Яндекс 360

Осторожно, Combine!

Обо мне

Пять лет занимаюсь
iOS разработкой

Последние три года
работаю в Яндекс Диске

Что такое Combine

Combine

реактивный фреймворк, предоставляющий декларативное API для обработки событий во времени

Combine

был выпущен Apple в 2019 году одновременно со SwiftUI, поэтому оба фреймворка были тесно связаны до 2023 и выхода фреймворка Observation

Предыстория

Вот, что говорит документация

By adopting Combine, you’ll make your code easier to read and maintain, by centralizing your event-processing code and eliminating troublesome techniques like nested closures and convention-based callbacks

О чем поговорим?

1. **Subjects** – что не так с CurrentValueSubject?
   Или нестабильный UI

2. Operators – Почему у вас течет память?

3. Contracts – Датарейсы и креши,
   а еще почему сложно написать экстеншены?

4. API – Почему начать использовать Combine,
   не так просто как другие third-party фреймворки

О чем поговорим?

1. **Subjects** – что не так с CurrentValueSubject?
   Или нестабильный UI

2. **Operators** – Почему у вас течет память?

3. Contracts – Датарейсы и креши,
   а еще почему сложно написать экстеншены?

4. API – Почему начать использовать Combine,
   не так просто как другие third-party фреймворки

О чем поговорим?

1. **Subjects** – что не так с CurrentValueSubject?
   Или нестабильный UI

2. **Operators** – Почему у вас течет память?

3. **Contracts** – Датарейсы и креши,
   а еще почему сложно написать экстеншены?

4. API – Почему начать использовать Combine,
   не так просто как другие third-party фреймворки

О чем поговорим?

1. **Subjects** – что не так с CurrentValueSubject?
   Или нестабильный UI

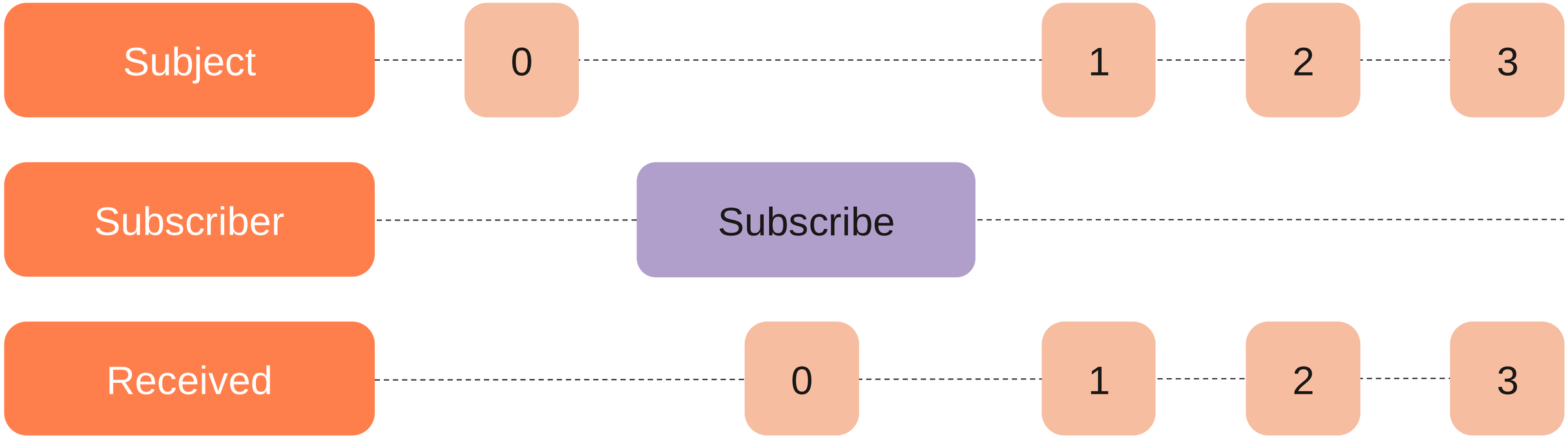
2. **Operators** – Почему у вас течет память?

3. **Contracts** – Датарейсы и креши,
   а еще почему сложно написать экстеншены?

4. **API** – Почему начать использовать Combine,
   не так просто как другие third-party фреймворки

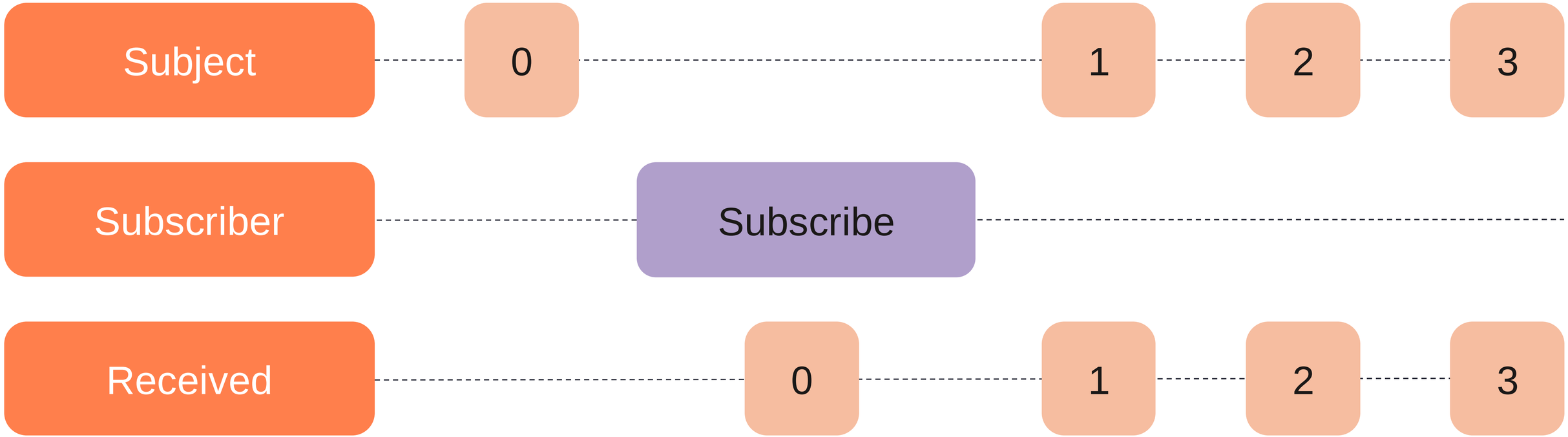
- Subjects
- Operators
- Contracts
- API

CurrentValueSubject

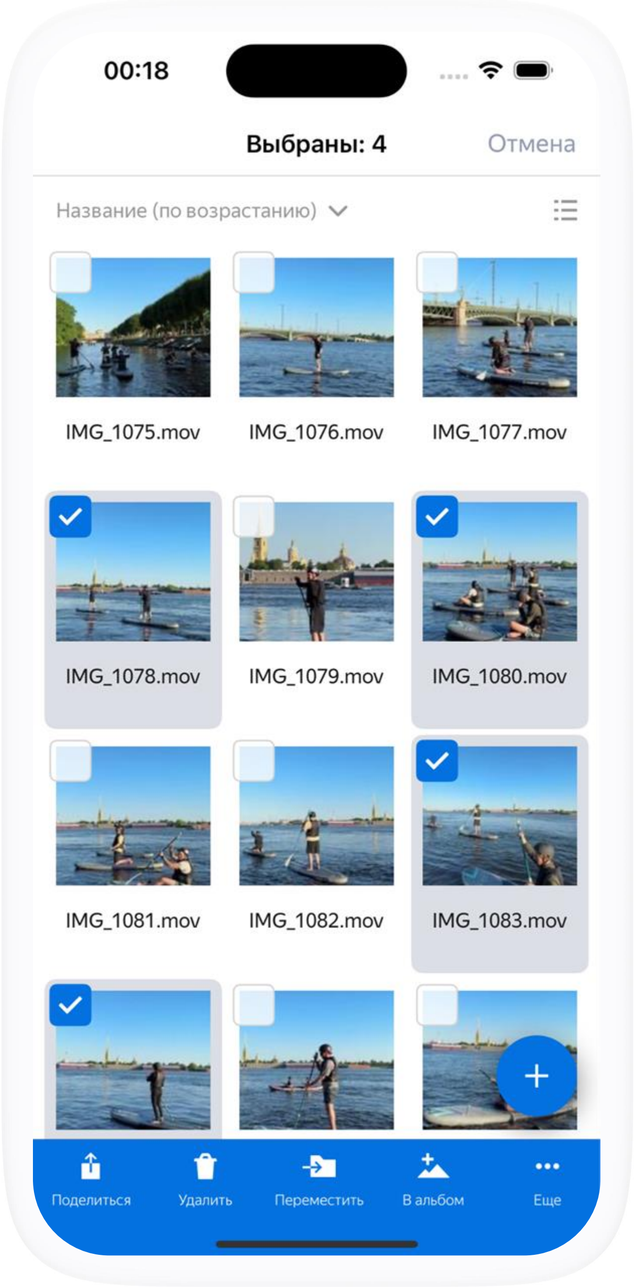


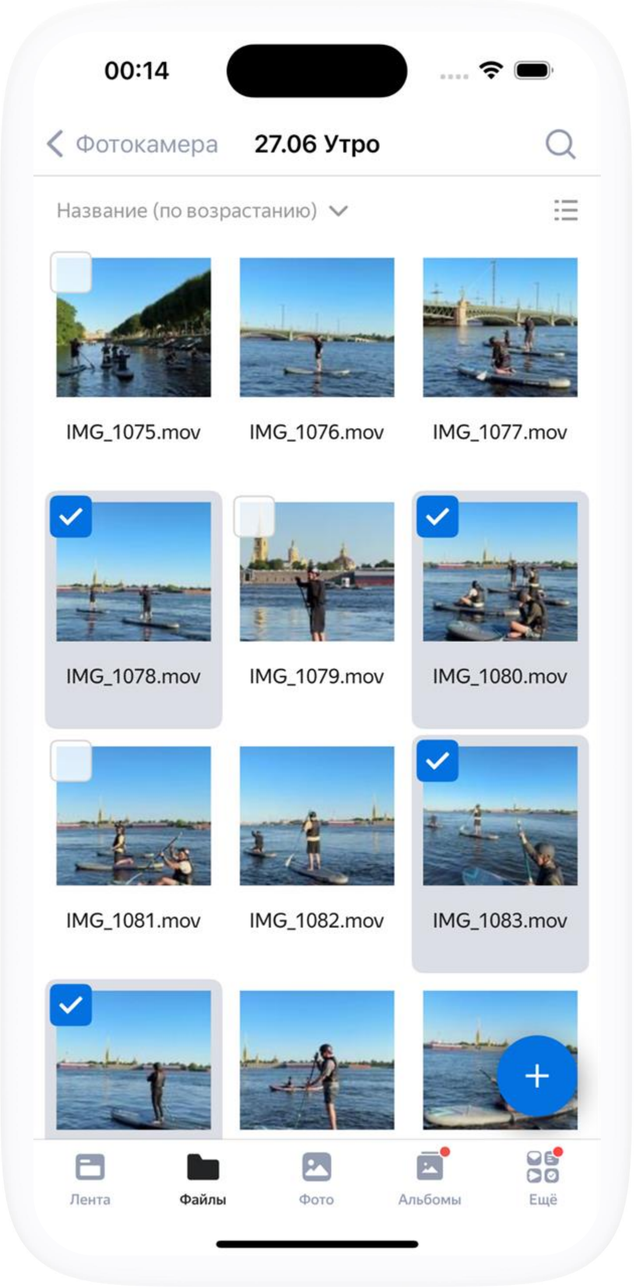
CurrentValueSubject

Текущее значение не потокобезопасно



Столкнулись с этим в файлах





CurrentValueSubject

```
let subject = CurrentValueSubject<Int, Never>(0)
let finishExpectation = expectation(description: "Subject finished")


let expectedResults: Set = [
    [],
    [1],
    [0, 1],
]
var results: [Int] = []
```

```
// Асинхронно отправляем новое значение
DispatchQueue.global().async {
    subject.send(1)
    subject.send(completion: .finished)
}

// Подписываемся на `subject`
subject
    .sink { completion in
        finishExpectation.fulfill()
    } receiveValue: { value in
        results.append(value)
    }

// Ждем получения `.finished`
waitForExpectations(timeout: 2)

XCTAssertTrue(expectedResults.contains(results))
```

CurrentValueSubject

```
let subject = CurrentValueSubject<Int, Never>(0)
let finishExpectation = expectation(description: "Subject finished")


let expectedResults: Set = [
    [],
    [1],
    [0, 1],
]
var results: [Int] = []
```

```
// Асинхронно отправляем новое значение
DispatchQueue.global().async {
    subject.send(1)
    subject.send(completion: .finished)
}

// Подписываемся на `subject`
subject
    .sink { completion in
        finishExpectation.fulfill()
    } receiveValue: { value in
        results.append(value)
    }

// Ждем получения `.finished`
waitForExpectations(timeout: 2)

XCTAssertTrue(expectedResults.contains(results))
```

CurrentValueSubject

```
let subject = CurrentValueSubject<Int, Never>(0)
let finishExpectation = expectation(description: "Subject finished")


let expectedResults: Set = [
    [],
    [1],
    [0, 1],
]
var results: [Int] = []
```

```
// Асинхронно отправляем новое значение
DispatchQueue.global().async {
    subject.send(1)
    subject.send(completion: .finished)
}

// Подписываемся на `subject`
subject
    .sink { completion in
        finishExpectation.fulfill()
    } receiveValue: { value in
        results.append(value)
    }

// Ждем получения `.finished`
waitForExpectations(timeout: 2)

XCTAssertTrue(expectedResults.contains(results))
```

CurrentValueSubject

```
let subject = CurrentValueSubject<Int, Never>(0)
let finishExpectation = expectation(description: "Subject finished")


let expectedResults: Set = [
    [],
    [1],
    [0, 1],
]
var results: [Int] = []
```

```
// Асинхронно отправляем новое значение
DispatchQueue.global().async {
    subject.send(1)
    subject.send(completion: .finished)
}

// Подписываемся на `subject`
subject
    .sink { completion in
        finishExpectation.fulfill()
    } receiveValue: { value in
        results.append(value)
    }

// Ждем получения `.finished`
waitForExpectations(timeout: 2)

XCTAssertTrue(expectedResults.contains(results))
```

CurrentValueSubject

```
let subject = CurrentValueSubject<Int, Never>(0)
let finishExpectation = expectation(description: "Subject finished")


let expectedResults: Set = [
    [],
    [1],
    [0, 1],
]
var results: [Int] = []
```

```
// Асинхронно отправляем новое значение
DispatchQueue.global().async {
    subject.send(1)
    subject.send(completion: .finished)
}

// Подписываемся на `subject`
subject
    .sink { completion in
        finishExpectation.fulfill()
    } receiveValue: { value in
        results.append(value)
    }

// Ждем получения `.finished`
waitForExpectations(timeout: 2)

XCTAssertTrue(expectedResults.contains(results))
```

CurrentValueSubject

```
let subject = CurrentValueSubject<Int, Never>(0)
let finishExpectation = expectation(description: "Subject finished")


let expectedResults: Set = [
    [],
    [1],
    [0, 1],
]
var results: [Int] = []
```

```
// Асинхронно отправляем новое значение
DispatchQueue.global().async {
    subject.send(1)
    subject.send(completion: .finished)
}

// Подписываемся на `subject`
subject
    .sink { completion in
        finishExpectation.fulfill()
    } receiveValue: { value in
        results.append(value)
    }

// Ждем получения `.finished`
waitForExpectations(timeout: 2)

XCTAssertTrue(expectedResults.contains(results))
```

CurrentValueSubject

```
let subject = CurrentValueSubject<Int, Never>(0)
let finishExpectation = expectation(description: "Subject finished")


let expectedResults: Set = [
    [],
    [1],
    [0, 1],
]
var results: [Int] = []
```

```
// Асинхронно отправляем новое значение
DispatchQueue.global().async {
    subject.send(1)
    subject.send(completion: .finished)
}

// Подписываемся на `subject`
subject
    .sink { completion in
        finishExpectation.fulfill()
    } receiveValue: { value in
        results.append(value)
    }

// Ждем получения `.finished`
waitForExpectations(timeout: 2)

XCTAssertTrue(expectedResults.contains(results))
```

CurrentValueSubject

```
let subject = CurrentValueSubject<Int, Never>(0)
let finishExpectation = expectation(description: "Subject finished")


let expectedResults: Set = [
    [],
    [1],
    [0, 1],
]
var results: [Int] = []
```

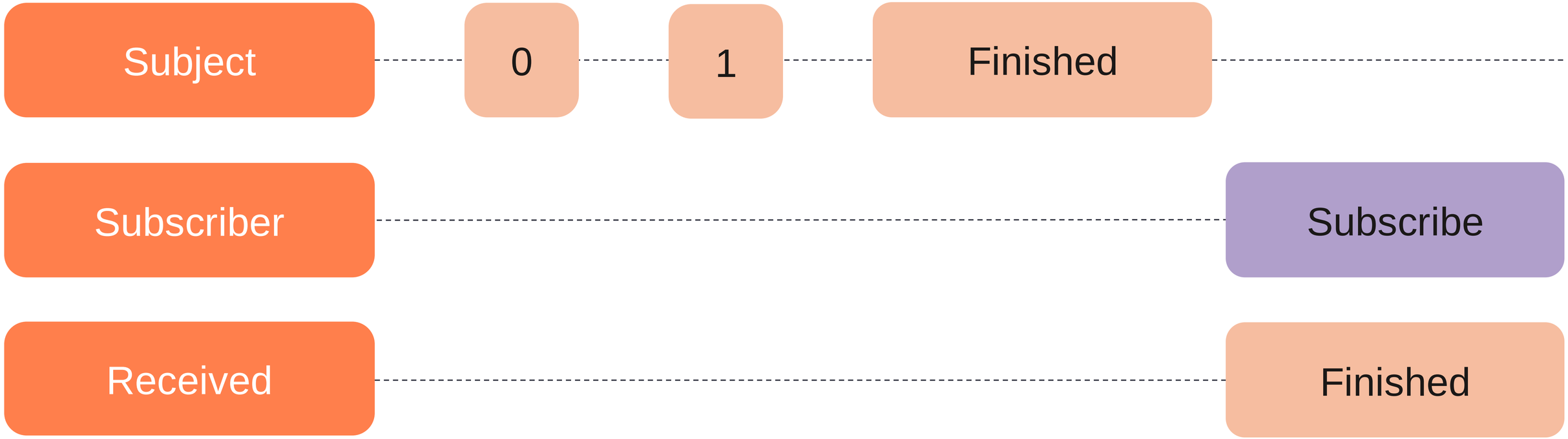
```
// Асинхронно отправляем новое значение
DispatchQueue.global().async {
    subject.send(1)
    subject.send(completion: .finished)
}

// Подписываемся на `subject`
subject
    .sink { completion in
        finishExpectation.fulfill()
    } receiveValue: { value in
        results.append(value)
    }

// Ждем получения `.finished`
waitForExpectations(timeout: 2)

XCTAssertTrue(expectedResults.contains(results))
```

CurrentValueSubject

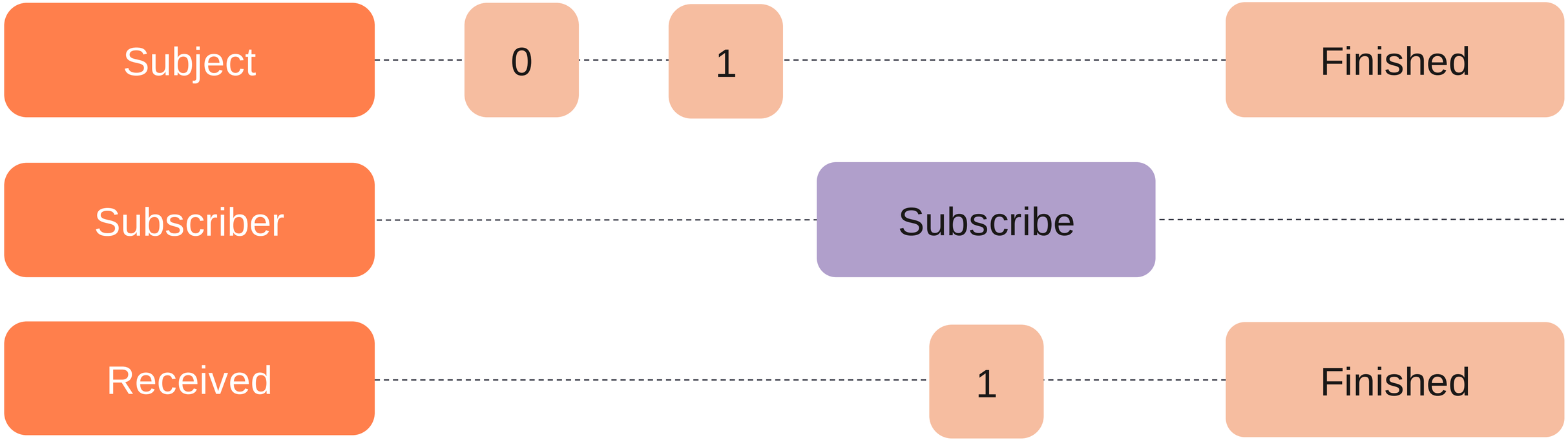
```
let subject = CurrentValueSubject<Int, Never>(0)
let finishExpectation = expectation(description: "Subject finished")


let expectedResults: Set = [
    [],
    [1],
    [0, 1],
]
var results: [Int] = []
```

```
// Асинхронно отправляем новое значение
DispatchQueue.global().async {
    subject.send(1)
    subject.send(completion: .finished)
}

// Подписываемся на `subject`
subject
    .sink { completion in
        finishExpectation.fulfill()
    } receiveValue: { value in
        results.append(value)
    }

// Ждем получения `.finished`
waitForExpectations(timeout: 2)

XCTAssertTrue(expectedResults.contains(results))
```

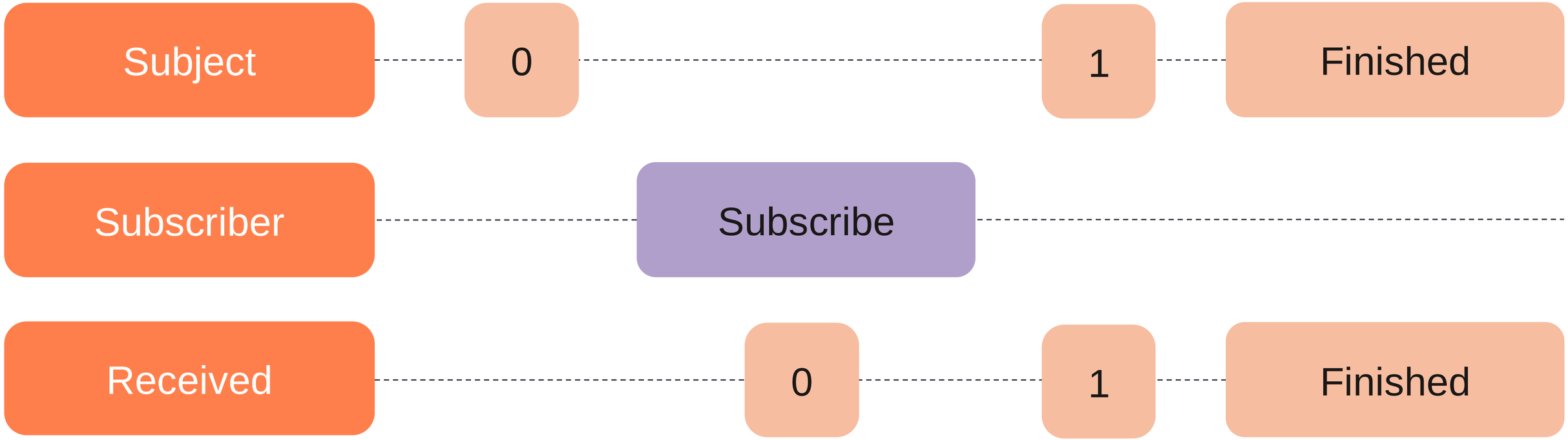
expectedResults = []



expectedResults = [1]



expectedResults = [0, 1]



CurrentValueSubject

```
let subject = CurrentValueSubject<Int, Never>(0)
let finishExpectation = expectation(description: "Subject finished")


let expectedResults: Set = [
    [],
    [1],
    [0, 1],
    [1, 0],
    [1, 1],
]
var results: [Int] = []
```

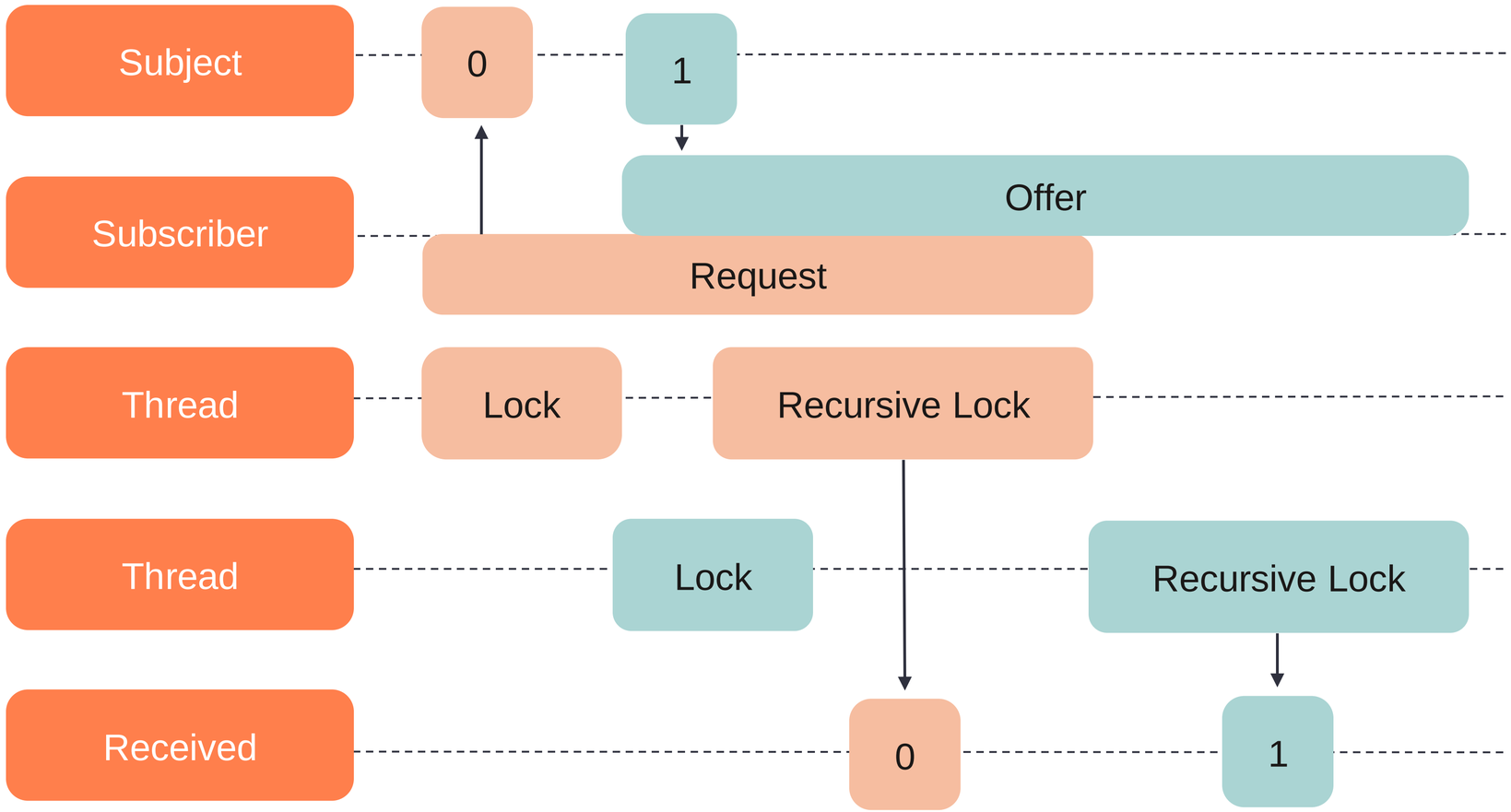
```
// Асинхронно отправляем новое значение
DispatchQueue.global().async {
    subject.send(1)
    subject.send(completion: .finished)
}

// Подписываемся на `subject`
subject
    .sink { completion in
        finishExpectation.fulfill()
    } receiveValue: { value in
        results.append(value)
    }

// Ждем получения `.finished`
waitForExpectations(timeout: 2)

XCTAssertTrue(expectedResults.contains(results))
```

CurrentValueSubject

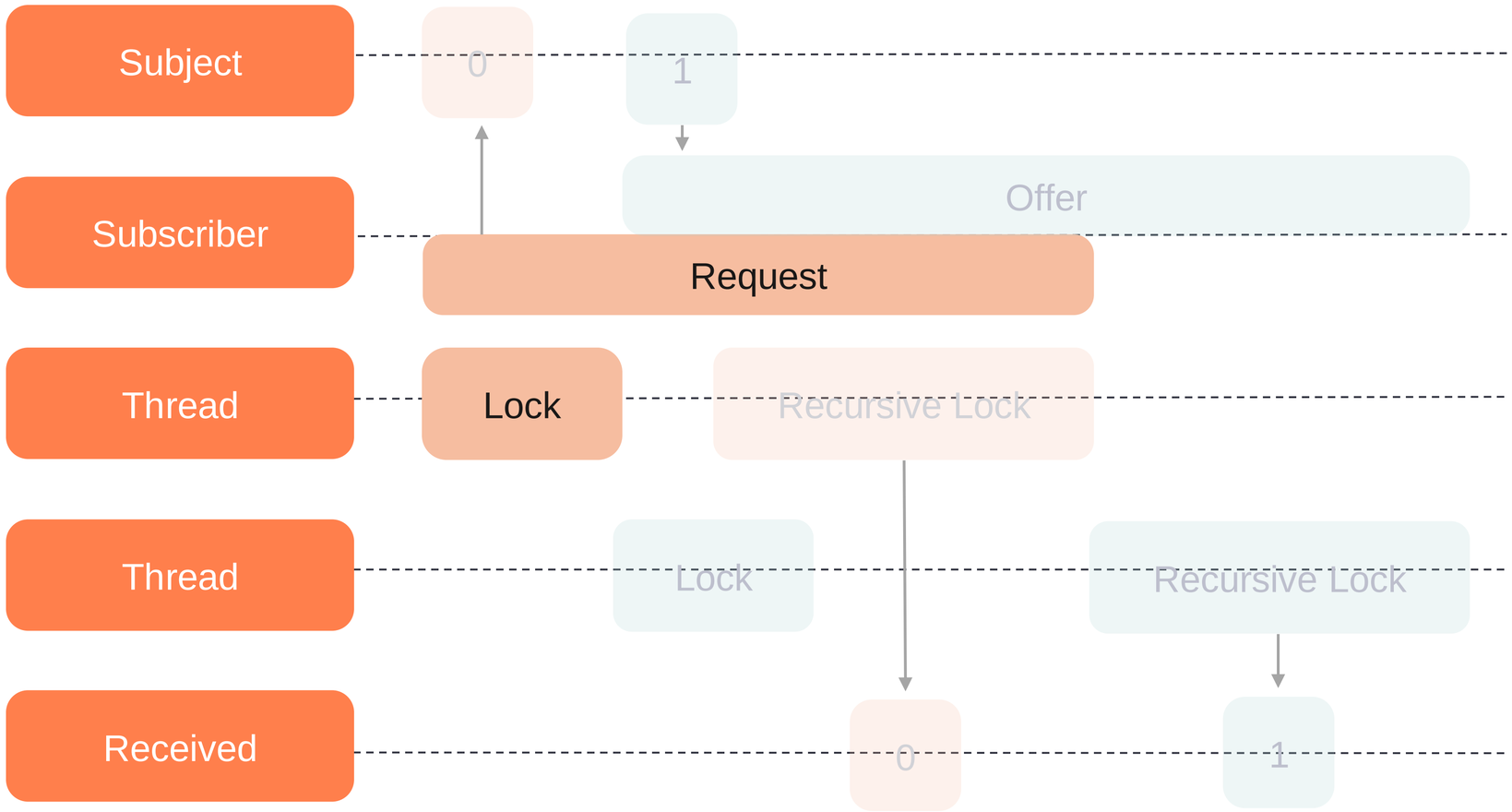
```
let subject = CurrentValueSubject<Int, Never>(0)
let finishExpectation = expectation(description: "Subject finished")


let expectedResults: Set = [
    [],
    [1],
    [0, 1],
    [1, 0],
    [1, 1],
]
var results: [Int] = []
```

```
// Асинхронно отправляем новое значение
DispatchQueue.global().async {
    subject.send(1)
    subject.send(completion: .finished)
}

// Подписываемся на `subject`
subject
    .sink { completion in
        finishExpectation.fulfill()
    } receiveValue: { value in
        results.append(value)
    }

// Ждем получения `.finished`
waitForExpectations(timeout: 2)

XCTAssertTrue(expectedResults.contains(results))
```

Что происходит

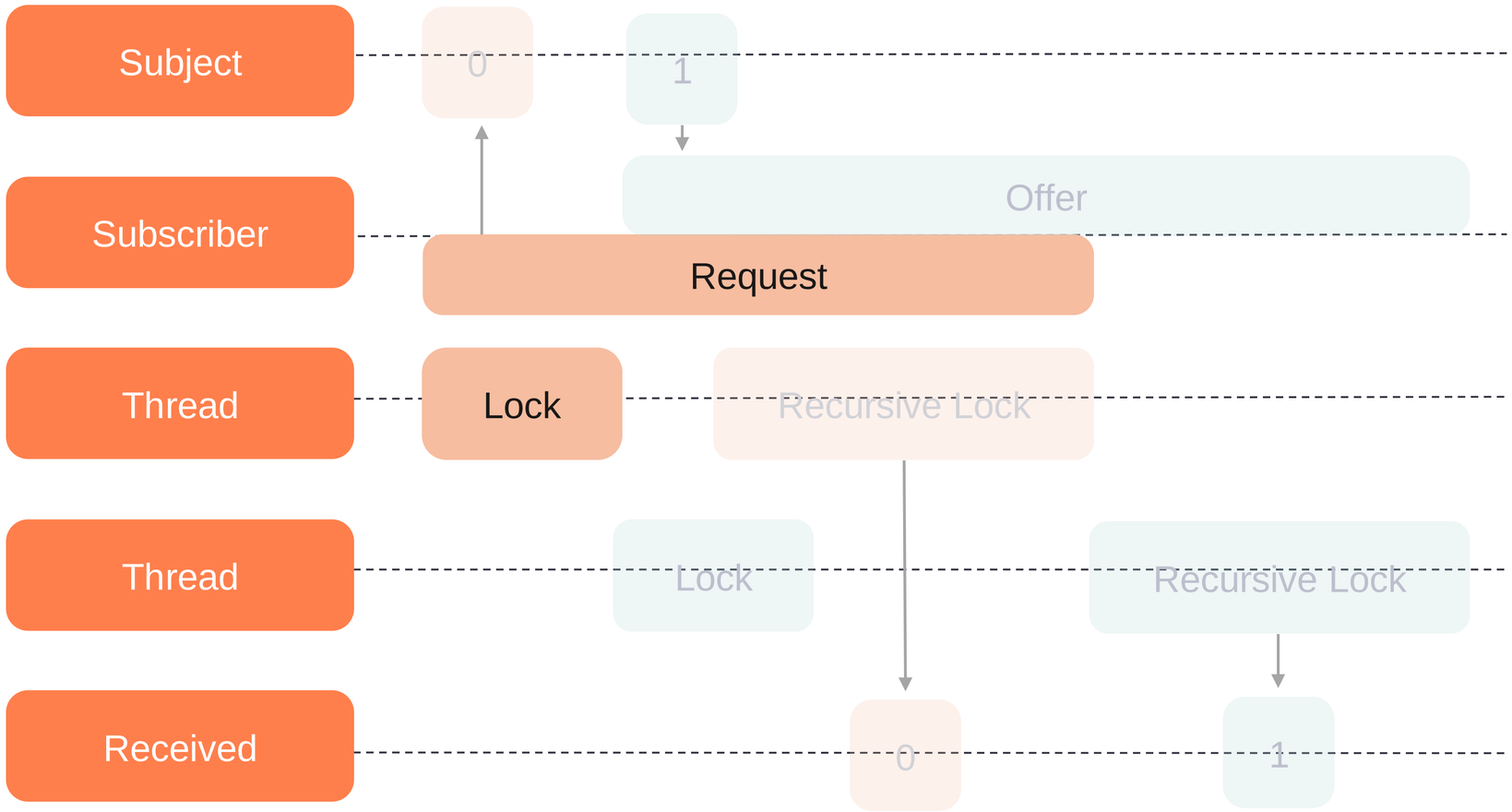


Что происходит



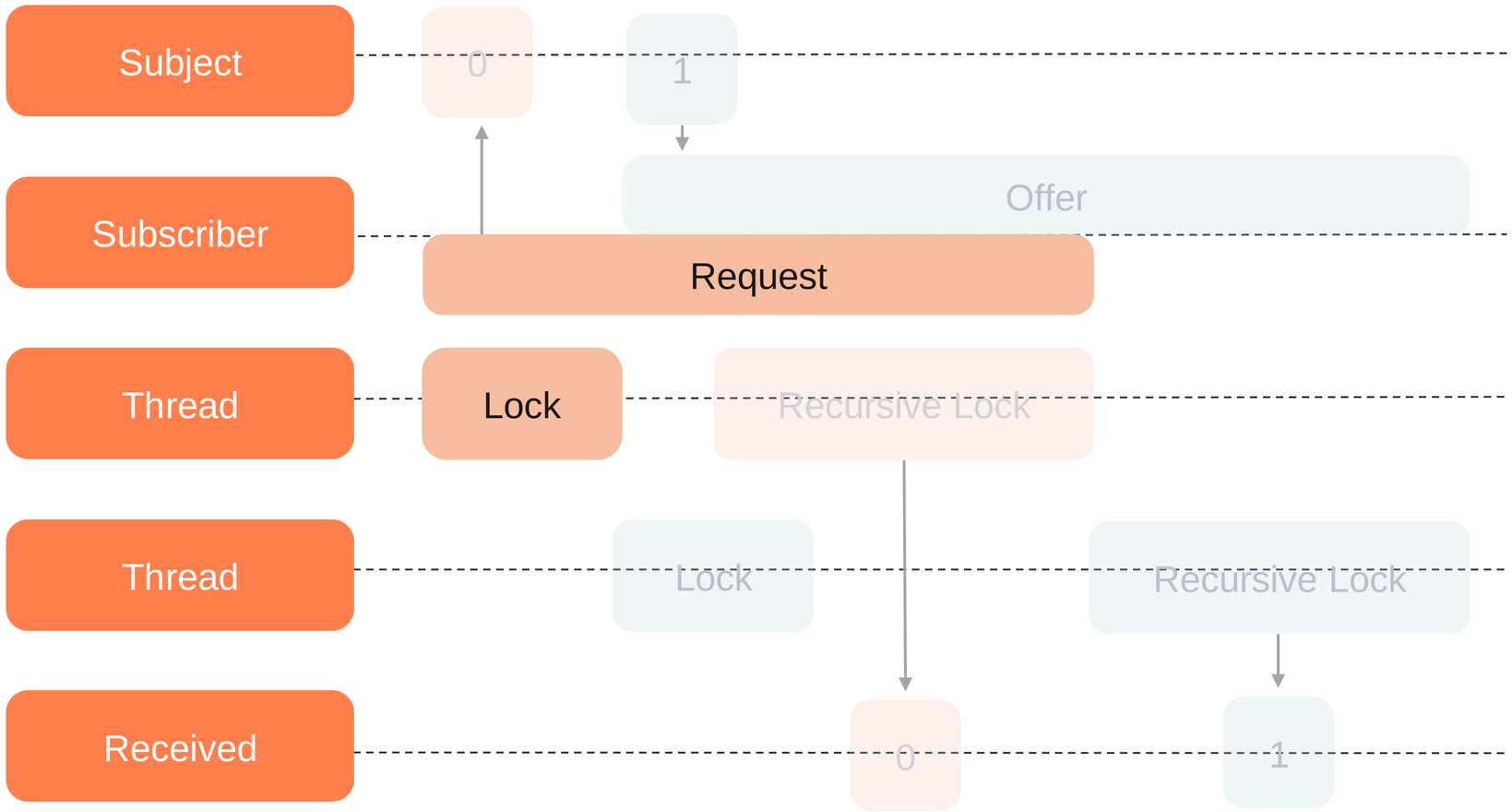
1. Захватить Lock
2. Проверить сколько значений нужно отправить подписчику
3. [request(_:)] Взять текущее значение из CurrentValueSubject
4. Отпустить Lock
5. Захватить RecursiveLock
6. Отправить значение подписчику
7. Отпустить RecursiveLock

Что происходит



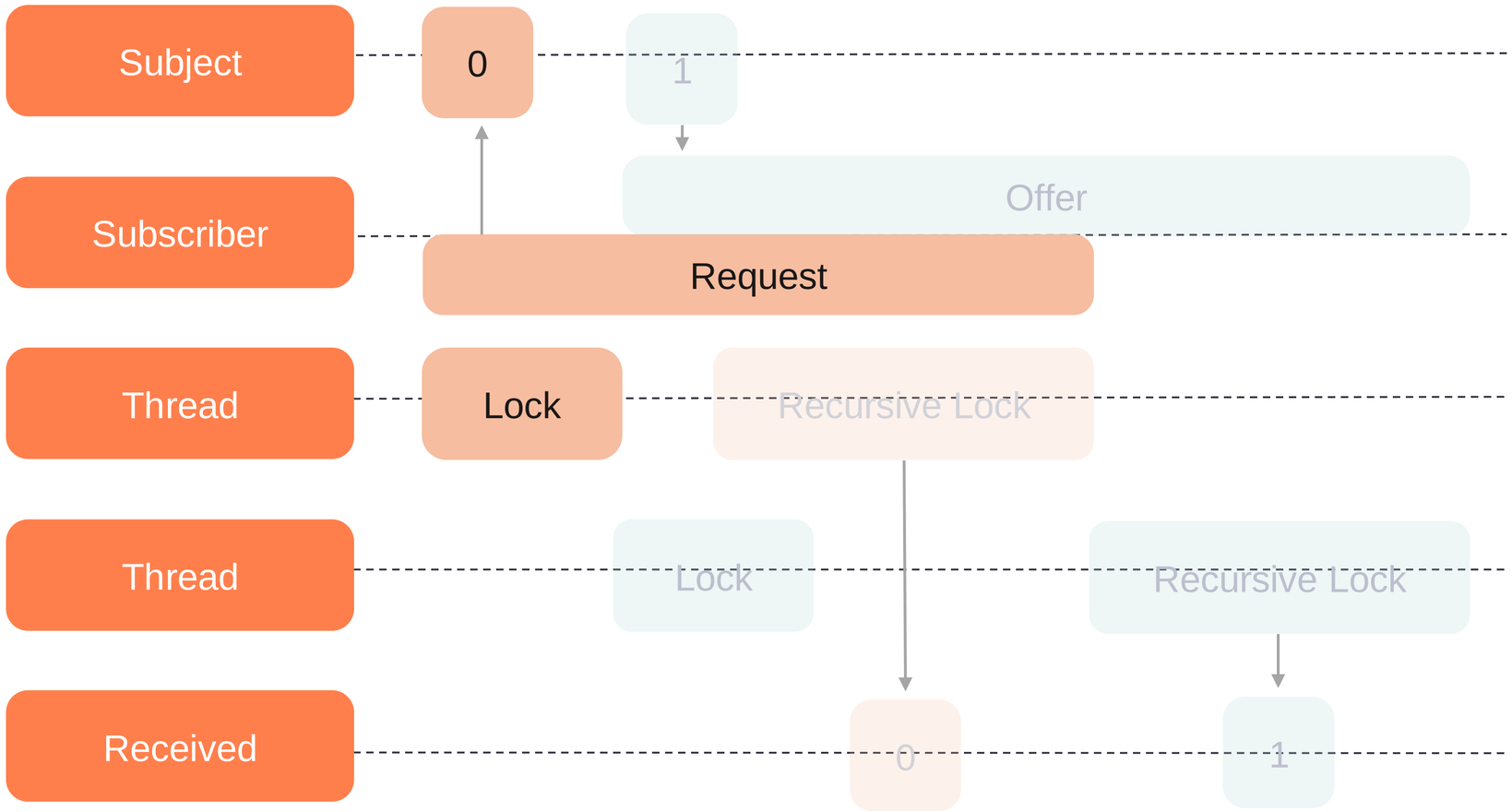
1. Захватить Lock
2. Проверить сколько значений нужно отправить подписчику
3. [request(_:)] Взять текущее значение из CurrentValueSubject
4. Отпустить Lock
5. Захватить RecursiveLock
6. Отправить значение подписчику
7. Отпустить RecursiveLock

Что происходит



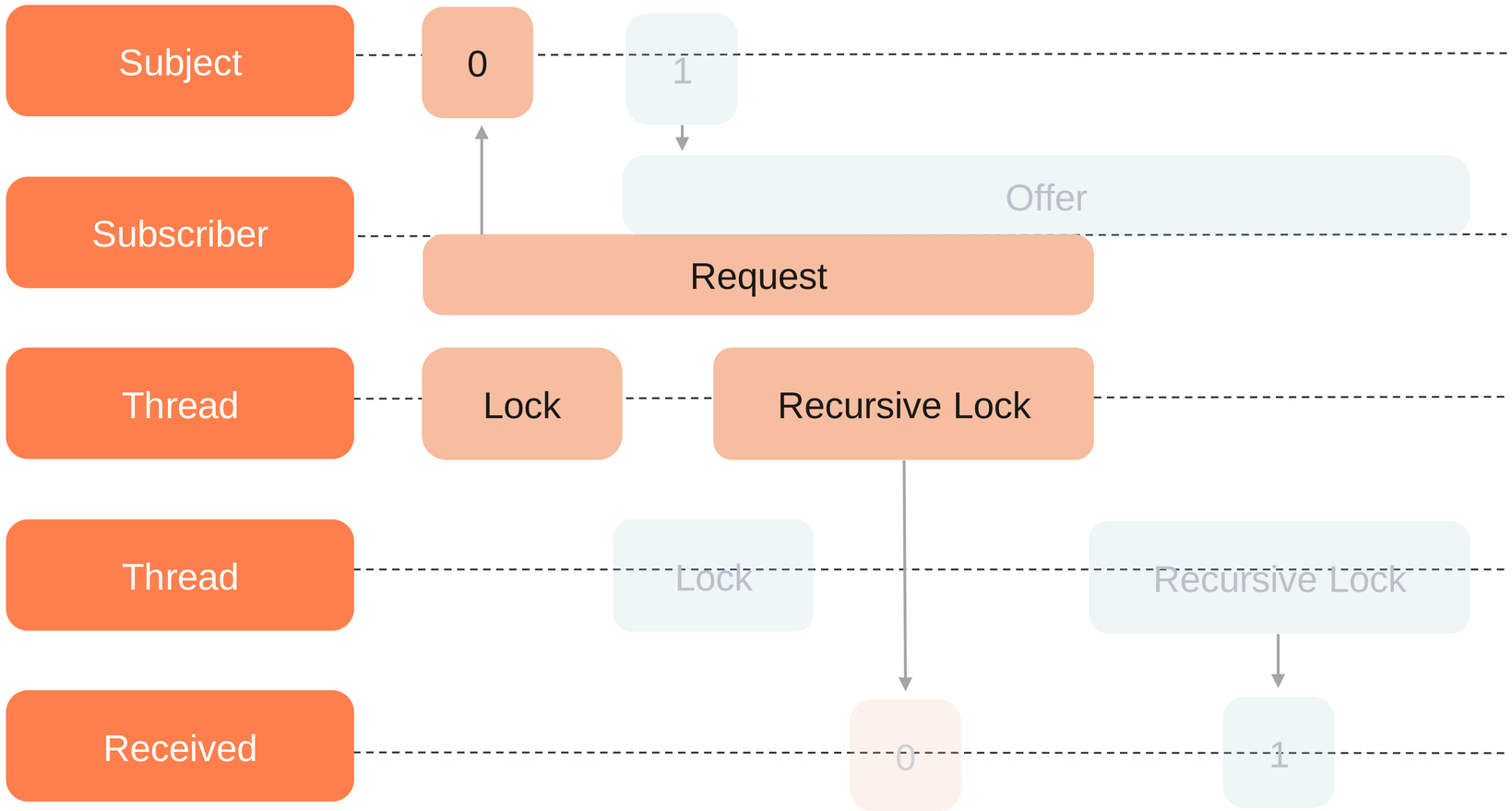
1. Захватить Lock
2. Проверить сколько значений нужно отправить подписчику
3. [request(_:)] Взять текущее значение из CurrentValueSubject
4. Отпустить Lock
5. Захватить RecursiveLock
6. Отправить значение подписчику
7. Отпустить RecursiveLock

Что происходит



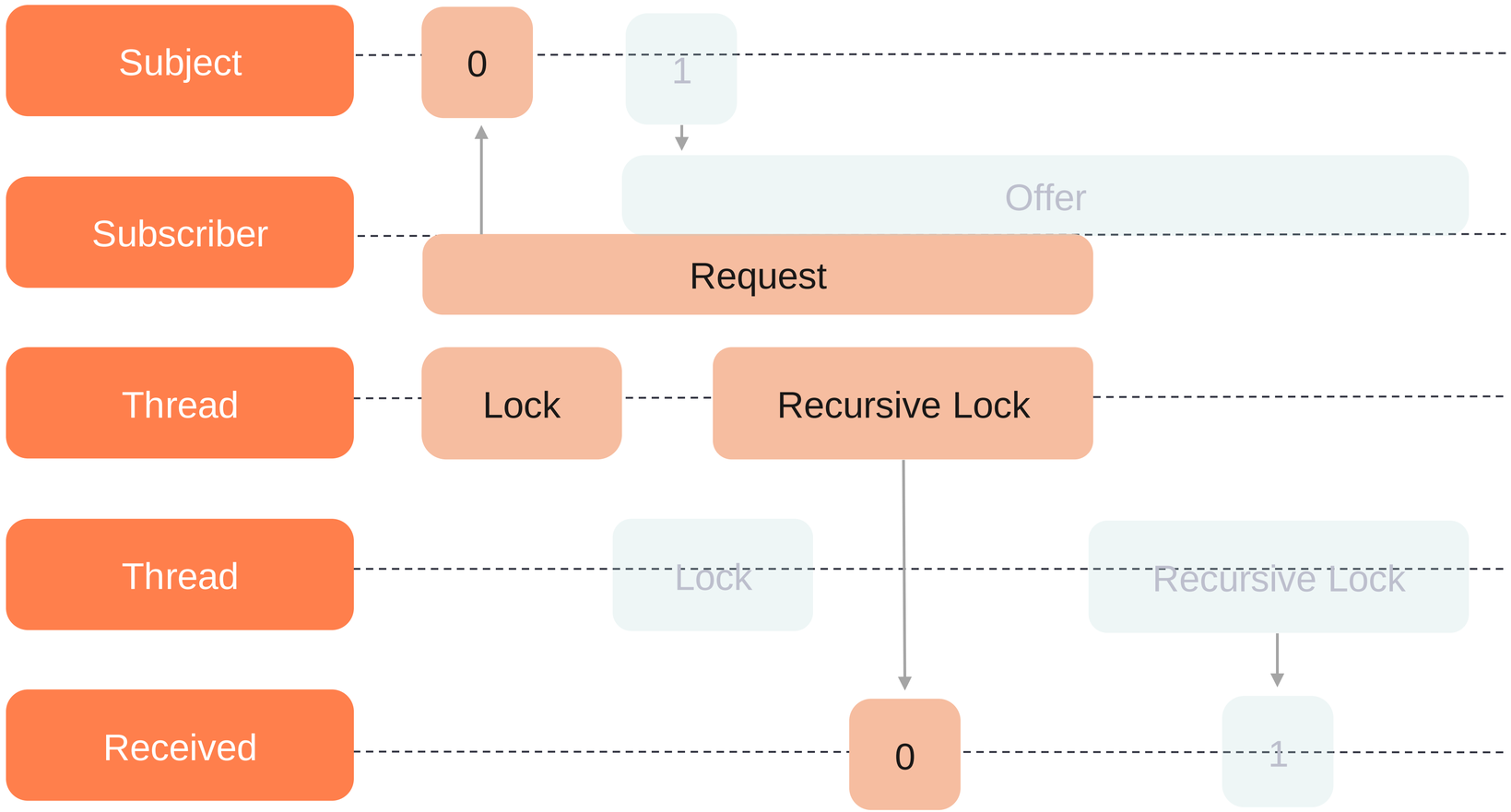
1. Захватить Lock
2. Проверить сколько значений нужно отправить подписчику
3. [request(_:)] Взять текущее значение из CurrentValueSubject
4. Отпустить Lock
5. Захватить RecursiveLock
6. Отправить значение подписчику
7. Отпустить RecursiveLock

Что происходит



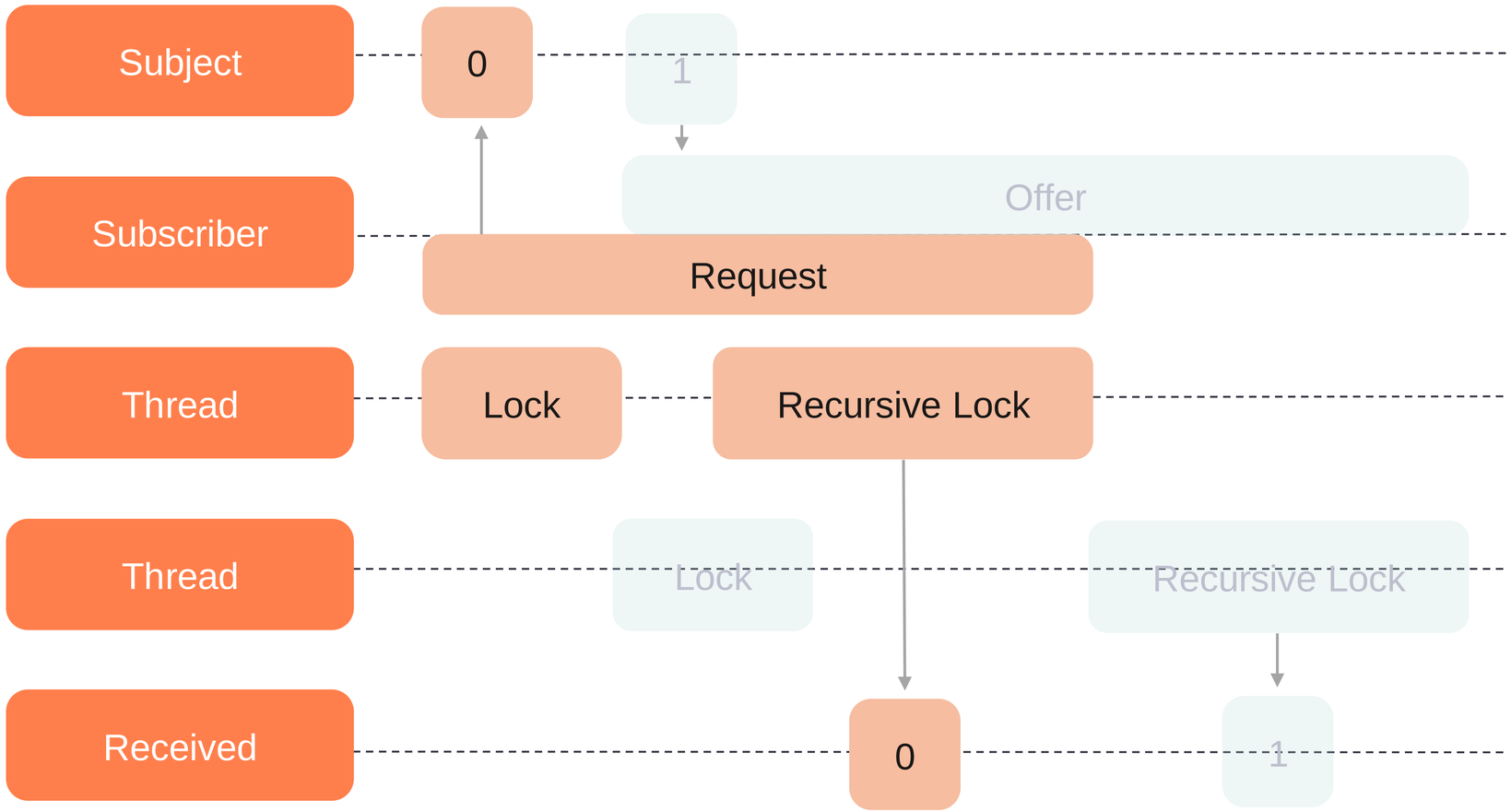
1. Захватить Lock
2. Проверить сколько значений нужно отправить подписчику
3. [request(_:)] Взять текущее значение из CurrentValueSubject
4. Отпустить Lock
5. Захватить RecursiveLock
6. Отправить значение подписчику
7. Отпустить RecursiveLock

Что происходит



1. Захватить Lock
2. Проверить сколько значений нужно отправить подписчику
3. [request(_:)] Взять текущее значение из CurrentValueSubject
4. Отпустить Lock
5. Захватить RecursiveLock
6. Отправить значение подписчику
7. Отпустить RecursiveLock

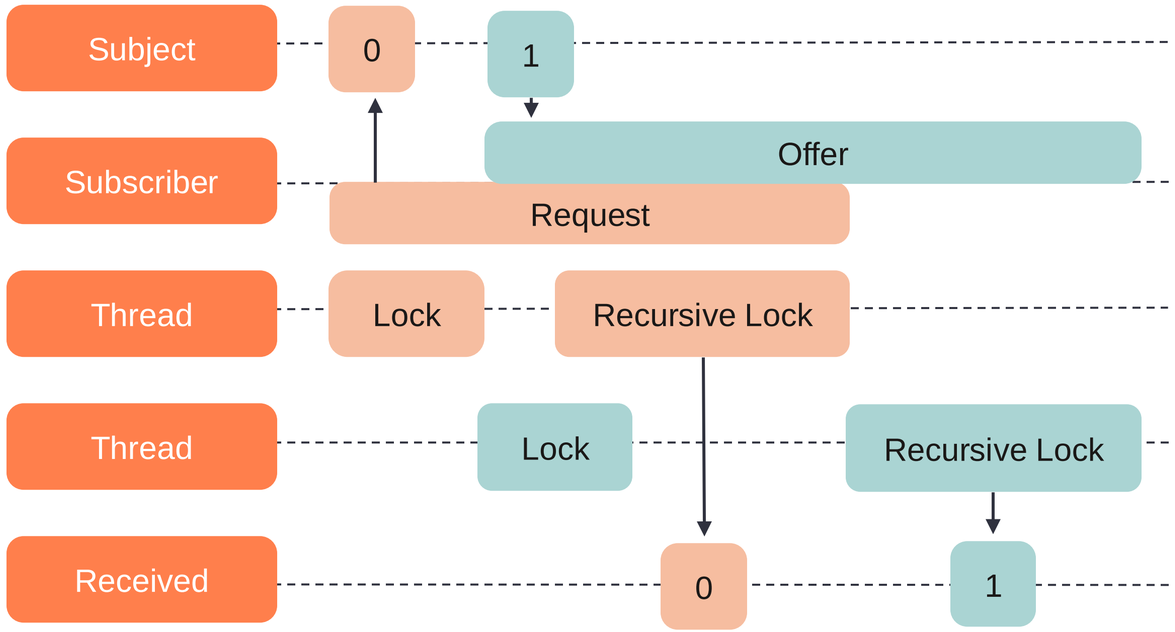
Что происходит



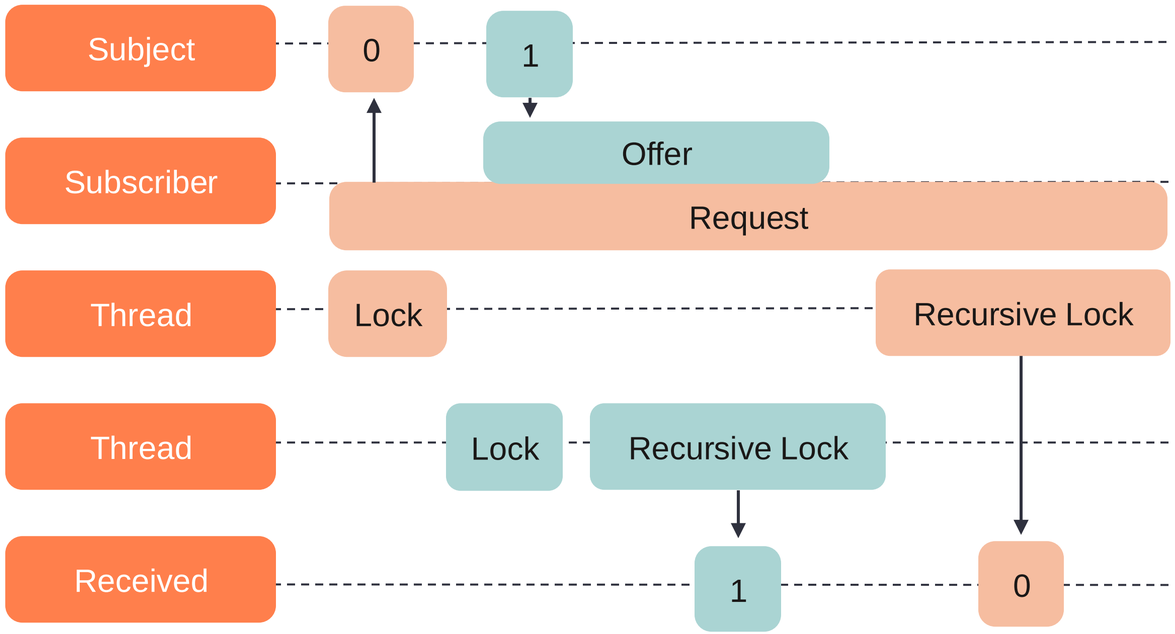
1. Захватить Lock
2. Проверить сколько значений нужно отправить подписчику
3. [request(_:)] Взять текущее значение из CurrentValueSubject
4. Отпустить Lock
5. Захватить RecursiveLock
6. Отправить значение подписчику
7. Отпустить RecursiveLock

Как получилось [1, 0]

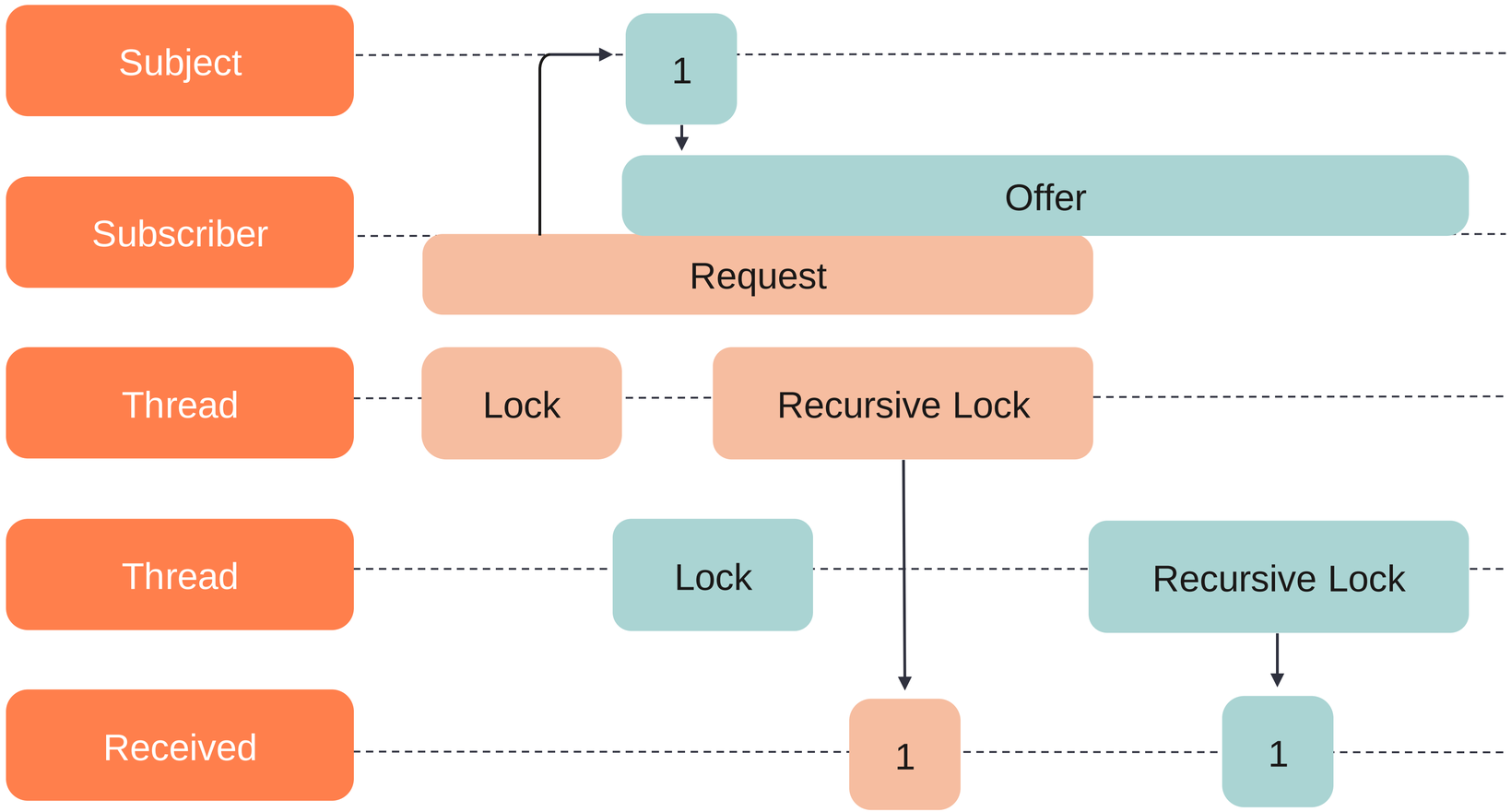
Как должно быть:



Как получилось:



Как получилось [1, 1]



1. Захватить Lock
2. Проверить сколько значений нужно отправить подписчику
3. [request(_:)] Взять текущее значение из CurrentValueSubject
4. Отпустить Lock
5. Захватить RecursiveLock
6. Отправить значение подписчику
7. Отпустить RecursiveLock

Выводы

Всегда слать события и подписываться
на CurrentValueSubject только с одного потока

- Subjects
- Operators
- Contracts
- API

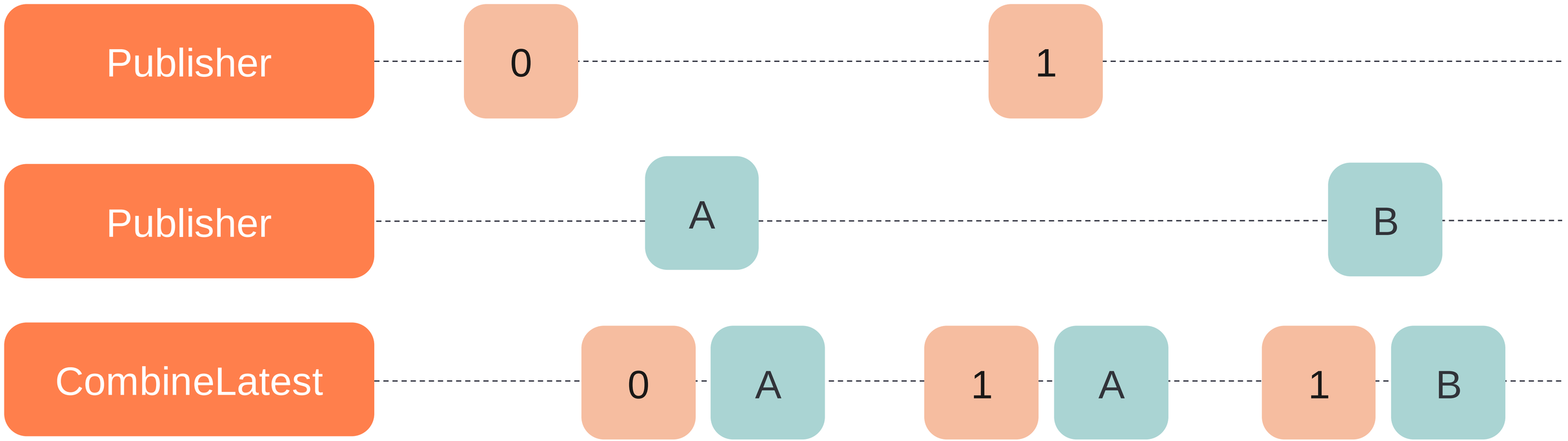
Operators

- CombineLatest
- SubscribeOn
- Multicast

Operators

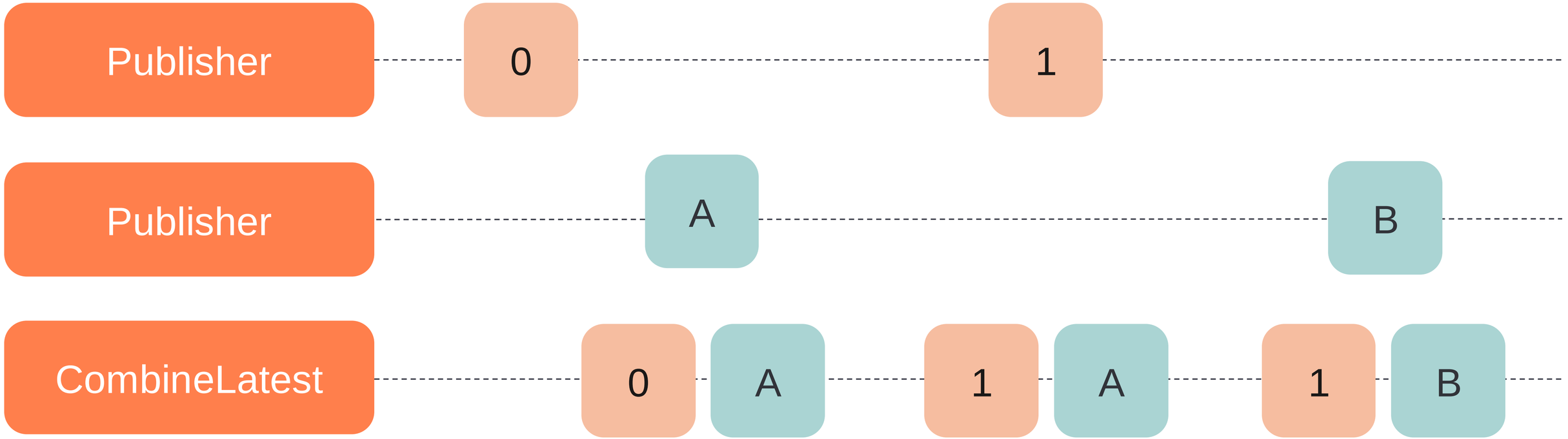
- CombineLatest
- SubscribeOn
- Multicast

CombineLatest



CombineLatest

Потеря эвентов



CombineLatest

```
let finishExpectation = expectation(description: "Subject finished")
let queue = DispatchQueue(label: "testCombineLatest")

let publisher0 = [0].publisher
let publisher1 = [1].publisher

var results: [Int] = []
```

```
publisher0
    .subscribe(on: queue)
    .combineLatest(publisher1)
    .sink { value in
        results.append(value.0)
        results.append(value.1)

        finishExpectation.fulfill()
    }

waitForExpectations(timeout: 1)

XCTAssertEqual([0, 1], results)
```

CombineLatest

```
let finishExpectation = expectation(description: "Subject finished")
let queue = DispatchQueue(label: "testCombineLatest")

let publisher0 = [0].publisher
let publisher1 = [1].publisher

var results: [Int] = []
```

```
publisher0
    .subscribe(on: queue)
    .combineLatest(publisher1)
    .sink { value in
        results.append(value.0)
        results.append(value.1)

        finishExpectation.fulfill()
    }

waitForExpectations(timeout: 1)

XCTAssertEqual([0, 1], results)
```

CombineLatest

```
let finishExpectation = expectation(description: "Subject finished")
let queue = DispatchQueue(label: "testCombineLatest")

let publisher0 = [0].publisher
let publisher1 = [1].publisher

var results: [Int] = []
```

```
publisher0
    .subscribe(on: queue)
    .combineLatest(publisher1)
    .sink { value in
        results.append(value.0)
        results.append(value.1)

        finishExpectation.fulfill()
    }

waitForExpectations(timeout: 1)

XCTAssertEqual([0, 1], results)
```

CombineLatest

```
let finishExpectation = expectation(description: "Subject finished")
let queue = DispatchQueue(label: "testCombineLatest")

let publisher0 = [0].publisher
let publisher1 = [1].publisher

var results: [Int] = []
```

```
publisher0
    .subscribe(on: queue)
    .combineLatest(publisher1)
    .sink { value in
        results.append(value.0)
        results.append(value.1)

        finishExpectation.fulfill()
    }

waitForExpectations(timeout: 1)

XCTAssertEqual([0, 1], results)
```

CombineLatest

```
let finishExpectation = expectation(description: "Subject finished")
let queue = DispatchQueue(label: "testCombineLatest")

let publisher0 = [0].publisher
let publisher1 = [1].publisher

var results: [Int] = []
```

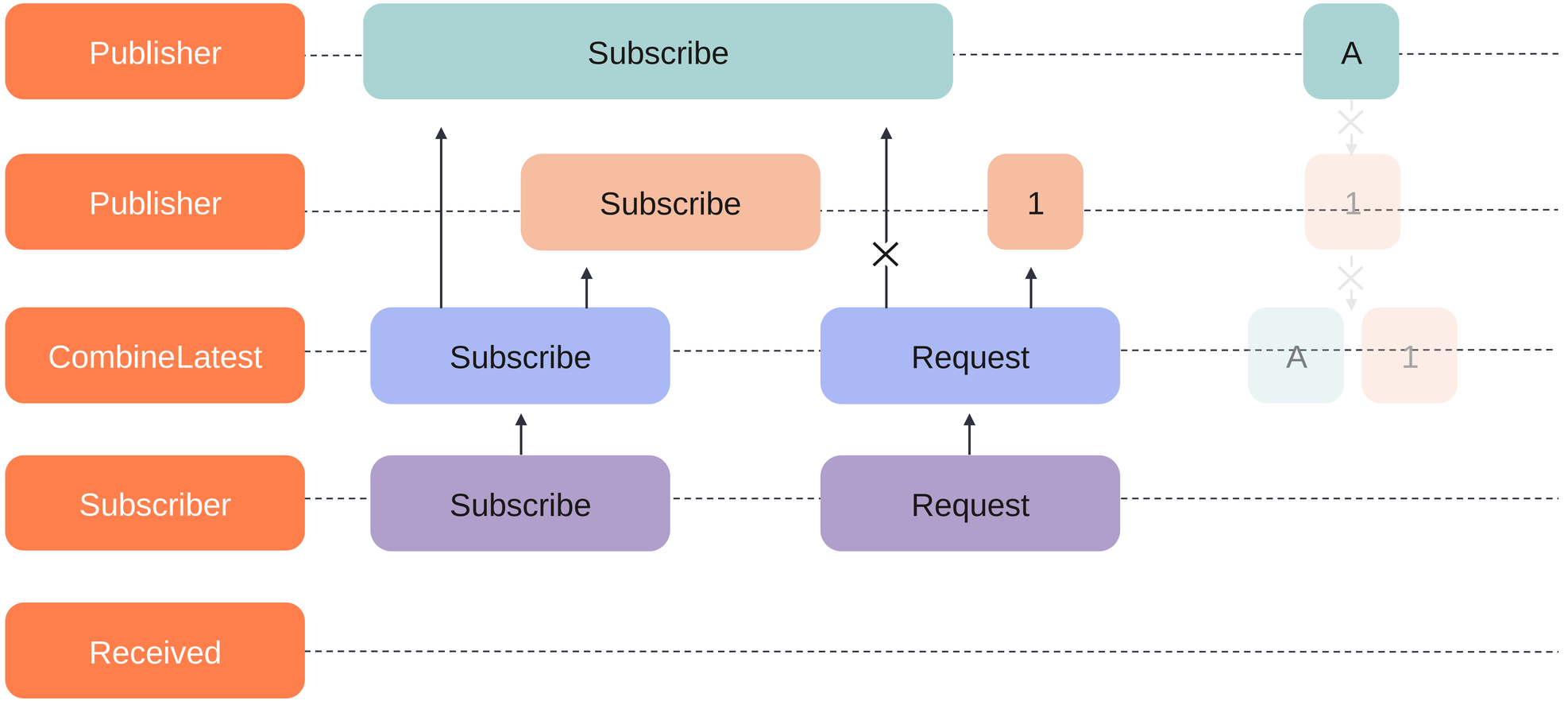
```
publisher0
    .subscribe(on: queue)
    .combineLatest(publisher1)
    .sink { value in
        results.append(value.0)
        results.append(value.1)

        finishExpectation.fulfill()
    }

waitForExpectations(timeout: 1)

XCTAssertEqual([0, 1], results)
```

CombineLatest

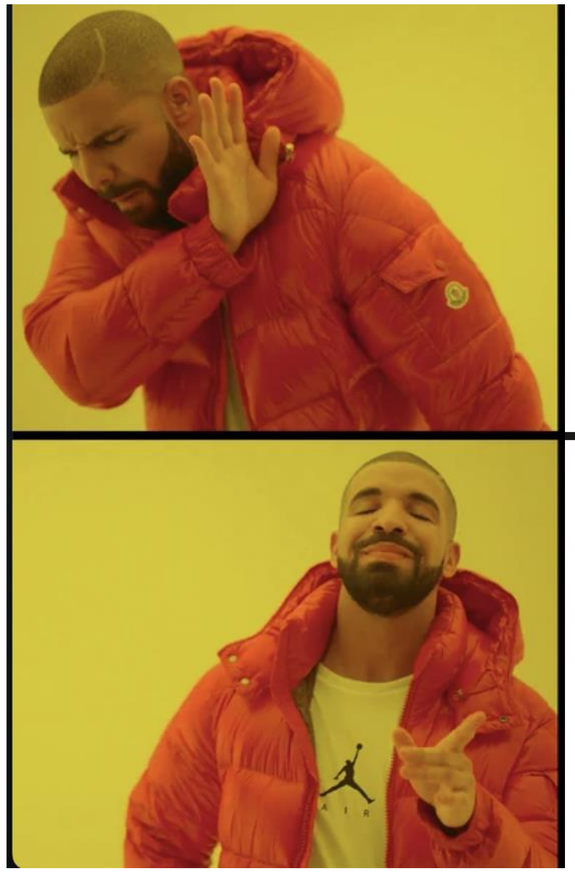
```
let finishExpectation = expectation(description: "Subject finished")
let queue = DispatchQueue(label: "testCombineLatest")

let publisher0 = [0].publisher
let publisher1 = [1].publisher

var results: [Int] = []
```

```
publisher0
    .subscribe(on: queue)
    .combineLatest(publisher1)
    .sink { value in
        results.append(value.0)
        results.append(value.1)

        finishExpectation.fulfill()
    }

waitForExpectations(timeout: 1) // Asynchronous wait failed

XCTAssertEqual([0, 1], results) // []
```

CombineLatest



Что с этим можно сделать?



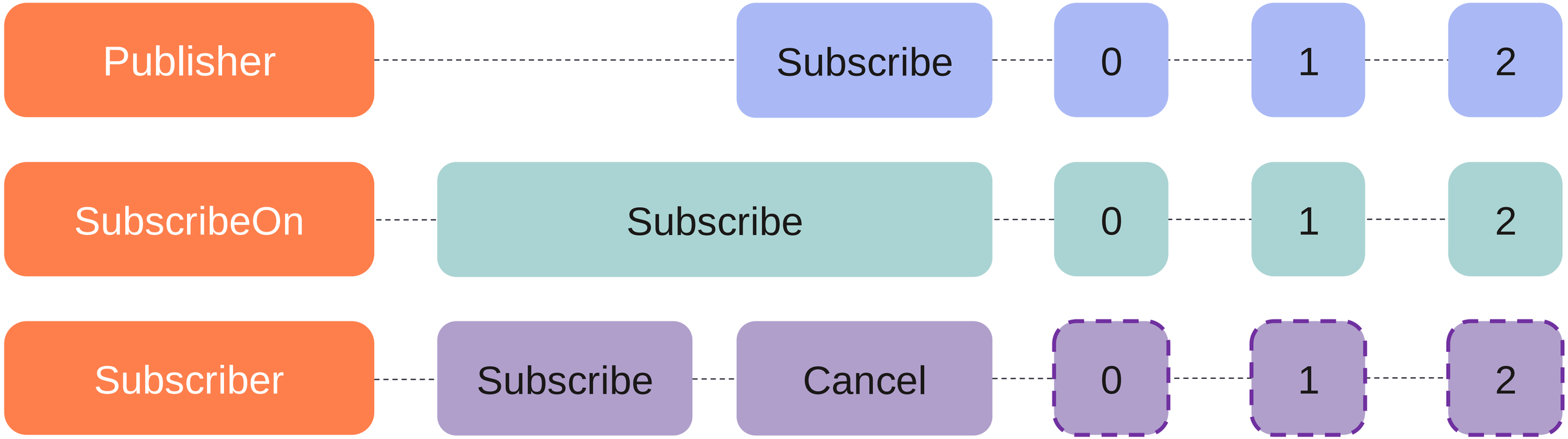
Написать свою реализацию
CombineLatest

Исправить баги в работе
subscribe(on:options:)

Operators

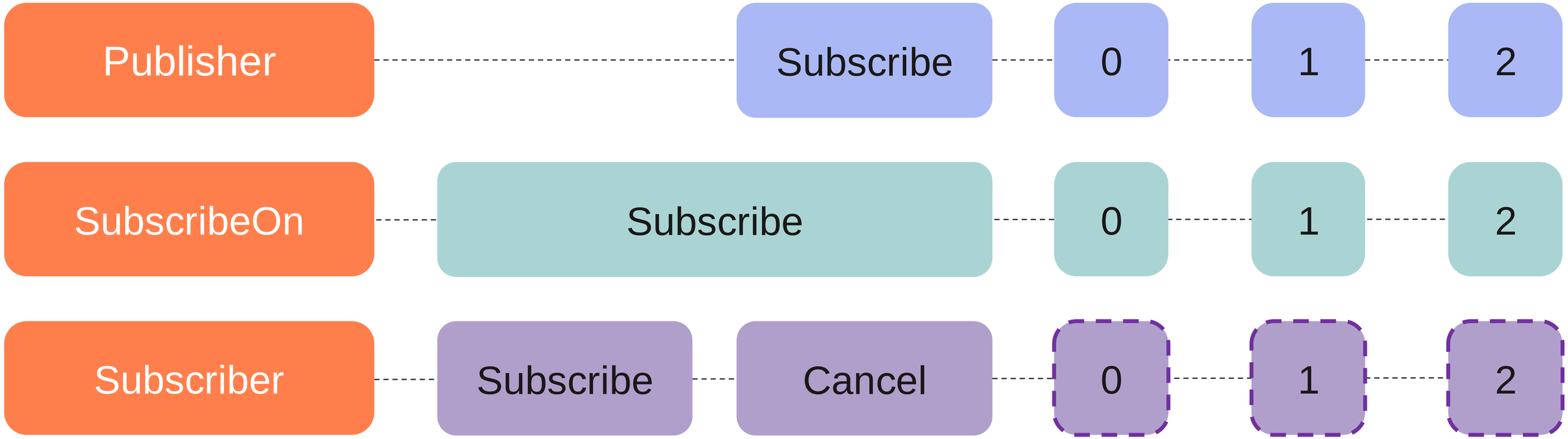
- CombineLatest
- SubscribeOn
- Multicast

SubscribeOn



SubscribeOn

Утечки памяти



SubscribeOn

```
let subscribeExpectation = expectation(description: "Sink subscribed")
let serialQueue = DispatchQueue(label: "testSubscriptionLeaks")
let subject = PassthroughSubject<Int, Never>()

var results: [Int] = []
```

```
let cancellable = subject
    .subscribe(on: serialQueue)
    .handleEvents(receiveSubscription: { _ in
        subscribeExpectation.fulfill()
    })
    .sink { value in
        results.append(value)
    }

// Пробуем отменить подписку до ее получения
cancellable.cancel()

wait(for: [subscribeExpectation], timeout: 1)

// Еще раз пробуем отменить подписку уже после получения
cancellable.cancel()

// Проверяем, отправляются ли значения
subject.send(1)

XCTAssertEqual([], results) // [1]
```

SubscribeOn

```
let subscribeExpectation = expectation(description: "Sink subscribed")
let serialQueue = DispatchQueue(label: "testSubscriptionLeaks")
let subject = PassthroughSubject<Int, Never>()

var results: [Int] = []
```

```
let cancellable = subject
    .subscribe(on: serialQueue)
    .handleEvents(receiveSubscription: { _ in
        subscribeExpectation.fulfill()
    })
    .sink { value in
        results.append(value)
    }

// Пробуем отменить подписку до ее получения
cancellable.cancel()

wait(for: [subscribeExpectation], timeout: 1)

// Еще раз пробуем отменить подписку уже после получения
cancellable.cancel()

// Проверяем, отправляются ли значения
subject.send(1)

XCTAssertEqual([], results) // [1]
```

SubscribeOn

```
let subscribeExpectation = expectation(description: "Sink subscribed")
let serialQueue = DispatchQueue(label: "testSubscriptionLeaks")
let subject = PassthroughSubject<Int, Never>()

var results: [Int] = []
```

```
let cancellable = subject
    .subscribe(on: serialQueue)
    .handleEvents(receiveSubscription: { _ in
        subscribeExpectation.fulfill()
    })
    .sink { value in
        results.append(value)
    }

// Пробуем отменить подписку до ее получения
cancellable.cancel()

wait(for: [subscribeExpectation], timeout: 1)

// Еще раз пробуем отменить подписку уже после получения
cancellable.cancel()

// Проверяем, отправляются ли значения
subject.send(1)

XCTAssertEqual([], results) // [1]
```

SubscribeOn

```
let subscribeExpectation = expectation(description: "Sink subscribed")
let serialQueue = DispatchQueue(label: "testSubscriptionLeaks")
let subject = PassthroughSubject<Int, Never>()

var results: [Int] = []
```

```
let cancellable = subject
    .subscribe(on: serialQueue)
    .handleEvents(receiveSubscription: { _ in
        subscribeExpectation.fulfill()
    })
    .sink { value in
        results.append(value)
    }

// Пробуем отменить подписку до ее получения
cancellable.cancel()

wait(for: [subscribeExpectation], timeout: 1)

// Еще раз пробуем отменить подписку уже после получения
cancellable.cancel()

// Проверяем, отправляются ли значения
subject.send(1)

XCTAssertEqual([], results) // [1]
```

SubscribeOn

```
let subscribeExpectation = expectation(description: "Sink subscribed")
let serialQueue = DispatchQueue(label: "testSubscriptionLeaks")
let subject = PassthroughSubject<Int, Never>()

var results: [Int] = []
```

```
let cancellable = subject
    .subscribe(on: serialQueue)
    .handleEvents(receiveSubscription: { _ in
        subscribeExpectation.fulfill()
    })
    .sink { value in
        results.append(value)
    }

// Пробуем отменить подписку до ее получения
cancellable.cancel()

wait(for: [subscribeExpectation], timeout: 1)

// Еще раз пробуем отменить подписку уже после получения
cancellable.cancel()

// Проверяем, отправляются ли значения
subject.send(1)

XCTAssertEqual([], results) // [1]
```

SubscribeOn

```
let subscribeExpectation = expectation(description: "Sink subscribed")
let serialQueue = DispatchQueue(label: "testSubscriptionLeaks")
let subject = PassthroughSubject<Int, Never>()

var results: [Int] = []
```

```
let cancellable = subject
    .subscribe(on: serialQueue)
    .handleEvents(receiveSubscription: { _ in
        subscribeExpectation.fulfill()
    })
    .sink { value in
        results.append(value)
    }

// Пробуем отменить подписку до ее получения
cancellable.cancel()

wait(for: [subscribeExpectation], timeout: 1)

// Еще раз пробуем отменить подписку уже после получения
cancellable.cancel()

// Проверяем, отправляются ли значения
subject.send(1)

XCTAssertEqual([], results) // [1]
```

SubscribeOn

```
let subscribeExpectation = expectation(description: "Sink subscribed")
let serialQueue = DispatchQueue(label: "testSubscriptionLeaks")
let subject = PassthroughSubject<Int, Never>()

var results: [Int] = []
```

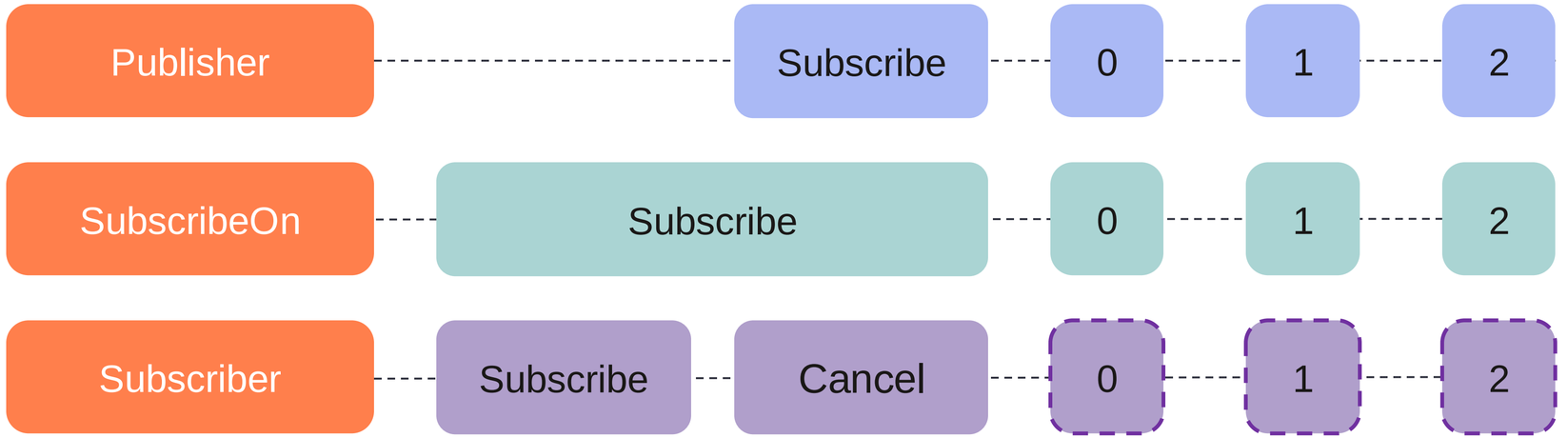
```
let cancellable = subject
    .subscribe(on: serialQueue)
    .handleEvents(receiveSubscription: { _ in
        subscribeExpectation.fulfill()
    })
    .sink { value in
        results.append(value)
    }

// Пробуем отменить подписку до ее получения
cancellable.cancel()

wait(for: [subscribeExpectation], timeout: 1)

// Еще раз пробуем отменить подписку уже после получения
cancellable.cancel()

// Проверяем, отправляются ли значения
subject.send(1)

XCTAssertEqual([], results) // [1]
```

SubscribeOn

```
let cancelExpectation = expectation(description: "Subscription cancelled")
let serialQueue = DispatchQueue(label: "testSubscriptionLeaks2")

var results: [Int] = []
```

```
let timer = Timer.publish(every: 1, on: .main, in: .common)
    .autoconnect()
    .map { _ in 1 }
    .scan(0, +)

let cancellable = timer
    .subscribe(on: serialQueue)
    .handleEvents(receiveCancel: {
        cancelExpectation.fulfill()
    })
    .sink { value in
        results.append(value)
    }

cancellable.cancel()

wait(for: [cancelExpectation], timeout: 5) // Asynchronous wait failed

XCTAssertEqual([], results) // [1, 2, 3, 4, 5]
```

Причины



After you receive one call to cancel(), subsequent calls shouldn’t do anything. Canceling should also eliminate any strong references it currently holds.

Что можно сделать?

- Не использовать subscribe(on:options:),
  заменить его DispatchQueue.async()
- Всегда вызывать cancel() на той же очереди,
  на которой был выполнен subscribe(on:options:)
- Написать безопасную версию subscribe(on:options:)

Что можно сделать?

- Не использовать subscribe(on:options:),
  заменить его DispatchQueue.async()
- Всегда вызывать cancel() на той же очереди,
  на которой был выполнен subscribe(on:options:)
- Написать безопасную версию subscribe(on:options:)

Что можно сделать?

- Не использовать subscribe(on:options:),
  заменить его DispatchQueue.async()
- Всегда вызывать cancel() на той же очереди,
  на которой был выполнен subscribe(on:options:)
- Написать безопасную версию subscribe(on:options:)

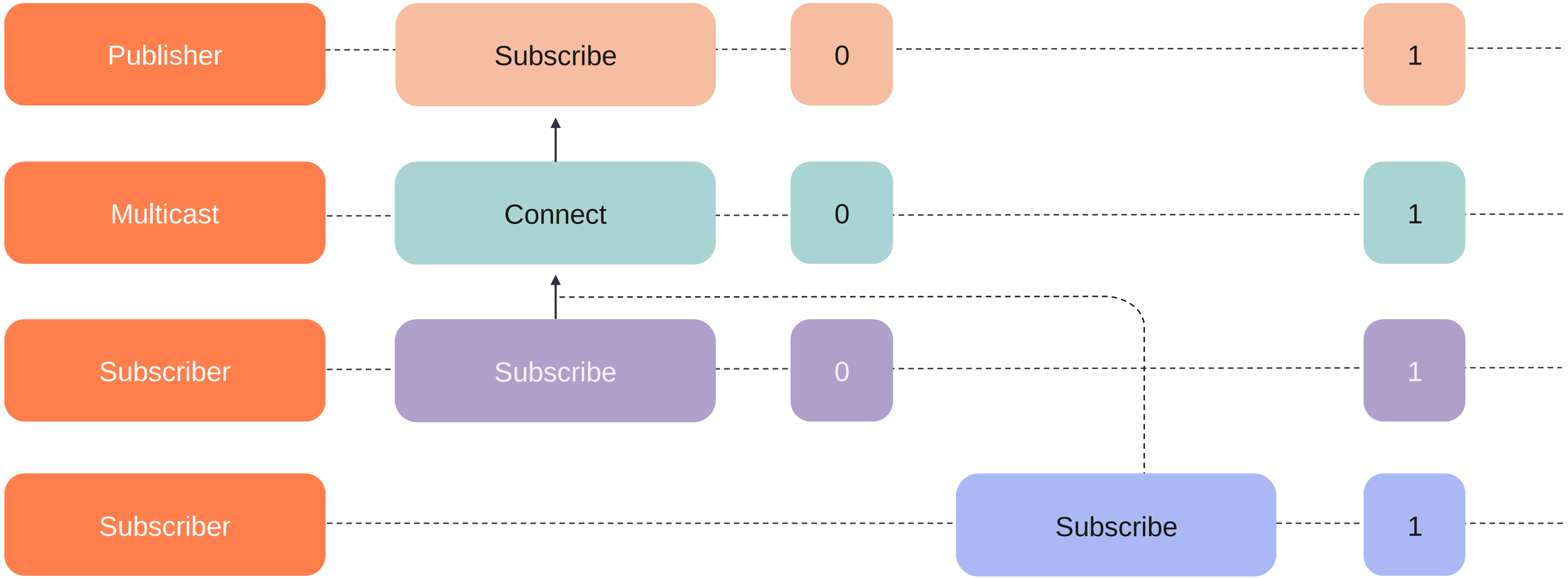
Operators

- CombineLatest
- SubscribeOn
- Multicast

Multicast


Matt Gallagher
Cocoa with Love
22 short tests of combine – Part 2: Sharing
August 17, 2019 by Matt Gallagher
Tags: combine, reactive-programming

Multicast


Publisher
Subscribe
0
1
Multicast
Connect
0
1
Subscriber
Subscribe
0
1
Subscriber
Subscribe
1

Multicast

```
let subject = PassthroughSubject<Int, Never>()

var upstreamResults: [Int] = []
var upstreamCancelled: Bool = false

var sinkResults: [Int] = []
var sinkCancelled: Bool = false
var sinkCancellable: (any Cancellable)?

var multicastCancellable: (any Cancellable)?
```

```
do {
    // Создаем мультикаст
    let multicast = subject
        .handleEvents(receiveOutput: {
            upstreamResults.append($0)
        }, receiveCancel: {
            upstreamCancelled = true
        })
        .multicast(subject: PassthroughSubject())

    // Создаем подписчика
    sinkCancellable = multicast
        .handleEvents(receiveCancel: {
            sinkCancelled = true
        })
        .sink { sinkResults.append($0) }

    // Активируем мультикаст
    multicastCancellable = multicast.connect()
}
```

Multicast

```
let subject = PassthroughSubject<Int, Never>()

var upstreamResults: [Int] = []
var upstreamCancelled: Bool = false

var sinkResults: [Int] = []
var sinkCancelled: Bool = false
var sinkCancellable: (any Cancellable)?

var multicastCancellable: (any Cancellable)?
```

```
do {
    // Создаем мультикаст
    let multicast = subject
        .handleEvents(receiveOutput: {
            upstreamResults.append($0)
        }, receiveCancel: {
            upstreamCancelled = true
        })
        .multicast(subject: PassthroughSubject())

    // Создаем подписчика
    sinkCancellable = multicast
        .handleEvents(receiveCancel: {
            sinkCancelled = true
        })
        .sink { sinkResults.append($0) }

    // Активируем мультикаст
    multicastCancellable = multicast.connect()
}
```

Multicast

```
let subject = PassthroughSubject<Int, Never>()

var upstreamResults: [Int] = []
var upstreamCancelled: Bool = false

var sinkResults: [Int] = []
var sinkCancelled: Bool = false
var sinkCancellable: (any Cancellable)?

var multicastCancellable: (any Cancellable)?
```

```
do {
    // Создаем мультикаст
    let multicast = subject
        .handleEvents(receiveOutput: {
            upstreamResults.append($0)
        }, receiveCancel: {
            upstreamCancelled = true
        })
        .multicast(subject: PassthroughSubject())

    // Создаем подписчика
    sinkCancellable = multicast
        .handleEvents(receiveCancel: {
            sinkCancelled = true
        })
        .sink { sinkResults.append($0) }

    // Активируем мультикаст
    multicastCancellable = multicast.connect()
}
```

Multicast

```
let subject = PassthroughSubject<Int, Never>()

var upstreamResults: [Int] = []
var upstreamCancelled: Bool = false

var sinkResults: [Int] = []
var sinkCancelled: Bool = false
var sinkCancellable: (any Cancellable)?

var multicastCancellable: (any Cancellable)?
```

```
do {
    // Создаем мультикаст
    let multicast = subject
        .handleEvents(receiveOutput: {
            upstreamResults.append($0)
        }, receiveCancel: {
            upstreamCancelled = true
        })
        .multicast(subject: PassthroughSubject())

    // Создаем подписчика
    sinkCancellable = multicast
        .handleEvents(receiveCancel: {
            sinkCancelled = true
        })
        .sink { sinkResults.append($0) }

    // Активируем мультикаст
    multicastCancellable = multicast.connect()
}
```

Multicast

```
let subject = PassthroughSubject<Int, Never>()

var upstreamResults: [Int] = []
var upstreamCancelled: Bool = false

var sinkResults: [Int] = []
var sinkCancelled: Bool = false
var sinkCancellable: (any Cancellable)?

var multicastCancellable: (any Cancellable)?
```

```
do {
    // Создаем мультикаст
    let multicast = subject
        .handleEvents(receiveOutput: {
            upstreamResults.append($0)
        }, receiveCancel: {
            upstreamCancelled = true
        })
        .multicast(subject: PassthroughSubject())

    // Создаем подписчика
    sinkCancellable = multicast
        .handleEvents(receiveCancel: {
            sinkCancelled = true
        })
        .sink { sinkResults.append($0) }

    // Активируем мультикаст
    multicastCancellable = multicast.connect()
}
```

Multicast

```
let subject = PassthroughSubject<Int, Never>()

var upstreamResults: [Int] = []
var upstreamCancelled: Bool = false

var sinkResults: [Int] = []
var sinkCancelled: Bool = false
var sinkCancellable: (any Cancellable)?

var multicastCancellable: (any Cancellable)?
```

```
do {
    // Создаем мультикаст
    let multicast = subject
        .handleEvents(receiveOutput: {
            upstreamResults.append($0)
        }, receiveCancel: {
            upstreamCancelled = true
        })
        .multicast(subject: PassthroughSubject())

    // Создаем подписчика
    sinkCancellable = multicast
        .handleEvents(receiveCancel: {
            sinkCancelled = true
        })
        .sink { sinkResults.append($0) }

    // Активируем мультикаст
    multicastCancellable = multicast.connect()
}
```

Multicast

```
let subject = PassthroughSubject<Int, Never>()

var upstreamResults: [Int] = []
var upstreamCancelled: Bool = false

var sinkResults: [Int] = []
var sinkCancelled: Bool = false
var sinkCancellable: (any Cancellable)?

var multicastCancellable: (any Cancellable)?
```

```
do {
    // Создаем мультикаст
    let multicast = subject
        .handleEvents(receiveOutput: {
            upstreamResults.append($0)
        }, receiveCancel: {
            upstreamCancelled = true
        })
        .multicast(subject: PassthroughSubject())

    // Создаем подписчика
    sinkCancellable = multicast
        .handleEvents(receiveCancel: {
            sinkCancelled = true
        })
        .sink { sinkResults.append($0) }

    // Активируем мультикаст
    multicastCancellable = multicast.connect()
}
```

Multicast

```
let subject = PassthroughSubject<Int, Never>()

var upstreamResults: [Int] = []
var upstreamCancelled: Bool = false

var sinkResults: [Int] = []
var sinkCancelled: Bool = false
var sinkCancellable: (any Cancellable)?

var multicastCancellable: (any Cancellable)?
```

```
do {
    // Создаем мультикаст
    let multicast = subject
        .handleEvents(receiveOutput: {
            upstreamResults.append($0)
        }, receiveCancel: {
            upstreamCancelled = true
        })
        .multicast(subject: PassthroughSubject())

    // Создаем подписчика
    sinkCancellable = multicast
        .handleEvents(receiveCancel: {
            sinkCancelled = true
        })
        .sink { sinkResults.append($0) }

    // Активируем мультикаст
    multicastCancellable = multicast.connect()
}
```

Multicast

```
do {
    // Создаем мультикаст
    let multicast = subject
        .handleEvents(receiveOutput: {
            upstreamResults.append($0)
        }, receiveCancel: {
            upstreamCancelled = true
        })
        .multicast(subject: PassthroughSubject())

    // Создаем подписчика
    sinkCancellable = multicast
        .handleEvents(receiveCancel: {
            sinkCancelled = true
        })
        .sink { sinkResults.append($0) }

    // Активируем мультикаст
    multicastCancellable = multicast.connect()
}
```

```
subject.send(1)

sinkCancellable?.cancel()

multicastCancellable?.cancel()

// Отправляем еще одно значение, чтобы убедиться
subject.send(2)

XCTAssertTrue(sinkCancelled) // true
XCTAssertTrue(upstreamCancelled) // false

XCTAssertEqual([1], sinkResults) // [1]
XCTAssertEqual([1], upstreamResults) // [1, 2]
```

Multicast

```
do {
    // Создаем мультикаст
    let multicast = subject
        .handleEvents(receiveOutput: {
            upstreamResults.append($0)
        }, receiveCancel: {
            upstreamCancelled = true
        })
        .multicast(subject: PassthroughSubject())

    // Создаем подписчика
    sinkCancellable = multicast
        .handleEvents(receiveCancel: {
            sinkCancelled = true
        })
        .sink { sinkResults.append($0) }

    // Активируем мультикаст
    multicastCancellable = multicast.connect()
}
```

```
subject.send(1)


sinkCancellable?.cancel()


multicastCancellable?.cancel()

// Отправляем еще одно значение, чтобы убедиться
subject.send(2)

XCTAssertTrue(sinkCancelled) // true
XCTAssertTrue(upstreamCancelled) // false

XCTAssertEqual([1], sinkResults) // [1]
XCTAssertEqual([1], upstreamResults) // [1, 2]
```

Multicast

```
do {
    // Создаем мультикаст
    let multicast = subject
        .handleEvents(receiveOutput: {
            upstreamResults.append($0)
        }, receiveCancel: {
            upstreamCancelled = true
        })
        .multicast(subject: PassthroughSubject())

    // Создаем подписчика
    sinkCancellable = multicast
        .handleEvents(receiveCancel: {
            sinkCancelled = true
        })
        .sink { sinkResults.append($0) }

    // Активируем мультикаст
    multicastCancellable = multicast.connect()
}
```

```
subject.send(1)


sinkCancellable?.cancel()


multicastCancellable?.cancel()

// Отправляем еще одно значение, чтобы убедиться
subject.send(2)

XCTAssertTrue(sinkCancelled) // true
XCTAssertTrue(upstreamCancelled) // false

XCTAssertEqual([1], sinkResults) // [1]
XCTAssertEqual([1], upstreamResults) // [1, 2]
```

Multicast

```
do {
    // Создаем мультикаст
    let multicast = subject
        .handleEvents(receiveOutput: {
            upstreamResults.append($0)
        }, receiveCancel: {
            upstreamCancelled = true
        })
        .multicast(subject: PassthroughSubject())

    // Создаем подписчика
    sinkCancellable = multicast
        .handleEvents(receiveCancel: {
            sinkCancelled = true
        })
        .sink { sinkResults.append($0) }

    // Активируем мультикаст
    multicastCancellable = multicast.connect()
}
```

```
subject.send(1)


sinkCancellable?.cancel()


multicastCancellable?.cancel()

// Отправляем еще одно значение, чтобы убедиться
subject.send(2)

XCTAssertTrue(sinkCancelled) // true
XCTAssertTrue(upstreamCancelled) // false

XCTAssertEqual([1], sinkResults) // [1]
XCTAssertEqual([1], upstreamResults) // [1, 2]
```

Multicast

```
do {
    // Создаем мультикаст
    let multicast = subject
        .handleEvents(receiveOutput: {
            upstreamResults.append($0)
        }, receiveCancel: {
            upstreamCancelled = true
        })
        .multicast(subject: PassthroughSubject())

    // Создаем подписчика
    sinkCancellable = multicast
        .handleEvents(receiveCancel: {
            sinkCancelled = true
        })
        .sink { sinkResults.append($0) }

    // Активируем мультикаст
    multicastCancellable = multicast.connect()
}
```

```
subject.send(1)


sinkCancellable?.cancel()


multicastCancellable?.cancel()

// Отправляем еще одно значение, чтобы убедиться
subject.send(2)

XCTAssertTrue(sinkCancelled) // true
XCTAssertTrue(upstreamCancelled) // false

XCTAssertEqual([1], sinkResults) // [1]
XCTAssertEqual([1], upstreamResults) // [1, 2]
```

Multicast

```
do {
    // Создаем мультикаст
    let multicast = subject
        .handleEvents(receiveOutput: {
            upstreamResults.append($0)
        }, receiveCancel: {
            upstreamCancelled = true
        })
        .multicast(subject: PassthroughSubject())

    // Создаем подписчика
    sinkCancellable = multicast
        .handleEvents(receiveCancel: {
            sinkCancelled = true
        })
        .sink { sinkResults.append($0) }

    // Активируем мультикаст
    multicastCancellable = multicast.connect()
}
```

```
subject.send(1)


sinkCancellable?.cancel()


multicastCancellable?.cancel()

// Отправляем еще одно значение, чтобы убедиться
subject.send(2)

XCTAssertTrue(sinkCancelled) // true
XCTAssertTrue(upstreamCancelled) // false

XCTAssertEqual([1], sinkResults) // [1]
XCTAssertEqual([1], upstreamResults) // [1, 2]
```

Multicast

```
do {
    // Создаем мультикаст
    let multicast = subject
        .handleEvents(receiveOutput: {
            upstreamResults.append($0)
        }, receiveCancel: {
            upstreamCancelled = true
        })
        .multicast(subject: PassthroughSubject())

    // Создаем подписчика
    sinkCancellable = multicast
        .handleEvents(receiveCancel: {
            sinkCancelled = true
        })
        .sink { sinkResults.append($0) }

    // Активируем мультикаст
    multicastCancellable = multicast.connect()
}
```

```
subject.send(1)


sinkCancellable?.cancel()


multicastCancellable?.cancel()

// Отправляем еще одно значение, чтобы убедиться
subject.send(2)

XCTAssertTrue(sinkCancelled) // true
XCTAssertTrue(upstreamCancelled) // false

XCTAssertEqual([1], sinkResults) // [1]
XCTAssertEqual([1], upstreamResults) // [1, 2]
```

Multicast

```
do {
    // Создаем мультикаст
    let multicast = subject
        .handleEvents(receiveOutput: {
            upstreamResults.append($0)
        }, receiveCancel: {
            upstreamCancelled = true
        })
        .multicast(subject: PassthroughSubject())

    // Создаем подписчика
    sinkCancellable = multicast
        .handleEvents(receiveCancel: {
            sinkCancelled = true
        })
        .sink { sinkResults.append($0) }

    // Активируем мультикаст
    multicastCancellable = multicast.connect()
}

subject.send(1)

sinkCancellable?.cancel()

multicastCancellable?.cancel()

// Отправляем еще одно значение, чтобы убедиться
subject.send(2)

XCTAssertTrue(sinkCancelled) // true
XCTAssertTrue(upstreamCancelled) // false

XCTAssertEqual([1], sinkResults) // [1]
XCTAssertEqual([1], upstreamResults) // [1, 2]
```

Multicast

```
do {
    // Создаем мультикаст
    let multicast = subject
        .handleEvents(receiveOutput: {
            upstreamResults.append($0)
        }, receiveCancel: {
            upstreamCancelled = true
        })
        .multicast(subject: PassthroughSubject())

    // Создаем подписчика
    sinkCancellable = multicast
        .handleEvents(receiveCancel: {
            sinkCancelled = true
        })
        .sink { sinkResults.append($0) }

    // Активируем мультикаст
    multicastCancellable = multicast.connect()
}
subject.send(1)


// Последняя сильная ссылка на мультикаст исчезает здесь
sinkCancellable?.cancel()


multicastCancellable?.cancel()

// Отправляем еще одно значение, чтобы убедиться
subject.send(2)

XCTAssertTrue(sinkCancelled) // true
XCTAssertTrue(upstreamCancelled) // false

XCTAssertEqual([1], sinkResults) // [1]
XCTAssertEqual([1], upstreamResults) // [1, 2]
```

Multicast

```
do {
    // Создаем мультикаст
    let multicast = subject
        .handleEvents(receiveOutput: {
            upstreamResults.append($0)
        }, receiveCancel: {
            upstreamCancelled = true
        })
        .multicast(subject: PassthroughSubject())

    // Создаем подписчика
    sinkCancellable = multicast
        .handleEvents(receiveCancel: {
            sinkCancelled = true
        })
        .sink { sinkResults.append($0) }

    // Активируем мультикаст
    multicastCancellable = multicast.connect()
}

subject.send(1)


// Последняя сильная ссылка на мультикаст исчезает здесь
sinkCancellable?.cancel()


// Теперь токен бесполезен
multicastCancellable?.cancel()

// Отправляем еще одно значение, чтобы убедиться
subject.send(2)

XCTAssertTrue(sinkCancelled) // true
XCTAssertTrue(upstreamCancelled) // false

XCTAssertEqual([1], sinkResults) // [1]
XCTAssertEqual([1], upstreamResults) // [1, 2]
```

Выводы

Если вы создаете мультикаст, то при его активации обязательно замыкайте его сильной ссылкой

```
let multicastCancellable = multicast.connect()
let cancellable = AnyCancellable { [multicast] in
    _ = multicast // !!! DO NOT DELETE THIS LINE !!!
    multicastCancellable.cancel()
}
```

- Subjects
- Operators
- Contracts
- API

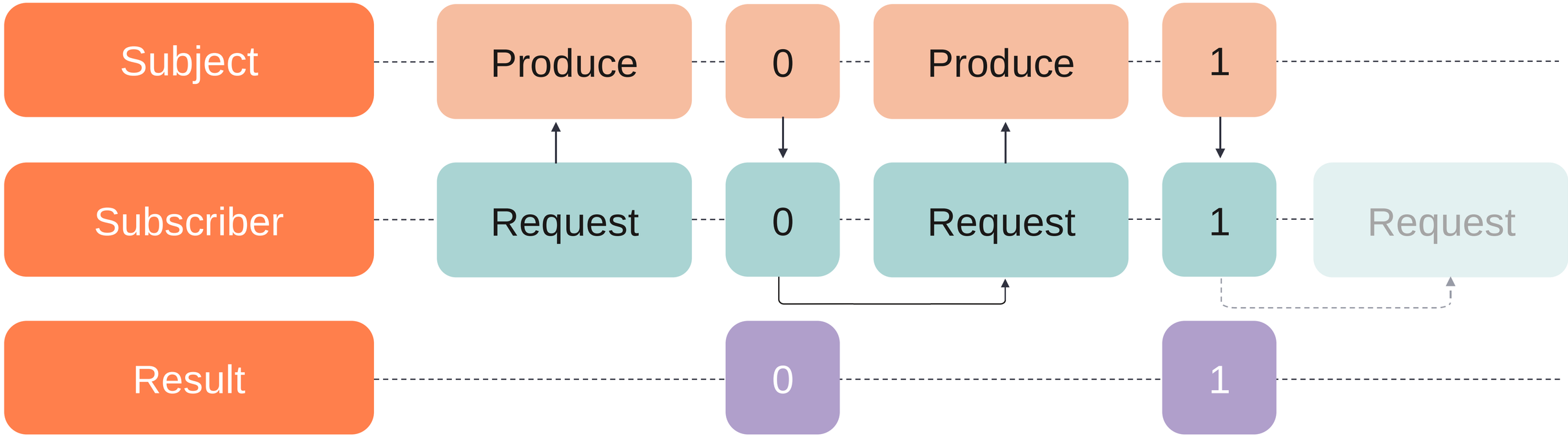
Contracts

- Back Pressure
- Scheduler’s
- Subscription

Contracts

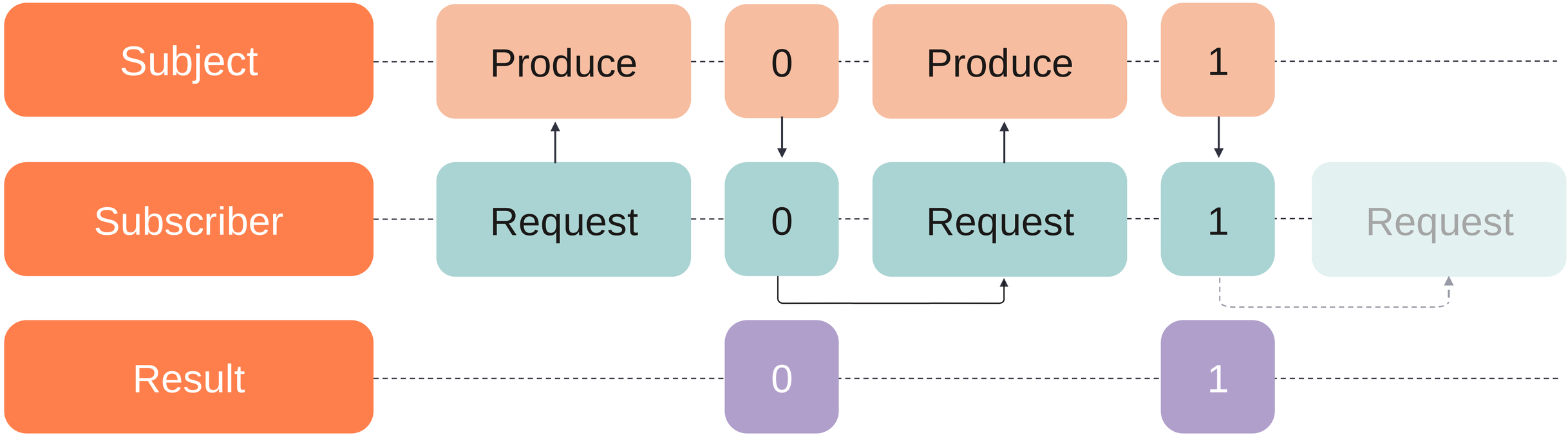
- Back Pressure
- Scheduler’s
- Subscription

Back Pressure



Back Pressure

Или почему оно не работает

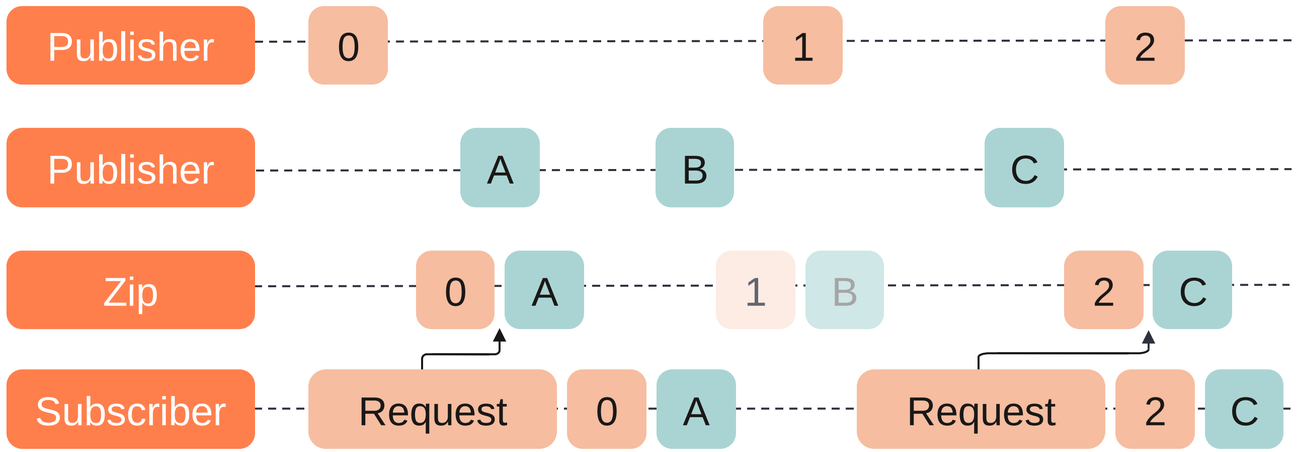


Back Pressure

- Zip
- CombineLatest
- Merge

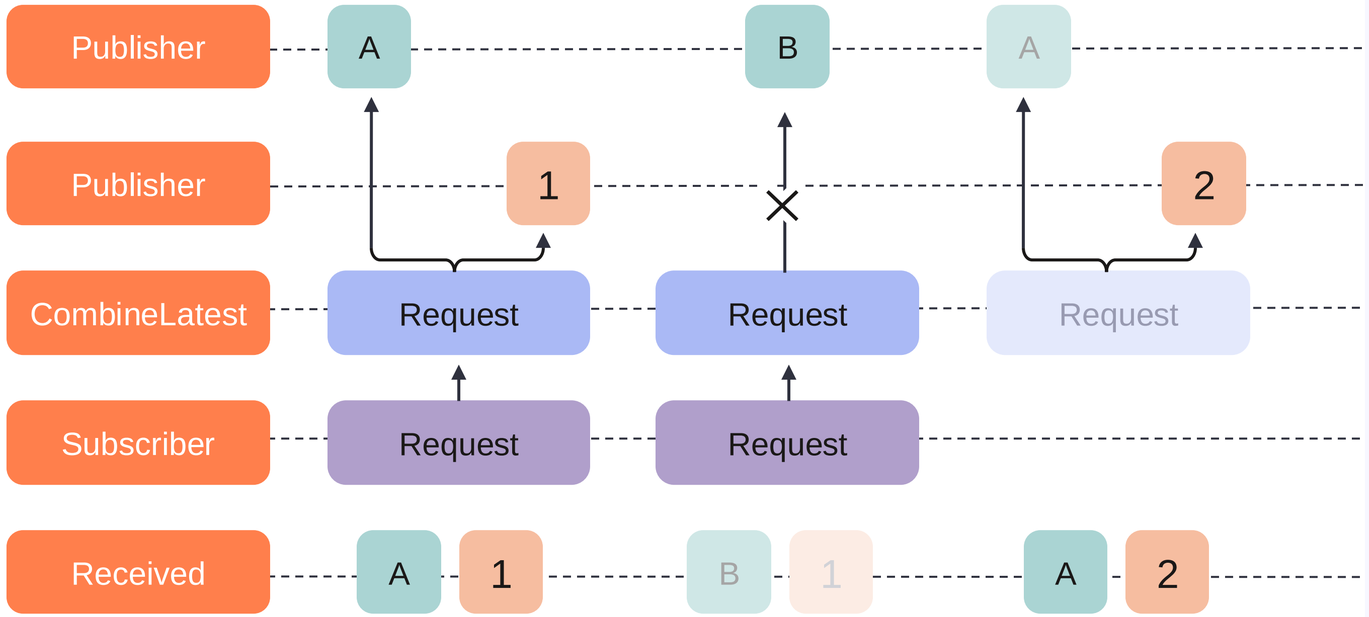
Подробности в статье,
дальше краткие результаты

Zip



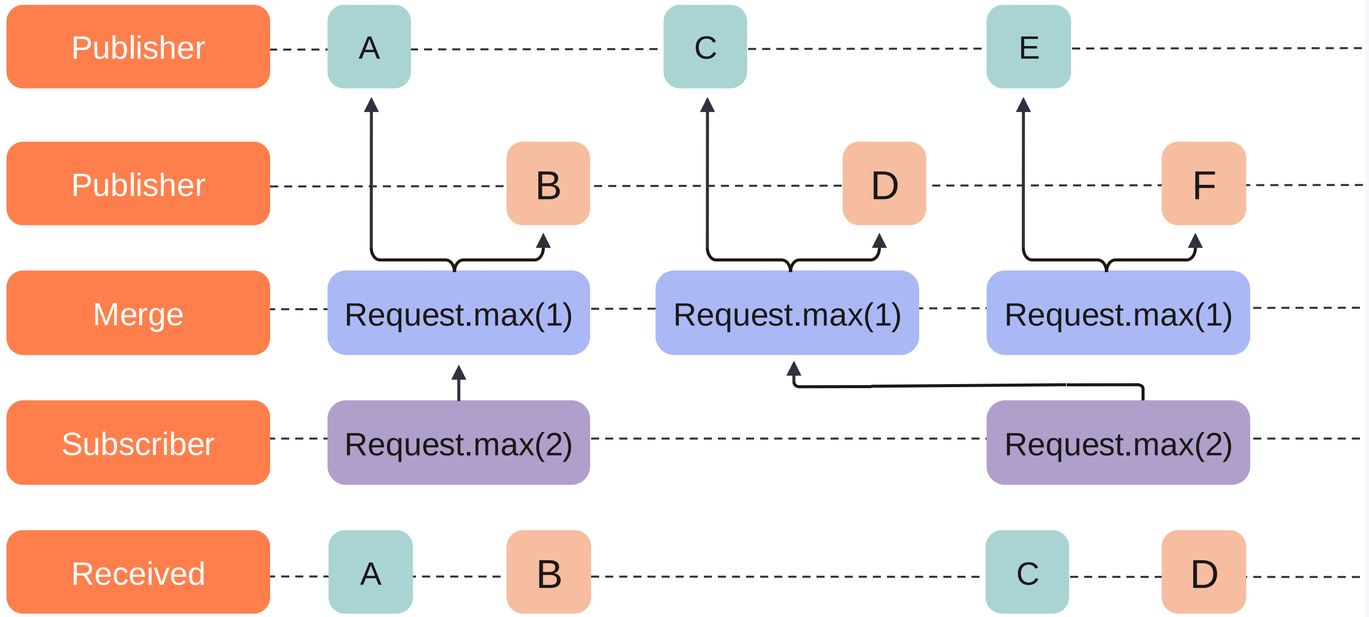

```
let otherPublisher = PassthroughSubject<Int, Never>()

(0 ... Int.max).publisher
    .zip(otherPublisher)
    .sink { _ in }
```

CombineLatest




```
publisher1
    .combineLatest( publisher2)
    .subscribe(
        // Подписчик, запрашивающий элементу по-одному
        AnySubscriber { subscription in
            subscription.request(.max(1))
        } receiveValue: { value in
            results.append("\(value.0)\(value.1)")
            return .max(1)
        }
    )

publisher1.send("A")
publisher2.send(1)

publisher1.send("B") // Будет проигнорирован
publisher2.send(2)

XCTAssertEqual(["A1", "B1", "B2"], results) // ["A1", "A2"]
```

Merge




```
publisher1
    .merge(with: publisher2)
    .subscribe(
        AnySubscriber { s in
            subscription = s
        } receiveValue: { value in
            results.append(value)
            return .none
        }
    )

subscription.request(.max(2))

publisher1.send("A")
publisher2.send("B") // Demand был удовлетворен

// Следующие эвенты должны быть проигнорированы
publisher1.send("C")
publisher2.send("D")

subscription.request(.max(2))

XCTAssertEqual(["A", "B", "C", "D"], results) // ["A", "B", "C", "D"]
```

Выводы

Back Pressure сломан

Что делать?

Всегда запрашивать .unlimited деманд,
как, к слову, и делает Sink, и использовать
проверенные ~~временем~~ RxSwift-ом:
throttle(for:scheduler:latest:),
debounce(for:scheduler:options:) и collect(_:options:)

Contracts

- Back Pressure
- Scheduler’s
- Subscription

Scheduler и ConcurrentQueue

- Нет гарантии порядка получения эвентов ConcurrentQueue
- Безопасность использования ConcurrentQueue
- Scheduler используют только async

Scheduler и ConcurrentQueue

- Нет гарантии порядка получения эвентов ConcurrentQueue
- Безопасность использования ConcurrentQueue
- Scheduler используют только async

Scheduler и ConcurrentQueue

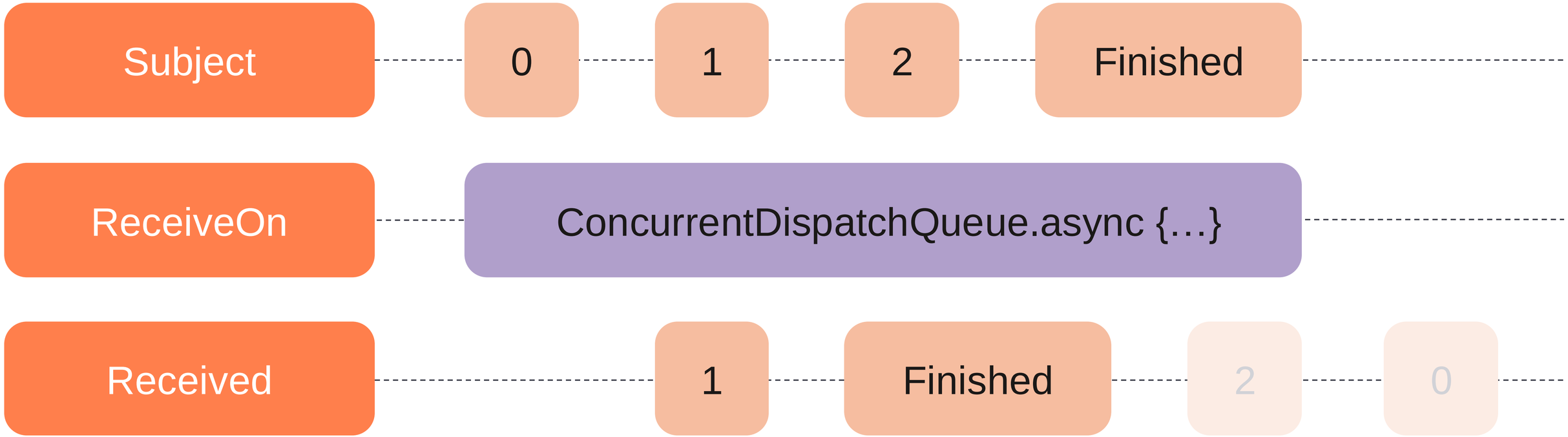
- Нет гарантии порядка получения эвентов ConcurrentQueue
- Безопасность использования ConcurrentQueue
- Scheduler используют только async

Scheduler и ConcurrentQueue

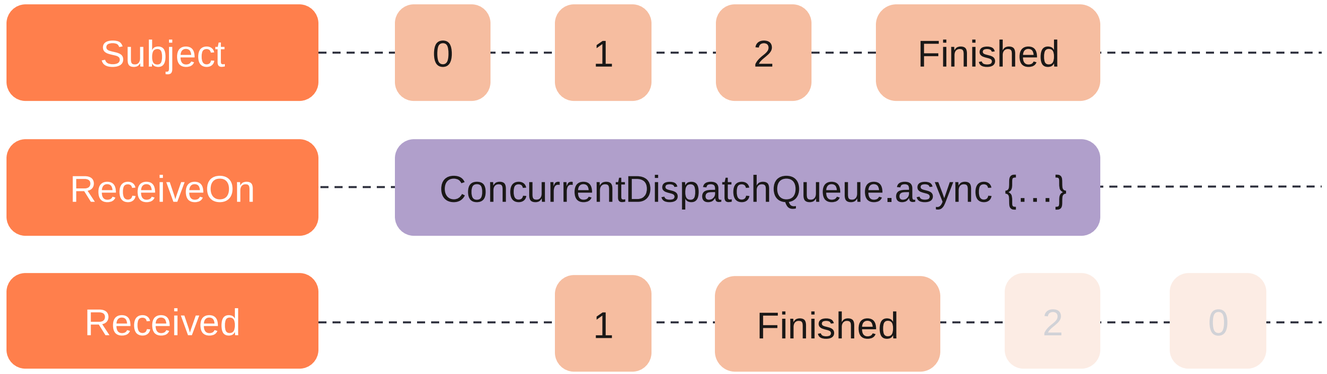
- Нет гарантии порядка получения эвентов ConcurrentQueue
- Безопасность использования ConcurrentQueue
- Scheduler используют только async

Scheduler

Нет гарантии порядка получения эвентов для ConcurrentQueue



Scheduler




```
let finishExpectation = expectation(description: "Publisher Finished")
var results: [Int] = []

Just(0)
    .receive(on: DispatchQueue.global())
    .sink { _ in
        finishExpectation.fulfill()
    } receiveValue: { value in
        results.append(value)
    }

waitForExpectations(timeout: 1)

XCTAssertEqual([0], results) // []
```

Scheduler

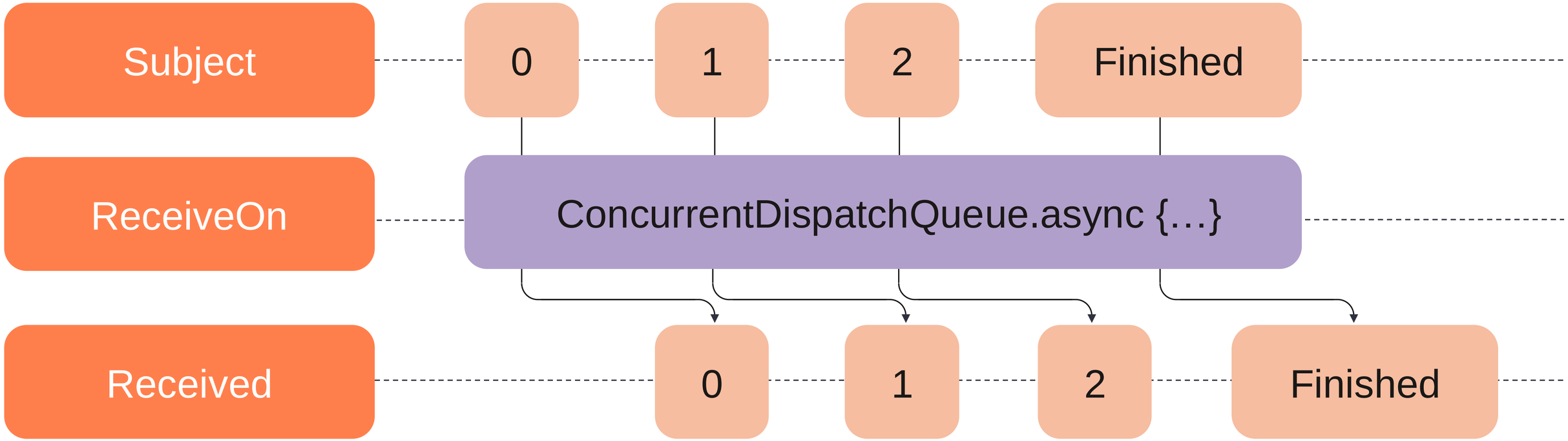
Дата рейсы при использовании бэкграунд очередей

```
Just(0)
    .subscribe(
        AnySubscriber { s in
            subscription = s
        }
    )

DispatchQueue.global().async {
    subscription?.request(.max(1))
}

DispatchQueue.global().async {
    subscription?.request(.max(1)) // Возможный EXC_BAD_ACCESS
}
```

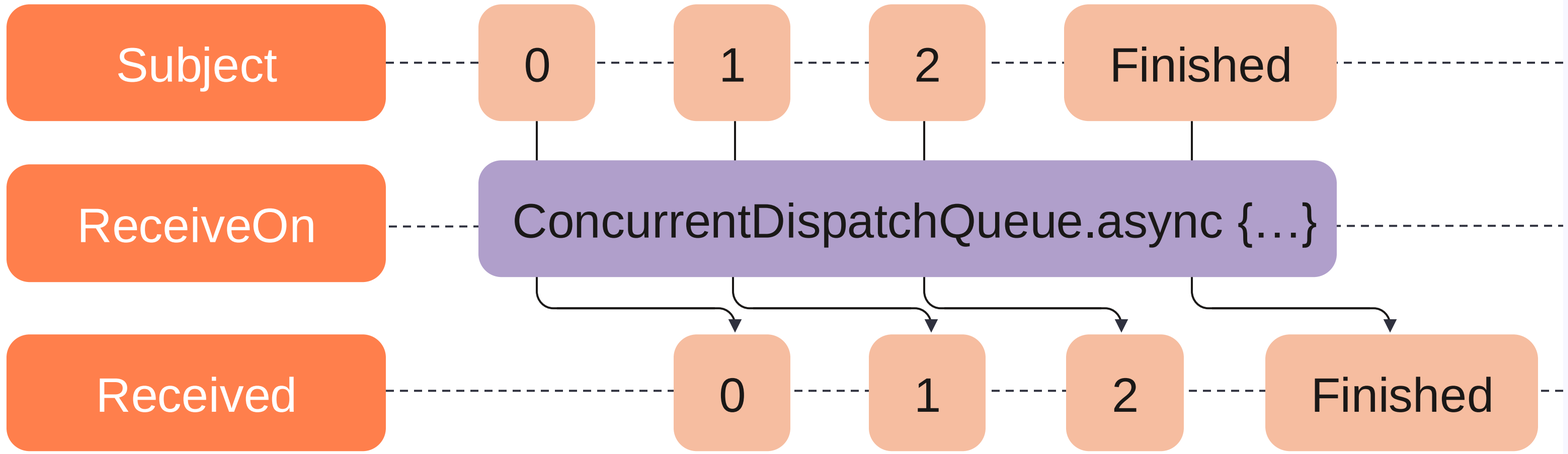
Thread 6: EXC_BAD_ACCESS (code=257, address=0x1fa59dd29)

Scheduler

Schedulers используют только async



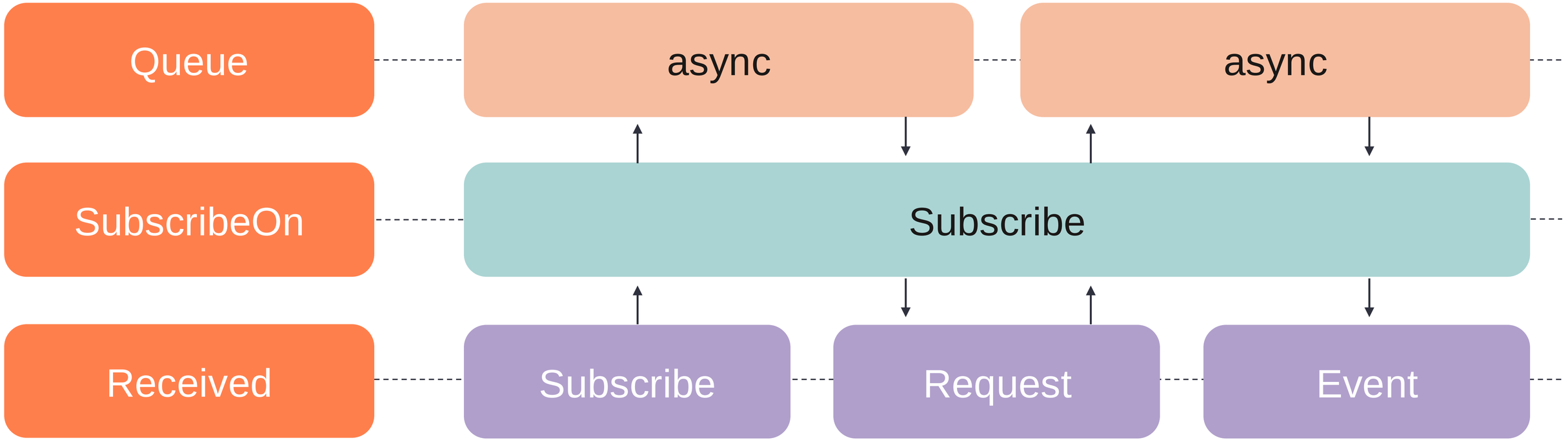
Scheduler



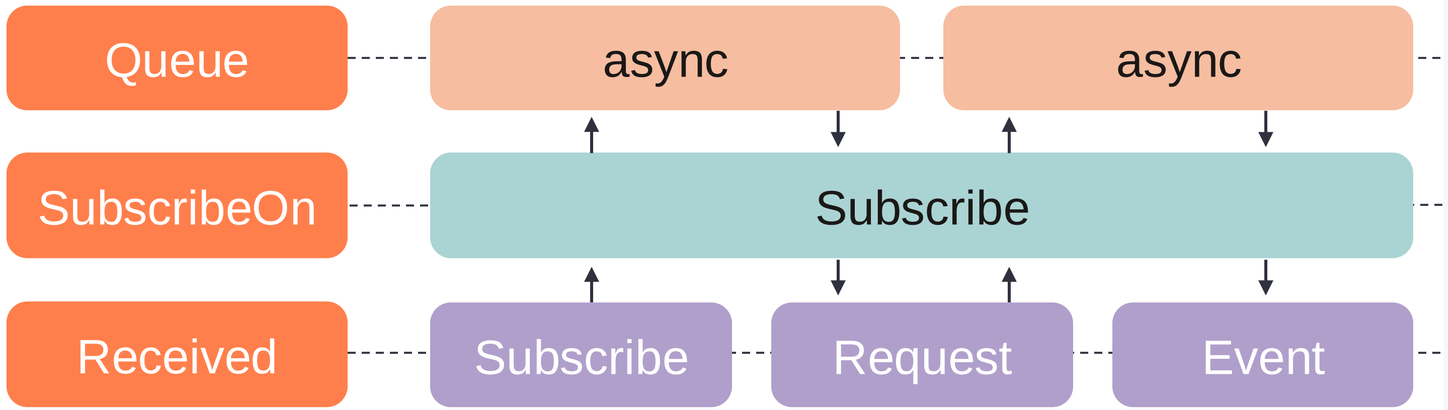

```swift
let subject = PassthroughSubject<Int, Never>()
var results: [Int] = []

subject
    .receive(on: DispatchQueue.main)
    .sink { value in results.append(value) }

// Отправляем эвент с основной очереди
subject.send(0)

// Ожидаем сразу же его получить
XCTAssertEqual([0], results) // []
```

Scheduler

Schedulers используют только async



Scheduler




```
let scheduler = TestScheduler()
var results: [Int] = []

Just(1)
    .subscribe(on: scheduler)
    .sink { value in results.append(value) }

scheduler.waitUntilAllScheduled()

XCTAssertEqual([1], results) // [1]
XCTAssertEqual(scheduler.schedulingCount, 0) // 0
XCTAssertEqual(scheduler.scheduledCount, 2) // 2
```

Что делать?

- Никогда использовать ConcurrentQueue
- Написать свой аналог MainScheduler или UIScheduler, чтобы избежать лишних переключений

Contracts

- Back Pressure
- Scheduler’s
- Subscription

Subscription

```
public protocol Subscription : Cancellable { /* … */ }
```

Subscription

```
/// Tells a publisher that it may send more values to the subscriber.
func request(_ demand: Subscribers.Demand)
```

Subscription

```
public protocol Cancellable {
    func cancel()
}
```

Subscription

```
let finishExpectation = expectation(description: "Publisher finished")
finishExpectation.expectedFulfillmentCount = 2

var subscription: (any Subscription)?
var cancellationCounter = 0
```

```
Just(0)
    .handleEvents(receiveCancel: {
        cancellationCounter += 1
    })
    .subscribe(
        AnySubscriber { s in
            subscription = s
        }
    )

DispatchQueue.global().async {
    subscription?.cancel()
    finishExpectation.fulfill()
}
DispatchQueue.global().async {
    subscription?.cancel()
    finishExpectation.fulfill()
}

waitForExpectations(timeout: 1)
XCTAssertEqual(1, cancellationCounter) // 2
```

Subscription

```
let finishExpectation = expectation(description: "Publisher finished")
finishExpectation.expectedFulfillmentCount = 2

var subscription: (any Subscription)?
var cancellationCounter = 0
```

```
Just(0)
    .handleEvents(receiveCancel: {
        cancellationCounter += 1
    })
    .subscribe(
        AnySubscriber { s in
            subscription = s
        }
    )

DispatchQueue.global().async {
    subscription?.cancel()
    finishExpectation.fulfill()
}
DispatchQueue.global().async {
    subscription?.cancel()
    finishExpectation.fulfill()
}

waitForExpectations(timeout: 1)
XCTAssertEqual(1, cancellationCounter) // 2
```

Subscription

```
let finishExpectation = expectation(description: "Publisher finished")
finishExpectation.expectedFulfillmentCount = 2

var subscription: (any Subscription)?
var cancellationCounter = 0
```

```
Just(0)
    .handleEvents(receiveCancel: {
        cancellationCounter += 1
    })
    .subscribe(
        AnySubscriber { s in
            subscription = s
        }
    )

DispatchQueue.global().async {
    subscription?.cancel()
    finishExpectation.fulfill()
}
DispatchQueue.global().async {
    subscription?.cancel()
    finishExpectation.fulfill()
}

waitForExpectations(timeout: 1)
XCTAssertEqual(1, cancellationCounter) // 2
```

Subscription

```
let finishExpectation = expectation(description: "Publisher finished")
finishExpectation.expectedFulfillmentCount = 2

var subscription: (any Subscription)?
var cancellationCounter = 0
```

```
Just(0)
    .handleEvents(receiveCancel: {
        cancellationCounter += 1
    })
    .subscribe(
        AnySubscriber { s in
            subscription = s
        }
    )

DispatchQueue.global().async {
    subscription?.cancel()
    finishExpectation.fulfill()
}
DispatchQueue.global().async {
    subscription?.cancel()
    finishExpectation.fulfill()
}

waitForExpectations(timeout: 1)
XCTAssertEqual(1, cancellationCounter) // 2
```

Subscription

```
let finishExpectation = expectation(description: "Publisher finished")
finishExpectation.expectedFulfillmentCount = 2

var subscription: (any Subscription)?
var cancellationCounter = 0
```

```
Just(0)
    .handleEvents(receiveCancel: {
        cancellationCounter += 1
    })
    .subscribe(
        AnySubscriber { s in
            subscription = s
        }
    )

DispatchQueue.global().async {
    subscription?.cancel()
    finishExpectation.fulfill()
}
DispatchQueue.global().async {
    subscription?.cancel()
    finishExpectation.fulfill()
}

waitForExpectations(timeout: 1)
XCTAssertEqual(1, cancellationCounter) // 2
```

Subscription

```
let finishExpectation = expectation(description: "Publisher finished")
finishExpectation.expectedFulfillmentCount = 2

var subscription: (any Subscription)?
var cancellationCounter = 0
```

```
Just(0)
    .handleEvents(receiveCancel: {
        cancellationCounter += 1
    })
    .subscribe(
        AnySubscriber { s in
            subscription = s
        }
    )

DispatchQueue.global().async {
    subscription?.cancel()
    finishExpectation.fulfill()
}
DispatchQueue.global().async {
    subscription?.cancel()
    finishExpectation.fulfill()
}

waitForExpectations(timeout: 1)
XCTAssertEqual(1, cancellationCounter) // 2
```

Subscription

```
let finishExpectation = expectation(description: "Publisher finished")
finishExpectation.expectedFulfillmentCount = 2

var subscription: (any Subscription)?
```

```
Just(0)
//    .handleEvents(receiveCancel: {
//        cancellationCounter += 1
//    })
    .subscribe(
        AnySubscriber { s in
            subscription = s
        }
    )

DispatchQueue.global().async {
    subscription?.cancel() // Возможный EXC_BAD_ACCESS
    finishExpectation.fulfill()
}
DispatchQueue.global().async {
    subscription?.cancel() // Возможный EXC_BAD_ACCESS
    finishExpectation.fulfill()
}

waitForExpectations(timeout: 1)
```

Subscription

```
let finishExpectation = expectation(description: "Publisher finished")
finishExpectation.expectedFulfillmentCount = 2

var subscription: (any Subscription)?
```

```
Just(0)
//    .handleEvents(receiveCancel: {
//        cancellationCounter += 1
//    })
    .subscribe(
        AnySubscriber { s in
            subscription = s
        }
    )

DispatchQueue.global().async {
    subscription?.cancel() // Возможный EXC_BAD_ACCESS
    finishExpectation.fulfill()
}
DispatchQueue.global().async {
    subscription?.cancel() // Возможный EXC_BAD_ACCESS
    finishExpectation.fulfill()
}

waitForExpectations(timeout: 1)
```

Subscription

```
let finishExpectation = expectation(description: "Publisher finished")
finishExpectation.expectedFulfillmentCount = 2

var subscription: (any Subscription)?
```

```
Just(0)
//    .handleEvents(receiveCancel: {
//        cancellationCounter += 1
//    })
    .subscribe(
        AnySubscriber { s in
            subscription = s
        }
    )

DispatchQueue.global().async {
    subscription?.request(.max(1)) // EXC_BAD_ACCESS
    finishExpectation.fulfill()
}
DispatchQueue.global().async {
    subscription?.request(.max(1)) // EXC_BAD_ACCESS
    finishExpectation.fulfill()
}

waitForExpectations(timeout: 1)
```

Что это значит?

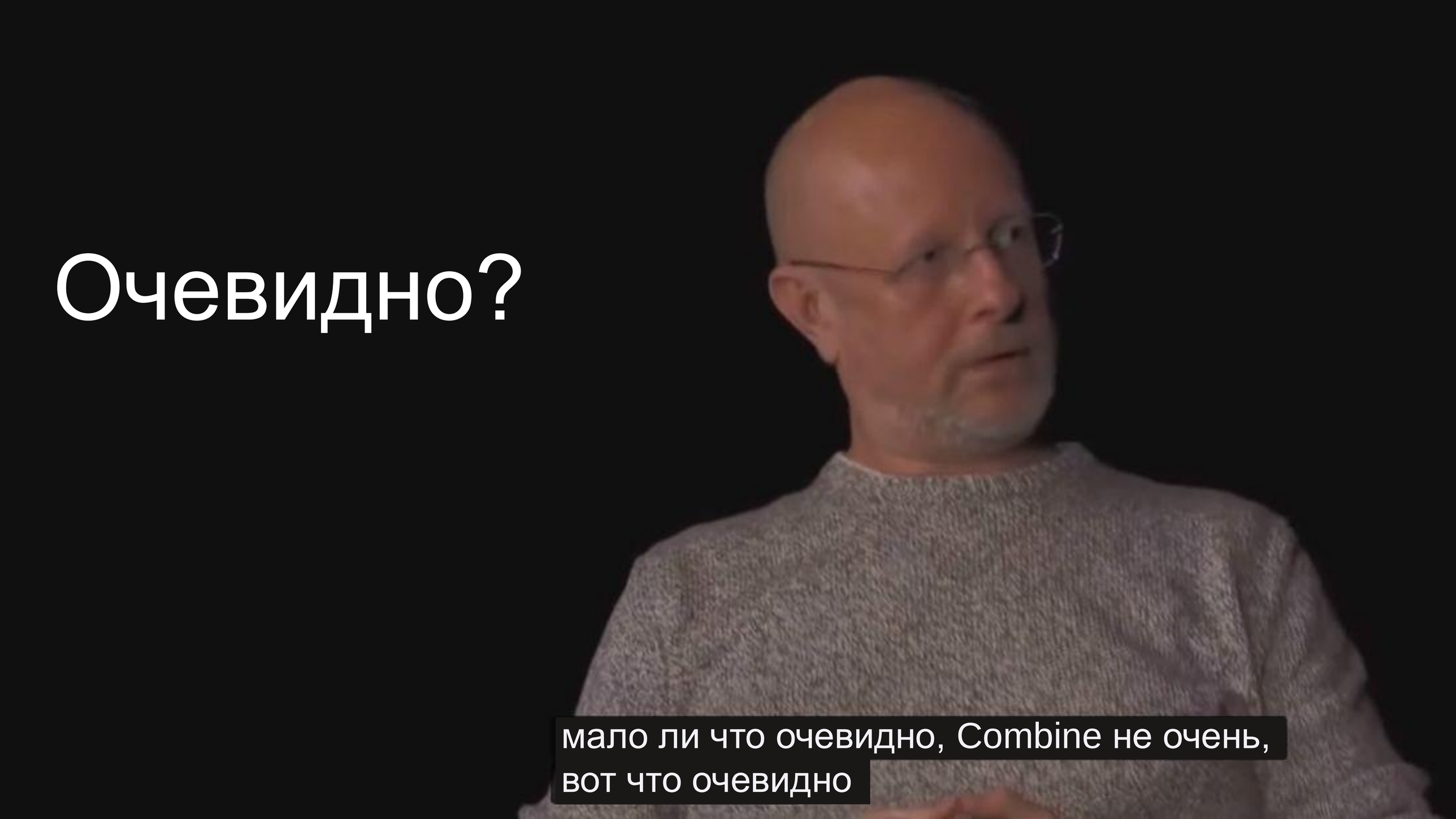
- Не все стандартные операторы соблюдают контракты Subscripton
- Всегда нужно сериализовать вызовы всех методов Subscription, иначе можно получить креши

Что это значит?

- Не все стандартные операторы соблюдают контракты Subscripton
- Всегда нужно сериализовать вызовы всех методов Subscription, иначе можно получить креши

Очевидно?

Очевидно?



Очевидно?

В **самой популярной** библиотеке с экстеншенами нет **ни одного** оператора с потокобезопасным cancel() и request(_:)

CombineExt

CombineExt no status | codecov 95%

platforms iOS 13 | macOS 10.15 | tvOS 13 | watchOS 6

pod v1.8.0 | Swift Package Manager compatible | Carthage compatible

CombineExt provides a collection of operators, publishers and utilities for Combine, that are not provided by Apple themselves, but are common in other Reactive Frameworks and standards.

The original inspiration for many of these additions came from my journey investigating Combine after years of RxSwift and ReactiveX usage.

All operators, utilities and helpers respect Combine's publisher contract, including backpressure.

1.7k stars

30 watching

151 forks

Выводы

Вызовы cancel() и request(_ demand:)
должны быть потокобезопасными

- Subjects
- Operators
- Contracts
- API

Нельзя просто так взять и начать писать на Combine

Или почему мы написали CombineKit

- **CombineExt** – последний коммит 2 года назад
- **CombineX** – последний коммит 3 года назад
- **OpenCombine** – последний коммит год назад

Lack of API

Lack of API

`AnyPublisher.create`

Lack of API

`RetryWhen`

`AnyPublisher.create`

Lack of API

`RetryWhen`

`AnyPublisher.create`

`WithLatestFrom`

Lack of API

`RetryWhen`

`AnyPublisher.create`

`WithLatestFrom`

`InclusivePrefixWhile`

Lack of API

RetryWhen

AnyPublisher.create

WithLatestFrom

InclusivePrefixWhile

Materialize

Lack of API

RetryWhen

AnyPublisher.create

Dematerialize

WithLatestFrom

InclusivePrefixWhile

Materialize

Lack of API

RetryWhen

AnyPublisher.create

Scan

Dematerialize

WithLatestFrom

InclusivePrefixWhile

Materialize

Lack of API

RetryWhen

MainScheduler

AnyPublisher.create

Scan

Dematerialize

WithLatestFrom

InclusivePrefixWhile

Materialize

Lack of API

RetryWhen

MainScheduler

AnyPublisher.create

Scan

Dematerialize

WithLatestFrom

InclusivePrefixWhile

Materialize

UIScheduler

Lack of API

RetryWhen

MainScheduler

CurrentValueSubject

AnyPublisher.create

Scan

Dematerialize

WithLatestFrom

InclusivePrefixWhile

Materialize

UIScheduler

Lack of API

RetryWhen

CombineLatest

MainScheduler

CurrentValueSubject

AnyPublisher.create

Scan

Dematerialize

WithLatestFrom

InclusivePrefixWhile

Materialize

UIScheduler

Lack of API

RetryWhen

CombineLatest

MainScheduler

CurrentValueSubject

AnyPublisher.create

Scan

Dematerialize

WithLatestFrom

InclusivePrefixWhile

Materialize

UIScheduler

Subscription

Lack of API

RetryWhen

CombineLatest

MainScheduler

CurrentValueSubject

AnyPublisher.create

Scan

Dematerialize

WithLatestFrom

InclusivePrefixWhile

Materialize

Multicast

UIScheduler

Subscription

Lack of API

RetryWhen

CombineLatest

MainScheduler

CurrentValueSubject

AnyPublisher.create

Scan

Dematerialize

Zip

WithLatestFrom

InclusivePrefixWhile

Materialize

Multicast

UIScheduler

Subscription

Lack of API

RetryWhen

CombineLatest

MainScheduler

CurrentValueSubject

AnyPublisher.create

Scan

Dematerialize

Zip

WithLatestFrom

InclusivePrefixWhile

Merge

Materialize

Multicast

UIScheduler

Subscription

Lack of API

RetryWhen

CombineLatest

MainScheduler

CurrentValueSubject

AnyPublisher.create

Scan

Schedulers

Dematerialize

Zip

WithLatestFrom

InclusivePrefixWhile

Merge

Materialize

Multicast

UIScheduler

Subscription

Lack of API

RetryWhen

CombineLatest

MainScheduler

BackPressure

CurrentValueSubject

AnyPublisher.create

Scan

Schedulers

Dematerialize

Zip

WithLatestFrom

InclusivePrefixWhile

Merge

Materialize

Multicast

UIScheduler

Subscription

Lack of API

RetryWhen

CombineLatest

MainScheduler

BackPressure

CurrentValueSubject

AnyPublisher.create

SubscribeOn

Scan

Schedulers

Dematerialize

Zip

WithLatestFrom

InclusivePrefixWhile

Merge

Materialize

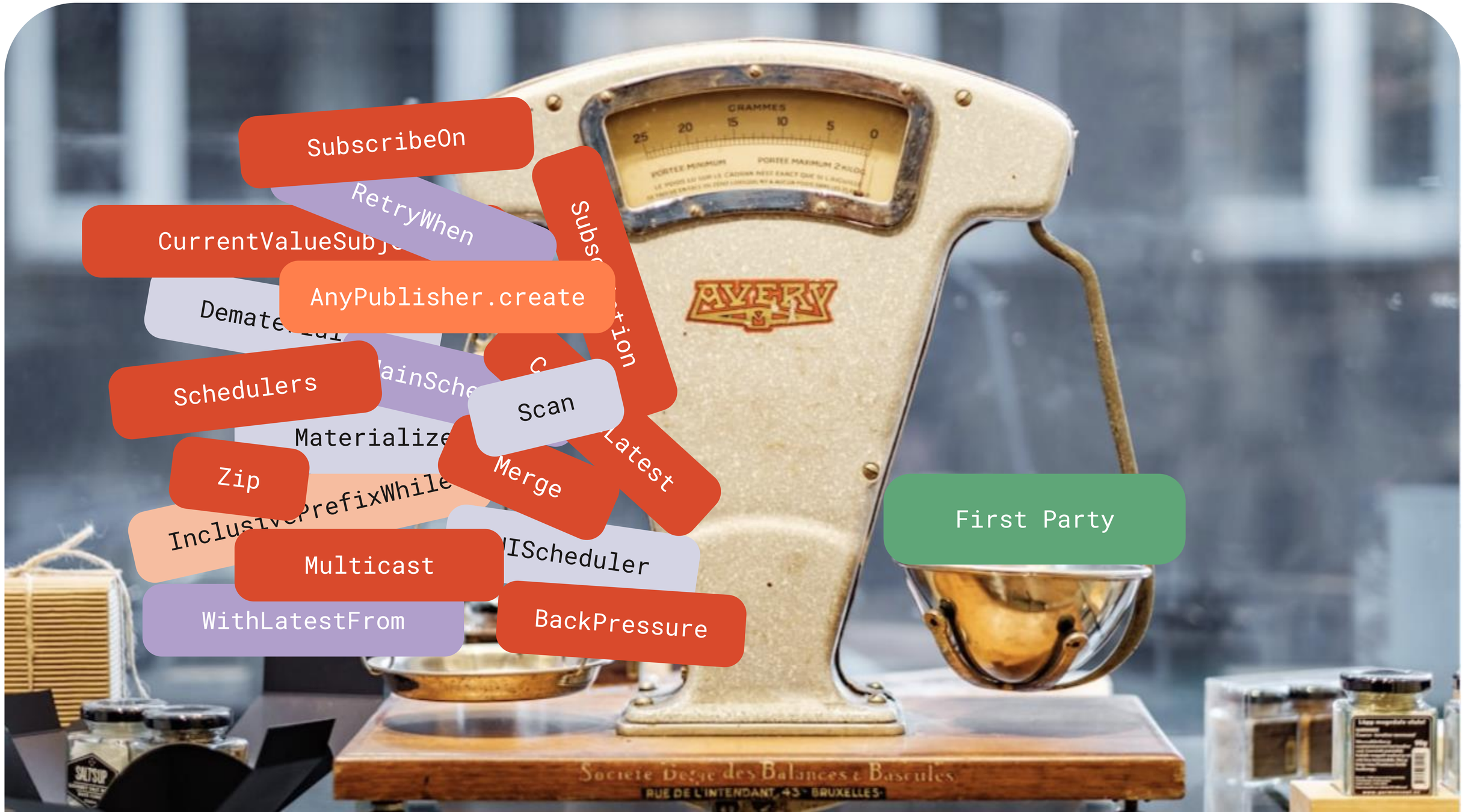
Multicast

UIScheduler

Subscription


GRAMMES
25 20 15 10 5 0
PORTEE MINIMUM
PORTEE MAXIMUM 2 KILOS
AVERY
Societe Belge des Balances à Bascules
RUE DE L'INTENDANT, 45 - BRUXELLES


SubscribeOn
RetryWhen
CurrentValueSubj
AnyPublisher.create
Schedulers
Scan
Materialize
Zip
Merge
Multicast
WithLatestFrom
BackPressure
AVERY
GRAMMES
Societe Belge des Balances et Bascules
RUE DE L'INTENDANT, 45 - BRUXELLES


SubscribeOn
RetryWhen
CurrentValueSubj
AnyPublisher.create
Schedulers
Scan
Materialize
Zip
Merge
Multicast
WithLatestFrom
BackPressure
First Party
AVERY
Societe Belge des Balances et Bascules
RUE DE L'INTENDANT, 45 - BRUXELLES

Выводы

Готовы ли вы променять плюсы first-party библиотеки на скорость и сложность разработки, а также проблемы с поддержкой уже написанного кода?