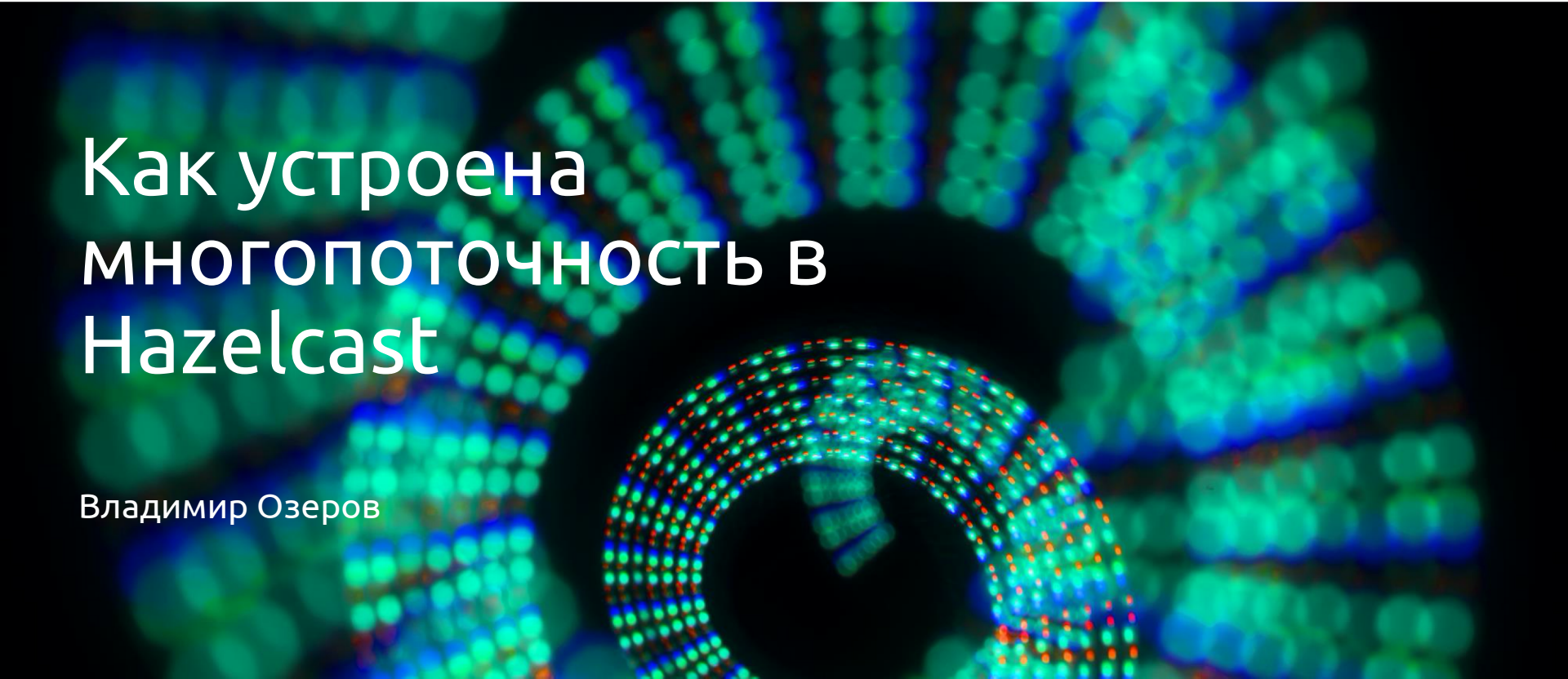
hazelcast

THE LEADING IN-MEMORY COMPUTING PLATFORM



Как устроена многопоточность в Hazelcast

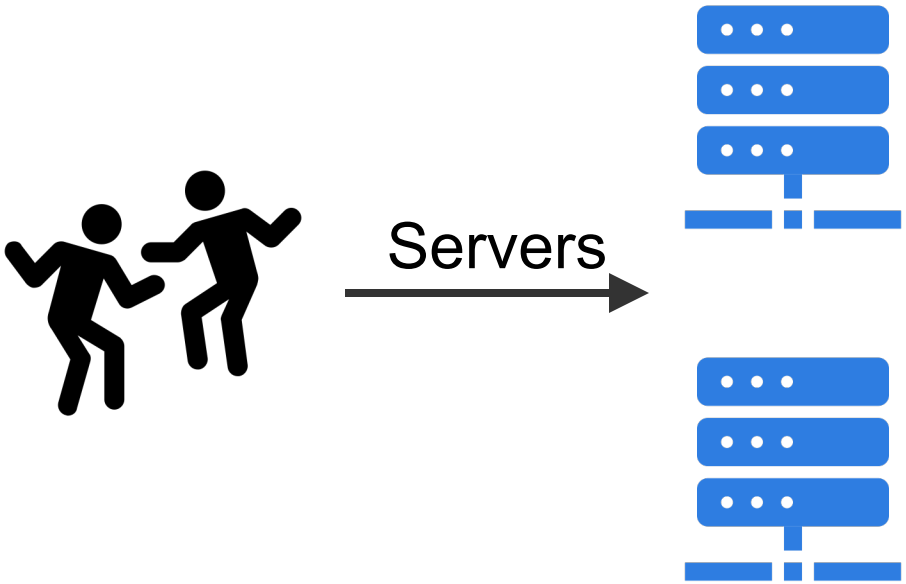
Владимир Озеров

Что такое Hazelcast?

- **In-Memory Computing Platform** - мы ускоряем ваши приложения

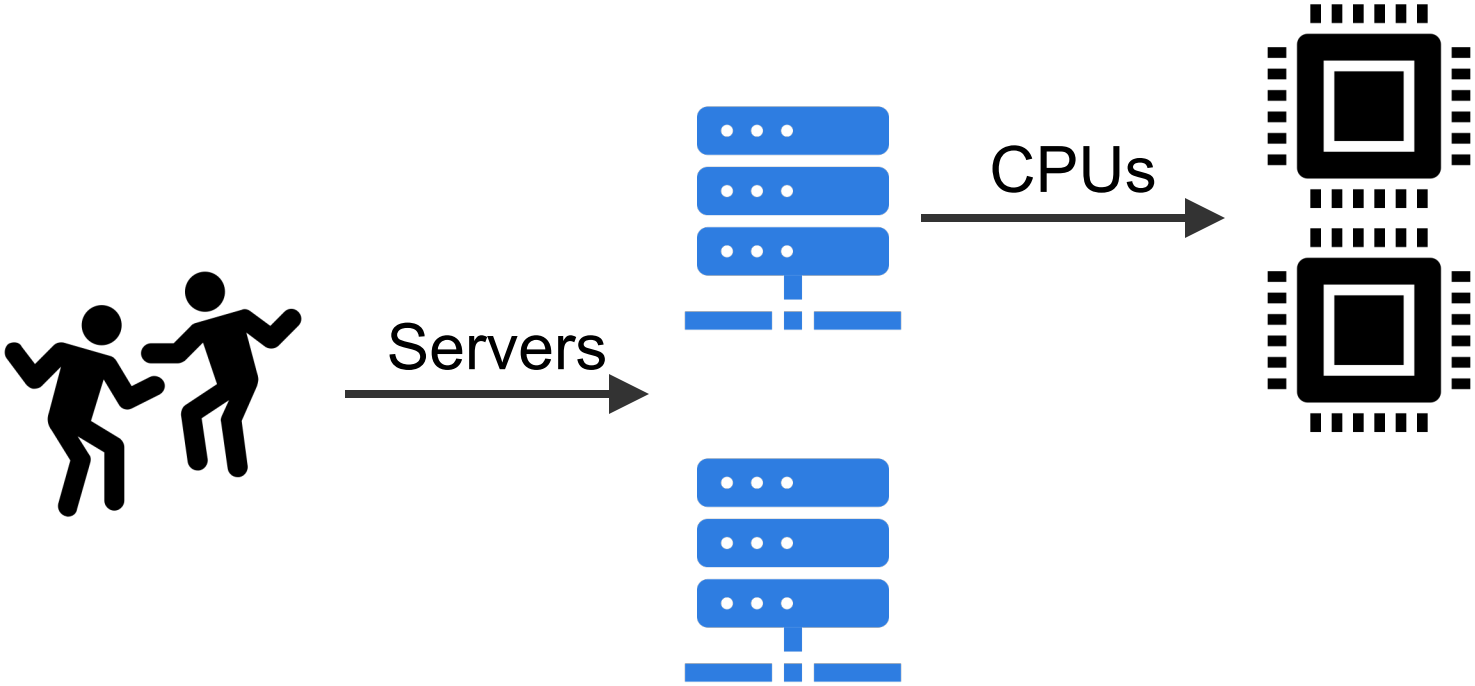
- Для этого нам нужна хорошая масштабируемость
  - В рамках кластера
  - В рамках одной машины

Масштабирование



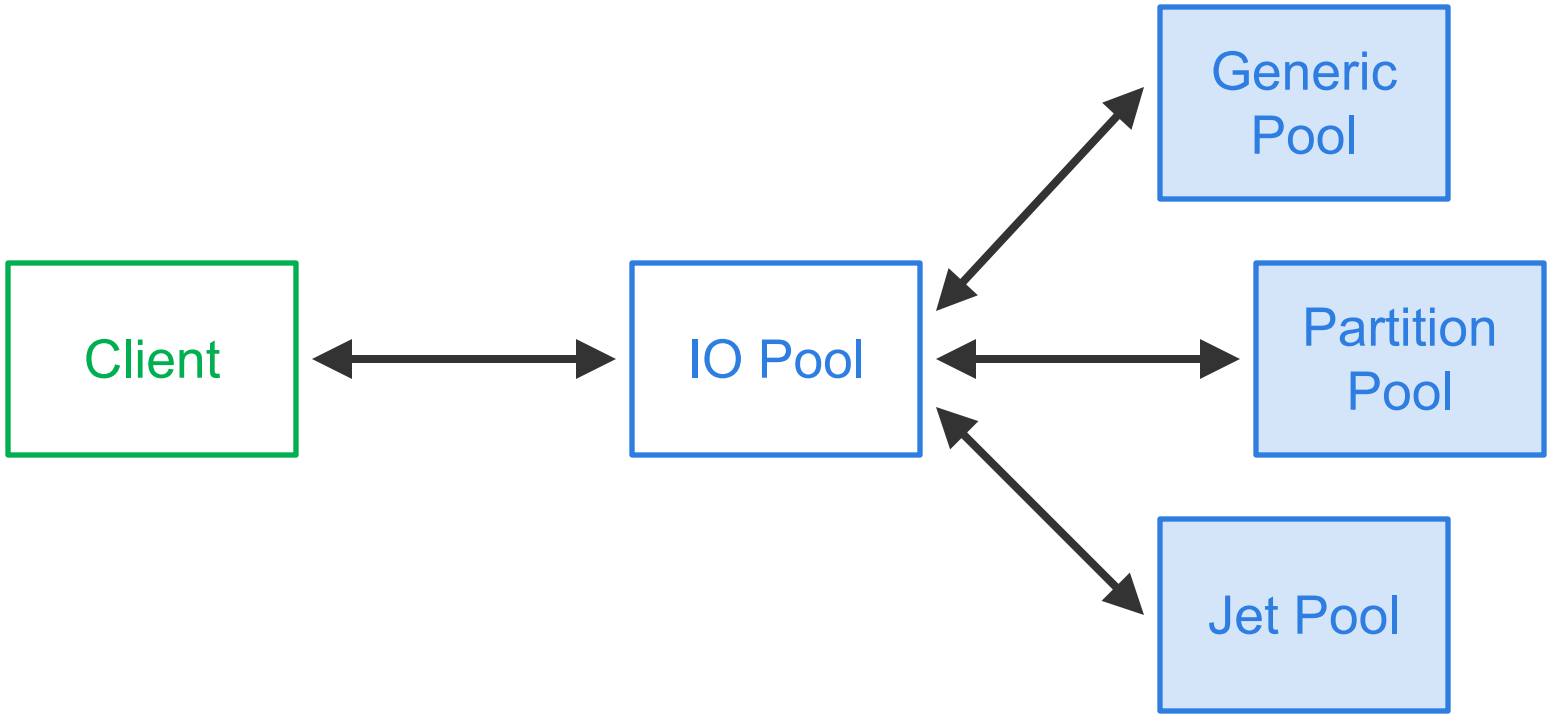
hazelcast

Масштабирование



hazelcast

Архитектура Hazelcast



hazelcast

О чем будет доклад?

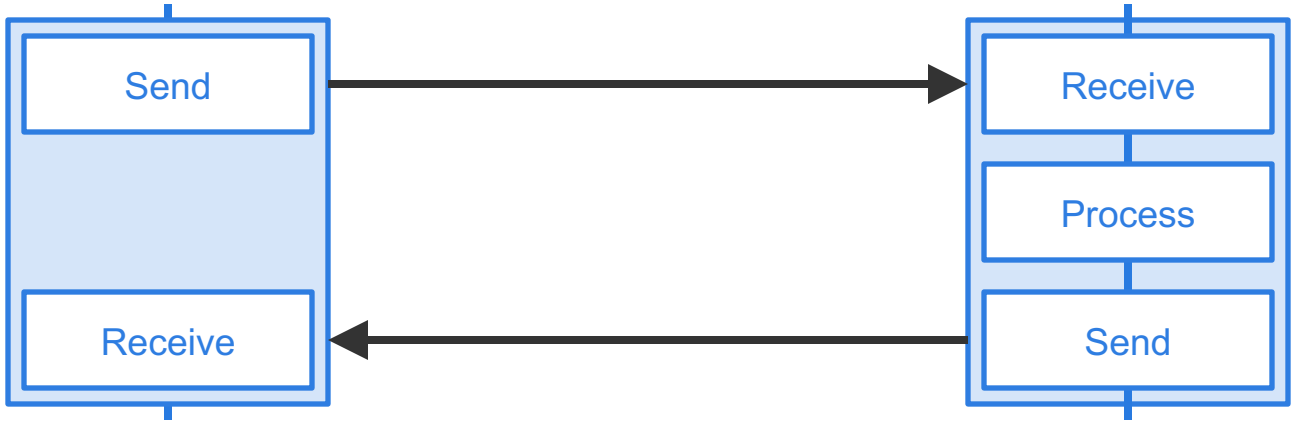
- Почему Hazelcast работает именно так?
- Какие альтерантивные архитектуры параллельной обработки применяют в распределенных продуктах?

Задача

```
class IMap<K, V> {
    void put(K key, V value) {
        // Записать ключ на удаленный сервер
    }
}
```

Их много, а он один: как обеспечить масштабируемость на одном сервере?

Blocking IO



Client

Server

hazelcast

Blocking IO

```
while (true) {
    Request req = receive();        // <= Block/input
    Response resp = execute(req);
    send(resp);                     // <= Output
}
```

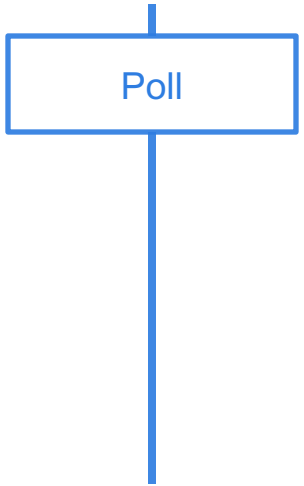
Blocking IO

- Предполагает наличие одного серверного потока на каждый клиентский поток
- Отличное решение для невысокой нагрузки: просто
- Плохо масштабируется
  - Ресурсы потока (стэк)
  - Scheduling (context switch, preemption)

Улучшаем: мультиплексирование

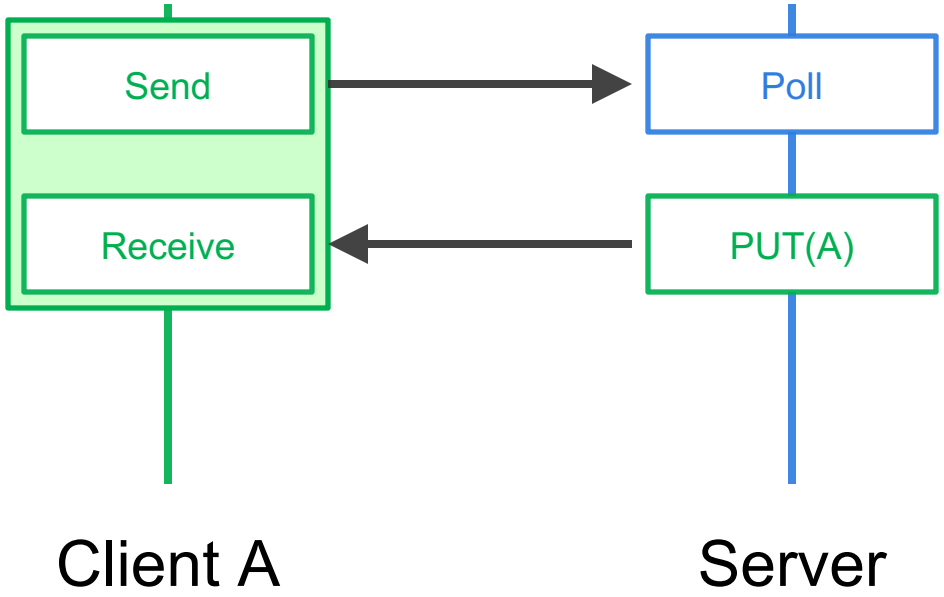
- Один поток для нескольких соединений
- Для этого необходимо non-blocking IO

Event loop



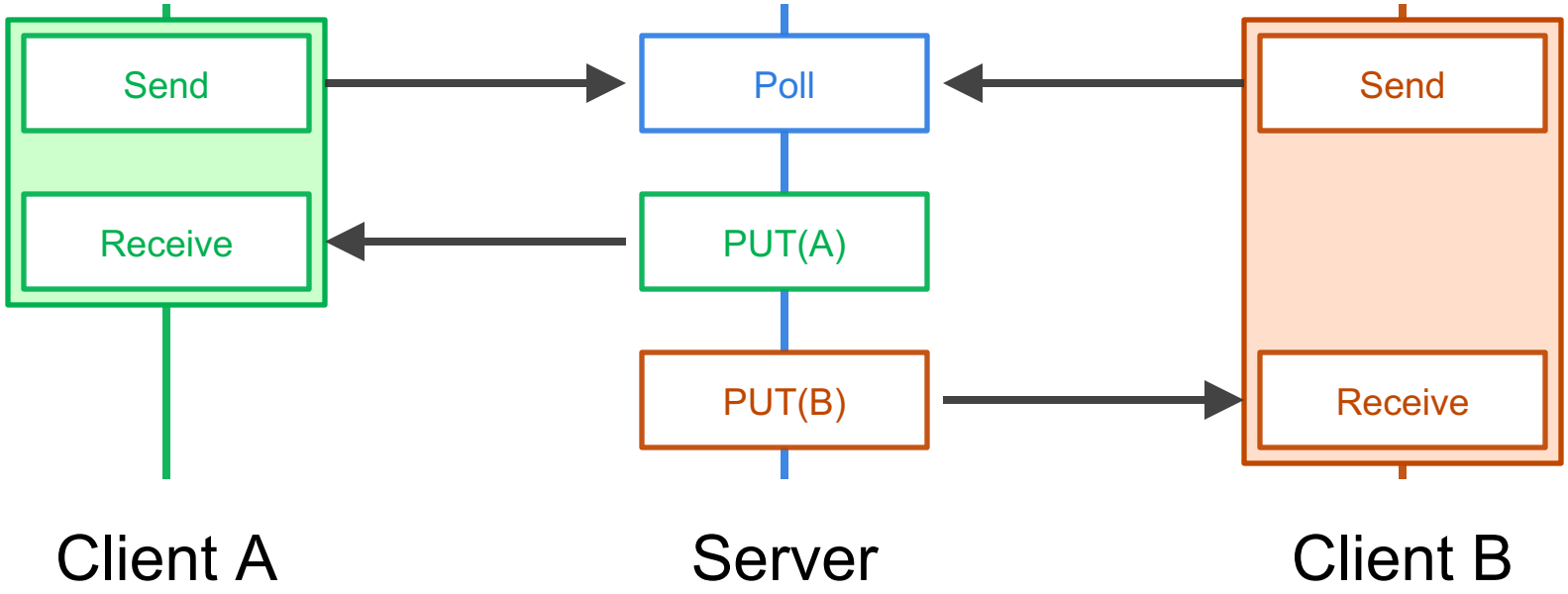
hazelcast

Event loop



hazelcast

Event loop



hazelcast

Event loop

```
while (true) {
    Set<SelectionKey> keys = select(); // <= Block

    for (SelectionKey key : keys) {
        processTask(key);              // <= Output
    }
}
```

Event loop: примеры

- Redis
- NodeJS

Event loop: анализ

- Справляется с большим количеством клиентов
- **Все** типы задач выполняются в одном потоке
  - Масштабируется, пока хватает одного CPU

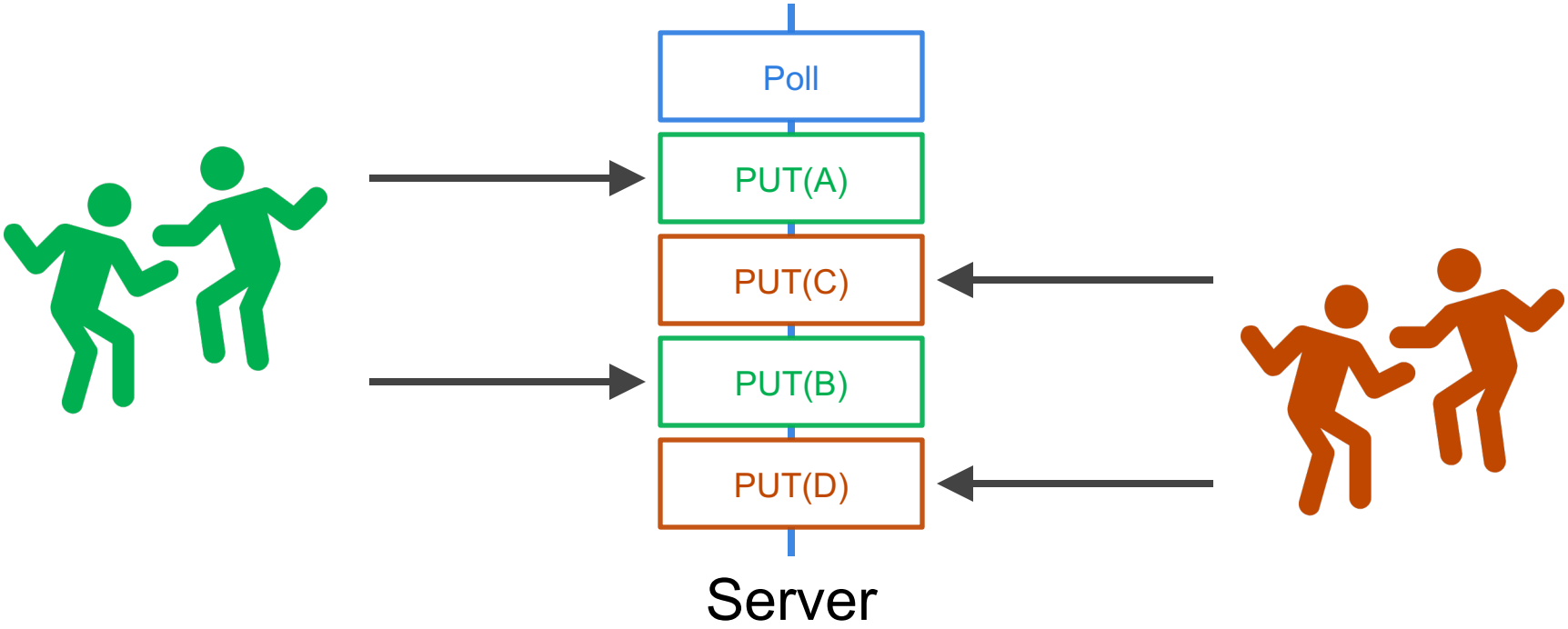
Event loop: анализ

- Справляется с большим количеством клиентов
- **Все** типы задач выполняются в одном потоке
  - Масштабируется, пока хватает одного CPU

Проблемы:

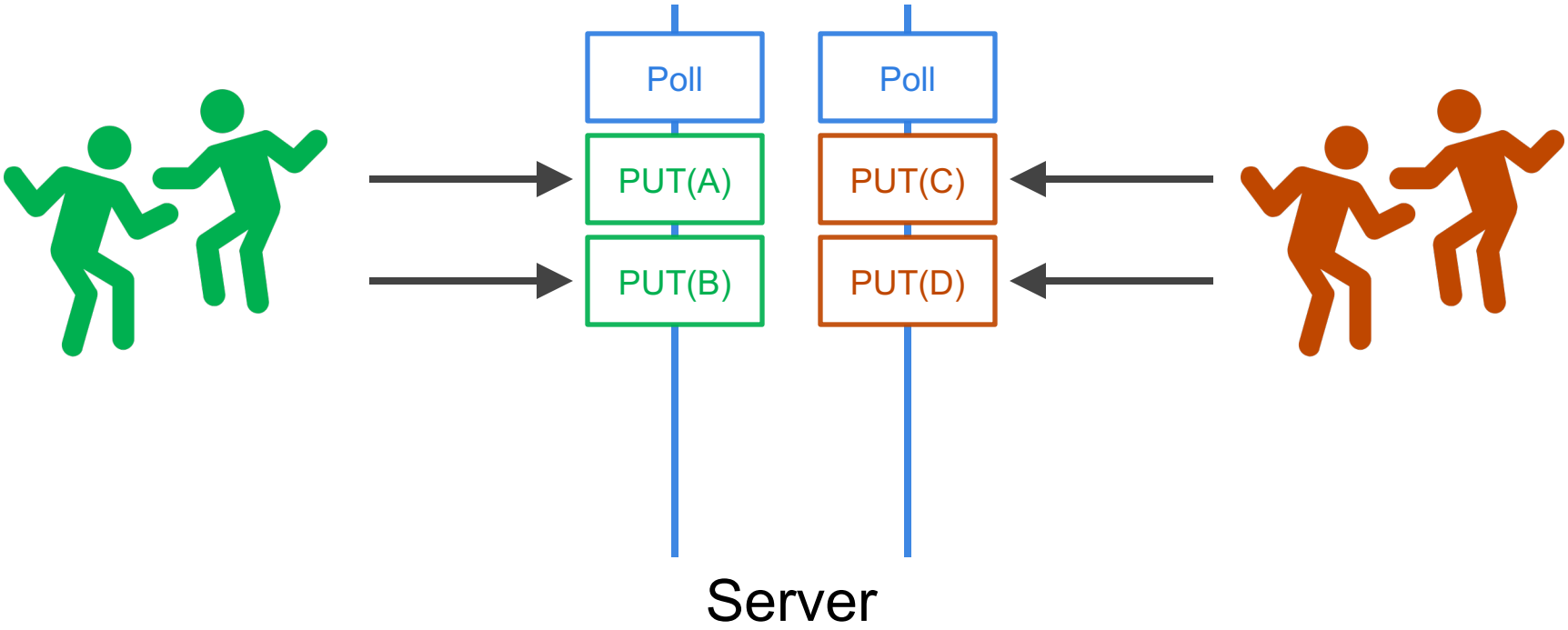
- Что делать, если одного CPU не хватает для обработки всех запросов?
- Что делать, если задачи разнородные, и могут занимать значительное время?

Event loop: параллелизм



hazelcast

Event loop: параллелизм



hazelcast

Event loop и параллелизм

- Если не хватает одного CPU, давайте запустим больше потоков!

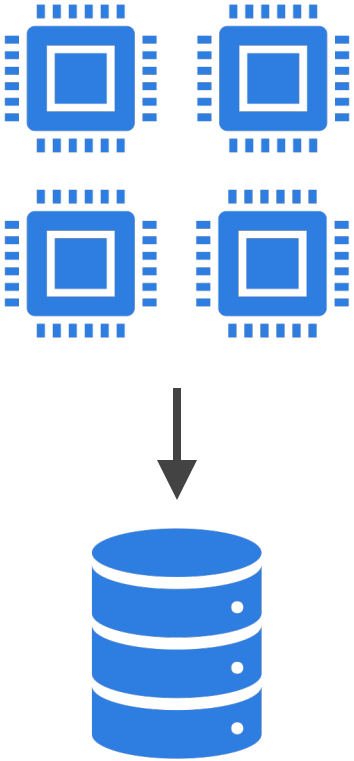
- Но как распределять задачи по потокам?
  - По клиентам?

Event loop и параллелизм

- Если не хватает одного CPU, давайте запустим больше потоков!

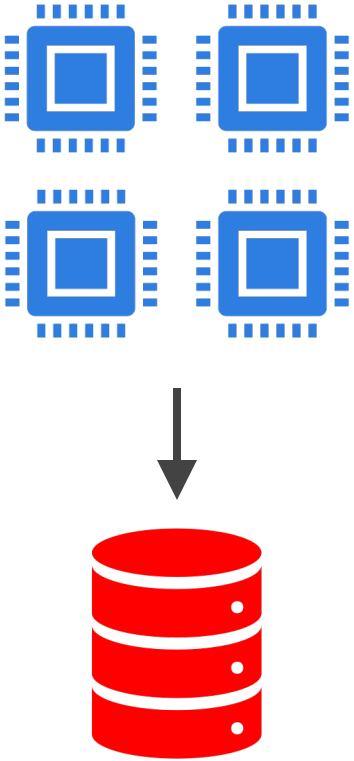
- Но как распределять задачи по потокам?
  - По клиентам?
  - **По данным!**

Доступ к данным



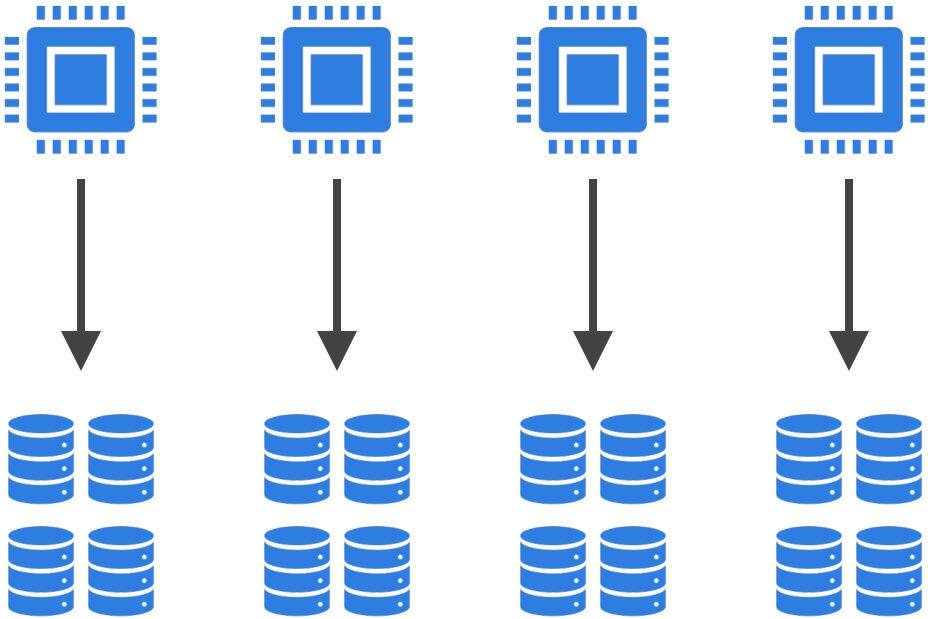
hazelcast

Доступ к данным

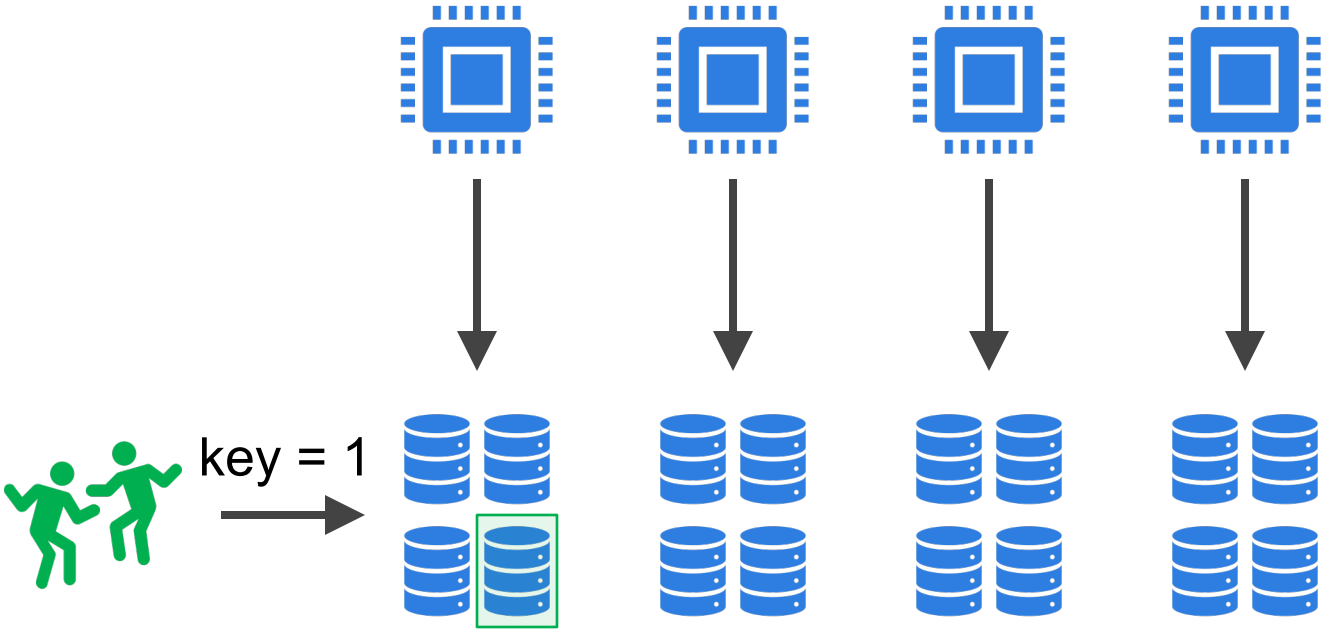


hazelcast

Доступ к данным: shared nothing

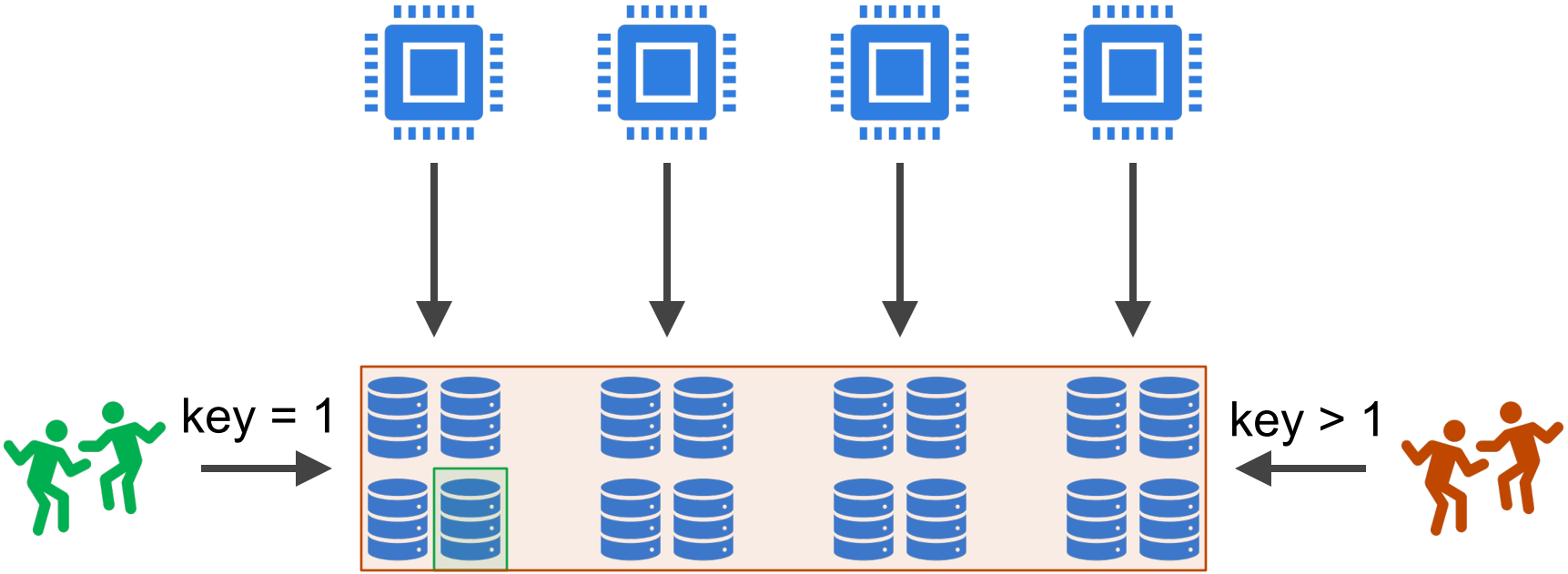


Доступ к данным: shared nothing



hazelcast

Доступ к данным: shared nothing

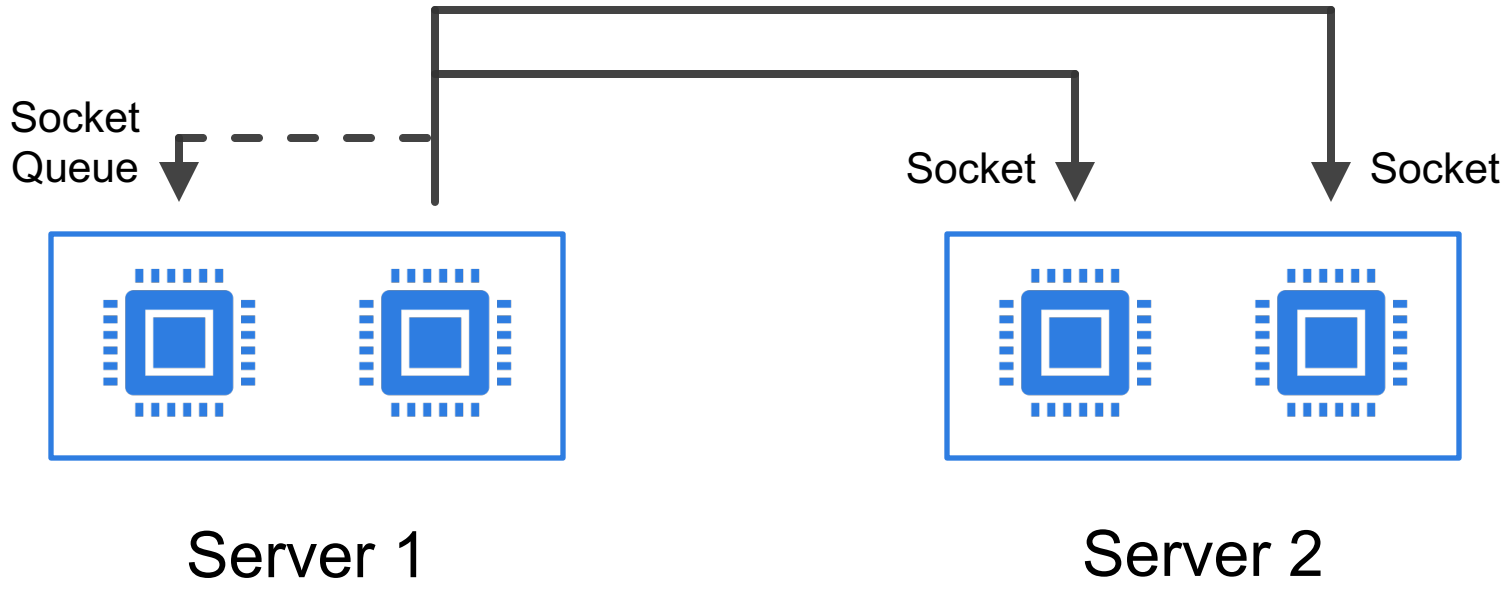


hazelcast

Shared nothing

- Один поток отвечает за подмножество данных
- Нет нужно беспокоиться о конкурентном доступе к данным
- Не очень комфортно для операций, которые затрагивают несколько partition-ов
  - Нужен map-reduce

Event loop: параллелизм на практике



hazelcast

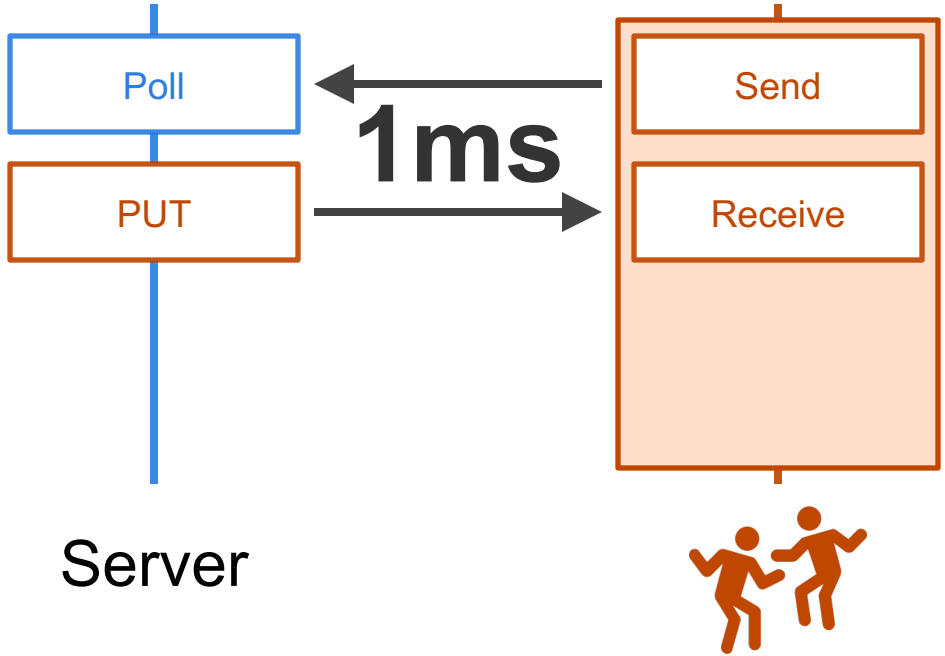
Event loop: параллелизм на практике

- Создаем несколько потоков, обычно по количеству ядер
- Каждый поток отвечает за подмножество данных (шард)
  - Не нужно контролировать конкурентный доступ к данным
- Потоки общаются через сокеты или очереди
- Практически нет накладных расходов на координацию потоков

Event loop: параллелизм на практике

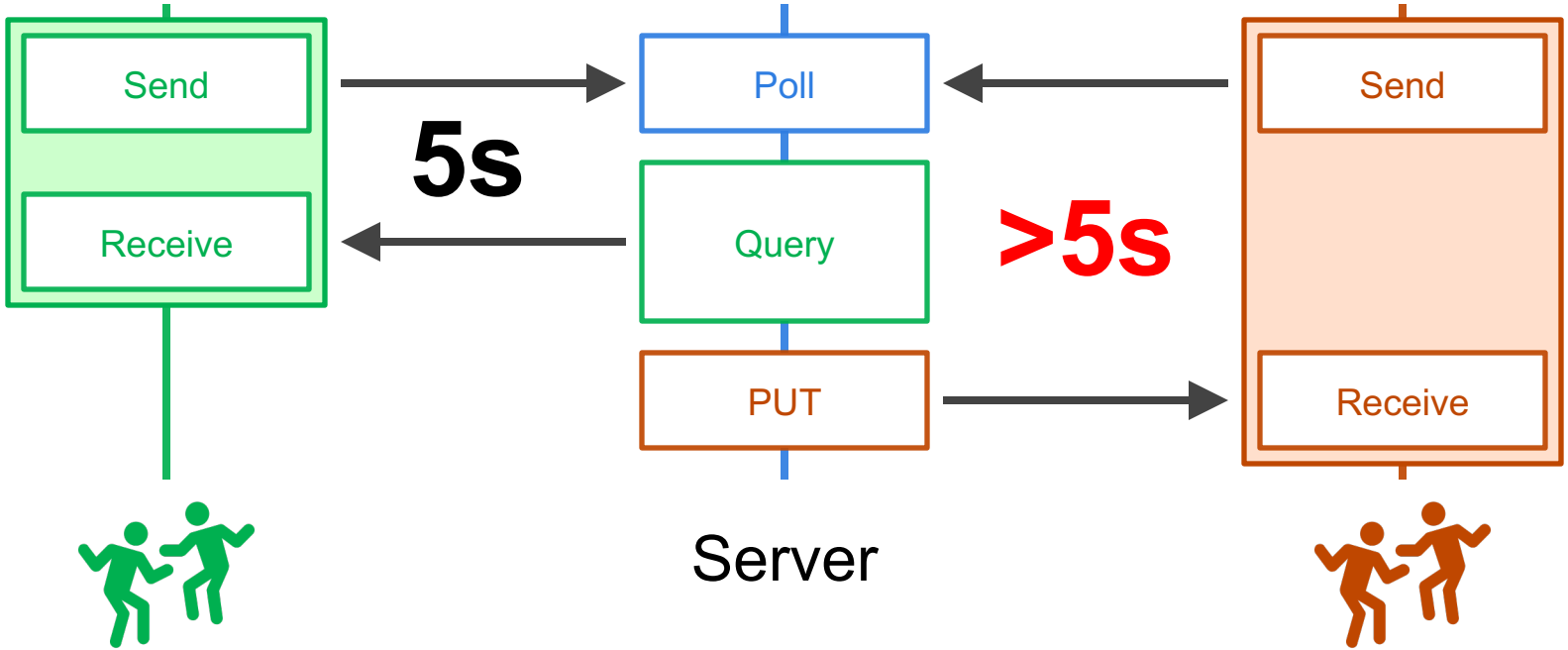
- Redis Cluster
- ScyllaDB
- DataStax Enterprise 6+

Event loop: разные операции



hazelcast

Event loop: разные операции



hazelcast

Варианты решения

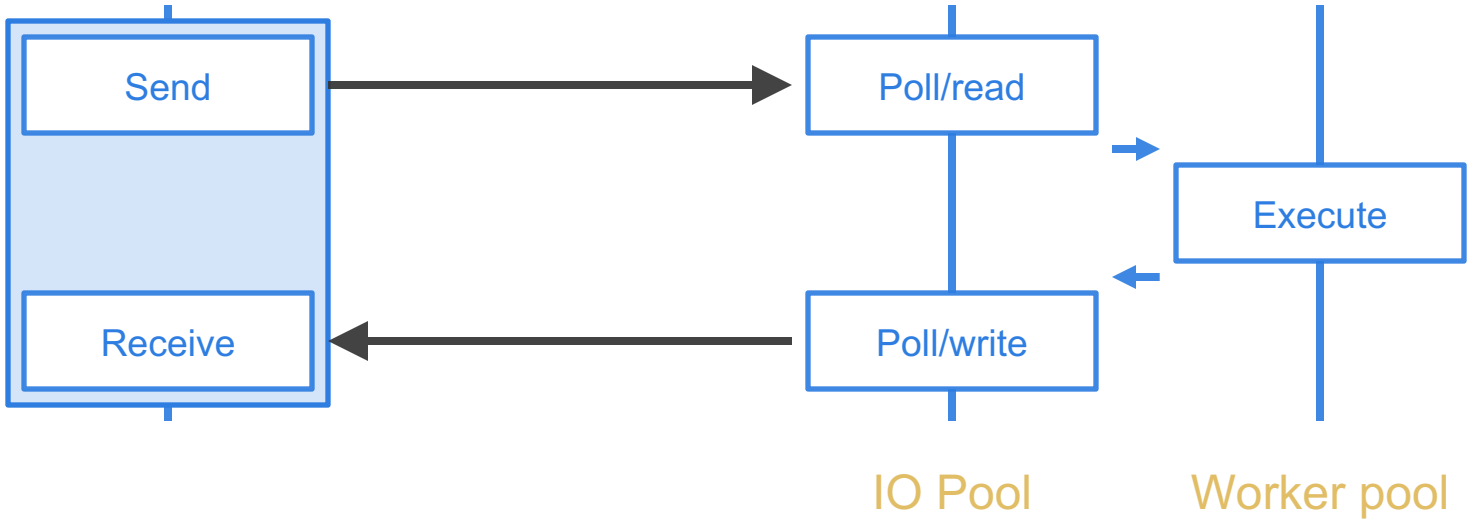
- Разные кластеры под разные нагрузки
  - Redis Cluster
- Кооперативная многозадачность: добровольно отдаем контроль из долгой задачи
- SEDA: staged event-driven architecture

Решение: SEDA

Staged event-driven architecture:

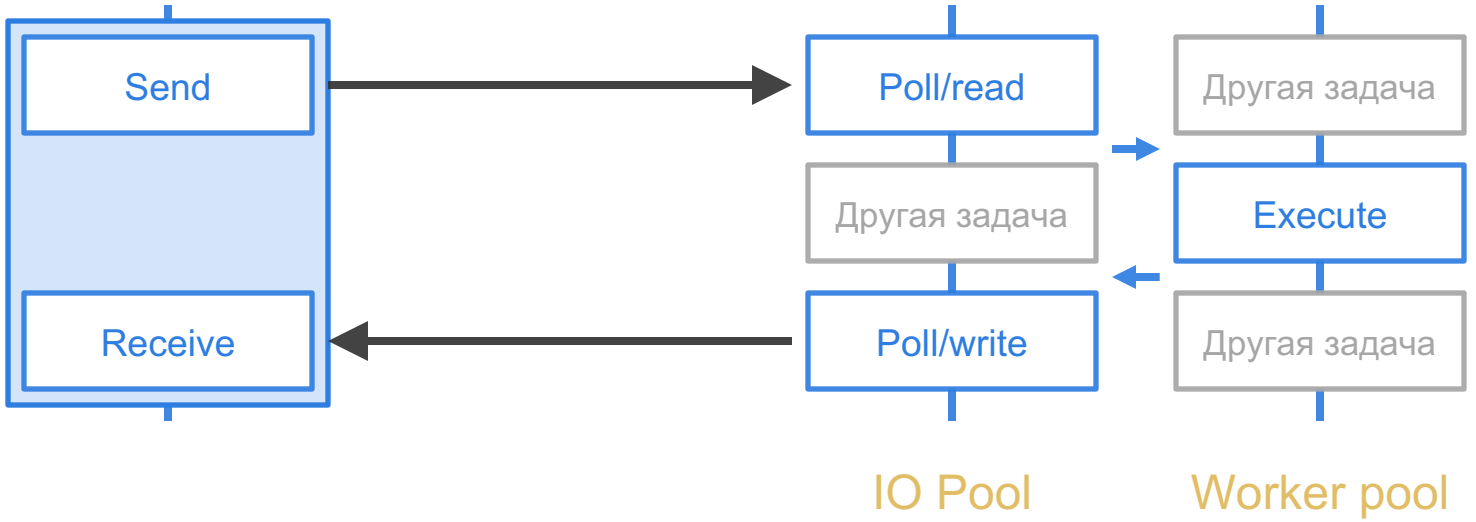
- Отдельный thread pool под каждый тип задач (stage)
- Стадии обмениваются задачами через очереди

SEDA на сервере



hazelcast

SEDA на сервере



hazelcast

Преимущества SEDA

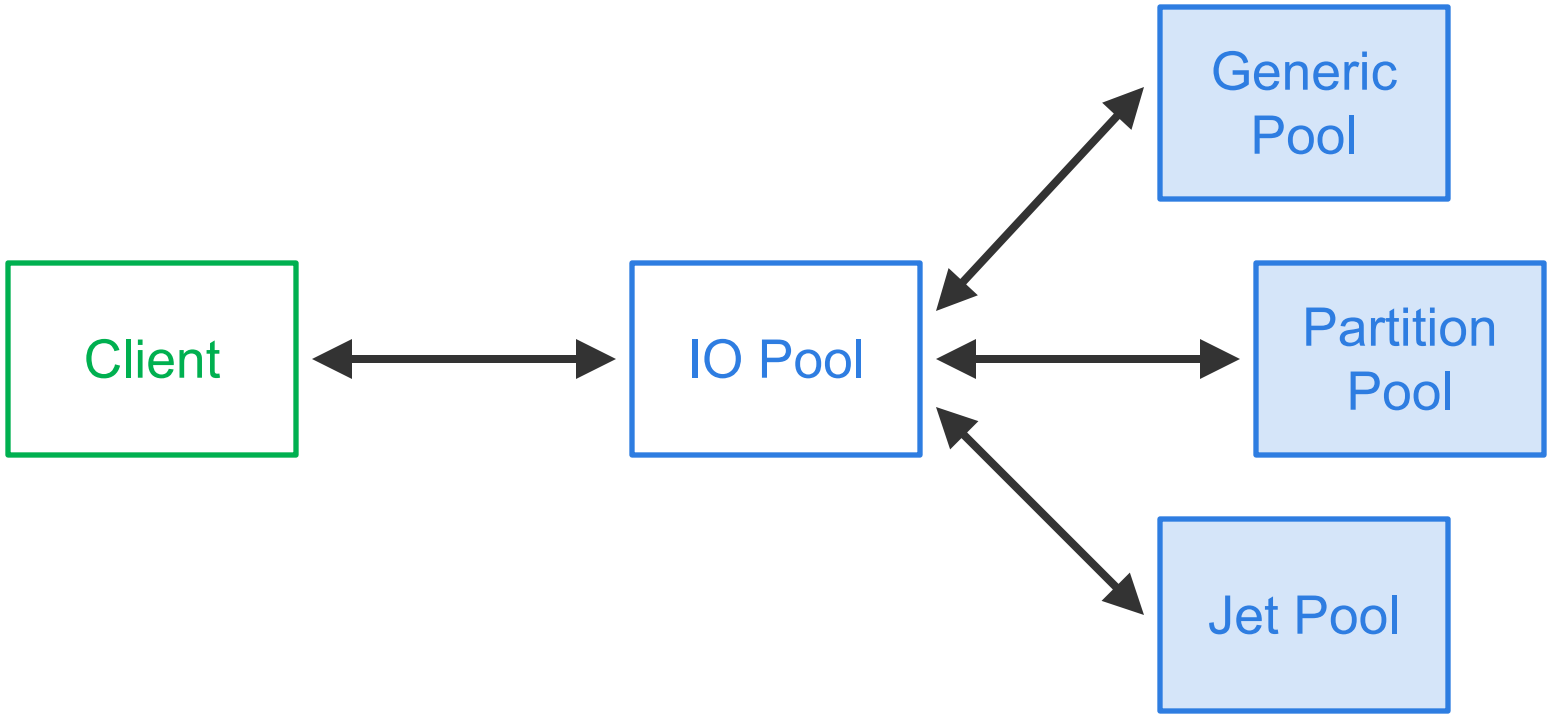
- Хорошая утилизация ресурсов и масштабируемость
  - Стадии не блокируют друг друга
- Гибкость: подходит для разных типов задач
- Относительно проста в реализации

Архитектура Hazelcast: SEDA

- Использует SEDA
- Стадии:
  - **IO pool** – чтение из сокета, запись в сокет
  - **Generic pool** – обработка произвольных задач
  - **Partition pool** – обработка key-value задач
  - **JET pool** – обработка streaming задач Hazelcast JET

- Типичный workflow:
  - `IO pool -> task pool -> IO pool`

Архитектура Hazelcast: SEDA

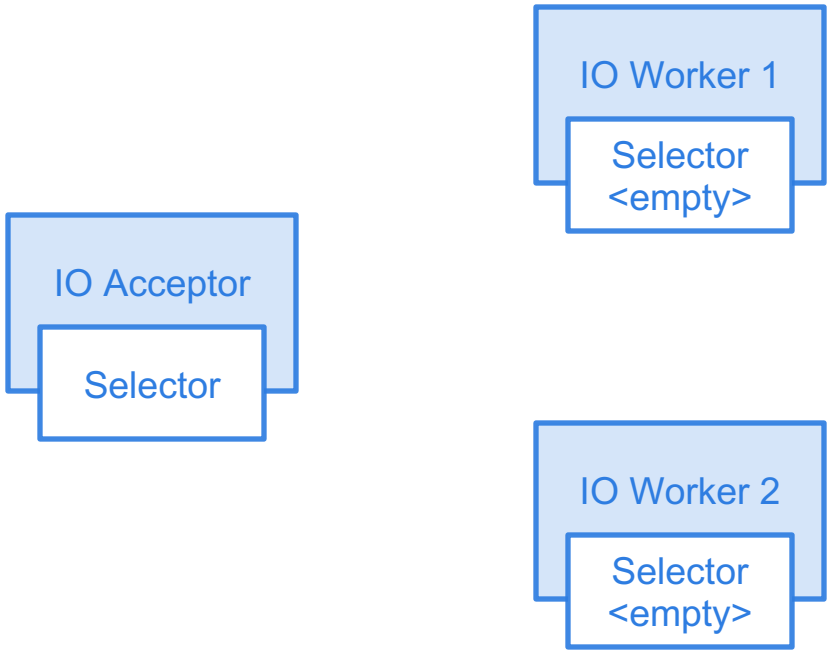


hazelcast

Реализация IO: соединение (1)

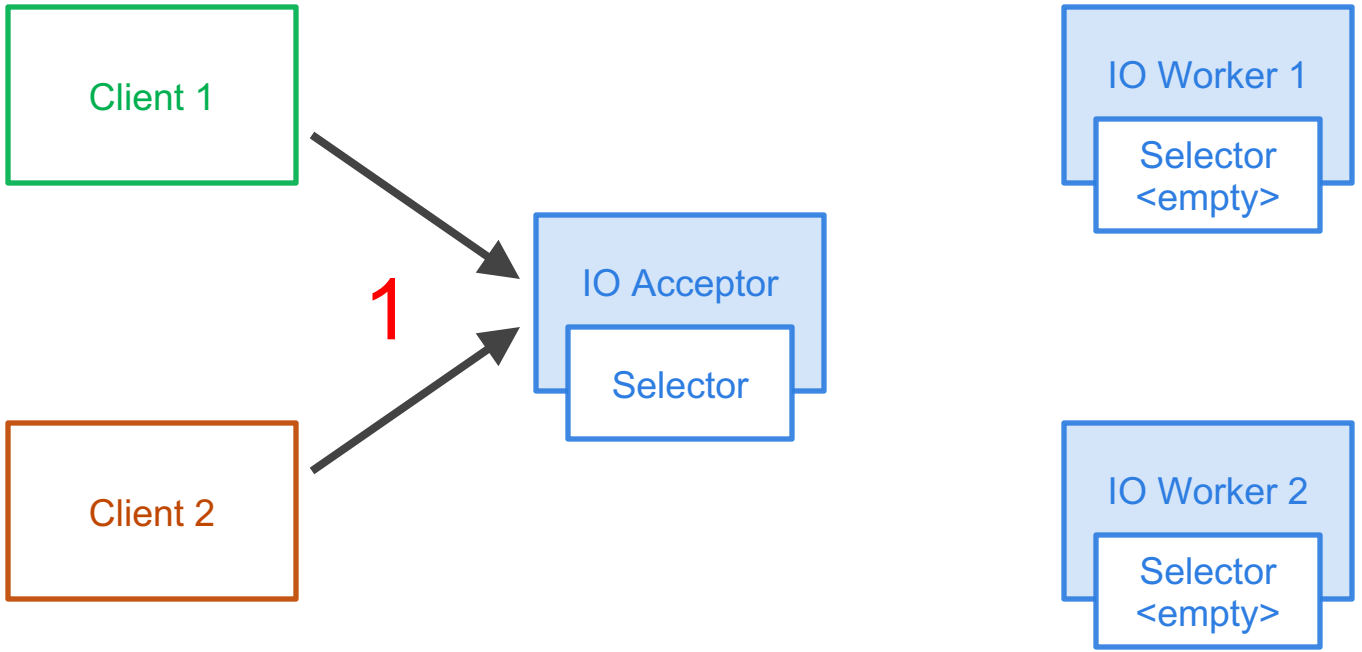
hazelcast

Реализация IO: соединение (1)



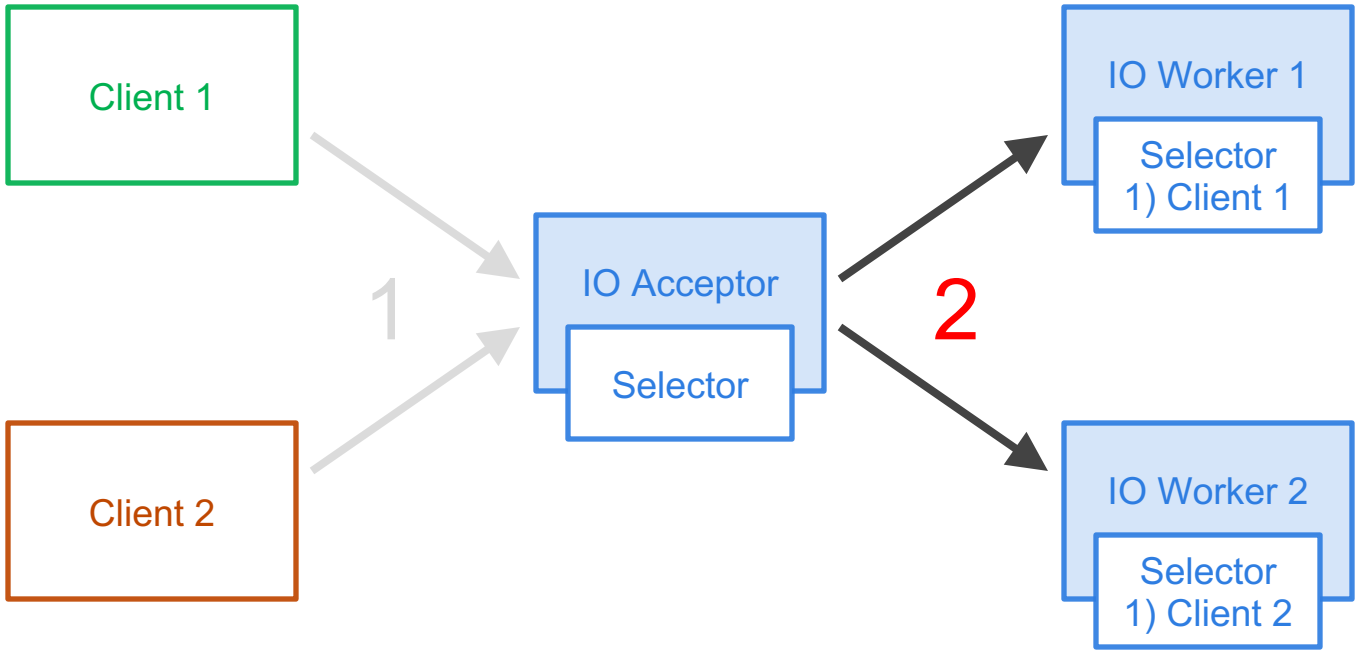
hazelcast

Реализация IO: соединение (1)



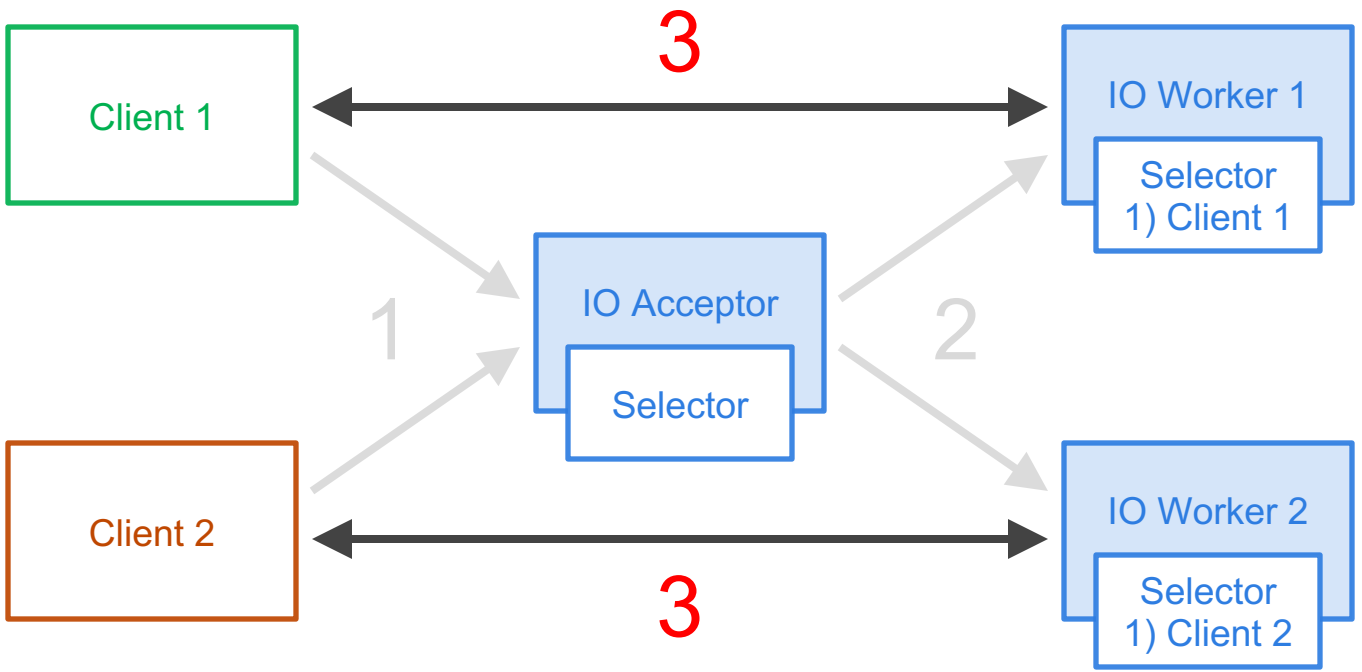
hazelcast

Реализация IO: соединение (2)



hazelcast

Реализация IO: нормальная работа (3)



hazelcast

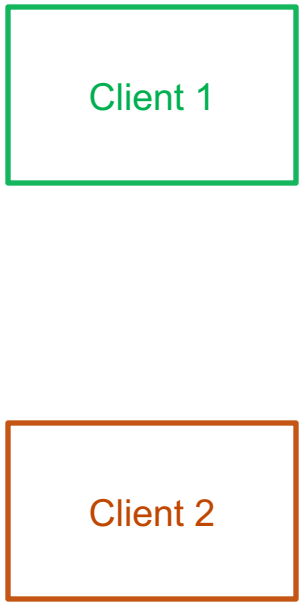
Реализация IO

1. **Acceptor thread** «слушает» входящие подключения, создает **SocketChannel** при подключении клиента

2. **Acceptor thread** выбирает **IO worker** с помощью round-robin алгоритма, передает ему **SocketChannel**

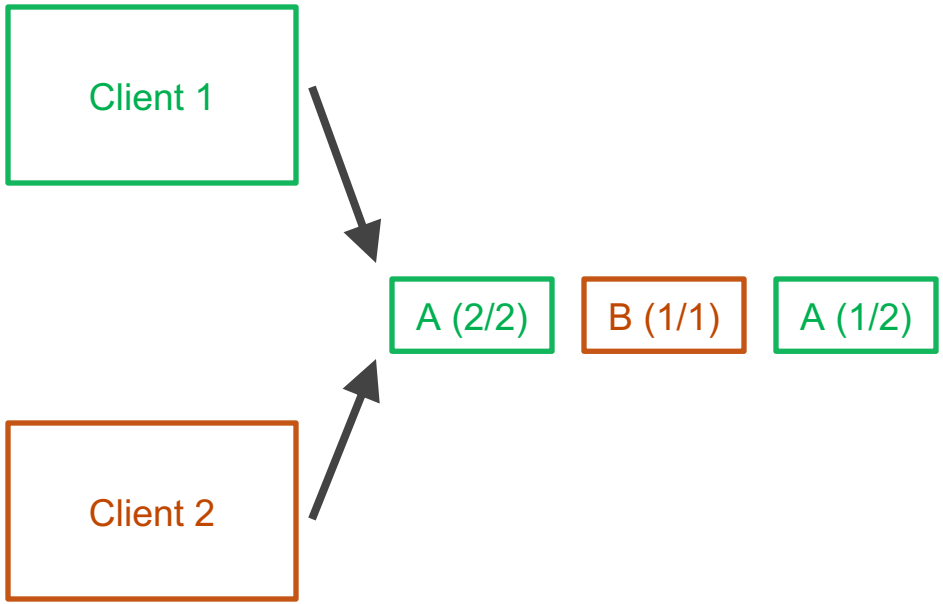
3. **IO Worker** регистриструет **SocketChannel** в своем **Selector**, и обеспечивает дальнейшую коммуникацию

Входящие запросы



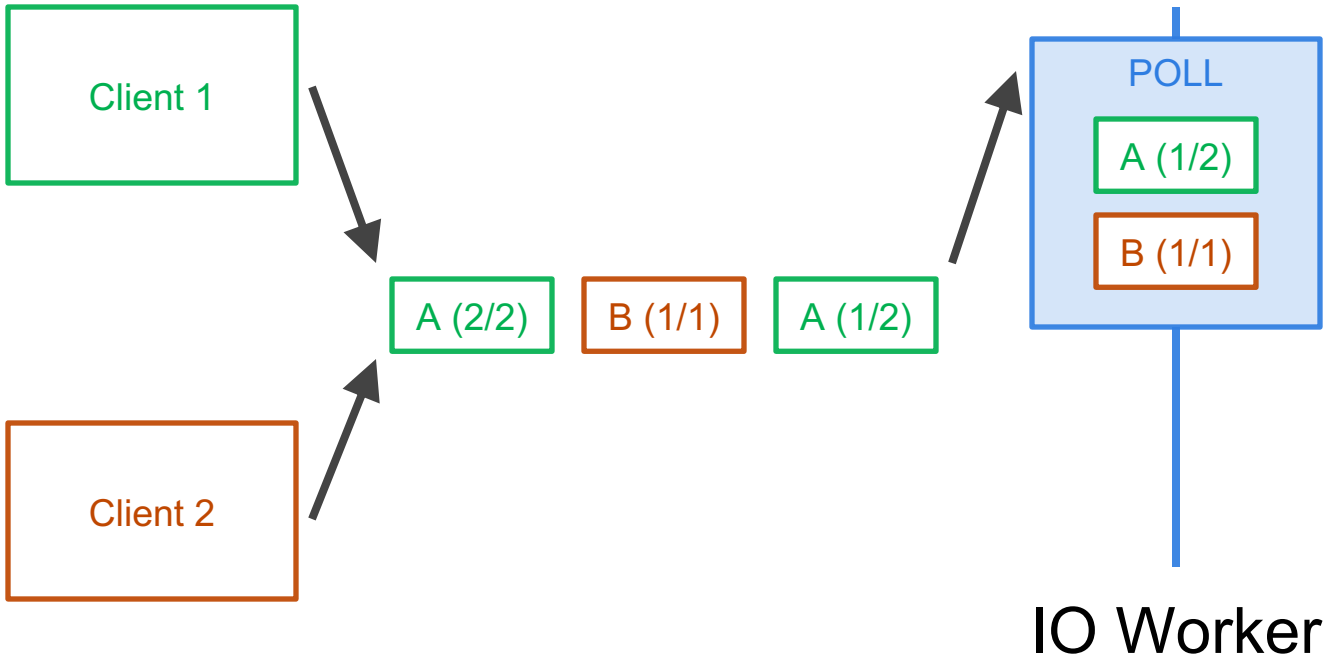
hazelcast

Входящие запросы



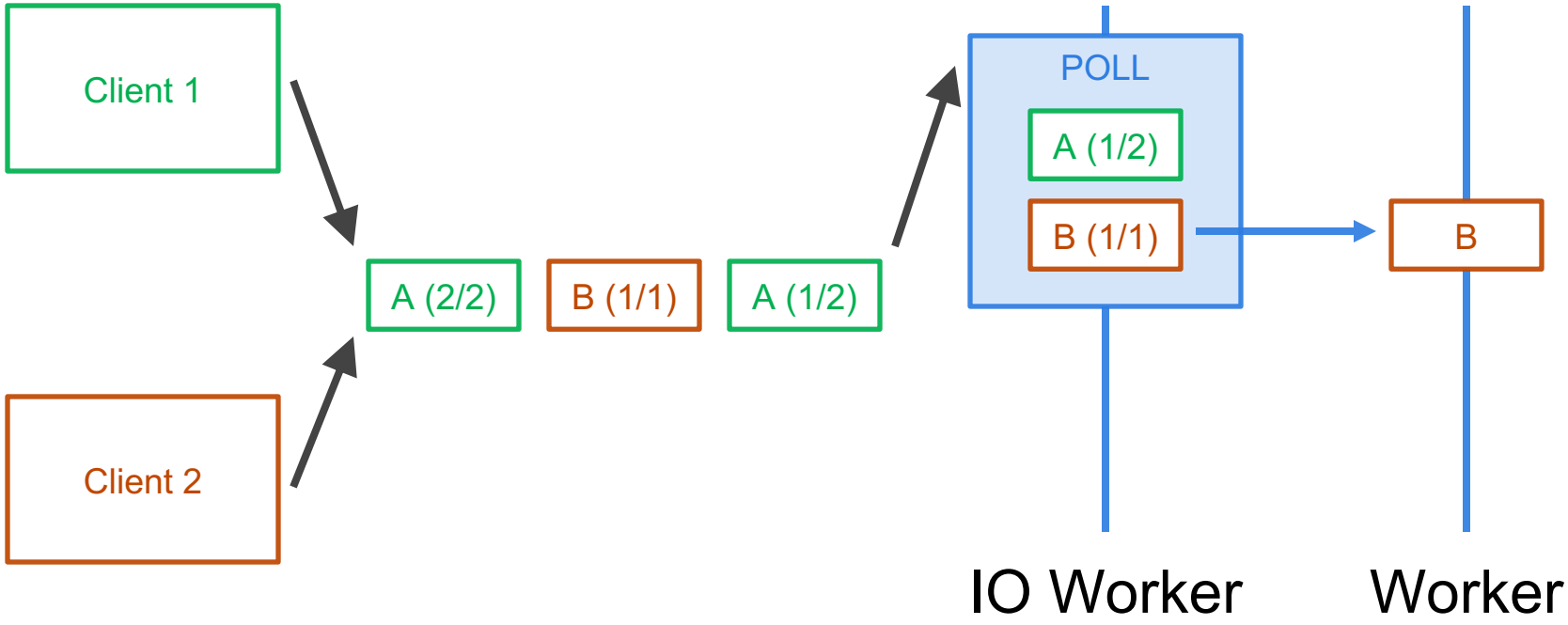
hazelcast

Входящие запросы



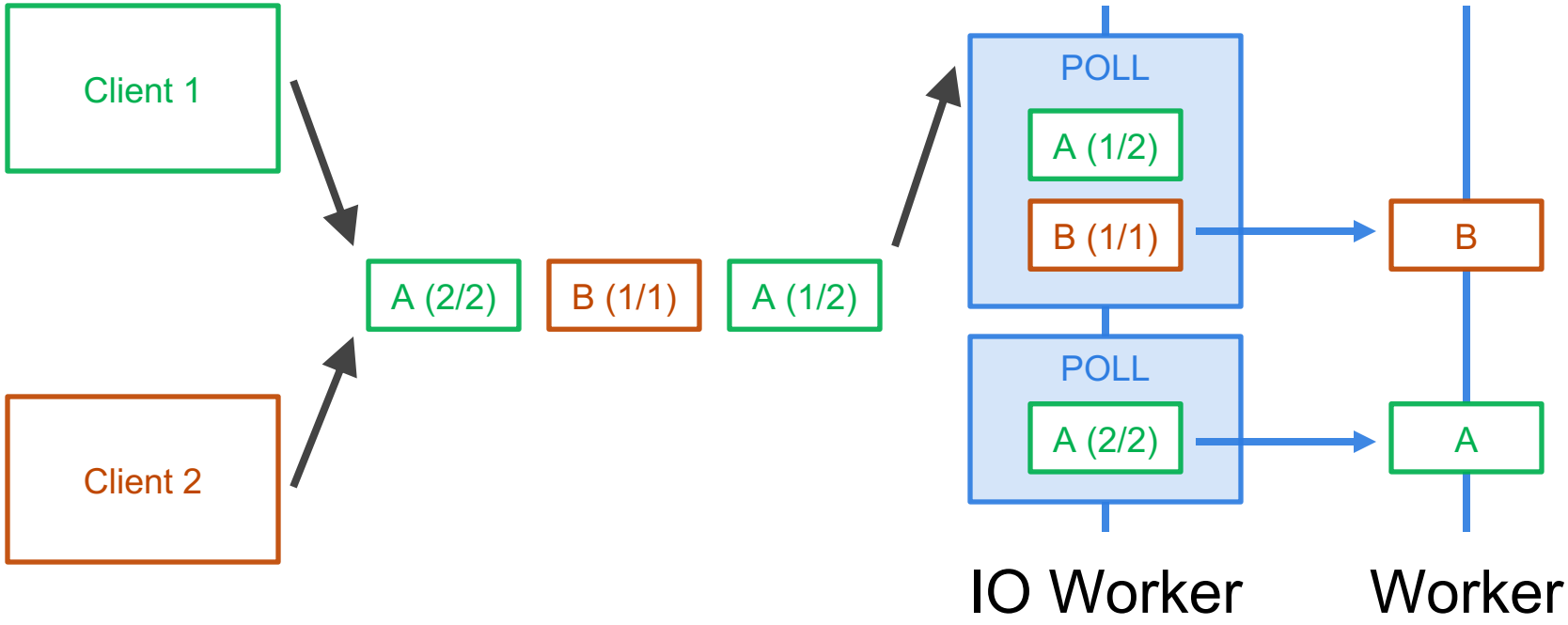
hazelcast

Входящие запросы



hazelcast

Входящие запросы



hazelcast

Входящие запросы

- Каждое сообщение содержит метаинформацию
  - **Длина**
  - **Тип собщения**

- Собираем сообщение из фрагментов, используя **длину**

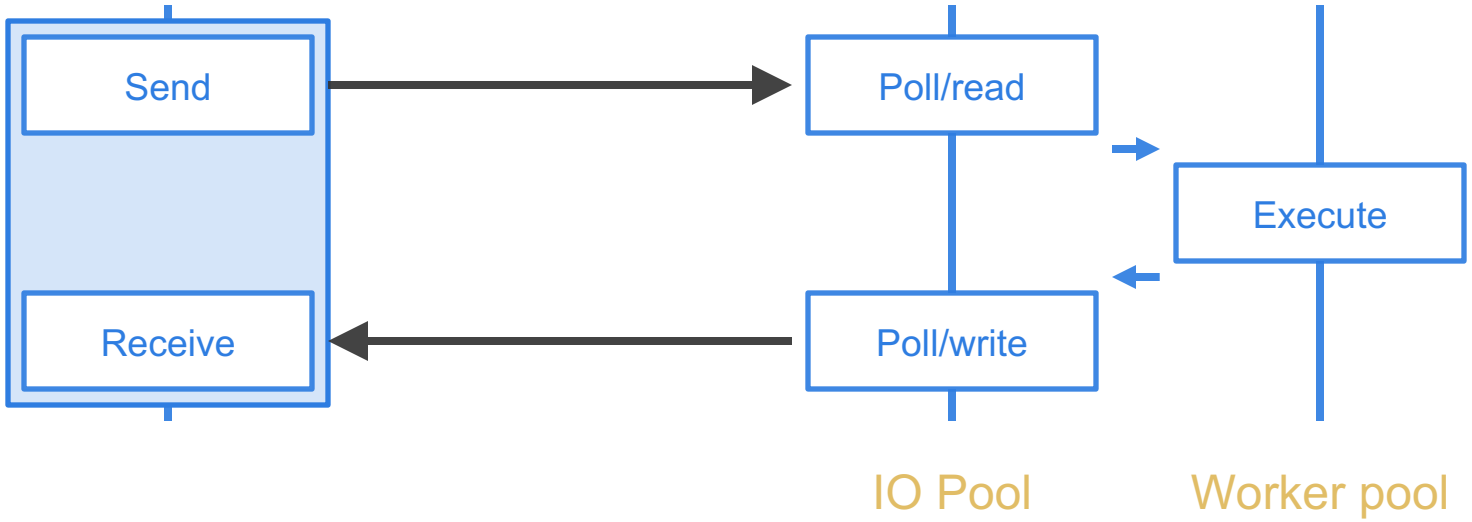
- Определяем стадию на основе **типа сообщения**, передаем задачу

Особенности

- Для обслуживания даже большого количества соединений достаточно небольшого количества IO workers (3-4 по умолчанию)
  - Потому что он только копирует байты и делает неблокирующие системные вызовы

- Соединения **перебалансируются** между worker-ами в случае возникновения дисбаланса

Блокирующий клиент

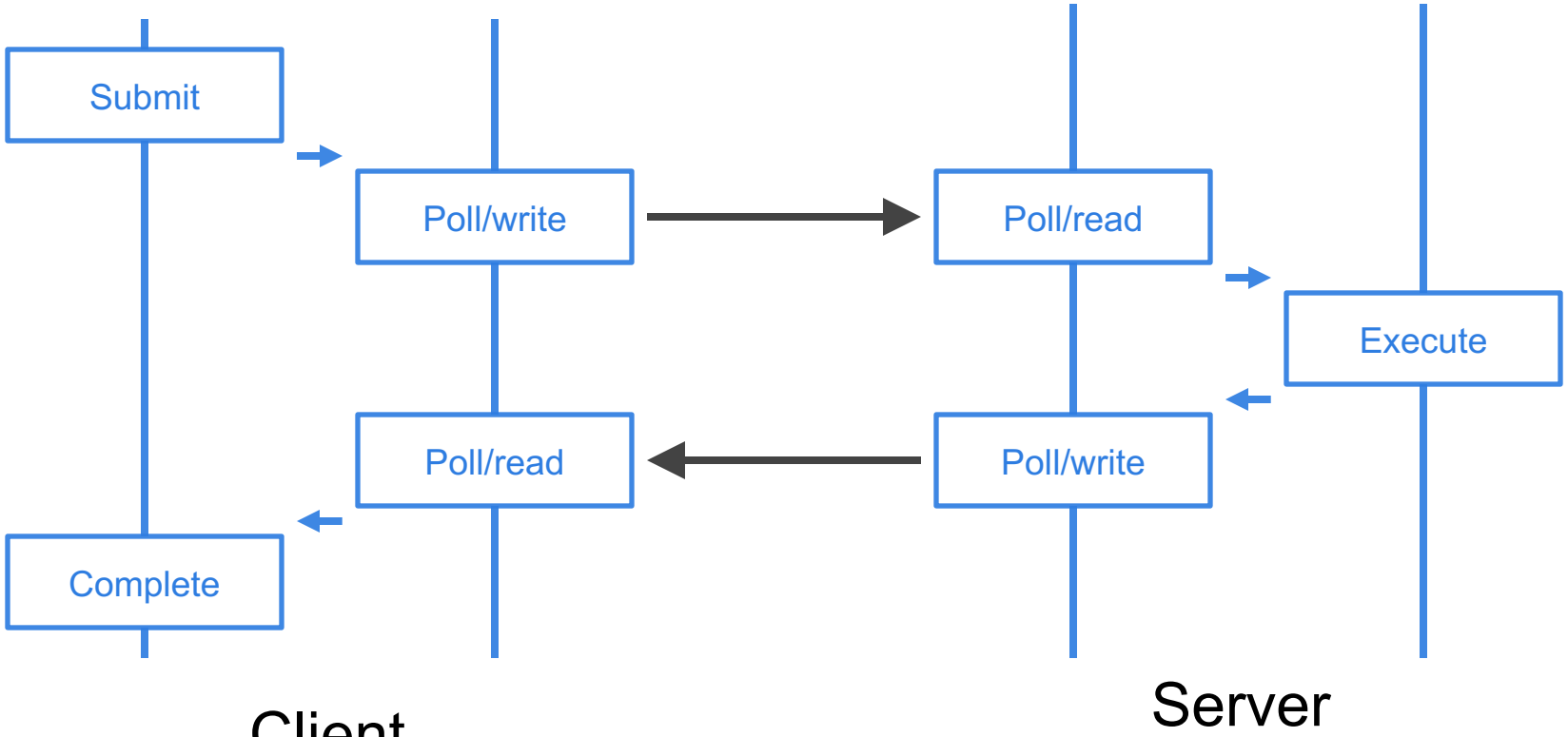


hazelcast

Мультиплексирование на клиенте

- Позволяет реализовать асинхронный API
  - `CompletableFuture<Void> fut = putAsync(key, value)`
- Достаточно одного соединения с сервером для обработки всех запросов

Мультиплексирование на клиенте

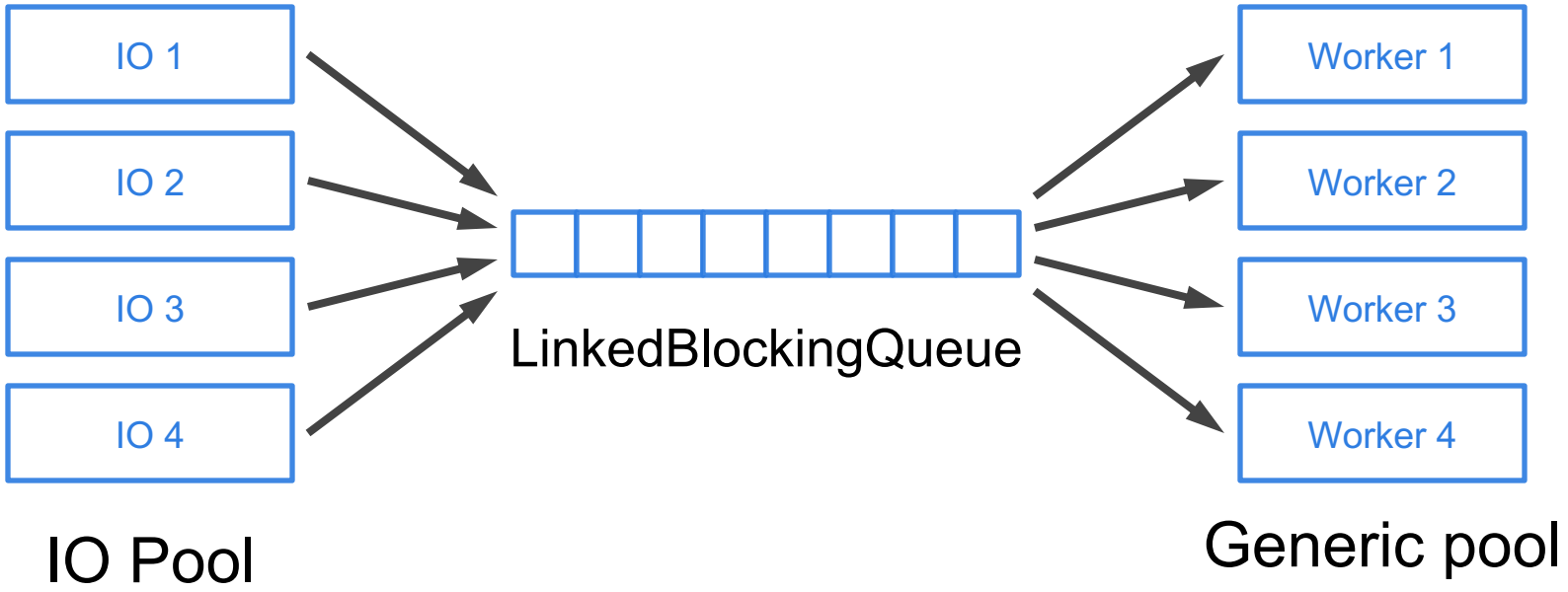


hazelcast

Архитектура стадий Hazelcast

- **Generic pool** – произвольные операции
- **Thread-per-partition** – key-value операции
- **Jet pool** – stream processing

Generic pool

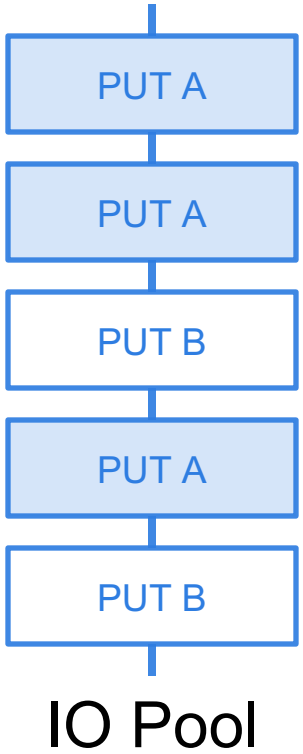


hazelcast

Generic pool

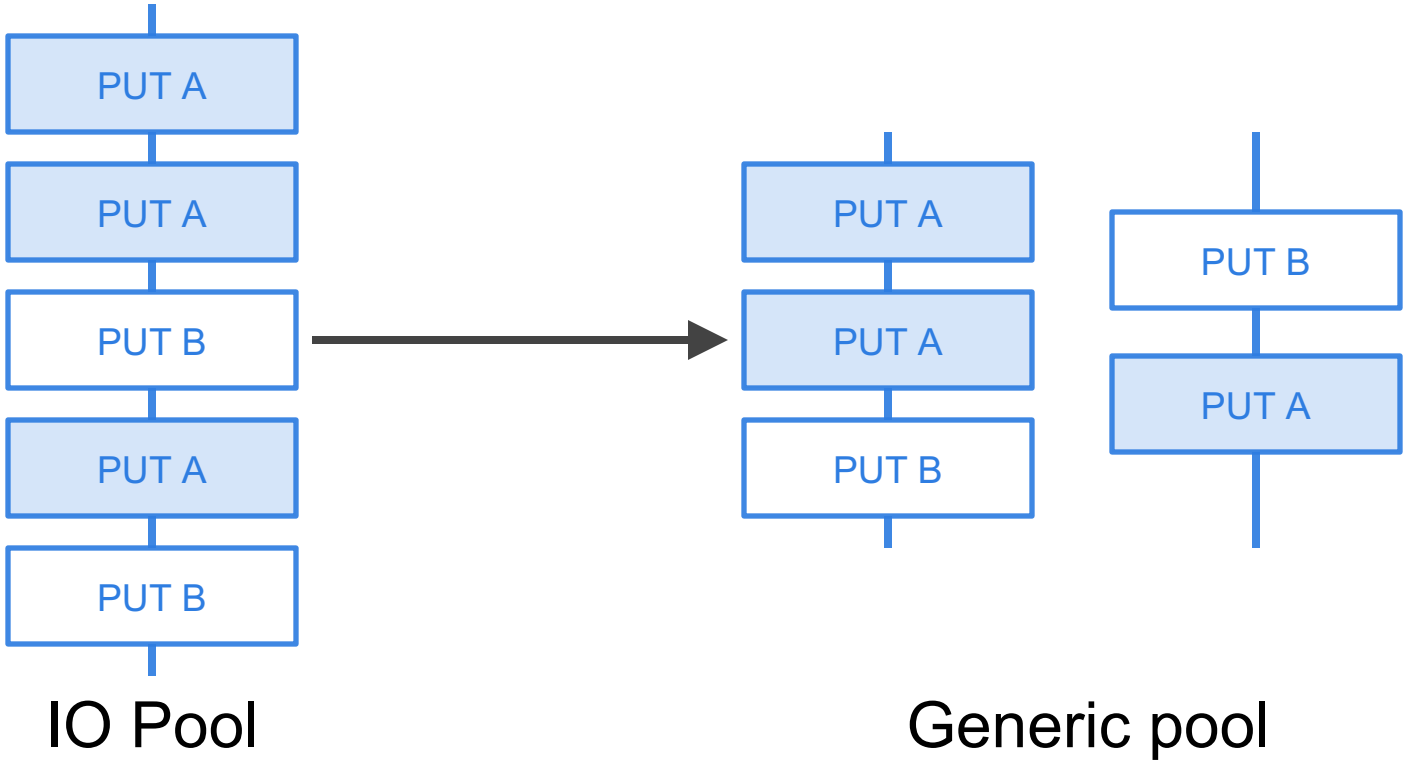
- Multiple-producer, multiple-consumer
- Подходит для выполнения произвольных операций пользователя

Как обрабатывать PUT/GET запросы?



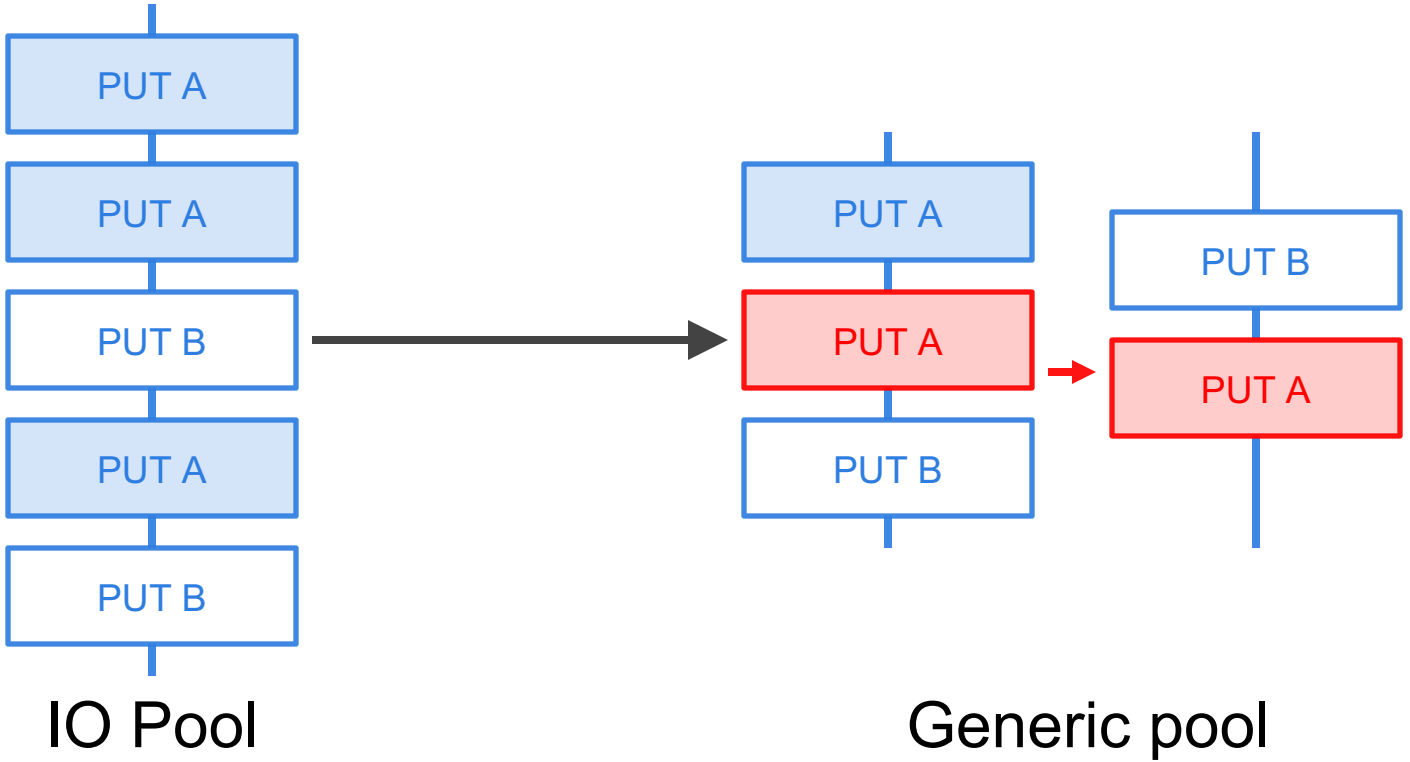
hazelcast

Как обрабатывать PUT/GET запросы?



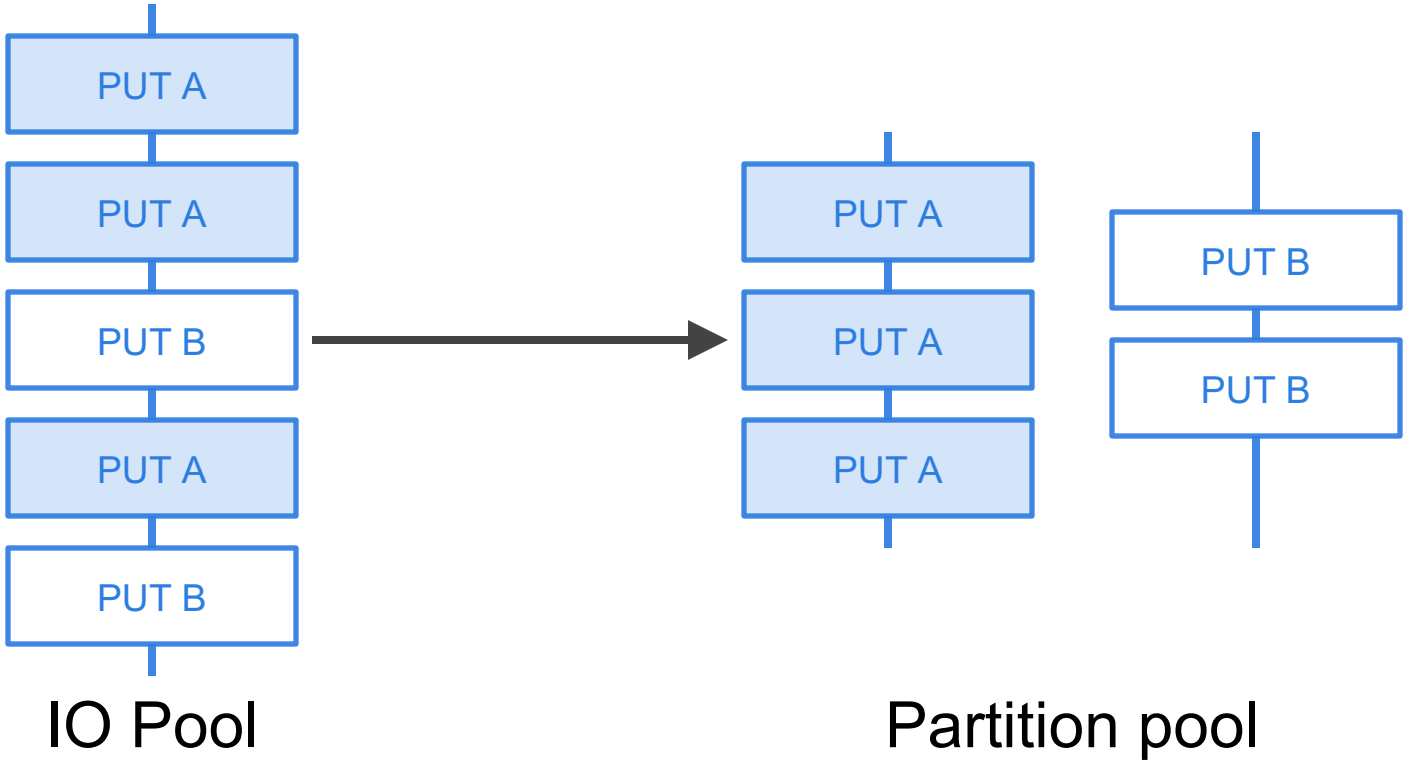
hazelcast

Как обрабатывать PUT/GET запросы?



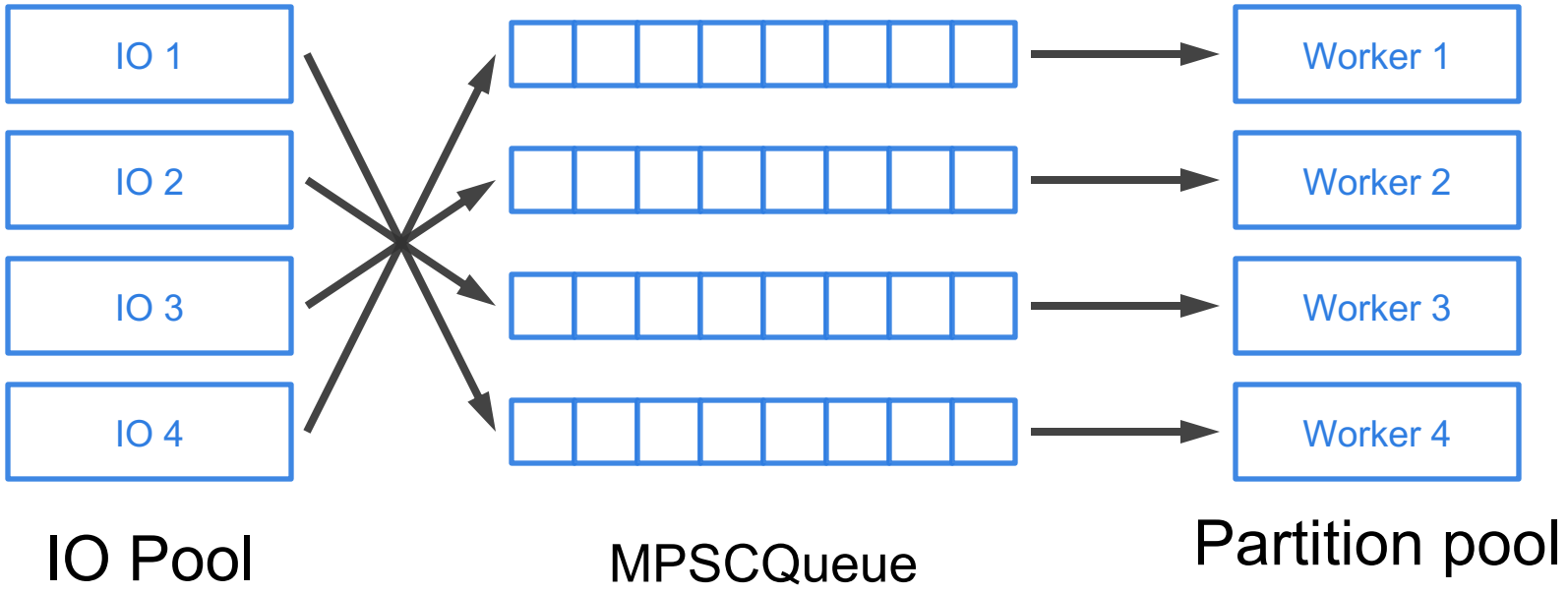
hazelcast

Thread-per-partition



hazelcast

Generic pool



hazelcast

Thread-per-partition

- Данные разделены на partitions
- Каждый partition «привязан» строго к одному worker
- Каждый worker имеет отдельную очередь (MPSCQueue)
  - **M**ultiple **P**roducer – много IO потоков передают задачи
  - **S**ingle **C**onsumer – только текущий worker забирает задачи из очереди

Thread-per-partition

Преимущества:

- Не нужно синхронизировать доступ к данным
- Меньше contention на очередях

Thread-per-partition

Преимущества:

- Не нужно синхронизировать доступ к данным
- Меньше contention на очередях

Недостатки

- Предполагает сбалансированную нагрузку на все partition-ы
- Плохо подходит для операций, которым нужны данные из нескольких partition-ов (требует map-reduce)

Структуры данных

Heap:

- `ConcurrentHashMap` (+ вариации)
- `ConcurrentSkipList`

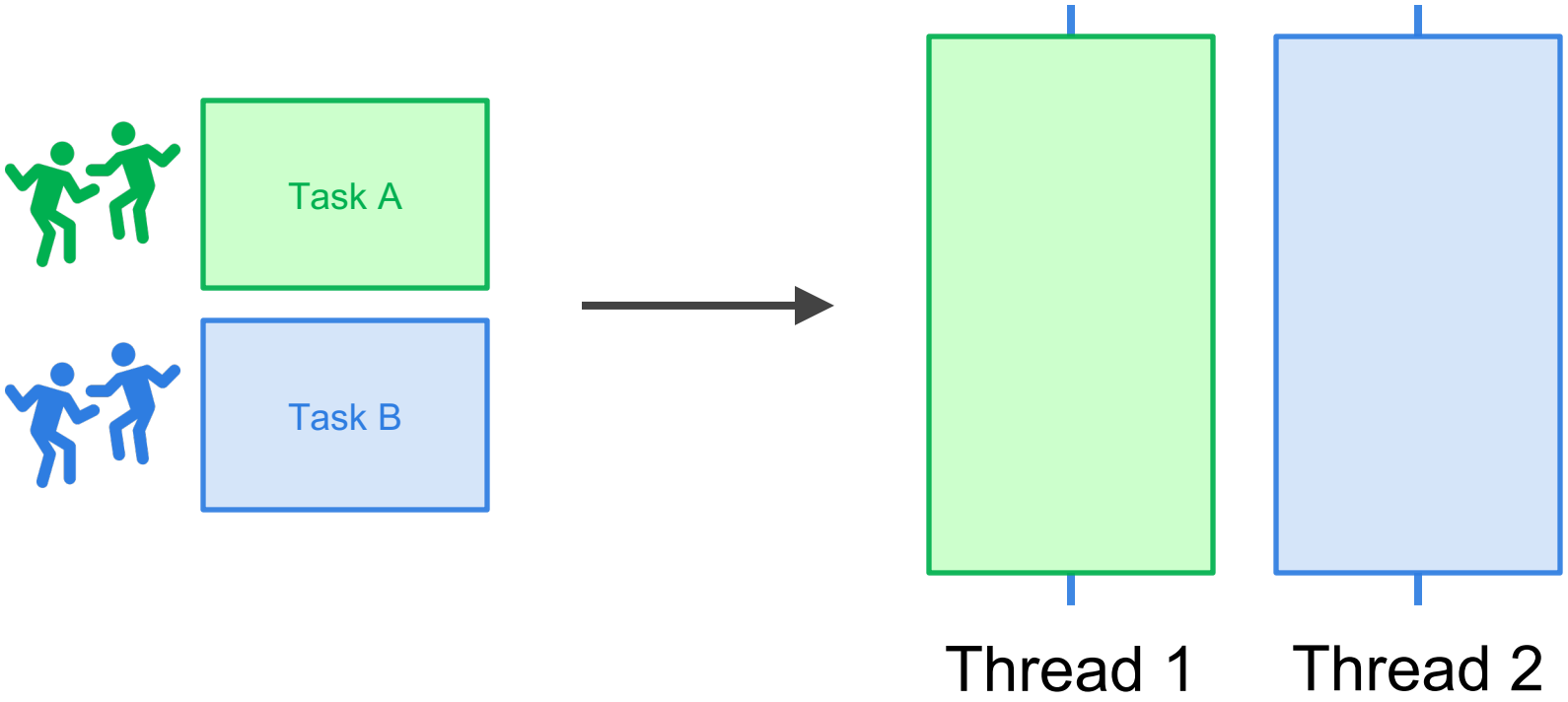
Offheap:

- Однопоточный hash map с открытой адресацией
- Однопоточное red-black tree

Jet pool

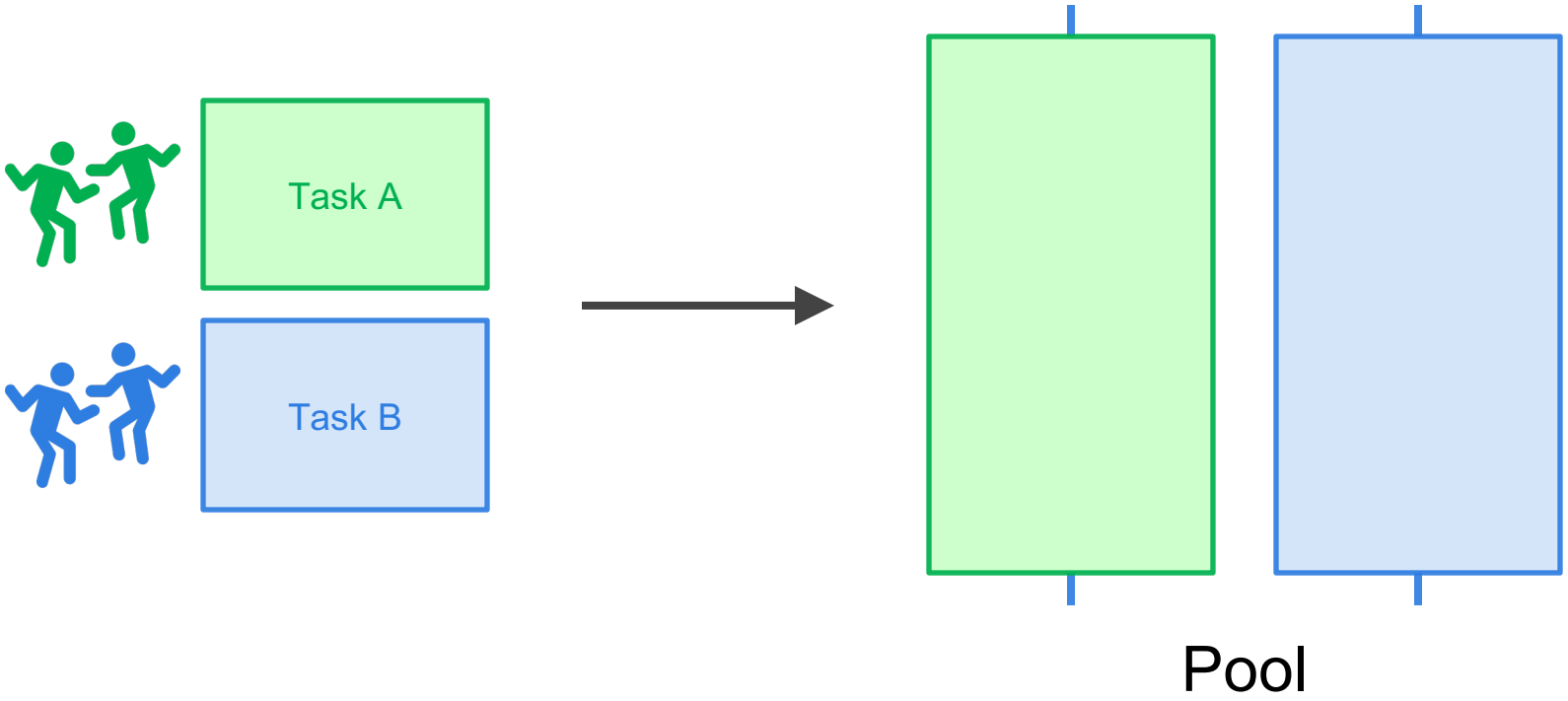
- **Hazelcast Jet –** stream processing framework
- Streaming задачи выполняются потенциально **неограниченное** время
- Как гаранитровать их прогресс, не создавая отдельный поток на каждую задачу?

Поток на задачу



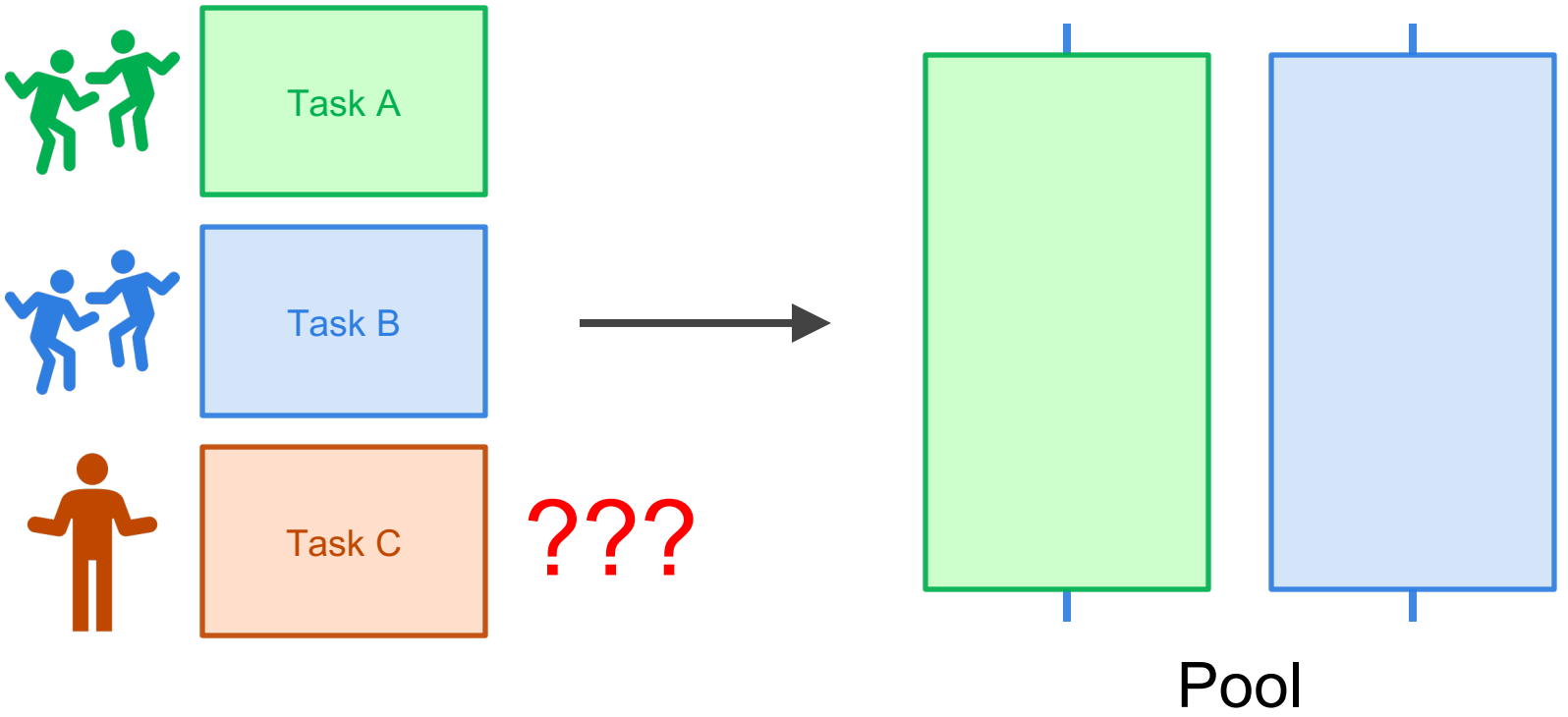
hazelcast

Последовательное выполнение в пуле



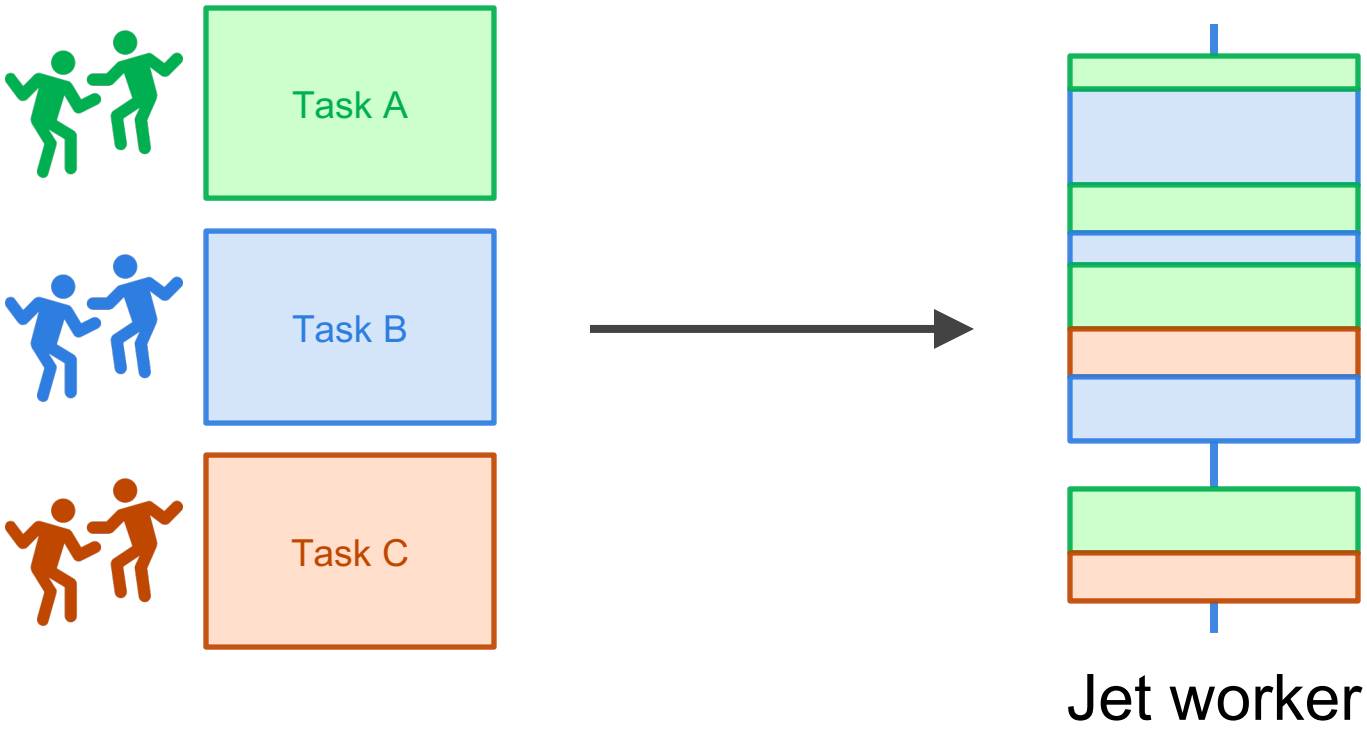
hazelcast

Последовательное выполнение в пуле



hazelcast

Jet pool: кооперативный пул



hazelcast

Jet pool: кооперативный пул

- Итерируемся по списку задач, и выполняем их последовательно

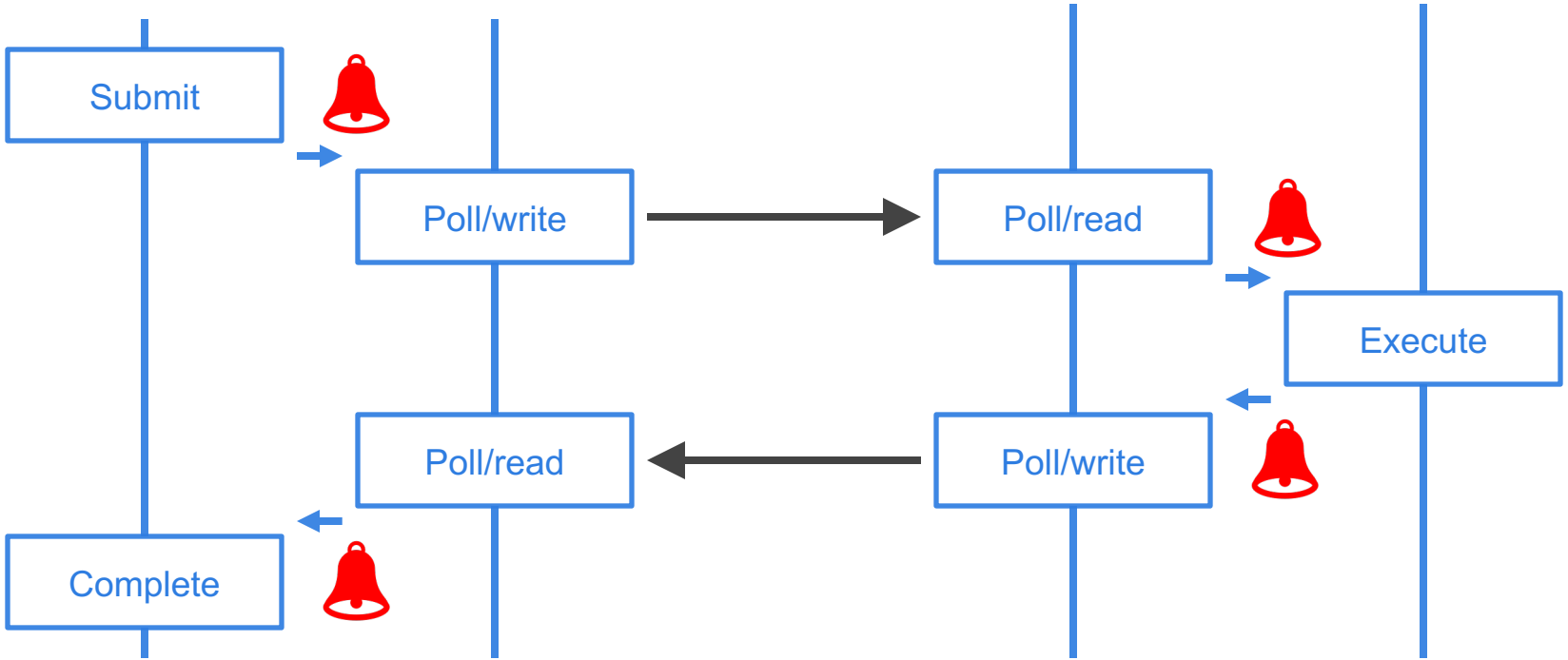
- Прекращаем выполнение задачи, если:
  - Нет входных данных (еще не пришли)
  - Не можем записать выходные данные (нет места)
  - Задача завершена

Jet pool: кооперативный пул

```
enum ProgressState {
    MADE_PROGRESS,
    NO_PROGRESS,
    DONE
}
```


hazelcast

Проблема: нотификации

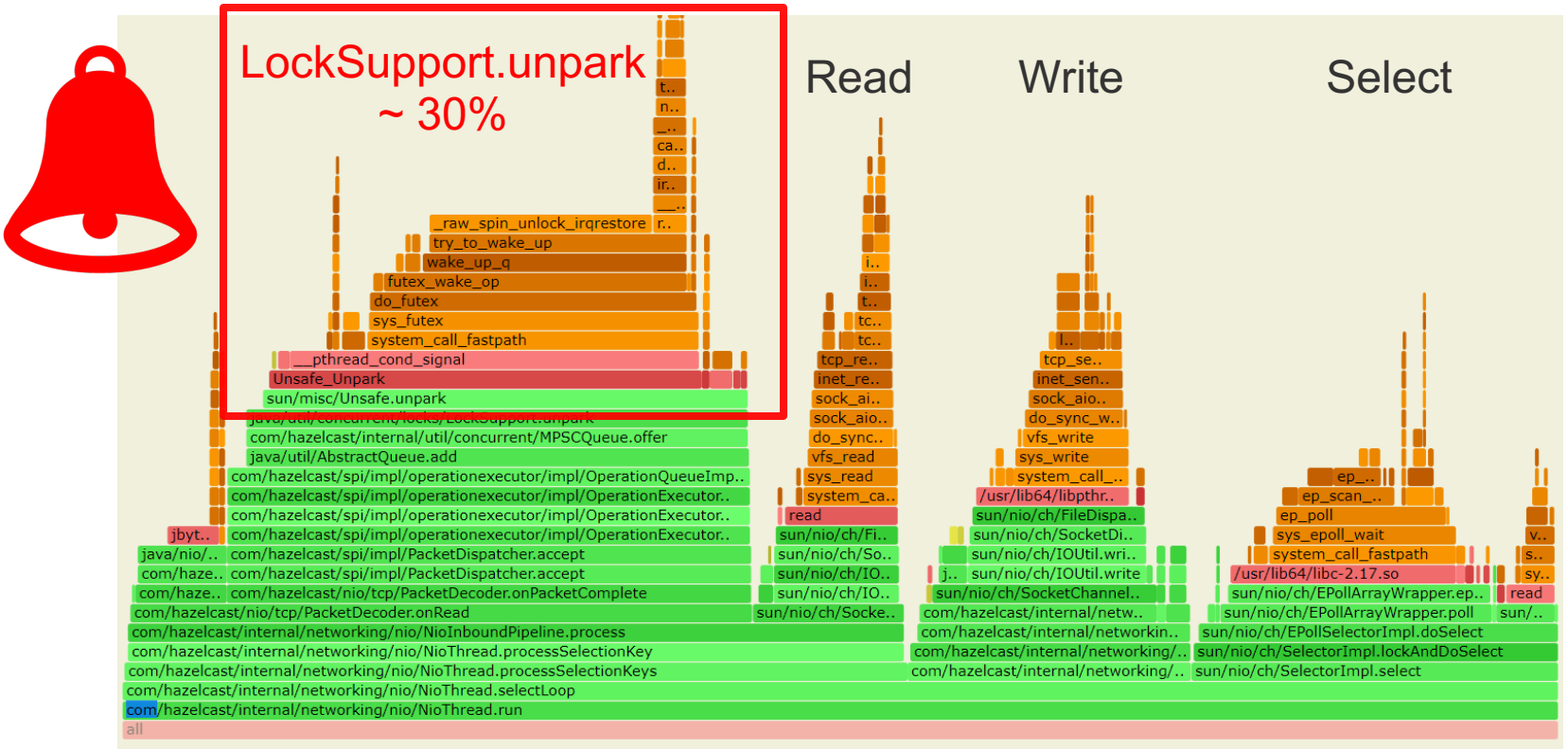


hazelcast

Нотификации: IO -> stage

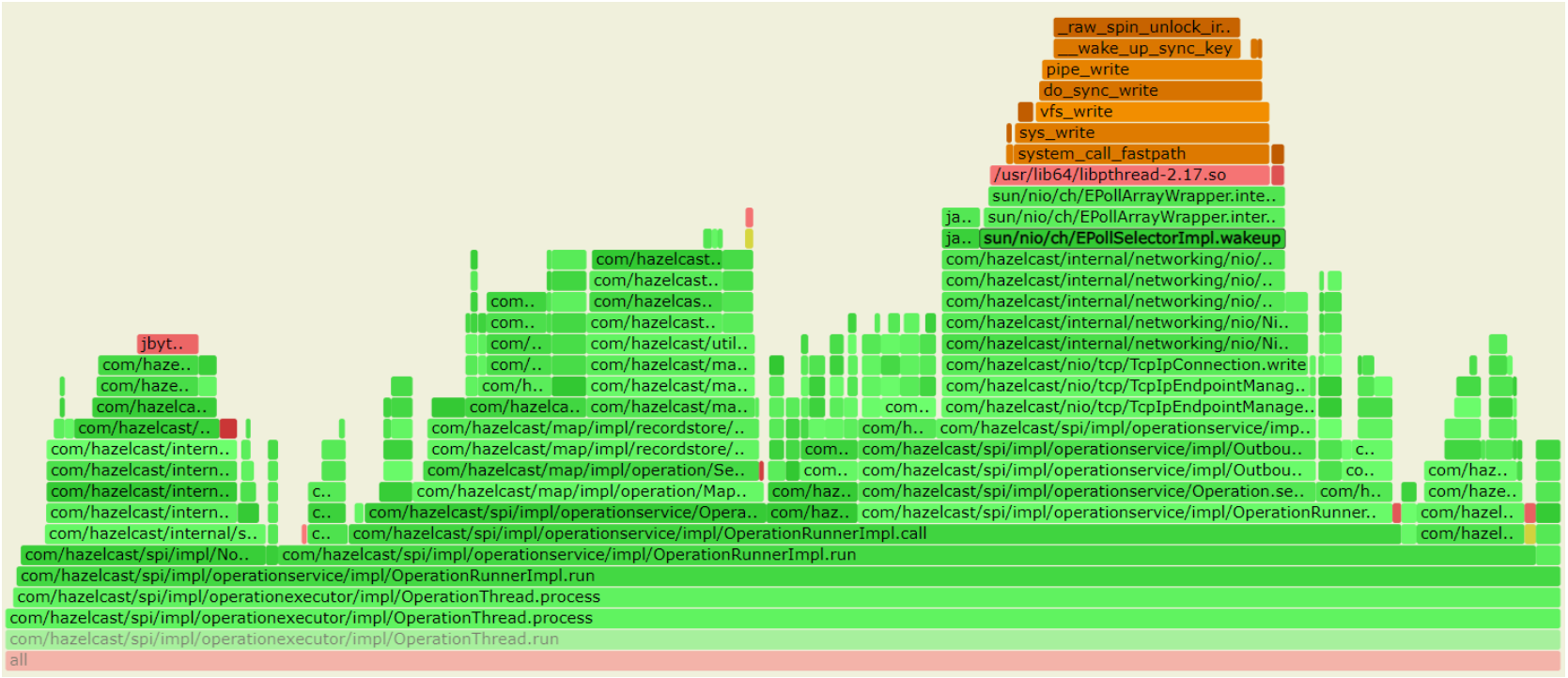
hazelcast

Нотификации: IO -> stage



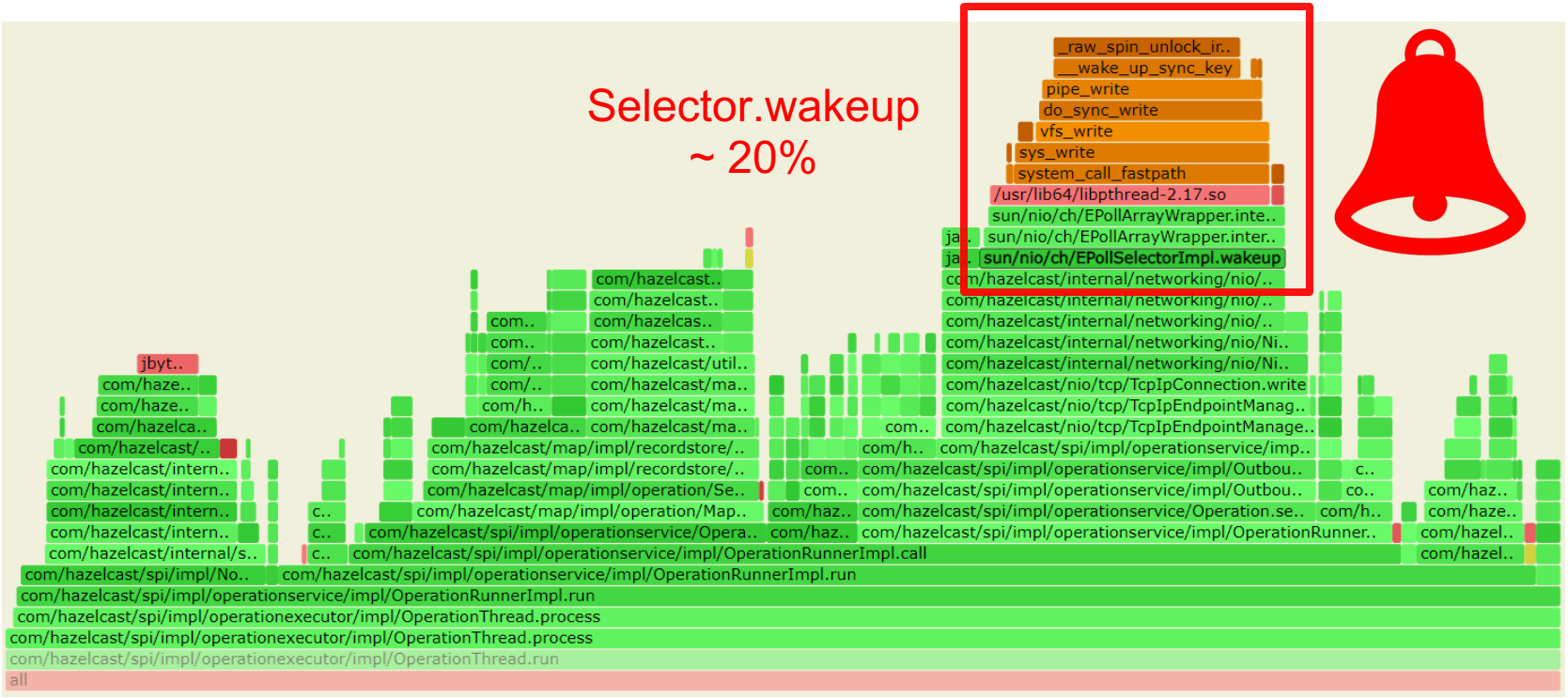
hazelcast

Нотификации: stage -> IO



hazelcast

Нотификации: stage -> IO



hazelcast

Нотификации: улучшаем

- Меньше нотификаций
  - Batching: одна нотификация на несколько задач
  - Не нотифицировать активный поток
- Меньше стадий
  - Нет потока – нет нотификации
- “Mechanical sympathy”
  - Выбор количества потоков
  - NUMA

Чтение из сети: плохо

```
while ((Packet packet = poll(selector)) != null) {
    Worker worker = getWorker(packet);

    worker.submit(packet); // <= Unpark
}
```

hazelcast

Чтение из сети: хорошо

```
List<Packet> packets = pollAll(selector);

Map<Worker, List<Packet>> mappedPackets =
    mapPackets(packets);

mappedPackets.forEach((worker, packets) -> {
  worker.submitAll(packets); // <= Unpark
});
```

hazelcast

Нотификация IO потока: плохо

```
ConcurrentLinkedQueue<Packet> queue;

void submitPacket(Packet packet) {
    queue.offer(packet);

    selector.wakeup();
}
```

Нотификация IO потока: хорошо

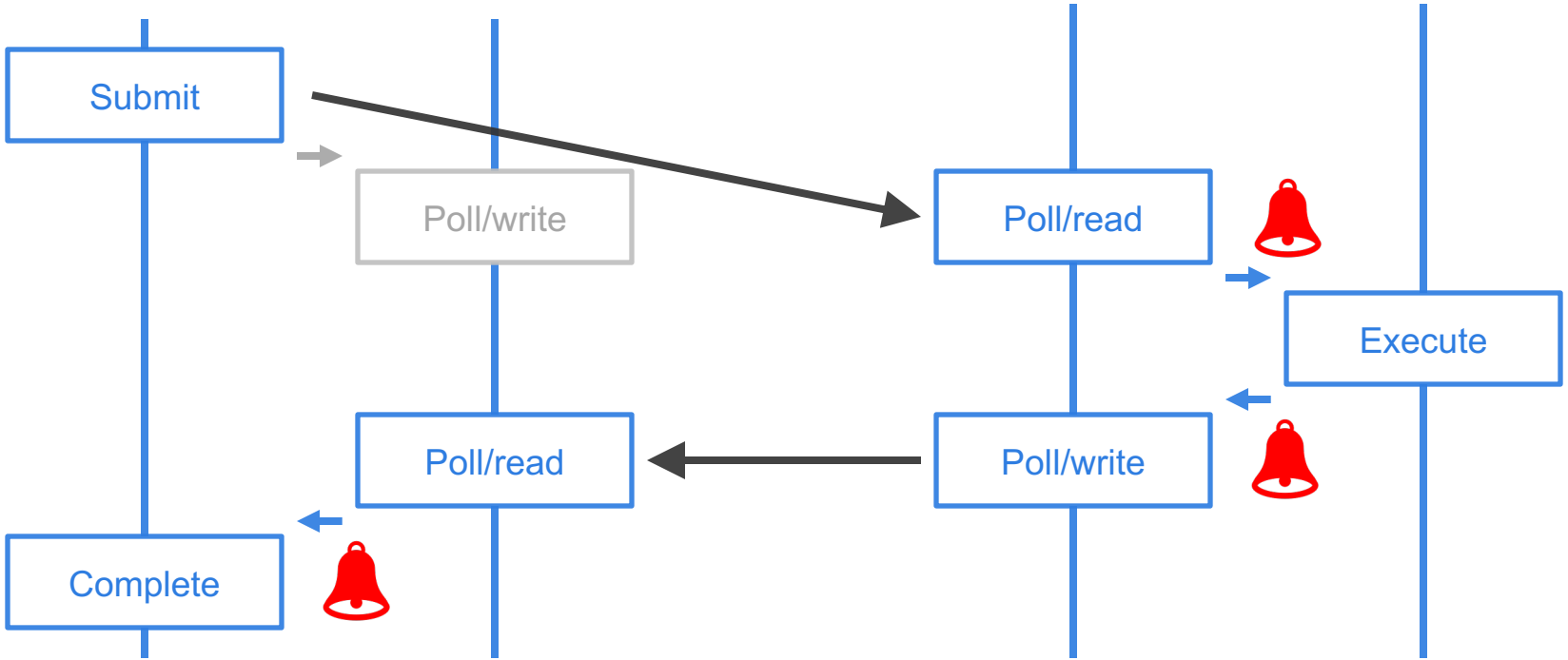
```
ConcurrentLinkedQueue<Packet> queue;
AtomicBoolean active;

void submitPacket(Packet packet) {
    queue.offer(packet);

    if (!active.get() && active.cas(false, true))
        selector.wakeup();
}
```

hazelcast

Меньше стадий



hazelcast

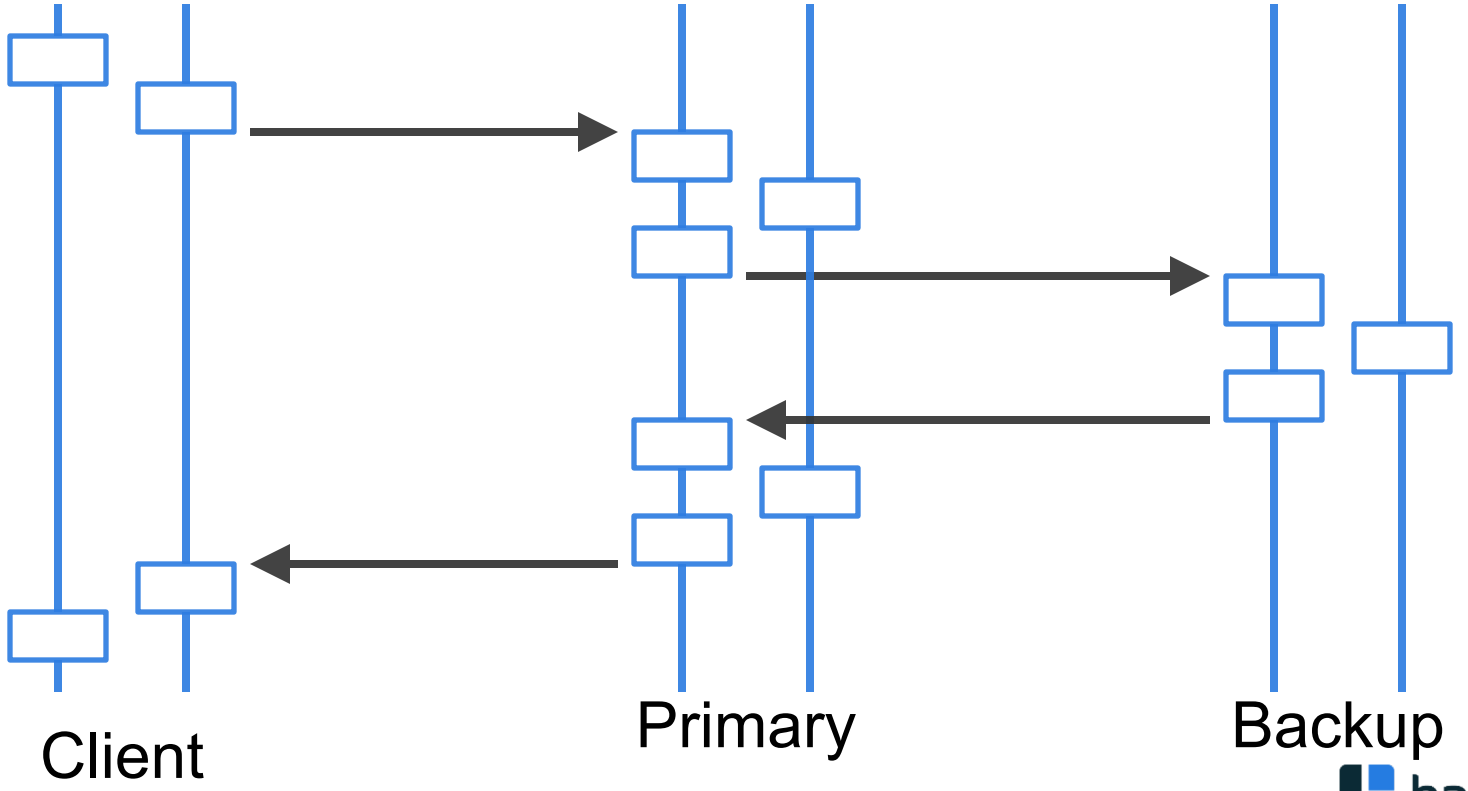
Меньше стадий

```
SocketChannel channel;
AtomicBoolean owner;

void sendPacket(Packet packet) {
    if (!owner.get() && owner.cas(false, true)) {
        send(channel, packet);
        owner.set(false);
    } else
        submitToIOWorker(packet);
}
```

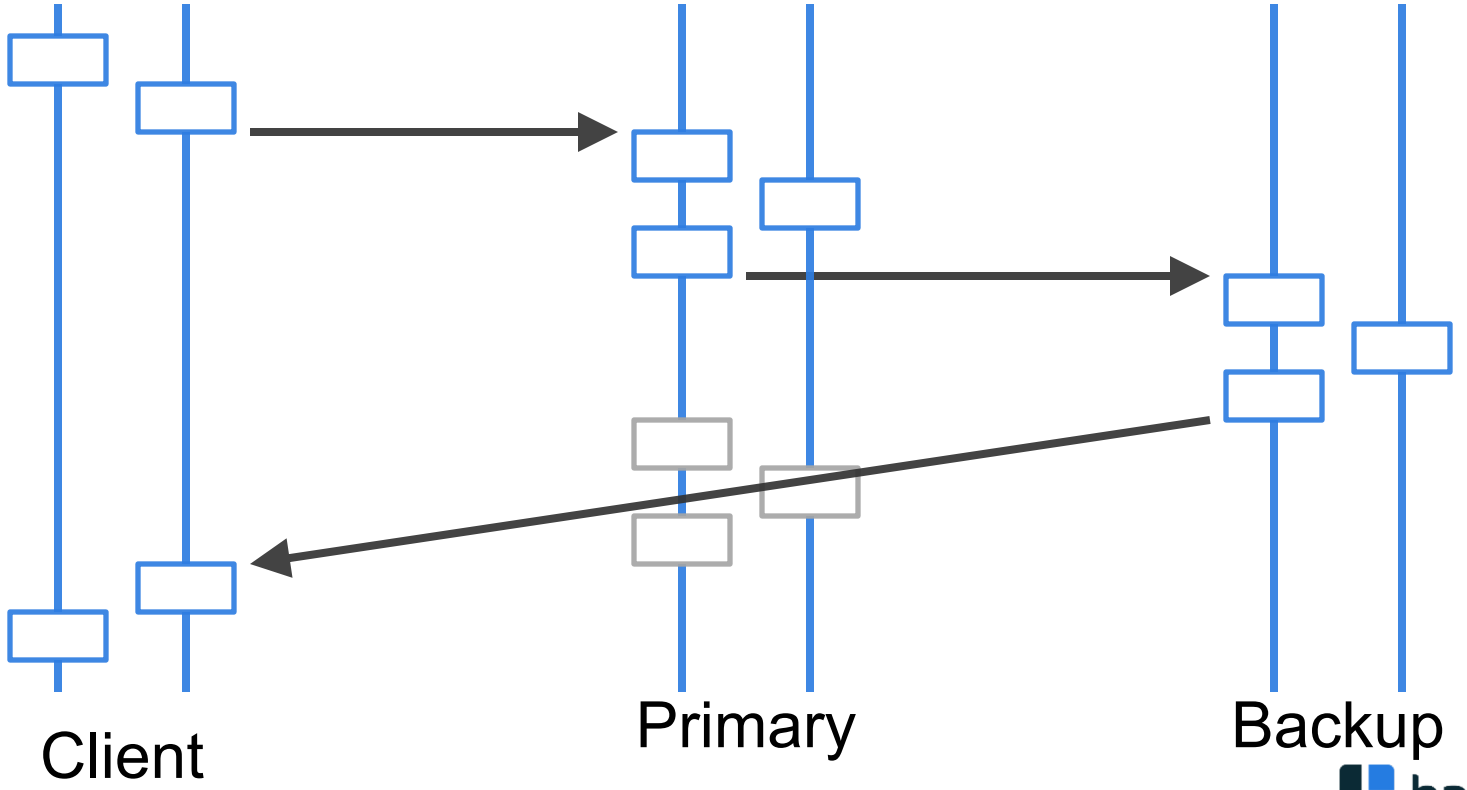
hazelcast

Меньше стадий: backup



hazelcast

Меньше стадий: backup notification



hazelcast

Количество потоков

- Продукты часто стартуют слишком много worker-потоков по умолчанию

```
== Runtime.availableProcessors()
```

- Результат: конкуренция за CPU, медленный доступ к памяти

Пример: NUMA

- AWS, c4.8xlarge, 36 vCPU, 2 NUMA nodes

Пример: NUMA

- AWS, c4.8xlarge, 36 vCPU, 2 NUMA nodes
- `2 сокета, 36 потоков: 390_000 ops/sec`

hazelcast

Пример: NUMA

- AWS, c4.8xlarge, 36 vCPU, 2 NUMA nodes

- 2 сокета, 36 потоков: **390_000** ops/sec
- 1 сокет, 36 потоков: **520_000** ops/sec

```
|    Runtime ms   | Switches | Average delay ms |
-------------------------------------------------
| 136527.294 ms |  2402220 | avg:    0.025 ms |
| 114643.330 ms |  4058827 | avg:    0.011 ms |
```


hazelcast

Пример: oversubscription

- 2 сокета, 36 потоков: **390_000** ops/sec
- 1 сокет, 36 потоков: **520_000** ops/sec
- 4 потока : **550_000** ops/sec

Runtime ms	Switches	Average delay ms
136527.294 ms	**2402220**	**avg: 0.025 ms**
114643.330 ms	**4058827**	**avg: 0.011 ms**
56609.212 ms	**1200057**	**avg: 0.007 ms**

Итого: как выполнять IO операции?

- Вариант 1: blocking IO
  - Хорошо для простых и ненагруженных сценариев
  - Простая реализация
  - Не масштабируется

- Вариант 2: event loop (non-blocking IO)
  - Хорошо масштабируется
  - Сложнее

Итого: где выполнять задачи?

- Вариант 1: все задачи в event loop
  - Идеально подходит для небольших однотипных задач
  - Проблемы при разнородной нагрузке

- Вариант 2: event loop + стадии (SEDA)
  - Универсальный и гибкий подход
  - Накладные расходы при передаче задач между потоками

Итого: где выполнять задачи?

- Вариант 1: все задачи в event loop
  - Идеально подходит для небольших однотипных задач
  - Проблемы при разнородной нагрузке

- Вариант 2: event loop + стадии (SEDA)
  - Универсальный и гибкий подход
  - Накладные расходы при передаче задач между потоками

- **Не являются взаимоисключающими!**

Итого: доступ к данным

- Вариант 1: shared
  - Подходит для больших задач (например, SQL)
  - Накладные расходы на координацию доступа к данным

- Вариант 2: shared nothing (sharding)
  - Подходит для маленьких задач (например, PUT/GET, OLTP)
  - Накладные расходы при доступе к нескольким шардам

Итого: архитектура Hazelcast

- Кратко:
  - Event loop + стадии (SEDA)
  - Преимущественно shared nothing

- Пулы потоков под разные задачи:
  - **IO pool** – event loop для IO операций
  - **Generic pool** – классический пул для compute задач
  - **Partition pool** – шардированный пул для key-value операций
  - **Jet pool** – кооперативный пул для стриминга

Итого: рекомендации пользователям

- Изучайте модели многопоточности, что бы лучше понимать, подходит ли продукт для ваших задач
  - Любой продукт – это набор компромиссов

- Правильно подбирайте количество потоков под окружение и профиль нагрузки
  - `numactl/taskset`

Контакты

@devozerov

hazelcast

Q&A

Вопросы?

hazelcast