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 IMDG

- Hazelcast In-Memory Data Grid (IMDG)
- Большой набор распределенных структур данных
- Показательный пример Мар, 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 IMDG Node.js client

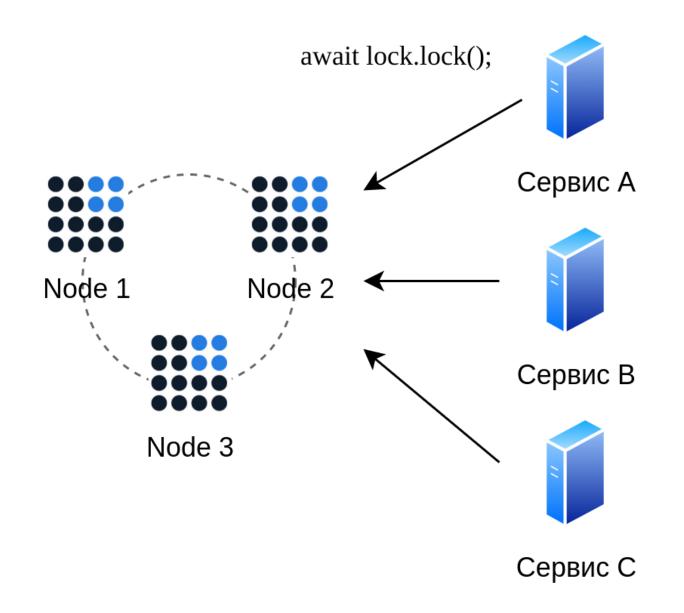
- https://github.com/hazelcast/hazelcast-nodejs-client
- P.S. А вот доклад про историю оптимизаций: https://youtu.be/CSnmpbZsVD4

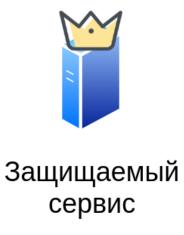
#### Старая реализация распределенных lock'ов

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

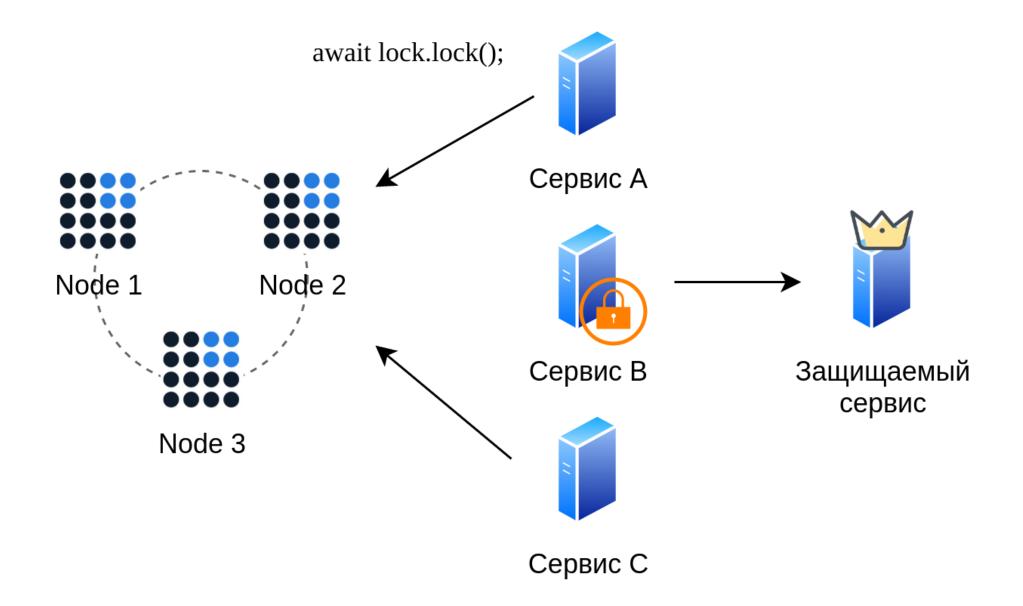
await lock.lock();
try {
    // "защищенный" код
    // (доступен только владельцу)
} finally {
    await lock.unlock();
}
```

#### Как это должно работать





#### Как это должно работать

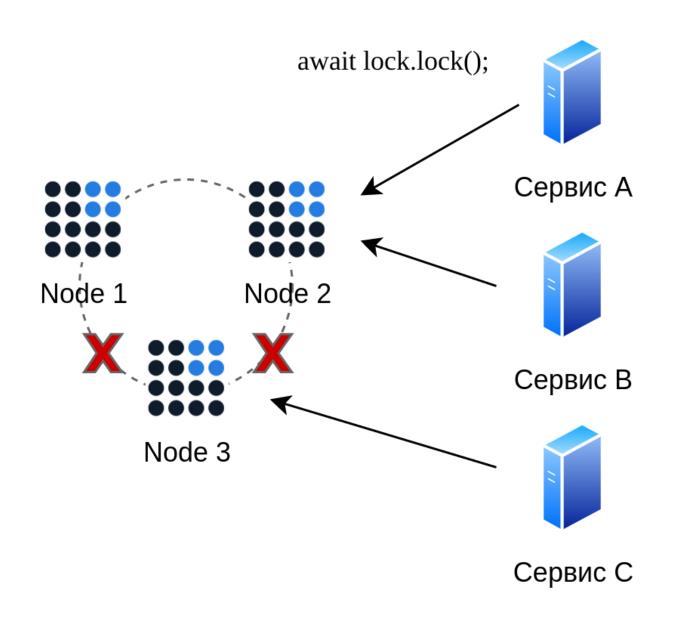


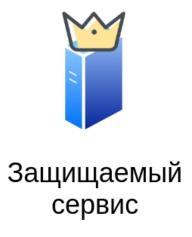
# Хьюстон, у нас проблема

- В 2017 Kyle Kingsbury (a.k.a. aphyr) потестировал Hazelcast
- https://jepsen.io/analyses/hazelcast-3-8-3

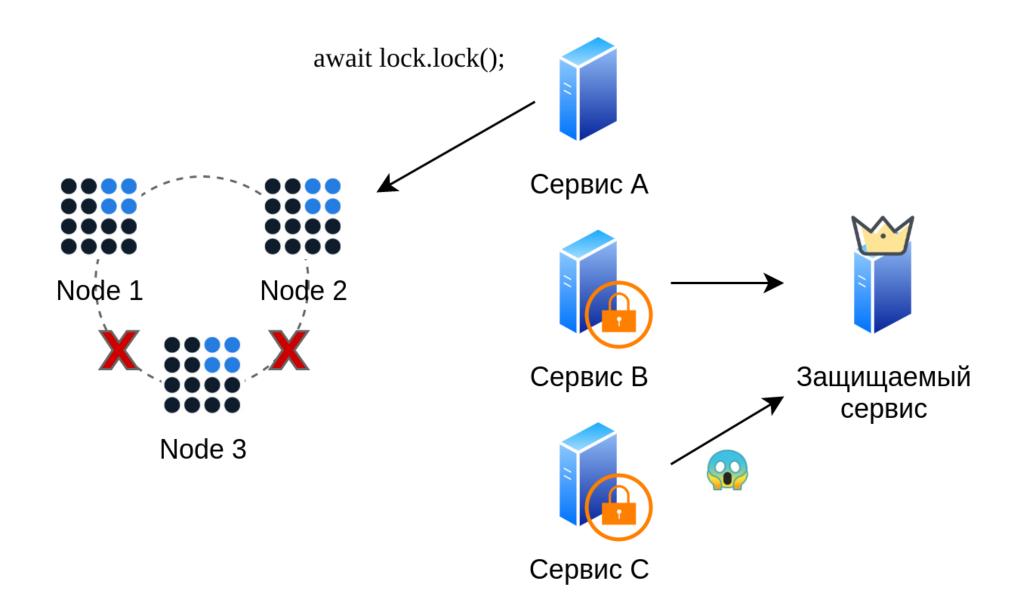


#### Как это работает в условиях network partition





#### Как это работает в условиях network partition



#### Наш ответ

- С версии 3.12 у нас есть СР Subsystem (т.е. production-ready Raft с Jepsen тестами и lock'ами с)
- https://docs.hazelcast.org/docs/4.0.3/manual/html-single/index.html#cp-subsystem

#### Новая реализация распределенных lock'ов

```
// У ней внутри Raft
const lock = await client.getCPSubsystem().getLock('my-lock');
const fence = await lock.lock();
try {
    // "защищенный" код
    // (доступен только владельцу)
} finally {
    await lock.unlock(fence);
```

P.S. A еще у нас есть Semaphore, AtomicLong и AtomicReference.

#### Что за зверь Raft?

- Алгоритм консенсуса, придуманный D.Ongaro и J.Ousterhout (Stanford) в 2013
- Некоторые юзкейсы:
  - Хранилище пар ключ-значение
  - Распределенные lock'и\* (и не только)
  - Выбор лидера
  - Ограничения на значения полей (например, уникальность)
  - Атомарный коммит (распределенные транзакции)

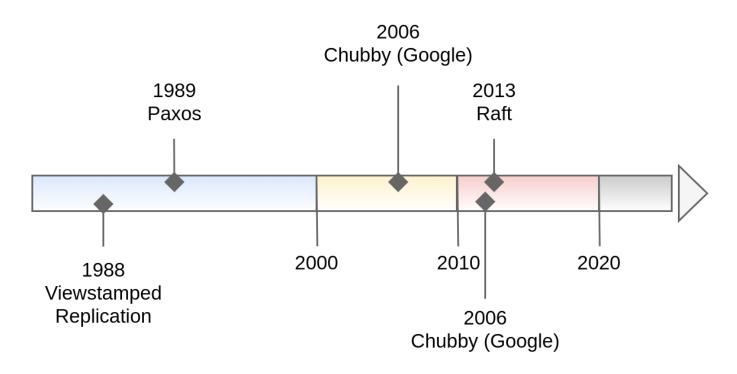
#### План на сегодня

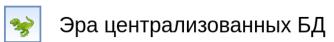
- Начинаем пугаться распределенных систем
- Знакомимся с моделями согласованности (consistency)
- САР теорема и прочие классификации
- Что за зверь алгоритм консенсуса?
- История: Paxos и его подвиды, Raft
- CASPaxos, как один из недавних Paxos-образных
- Pet project: CASPaxos на Node.js

Начинаем пугаться распределенных систем

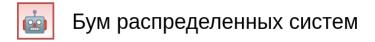


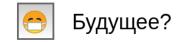
### Упрощенная до ужаса история







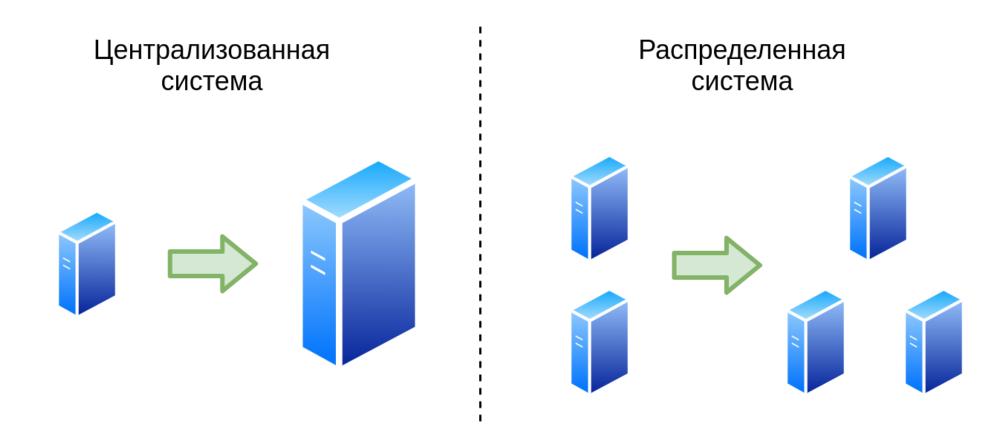




#### Распределенная система

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

### Отличия: масштабирование



#### Отличия: работа с данными

Централизованная система

const data = await readData();



Распределенная система

const data = await readData();



#### Отличия: вероятность отказа

Централизованная система

P(S)



Распределенная система

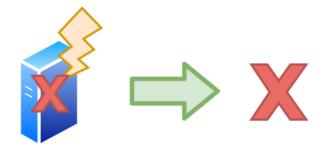
$$P(S1 + S2) = P(S1) + P(S2) - P(S1 \times S2)$$





#### Отличия: критичность отказа

Централизованная система





#### 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

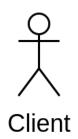
- 1. Отказоустойчивость
- 2. Масштабируемость
- 3. Удобство поддержки (мониторинг, администрирование)

#### 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)

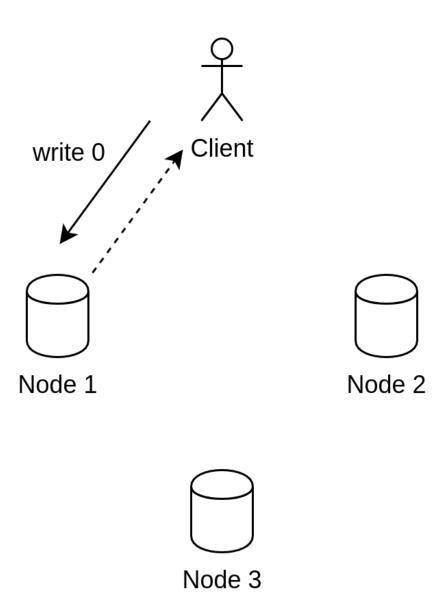
#### Фигня вопрос - сейчас придумаем алгоритм

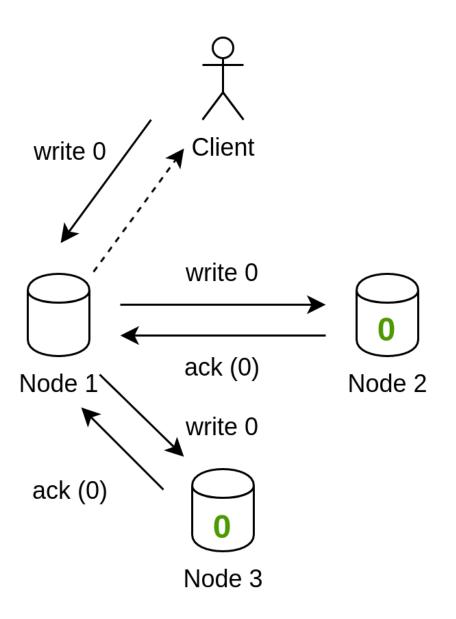
- 1. Любой узел принимает клиентские запросы (прочитать/записать)
- 2. Затем отправляет операцию на все остальные узлы
- 3. Ждет ответов от большинства (консенсус жеж 😜)
- 4. Дождавшись консенсуса, отправляет клиенту сообщение об успехе

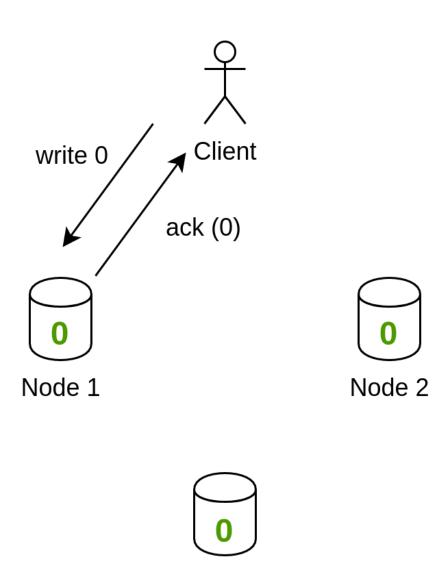




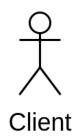




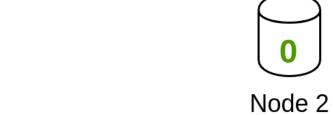




Node 3

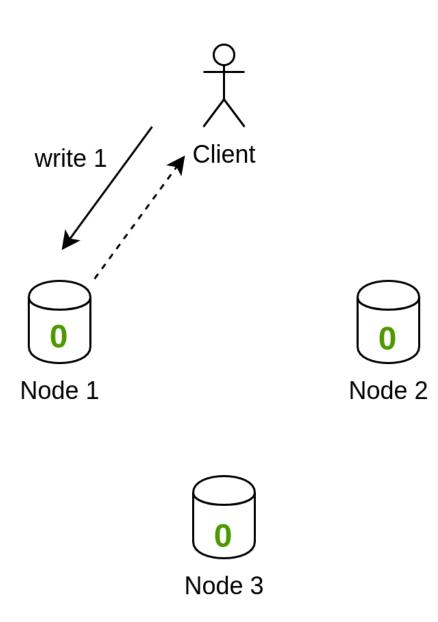


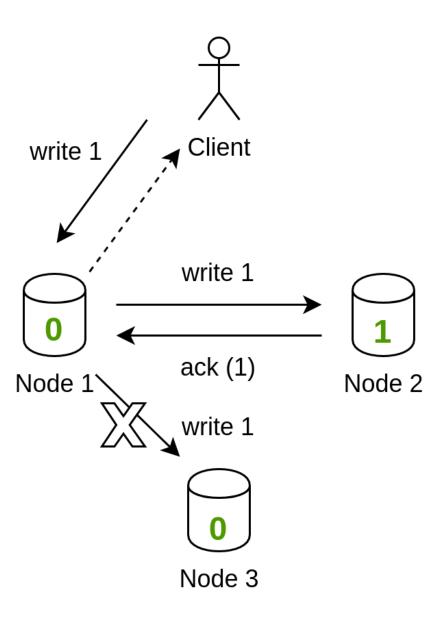


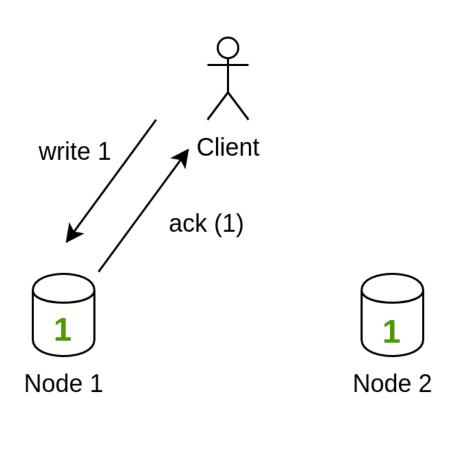




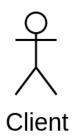
Node 3













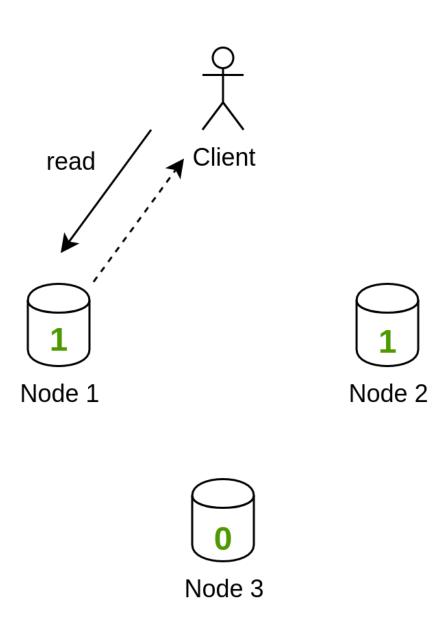
Node 1

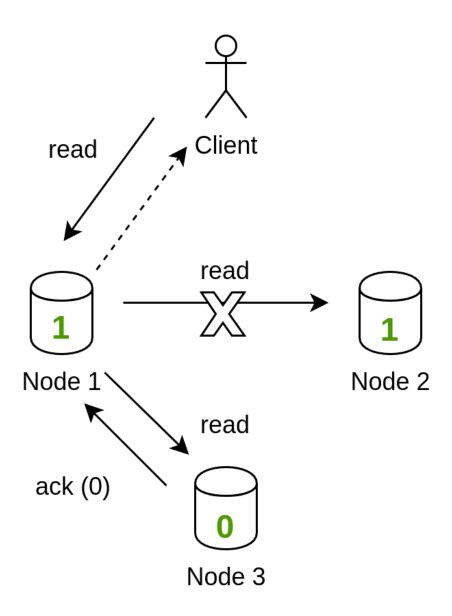


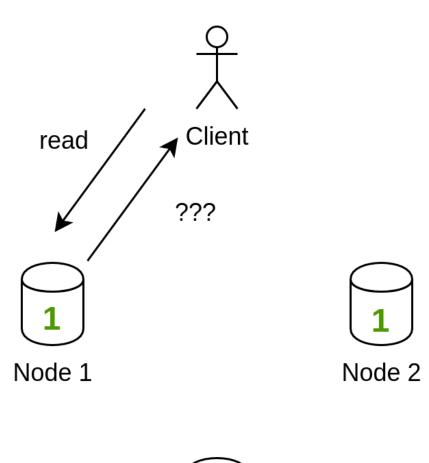
Node 2



Node 3







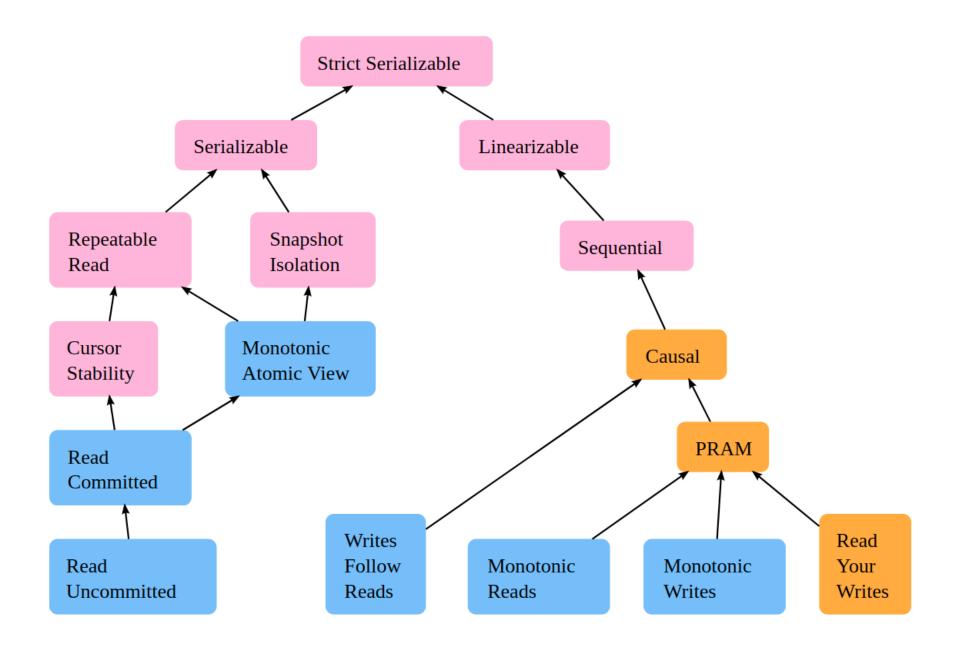


> Знакомимся с моделями согласованности (consistency)



# Неформальное определение

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



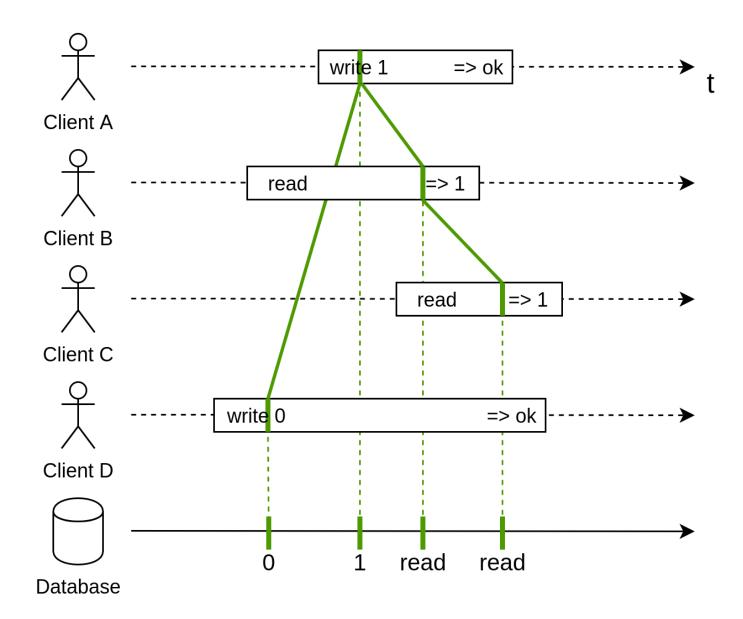
## Linearizable consistency model

- 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"

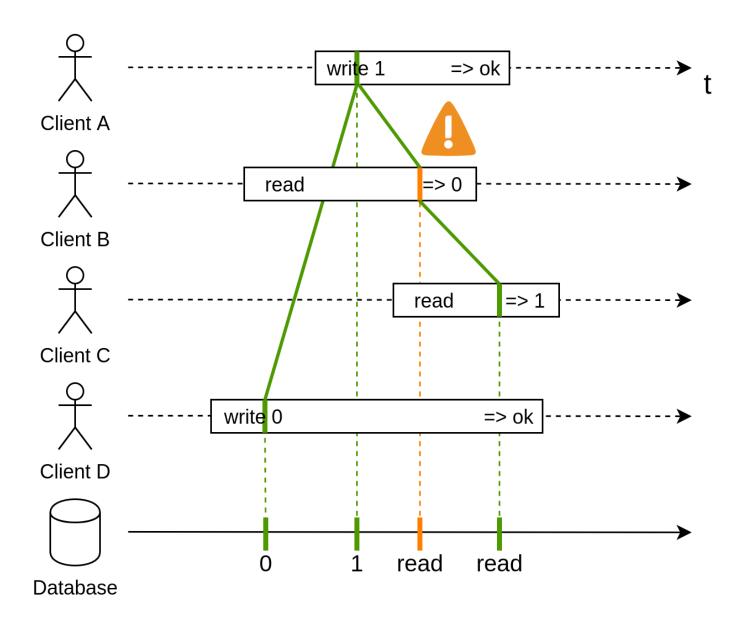
## Неформальное определение linearizability

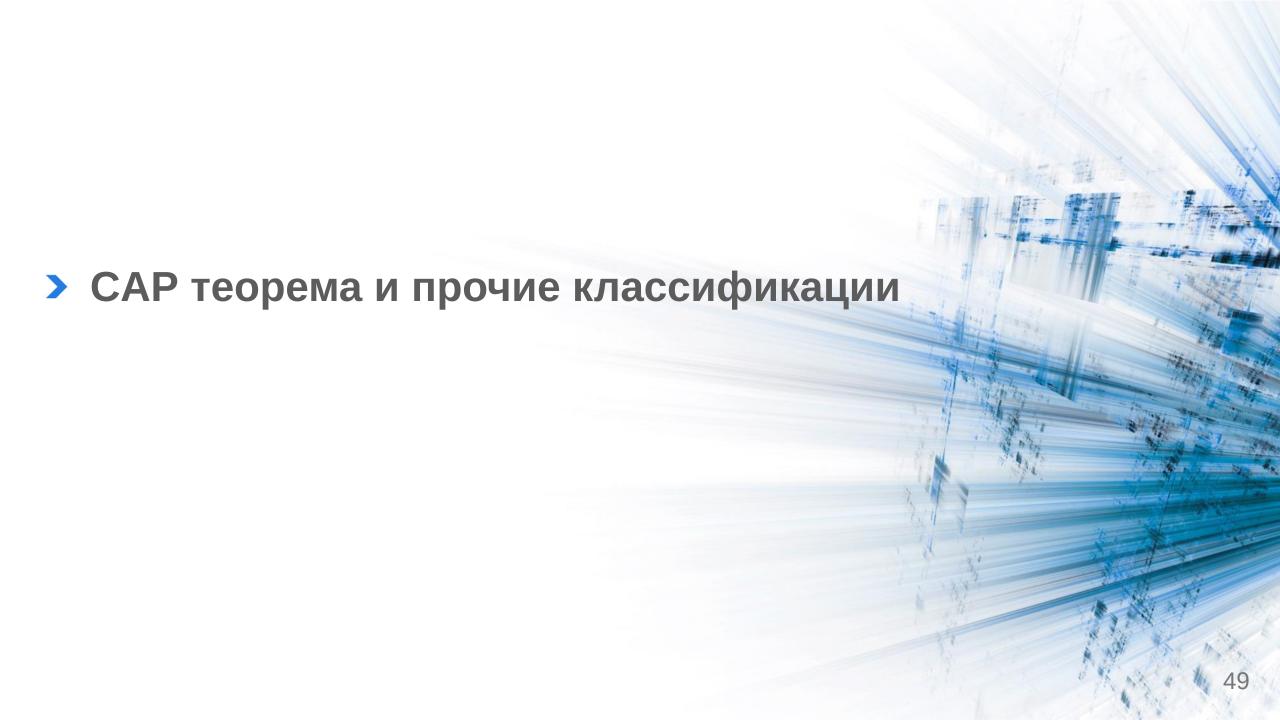
- Система гарантирует, что каждая операция выполняется атомарно, в **некотором** (общем для всех клиентов) порядке, не противоречащим порядку выполнения операций в реальном времени
- Т.е. если операция А завершается до начала операции В, то В должна учитывать результат выполнения операции А

# Допустимый порядок



# Недопустимый порядок





### САР теорема

- Сформулирована Eric Brewer в 1998, как утверждение
- В 2002 появилось формальное доказательство
- Рассматривается система с одним регистром
- CAP:
  - Consistency: подразумевается линеаризуемость
  - Availability: каждый запрос, полученный узлом системы, должен приводить к ожидаемому ответу (не к ошибке)
  - Partition tolerance: подразумевается коммуникация 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 (АР)
- Согласована (СР)

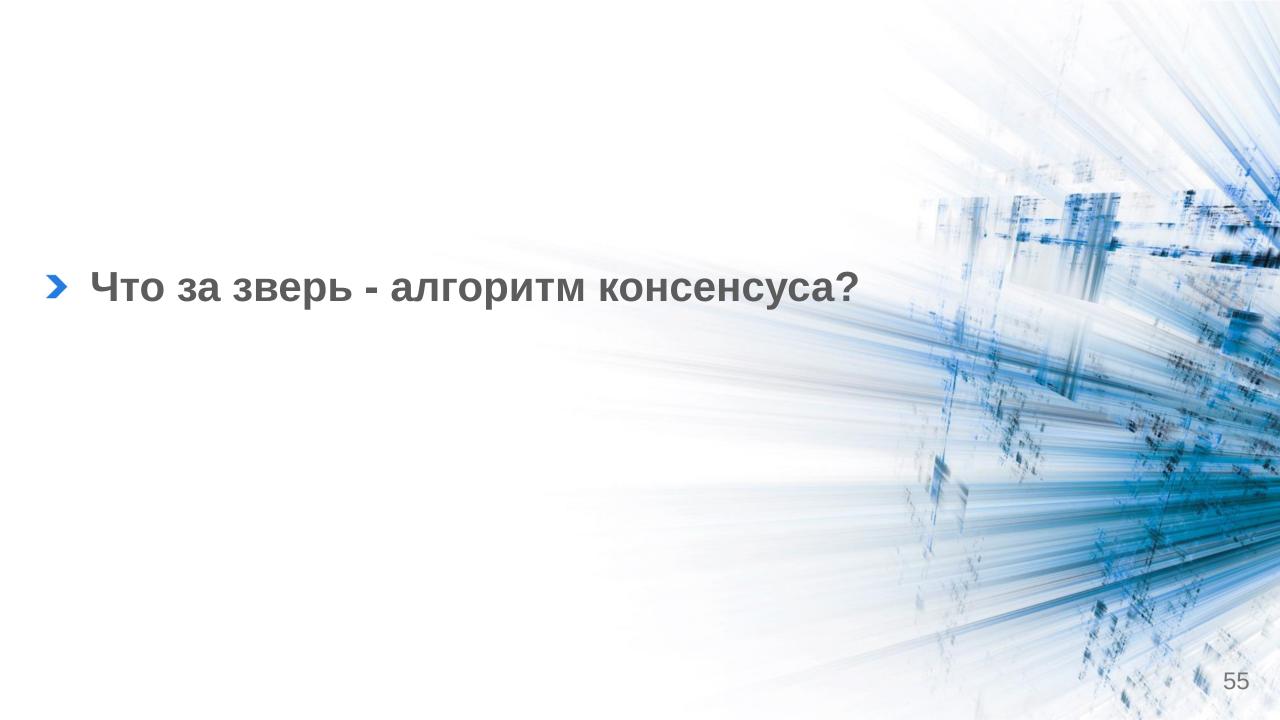
P.S. CA опции в CAP теореме нет и в помине

### Критика

- Однобокая классификация, которая почему-то прижилась
- Например, РСУБД с одной read-only репликой не является ни СР, ни АР
- Теорема ничего не говорит о времени отклика системы, т.е. АР 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



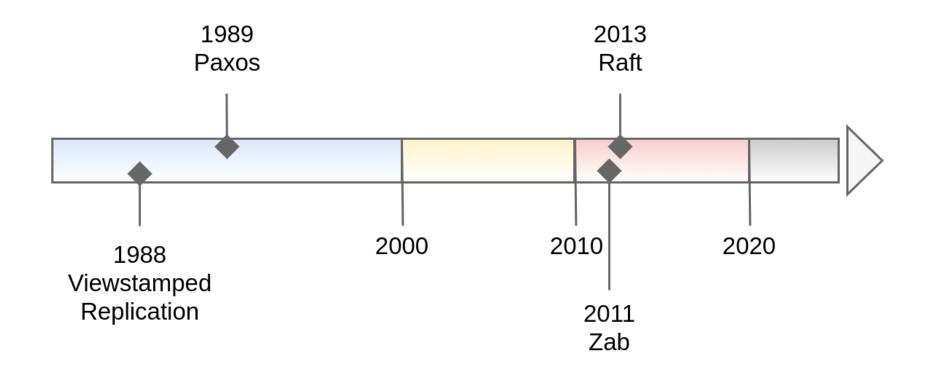
### Неформальное определение

- 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 "скандальный" FLP result (1985)

## 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 СР системах, с точки зрения пресловутой САР теоремы

# Краткий список алгоритмов консенсуса



Paxos и его подвиды: Chubby, Spanner

Raft: Consul, etcd, Hazelcast, RethinkDB, TiKV и т.д.

Zab: ZooKeeper

## Верификация корректности: теория

- TLA+ (L.Lamport)
- Язык для моделирования программ и систем, распределенных и не только

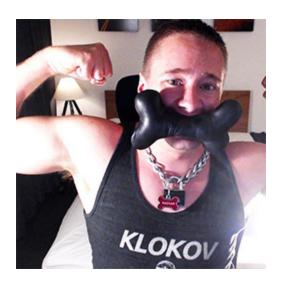
```
EXTENDS TLC

(* --algorithm hello_world
variable s \in {"Hello", "World!"};
begin
   A:
    print s;
end algorithm; *)
```

# Верификация корректности: практика

• Jepsen (K.Kingsbury a.k.a. aphyr)

```
INFO jepsen.core - Analysis invalid! (ノ成益 の) ノ 上上
```



### Темная сторона алгоритмов консенсуса

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



#### Vanilla Paxos

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

САЅРахоѕ, как один
 из недавних Рахоѕ-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

#### Фаза 1



1) Отправляет функцию *f*.



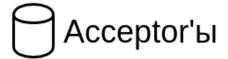
### Proposer

2) Генерирует номер бюллетеня

**B** (seq + ID) и отправляет сообщение **prepare** вместе с **B** ассерtor'ам.

4а) Ждет *F* + 1 подтверждений.Если все они содержат Ø,берет Ø в качестве текущего состояния.

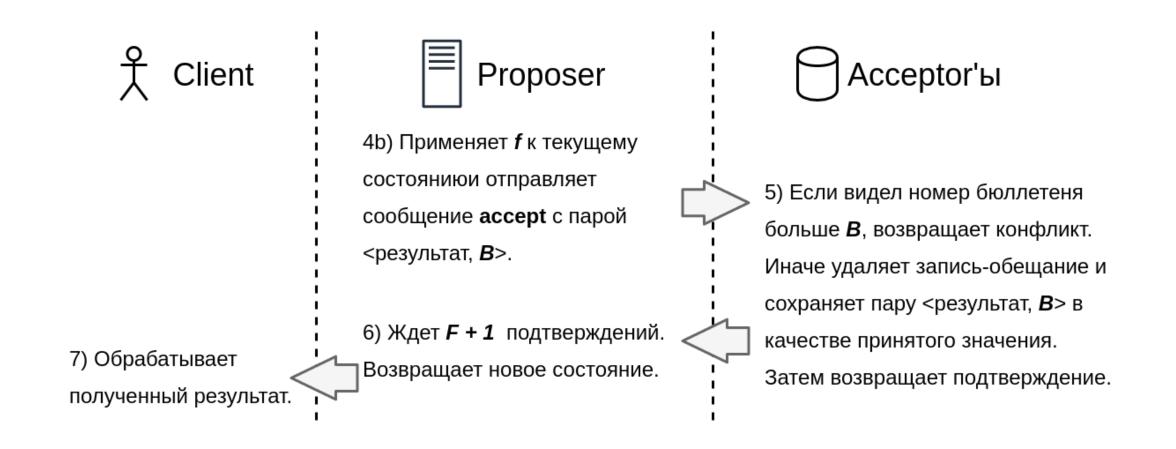
Иначе берет значение с наибольшим номером бюллетеня.



3) Если уже видел номер больше **В**, возвращает сообщение о конфликте.

Иначе сохраняет **В** в качестве записи-обещания и возвращает Ø или <принятое\_значение, его\_номер\_бюллетеня>.

#### Фаза 2



# Связь с Synod

CASPaxos эквивалентен Synod, если взять функцию:

```
x \rightarrow if x = \emptyset then val0 else x
```

### CAS регистр на основе CASPaxos

#### Чтение:

```
x -> x
```

Инициализация значением valo:

```
x \rightarrow if x = \emptyset then (0, val0) else x
```

Запись значения val1 при условии версии 3:

```
x \to if x = (3, *) then (4, val1) else x
```

### Хранилище пар ключ-значение на основе CASPaxos

- 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-проект: CASPaxos на Node.js

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

 $\bigvee$ 

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

### Чего не хватает до production-ready?

- Применить оптимизации для Paxos, например, CASPaxos 1RTT
- Поддержка динамической конфигурации, включая версионирование и сохранение на диск
- Поддержка операции delete. Нужен фоновый процесс аля GC (см. статью)
- Интеграция со встраиваемым хранилищем ключ-значение, например, RocksDB
- Перейти с коммуникации для внутренних нужд по HTTP к plain TCP
- Обработка ошибок и логирование
- И много чего еще



# Призыв к действию

- 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/