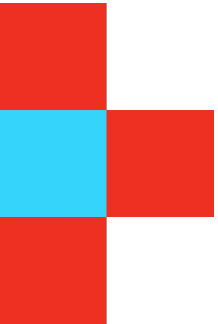
To be reactive...

RxJava



Joker<?>

Альфа-Лаборатория

Senior developer

Backend & DevOps

Максим Гореликов

@gorelikoff

gorelikov.max@gmail.com

Краткое содержание

- Предыстория
- Ищем варианты

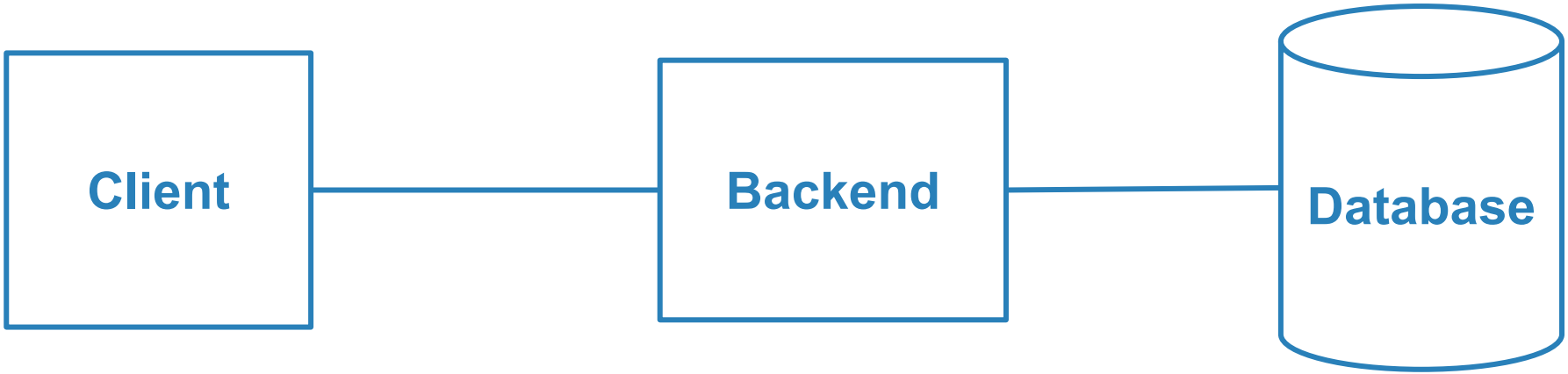
Краткое содержание

- Предыстория
- Ищем варианты
- Обзор RxJava
- Что там под капотом?

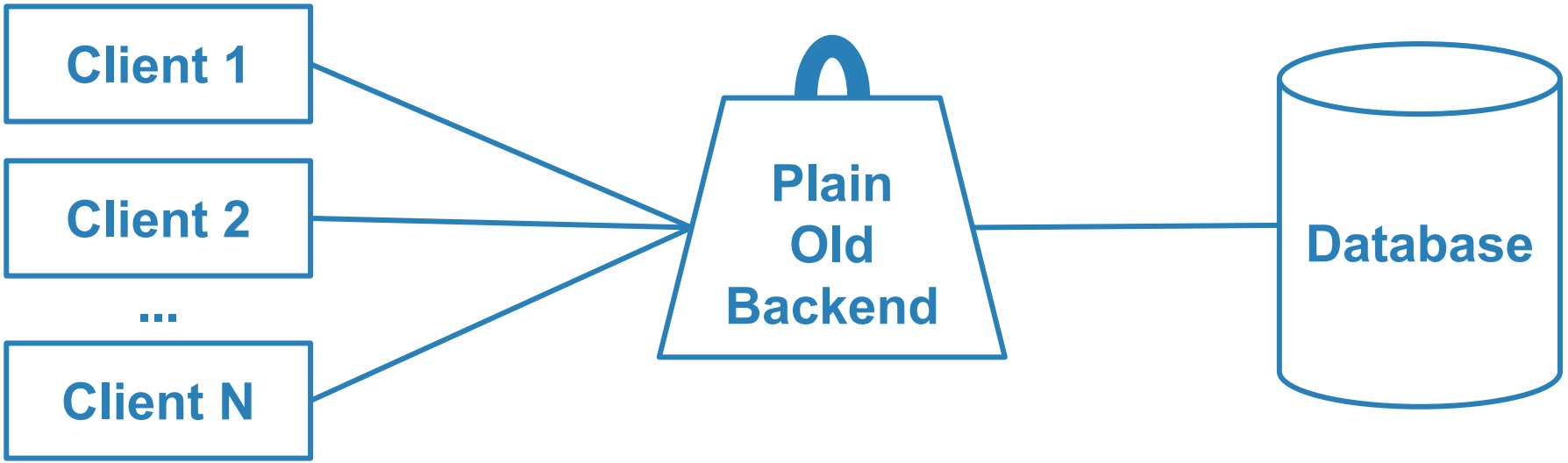
Краткое содержание

- Предыстория
- Ищем варианты
- Обзор RxJava
- Что там под капотом?
- А дальше?
- Выводы

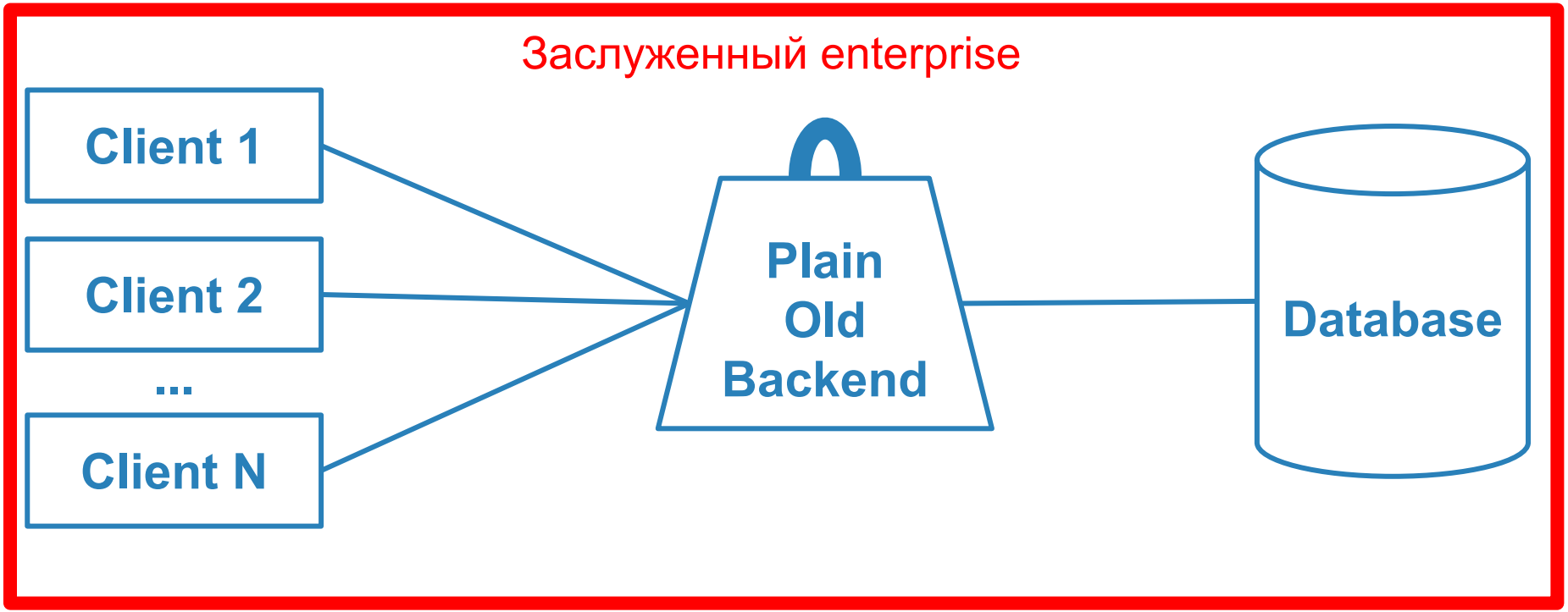
Знакомая архитектура?



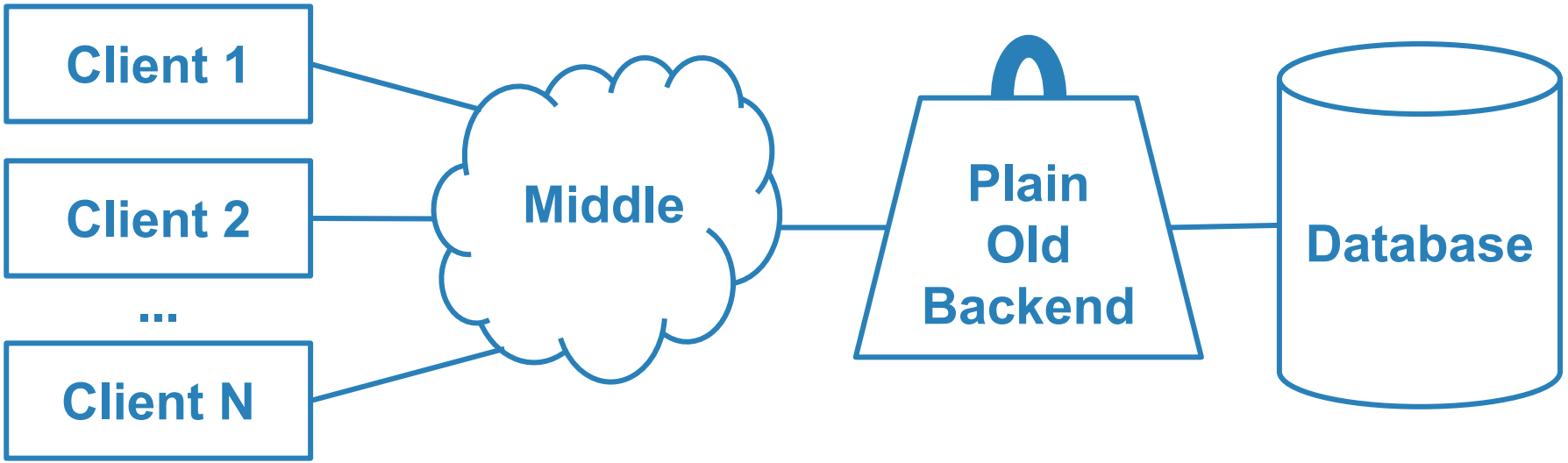
Наш вариант



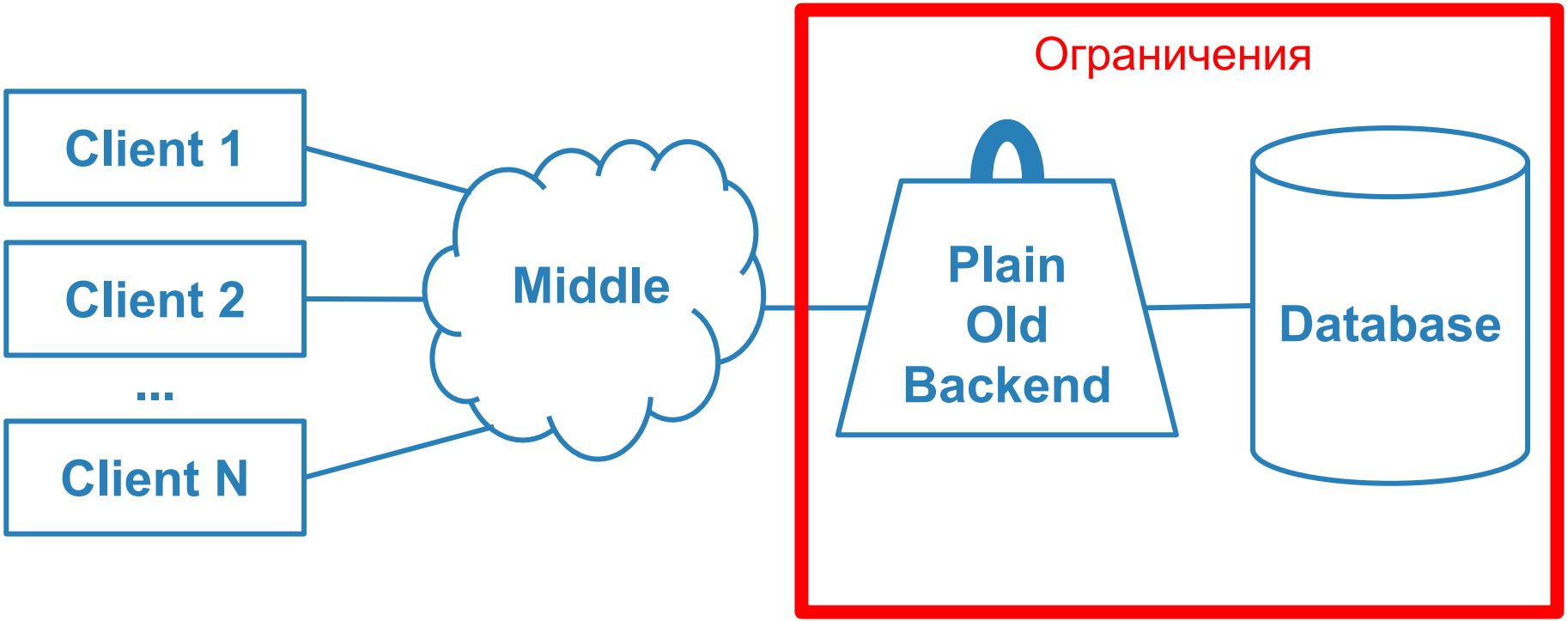
Наш вариант


Заслуженный enterprise
Client 1
Client 2
...
Client N
Plain
Old
Backend
Database

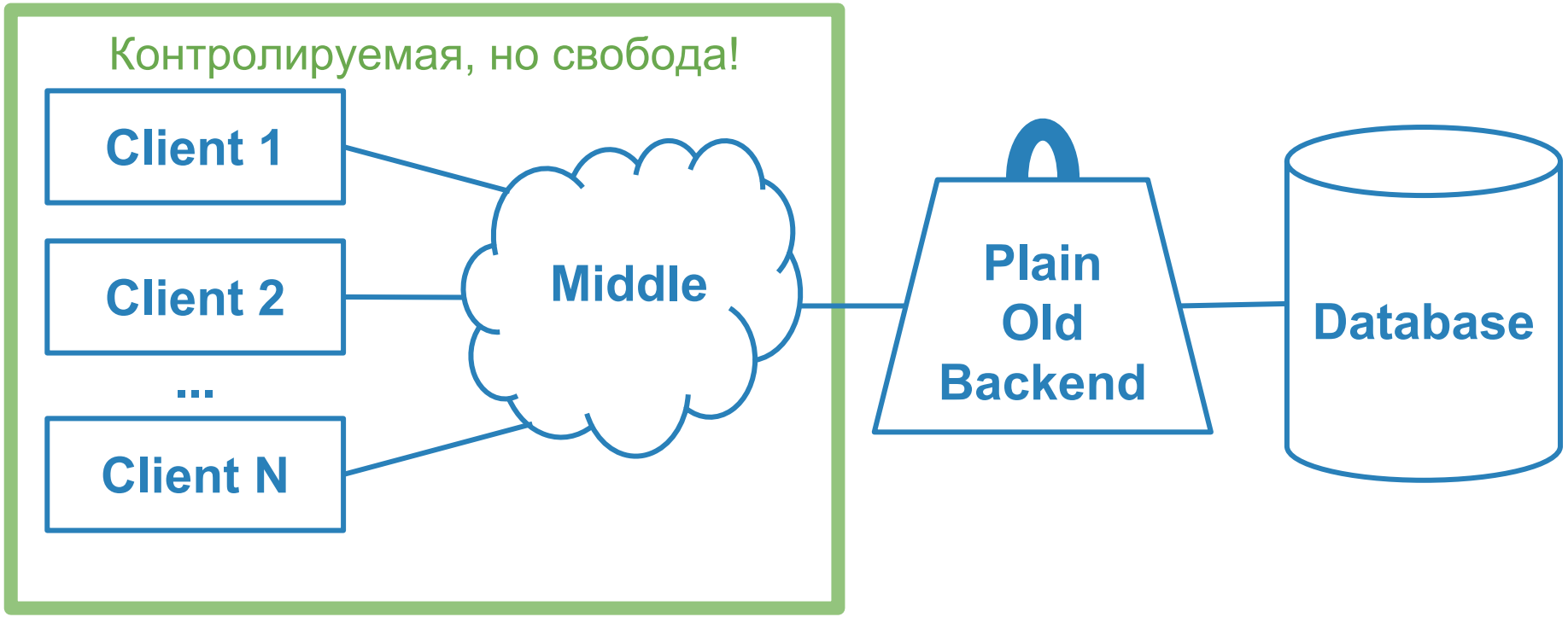
Новый вариант



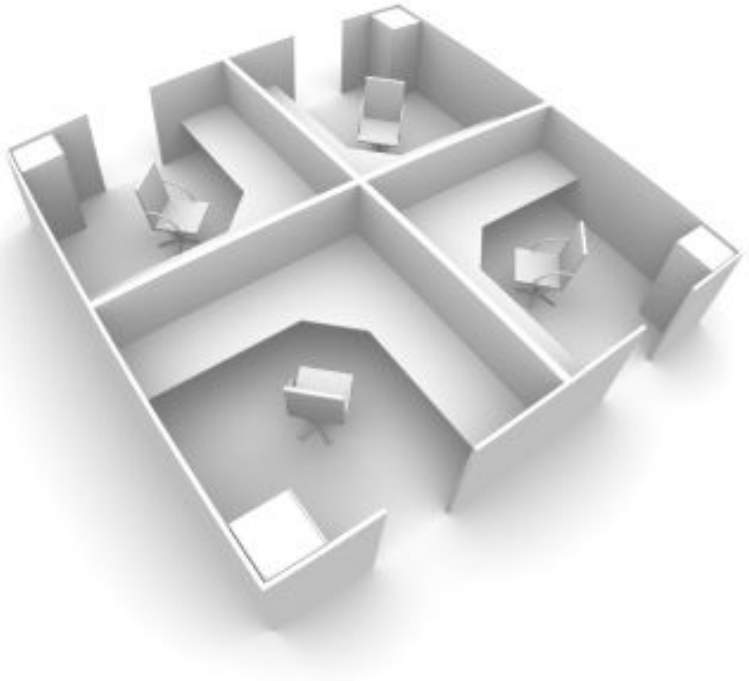
Новый вариант



Новый вариант

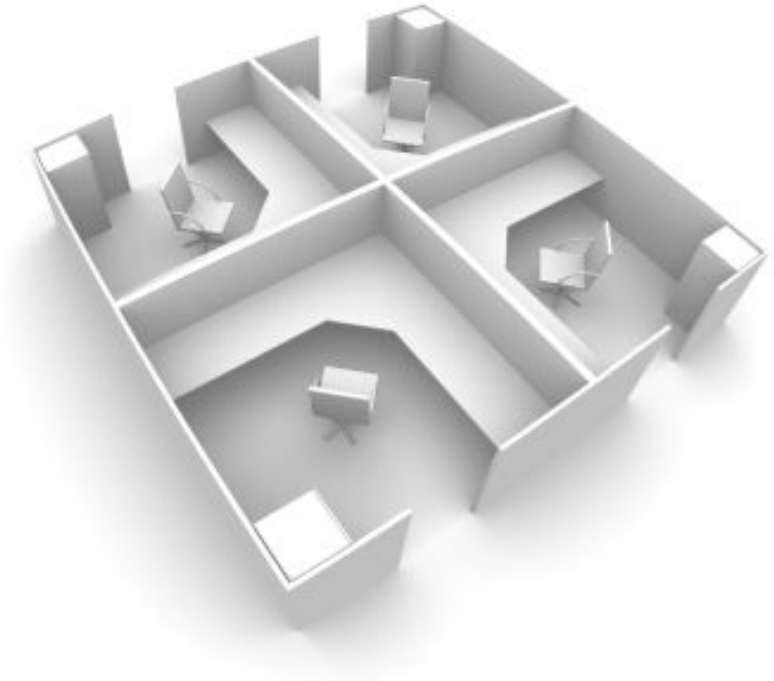


Middle



Middle. Внимание, buzzwords!

Микросервисы



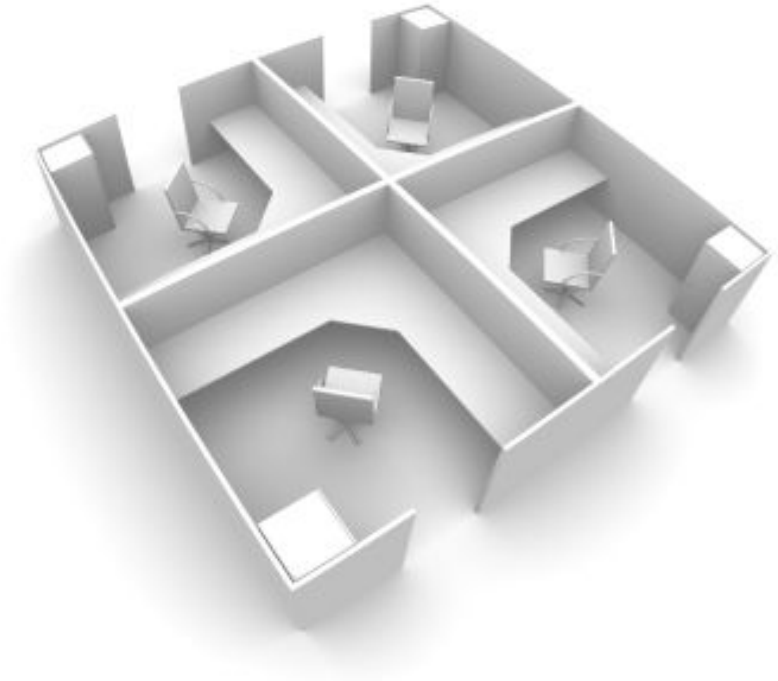
DevOps

Middle. Внимание, buzzwords!

Микросервисы

Spring cloud

NETFLIX | OSS



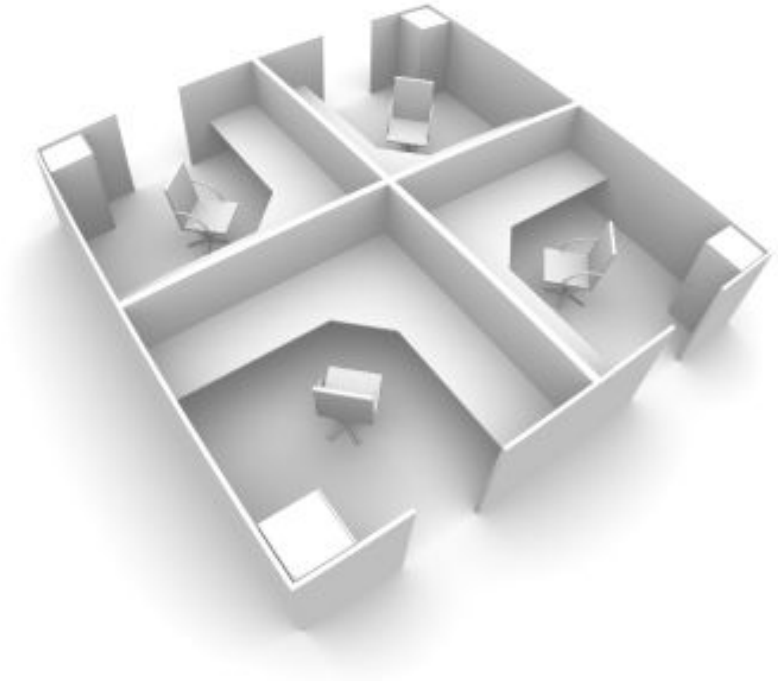
DevOps

Middle. Внимание, buzzwords!

Микросервисы

Spring cloud

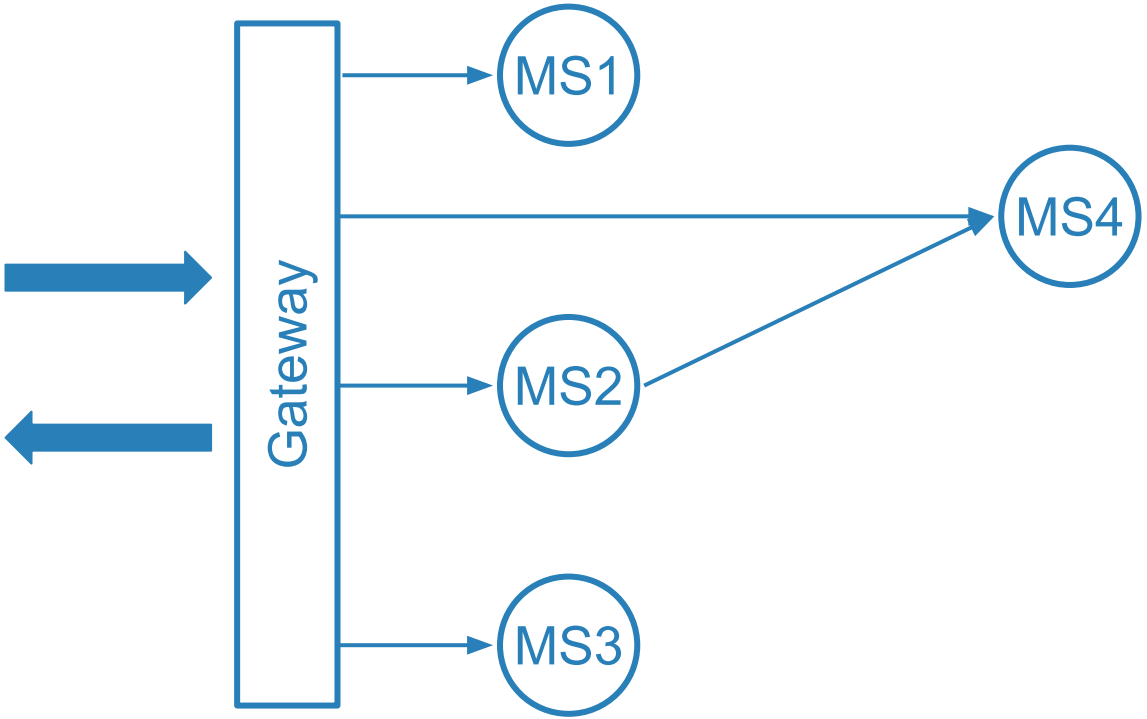
NETFLIX | OSS



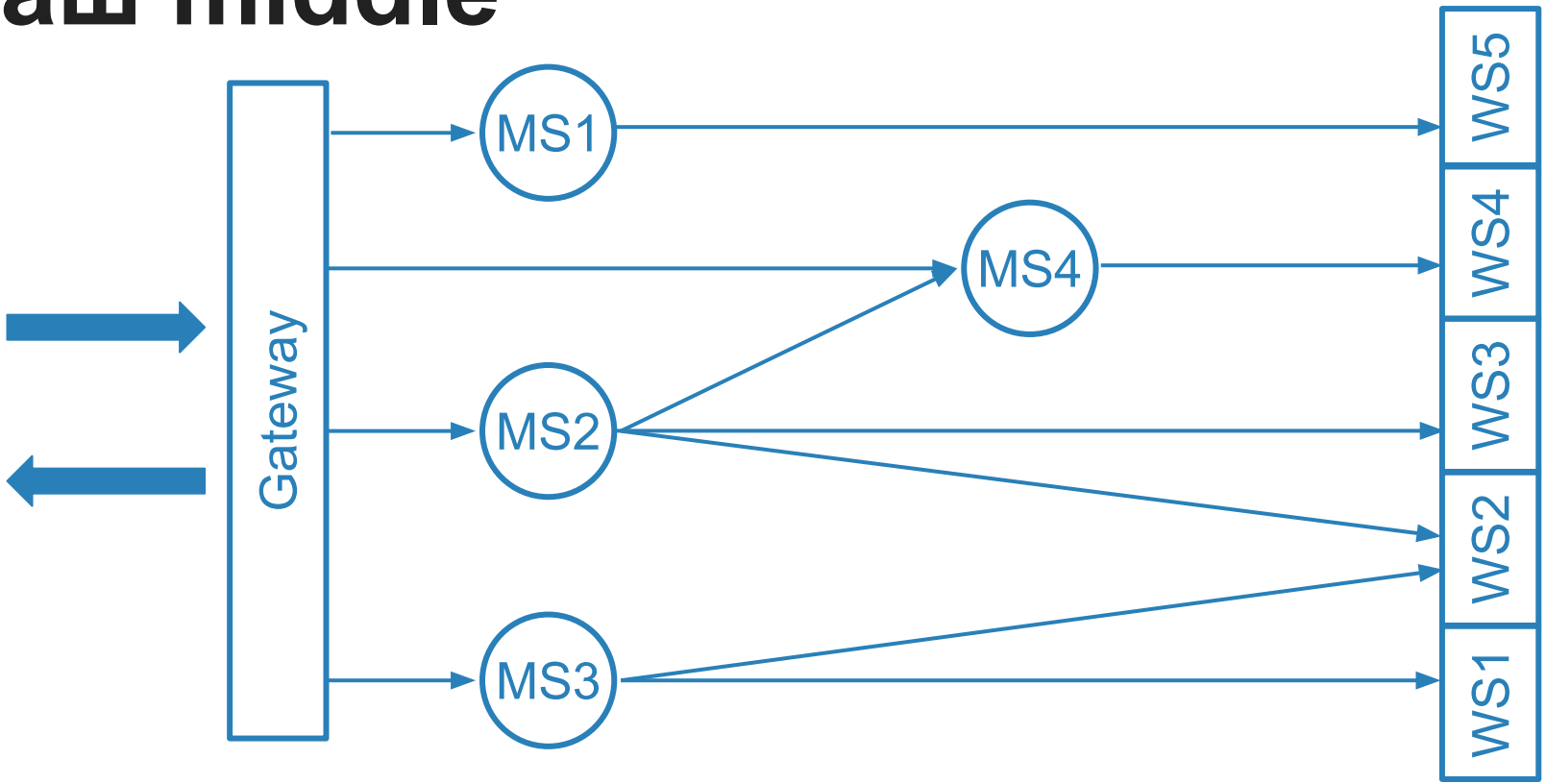
DevOps

MESOS

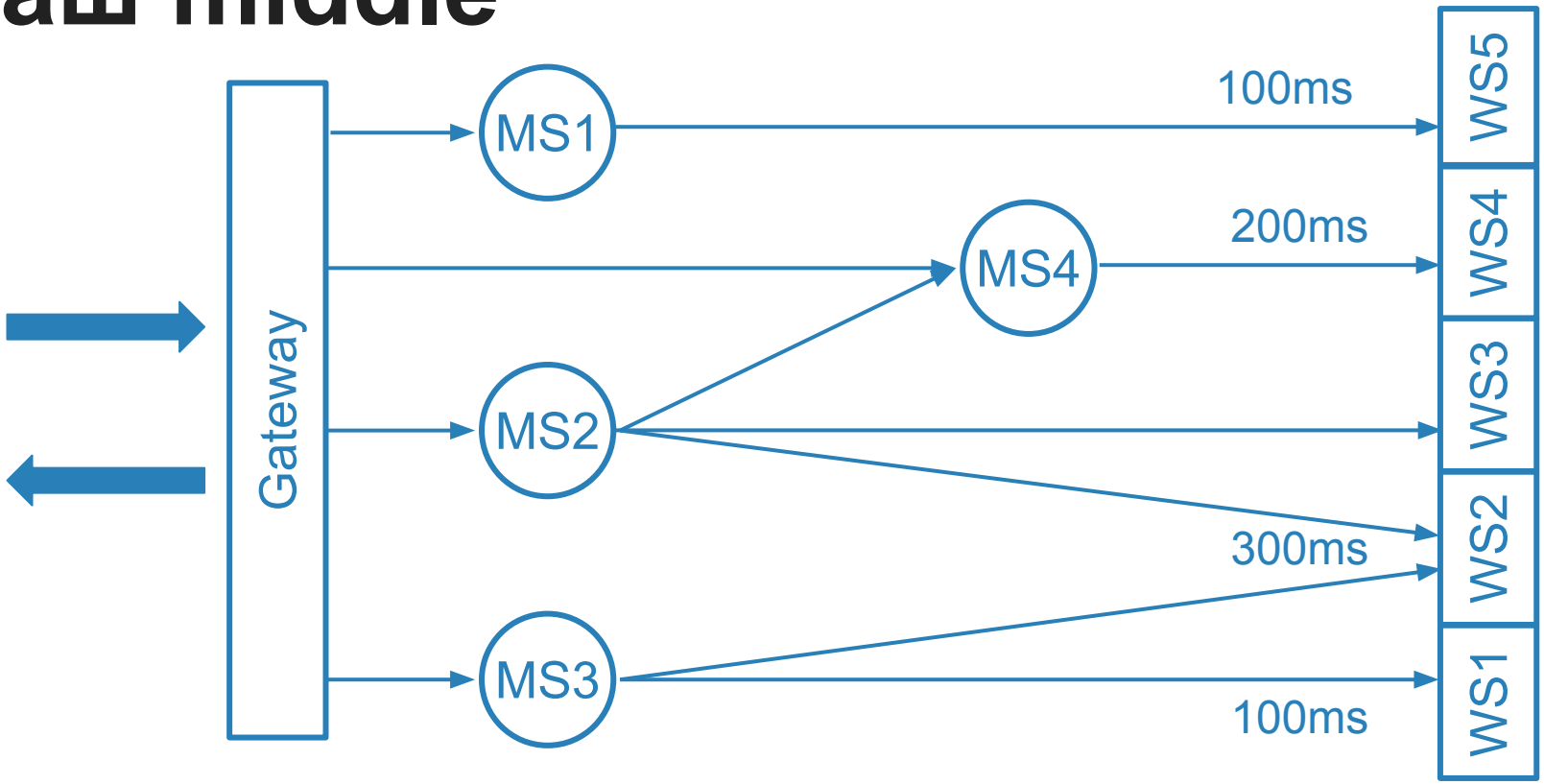
Наш middle



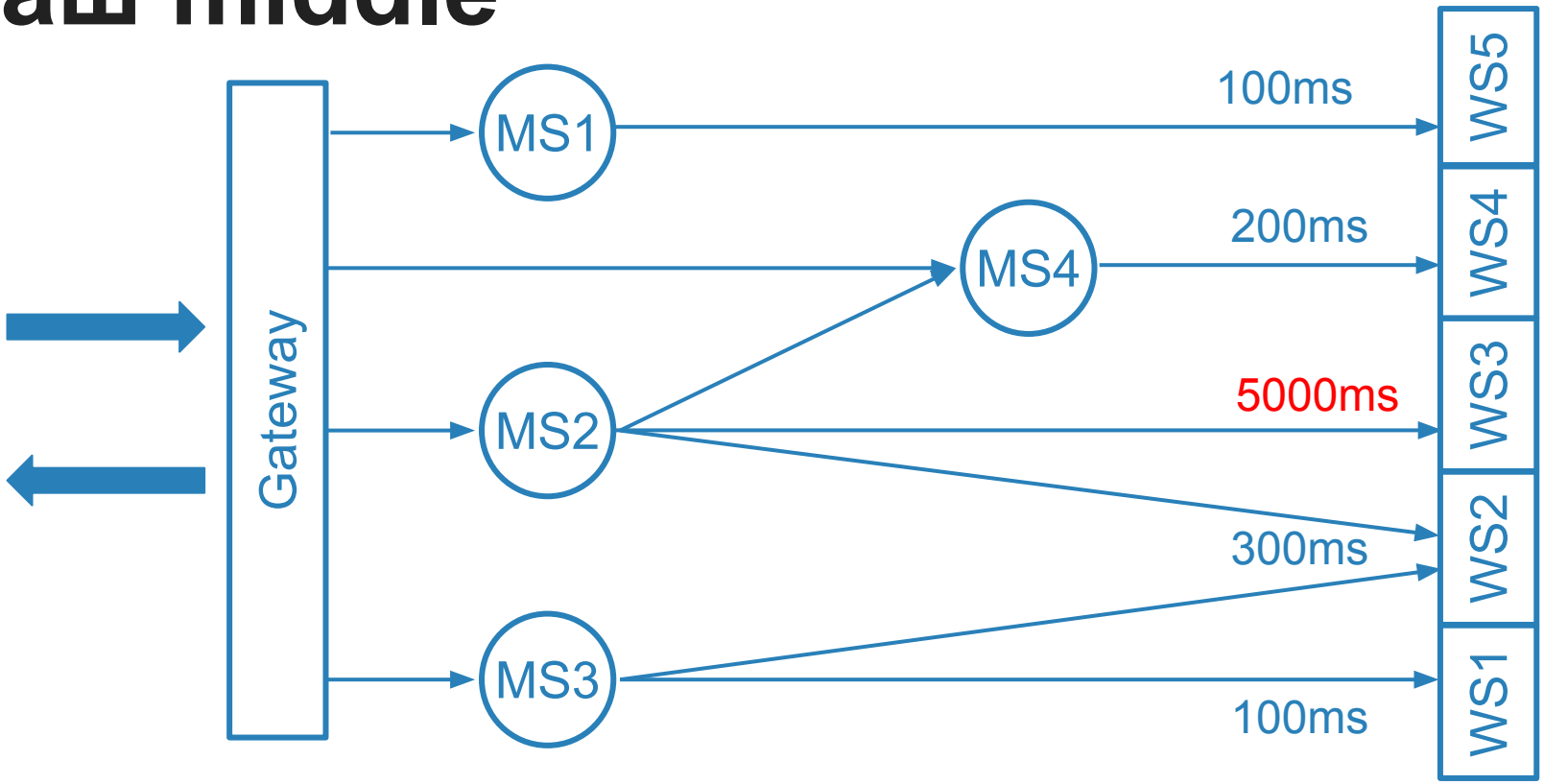
Наш middle



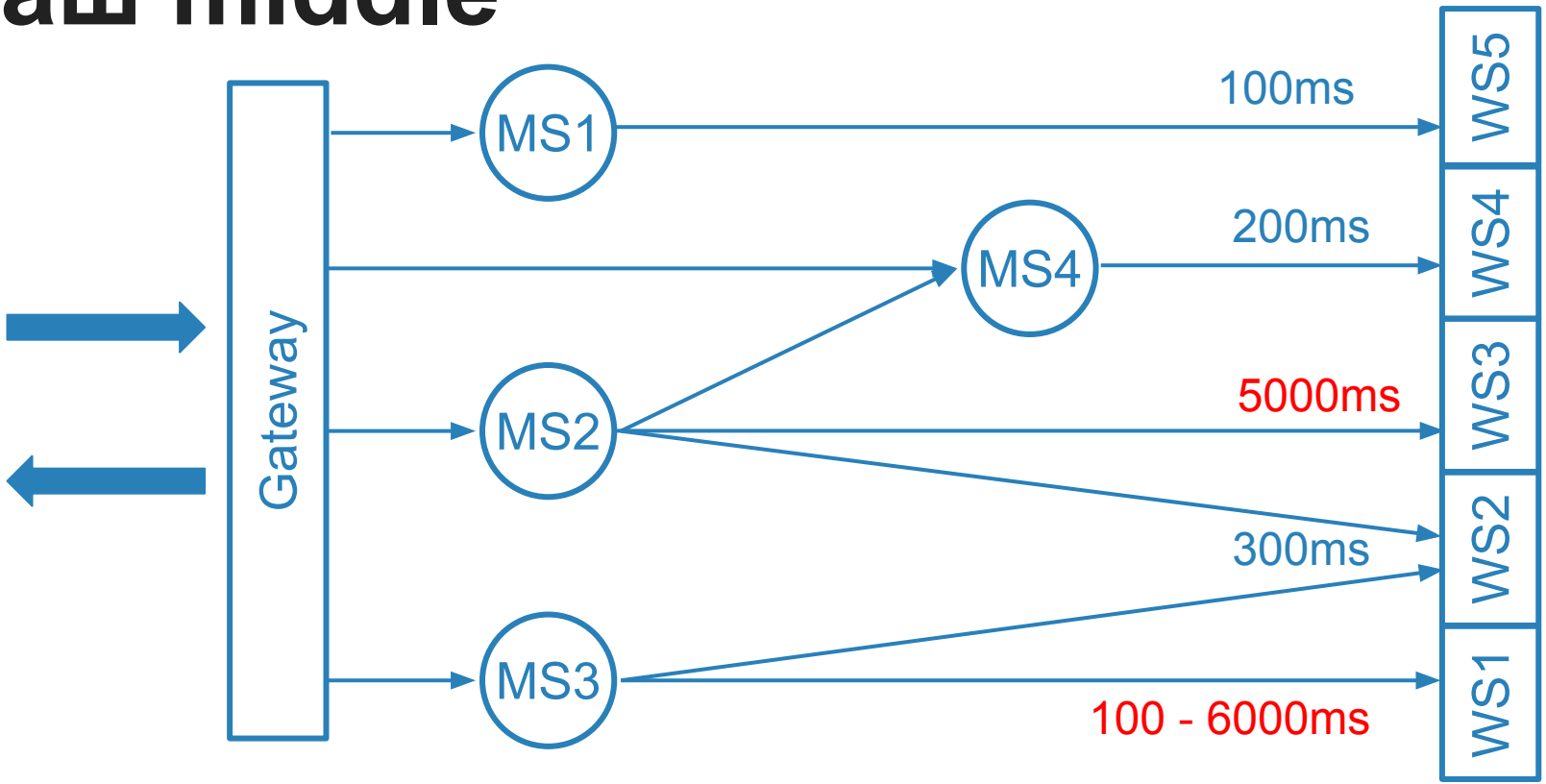
Наш middle


Gateway
MS1
MS2
MS3
MS4
100ms
200ms
300ms
100ms
WS5
WS4
WS3
WS2
WS1

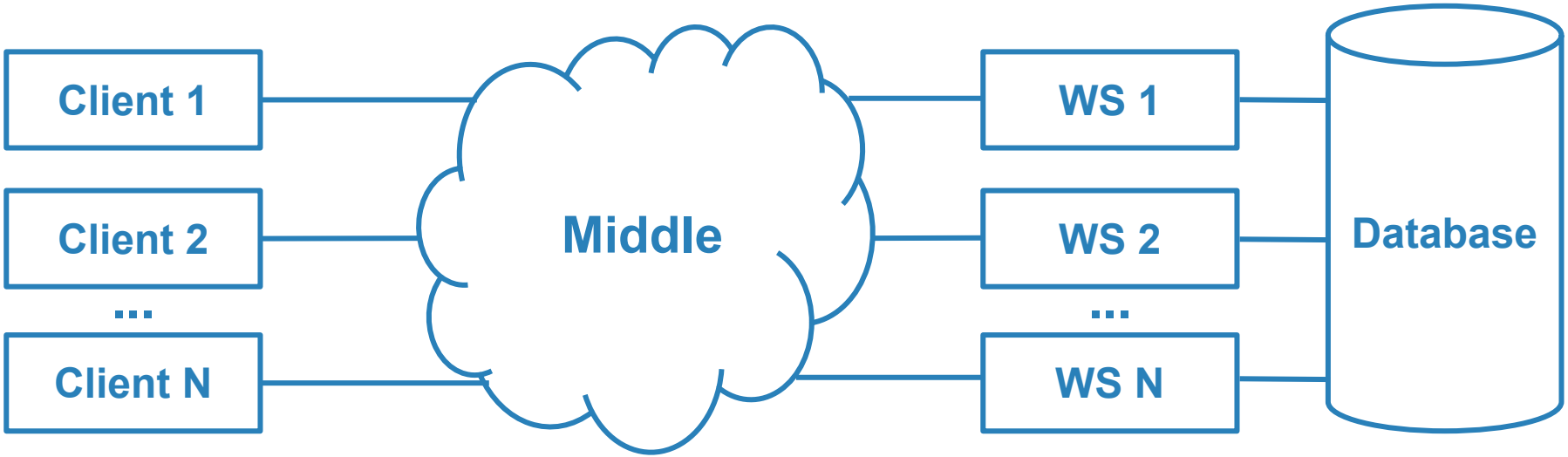
Наш middle


Gateway
MS1
MS2
MS3
MS4
100ms
200ms
5000ms
300ms
100ms
WS5
WS4
WS3
WS2
WS1

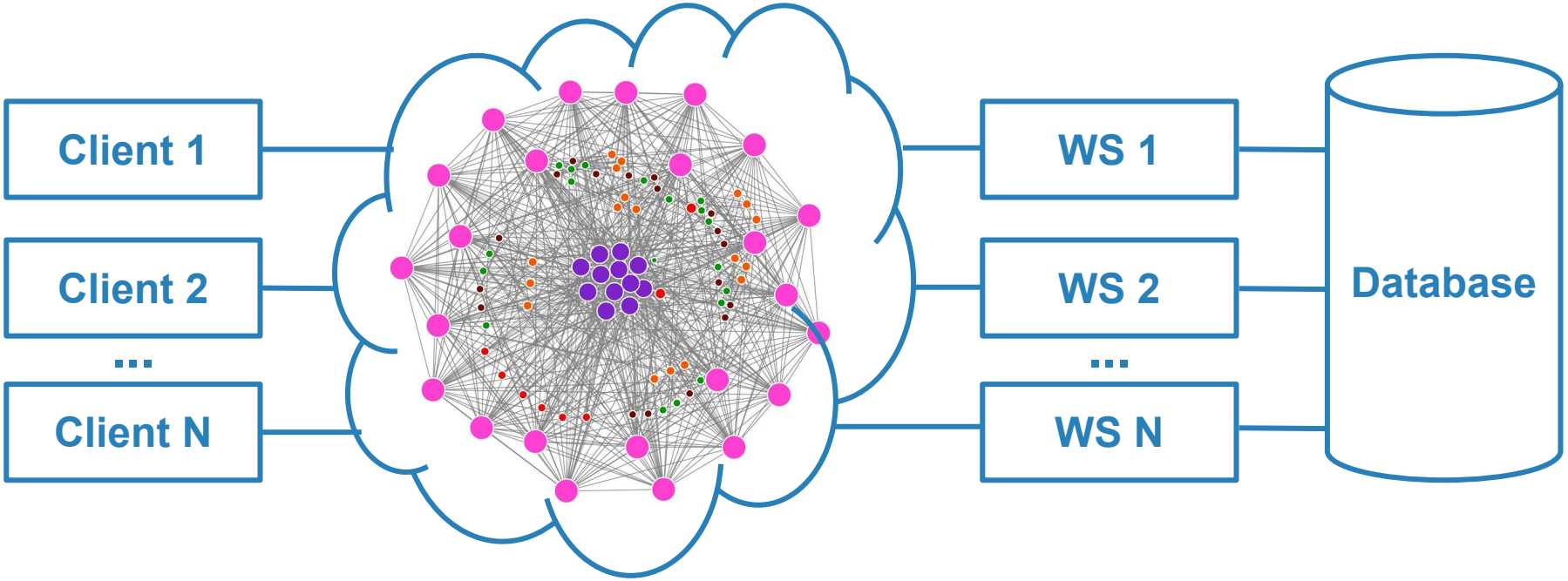
Наш middle


Gateway
MS1
MS2
MS3
MS4
100ms
200ms
5000ms
300ms
100 - 6000ms
WS5
WS4
WS3
WS2
WS1

Архитектура



Архитектура. Паутинка-style.



Условия

- Middle занимается сбором данных из различных источников
- Время ответа различных источников может сильно различаться и нам неподконтрольно

Желания

- Хотим удобную, параллельную загрузку данных с возможностью обработки по мере их готовности
- Хотим сами отдавать данные по мере их готовности

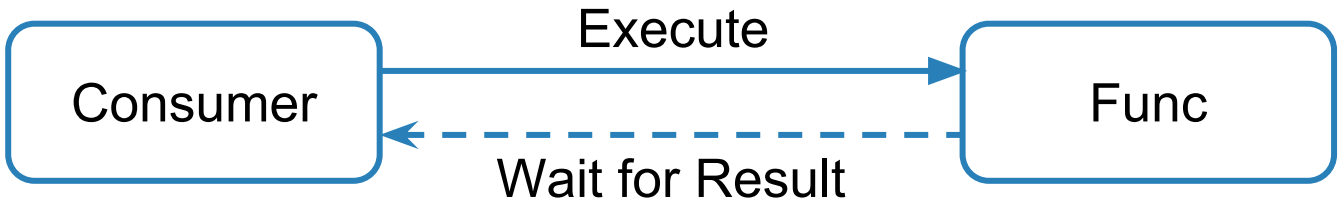
Желания

- Хотим удобную, параллельную загрузку данных с возможностью обработки по мере их готовности
- Хотим сами отдавать данные по мере их готовности

Базовые стратегии

- Pull-based стратегия
- Push-based стратегия

Pull-based стратегия



Pull-based стратегия

- Подходит для случаев, когда данные готовы, например, параллельные вычисления

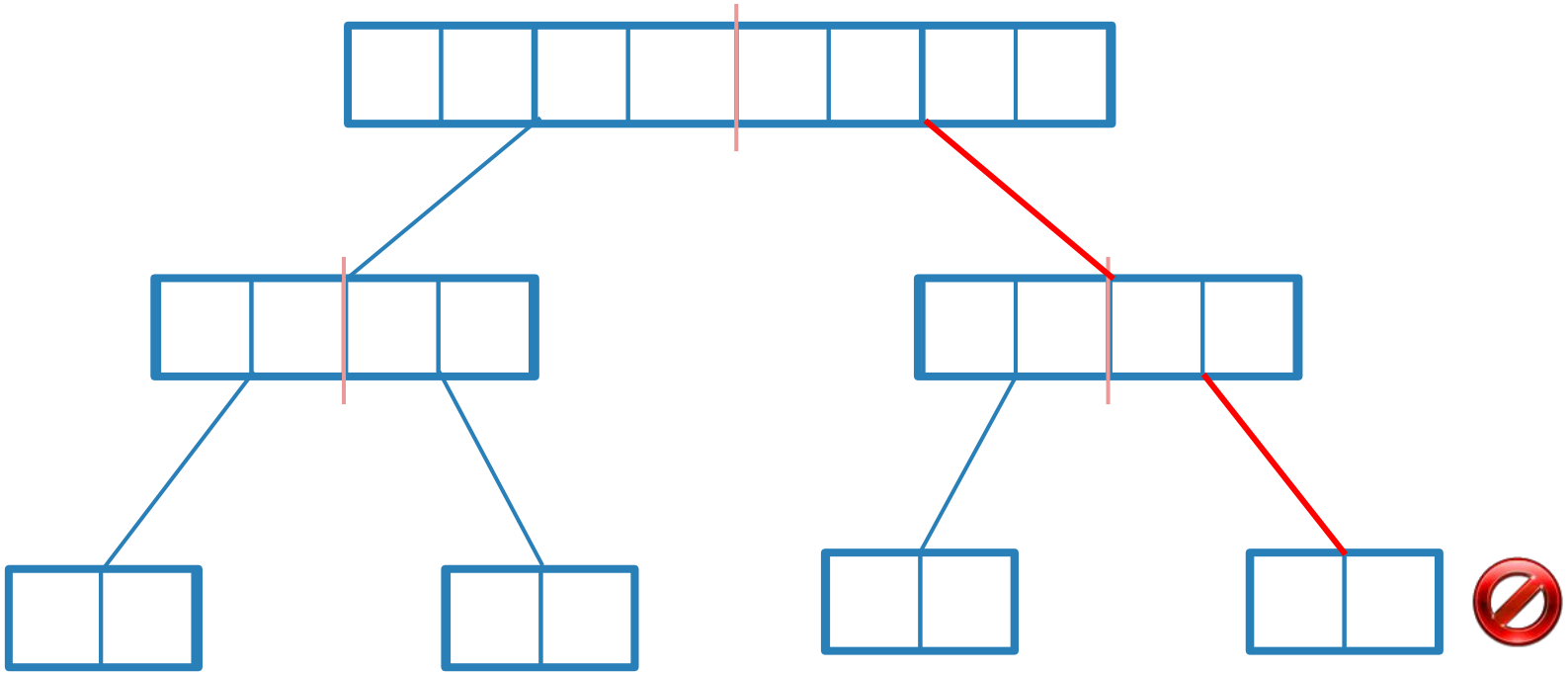
Pull-based стратегия

- Подходит для случаев, когда данные готовы, например, параллельные вычисления
- На ней основаны j.u.Stream и Fork/Join-фреймворки

Pull-based стратегия

- Подходит для случаев, когда данные готовы, например, параллельные вычисления
- На ней основаны j.u.Stream и Fork/Join-фреймворки
- Плохо подходит для случаев с внешними блокирующими вызовами

ForkJoin



Pull-based. Посмотреть.

Fork/Join: особенности реализации... (А. Шипилев и С. Куксенко)

Push-based стратегия

Push-based стратегия

- Подходит для случаев с блокирующими вызовами

Push-based стратегия

- Подходит для случаев с блокирующими вызовами
- Сложнее модель исполнения в многопоточности
- Сложнее контролировать ресурсы для исполнения

Push-based стратегия

- Подходит для случаев с блокирующими вызовами
- Сложнее модель исполнения в многопоточности
- Сложнее контролировать ресурсы для исполнения
- На ней основан CompletableFuture

Push-based. Посмотреть

CompletableFuture in Java 8, asynchronous processing done right

(Tomasz Nurkiewicz)

А можно и то и другое?

The masochist-only CountedCompleter class requires manual continuation passing transforms of fork/join style processing that can co-exist with reactive processing.

Doug Lea

А если очень хочется?

And there exists a hybrid solution that we use in "Reactive Streams", which at runtime dynamically switches between push and pull depending on if the consumer can't keep up or the producer can't keep up, without having to block or do any load-shedding.

Viktor Klang

Reactive Streams

Инициатива в попытке создать стандарт для асинхронной обработки данных с возможностью неблокирующего контроля над нагрузкой (backpressure).

Reactive Streams. Реализации

- Akka streams
- Project reactor
- ReactiveX (RxJava, RxScala, RxJs, etc.)

Смотрим бенчмарки?

Benchmark	times	lib = akka	lib = reactor1	lib = reactor2	lib = rxjava1	lib = rxjava2
Range	1	76 597,841	15 901 018,085	25 968 685,147	30 033 350,398	32 546 345,010
Range	1000	15 502,677	105 745,444	115 083,473	176 703,082	210 581,846
Range	1000000	23,234	151,836	157,884	187,971	253,364
RangeFlatMapJust	1	27 374,001	1 635 018,544	25 159 149,117	25 145 579,152	30 466 727,689
RangeFlatMapJust	1000	152,706	5 918,566	50 463,710	50 875,008	42 177,060
RangeFlatMapJust	1000000	0,166	6,057	44,484	47,034	40,447
RangeFlatMapRange	1	17 742,993	1 455 855,755	12 548 670,076	10 056 620,243	22 496 154,404
RangeFlatMapRange	1000	51,869	4 346,177	5 125,560	4 595,596	7 637,643
RangeFlatMapRange	1000000	0,054	3,946	5,376	4,757	8,192

А если честно...

- Модель RxJava выглядит проще и привычнее акторов

А если честно...

- Модель RxJava выглядит проще и привычнее акторов
- Reactor выглядит скорее как основа для фреймворков

А если честно...

- Модель RxJava выглядит проще и привычнее акторов
- Reactor выглядит скорее как основа для фреймворков
- RxJava - одна библиотека весом в 1Mb без доп. зависимостей

А если честно...

- Модель RxJava выглядит проще и привычнее акторов
- Reactor выглядит скорее как основа для фреймворков
- RxJava - одна библиотека весом в 1Mb без доп. зависимостей
- **Документация** по ReactiveX

RxJava

Библиотека, основанная на шаблоне Observer и построенная по правилам reactive streams

Rx. История

- Rx.NET 17.11.2009
- RxJS 17.03.2010
- RxJava 1.0 (by Netflix) xx.11.2012
- RxJava 2.0 ??.??.????

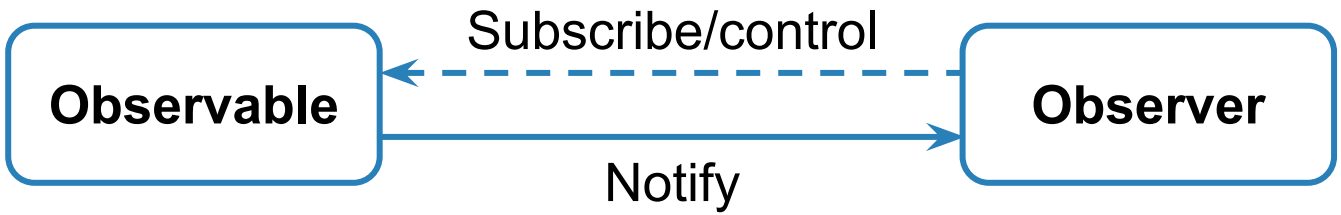
Rx. Полиглот

- RxScala
- RxClojure
- RxGroovy
- RxJRuby
- Rx.ruby
- RxCpp
- RxPY

- RxKotlin
- RxSwift
- RxPHP

- RxNetty
- RxAndroid
- RxCocoa

RxJava. Базовые элементы



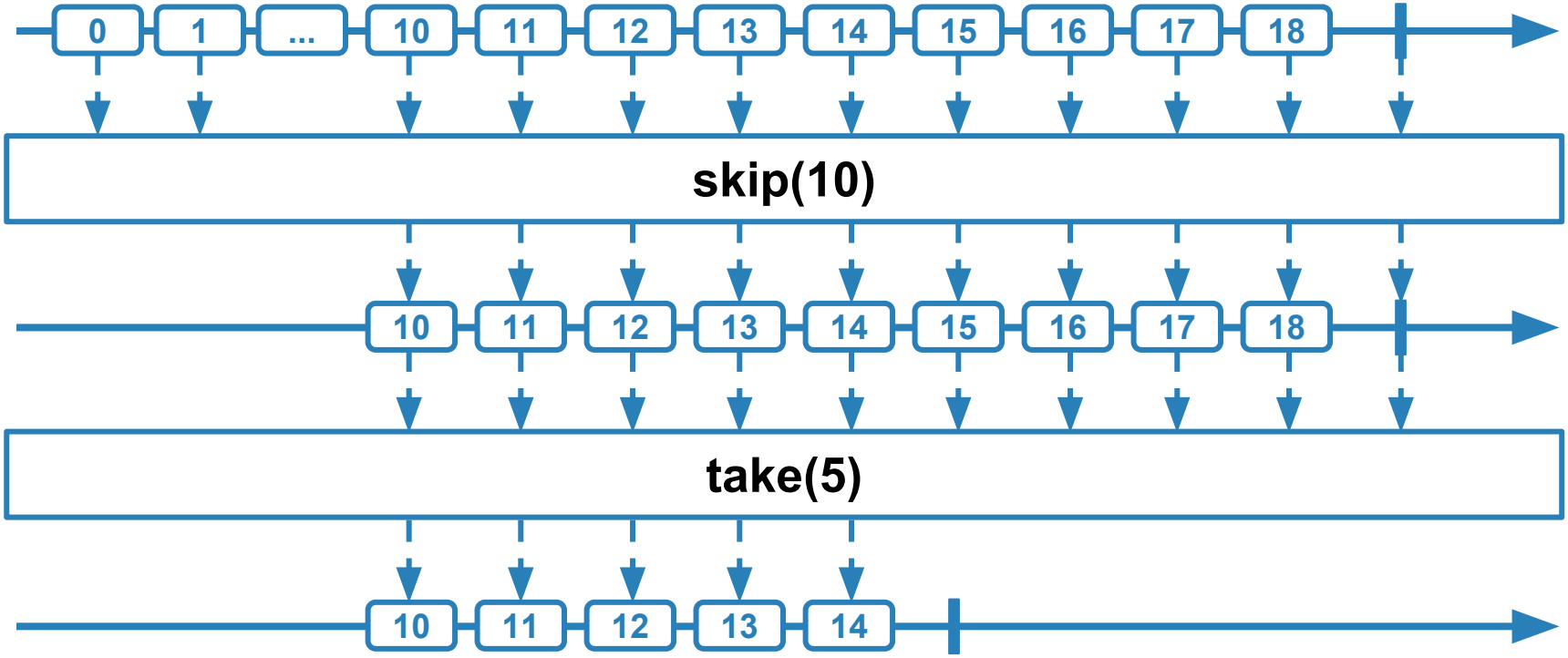
Observer

- onNext
- onError
- onCompleted

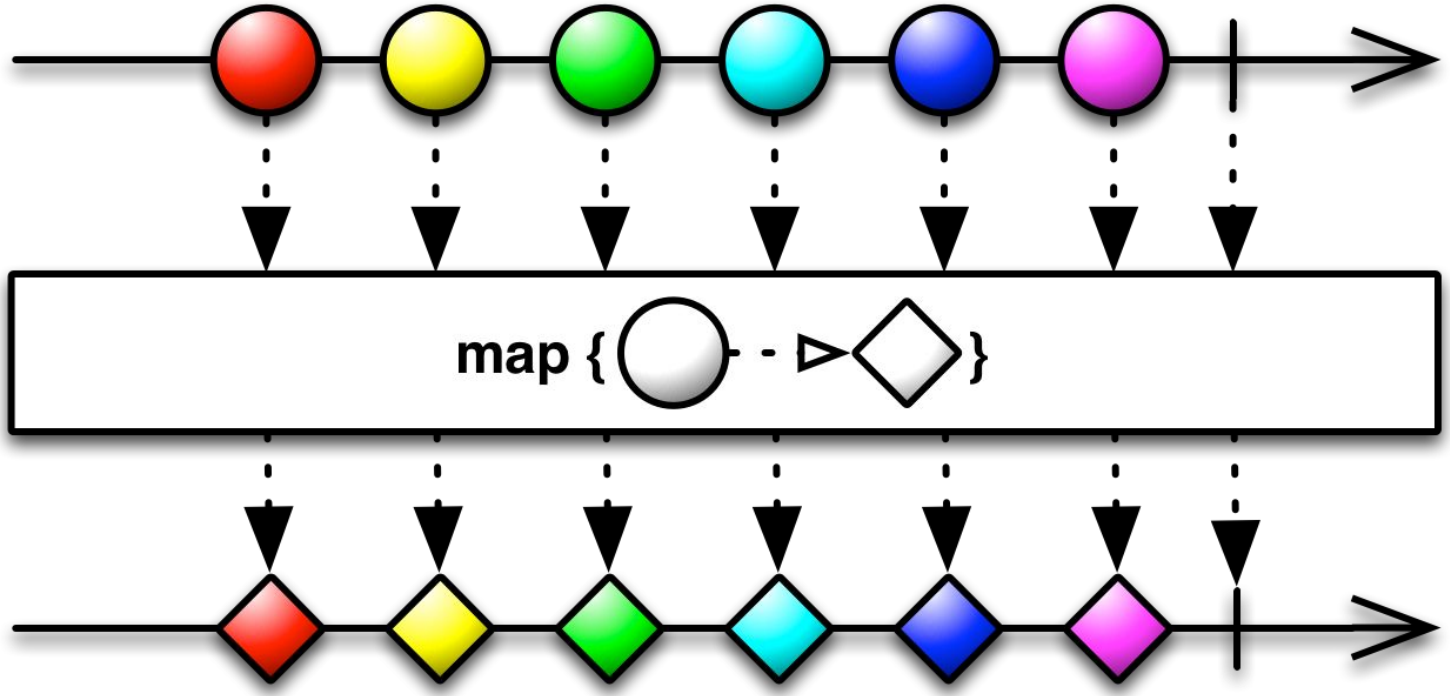
Observable

- subscribe
- Еще ~300 методов-операторов

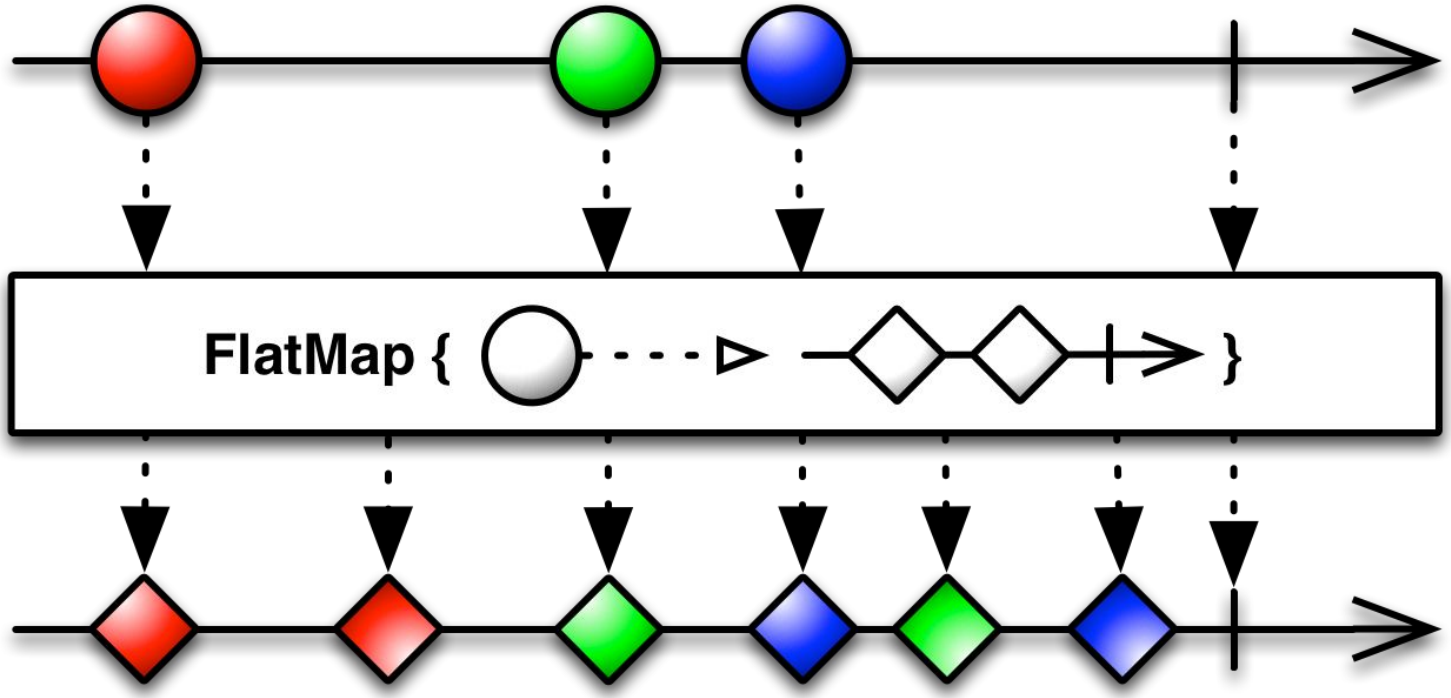
ReactiveX Marbles



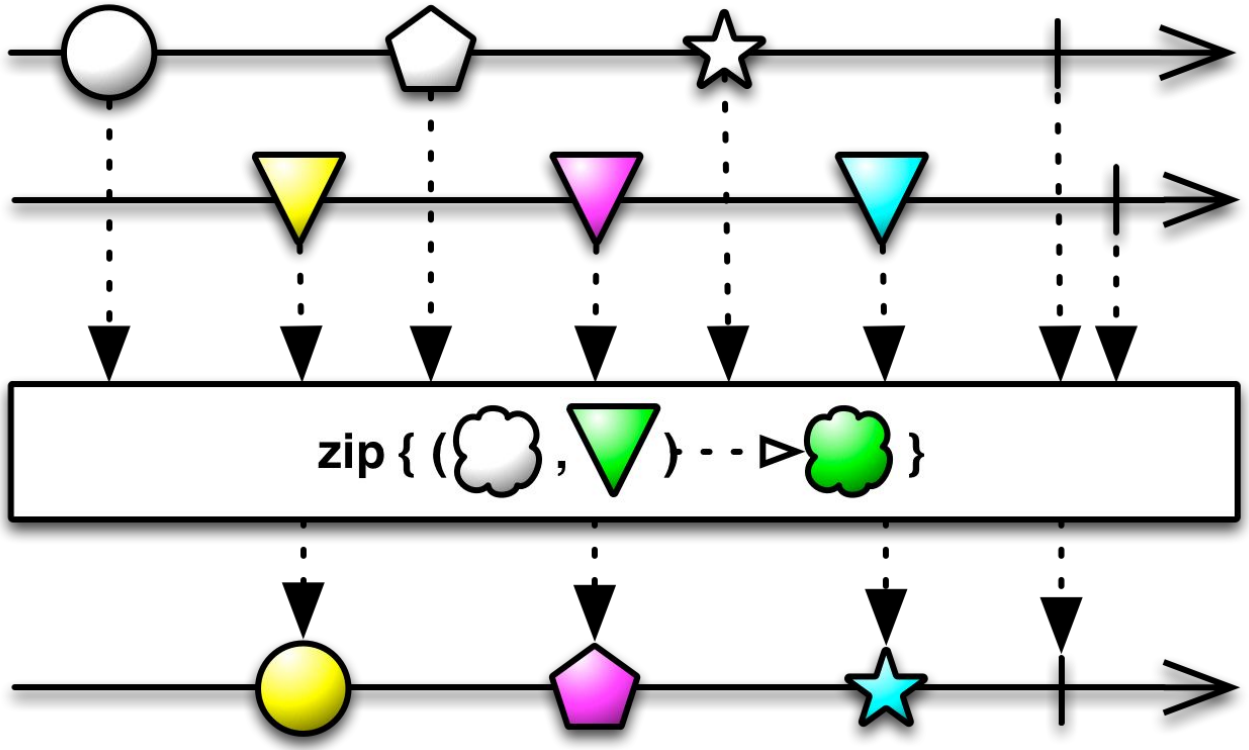
Оператор Map



Оператор FlatMap



Оператор Zip



Пример цепочки

```
Observable.zip(requestCards(userId),requestAccounts(userId),
                (cards, accounts) -> collate(cards, accounts))
```

Пример цепочки

```
Observable.zip(requestCards(userId),requestAccounts(userId),
                    (cards, accounts) -> collate(cards, accounts))
              .flatMap(sources -> loadPreferences(sources))
```

Пример цепочки

```
Observable.zip(requestCards(userId),requestAccounts(userId),
                    (cards, accounts) -> collate(cards, accounts))
                .flatMap(sources -> loadPreferences(sources))
                .map(preferedSources -> range(preferedSources))
```

Пример цепочки

```
Observable.zip(requestCards(userId),requestAccounts(userId),
                    (cards, accounts) -> collate(cards, accounts))
                .flatMap(sources -> loadPreferences(sources))
                .map(preferedSources -> range(preferedSources))
                .onErrorResumeNext(err -> {
                    log.error("Some error", err);
                    return loadFromCache(user);
                })
```

Пример цепочки

```
Observable.zip(requestCards(userId),requestAccounts(userId),
                    (cards, accounts) -> collate(cards, accounts))
          .flatMap(sources -> loadPreferences(sources))
          .map(preferedSources -> range(preferedSources))
          .onErrorResumeNext(err -> {
              log.error("Some error", err);
              return loadFromCache(user);
          })
          .subscribe(result -> render(result));
```

Сюрприз

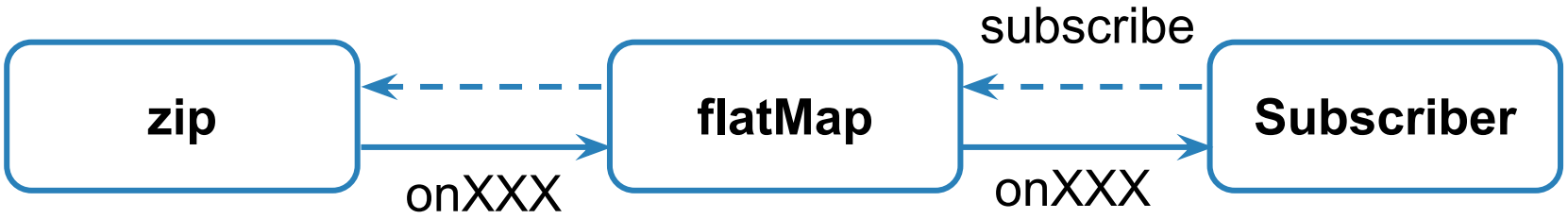
По умолчанию observable-цепочка исполняется в текущем потоке.

Сюрприз

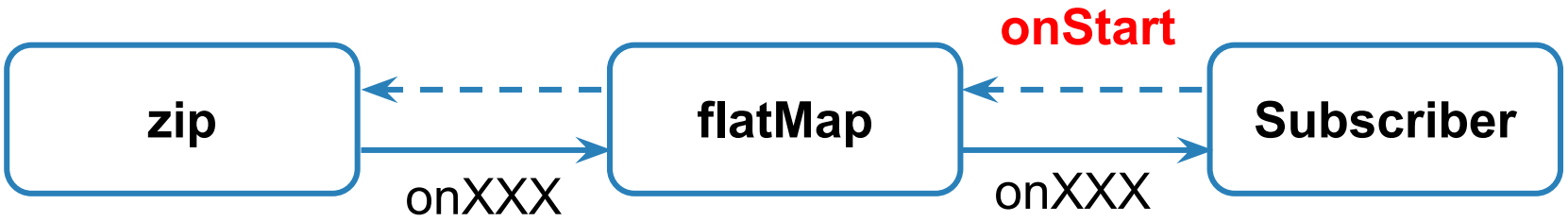
По умолчанию observable-цепочка исполняется в текущем потоке.

Да-да и даже zip!

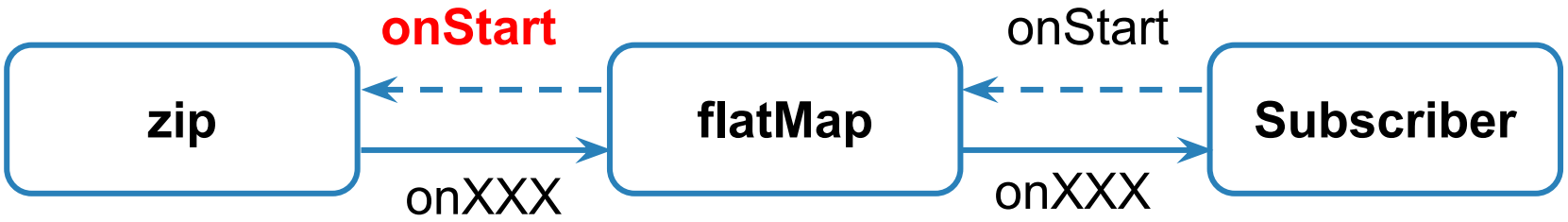
Как это работает



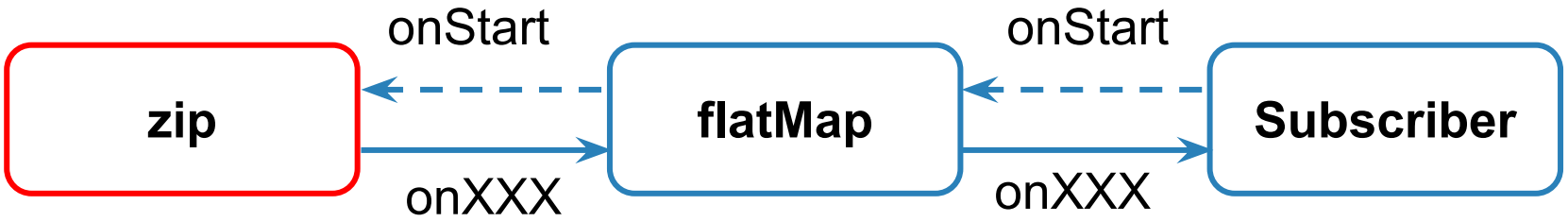
Как это работает



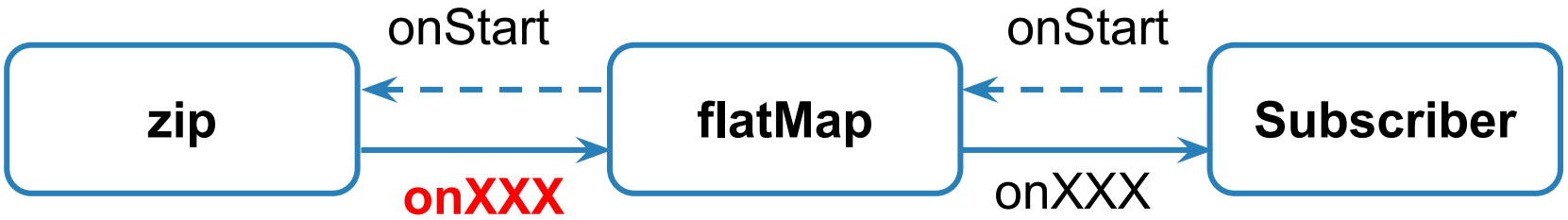
Как это работает



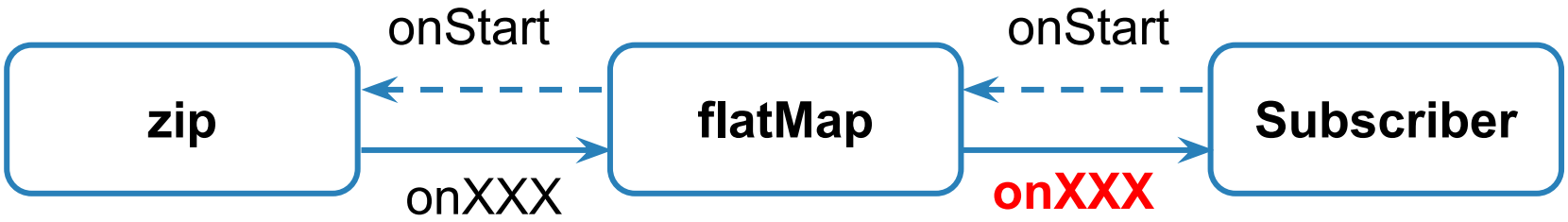
Как это работает



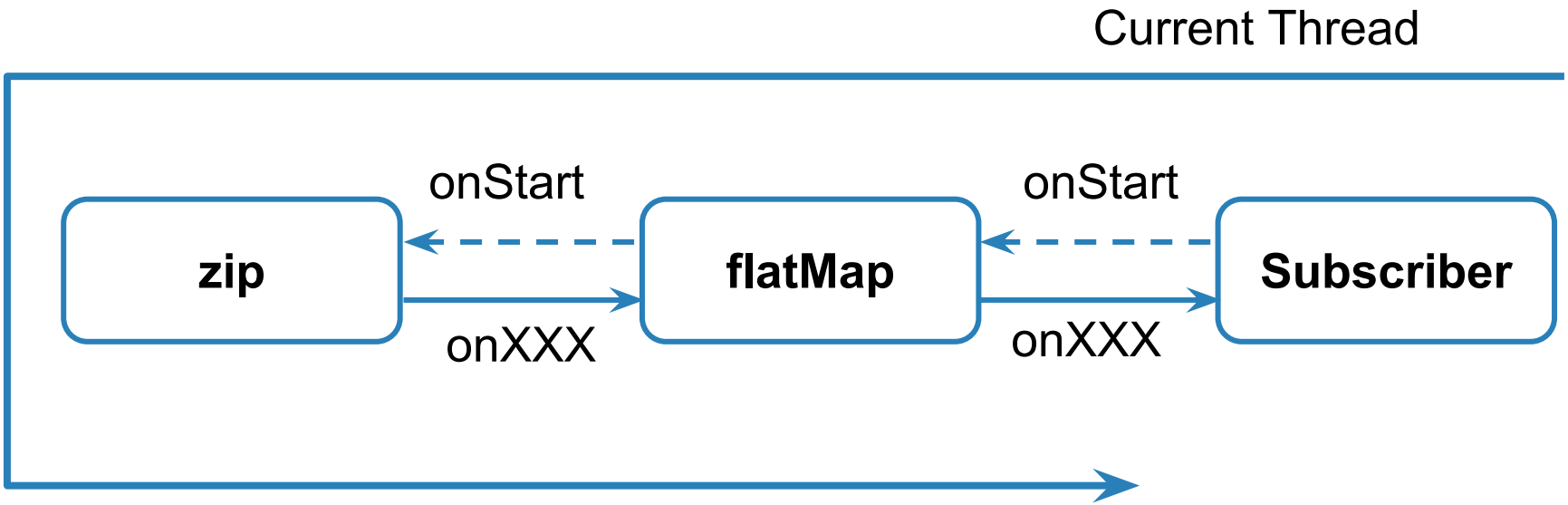
Как это работает

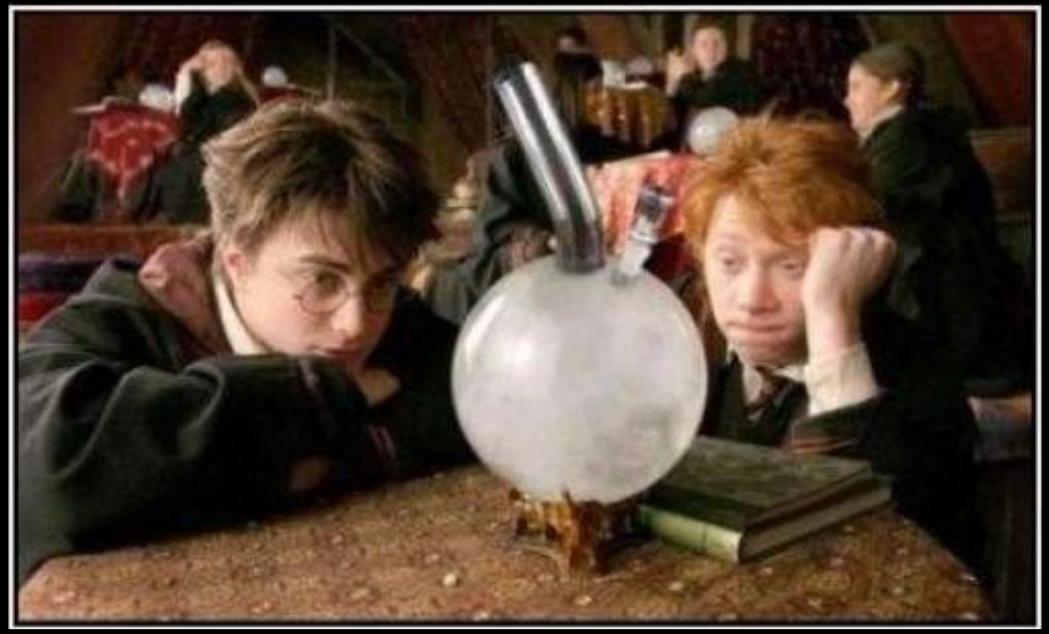


Как это работает



Работает на текущем потоке





Magic is impossible

without special equipment

Подключаем потоки

- subscribeOn
- observeOn

Подключаем потоки

- **subscribeOn**
- observeOn

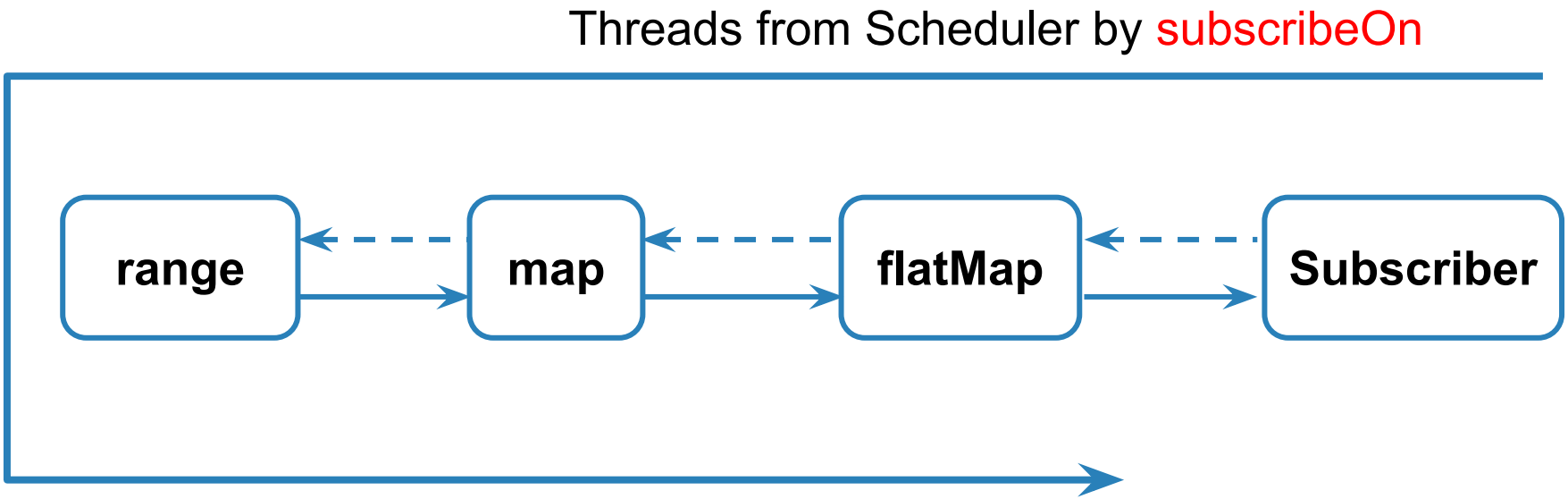
SubscribeOn. Пример

```
Observable.range(1,1000)
          .map(res -> res + 1)
          .flatMap(res -> Observable.just(res))
          .subscribeOn(Schedulers.computation())
          .subscribe(result -> System.out.println(result))
```

SubscribeOn. Изнутри

```
ExecutorService exec = Executors.newFixedThreadPool(2);

Subscriber<T> subscribeOn = o -> {
    exec.submit(() -> sourceSubject.subscribe(o));
};
```

SubscribeOn. Результат



SubscribeOn. Дважды

```
Observable.range(1,1000)
          .subscribeOn(Schedulers.io())
          .map(res -> res + 1)
          .flatMap(res -> Observable.just(res))
          .subscribeOn(Schedulers.computation())
          .subscribe(result -> System.out.println(result))
```

SubscribeOn. Изнутри

```
ExecutorService exec = Executors.newFixedThreadPool(2);
ExecutorService exec2 = Executors.newFixedThreadPool(2);

Subscriber<T> subscribeOn = o -> {
 exec2.submit(() ->
     exec.submit(() -> sourceSubject.subscribe(o))
 );
};
```

SubscribeOn. Дважды

```
Observable.range(1,1000)
          .subscribeOn(Schedulers.io())
          .map(res -> res + 1)
          .flatMap(res -> Observable.just(res))
          .subscribeOn(Schedulers.computation())
          .subscribe(result -> System.out.println(result))
```

SubscribeOn. Рекомендации

Не стоит добавлять subscribeOn внутри своих библиотек, так как пользователи не смогут переопределить заданное поведение.

Подключаем потоки

- subscribeOn
- **observeOn**

ObserveOn. Пример

```
Observable.range(1,1000)
          .map(res -> res + 1)
          .observeOn(Schedulers.io())
          .flatMap(res -> Observable.just(res))
          .subscribeOn(Schedulers.computation())
          .subscribe(result -> System.out.println(result))
```

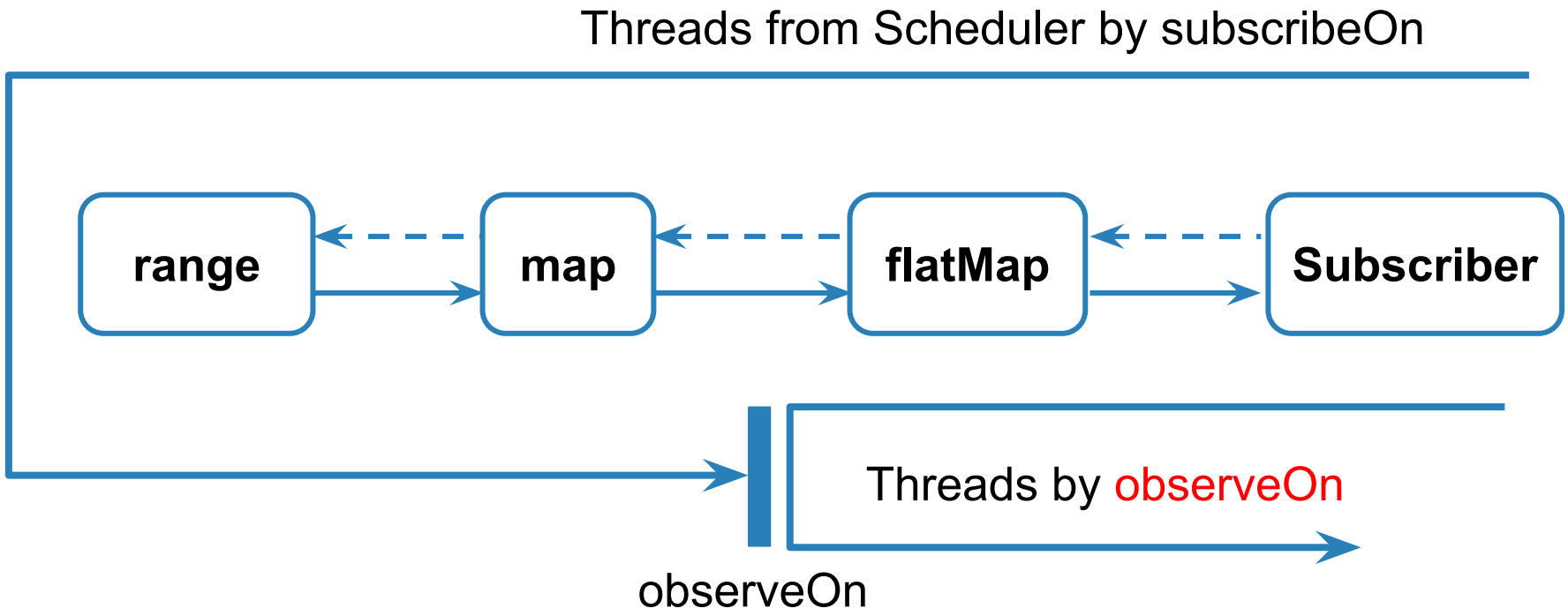
ObserveOn. Изнутри

```
ExecutorService exec = Executors.newFixedThreadPool(2);

Subscriber<T> observeOn = o -> {
   sourceSubject.subscribe(new Observer<T>() {
     @Override
     public void onNext(T t) {
       exec.submit(() -> o.onNext(t));
     }
     ...
 });
};
```

ObserveOn. Пример

```
Observable.range(1,1000)
          .map(res -> res + 1)
          .observeOn(Schedulers.io)
          .flatMap(res -> Observable.just(res))
          .subscribeOn(Schedulers.computation())
          .subscribe(result -> System.out.println(result))
```

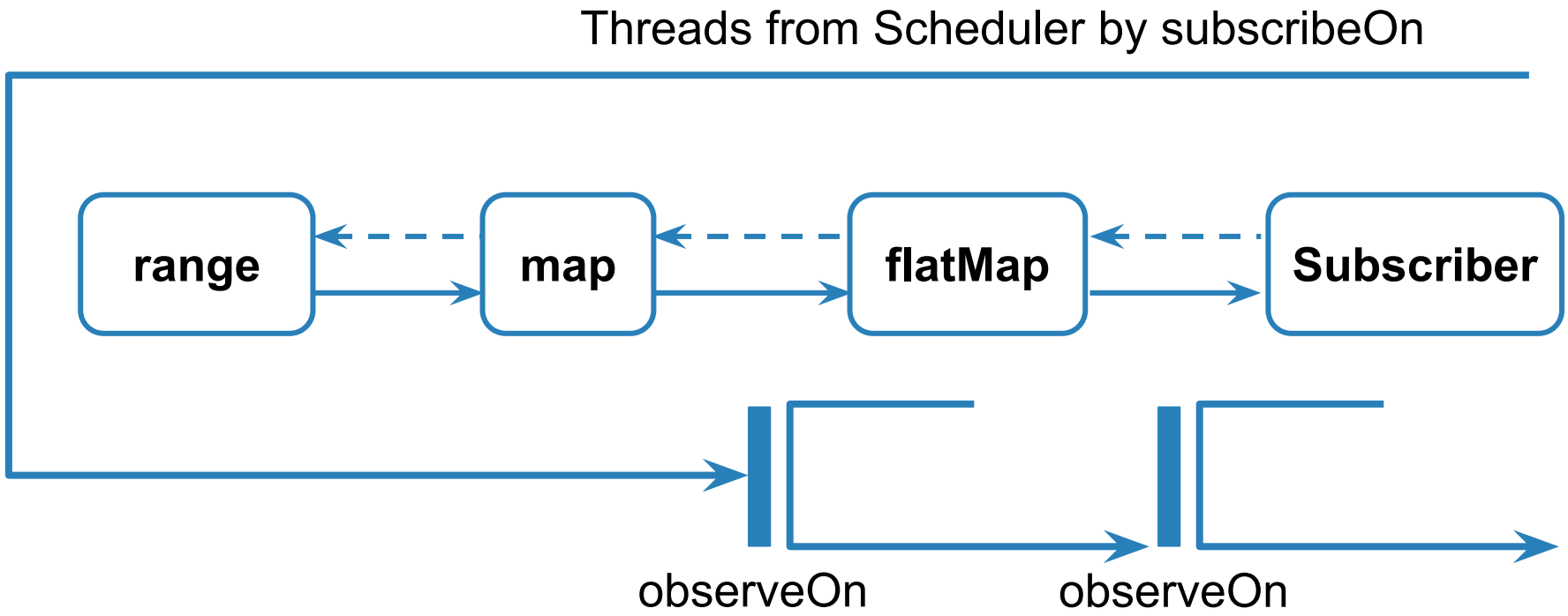
ObserveOn. Результат



ObserveOn. Дважды

```
Observable.range(1,1000)
          .map(res -> res + 1)
          .observeOn(Schedulers.io())
          .flatMap(res -> Observable.just(res))
          .subscribeOn(Schedulers.computation())
          .observeOn(Schedulers.newThread())
          .subscribe(result -> System.out.println(result))
```

ObserveOn. Разбиваем дальше



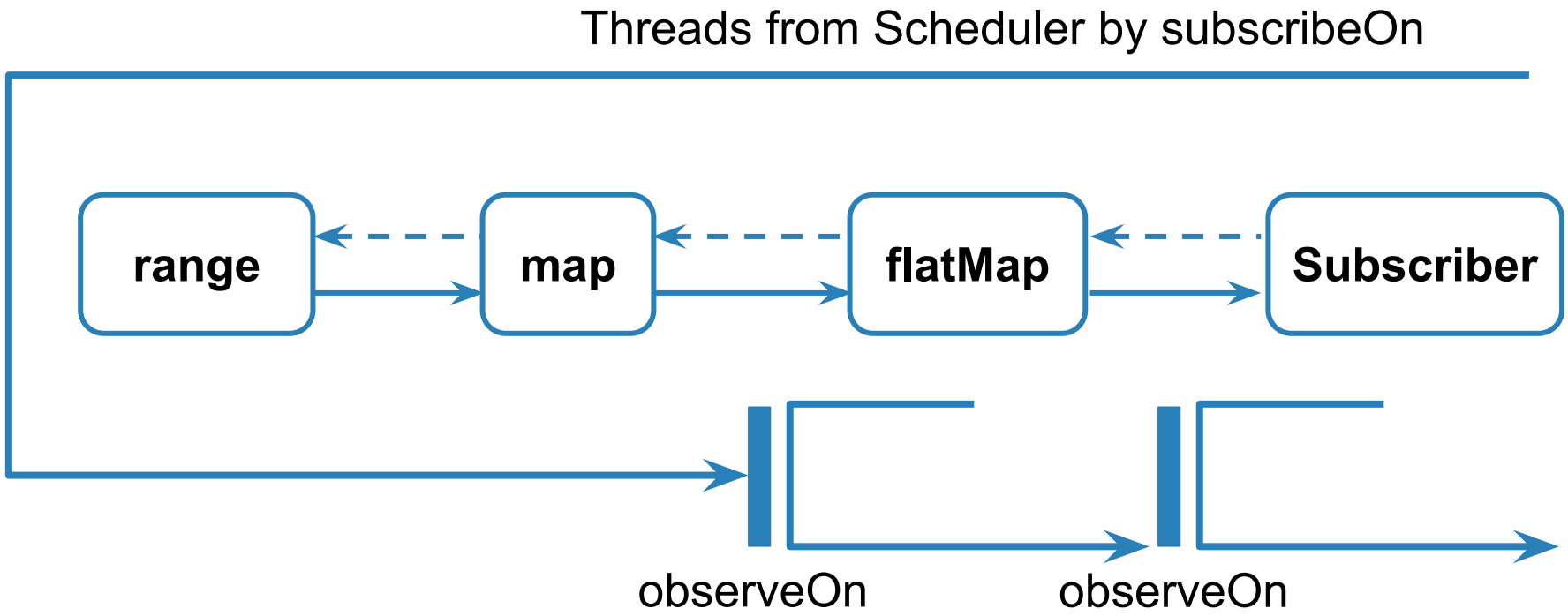
Роль методов

- subscribeOn - отвечает за pool, на котором исполняется Producer

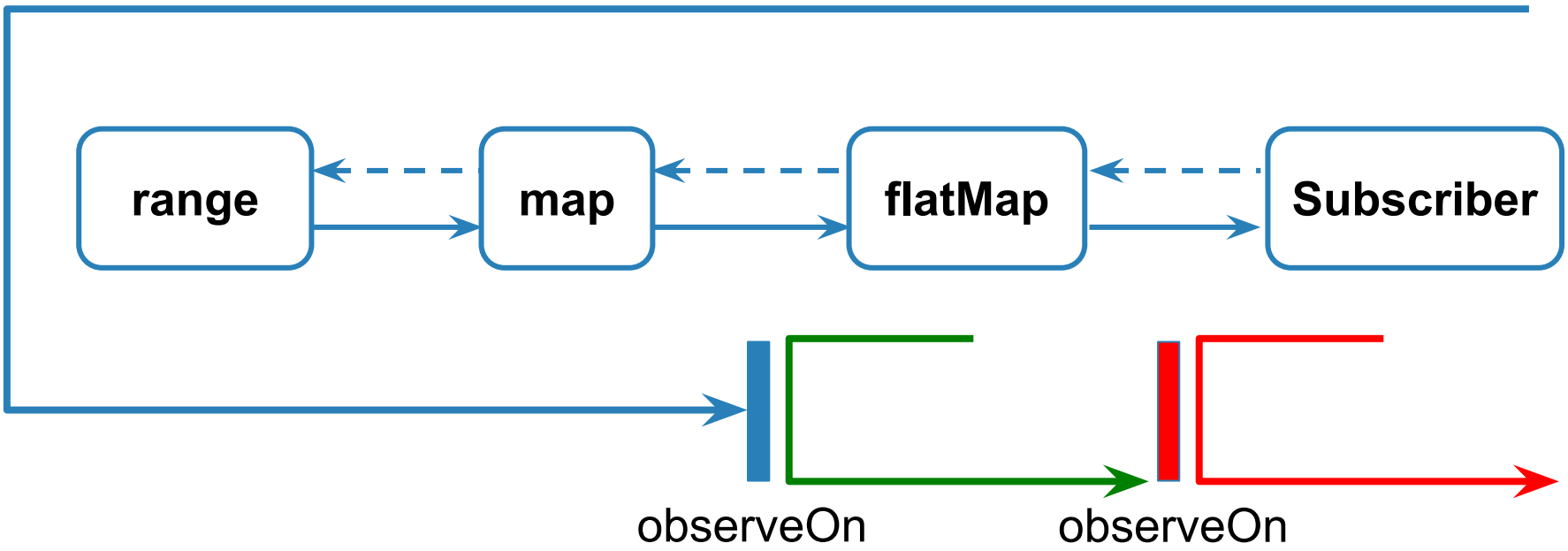
Роль методов

- subscribeOn - отвечает за pool, на котором исполняется Producer
- observeOn - отвечает за pool, на котором исполняются операторы и подписчик

Ситуация


Threads from Scheduler by subscribeOn
range
map
flatMap
Subscriber
observeOn
observeOn

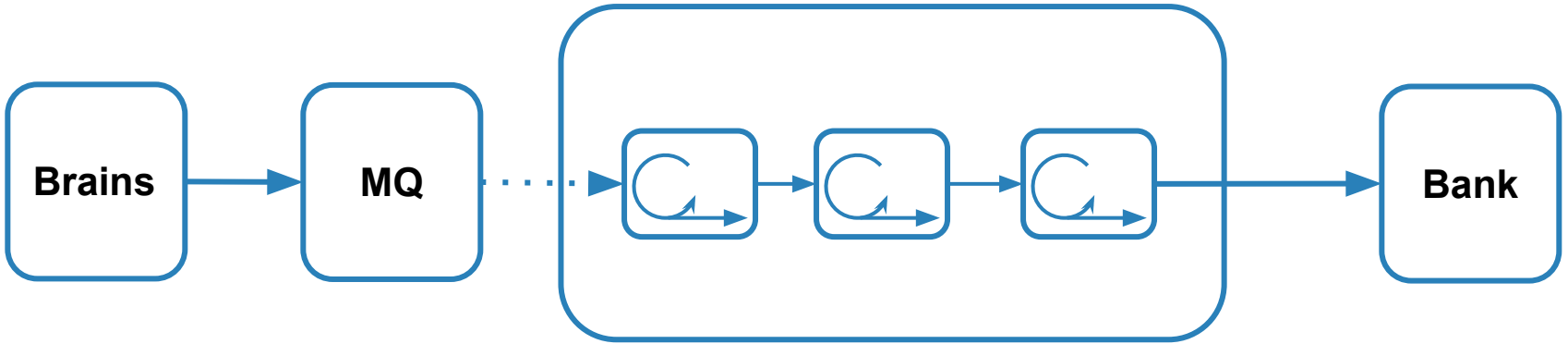
Подписчик может не успевать



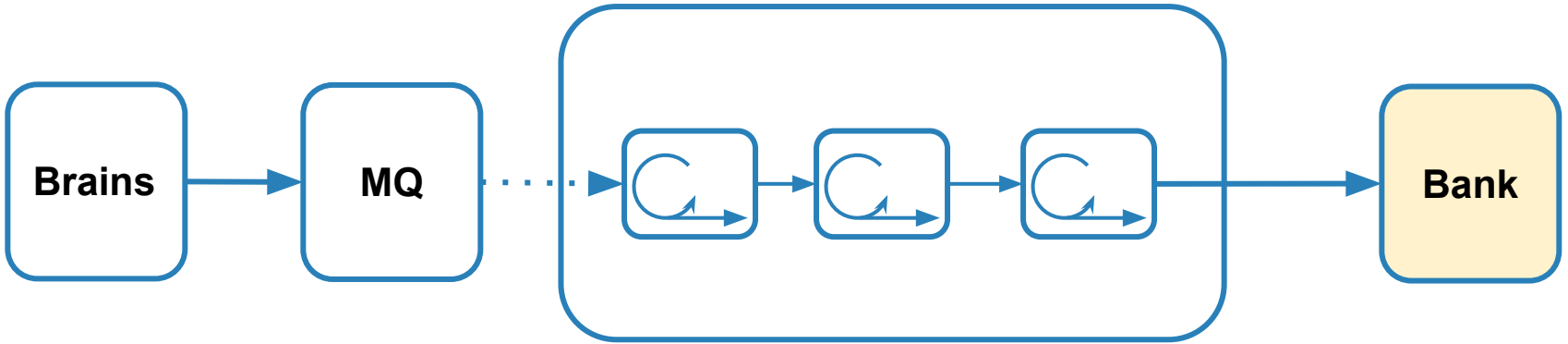
Backpressure

Механизм для балансировки нагрузки без блокировок и сброса событий

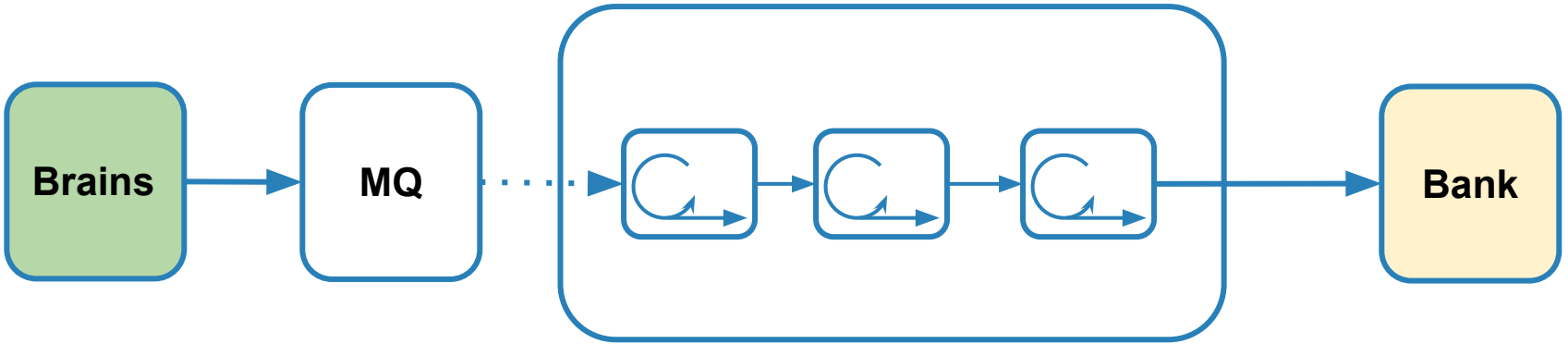
Backpressure. Пример



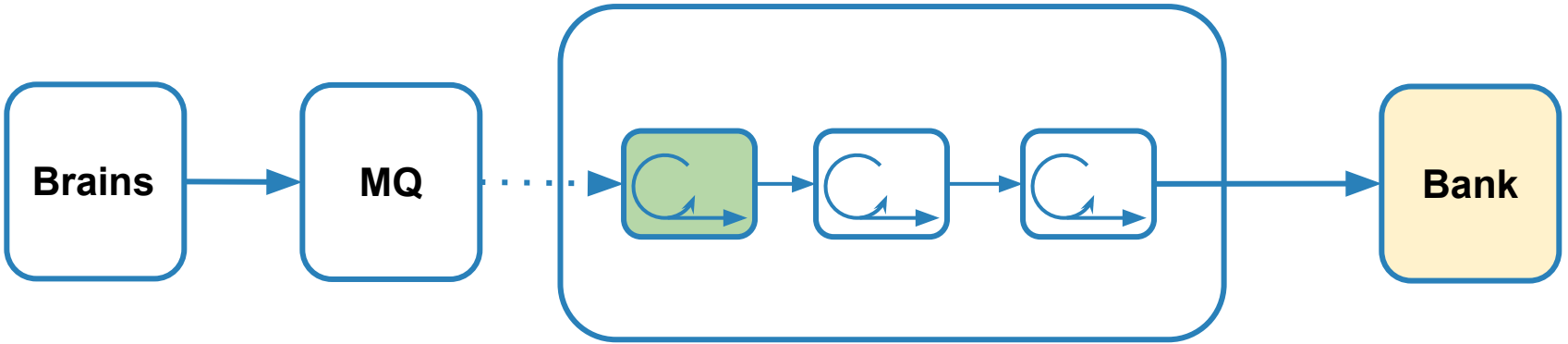
Backpressure. Пример



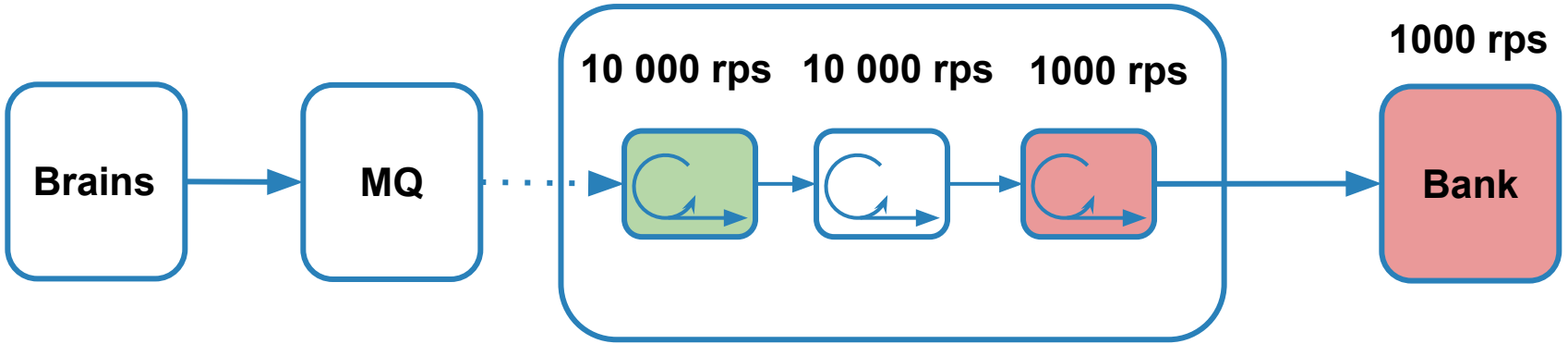
Backpressure. Пример



Backpressure. Пример



Backpressure. Пример

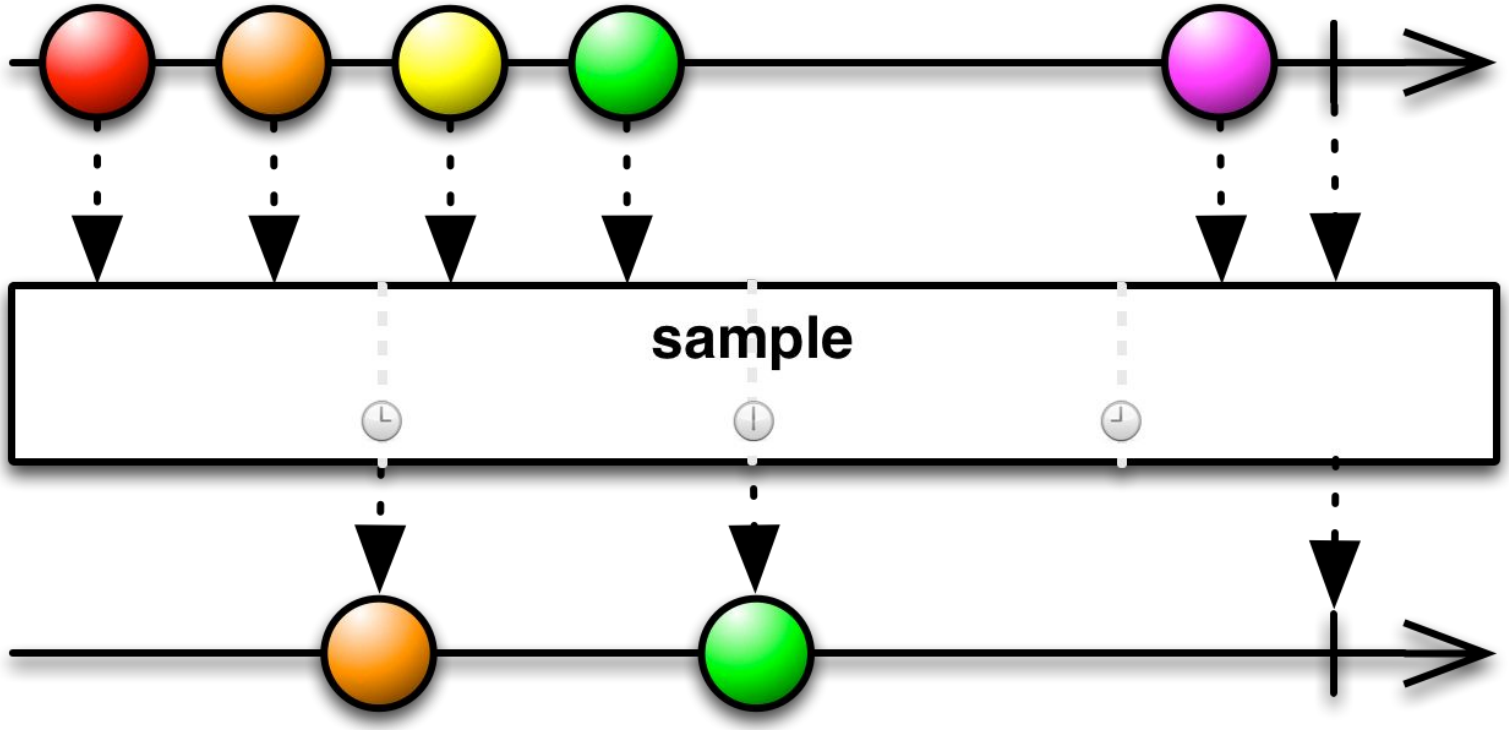


Нет времени разбираться!

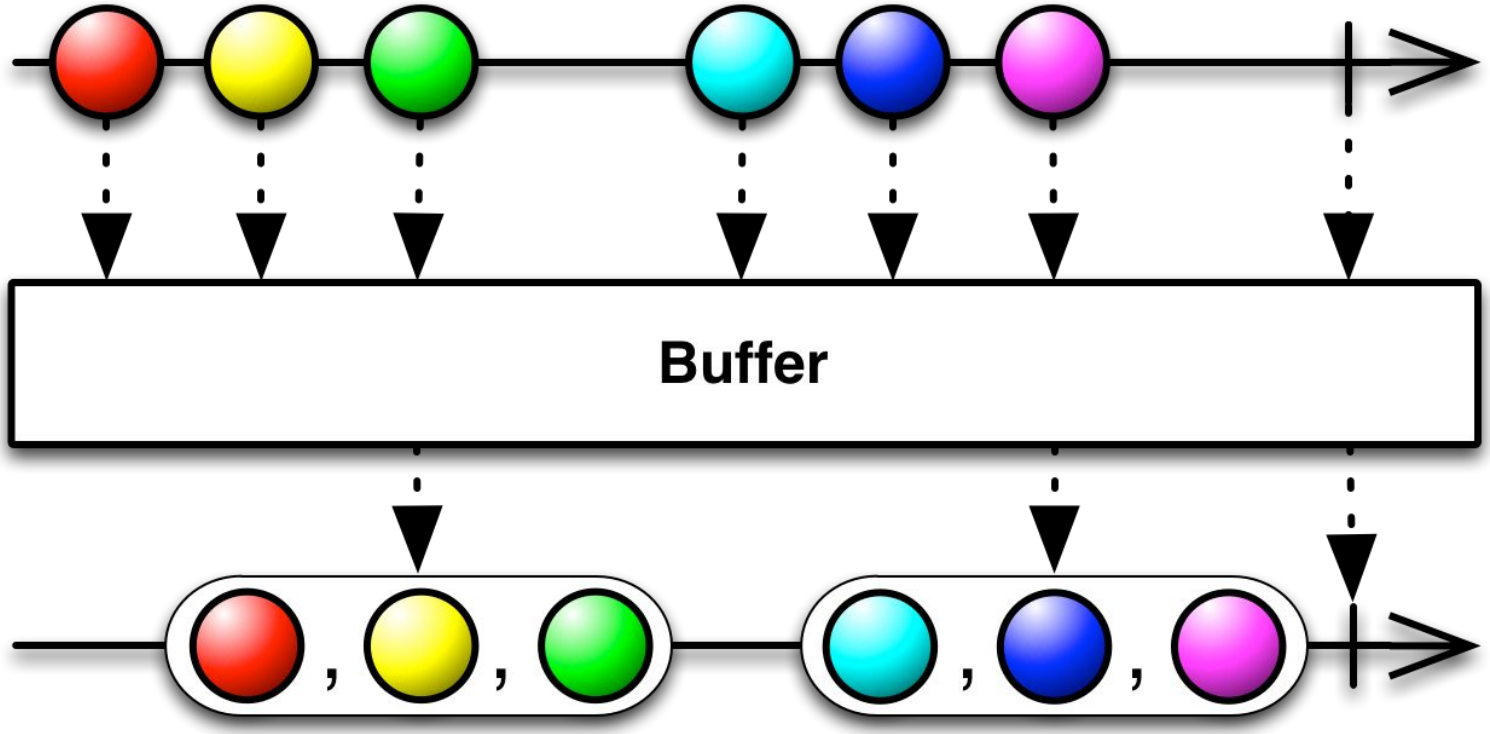
Есть варианты

- Sample (Throttling)
- Buffer
- Window
- ...

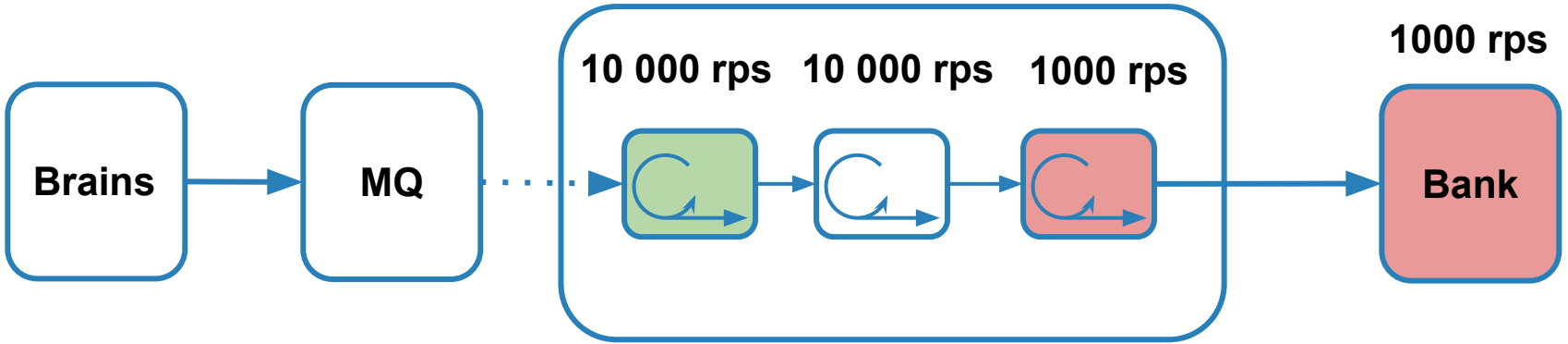
Sample(Throttling)



Buffer



Backpressure. Пример



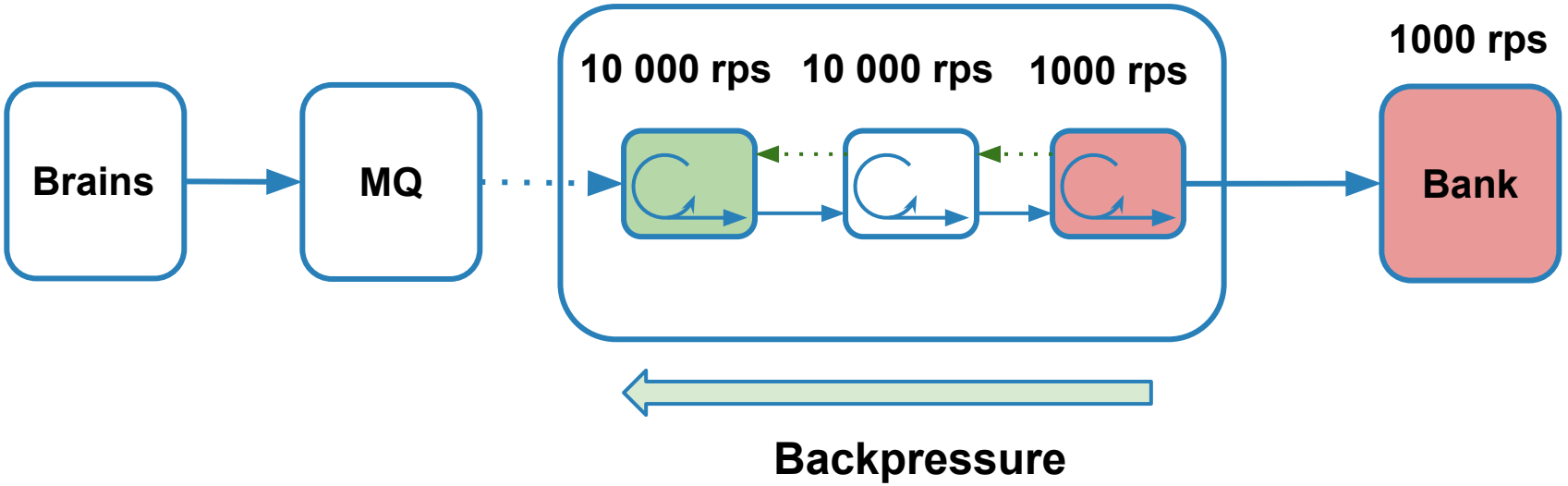
Backpressure. Pull-based

```
sourceSubject.subscribe(new Subscriber() {

 @Override
 public void onNext(Object o) {
   //process signal
   request(10);
 }
});
```

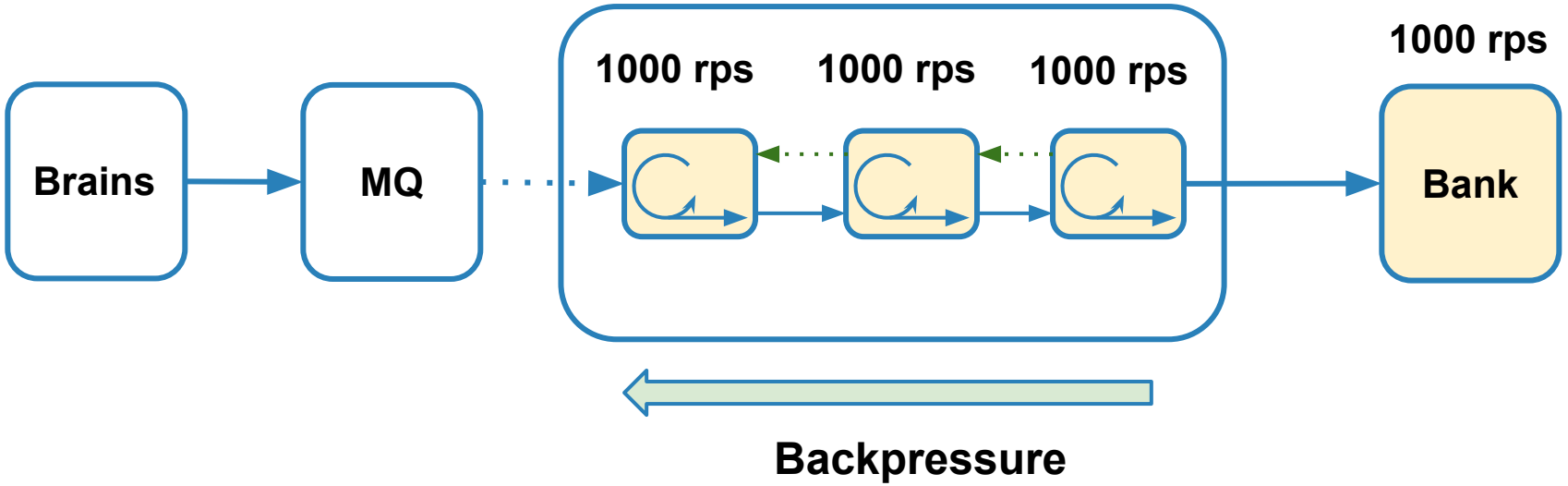
Subscribe. Push-based

```
class Subscriber() {

 @Override
 public void onStart(Object o) {
   request(Long.MAX_VALUE);
 }
});
```

Backpressure. Пример



Backpressure. Пример



Основы изучены

- Написание простейших цепочек
- Распределение работы по потокам
- Балансировка нагрузки через backpressure

Немного углубимся

В большинстве случаев вам хватит стандартного набора операторов, но рано или поздно захочется расширить функционал.

Observable.Operator

```
Observable.Operator<Integer, Integer> operator = subscriber -> {
 new Subscriber<Integer>() {
   @Override
   public void onCompleted() {

   }
   @Override
   public void onError(Throwable e) {

   }
   @Override
   public void onNext(Integer integer) {
      //do some work
   }
 };
```

Пишем свой оператор

```
Observable.Operator<Integer, Integer> take100 = subscriber -> {

      return new Subscriber<Integer>() {
          public void onNext(Integer el) {
            if(el < 100)
              subscriber.onNext(el);
            else
              onCompleted();
          }

      …
};
```

Встраиваем оператор

```
Observable.range(1,1000)
    .lift(take100)
    .observeOn(Schedulers.newThread())
    .flatMap(res -> someBlockingCall())
    .subscribe(result -> System.out.println(result));
```

Работает!

Еще и многопоточно!

Немного подправим оператор

```
Observable.range(1,1000)
    .lift(take200)
    .observeOn(Schedulers.newThread())
    .flatMap(res -> someBlockingCall(res))
    .subscribe(res -> System.out.println(res));
```

Упс!

rx.exceptions.MissingBackpressureException

Ищем плавающие баги

Разрыв обратной связи

Исправляем оператор

```
Observable.Operator<Integer, Integer> take100 = subscriber -> {

     return new Subscriber<Integer>(subscriber) {
         public void onNext(Integer el) {
           if(el < 200)
             subscriber.onNext(el);
           else
             onCompleted();
         }


     …
};
```

А почему работало?

Внутри обертки OperatorObserveOn есть Queue размером в 128 элементов, потому что backpressure был не всегда.

А почему 128?

Читайте benchmark в комментариях к очереди в observeOn :)

Оператор пропуска нечетных

```
Observable.Operator<Integer, Integer> skipOdd = subs -> {
 return new Subscriber<Integer>(subs) {

   public void onNext(Integer i) {
     if (i % 2 == 0)
       subs.onNext(i);
   }
  ...
};
```

Подключаем оператор

```
Observable.Operator<Integer, Integer> skipOdd = subs -> {
 return new Subscriber<Integer>(subs) {

   public void onNext(Integer i) {
     if (i % 2 == 0)
       subs.onNext(i);
   }
  ...
};

Observable.range(1, 100)
   .lift(skipOdd)
   .take(10)
   .subscribe(res -> System.out.println(res));
```

Вывод. Чего-то не хватает

```
2
4
6
8
10
```

Вспоминаем про backpressure

```
Observable.Operator<Integer, Integer> skipOdd = subs -> {
 return new Subscriber<Integer>(subs) {

   public void onNext(Integer i) {
     if (i % 2 == 0)
       subs.onNext(i);
     else
       request(1);
   }
  ...
};
```

Продолжаем писать оператор

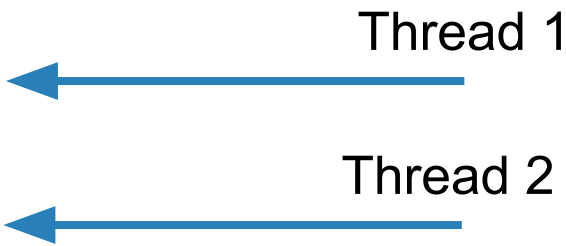
```
Operator<Integer, Integer> take200 = subscriber -> {

      return new Subscriber<Integer>() {
          public void onNext(Integer el) {
            if(el < 200)
              subscriber.onNext(el);
            else
              onCompleted();
          }
          public void onCompleted() {
            subscriber.onCompleted();
            unsubscribe();
          }
      …
};
```

Есть потоки, есть проблемы

```
Operator<Integer, Integer> take200 = subscriber -> {

      return new Subscriber<Integer>() {
          public void onNext(Integer el) {
            if(el < 200)
              subscriber.onNext(el);
            else
              onCompleted();
          }
          public void onCompleted() {
            subscriber.onCompleted();
            unsubscribe();
          }
      …
};
```

Есть потоки, есть проблемы

```
Operator<Integer, Integer> take200 = subscriber -> {

      return new Subscriber<Integer>() {
          public void onNext(Integer el) {
            if(el < 200)
              subscriber.onNext(el);
            else
              onCompleted();
          }
          public void onCompleted() {
            subscriber.onCompleted();
            unsubscribe();
          }
      …
};
```



Гарантии #1-2

- События onNext, onError и onComplete должны быть доставлены в Subscriber последовательно

Гарантии #1-2

- События onNext, onError и onComplete должны быть доставлены в Subscriber последовательно

- События onError и onCompleted терминальны

Добавляем сериализацию

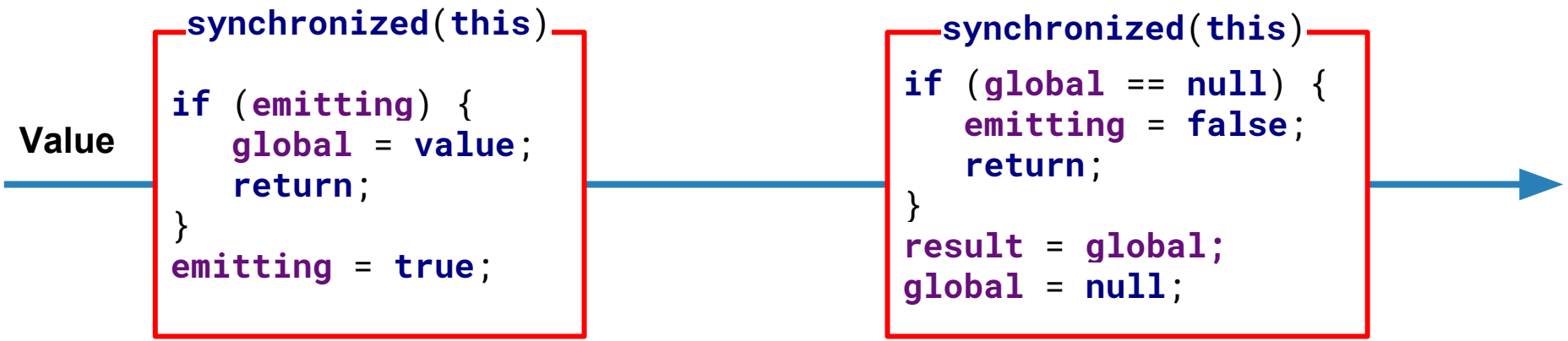
```
Operator<Integer, Integer> take100 = c -> {
     Subscriber<Integer> subscriber = new SerializedSubscriber<>(c);
     return new Subscriber<Integer>() {
         public void onNext(Integer el) {
           if(el < 100)
             subscriber.onNext(el);
           else
             onCompleted();
         }
         public void onCompleted() {
           subscriber.onCompleted();
           unsubscribe();
         }
     …
};
```

Есть потоки, нет проблем

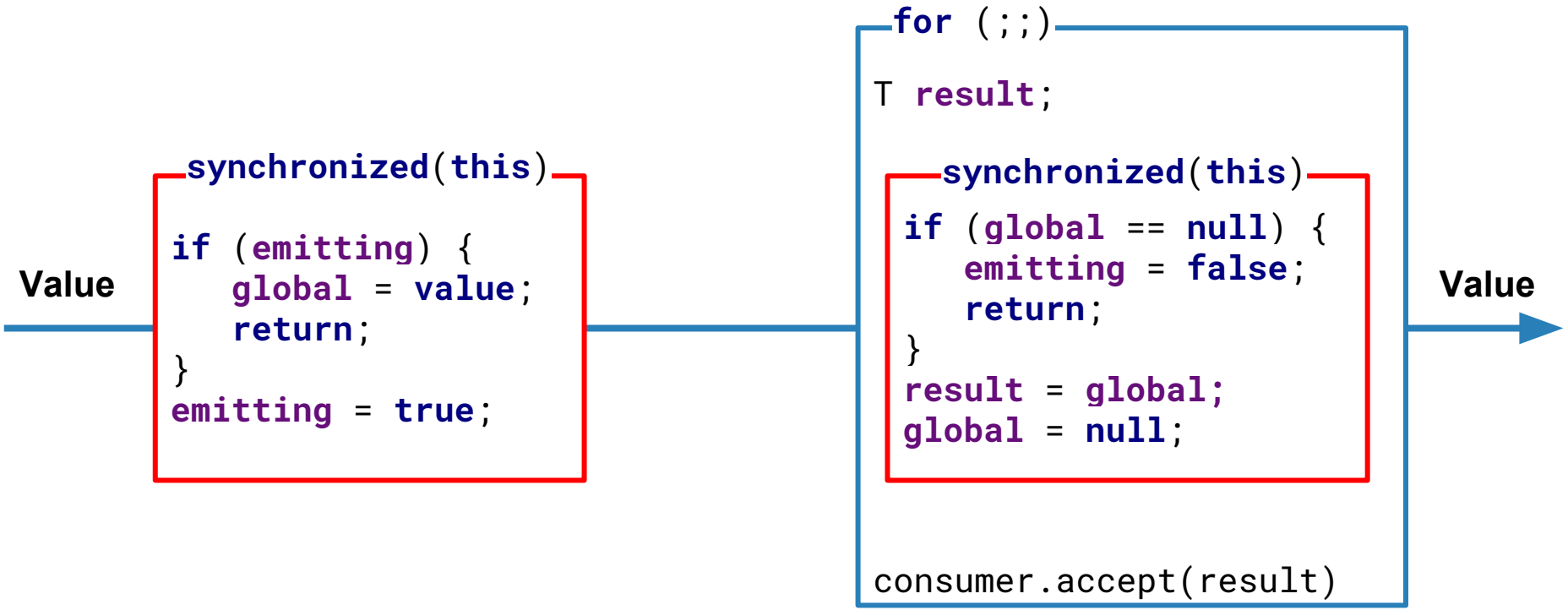
```
Observable.range(1,1000)
          .lift(take100)
          .flatMap(sources -> loadPreferences(sources))
          .observeOn(Schedulers.io())
          .subscribe(res -> System.out.println(res))
```

Serialization

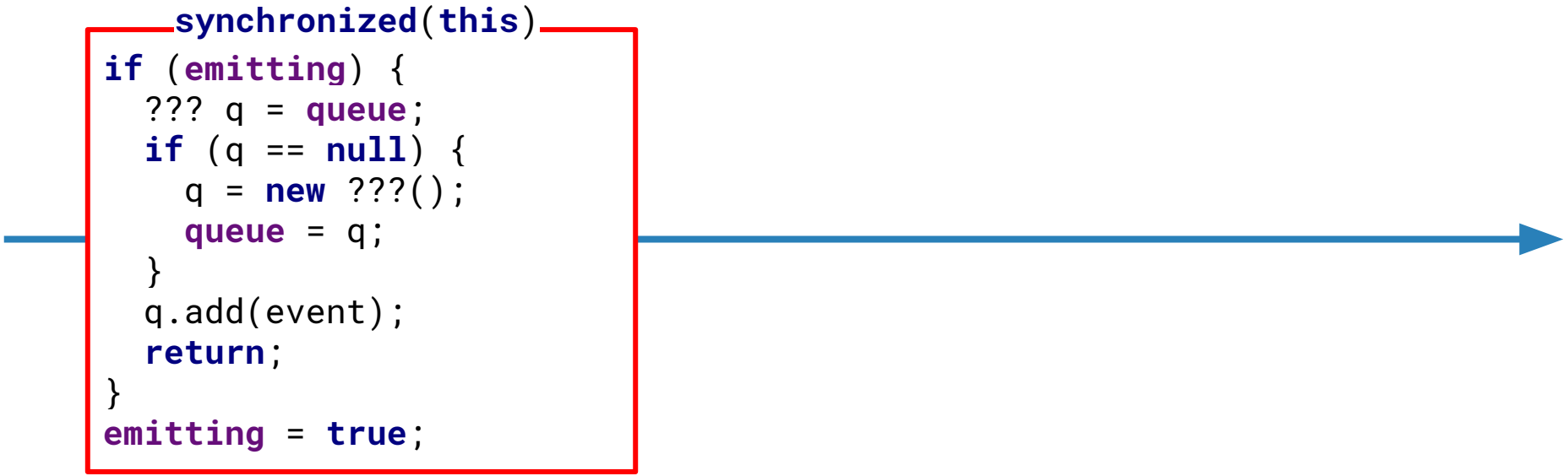
Serialization



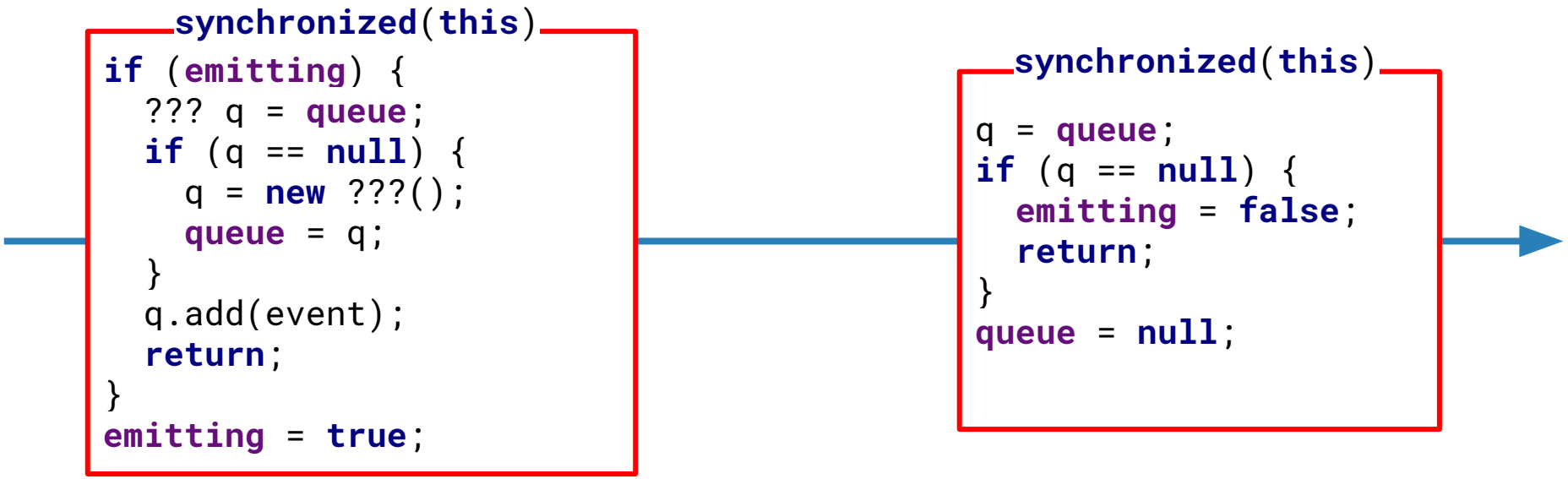
Serialization



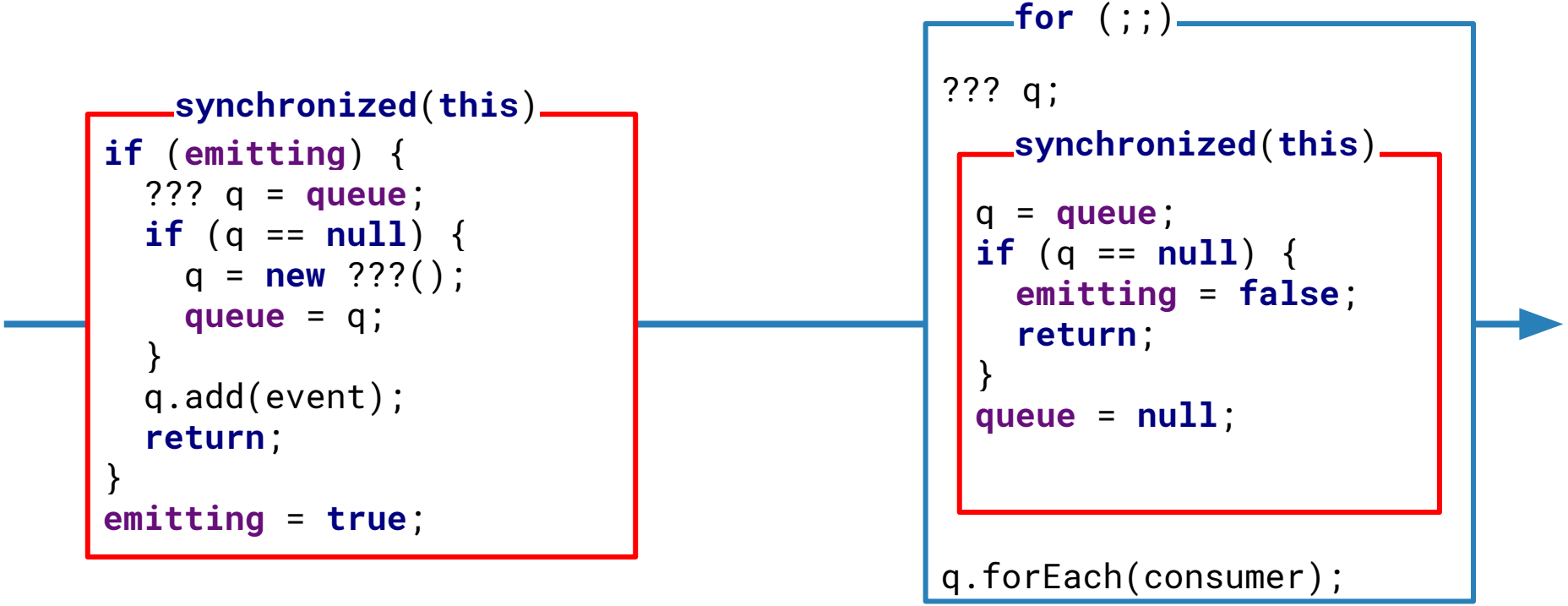
Serialization



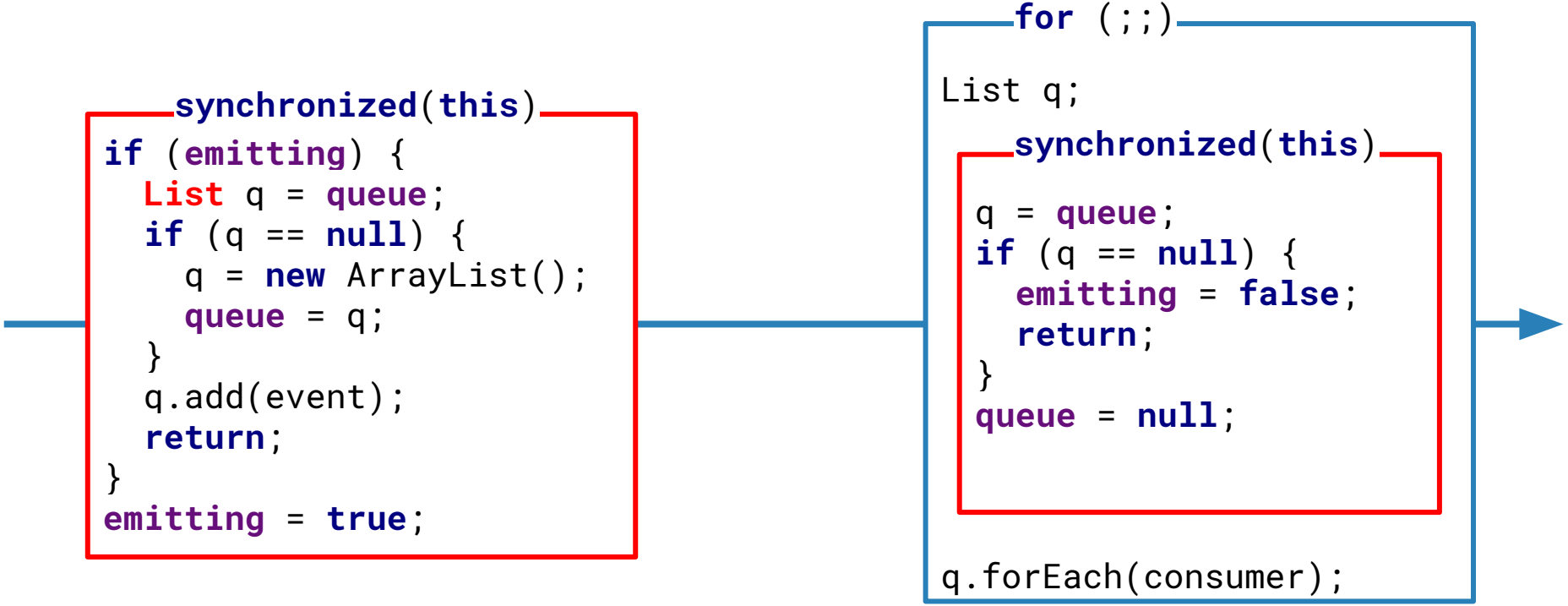
Serialization



Serialization



Serialization



Serialization. Emitter-loop

Serialization. Emitter-loop

- Блокирующий

Serialization. Emitter-loop

- − Блокирующий
- + Могут происходить оптимизации Biased locking и Lock elision

Serialization. Emitter-loop

- − Блокирующий
- \+ Могут происходить оптимизации Biased locking и Lock elision
- ● Работает внутри RxJava version 1.X

Serialization. Queue-drain

```
final Queue<T> queue = new ????Queue<>();
final AtomicInteger wip = new AtomicInteger();

public void drain(T value) {
 queue.offer(Objects.requireNonNull(value));     // (1)
 if (wip.getAndIncrement() == 0) {
   do {
     wip.set(1);
     T v;
     while ((v = queue.poll()) != null) {
       consumer.accept(v);
     }
   } while (wip.decrementAndGet() != 0);
 }
}
```

Serialization. Queue-drain

```
final Queue<T> queue = new ????Queue<>();
final AtomicInteger wip = new AtomicInteger();

public void drain(T value) {
 queue.offer(Objects.requireNonNull(value));
 if (wip.getAndIncrement() == 0) {
   do {
     wip.set(1);
     T v;
     while ((v = queue.poll()) != null) {
       consumer.accept(v);
     }
   } while (wip.decrementAndGet() != 0);
 }
}
```

Serialization. Queue-drain

```
final Queue<T> queue = new ????Queue<>();
final AtomicInteger wip = new AtomicInteger();

public void drain(T value) {
 queue.offer(Objects.requireNonNull(value));
 if (wip.getAndIncrement() == 0) {
   do {
     wip.set(1);
     T v;
     while ((v = queue.poll()) != null) {
       consumer.accept(v);
     }
   } while (wip.decrementAndGet() != 0);
 }
}
```

Serialization. Queue-drain

```
final Queue<T> queue = new ????Queue<>();
final AtomicInteger wip = new AtomicInteger();

public void drain(T value) {
 queue.offer(Objects.requireNonNull(value));
 if (wip.getAndIncrement() == 0) {
   do {
     wip.set(1);
     T v;
     while ((v = queue.poll()) != null) {
       consumer.accept(v);
     }
   } while (wip.decrementAndGet() != 0);
 }
}
```

Serialization. Queue-drain

```
final Queue<T> queue = new ????Queue<>();
final AtomicInteger wip = new AtomicInteger();

public void drain(T value) {
 queue.offer(Objects.requireNonNull(value));
 if (wip.getAndIncrement() == 0) {
   do {
     wip.set(1);
     T v;
     while ((v = queue.poll()) != null) {
       consumer.accept(v);
     }
   } while (wip.decrementAndGet() != 0);
 }
}
```

Serialization. Queue-drain

```
final Queue<T> queue = new ????Queue<>();
final AtomicInteger wip = new AtomicInteger();

public void drain(T value) {
 queue.offer(Objects.requireNonNull(value));
 if (wip.getAndIncrement() == 0) {
   do {
     wip.set(1);
     T v;
     while ((v = queue.poll()) != null) {
       consumer.accept(v);
     }
   } while (wip.decrementAndGet() != 0);
 }
}
```

Serialization. Queue-drain

```
final Queue<T> queue = new MpscLinkedQueue<>();
final AtomicInteger wip = new AtomicInteger();

public void drain(T value) {
 queue.offer(Objects.requireNonNull(value));
 if (wip.getAndIncrement() == 0) {
   do {
     wip.set(1);
     T v;
     while ((v = queue.poll()) != null) {
       consumer.accept(v);
     }
   } while (wip.decrementAndGet() != 0);
 }
}
```

MpscLinkedQueue

- Multiple producer Single consumer очередь из пакета JCTools

MpscLinkedQueue

- Multiple producer Single consumer очередь из пакета JCTools
- Wait-free offer и Lock-free poll

MpscLinkedQueue. Почитать

Porting Pitfalls:

Turning D.Vyukov MPSC Wait-free queue into a j.u.Queue

Serialization. Queue-drain

+ Почти неблокирующий

Serialization. Queue-drain

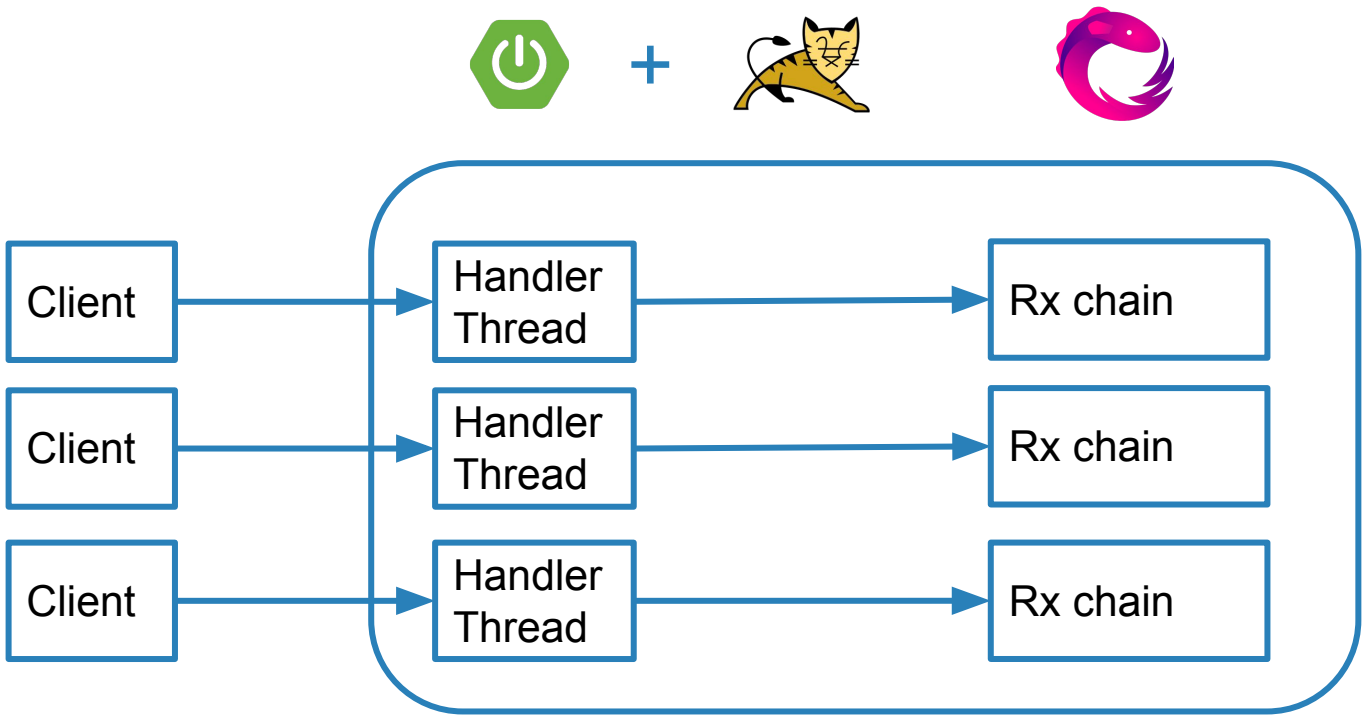
+ Почти неблокирующий
- Работает внутри RxJava version 2.X

Выдохнули :)

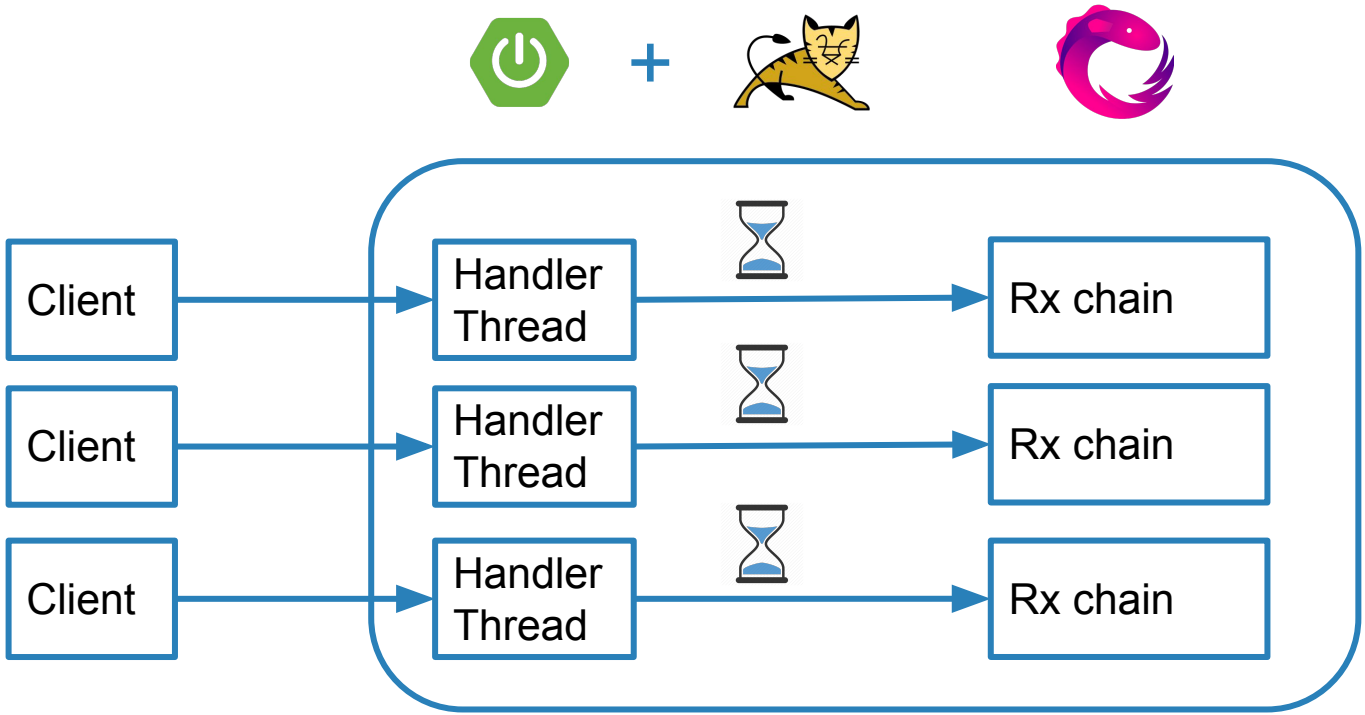
- Знаем, как выглядит Rx снаружи
- Понимаем, что внутри Rx

А как же передача данных?

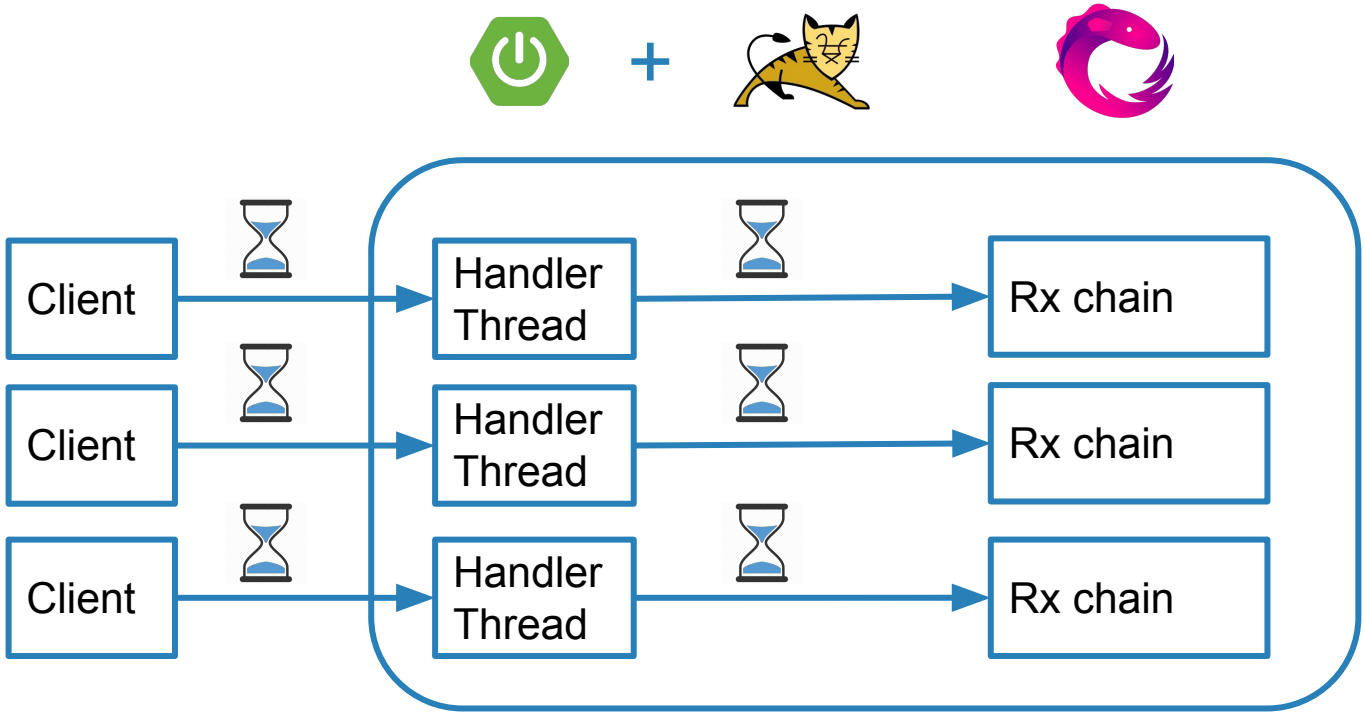
ThreadPerRequest



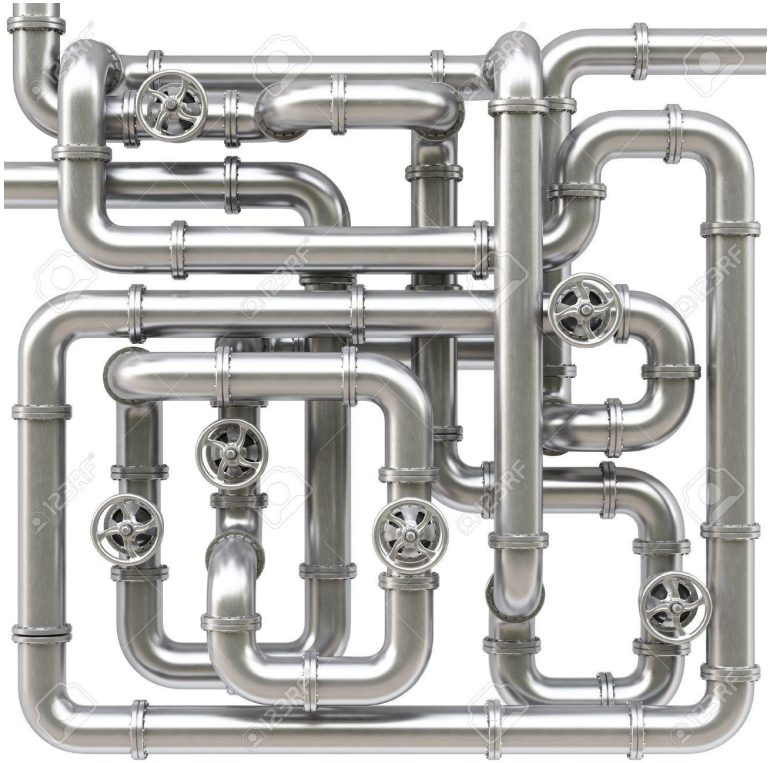
Клиент все равно ждет



Клиент все равно ждет



Канал данных не завершен



Дальнейшие шаги

- Chunked Transfer Encoding для передачи данных по частям

Дальнейшие шаги

- Chunked Transfer Encoding для передачи данных по частям
- Non-blocking NIO и Async Servlet

Выводы

- Базовый набор операторов RxJava прост и удобен для построения цепочек сбора и обработки данных

Выводы

- Базовый набор операторов RxJava прост и удобен для построения цепочек сбора и обработки данных
- Писать свои операторы просто, но надо понимать механику, ошибки могут запомниться вам надолго

Выводы

- Базовый набор операторов RxJava прост и удобен для построения цепочек сбора и обработки данных
- Писать свои операторы просто, но надо понимать механику, ошибки могут запомниться вам надолго
- Rx-цепочки внутри синхронного приложения не являются полноценным решением

Почитать / Посмотреть

- http://www.reactivemanifesto.org/ - прочитать и подписать
- https://github.com/reactive-streams/reactive-streams-jvm/
- http://reactivex.io/ - документация ReactiveX
- https://github.com/ReactiveX/RxJava/wiki - wiki проекта
- http://akarnokd.blogspot.ru/ - блог о внутренностях RxJava
- http://tomstechnicalblog.blogspot.ru/ - понятные объяснения

Вопросы?