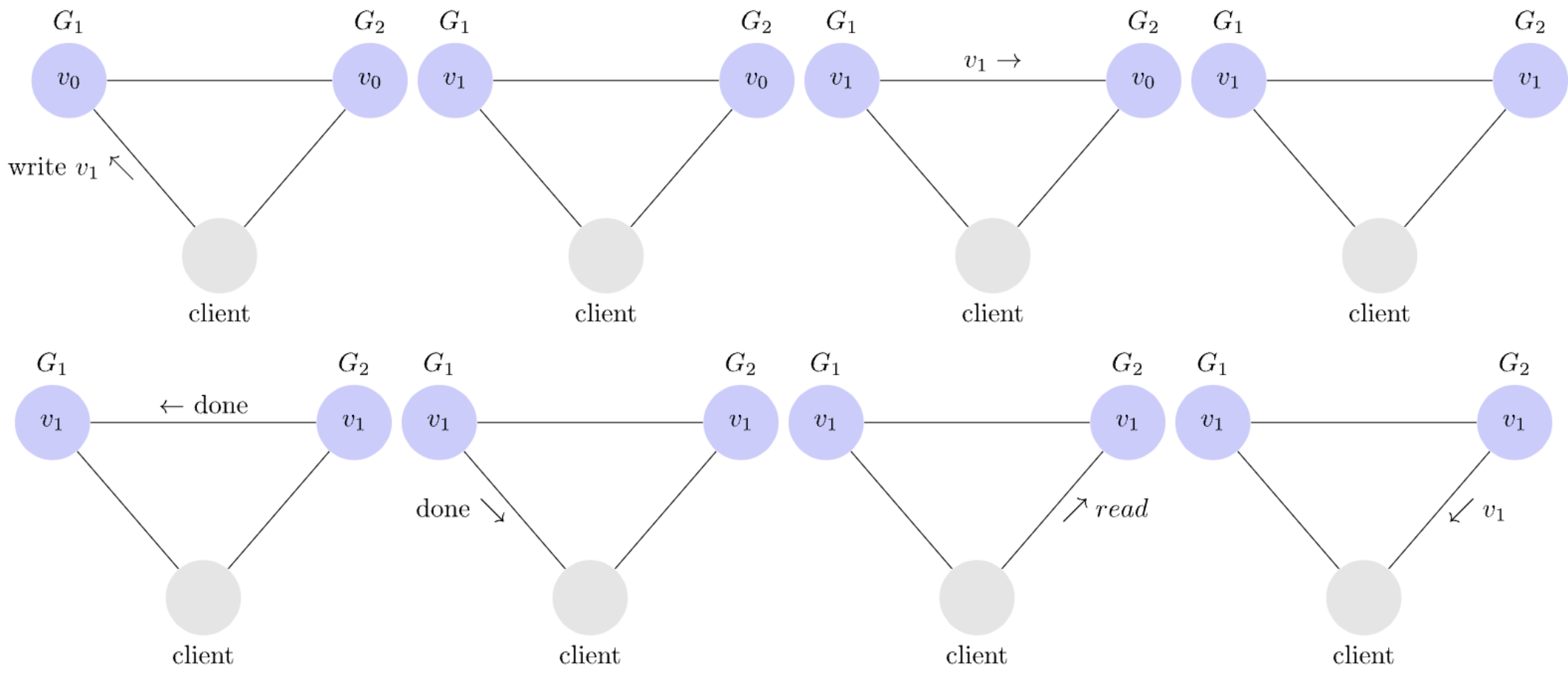
Чтение



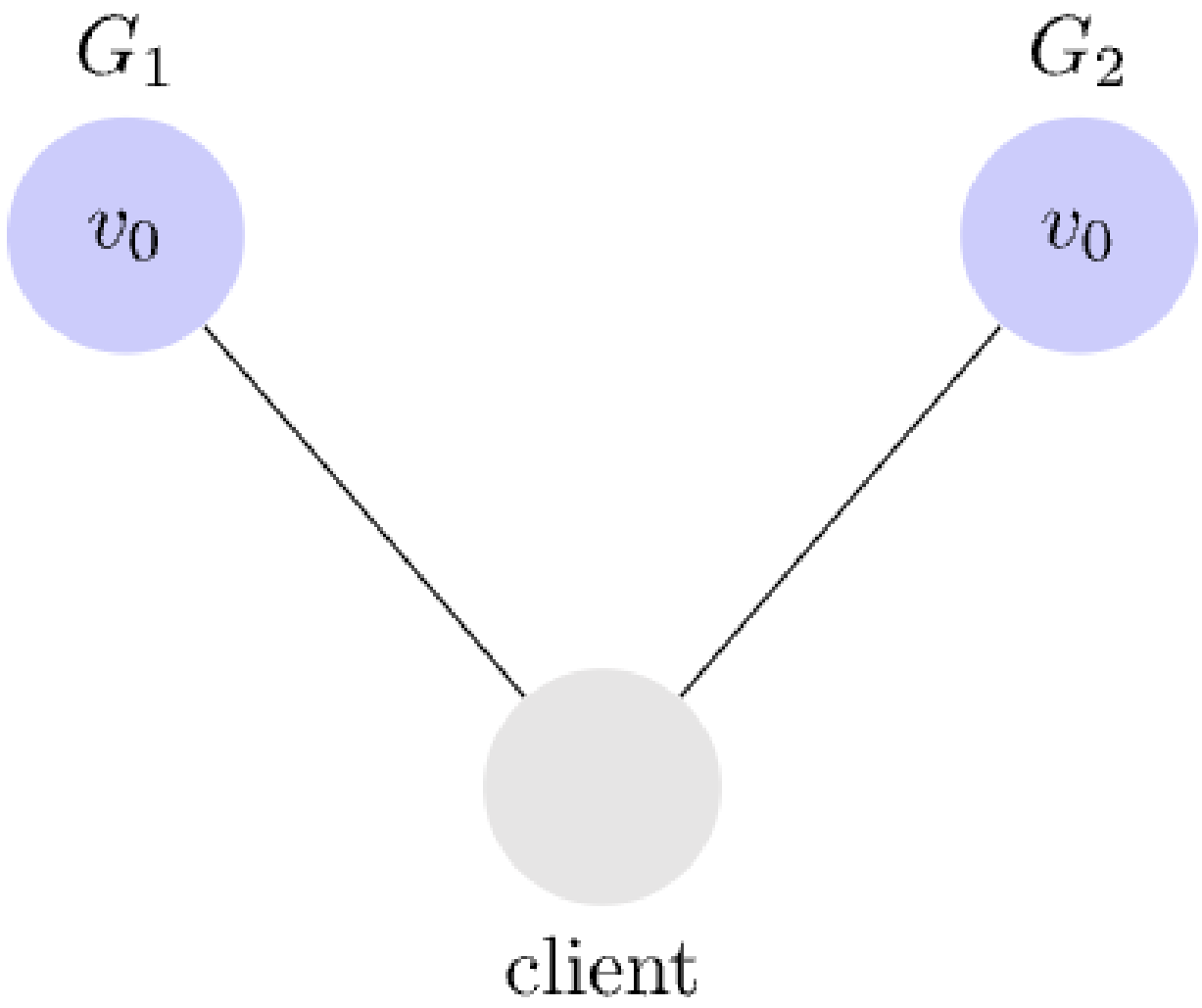
Не консистентная система

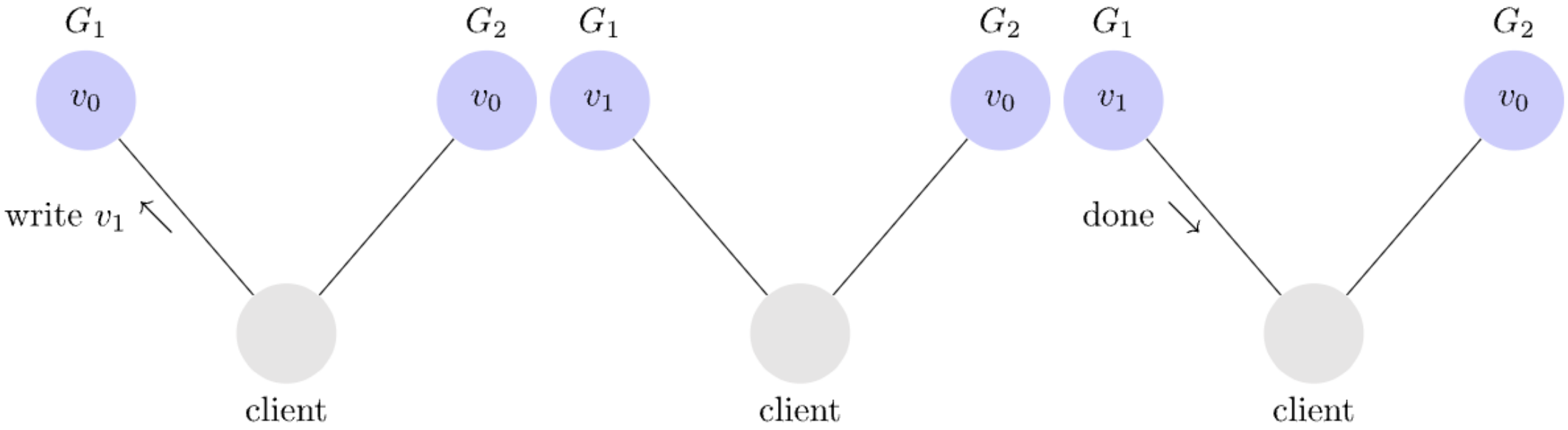


Консистентная система



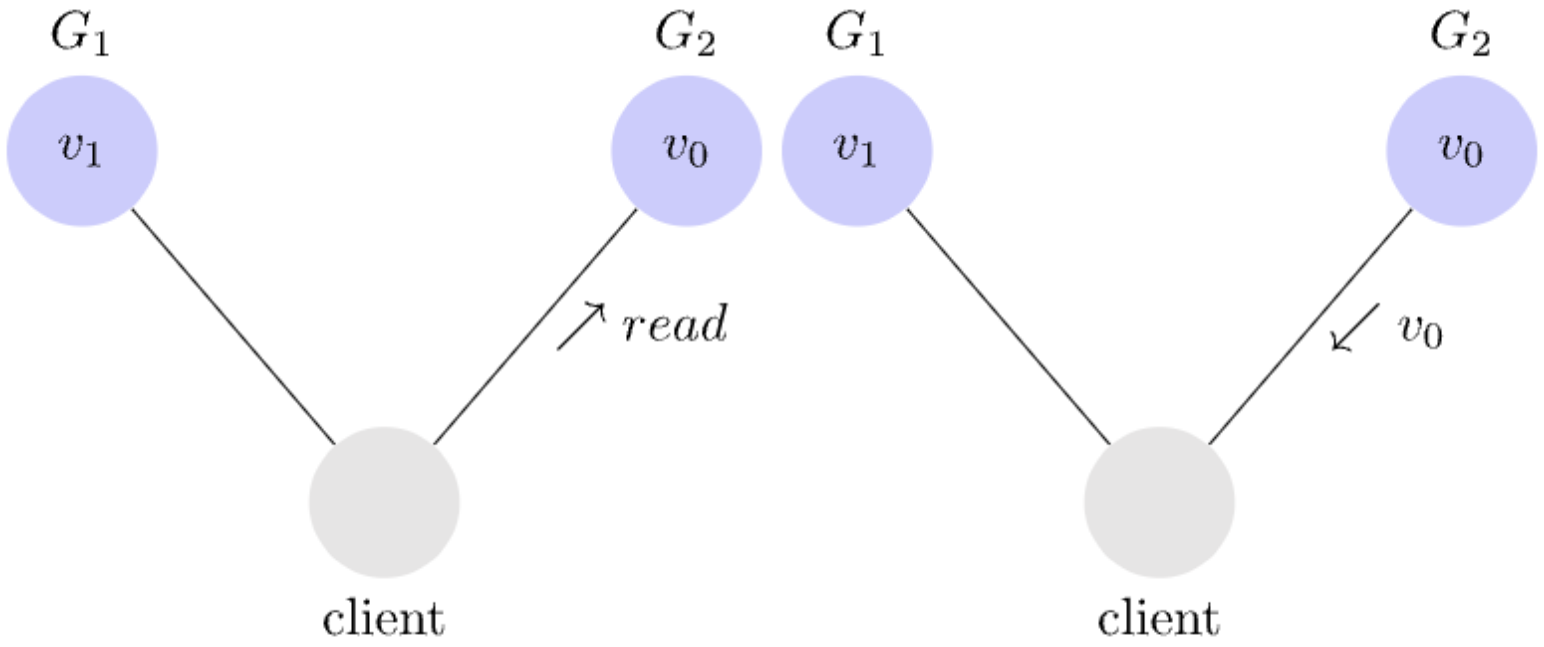
Пусть система удовлетворяет всем свойствам: consistent + available + partition tolerance





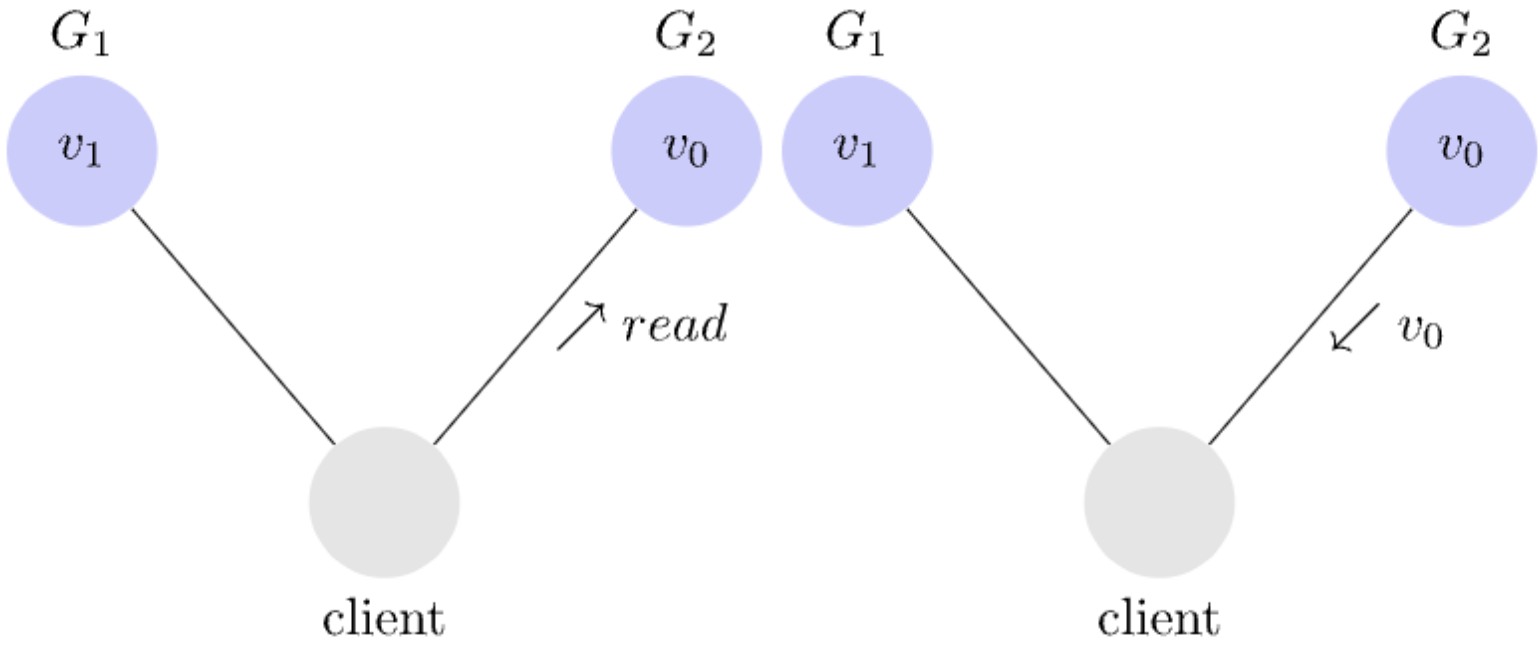
Пишем данные

Далее наш клиент запрашивает запись v1 в G1. Поскольку наша система доступна, G1 должен ответить. Однако, поскольку сеть разделена на части, G1 не может реплицировать свои данные на G2.



Читаем данные

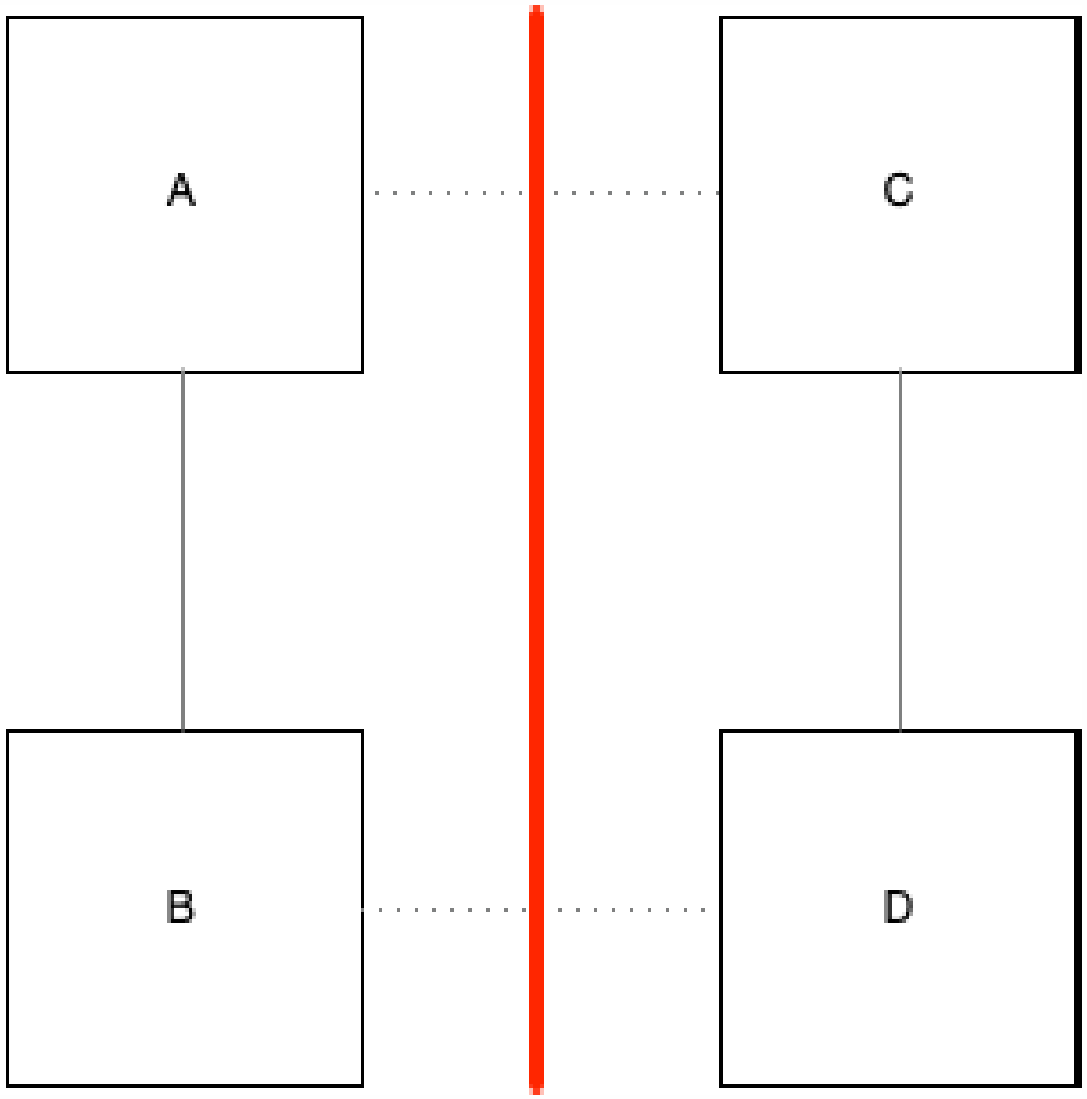
Далее мы попросим нашего клиента отправить запрос на чтение к G2. И снова, поскольку наша система доступна, G2 должен ответить. А поскольку сеть разделена, G2 не может обновить свое значение из G1. Она возвращает v0.



Не консистентность

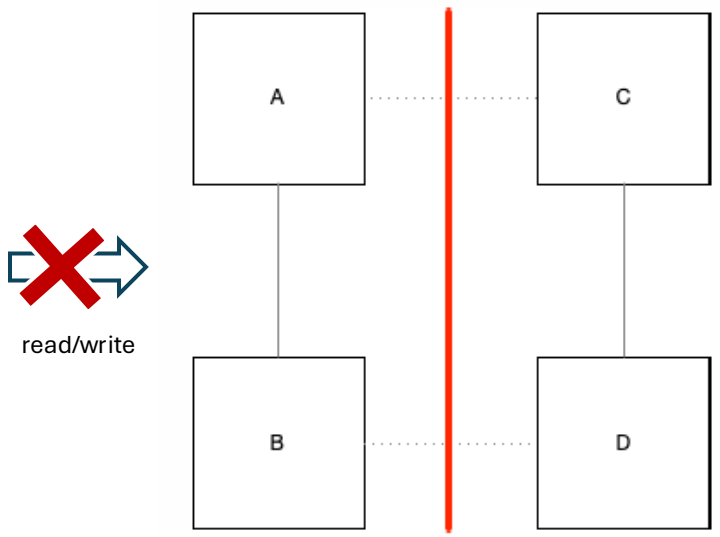
G2 возвращает v0 нашему клиенту после того, как клиент уже записал v1 в G1. Это несовместимо.

Что делать если сеть распалась на две части?



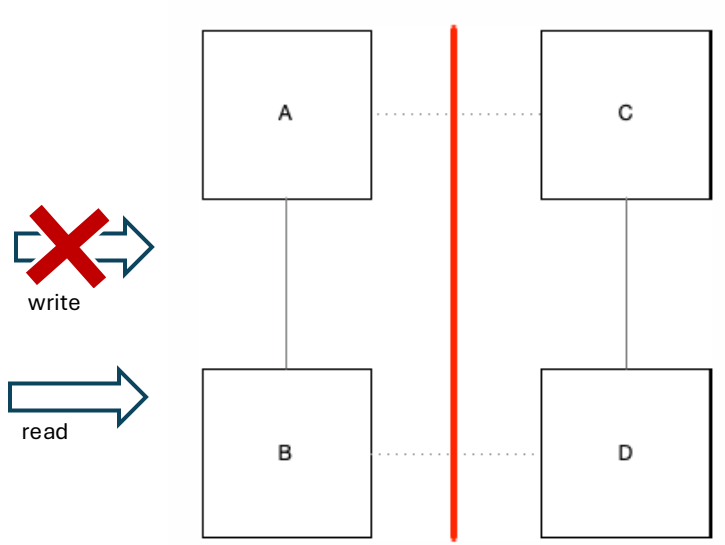
Что делать если сеть распалась на две части?

- 1 Не принимаем запросы (AC)



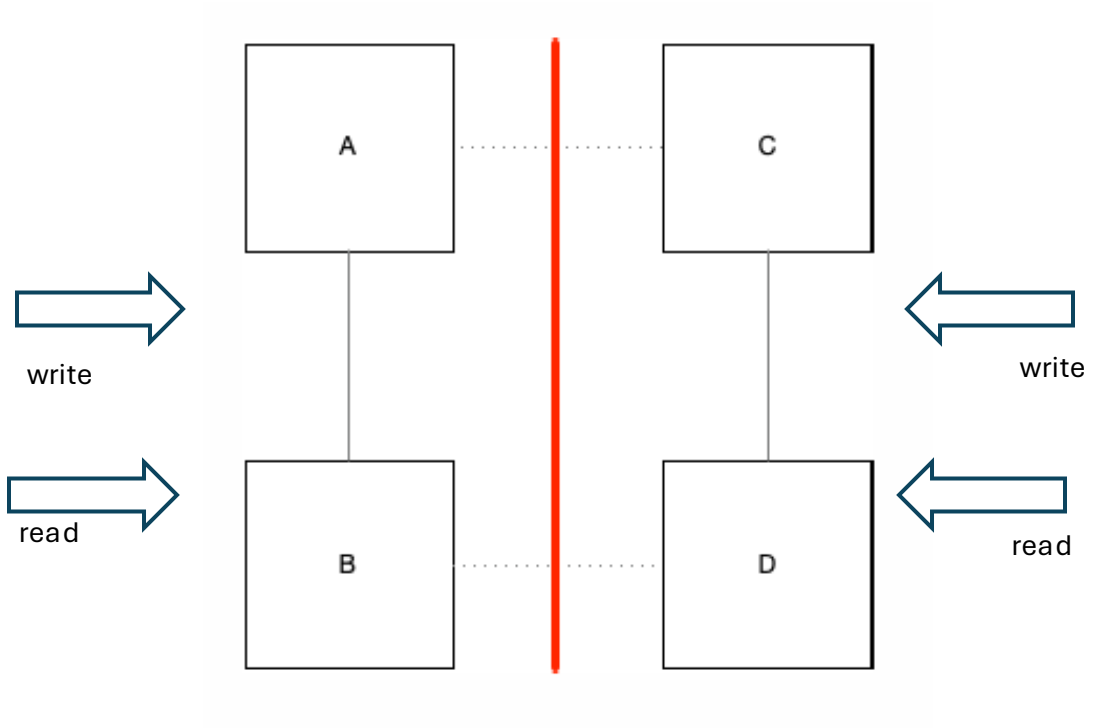
Что делать если сеть распалась на две части?

- 2 Разрешаем чтение, запрещаем запись (CP)

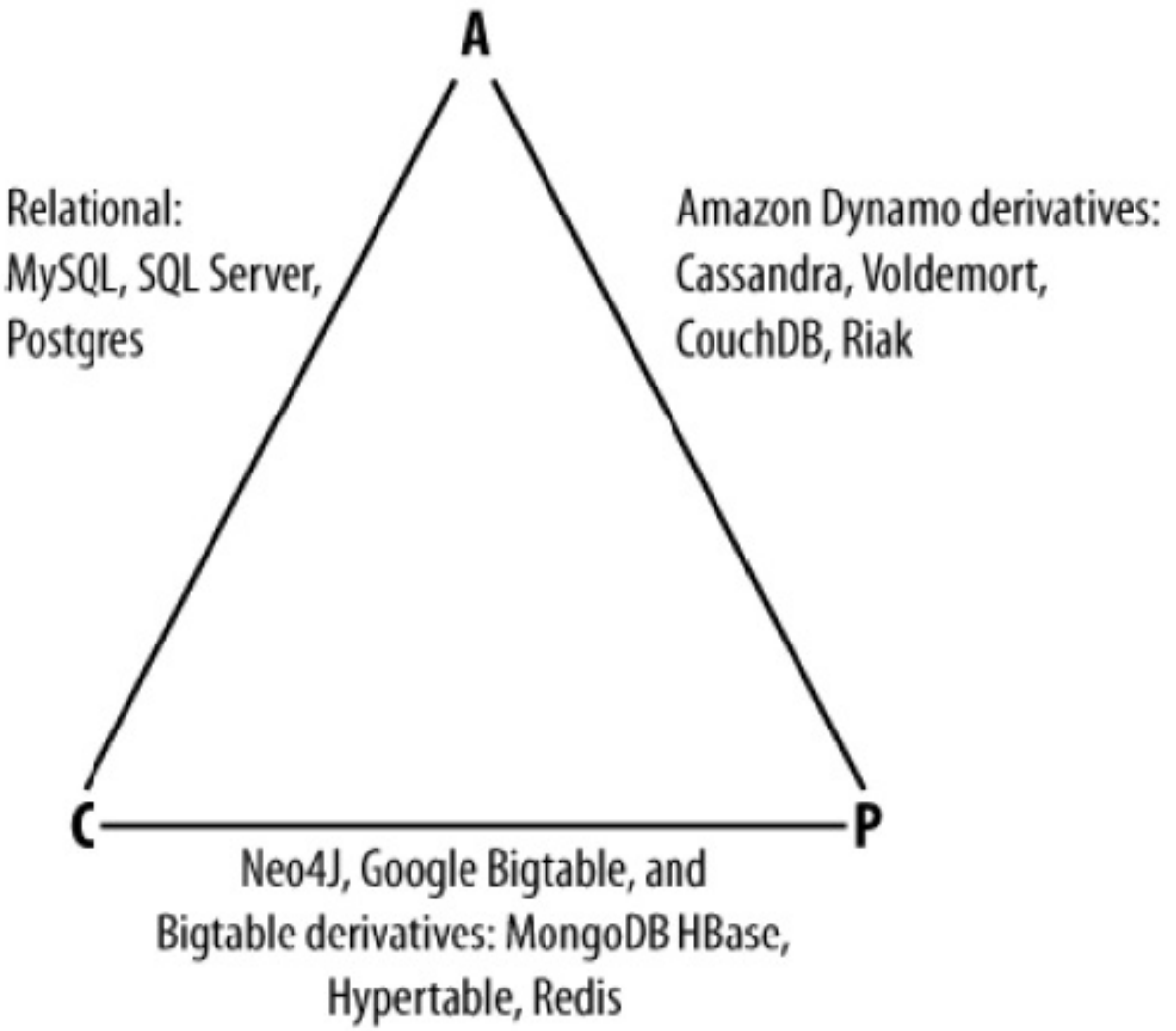


Что делать если сеть распалась на две части?

- 3 Разрешаем чтение и запись (AP)



CAP теорема и СУБД



Распределенные системы

Фактический выбор между AP и CP

Тактики работы с несогласованными данными

Алгоритмы согласования

это механизм, в распределенной системе, который позволяет участникам, в данной системе, достичь соглашения о конкретной единице информации.

Тактики работы с несогласованными данными

1. Оптимистичная согласованность
2. Двухфазные коммиты (2PC)
3. Паттерн «Сага» (Saga)
4. Различные CP leader-based алгоритмы

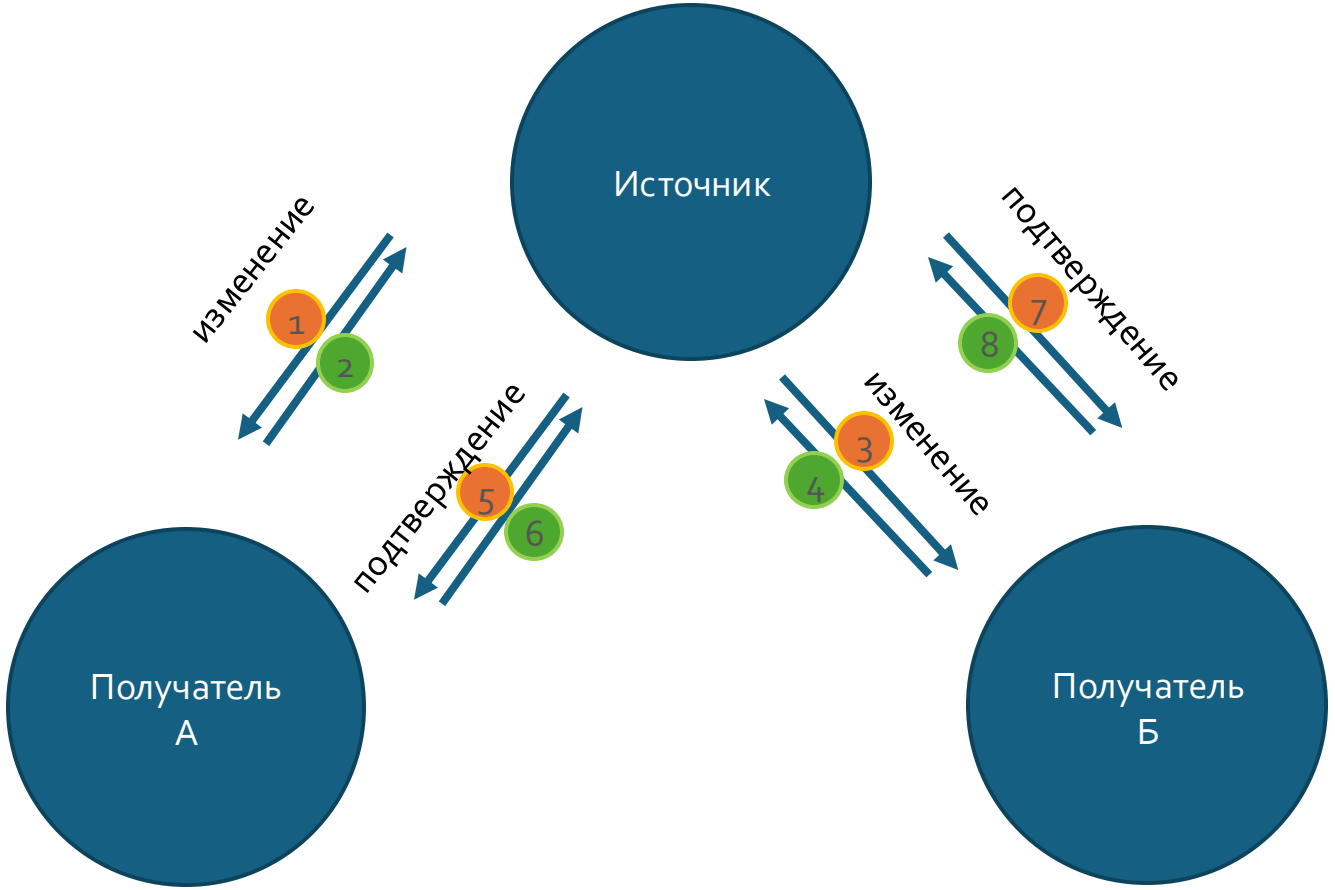
Оптимистичная согласованность

анти-паттерн

- Считаем, что большинство операций завершаются успешно.
- Дополнительные действия производим уже по факту произошедшего сбоя.
- Уменьшаем издержки для большинства операций, что приводит к повышению производительности.

2 Двухфазные комиты

Одновременная запись в несколько хранилищ



Двухфазные коммиты

1. Создать объект «транзакция», который содержит данные для записи в источники данных A и B
2. Запись данных в источник данных A
    - Атомарная (в локальной транзакции внутри источника данных)
    - Метка транзакции
    - Если данные уже занесены, то ничего не делаем
3. Запись данных в источник данных B
4. Подтверждение и удаление объекта «транзакция»

Двухфазные коммиты: алгоритм

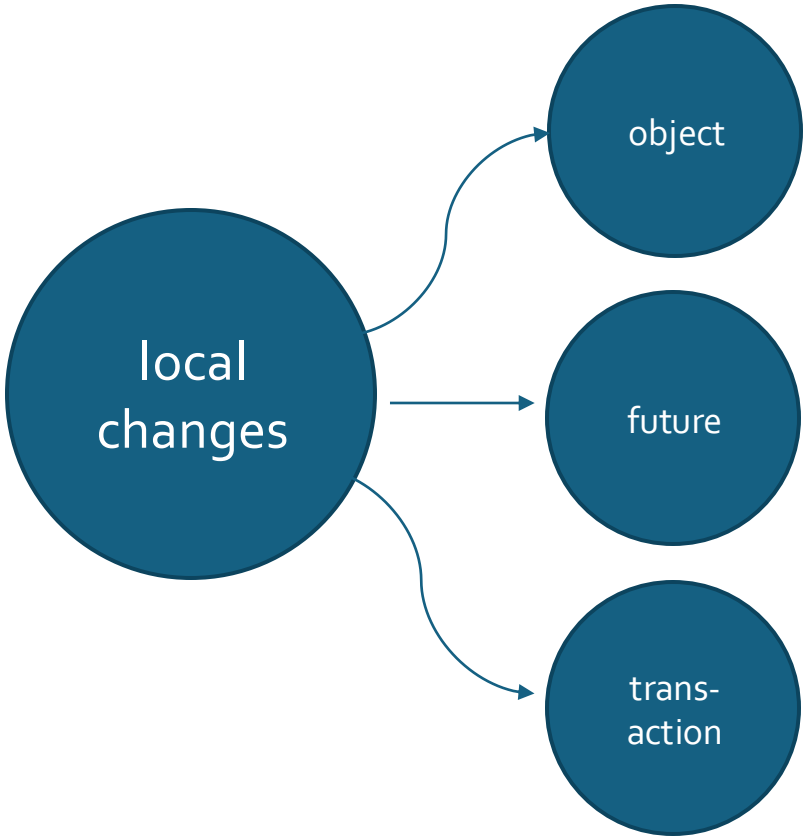
Необходимо доп. хранилище local_changes, содержит записи вида:

- Ссылка на объект
- Как будет выглядеть объект после успешного завершения транзакции
- Ссылка на транзакцию

Транзакции будут содержать два флага:

committed = false по умолчанию

aborted = false по умолчанию



Двухфазные коммиты:

чтение

- Если local_changes пустой для данного объекта, то возвращаем данные из БД

- Если local_changes не пустой, проверяем состояние транзакции:

  • committed = true –возвращаем данные из local_changes

  • aborted = true –возвращаем данные из БД

  • Иначе конфликт –действия зависят от уровня изолированности транзакций

Двухфазные коммиты:

запись

- Если **local_changes** пустой для данного объекта, то записываем свои данные и ссылку на свою транзакцию в **local_changes**

- Если **local_changes** не пустой, проверяем состояние транзакции
  - **committed = true** записываем данные из **local_changes** в БД, затем записываем свои данные и ссылку на свою транзакцию в local_changes
  - **aborted = true** записываем свои данные и ссылку на свою транзакцию в **local_changes**
  - Иначе конфликт действия зависят от уровня изолированности транзакций: нужно предусмотреть варианты отмены «висящей» транзакции (aborted = true)

Двухфазные транзакции: результат

1. Создать транзакцию
2. Обращаемся на запись в источник данных A и B
3. Подготавливаем коммит в источниках A и B
4. Если транзакция ещё жива, то делаем Commit в источниках A и B
5. Cleanup: проходим по затронутым объектам, и если в local_changes есть законченная транзакция, то применяем изменения и чистим local_changes

Двухфазные комиты

Преимущества:

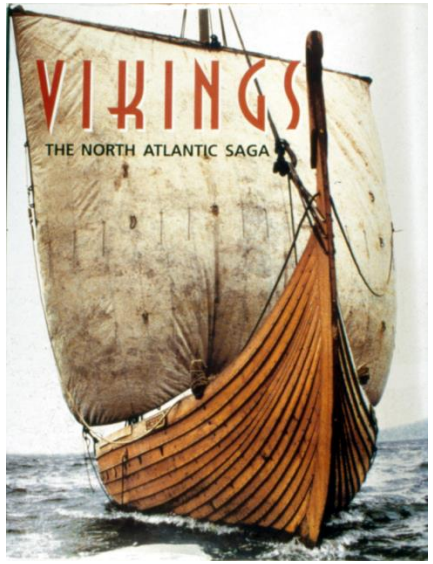
- Поддерживают сильную согласованность в распределённой системе

Недостатки:

- Современные NoSQL БД и брокеры сообщений не поддерживают двухфазные коммиты, требуется реализация в коде.
- Требования к надёжности сервисов и сетевого оборудования противоречат целям микросервисной архитектуры

Вывод: двухфазные коммиты являются антипаттерном микросервиснойархитектуры

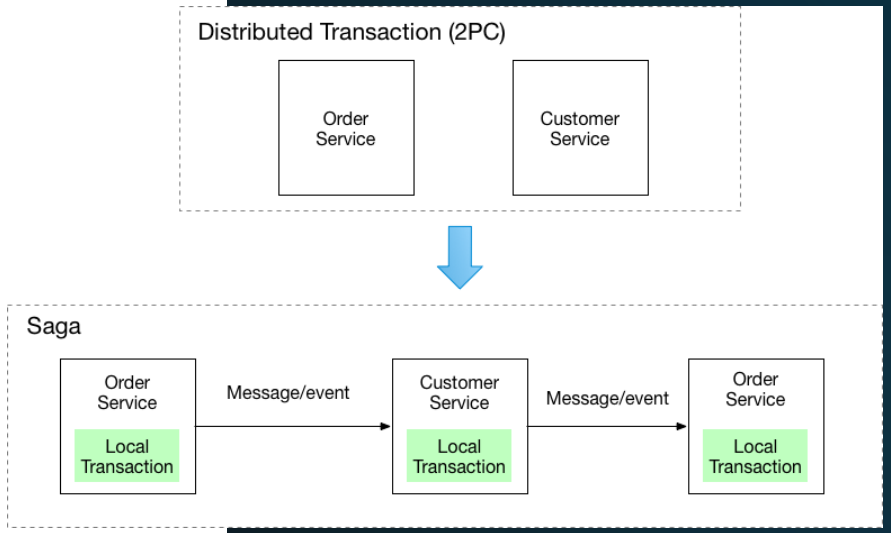
3 Паттерн Saga



Паттерн «Сага» («Повествование», Saga)

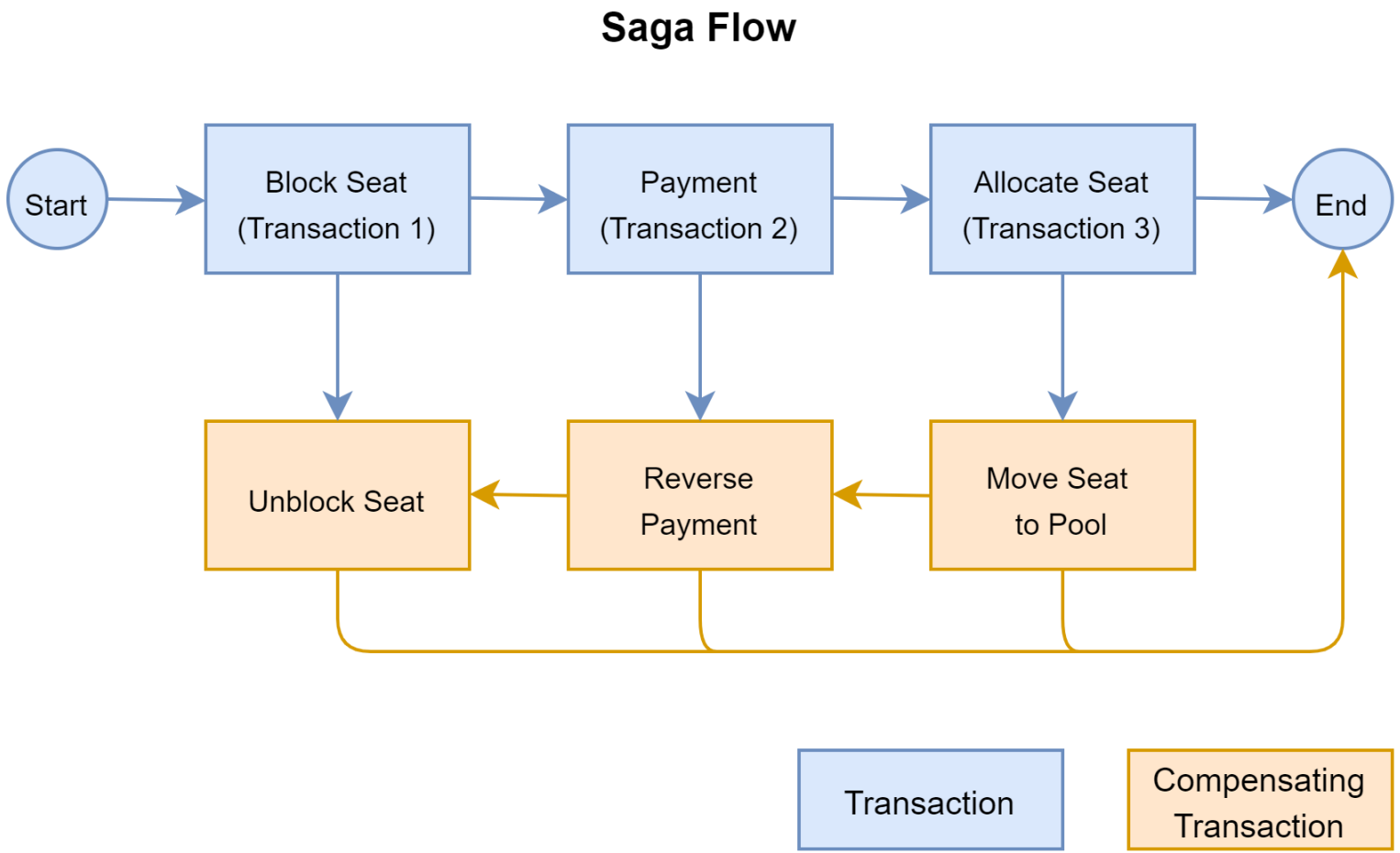
Механизм, обеспечивающий согласованность данных в микросервисной архитектуре без применения распределенных транзакций.

- Сага создается для каждой системной команды, которой нужно обновлять данные в нескольких сервисах.
- Это последовательность локальных ACID-транзакций, каждая из которых обновляет данные в одном сервисе
- В случае сбоя на всех сервисах, где локальная транзакция уже завершена, выполняется компенсирующая транзакция



https://www.baeldung.com/wp-content/uploads/sites/4/2021/04/Figure-3.png

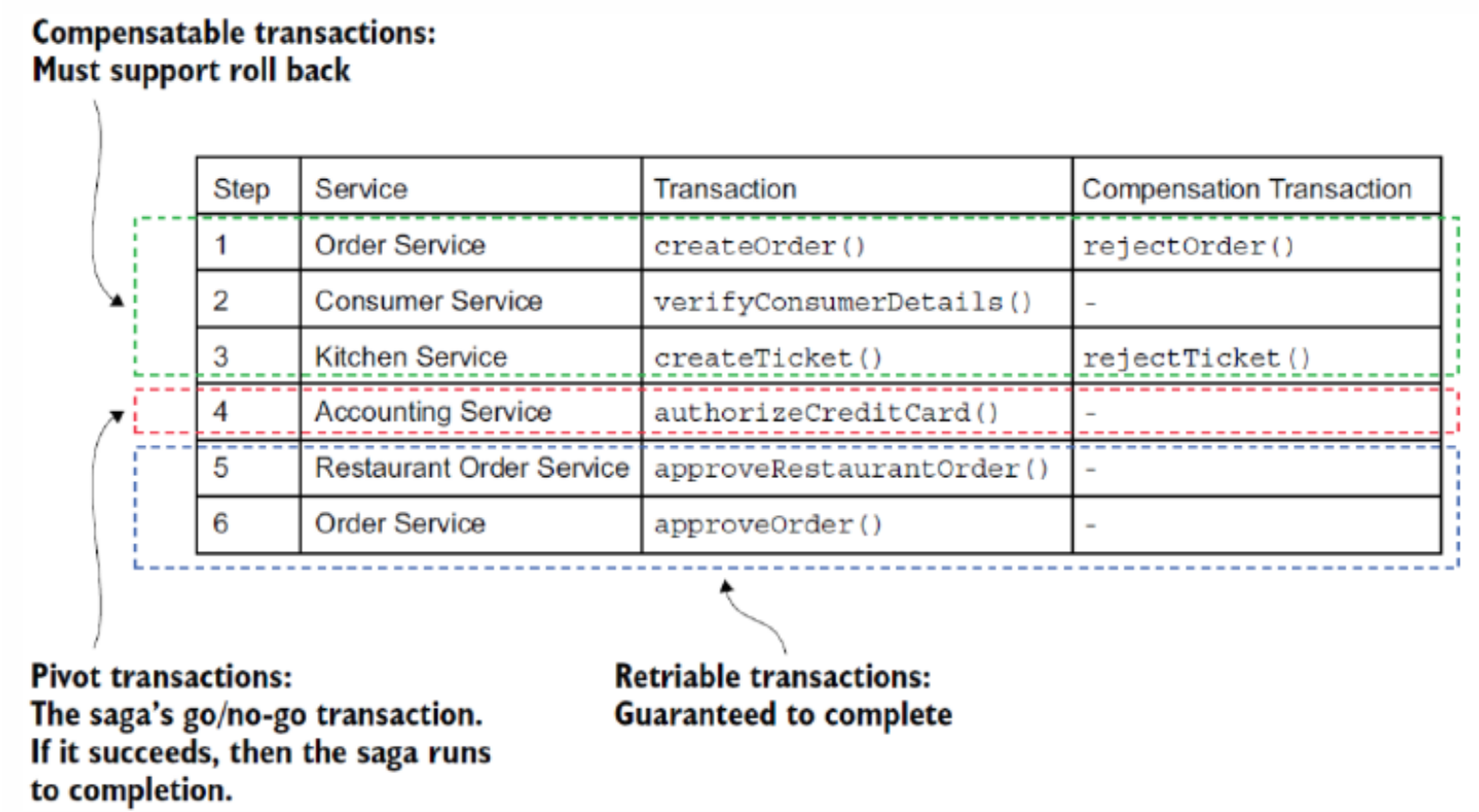
Паттерн Сага



Паттерн Сага

- Не применяем распределённые транзакции
- На каждом сервисе выполняется локальная транзакция
- Все транзакции объединены в Сагу
- **Компенсируемые** транзакции (противотранзакции)
- **Поворотная** транзакция определяет успешность саги
- **Повторяемые** транзакции выполняются после поворотной и повторяются до успеха

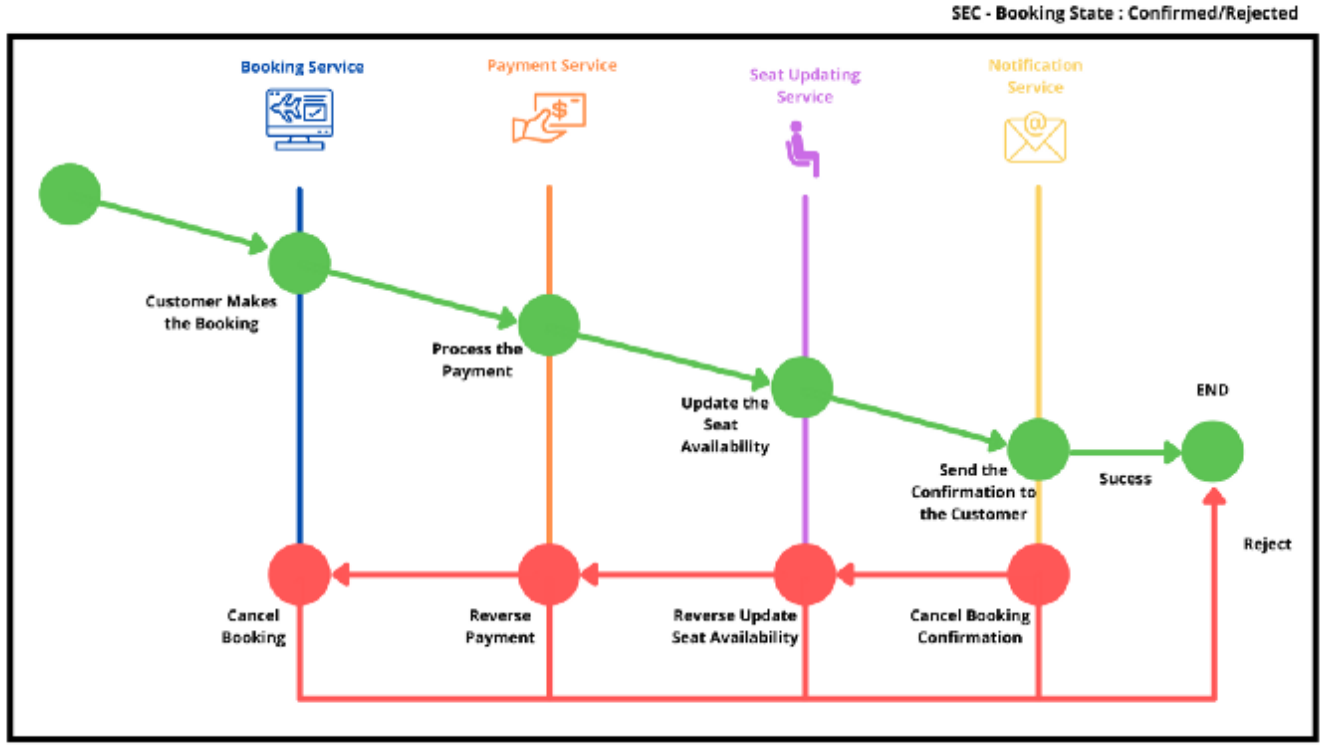
Поворотная транзакция (pivot)



Step	Service	Transaction	Compensation Transaction
1	Order Service	`createOrder()`	`rejectOrder()`
2	Consumer Service	`verifyConsumerDetails()`	-
3	Kitchen Service	`createTicket()`	`rejectTicket()`
4	Accounting Service	`authorizeCreditCard()`	-
5	Restaurant Order Service	`approveRestaurantOrder()`	-
6	Order Service	`approveOrder()`	-

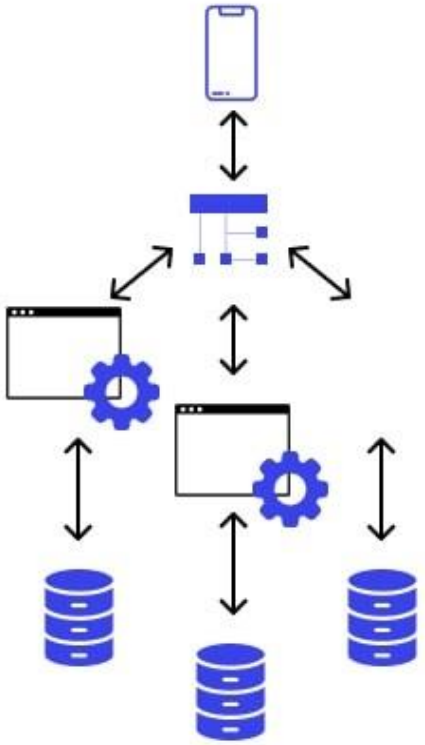
Сага при оркестрации сервисов

В Orchestration-Based Saga один оркестрант (организатор) управляет всеми транзакциями и направляет сервисы на выполнение локальных транзакций.

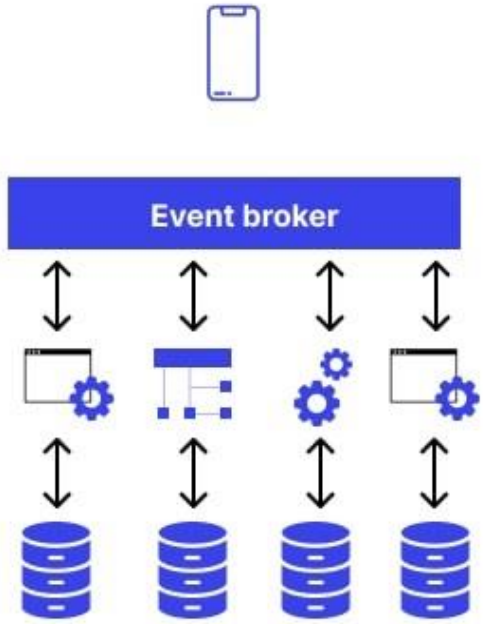


- https://blog.bitsrc.io/how-to-use-saga-pattern-in-microservices-9eaadde79748

Оркестрация и хореография

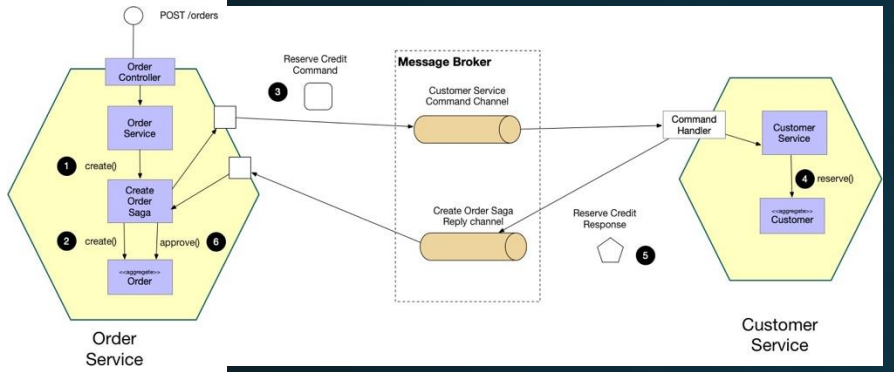


wallarm



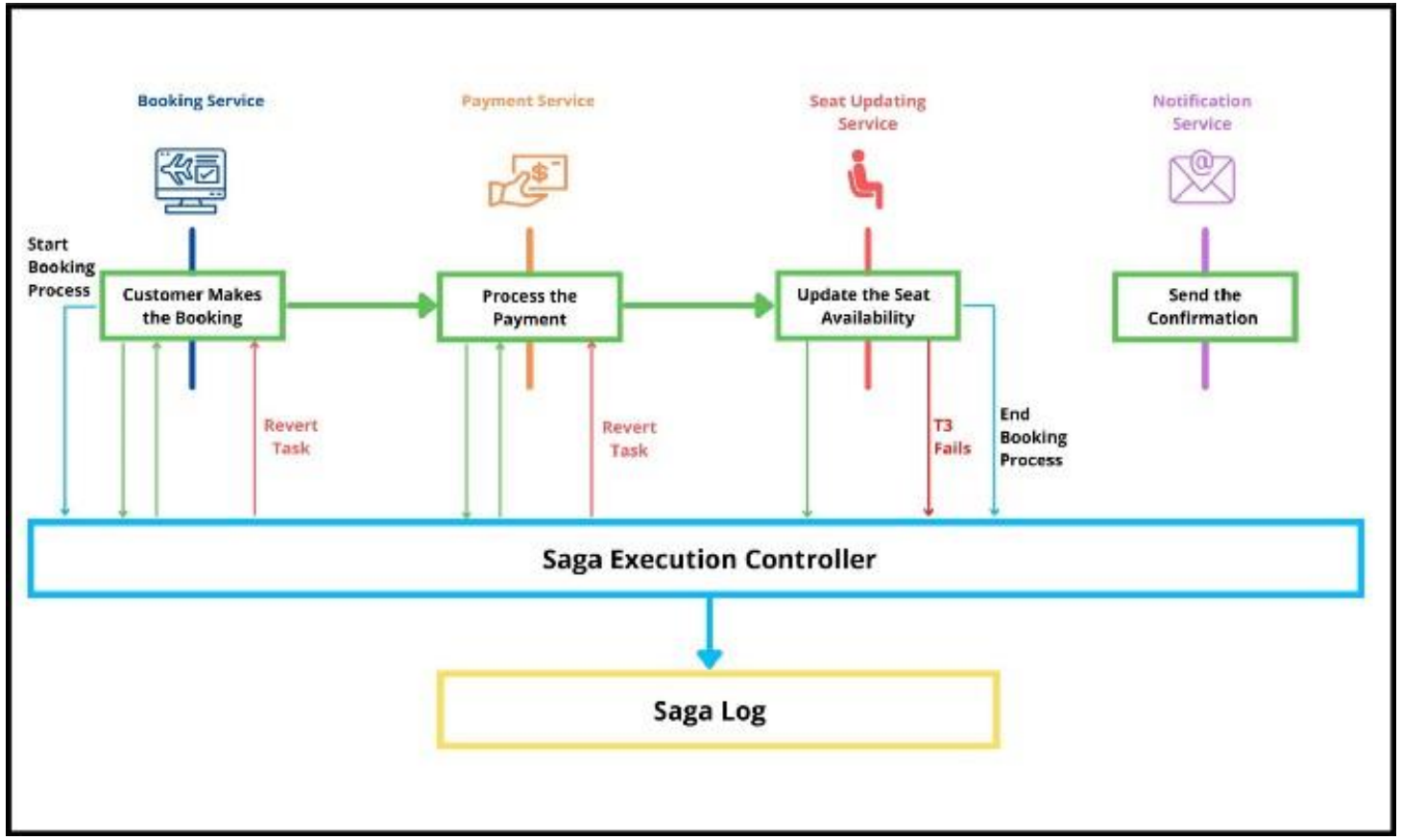
Сага – оркестратор: пример

- Сервис заказов получает запрос POST/orders и создает сагу-оркестратор «Create Order»
- Сага-оркестратор создает заказ в состоянии PENDING
- Сага-оркестратор отправляет команду резервирования в сервис пользователей
- Сервис пользователей пытается зарезервировать товар
- Сервис пользователей отправляет ответное сообщение с указанием результата
- Сага-оркестратор одобряет или отклоняет заказ



Сага при хореографии сервисов

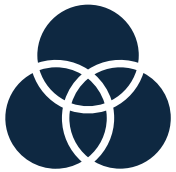
В Saga на основе хореографии все сервисы, являющиеся частью распределенной транзакции, публикуют новое событие после завершения своей локальной транзакции.



Паттерн Сага: выводы

Преимущества:

Позволяет приложению поддерживать согласованность в конечном счёте без использования распределенных транзакций



Недостатки:

Модель программирования является более сложной, требуются компенсирующие транзакции

Сага поддерживает ACD модель (нет изолированности)

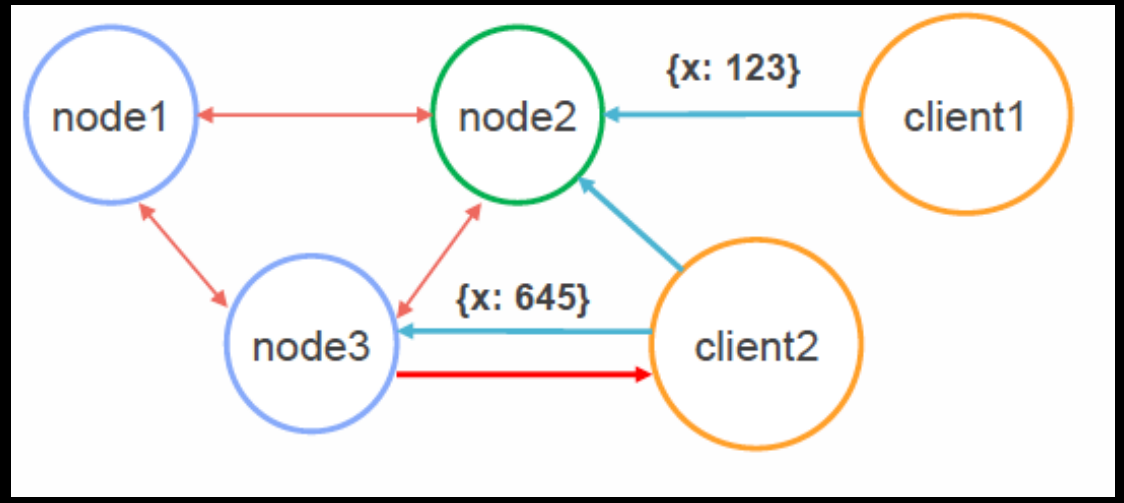
Проблемы:

Чтобы быть надежным, сервис должен атомарно обновить свою базу данных и опубликовать сообщение/событие.

4 Leader-based CP алгоритмы

Принципы

- **Согласование изменений.** Выбирается лидер
- **Все изменения** идут через **лидера**
- Все **другие** ноды являются **follower’ами**



Где применяются?

- Синхронизация состояния (изменений): базы данных, блокчейн
- Разрешать коллизии (за счет голосования): ZAB (kafka)
- Оркестрация:docker swarm

RAFT

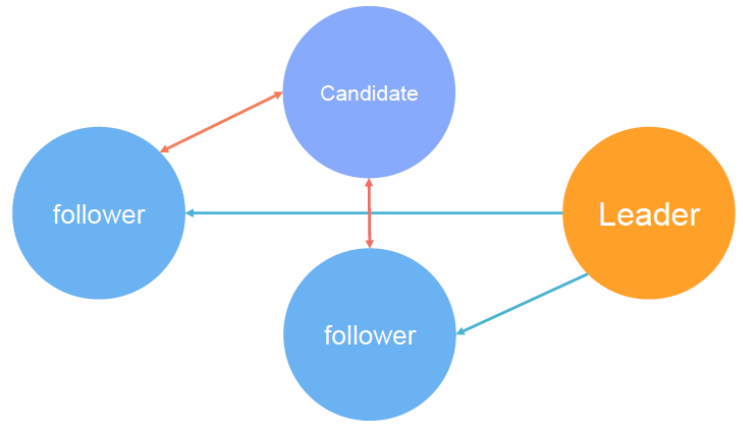
пример алгоритма

RAFT

- CP алгоритм
- Синхронная система: использует таймеры
- Переработанный концепт алгоритма paxos

Принцип работы

- Leader –принимает изменения и реплицирует от клиента
- Follower –принимает изменения от leader
- Candidate –должен стать leader или follower



https://raft.github.io/

Demo

Транзакционная отправка сообщений

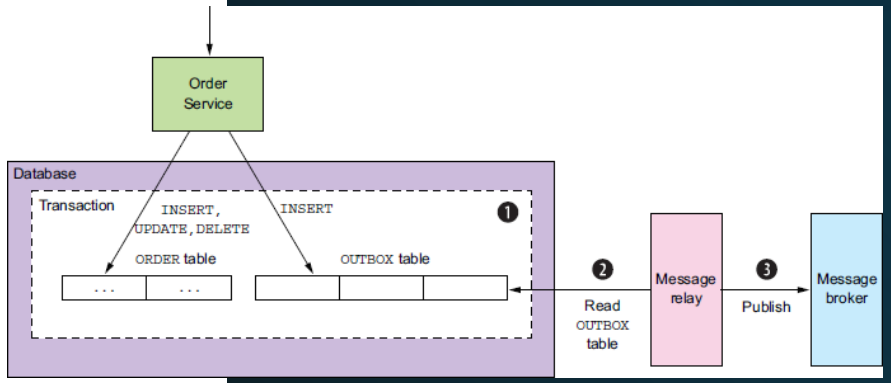
Паттерны транзакционного обмена сообщениями Крис Ричардсон

- Публикация событий (Transactional outbox)
- Опрашивающий издатель (Polling publisher)
- Механизм захвата изменений данных (Change Data Capture)
- Порождение событий (Event sourcing)

1 Transactional outbox

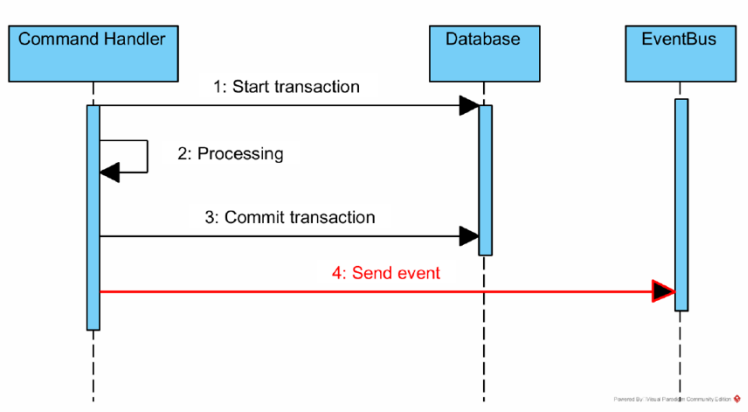
Публикация событий (Transactional outbox)

- У сервиса, отправляющего сообщения, есть таблица OUTBOX
- В рамках транзакции, которая создает, обновляет и удаляет бизнес-объекты, сервис шлет интеграционные сообщения, вставляя их в эту таблицу.
- Поскольку это локальная ACID-транзакция, атомарность гарантируется.



Публикация событий

- У сервиса, отправляющего сообщения, есть таблица OUTBOX
- В рамках транзакции сервис шлет сообщения, вставляя их в неё
- Поскольку это локальная ACID-транзакция, атомарность гарантируется



Transactional outbox

- Преимущества:

  Не требуются двухфазные коммиты

- Недостатки:

  Потенциально подвержен ошибкам, так как разработчик может забыть опубликовать сообщение/событие после обновления базы данных

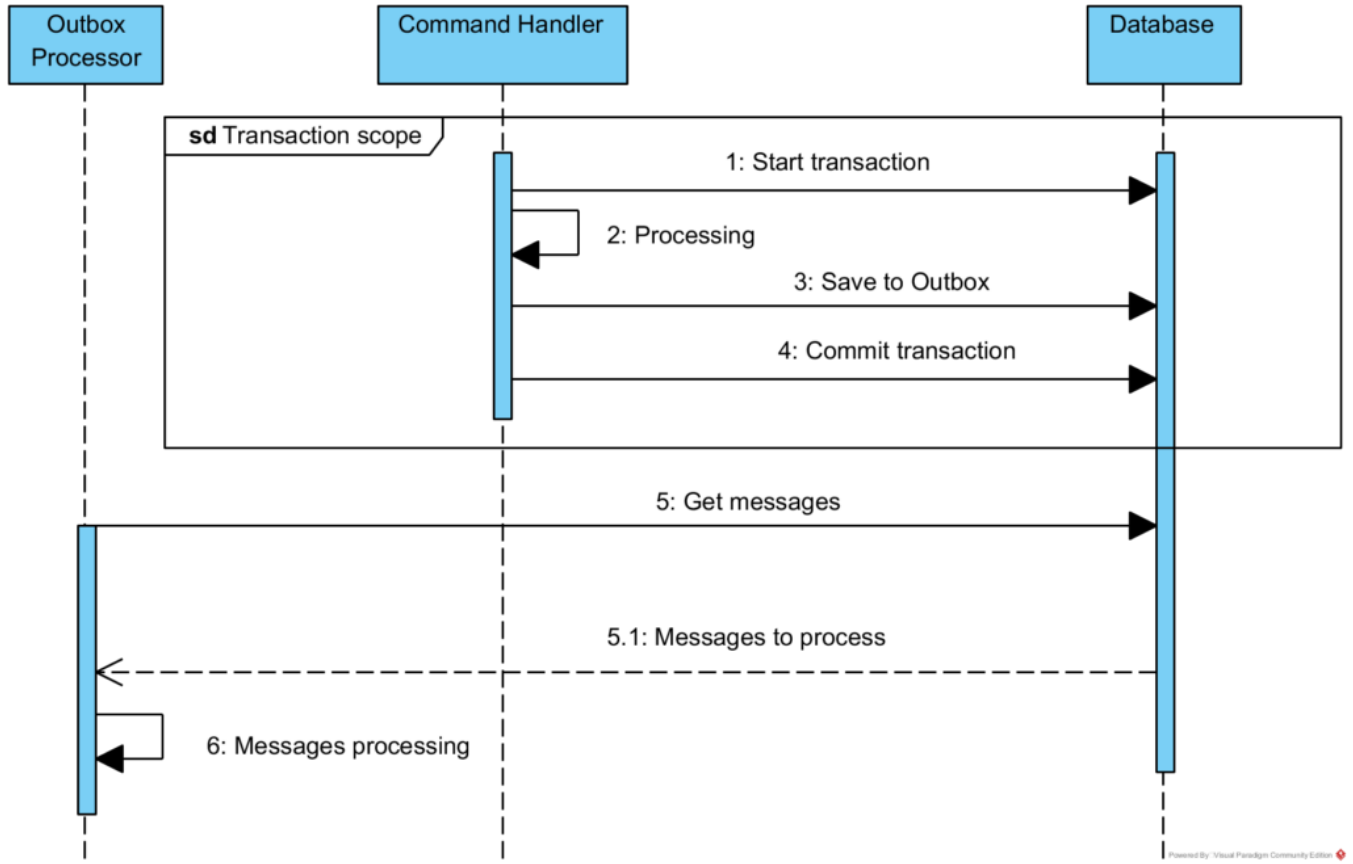
- Проблемы:

  Ретранслятор сообщений может публиковать сообщение более одного раза, в результате потребитель сообщения должен быть идемпотентным

2 Polling publisher

Опрашивающий издатель (Polling publisher)

- Периодически опрашиваем таблицу сообщений (OUTBOX) и публикуем новые сообщения
- После чего удаляем их из таблицы



Опрашивающий издатель

Преимущества:

- Работает с любой базой данных SQL

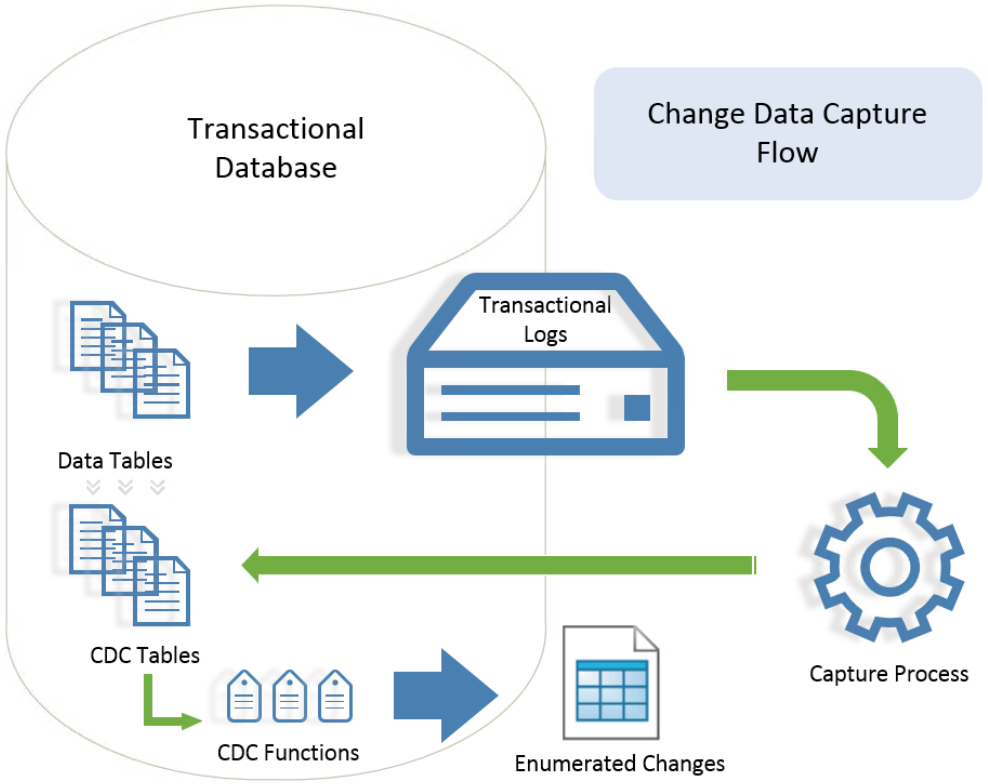
Недостатки:

- Сложно публиковать события попорядку
- Не все базы данных NoSQL поддерживают этот шаблон

3. Change Data Capture

Механизм захвата изменений

Принцип работы



https://i0.wp.com/knowledge-junction.com/wp-content/uploads/2018/02/Change-Data-Capture-CDC-Flow.png?w=985&ssl=1

CDC

- Плюсы
  - Не требует реализации бизнес-логики на уровне «транслятора»
  - Не нагружает базу данных
  - Гарантированно передает все изменения
- Минусы
  - Ориентирован на физическую модель хранения (увеличивается связанность)
  - Зависит от реализации баз данных

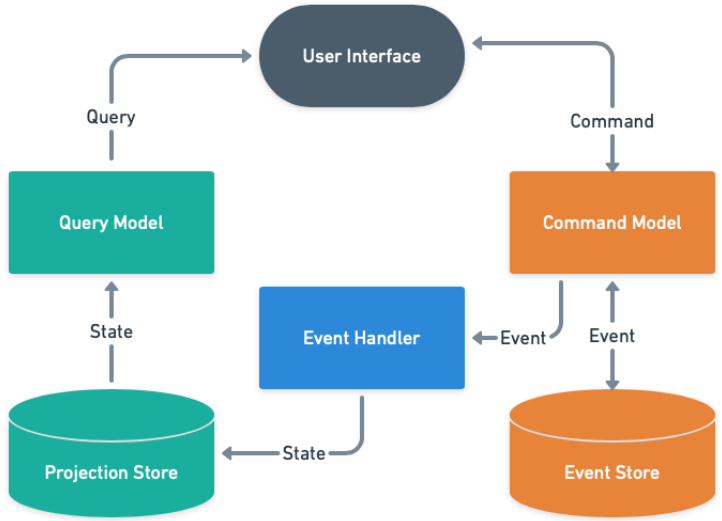
Пример https://debezium.io/

5 Event Sourcing

Отличие от предыдущих подходов (паттерн «Публикация событий») в том, что события хранятся постоянно, а не временно

Порождение событий

- События хранятся постоянно
- Состояние записи БД определяется последовательным применением событий, начиная с первого
- Возможно хранение промежуточных состояний (snapshots) и применение только части событий



Преимущества

- Простая модель хранения
- Упрощенное тестирование. Тесты в основном проверяют, что именно произошло с данной сущностью. Обычно мы проверяем, что произошло, опуская проверку того, что не произошло.
- Хранит журнал того, что произошло в системе
- Позволяет "восстановить" состояние объекта к заданному моменту времени
- Требует CQRS, поэтому довольно сложна в реализации

Литература

- Fundamentals of Software Architecture by Mark Richards, Neal Ford, Publisher: O'Reilly Media, Inc. Release Date: January 2020 ISBN: 9781492043454
- Designing Distributed Systems: Patterns and Paradigms for Scalable, Reliable Services, Brendan Burns, Publisher: O'Reilly Media, Inc. Release Date: March 2018

Что почитать

На сегодня все

ddzuba@yandex.ru