

- История одной оптимизации производительности Node.js библиотеки

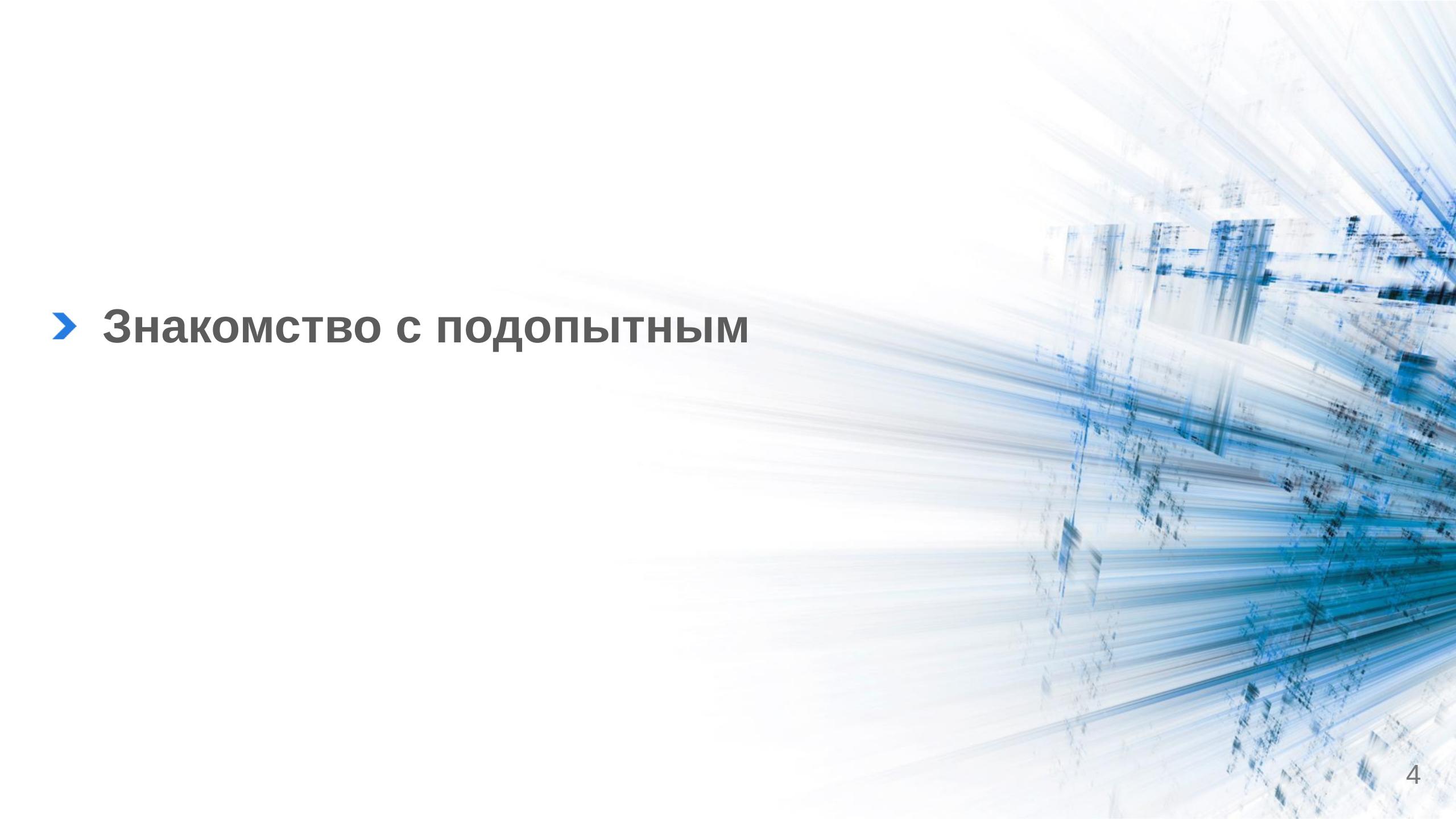
Андрей Печкуров

# О докладчике

- Пишу на Java (10+ лет), Node.js (5+ лет)
- Интересы: веб, архитектура, распределенные системы, производительность
- Можно найти тут:
  - <https://twitter.com/AndreyPechkurov>
  - <https://github.com/puzpuzpuz>
  - <https://medium.com/@apechkurov>

# О докладе

- Основные темы:
  - Подход к оптимизации производительности Node.js библиотек
  - История некоторых наших оптимизаций
  - Немного выводов
- Подопытный:
  - Клиентская Node.js библиотека Hazelcast IMDG
- Аудитория:
  - Все, кто разрабатывает сетевые приложения на Node.js



## ➤ Знакомство с подопытным



- 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>
- Node.js 4+
- Стэк: TypeScript, promisified API (bluebird)
- Библиотека еще довольно молода: первый стабильный релиз - май 2019

# Особенности библиотеки

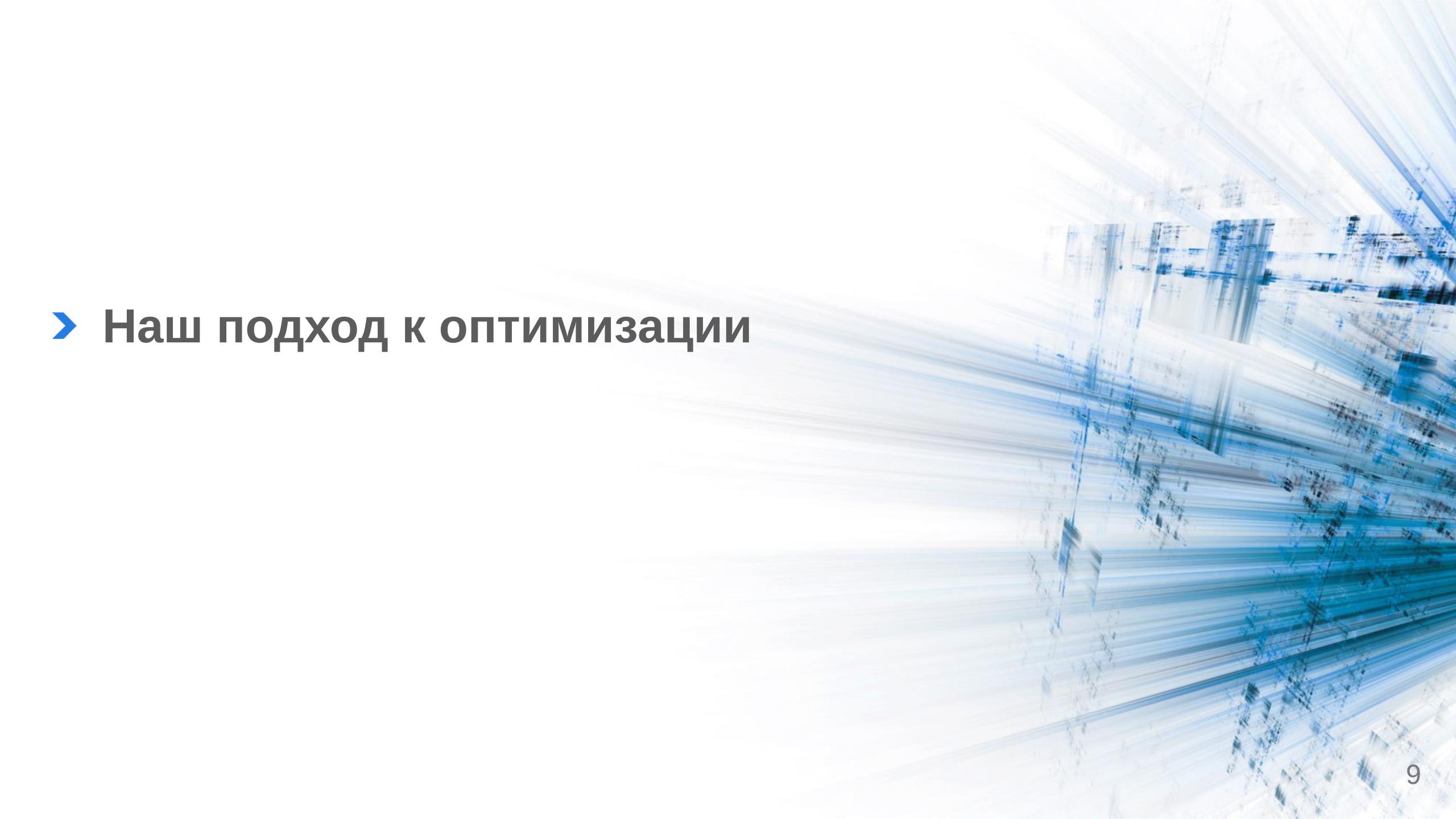
- "Умная" клиентская библиотека (знает топологию кластера, отправляет heartbeat'ы и т.д.)
- Общается с нодами кластера по [открытыму бинарному протоколу](#) поверх TCP
- Поддерживает множество распределенных структур данных
- Умеет near cache, retry on failure, client stats и многое другое
- Как следствие - относительно большая кодовая база (~34К строк TS кода, не учитывая тесты)

# Пример использования

```
const Client = require('hazelcast-client').Client;

const client = await Client.newHazelcastClient();
const cache = await client.getMap('my-awesome-cache');

await cache.set('foo', 'bar');
const cached = await cache.get('foo');
console.log(cached); // bar
```



## ➤ Наш подход к оптимизации

## Начальные цели

- Анализ текущей производительности перед стабильным релизом
- Включение в релиз "быстрых" правок (при необходимости)
- Постановка планов по дальнейшему анализу и оптимизации
- *Спойлер:* о текущих планах мы поговорим позже

# Оптимизация?



# Возможные метрики

- Сетевая клиентская библиотека
- I/O bound нагрузка
- Основные метрики:
  - Операции в секунду (throughput)
  - Время выполнения операции (условно, latency)
- Вспомогательные метрики:
  - Загрузка процессора
  - Потребление памяти

## Выбор метрик

- Оптимизируем throughput
- Желаемые значения:  $\forall (\forall)$



- Наш подход к оптимизации:  
бенчмарки, виды экспериментов  
и инструменты анализа

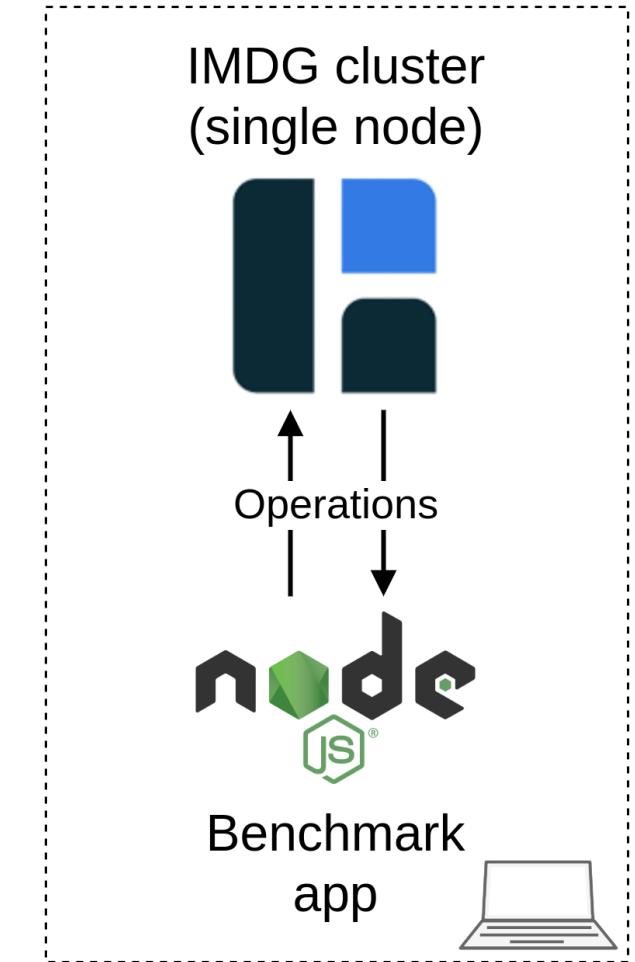
# Бенчмарк

```
const benchmark = new Benchmark({
    nextOp: () => map.get('foo'), // функция-фабрика для операций
    totalOpsCount: REQ_COUNT,     // общее число операций, 1 млн
    batchSize: BATCH_SIZE         // лимит на кол-во операций, 100
});

await benchmark.run();
```

# Сценарий бенчмарка

- Приложение-бенчмарк + кластер IMDG (1 нода)
- Фиксированные версии Linux, Node.js, IMDG и т.д.
- Операции: `Map#get()` и `Map#set()`
- Данные: строки с ASCII-символами (3 В, 1 КВ, 100 КВ)
- Замер: несколько запусков и вычисление среднего результата
- Каждый запуск: 1 млн операций с лимитом 100



# Горячий путь для бенчмарка 🔥

1. Старт операции (создание `Promise`)
2. Сериализация сообщения в бинарный формат
3. Отправка в сеть в `socket.write(...)`
4. Чтение фрейма в `socket.on('data', ...)`
5. Десериализация ответного сообщения
6. Вызов `resolve()` у `Promise`'а операции



## Вид эксперимента #1

- Proof of concept (PoC)
- Простейший способ опробовать оптимизацию на кодовой базе
- Все средства хороши, но нужен весь функционал кода на горячем пути

## Вид эксперимента #2

- Микробенчмарки позволяют быстро проверить гипотезу и/или обосновать результаты РоС
- *Предупреждение:* могут показывать температуру в Антарктиде
- Использовался фреймворк [Benchmark.js](#) (+ node-microtime)

# Инструмент #1

- Стандартный профилировщик Node.js:

```
node --prof app.js
```

- Основан на V8 sample-based profiler
- Учитывает JS и C++ код
- Можно получить человекочитаемое представление:

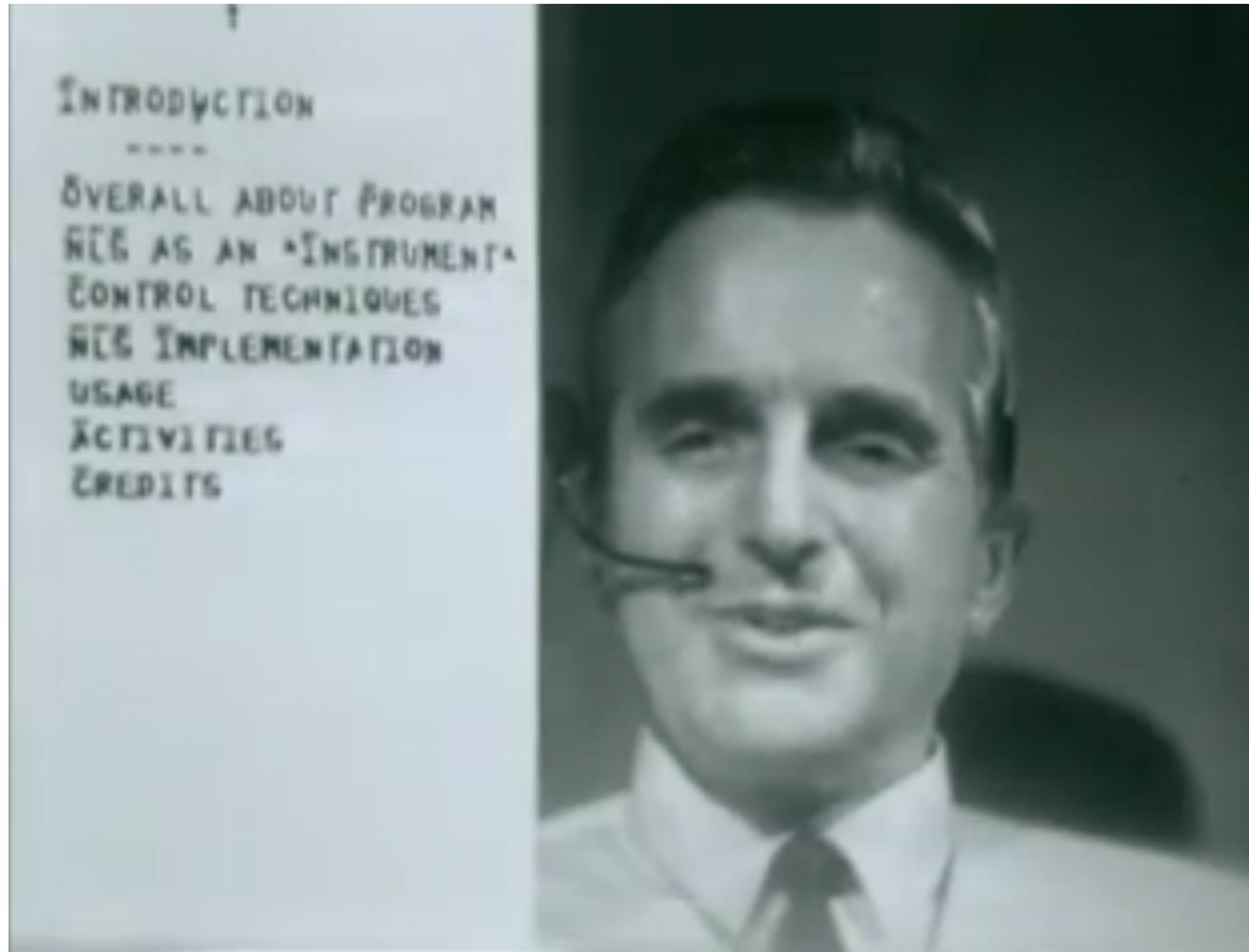
```
node --prof-process isolate-0xffffffffffff-v8.log > processed.txt
```

## Инструмент #2

- Визуализация отчета профилировщика в виде flame graph
- Отлично работает для event loop'a Node.js и помогает обнаруживать узкие места
- Спасибо Brendan Gregg, Netflix, [придумавшему подход](#) в 2013
- Мы использовали популярный инструмент - [0x](#) (умеет V8, perf, DTrace)

```
$ npm install -g 0x
$ 0x -o app.js
```

# It's flame graph demo time!

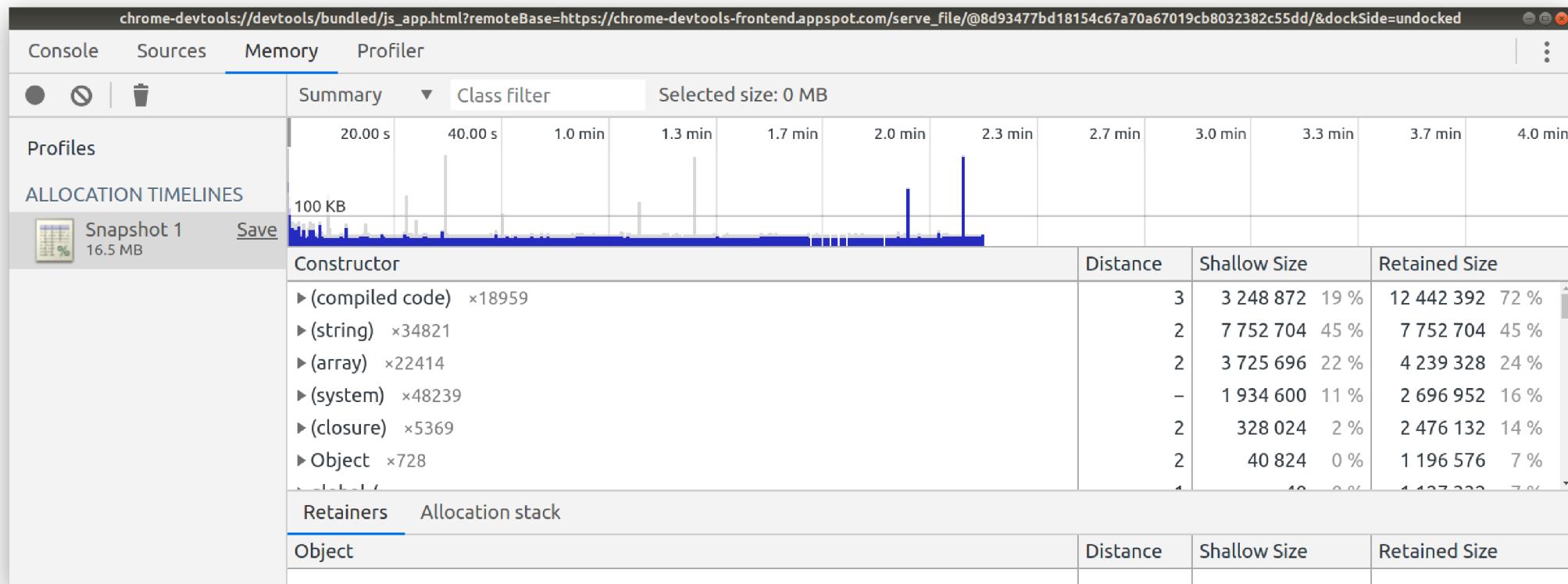


# Пример flame graph из реального мира



# Инструмент #3

- Профилировщик памяти из Chrome DevTools (Node.js)
- Умеет делать heap snapshot, отслеживать аллокации и не только



- История наших оптимизаций:  
гипотезы, эксперименты, замеры

# Включаем

- Отправляемся в весну 2019 года во времена v0.10

## Базовый замер

	get() 3 В	get() 1 KB	get() 100 KB	set() 3 В	set() 1 KB	set() 100 KB
v0.10.0	90 933	23 591	105	76 011	44 324	1 558

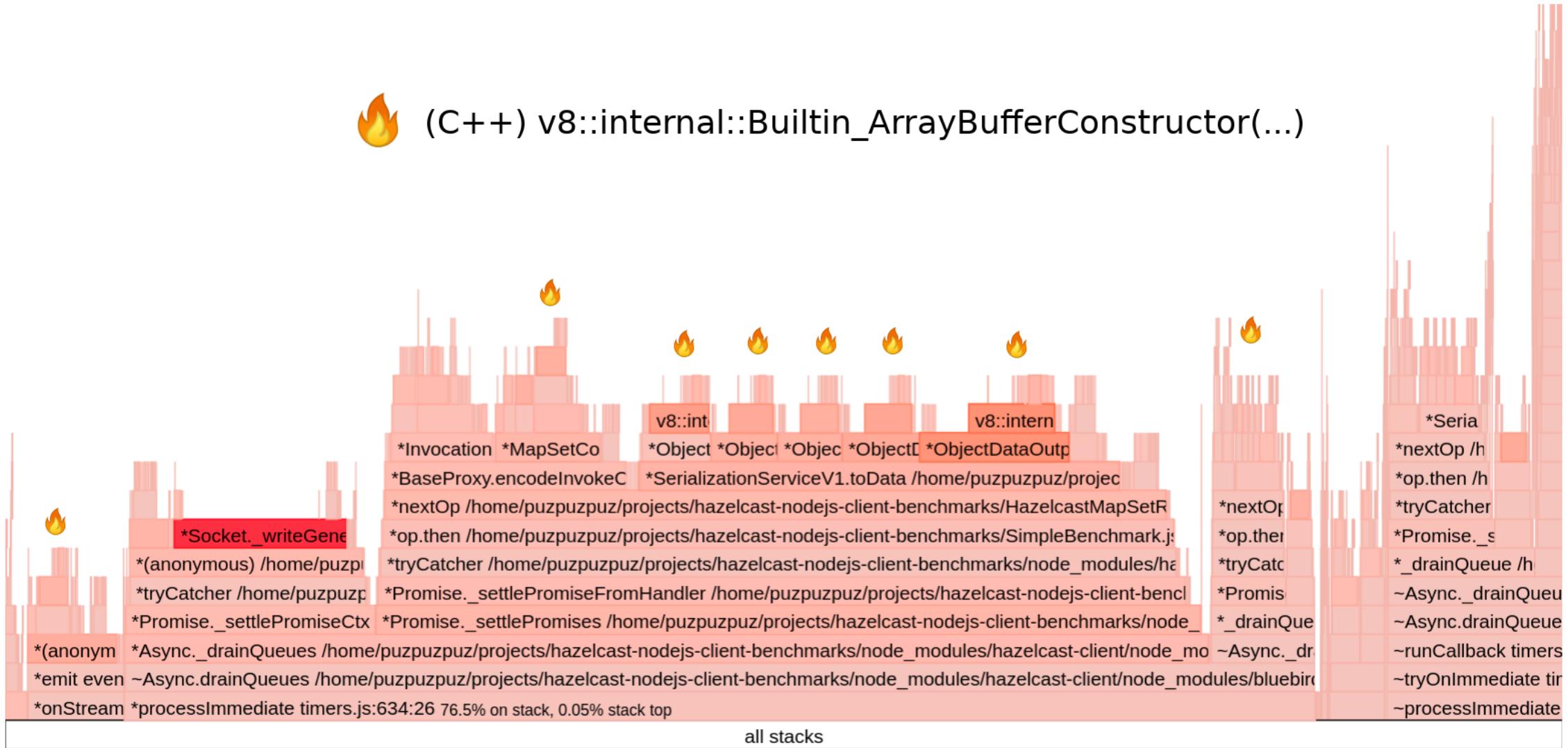
\* Абсолютные значения не важны (все замеры сделаны на моем ноутбуке)

## Пора анализировать

- Сначала профилируем сценарий "тяжелой" записи ( `Map#set()` ) и видим...

# Профильтровщик, приди! (запись 3 В)

(C++) v8::internal::Builtin\_ArrayBufferConstructor(...)



# Профилировщик, приди! (запись 3 В)

```
...
[C++ entry points]:
  ticks    cpp    total   name
  2775    37.8%  21.8%  v8::internal::Builtin_ArrayBufferConstructor(...)
   991    13.5%   7.8%  __libc_write
   329     4.5%   2.6%  v8::internal::Builtin_ArrayConcat(....)
...
```

# Хьюстон, у нас аллокации

- Для работы с бинарными данными, конечно, используется `Buffer`
- В на горячем пути много `Buffer#alloc()/#allocUnsafe()`, а это "дорогая" операция
- Во время сериализации одной операции происходит несколько аллокаций, а затем буферы копируются в финальный
- Это упрощает код, но производительность страдает
- Сначала пробуем РоС с очень простой полумерой

# РоС с полумерой

```
export class ObjectDataOutput implements DataOutput {  
  
    protected buffer: Buffer;  
    // ...  
  
    constructor() {  
        // пробуем аллоцировать жадно  
        -      this.buffer = Buffer.allocUnsafe(1);  
        +      this.buffer = Buffer.allocUnsafe(1024);  
  
        // ...  
    }  
}
```

## Замер производительности PoC

	get() 3 B	get() 1 KB	get() 100 KB	set() 3 B	set() 1 KB	set() 100 KB
v0.10.0	90 933	23 591	105	76 011	44 324	1 558
PoC	104 854	24 929	109	95 165	52 809	1 581
	+15%	+5%	+3%	+25%	+19%	+1%

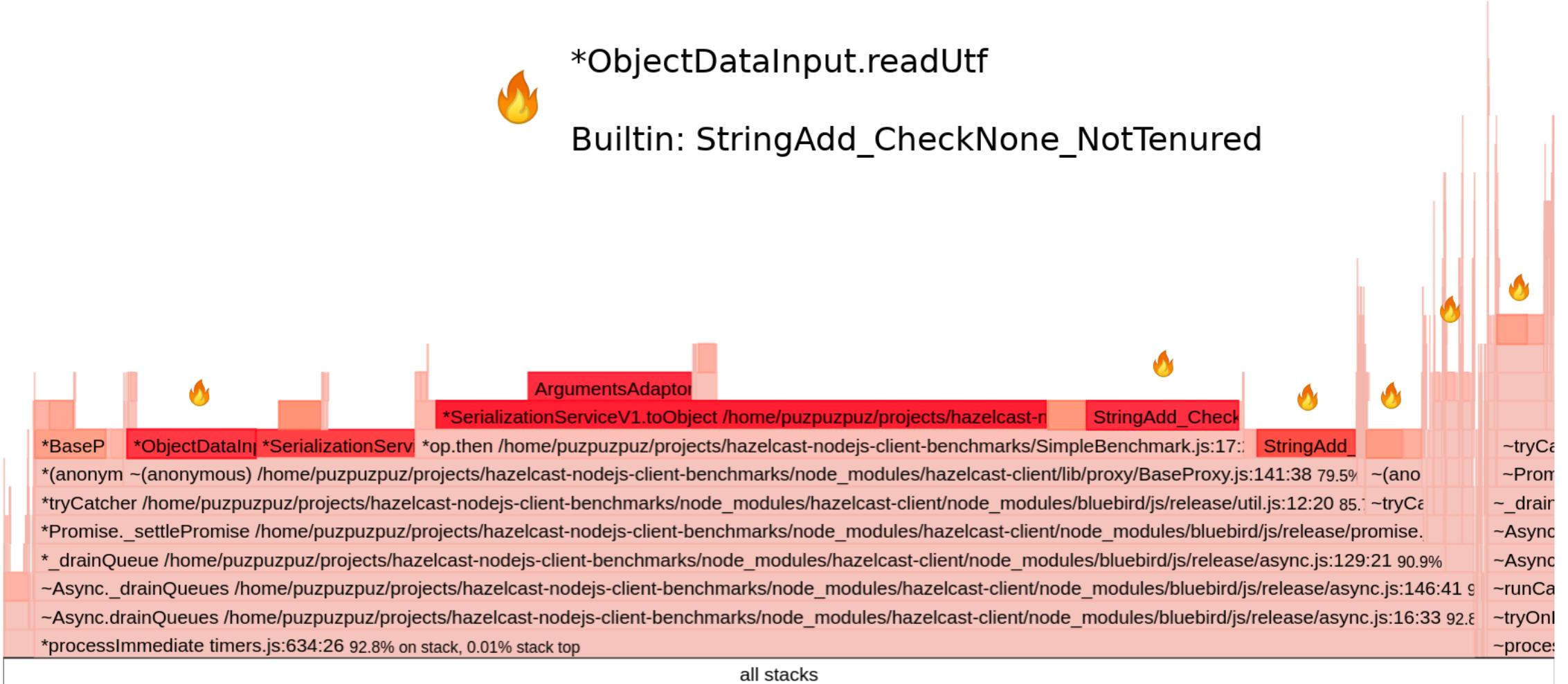
## Промежуточные итоги

- Гипотеза верна и правка идет в ближайший релиз
- Нужно избавиться от оставшихся избыточных аллокаций в будущих релизах
- Так что же у нас с зависимостью от размера данных?

## Снова пора анализировать

- Теперь профилируем сценарий "тяжелого" чтения ( `Map#get()` ) и видим...

# Профилировщик, приди! (чтения 100 KB)



# А ЧТО ЭТО У НАС ТАМ?

```
private readUTF(pos?: number): string {
    const len = this.readInt(pos);
    // ...
    for (let i = 0; i < len; i++) {
        let charCode: number;
        leadingByte = this.readByte(readingIndex) & MASK_1BYTE;
        readingIndex = this.addOrUndefined(readingIndex, 1);
        const b = leadingByte & 0xFF;
        switch (b >> 4) {
            // ...
        }
        result += String.fromCharCode(charCode);
    }
    return result;
}
```

# А ЧТО ЭТО У НАС ТАМ?

```
private readUTF(pos?: number): string {
    const len = this.readInt(pos);
    // ...
    for (let i = 0; i < len; i++) {
        let charCode: number;
        leadingByte = this.readByte(readingIndex) & MASK_1BYTE;
        readingIndex = this.addOrUndefined(readingIndex, 1);
        const b = leadingByte & 0xFF;
        switch (b >> 4) {
            // ...
        }
        result += String.fromCharCode(charCode);
    }
    return result;
}
```

# А ЧТО ЭТО У НАС ТАМ?

```
private readUTF(pos?: number): string {
    const len = this.readInt(pos);
    // ...
    for (let i = 0; i < len; i++) {
        let charCode: number;
        leadingByte = this.readByte(readingIndex) & MASK_1BYTE;
        readingIndex = this.addOrUndefined(readingIndex, 1);
        const b = leadingByte & 0xFF;
        switch (b >> 4) {
            // ...
        }
        result += String.fromCharCode(charCode);
    }
    return result;
}
```

# Предварительная оптимизация?

- Итак, у нас нестандартная (де)серIALIZАЦИЯ UTF-8 строк
- Похоже на предварительную оптимизацию (привет, Java-клиент)
- Почему бы не сравнить со стандартным API?

```
// сериализация  
buf.write(inStr, start, end, 'utf8');  
// десериализация  
const outStr = buf.toString('utf8', start, end);
```

- Пишем микробенчмарк и делаем замер

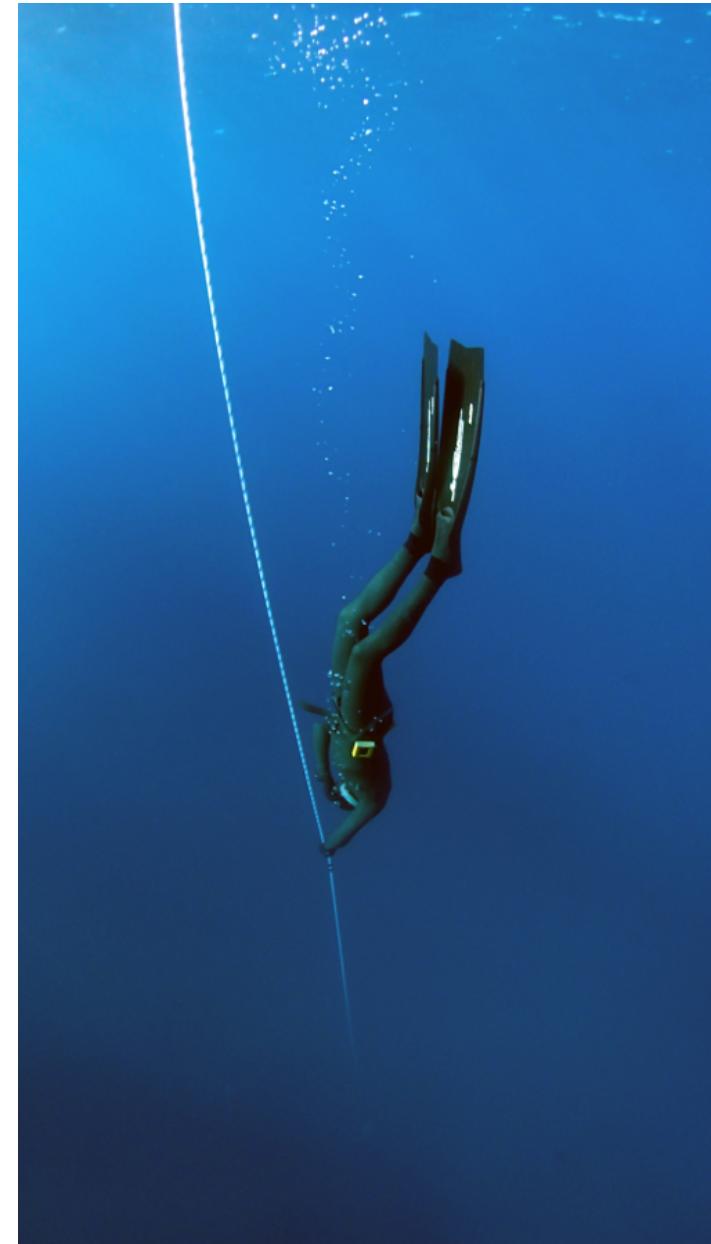
## Микробенчмарк

	<b>100 В ASCII</b>	<b>100 KB ASCII</b>	<b>100 В UTF</b>	<b>100 KB UTF</b>
custom	1 515 803	616	1 093 390	613
standard	11 297 821	68 721	1 311 610	794
	<b>+645%</b>	<b>+11 056%</b>	<b>+20%</b>	<b>+29%</b>

\* Результаты для десериализации в ops/sec

## Нужно идти глубже

- Buffer#toString()
- node:buffer.js#stringSlice()
- node:node\_buffer.cc#StringSlice()
- node:StringBytes#Encode()
- v8:String#NewFromUtf8()
- v8:Factory#NewStringFromUtf8()
- v8:Factory#NewStringFromOneByte()



# Что там, на глубине?

```
// v8:Factory#NewStringFromUtf8()
MaybeHandle<String> Factory::NewStringFromUtf8(
    Vector<const char> string,
    PretenureFlag pretenure
) {
    // Check for ASCII first since this is the common case.
    const char* ascii_data = string.start();
    int length = string.length();
    int non_ascii_start = String::NonAsciiStart(ascii_data, length);
    if (non_ascii_start >= length) {
        // If the string is ASCII, we do not need to convert
        // the characters since UTF8 is backwards compatible with ASCII.
        return
            NewStringFromOneByte(
                Vector<const uint8_t>::cast(string), pretenure);
    }
    // ...
}
```

# Что там, на глубине?

```
// v8:Factory#NewStringFromUtf8()
MaybeHandle<String> Factory::NewStringFromUtf8(
    Vector<const char> string,
    PretenureFlag pretenure
) {
    // Check for ASCII first since this is the common case.
    const char* ascii_data = string.start();
    int length = string.length();
    int non_ascii_start = String::NonAsciiStart(ascii_data, length);
    if (non_ascii_start >= length) {
        // If the string is ASCII, we do not need to convert
        // the characters since UTF8 is backwards compatible with ASCII.
        return
            NewStringFromOneByte(
                Vector<const uint8_t>::cast(string), pretenure);
    }
    // ...
}
```

# Что там, на глубине?

```
// v8:Factory#NewStringFromUtf8()
MaybeHandle<String> Factory::NewStringFromUtf8(
    Vector<const char> string,
    PretenureFlag pretenure
) {
    // Check for ASCII first since this is the common case.
    const char* ascii_data = string.start();
    int length = string.length();
    int non_ascii_start = String::NonAsciiStart(ascii_data, length);
    if (non_ascii_start >= length) {
        // If the string is ASCII, we do not need to convert
        // the characters since UTF8 is backwards compatible with ASCII.
        return
            NewStringFromOneByte(
                Vector<const uint8_t>::cast(string), pretenure);
    }
    // ...
}
```

## Итоги микробенчмарка

- Гипотеза выглядит весьма убедительно
- Реализуем РоС и делаем замер

## РоС для сериализации

	get() 3 B	get() 1 KB	get() 100 KB	set() 3 B	set() 1 KB	set() 100 KB
v0.10.0	90 933	23 591	105	76 011	44 324	1 558
РоС	122 458	104 090	7 052	110 083	73 618	8 428
	+34%	+341%	+6 616%	+45%	+66%	+440%

## Промежуточные итоги

- Гипотеза про (де)сериализацию верна и правка идет в ближайший релиз
- Перед релизом, конечно же, делаем еще один замер

# Первый публичный релиз

	get() 3 B	get() 1 KB	get() 100 KB	set() 3 B	set() 1 KB	set() 100 KB
v0.10.0	90 933	23 591	105	76 011	44 324	1 558
v3.12	132 855	120 670	8 756	127 291	94 625	10 617
	+46%	+411%	+8 239%	+67%	+113%	+581%

# Editor's Cut "быстрых" правок

1. Кучка ложных гипотез и экспериментов (🗑)
2. Эксперимент с пулом буферов для сериализации
  - Неудачный, т.к. прибавка незначительная
  - `Buffer#allocUnsafe()` и так [использует](#) пул (8 KB by default)
  - Результат: отложен в backlog
3. Эксперимент с Write Queue (aka Automated Pipelining)
  - Простой РоС показал увеличение write throughput на 20-25%
  - Результат: оказался перспективным и был включен в следующий релиз

## Снова запрыгиваем в 🚗⌚

- Пропускаем кучу экспериментов, несколько оптимизаций и один релиз
- Возвращаемся обратно в настоящее
- Помните про лишние аллокации Buffer ?

## Работа с backlog'ом: аллокации Buffer

- Поскольку правки объемные, было решено начать с РоС
- Позволит оценить, оправдает ли оптимизация вложенные усилия
- РоС реализует оптимизацию только для Map#get() and Map#set()
- Реализуем и делаем замер...

## РоС без лишних аллокаций

	get() 3 B	get() 1 KB	get() 100 KB	set() 3 B	set() 1 KB	set() 100 KB
v3.12.1	173 611	161 812	10 879	172 028	82 747	8 208
РоС	222 172	192 122	12 594	205 254	109 051	11 630
	+28%	+19%	+16%	+19%	+32%	+42%

\* Замеры с включенным Automated Pipelining

## Сравним с тем, что было в начале

	get() 3 B	get() 1 KB	get() 100 KB	set() 3 B	set() 1 KB	set() 100 KB
v0.10.0	90 933	23 591	105	76 011	44 324	1 558
PoC	222 172	192 122	12 594	205 254	109 051	11 630
	+144%	+714%	+11 894%	+170%	+146%	+646%

\* Замеры с включенным Automated Pipelining

Наконец-то!



## Текущие планы

1. Реализовать оптимизацию с аллокациями Buffer
2. Включить замеры производительности в релиз цикл
3. Продолжить оптимизацию: гипотезы, эксперименты, замеры

## ➤ Выводы

## Вывод #1

### **Не пытайтесь оптимизировать все и сразу**

- Сначала - выбор наиболее ходовых сценариев, окружения и характера нагрузки
- Только потом - бенчмарк и анализ
- Не стоит сразу вносить сложные правки, начинайте с поиска "простых" оптимизаций
- *Примечание:* очень здорово, когда большинство неоптимальных решений отсеиваются еще на этапе разработки

## Вывод #2

**Бенчмарки должны быть детерминированными и "справедливыми"**

- Должны быть воспроизводимы в любой момент
- Окружение в рамках одного анализа должно быть зафиксировано
-  vs. : при сравнении стоит идти небольшими шагами, поэтапно
- Прогрев бенчмарка, запуск множество раз и мат. статистика сделают сравнение более корректным

## Вывод #3

### **Семь раз замерь, один релизни**

- Не стоит делать поспешных выводов после замера вашего микробенчмарка/РоС/нескольких отдельных оптимизаций
- С оптимизациями стоит сомневаться во всем и все перепроверять

## Вывод #4

**Выбирайте инструменты под задачу, а не наоборот**

- Не стоит пытаться решить любую проблему с помощью, например, flame graphs
- При необходимости подбирайте другие инструменты и варьируйте бенчмарки

## Вывод #5

**"Нельзя просто так взять и оптимизировать  
приложение/библиотеку/фреймворк" © Боромир**

- Если вы что-то оптимизировали N месяцев назад, то о текущей производительности ничего не известно
- Оптимизация производительности - это процесс, а не однократное действие
- Планируйте оптимизацию:
  - Включайте замеры в цикл релизов
  - Ведите backlog для оптимизаций

# Спасибо за внимание!

Выбирайте оптимизированные  
приложения/библиотеки/фреймворки для ваших проектов :)



## Полезные ссылки

- <https://hazelcast.org>
- <https://github.com/hazelcast/hazelcast-nodejs-client>
- <https://nodejs.org/en/docs/guides/simple-profiling>
- <https://nodejs.org/en/docs/guides/dont-block-the-event-loop>
- <https://blog.insiderattack.net/event-loop-and-the-big-picture-nodejs-event-loop-part-1-1cb67a182810>
- <https://www.ibm.com/developerworks/library/j-benchmark2/index.html>

➤ Bonus unlocked 

## Снова включаем

- Отправляемся во времена разработки релиза v3.12.1
- И начинаем с отложенной перспективной оптимизации

# Оптимизация Automated Pipelining

- Идея в объединении сообщений перед записью в сокет
- В результате делается меньше "дорогих" вызовов `Socket#write()`
- Подобная оптимизация использована в классе `WriteQueue` в DataStax Node.js Driver for Apache Cassandra

# Pipelining

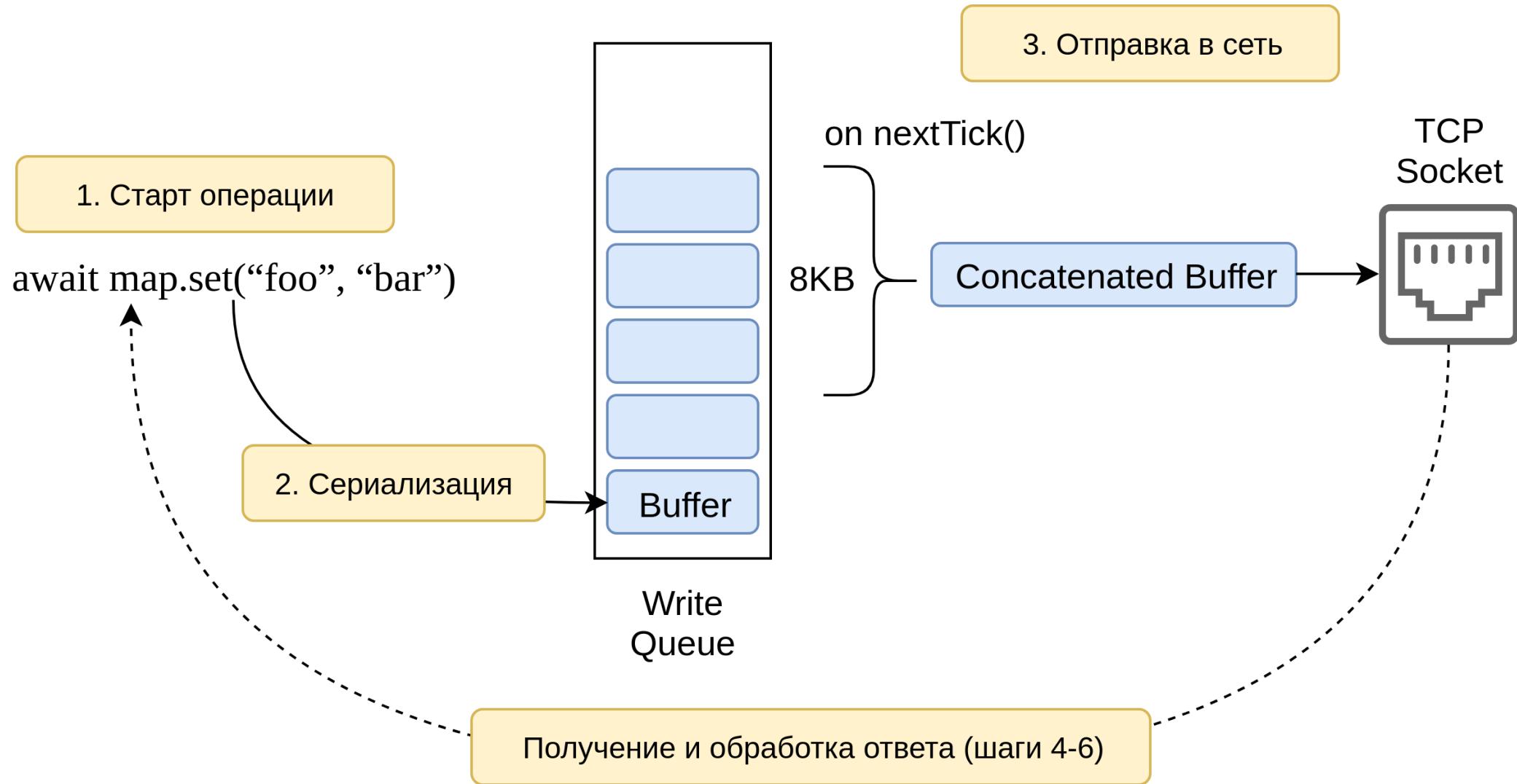
- В блокирующихся клиентах схожий прием называют **pipelining**
- Пример pipelining API в Java:

```
Pipelining<String> pipelining = new Pipelining<>(10);
for (int i = 0; i < 100; i++) {
    pipelining.add(map.getAsync(i));
}
// блокируемся и ждем результат
pipelining.results();
```

# Automated Pipelining

- В нашем случае, объединение происходит неявным образом, отсюда название - automated pipelining
- Достоинство:
  - Большее число приложений оказываются в выигрыше, поскольку оптимизация не требует изменения кода приложения
- Недостаток:
  - Из обычного pipelining можно "выжать" лучшую производительность

# Логика работы



# Особенности

- Для больших сообщений, записываемых в сокет, производительность может упасть
- Поэтому добавлена настройка библиотеки, включающая "обычный" режим
- Порог записи в сеть тоже вынесен в настройки

# Снова релиз?

- Automated Pipelining вошла в ближайший релиз
- Кроме этого, туда вошли миорные оптимизации для горячего пути:
  - Убрали мусор от `new Date().getTime()` (спасибо, `Date.now()`)
  - Убрали мусор от лишних конвертаций `number <-> Long` (используется `long.js`)
  - Избавились от аллокаций буферов в edge cases, например, при чтении больших сообщений (> 128 KB)
- Заметной прибавки в нашем бенчмарке миорные оптимизации не дали, но они все равно полезны
- Перед релизом снова все замеряем...

## Второй публичный релиз

	get() 3 В	get() 1 KB	get() 100 KB	set() 3 В	set() 1 KB	set() 100 KB
v3.12	132 855	120 670	8 756	127 291	94 625	10 617
v3.12.1	173 611	161 812	10 879	172 028	82 747	8 208
	+30%	+34%	+24%	+35%	-13%	-23%

\* Замеры с включенным Automated Pipelining