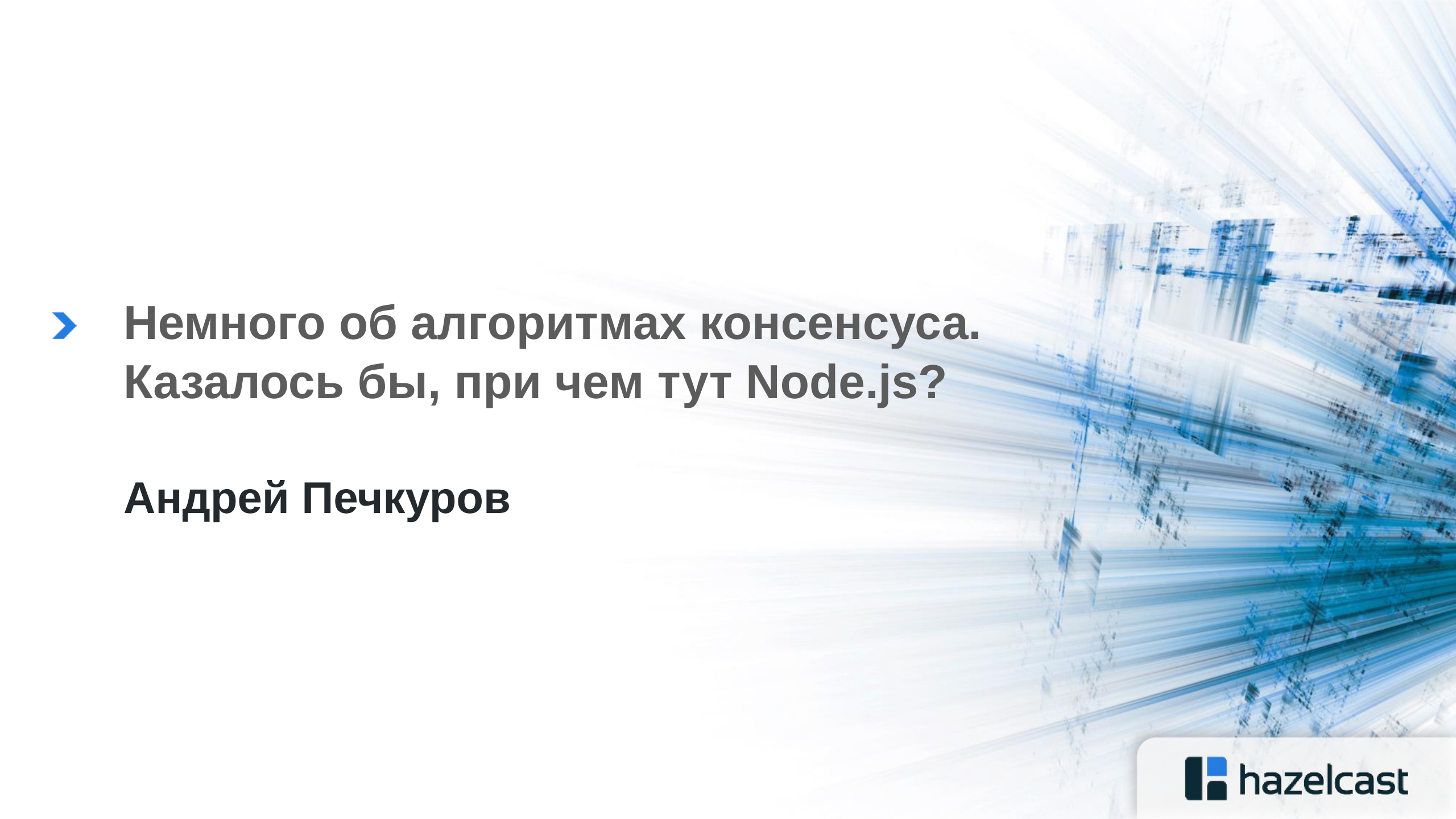
Немного об алгоритмах консенсуса. Казалось бы, при чем тут Node.js?

Андрей Печкуров



О докладчике

- Пишу на Java (очень долго), Node.js (долго)
- Node.js core collaborator
- Интересы: веб, архитектура, распределенные системы, производительность
- Можно найти тут:
  - https://twitter.com/AndreyPechkurov
  - https://github.com/puzpuzpuz
  - https://medium.com/@apechkurov

- Hazelcast In-Memory Data Grid (IMDG)
- Большой набор распределенных структур данных
- Показательный пример - `Map`, который часто используют как кэш
- Написана на Java, умеет embedded и standalone режимы
- Хорошо масштабируется вертикально и горизонтально
- Часто используется в high-load и low-latency приложениях
- Области применения: IoT, in-memory stream processing, payment processing, fraud detection и т.д.

- Hazelcast In-Memory Data Grid (IMDG)
- Хотите production-ready Raft? У нас есть CP Subsystem (с Jepsen тестами и lock'ами 🙂)
- https://docs.hazelcast.org/docs/4.0.1/manual/html-single/index.html#cp-subsystem

Hazelcast IMDG Node.js client

- https://github.com/hazelcast/hazelcast-nodejs-client
- Поддерживает CP Subsystem
- P.S. А вот доклад про историю оптимизаций: https://youtu.be/CSnmpbZsVD4

Делай раз

```
const lock = await client.getCPSubsystem().getLock('my-lock');

const fence = await lock.lock();
try {
    // "защищенный" код
    // (доступен только владельцу)
} finally {
    await lock.unlock(fence);
}
```

Делай два

```
const ref = await client.getCPSubsystem().getAtomicReference('my-ref');

await ref.set({ foo: 'bar' });

// атомарно заменяем значение
// (только если оно совпадает с ожидаемым)
const result = await ref.compareAndSet(
  { foo: 'bar' },
  { bar: 'baz' }
);
```

Делай три

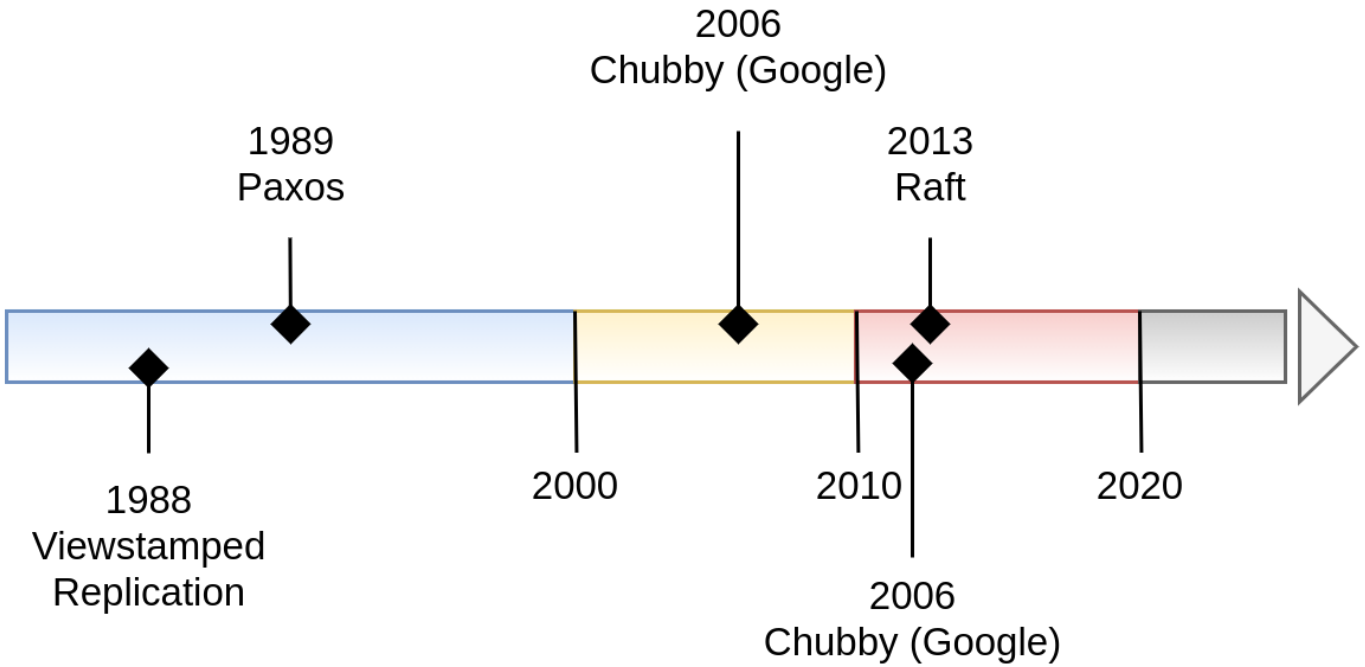
```
const counter = await client.getCPSubsystem().getAtomicLong('my-counter');

// атомарно увеличиваем счетчик
const knownValue = await counter.incrementAndGet();

// атомарно заменяем значение
// (только если оно совпадает с ожидаемым)
const result = await counter.compareAndSet(knownValue, 42);
```

План на сегодня

- Начинаем пугаться распределенных систем
- Знакомимся с видами согласованности (consistency)
- CAP теорема и прочие классификации
- Что за зверь - алгоритм консенсуса?
- История: Paxos и его подвиды, Raft
- CASPaxos, как один из недавних Paxos-образных
- Pet project: CASPaxos на Node.js

- **Начинаем пугаться распределенных систем**

Упрощенная до ужаса история



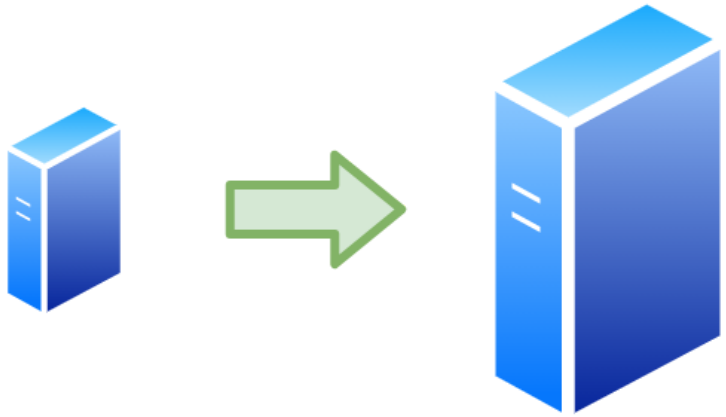
- Эра централизованных БД
- Становление распределенных систем
- Бум распределенных систем
- Будущее?

Распределенная система

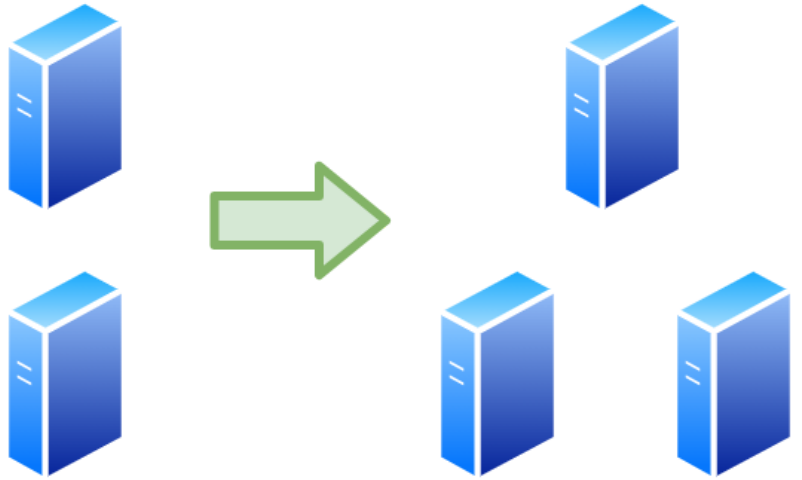
- Назовем распределенной систему, хранящую состояние (общее) на нескольких машинах, соединенных сетью
- Для определенности будем подразумевать хранилище пар ключ-значение

Принципиальные отличия

Централизованная система



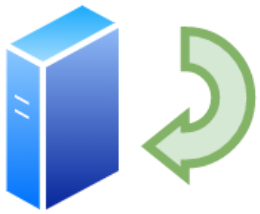
Распределенная система



Еще принципиальные отличия

Централизованная система

```
const data = await readData();
```



Распределенная система

```
const data = await readData();
```

А еще?

Централизованная система

$P(S)$

Распределенная система

$P(S1 + S2) = P(S1) + P(S2) – P(S1 \times S2)$

Fallacies of distributed computing

- Инженеры из Sun (R.I.P.) еще в 94м сформулировали такой список:
  - **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
- P.S. Добавим сюда "**Clocks are in sync**"

Сеть 🌐

- Мы работаем с асинхронными сетями
- Отправленный запрос может:
  - Быть потерян при отправке туда/обратно
  - Находиться в очереди ожидания отправки (если сеть под нагрузкой)
  - Быть получен, но машина-адресат дала сбой до или во время обработки
- Единственный способ подтверждения - получить ответ

Локальные 🕐

- Каждая машина работает с локальным временем (`Date.now()`)
- Локальные монотонные часы (`process.hrtime`) помогают, но далеко не всегда
- В действительности нам хотелось бы иметь глобальные часы

Глобальные 🕐

- Увы, глобальные (синхронизированные) часы невозможны без специального железа
- Байка: в Google Spanner часы в ЦОД синхронизированы в пределах 7 мс (атомные часы + GPS)
- Оффтопик: векторные "часы" частично решают проблему

Чего мы ждем от распределенной системы? 🤔

Мой личный список

1. Отказоустойчивость
2. Масштабируемость
3. Удобство поддержки (мониторинг, администрирование)

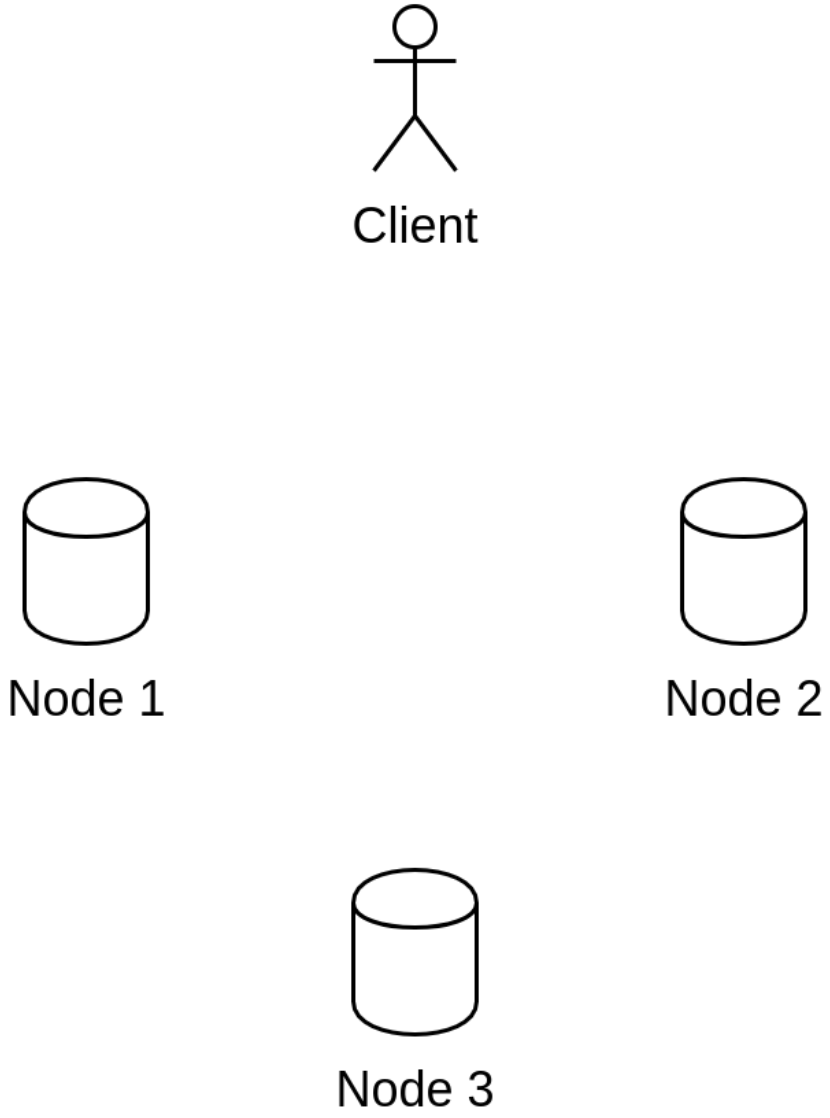
Чего мы еще ждем?

- Допустим, что мы ждем того же поведения, что и у централизованного хранилища
- А именно - с клиентской стороны поведение должно быть, как если бы это была централизованная система (пока остановимся на этой формулировке)

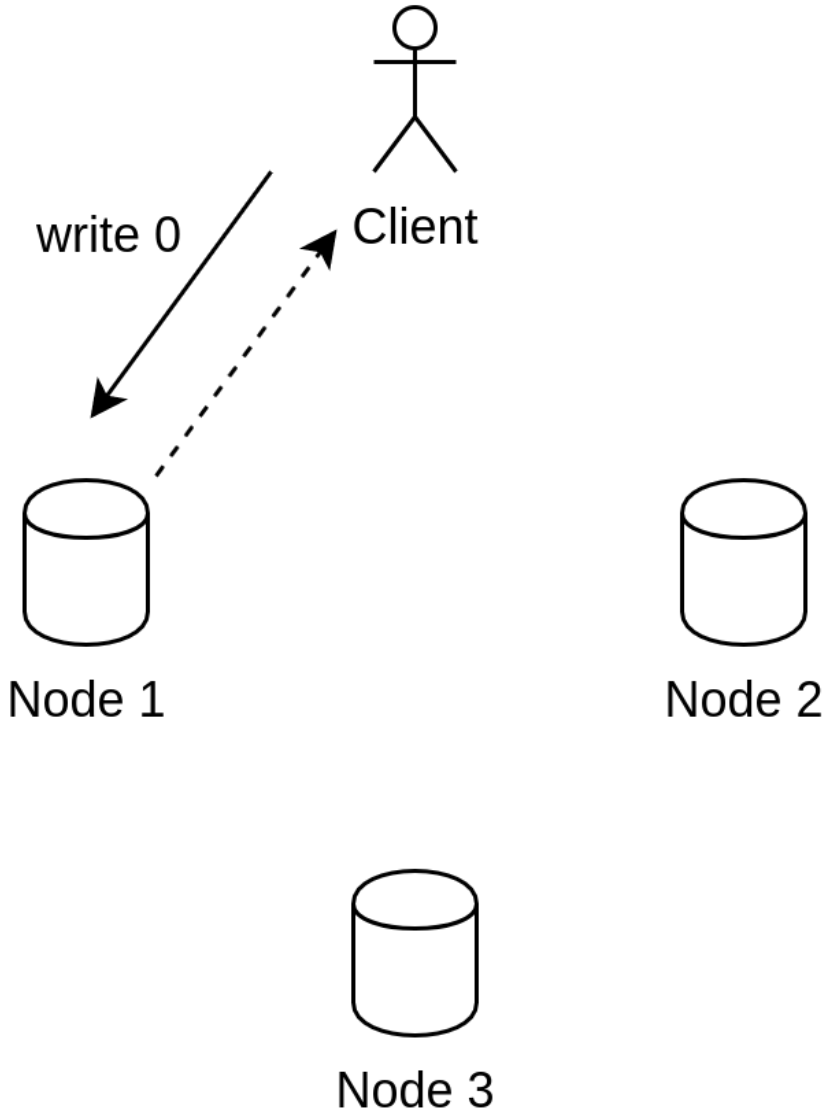
Фигня вопрос - сейчас придумаем алгоритм

1. Любой узел принимает клиентские запросы (прочитать/записать)
2. Затем - отправляет операцию на все остальные узлы
3. Ждет ответов от большинства (консенсус жеж 😄)
4. Дождавшись консенсуса, отправляет клиенту сообщение об успехе

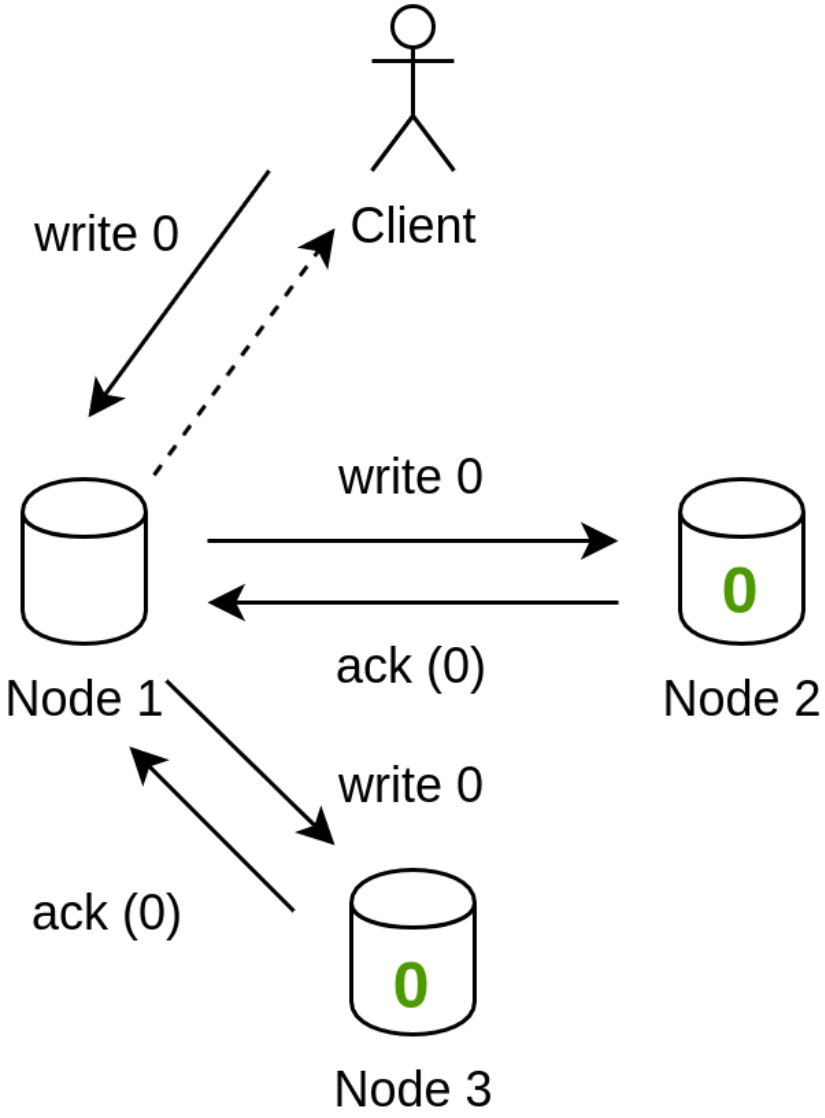
Что не так с нашим изобретением?



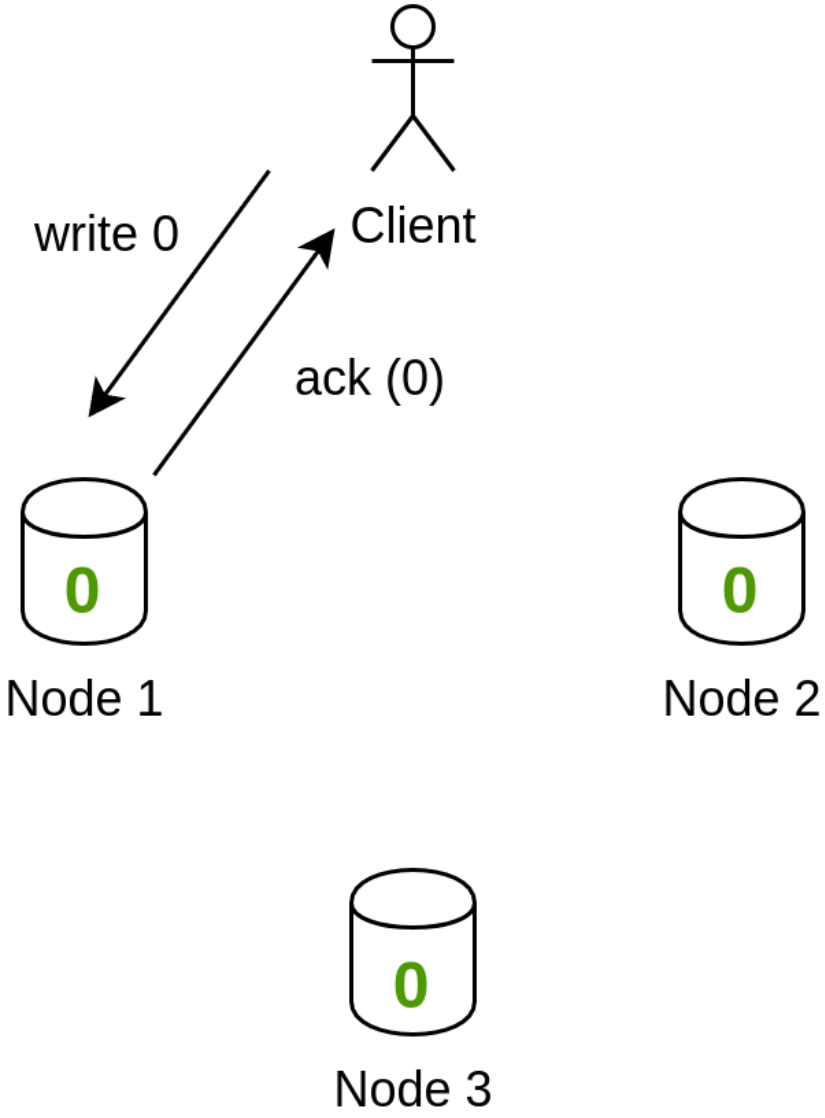
Что не так с нашим изобретением?



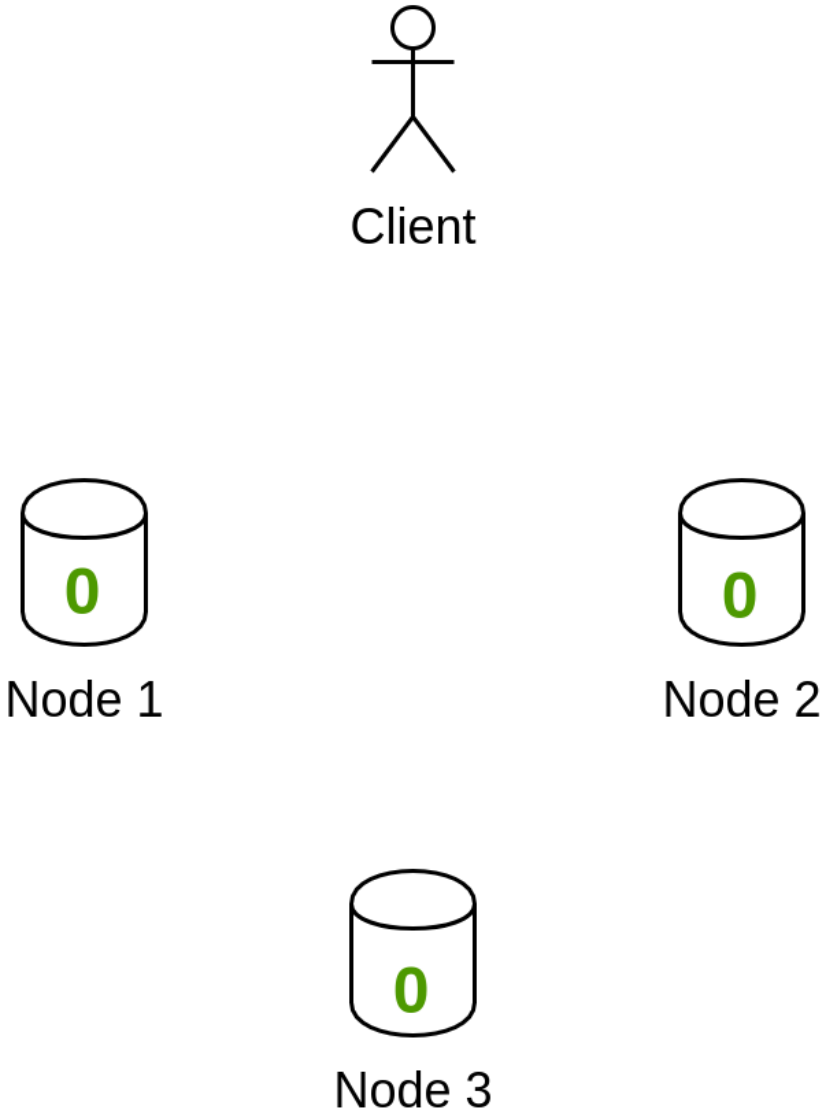
Что не так с нашим изобретением?



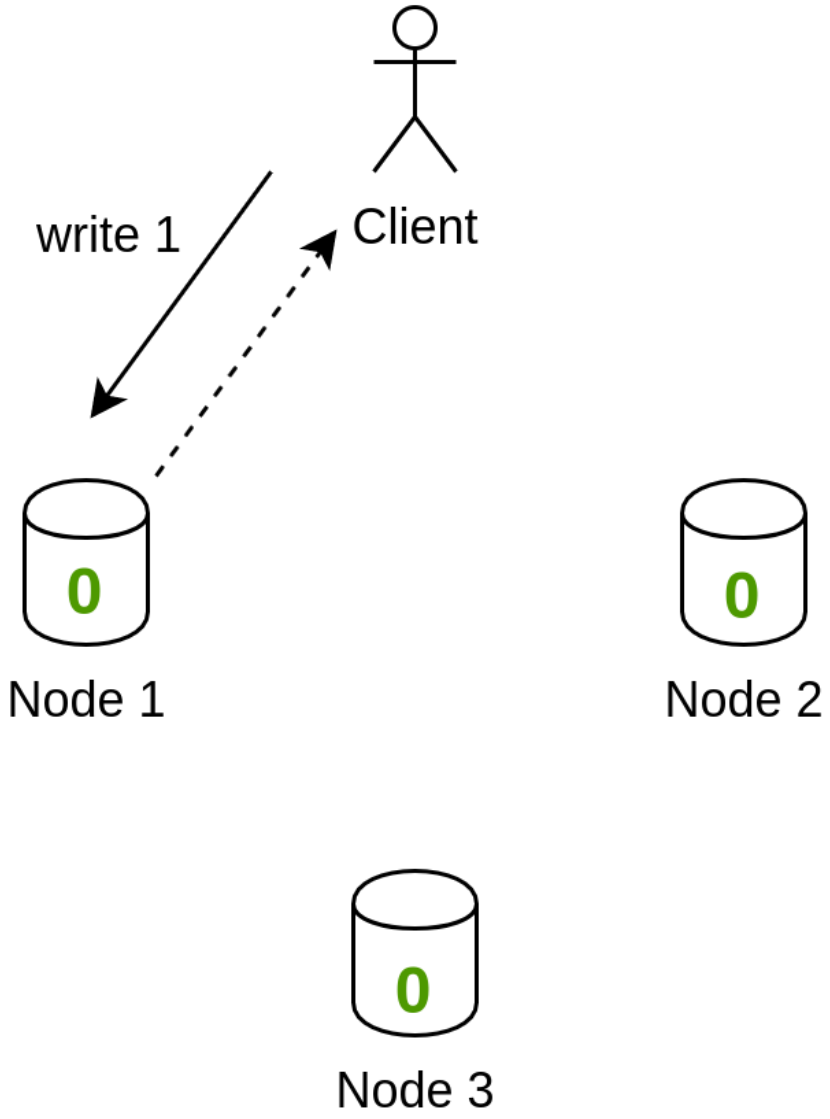
Что не так с нашим изобретением?



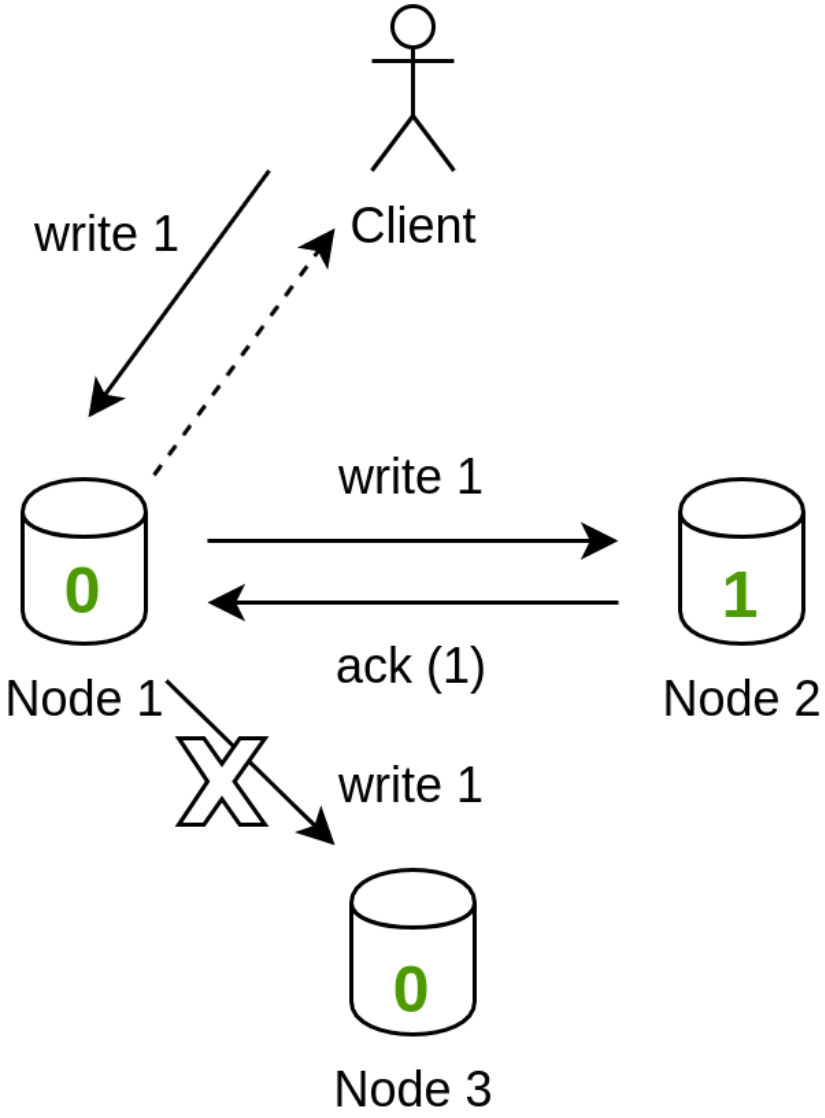
Что не так с нашим изобретением?



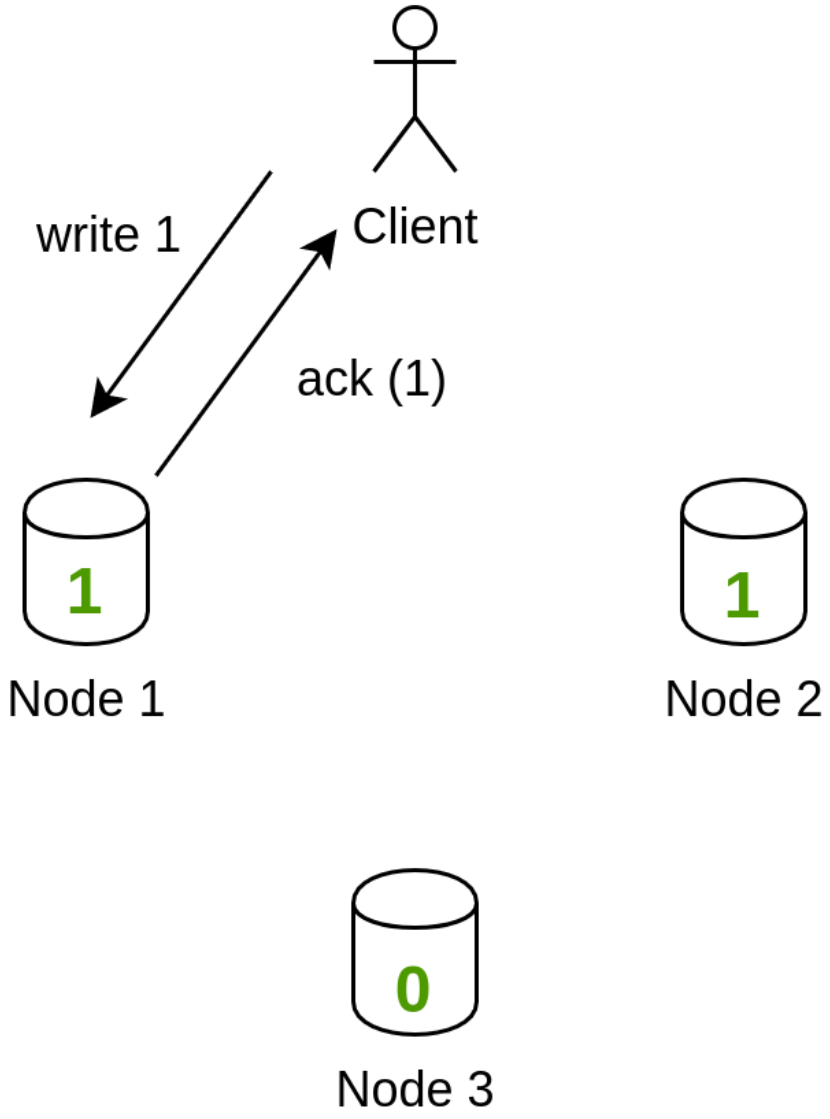
Что не так с нашим изобретением?



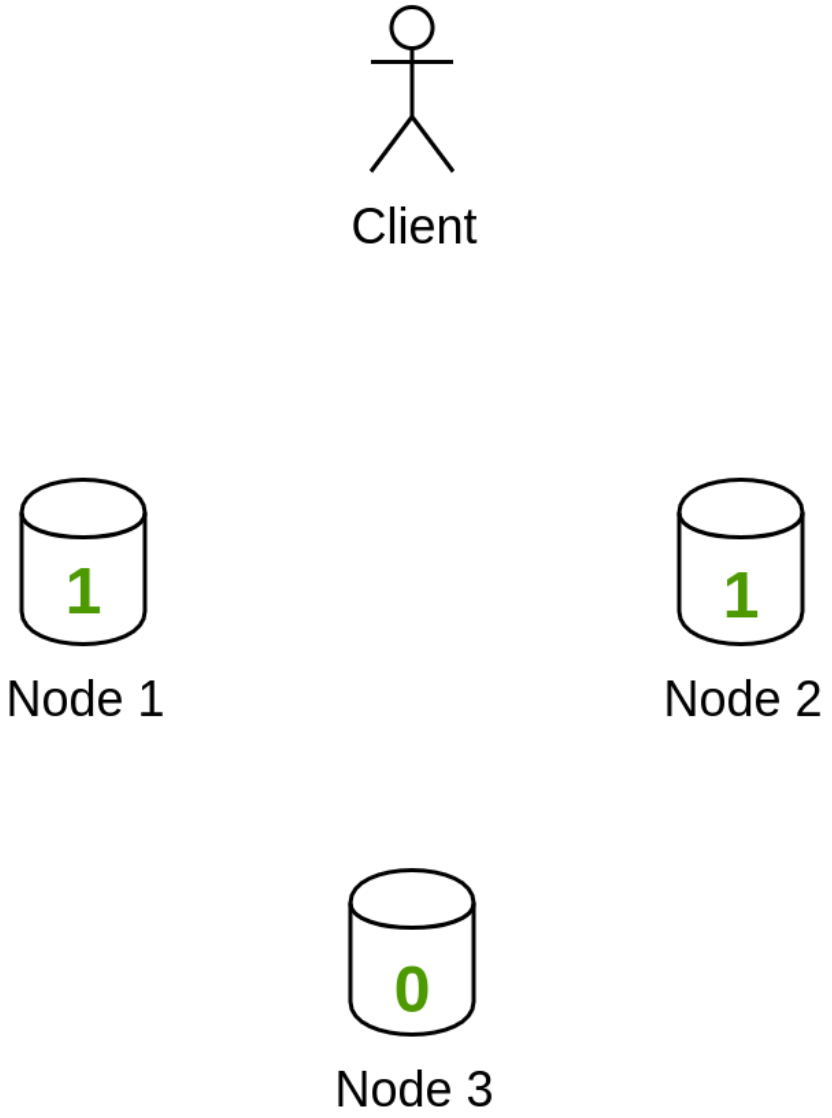
Что не так с нашим изобретением?



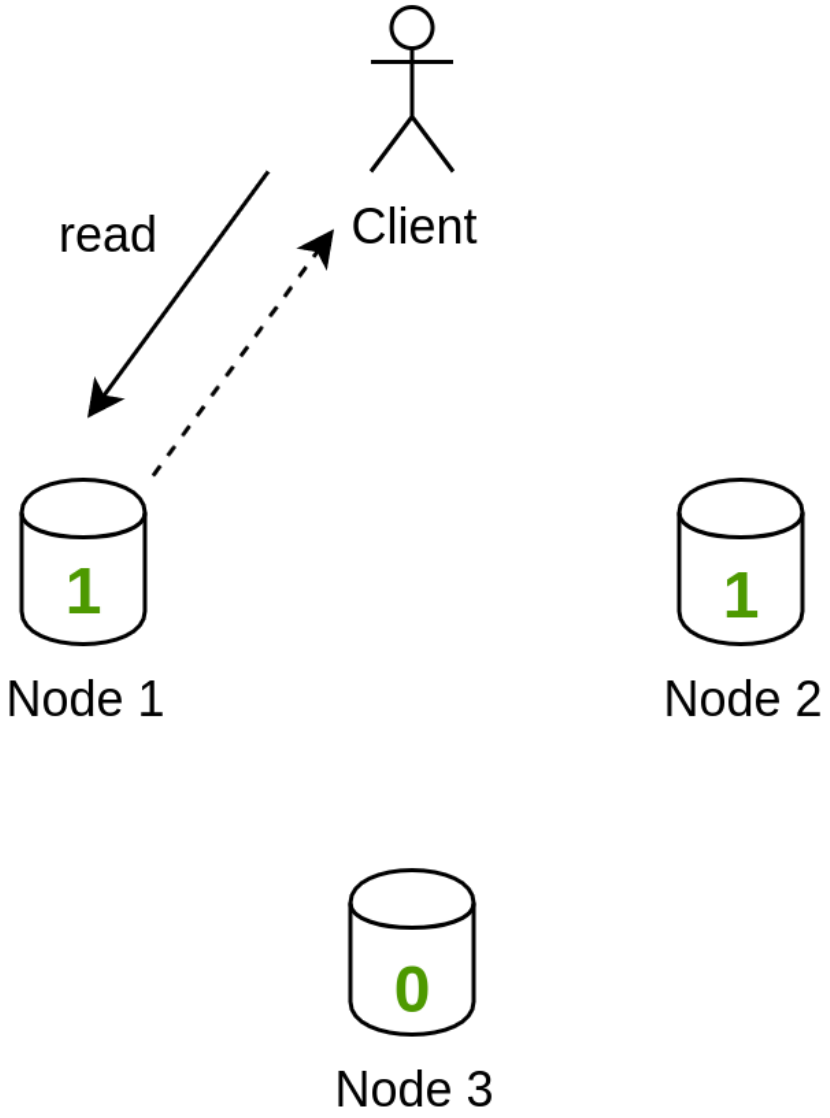
Что не так с нашим изобретением?



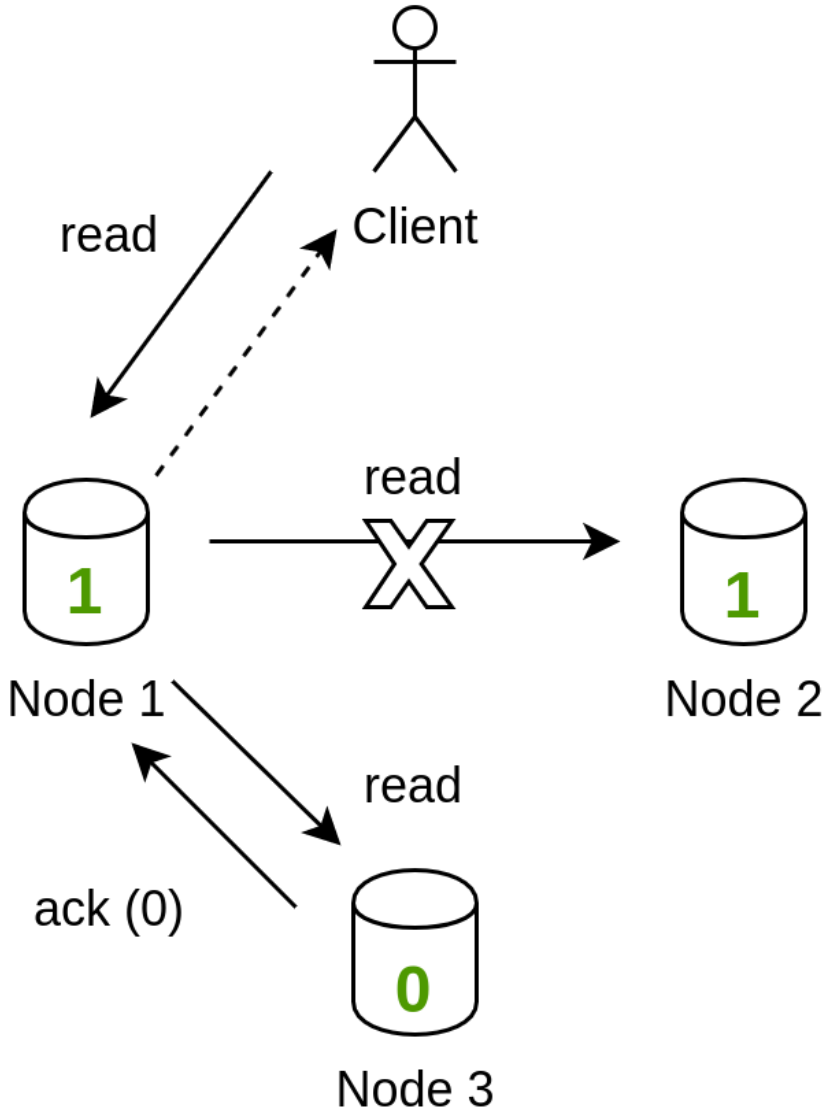
Что не так с нашим изобретением?



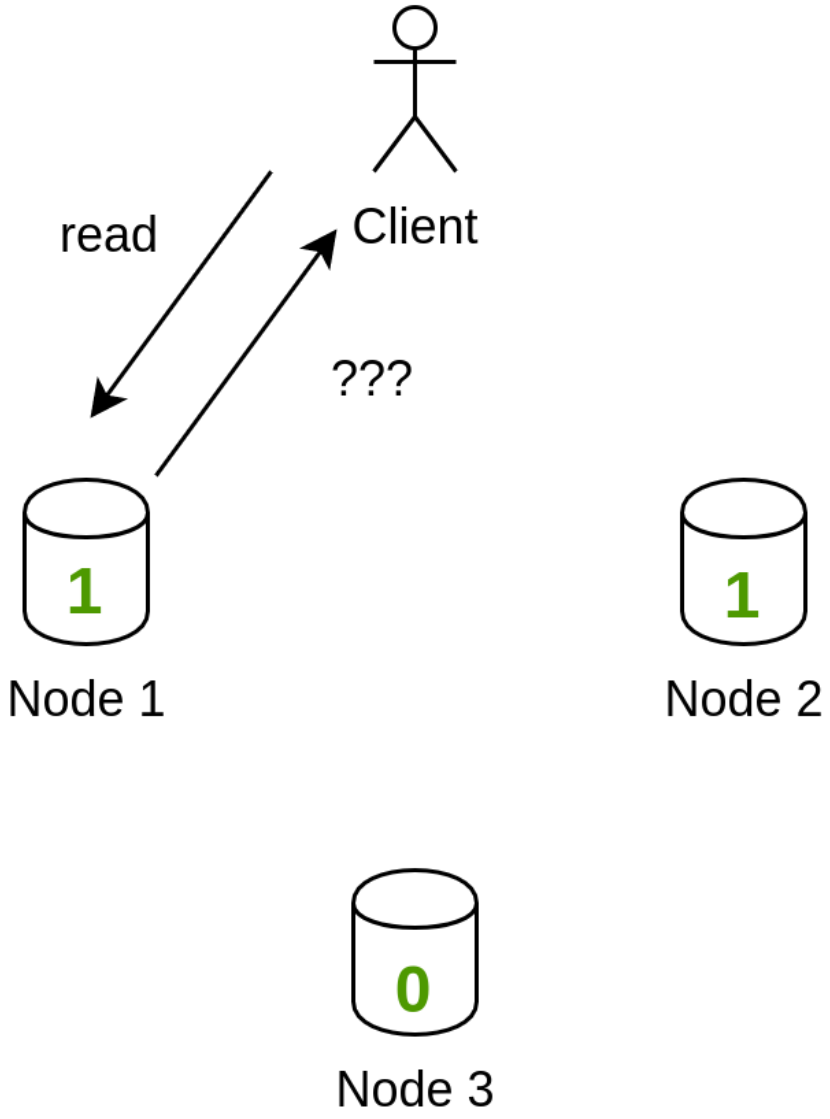
Что не так с нашим изобретением?



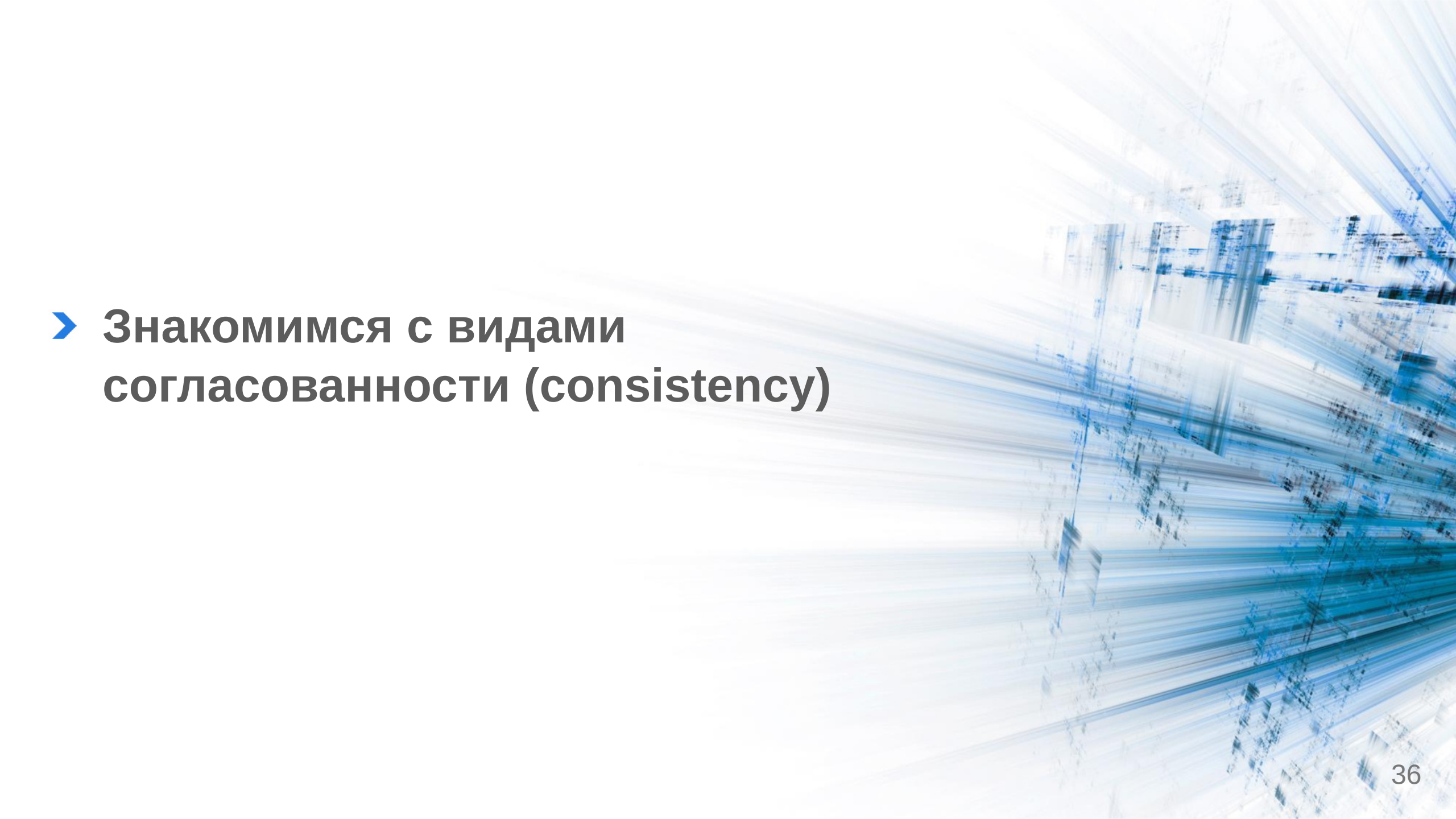
Что не так с нашим изобретением?



Что не так с нашим изобретением?



- **Знакомимся с видами согласованности (consistency)**



Неформальное определение

Модель согласованности (consistency model) - это гарантии, которые система (внезапно, не только распределенная) предоставляет относительно набора поддерживаемых операций

Eventual consistency

Система гарантирует, что данные будут доступны на всех узлах через какое-то (неопределенное) время после завершения операций записи

Monotonic reads

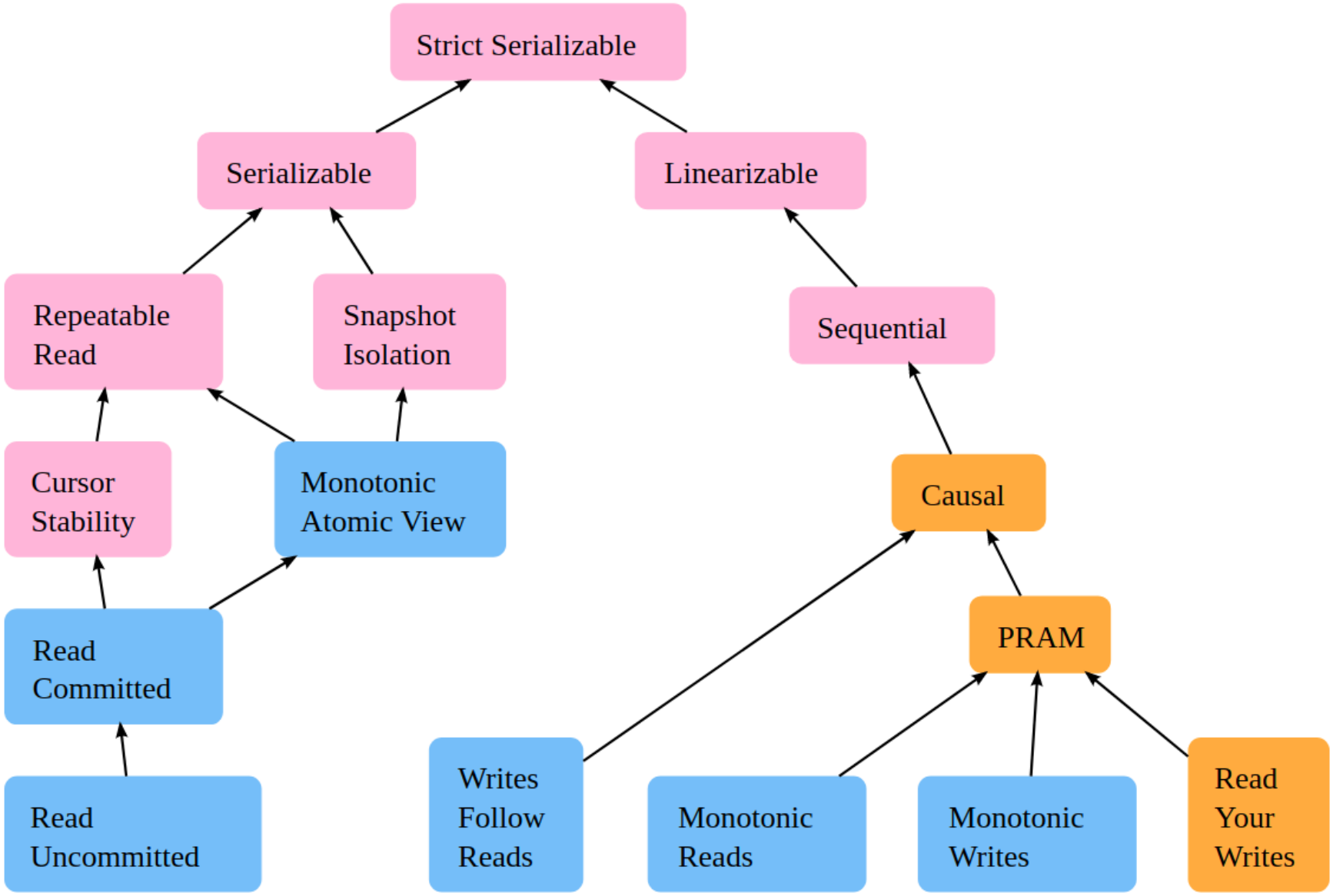
Система гарантирует, что если какой-то (конкретный) клиент читает запись, последовательные чтения той же записи вернут то же самое, или более познее значение

Linearizability

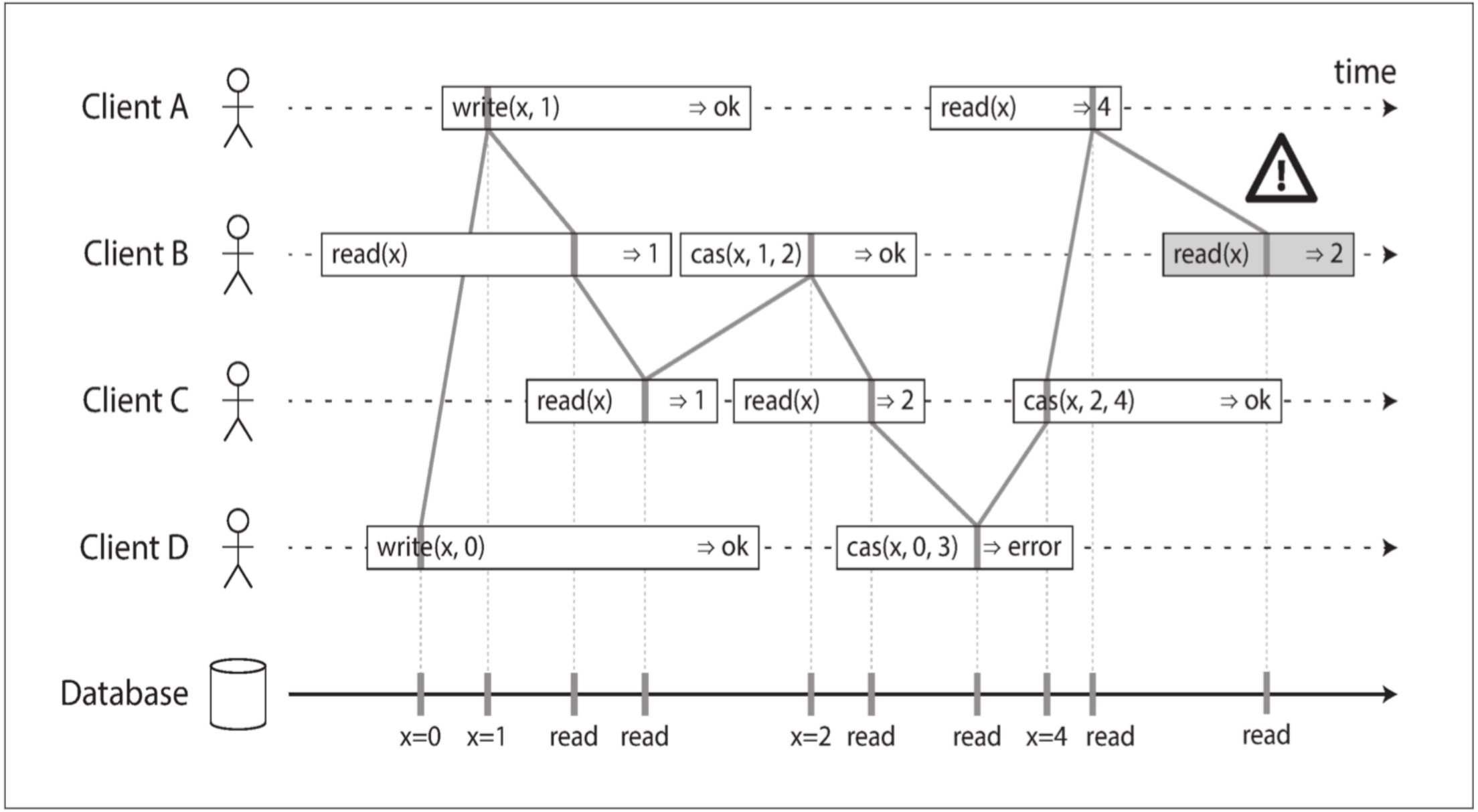
- Одна из наиболее строгих (сильных) моделей согласованности для **одного** объекта
- Именно ее мы и подразумевали (надеюсь) ранее:
  "с клиентской стороны поведение должно быть, как если бы это была централизованная система"

Неформальное определение linearizability

- Система гарантирует, что каждая операция выполняется атомарно, в **некотором** (общем для всех клиентов) порядке, не противоречащим порядку выполнения операций в реальном времени
- Т.е. если операция A завершается до начала операции B, то B должна учитывать результат выполнения операции A

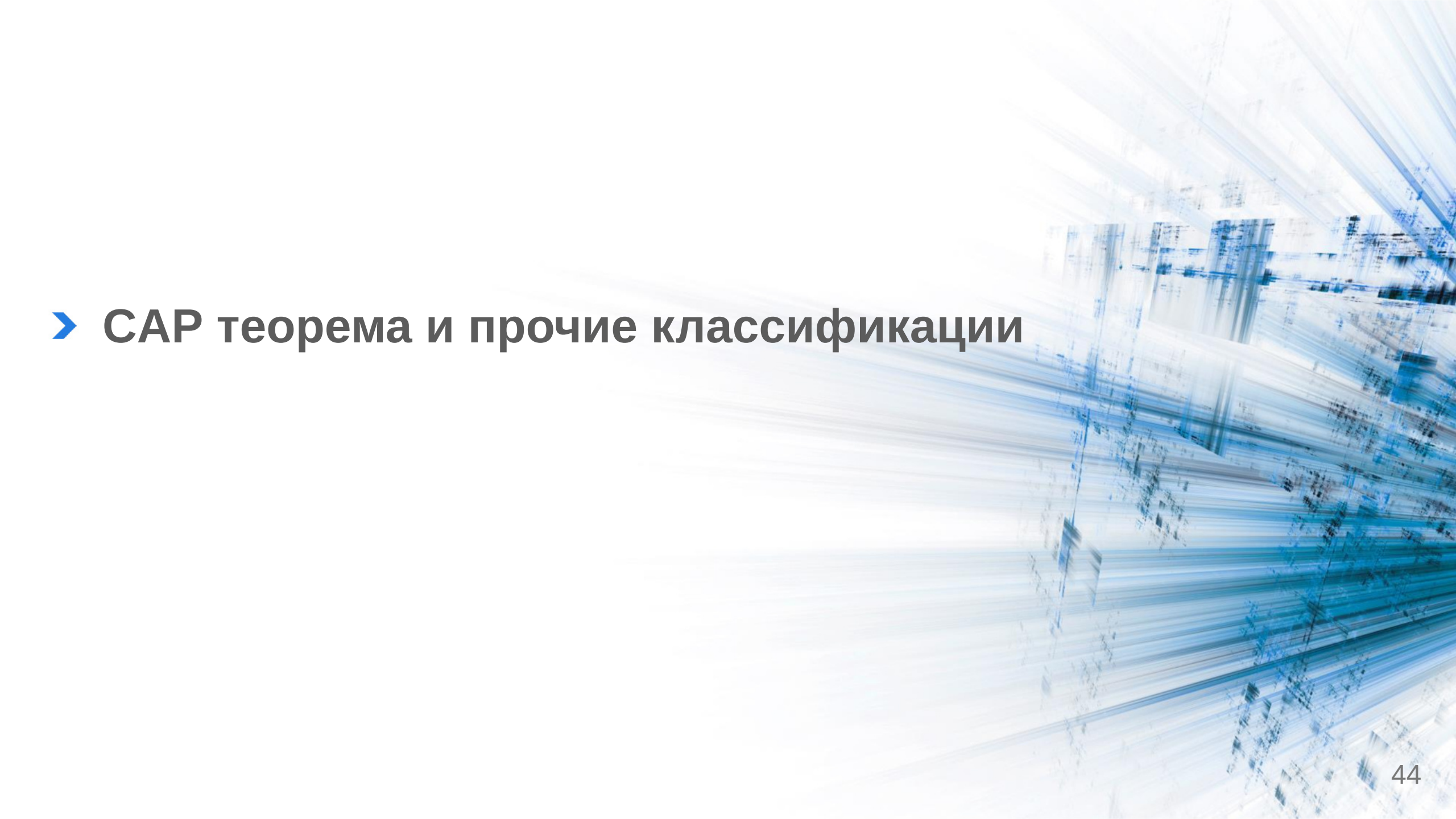


Source: https://jepsen.io/consistency



Source: DDIA book

- CAP теорема и прочие классификации



CAP теорема

- Сформулирована Eric Brewer в 1998, как утверждение
- В 2002 появилось формальное доказательство
- Рассматривается система с одним регистром
- CAP:
  - Consistency: здесь подразумевается линеаризуемость
  - Availability: каждый запрос, полученный нормально функционирующим узлом системы, должен приводить к ожидаемому ответу (не к ошибке)
  - Partition tolerance: подразумевает коммуникацию через асинхронную сеть

Partition tolerance

Network partition - сценарий, когда узлы продолжают функционировать, но некоторые из них не могут общаться между собой

Неформальное определение

В условиях network partition рассматриваемая система может быть:

- Доступна (AP)
- Согласована (CP)

P.S. CA опции в CAP теореме нет и в помине

Критика

- Однобокая классификация, которая почему-то прижилась
- Например, РСУБД с одной read-only репликой не является ни CP, ни AP
- Теорема ничего не говорит о времени отклика системы, т.е. AP система может отвечать сколь угодно медленно
- Наконец, network partition - далеко не единственный сценарий отказа

Альтернативы

- PACELC теорема - расширение CAP теоремы (Daniel J. Abadi, 2010)
- PAC = PA | PC (та же CAP теорема)
- ELC = EL | EC:
  - E: else
  - L: latency
  - C: consistency

- Что за зверь - алгоритм консенсуса?

Неформальное определение

- Алгоритм консенсуса - алгоритм, позволяющий узлам системы достигнуть консенсус, т.е. принять совместное решение **о том или ином** действии
- Под действием понимают **однократное** изменение регистра (например, в Paxos)
- С точки зрения отказов рассматривают только non-Byzantine faults
- P.S. А еще есть FLP result (1985), доказывающий невозможность гарантированного достижения консенсуса за ограниченное время в асинхронных системах (🤯)

Связь с linearizability

- На практике консенсус нужен на **последовательности** действий
- Поэтому для реализаций встречается определение atomic broadcast (total order broadcast) - консенсус тут нужен для принятия решения о следующем действии
- Можно показать, что linearizability и total order broadcast можно свести друг к другу
- А значит, мы говорим о CP системах, с точки зрения пресловутой CAP теоремы

Краткий список алгоритмов консенсуса

- Viewstamped Replication - 1988
- Paxos и его подвиды (Chubby, Spanner) - 1989
- Raft (Consul, etcd, Hazelcast IMDG, RethinkDB, TiKV, etc.) - 2013
- ZAB (ZooKeeper)
- И другие специфичные для конкретного продукта алгоритмы

Верификация корректности (неполный список вариантов)

- TLA+ (L.Lamport) - теория
- Jepsen (K.Kingsbury) - практика

Применение алгоритмов консенсуса

- Linearizable key-value storage
- Distributed locks*
- Leader election
- Constraints (e.g. uniqueness)
- Atomic commit (distributed transactions)

Темная сторона алгоритмов консенсуса

- Производительность напрямую зависит от задержек сети
- Высокая сложность реализации и верификации
- Для работы кластера требуется строгое большинство узлов (что логично)

- История: Paxos и его подвиды, Raft

Vanilla Paxos

- Paxos (1998) - p2p (leaderless)
- В основе - Synod/Single-Decree Paxos
- Позволяет принять строго одно решение (о значении регистра)
- Часто под Paxos подразумевают семейство алгоритмов

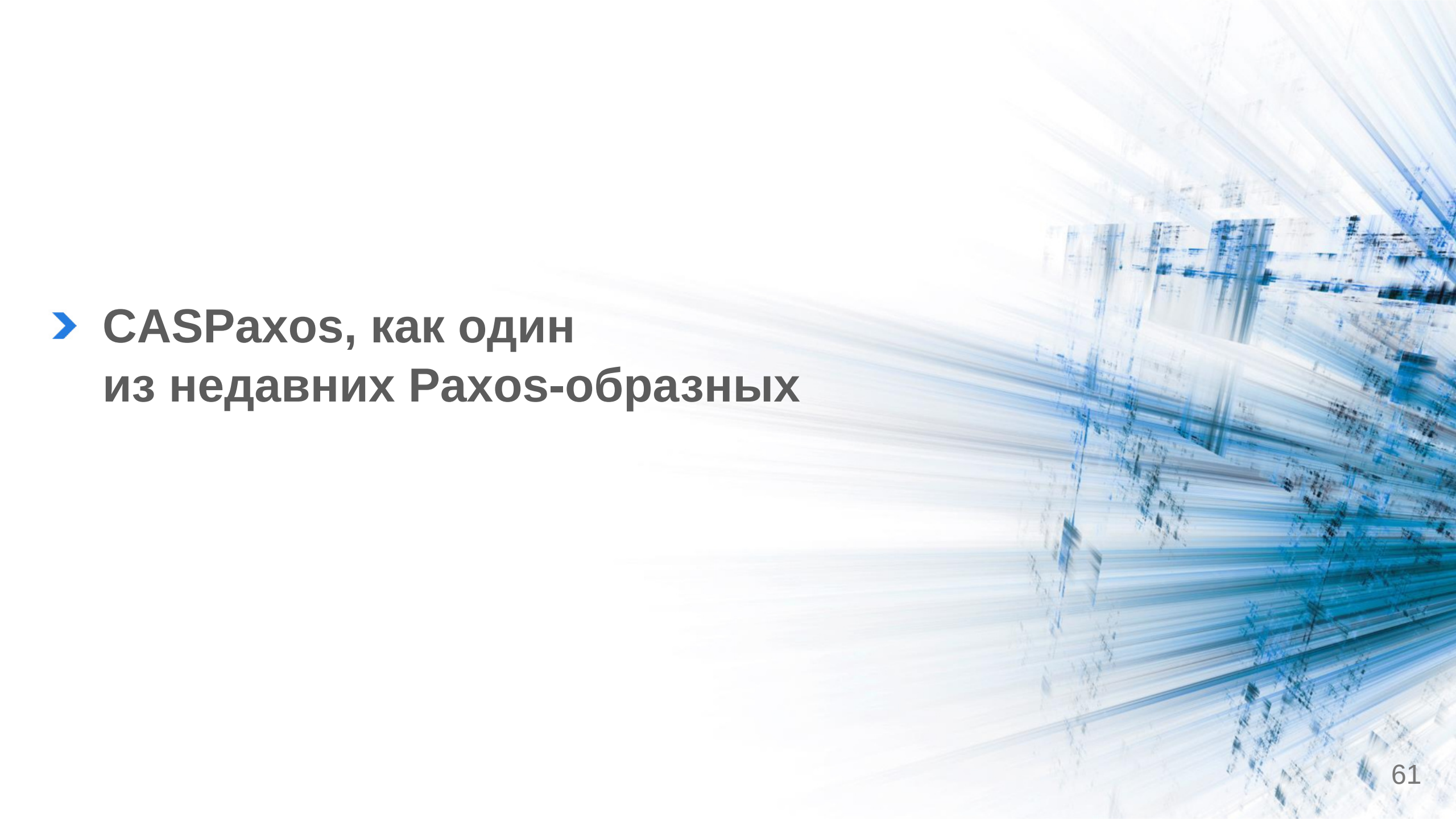
Подвиды Paxos

- Multipaxos (2001)
- FastPaxos (2004~2005)
- Generalized Paxos (2004~2005)
- Mencius (2008)
- Multicoordinated Paxos (2006)
- Vertical Paxos (2009)
- Ring Paxos (2010) / Multi-Ring Paxos (2010)
- SPaxos (2012) / EPaxos (2013) / FPaxos (2016) / KPaxos (2017) / WPaxos (2017)
- **CASPaxos** (2018)
- SDPaxos (2018)

Raft

- Raft (2013) - single leader, replicated log
- Время делится на периоды (term), каждый из которых начинается в выбора лидера
- Упор сделан на простоту понимания алгоритма
- https://raft.github.io

- **CASPaxos, как один из недавних Paxos-образных**



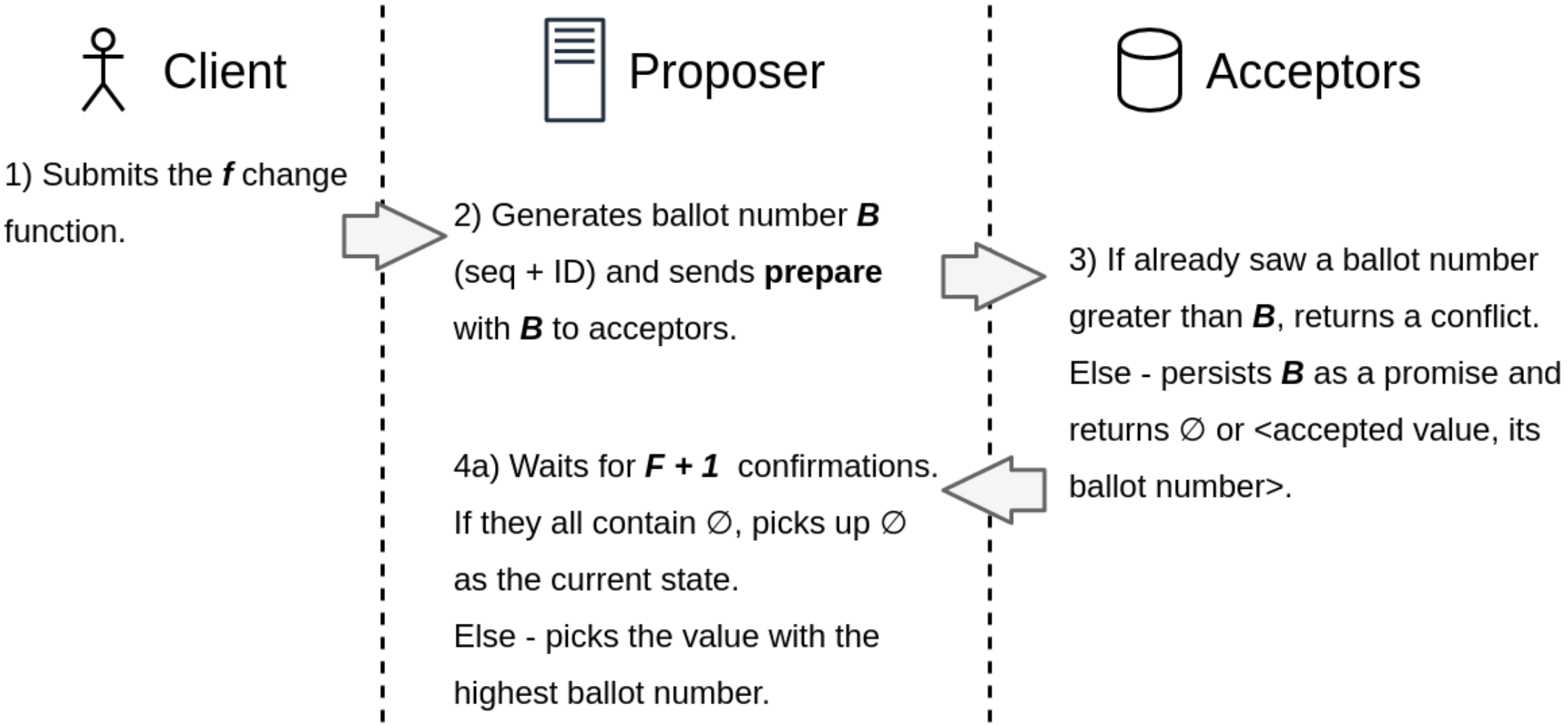
CASPaxos

- CASPaxos (D.Rystsov, 2018) - p2p, replicated state
- CAS - compare-and-set/compare-and-swap
- Модифицирует Paxos (Synod), а не просто использует его как компонент для построения системы

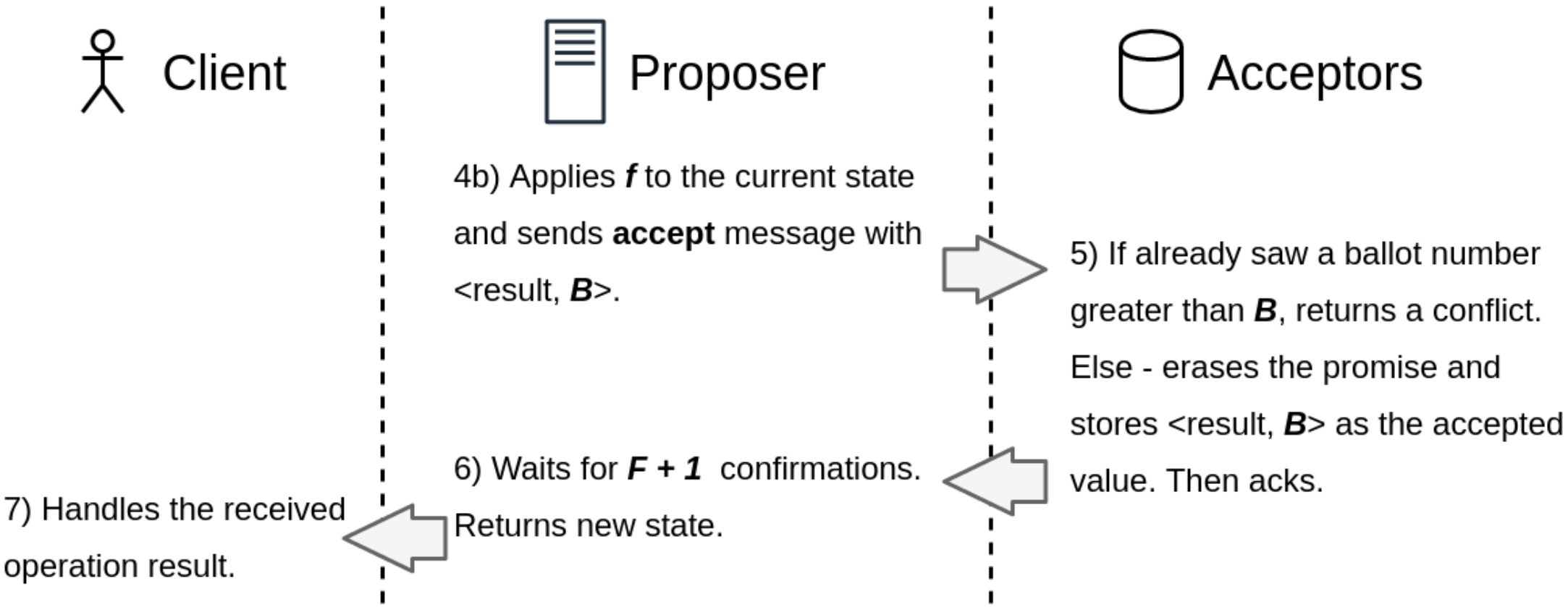
Основы CASPaxos

- Использует один регистр (объект)
- Вводит несколько ролей для процессов:
  - Clients: клиенты системы, отправляют запросы к Proposer
  - Proposers: принимают запросы клиентов, генерируют уникальные номера (proposal ID) и общаются с Acceptors
  - Acceptors: могут принимать предложения Proposers, хранят принятое состояние
- Для сохранения работоспособности системы до `F` отказов, нужны `2F + 1` Acceptors

Phase 1


Client
Proposer
Acceptors
1) Submits the f change function.
2) Generates ballot number B (seq + ID) and sends prepare with B to acceptors.
3) If already saw a ballot number greater than B, returns a conflict. Else - persists B as a promise and returns ∅ or <accepted value, its ballot number>.
4a) Waits for F + 1 confirmations. If they all contain ∅, picks up ∅ as the current state. Else - picks the value with the highest ballot number.

Phase 2


Client
Proposer
Acceptors
4b) Applies *f* to the current state and sends **accept** message with <result, *B*>.
5) If already saw a ballot number greater than *B*, returns a conflict. Else - erases the promise and stores <result, *B*> as the accepted value. Then acks.
6) Waits for *F + 1* confirmations. Returns new state.
7) Handles the received operation result.

Связь с Synod

CASPaxos эквивалентен Synod, если взять функцию:

```
x -> if x = ∅ then val0 else x
```

CAS регистр на основе CASPaxos

Чтение:

```
x -> x
```

Инициализация значением `val0`:

```
x -> if x = ∅ then (0, val0) else x
```

Запись значения `val1` при условии версии `3`:

```
x -> if x = (3, *) then (4, val1) else x
```

CASPaxos-based key-value storage

- Хранилище - это набор именованных экземпляров CASPaxos, по одному на ключ
- Преимущества и недостатки подхода в сравнении с тем же Raft выходят за рамки доклада

Pet project: CASPaxos на Node.js

https://github.com/gryadka/js (Node.js - proposers, Redis - acceptors)

||

\/

https://github.com/puzpuzpuz/ogorod (Node.js - proposers & acceptors)

Ogorod TODOs

- Add applicable *Paxos optimizations, like CASPaxos 1RTT optimization
- Dynamic configuration support, including proposer configuration versioning and persistence
- Implement delete operation support. Requires a GC-like background process (see the CASPaxos paper for more details)
- Use an embedded key-value storage, like RocksDB, for data persistence
- Swap HTTP with plain TCP communication for the internal API
- Implement proper error handling and logging
- And many-many more

Call to Action

- Распределенных систем бояться - на server-side не ходить
- Все, кому интересны высокопроизводительные библиотеки (и распределенные системы) - welcome
- https://github.com/hazelcast/hazelcast-nodejs-client
- P.S. Contributions are welcome as well

Спасибо за внимание!

Полезные книги и ссылки

- Designing Data-Intensive Applications, Martin Kleppmann, 2017
- CASPaxos: Replicated State Machines without logs, Denis Rystsov, 2018 - https://arxiv.org/abs/1802.07000
- Paxos Made Simple, Leslie Lamport, 2001 - https://lamport.azurewebsites.net/pubs/paxos-simple.pdf
- https://raft.github.io/raft.pdf
- https://jepsen.io
- https://martin.kleppmann.com/2015/05/11/please-stop-calling-databases-cp-or-ap.html
- https://vadosware.io/post/paxosmon-gotta-concensus-them-all/