Combine Architecture

(as of Xcode 11 Beta 5)

2019/08/05 #combine_gorilla

Yasuhiro Inami / @inamiy

WWDC19

#WWDC19

Introducing Combine

Tony Parker, Foundation

Combine.framework

- Official Reactive Programming framework by Apple
    - iOS 13 or later
    - Essential for building data flows in SwiftUI
- Typed `Errors`, no hot / cold `Observable` type separation
- **Rx operators as generic types**
- Supports **Non-blocking Backpressure**

Rx operators as generic types

```
extension Publishers {
    // Used in `func map`.
    struct Map<Upstream, Output> : Publisher where … {
        let upstream: Upstream
        let transform: (Upstream.Output) -> Output
    }

    // Used in `func append` / `func prepend`.
    struct Concatenate<Prefix, Suffix> : Publisher where … {
        let prefix: Prefix
        let suffix: Suffix
    }
}
```

```swift
let publisher = Result<Int, Never>.Publisher(1)
    .append(2)
    .map { $0 }

// Q. What is `type(of: publisher)` ?
```

```swift
let publisher = Result<Int, Never>.Publisher(1)
    .append(2)
    .map { $0 }

/*
    Publishers.Map<
        Publishers.Concatenate<
            Result<Int, Never>.Publisher,
            Publishers.Sequence<[Int], Never>
        >,
        Int
    >
 */
```

```
let publisher = Just<Int>(1)
    .append(2)
    .map { $0 }

// Q. What is `type(of: publisher)` ?
```

```
let publisher = Just<Int>(1)
    .append(2)
    .map { $0 }

// Q. What is `type(of: publisher)` ?

/*
    Publishers.Sequence<[Int], Never>
 */
```

```swift
let publisher = Just<Int>(1)
    .append(2)
    .map { $0 }
    .map { "\($0)"}
    .compactMap(Int.init)

// Q. What is `type(of: publisher)` ?
```

```swift
let publisher = Just<Int>(1)
    .append(2)
    .map { $0 }
    .map { "\($0)"}
    .compactMap(Int.init)

// Q. What is `type(of: publisher)` ?

/*
    Publishers.Sequence<[Int], Never>
 */
```

Rx Operator Fusion

Publisher.map

```
extension Publisher {
    /// Default `map` (wraps to `Map<...>`).
    func map<T>(_ transform: @escaping (Output) -> T)
        -> Publishers.Map<Self, T>
    {
        return Publishers.Map(
            upstream: self,
            transform: transform
        )
    }
}
```

Publishers.Map.map

```
extension Publishers.Map {
    /// Overloaded `map` that optimizes 2 consecutive `map`s
    /// into a single `Map` (no wrap e.g. `Map<Map<...>>`).
    func map<T>(_ transform: @escaping (Output) -> T)
        -> Publishers.Map<Upstream, T>
    {
        return Publishers.Map(upstream: upstream) {
            // Transform composition 🎉
            transform(self.transform($0))
        }
    }
}
```

Publishers.Sequence.map

```
extension Publishers.Sequence {
    /// Another overloaded `map` that optimizes
    /// by not even wrapping with a single `Map` at all.
    /// (This is a `Sequence` to `Sequence` mapping function!)
    func map<T>(_ transform: (Elements.Element) -> T)
        -> Publishers.Sequence<[T], Failure>
    {
        return Publishers.Sequence(
            sequence: sequence.map(transform)
        )
    }
}
```

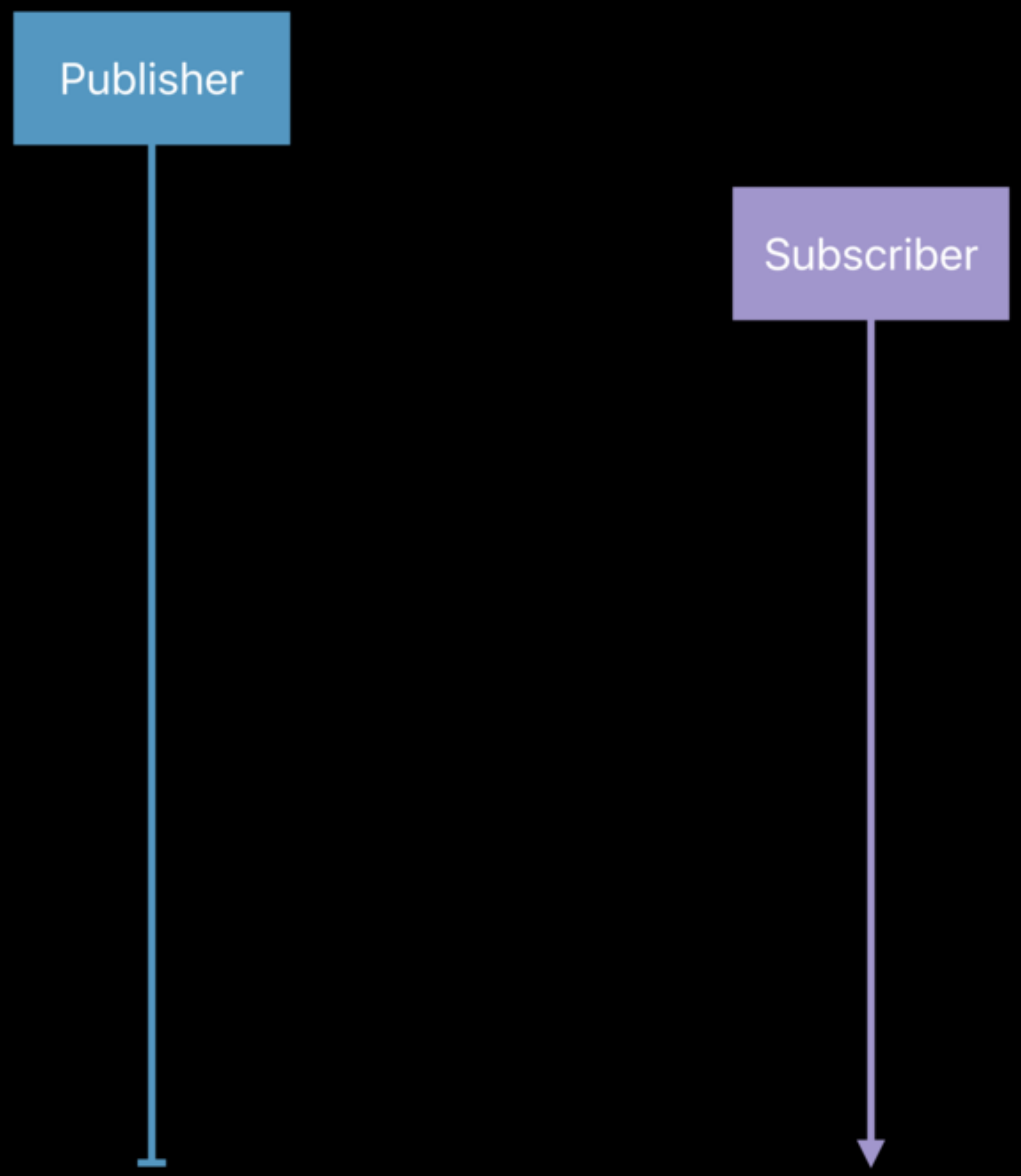
Rx Operator Fusion

Many Sequence-like methods are imported as Rx operators with **overloads for pipeline optimization at compile time** 🎉

- `map` / `compactMap`
- `filter` / `drop` / `dropFirst` / `prefix`
- `reduce` / `scan`
- `append` / `prepend`
- `removeDuplicates`, etc

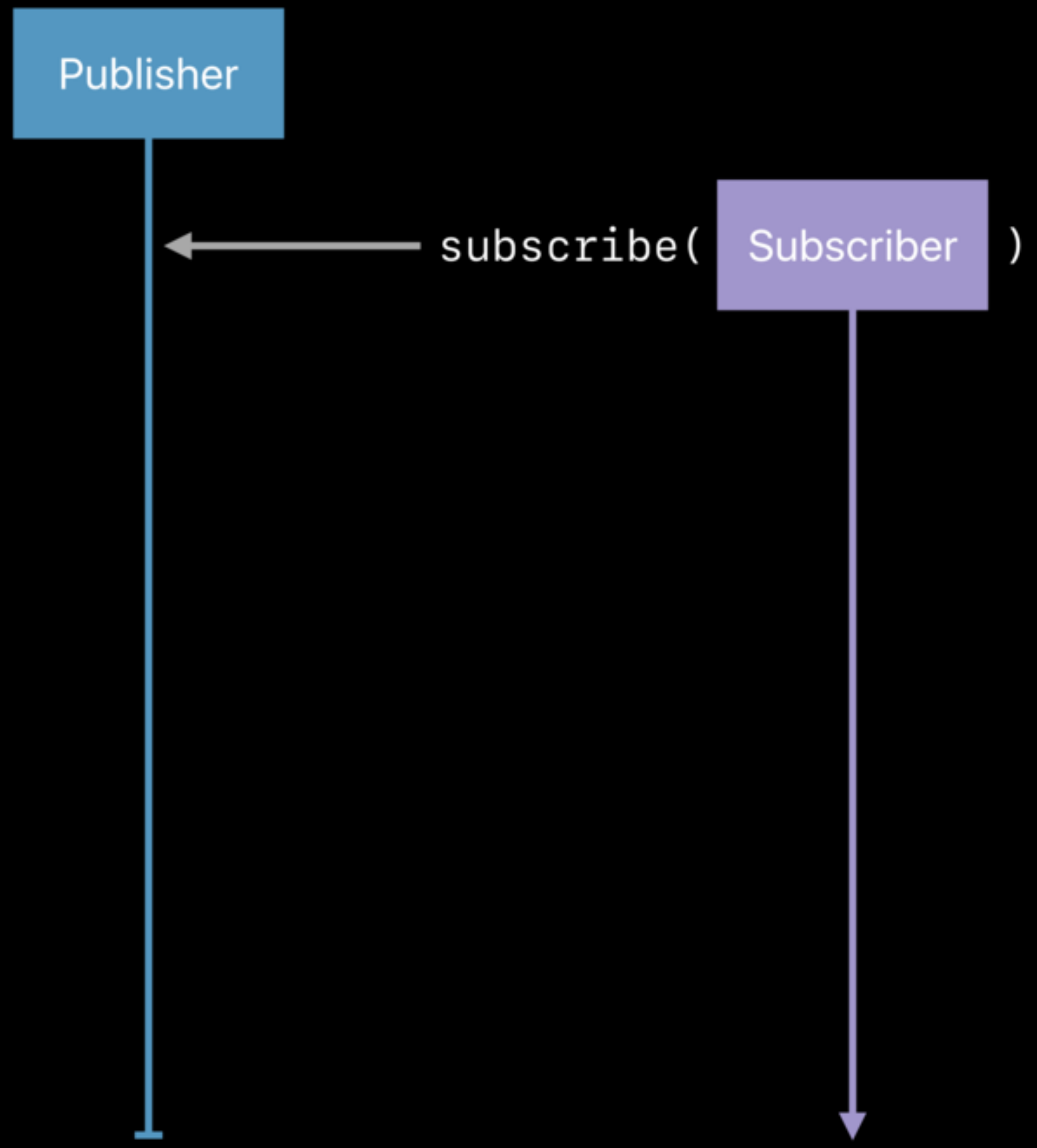
Non-blocking Backpressure

The Pattern



The Pattern

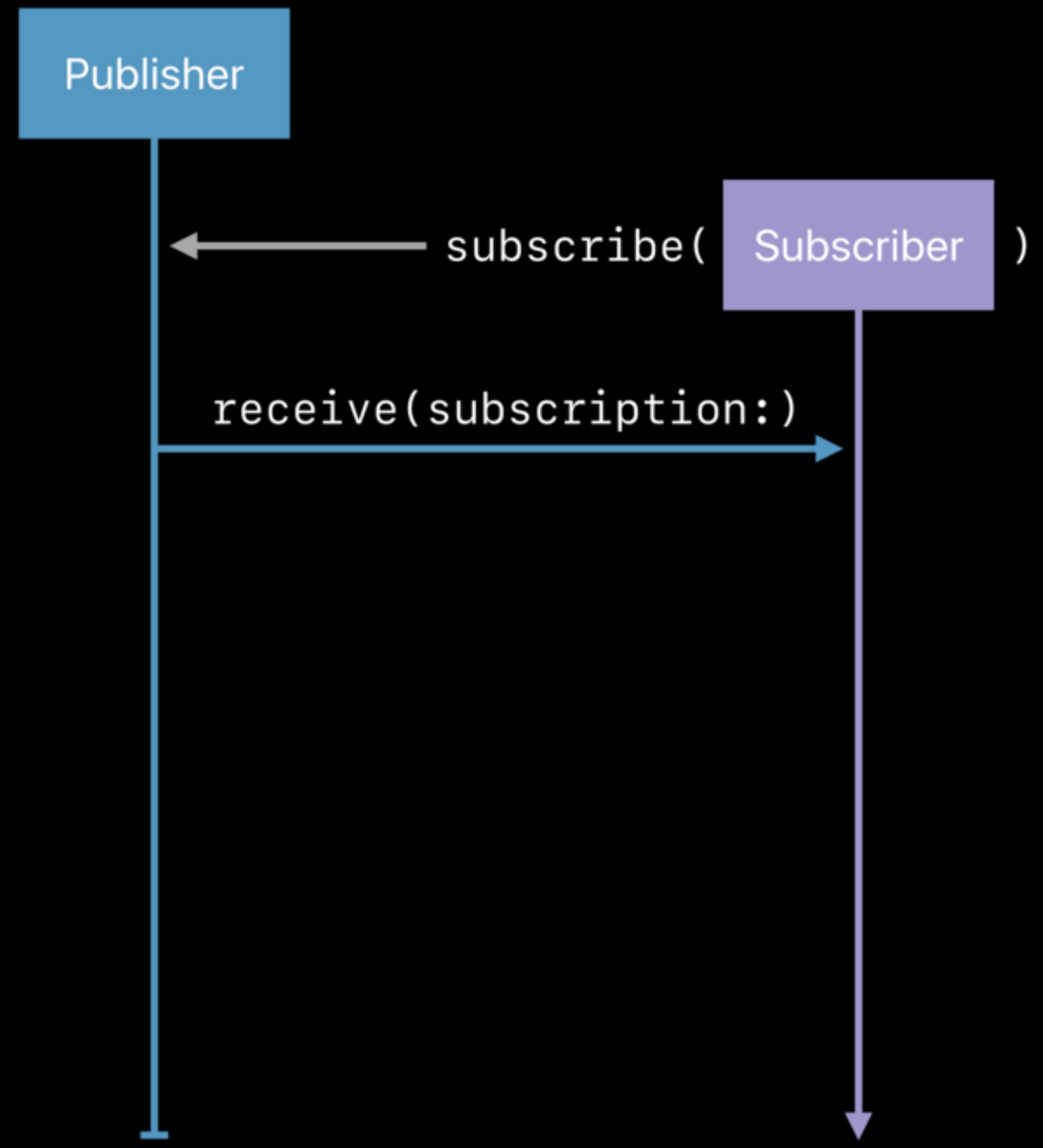
`Subscriber` is attached to `Publisher`



The Pattern

`Subscriber` is attached to `Publisher`

`Publisher` sends a `Subscription`

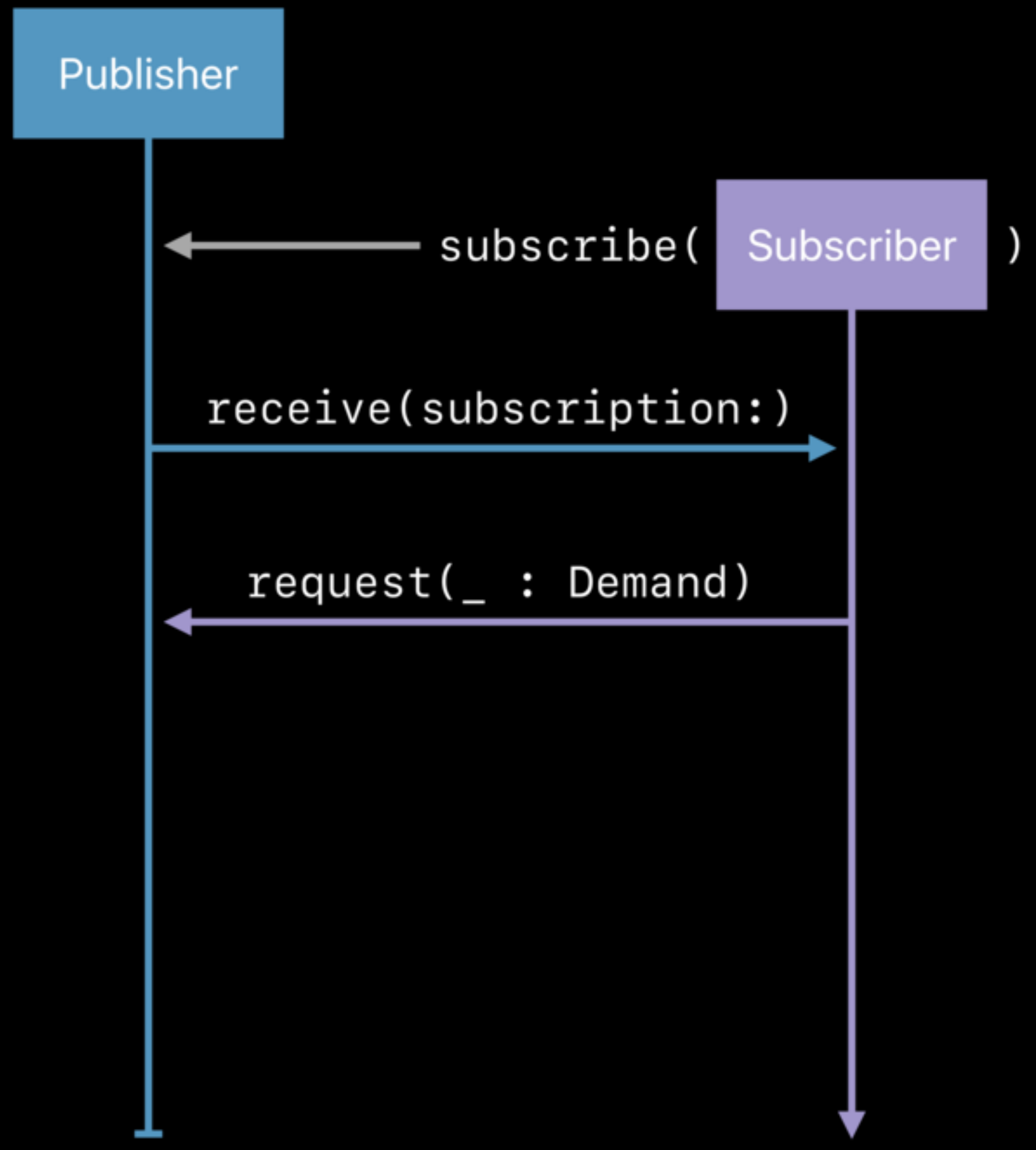


The Pattern

`Subscriber` is attached to `Publisher`

`Publisher` sends a `Subscription`

`Subscriber` requests *N* values



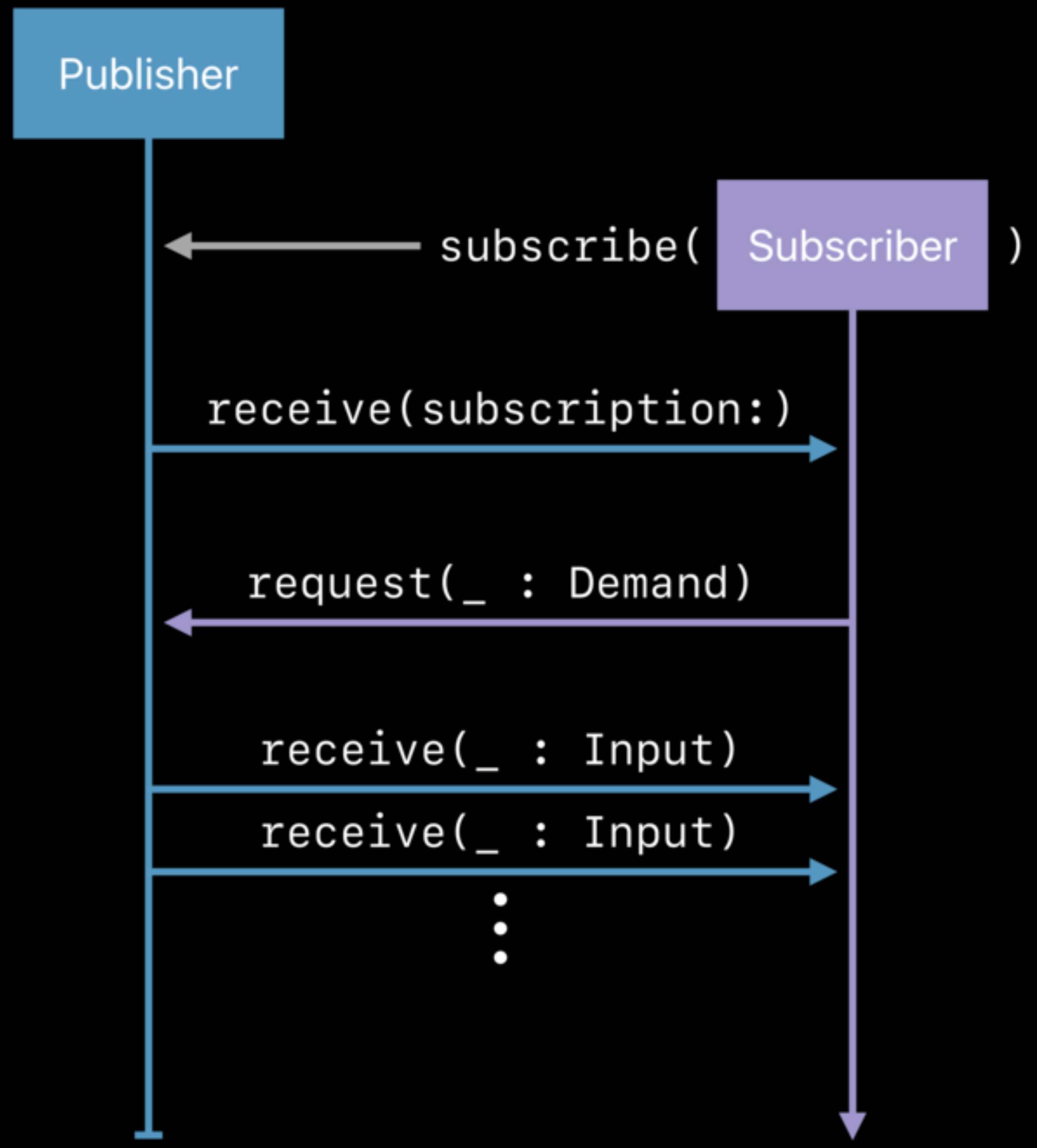
The Pattern

`Subscriber` is attached to `Publisher`

`Publisher` sends a `Subscription`

`Subscriber` requests *N* values

`Publisher` sends *N* values or less



The Pattern

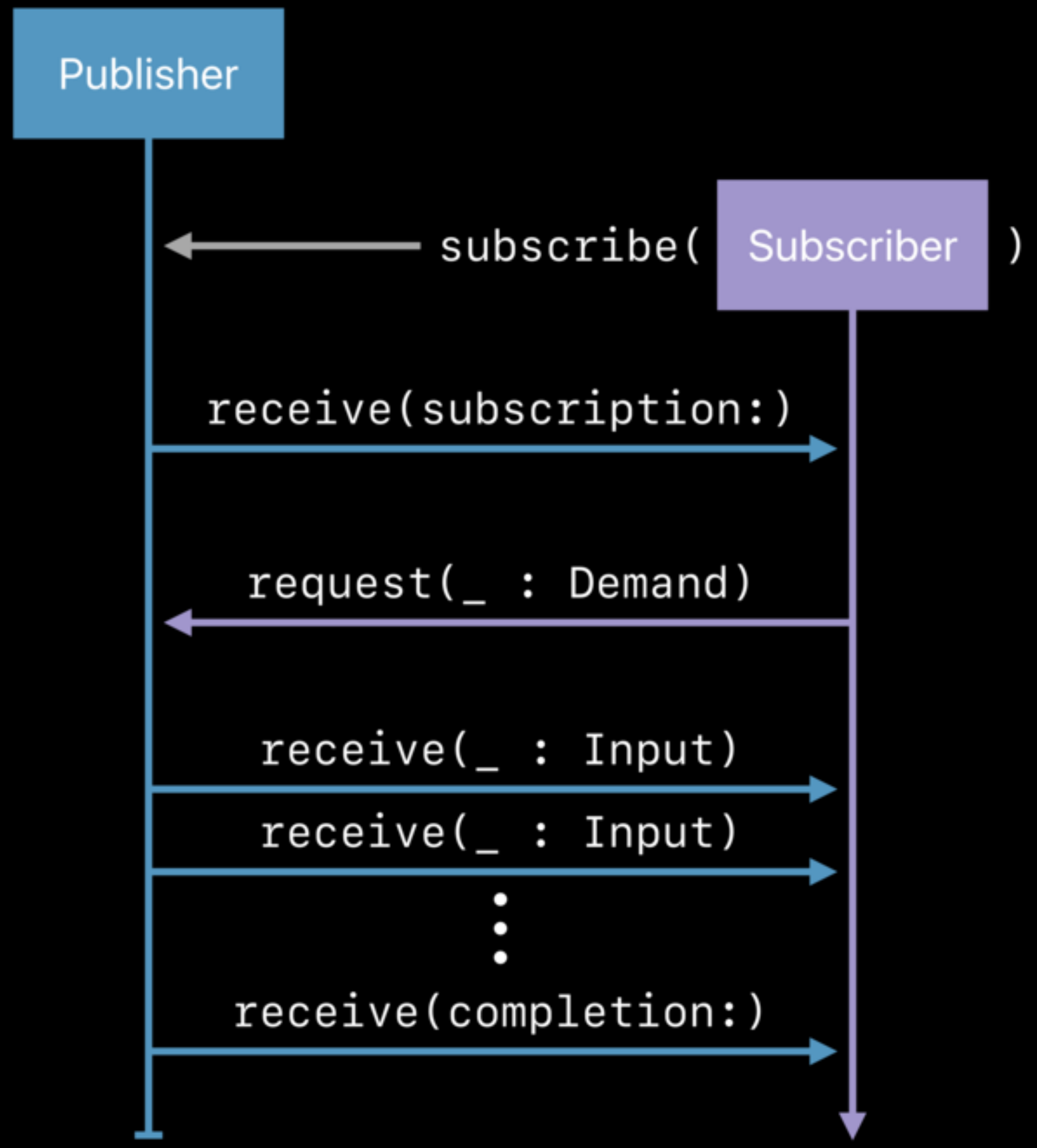
`Subscriber` is attached to `Publisher`

`Publisher` sends a `Subscription`

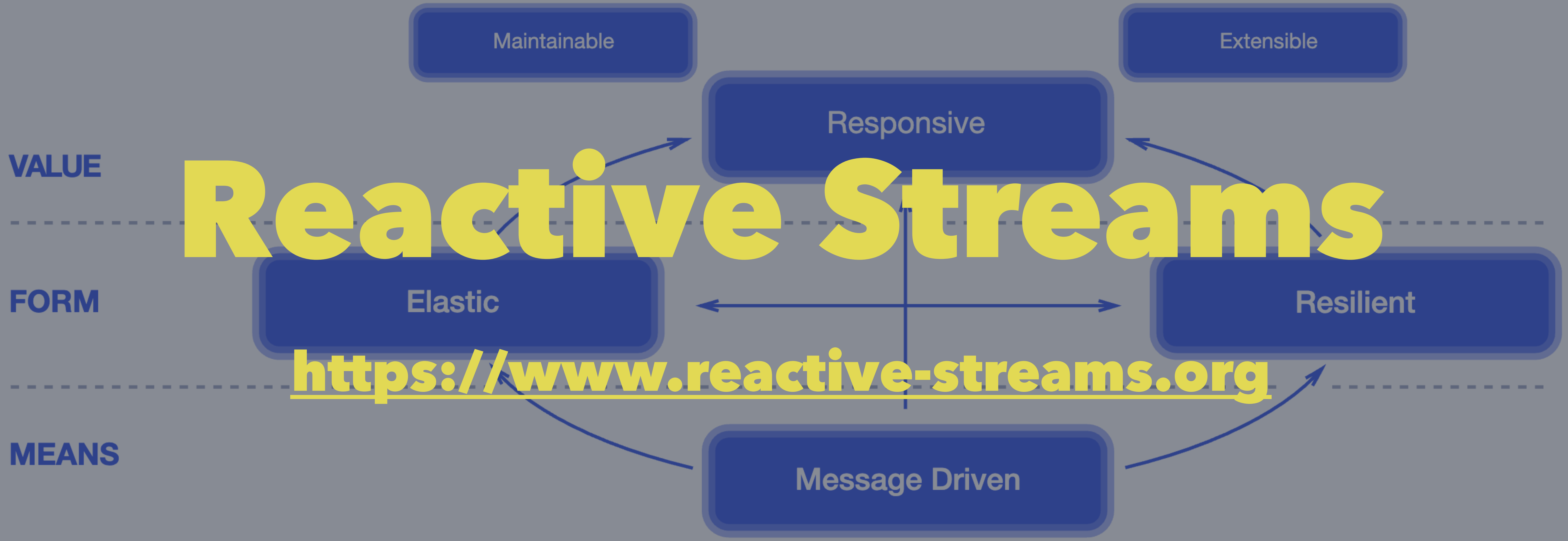
`Subscriber` requests *N* values

`Publisher` sends *N* values or less

`Publisher` sends completion



Reactive Streams



https://www.reactive-streams.org

Reactive Streams

1. Asynchronous stream processing (RxSwift, ReactiveSwift)
2. **Non-blocking back pressure** (New!)
    - **Slow `Subscriber` can request values from fast `Publisher`** at its own pace manually (Interactive Pull)

- Initiative found since 2013
- Implemented in RxJava 2 Flowable, Akka Streams, etc
- Interface is supported in Java 9 Flow API

```java
final class Flow { // Java 9 Flow API / reactive-streams-jvm
    static interface Publisher<T> {
        void subscribe(Subscriber<? super T> subscriber);
    }

    static interface Subscriber<T> {
        void onSubscribe(Subscription subscription);
        void onNext(T item);
        void onError(Throwable throwable);
        void onComplete();
    }

    static interface Subscription {
        void request(long n);
        void cancel();
    }

    static interface Processor<T,R> extends Subscriber<T>, Publisher<R> {}
}
```

```java
final class Flow { // Java 9 Flow API / reactive-streams-jvm
    static interface Publisher<T> {
        void subscribe(Subscriber<? super T> subscriber);
    }

    static interface Subscriber<T> {
        void onSubscribe(Subscription subscription);
        void onNext(T item);
        void onError(Throwable throwable);
        void onComplete();
    }

    static interface Subscription {
        void request(long n);
        void cancel();
    }

    static interface Processor<T,R> extends Subscriber<T>, Publisher<R> {}
}
```

```swift
protocol Publisher { // Swift Combine
    associatedtype Output
    associatedtype Failure : Error

    func receive<S>(subscriber: S)
      where S : Subscriber,
            Self.Failure == S.Failure, Self.Output == S.Input
}

protocol Subscriber : CustomCombineIdentifierConvertible {
    associatedtype Input
    associatedtype Failure : Error

    func receive(subscription: Subscription)
    func receive(_ input: Self.Input) -> Subscribers.Demand
    func receive(completion: Subscribers.Completion<Self.Failure>)
}
```

```
protocol Subscription : Cancellable, ... {
    func request(_ demand: Subscribers.Demand)
    // + func cancel()
}

extension Subscribers {
    struct Demand : Equatable, Comparable, Hashable, ... {
        static var unlimited: Subscribers.Demand { get }
        static func max(_ value: Int) -> Subscribers.Demand
    }
}

protocol Subject : AnyObject, Publisher {
    func send(subscription: Subscription)
    func send(_ value: Self.Output)
    func send(completion: Subscribers.Completion<Self.Failure>)
}
```

Java Flow(able) V.S. Swift Combine

- Mostly identical APIs
    - Generic interface V.S. Protocol associatedtype
    - Combine has more type-safe interfaces (e.g. `Demand`)
    - Combine does not rely on subclassing (vtable)
- Combine only supports backpressure-able types
    - More difficult for 3rd party to implement new Rx operators with backpressure support

Subscriber `request` Example

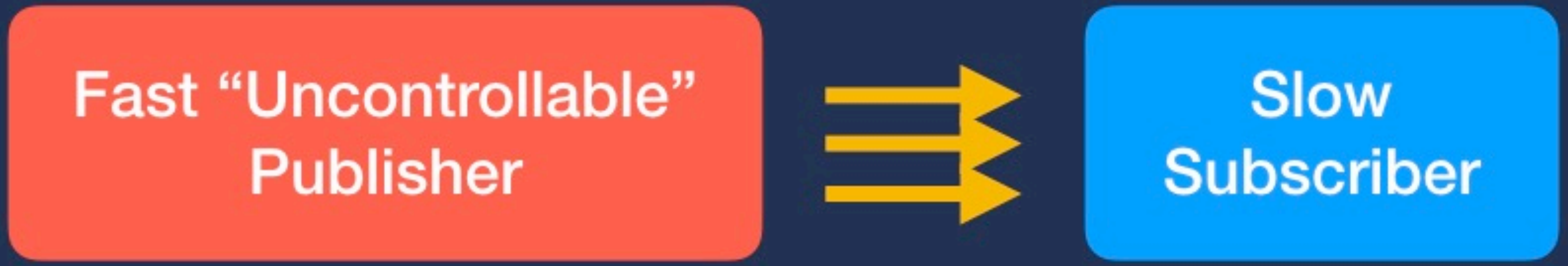
```
class MySubscriber: Subscriber { // Custom Subscriber example
    var subscription: Subscription? // subscriber retains subscription

    func receive(subscription: Subscription) {
        self.subscription = subscription
        subscription.request(.max(1)) // request 1 value
    }

    func receive(_ input: Int) -> Subscribers.Demand {
        runAsyncSideEffect(input: input, completion: { [weak self] in
            self?.subscription?.request(.max(1)) // asynchronous
        })

        runSyncSideEffect(input: input)
        return .max(1) // Combine supports synchronous returning demand
    }
}
```

```
class MySubscriber: Subscriber { // Custom Subscriber example
    var subscription: Subscription? // subscriber retains subscription

    func receive(subscription: Subscription) {
        self.subscription = subscription
        subscription.request(.max(1)) // request 1 value
    }

    func receive(_ input: Int) -> Subscribers.Demand {
        runAsyncSideEffect(input: input, completion: { [weak self] in
            self?.subscription?.request(.max(1)) // asynchronous
        })

        runSyncSideEffect(input: input)
        return .max(1) // Combine supports synchronous returning demand
    }
}
```

```swift
class MySubscriber: Subscriber { // Custom Subscriber example
    var subscription: Subscription? // subscriber retains subscription

    func receive(subscription: Subscription) {
        self.subscription = subscription
        subscription.request(.max(1)) // request 1 value
    }

    func receive(_ input: Int) -> Subscribers.Demand {
        runAsyncSideEffect(input: input, completion: { [weak self] in
            self?.subscription?.request(.max(1)) // asynchronous
        })

        runSyncSideEffect(input: input)
        return .max(1) // Combine supports synchronous returning demand
    }
}
```

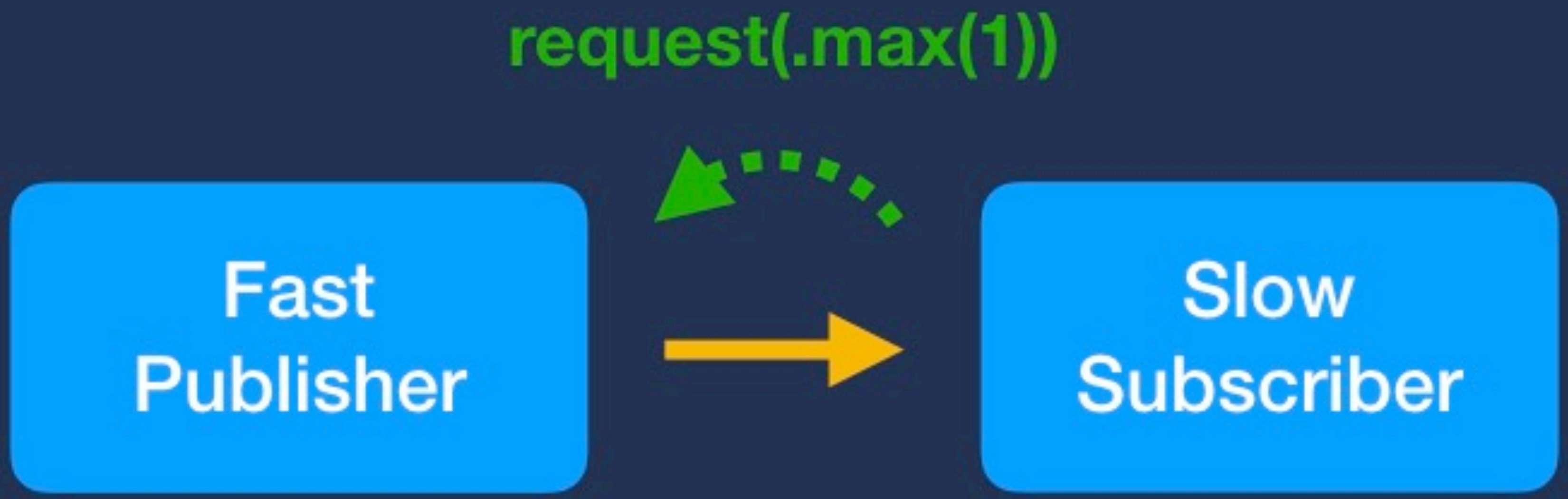
```swift
class MySubscriber: Subscriber { // Custom Subscriber example
    var subscription: Subscription? // subscriber retains subscription

    func receive(subscription: Subscription) {
        self.subscription = subscription
        subscription.request(.max(1)) // request 1 value
    }

    func receive(_ input: Int) -> Subscribers.Demand {
        runAsyncSideEffect(input: input, completion: { [weak self] in
            self?.subscription?.request(.max(1)) // asynchronous
        })

        runSyncSideEffect(input: input)
        return .max(1) // Combine supports synchronous returning demand
    }
}
```

Backpressure Strategies



Fast “Uncontrollable” Publisher

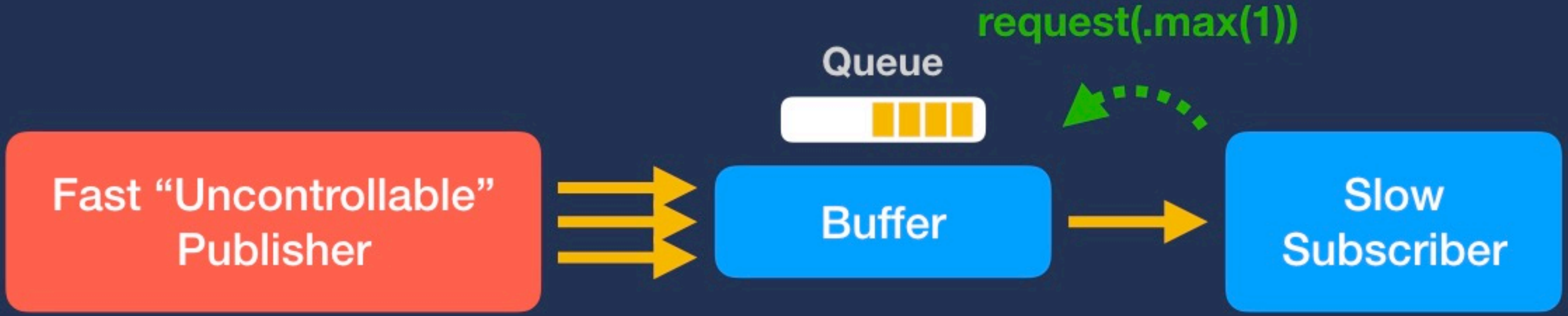
Slow Subscriber



request(.max(1))

Fast Publisher

Slow Subscriber



request(.max(1))
Queue
Fast “Uncontrollable” Publisher
Buffer
Slow Subscriber

Backpressure Strategies

1. **Callstack blocking** on the same thread (❌ not preferred)
2. **Interactive Pull**
   Topmost cold upstream **listens to downstream's `request`** and iterates the emission manually
3. **Bounded Buffer & Queue-Drain**
   Intermediate stream holds **internal finite-size buffer** to enqueue and pull values (Queue-Drain) for asynchronous boundaries

Non-Interactive Pull (Push only)

`Publishers.Sequence` **NOT listening to** `request`

```
struct Sequence<Elements, Failure> : Publisher
  where Elements : Sequence, Failure : Error {
    ...
    func receive<S: Subscriber>(subscriber: S) where … {

        for value in sequence where !isCancelled {
            subscriber.receive(value) // push inside the loop
        }
        subscriber.receive(completion: .finished)
    }
}
```

Imagine...

```
let infiniteIssues
    = (1...).lazy.map(Issue.init(id:))

let me = SlowSubscriber(...) // Can work 1 issue per day

Publishers.Sequence(infiniteIssues)
    .subscribe(me) // Goodbye, cruel world 😇
```

Immediate infinite tasks will ~~kill me~~ block the thread.

Imagine…

```
Publishers.Sequence(infiniteIssues)
     .delay(for: day, scheduler: DispatchQueue.main)
     .subscribe(me) // Yay, schedule is delayed 🙌
```

Asynchronizing (e.g. `Delay`, `ReceiveOn`) tasks will cause `DispatchQueue` (unbounded async boundary) to be exhausted.

Imagine…

```
Publishers.Sequence(infiniteIssues)
    .debounce(for: day, scheduler: DispatchQueue.main)
    .subscribe(me) // Let's throw away some tasks 🚯
```

Debounce / `Throttle` will discard some tasks which may not be a desirable solution.

Interactive Pull

Publishers.Sequence **listening to** request

```
struct Sequence<Elements, Failure> : Publisher
  where Elements : Sequence, Failure : Error {
    ...
    func receive<S: Subscriber>(subscriber: S) where ··· {
        let innerSubscription = InnerSubscription(
            sequence: sequence,
            downstream: subscriber
        )
        subscriber.receive(subscription: innerSubscription)
    }
}
```

```
private final class InnerSubscription<...> : Subscription, ⋯ { // Pseudocode
    var iterator: Iterator
    @Atomic var remaining: Demand = .none
    ...
    func request(_ demand: Subscribers.Demand) {
        guard $remaining.modify { $0 += demand } == .none else {
            return // no-reentrant
        }
        while remaining > 0 {
            if let nextValue = iterator.next() { // interactive pull
                remaining += downstream.receive(nextValue) - 1
            } else {
                _downstream?.receive(completion: .finished)
                cancel()
            }
        }
    }
}
```

Bounded Buffer & Queue-Drain

- **Batch:** `Buffer`, `CollectByCount`, `CollectByTime`
- **Async:** `ReceiveOn`, `Delay`
- **Combining:** `FlatMap`, `Merge`, `CombineLatest`, `Zip`, `Concatenate`, `SwitchToLatest`
- **Multicast:** `MakeConnectable` / `Multicast` / `Autoconnect`

(Note: Many Combine's operators are still unbound yet)

```swift
// For `Buffer`.
enum PrefetchStrategy {
    case keepFull
    case byRequest
}

// For `Buffer`.
enum BufferingStrategy<Failure> where Failure : Error {
    case dropNewest
    case dropOldest
}

// For `CollectByTime`.
enum TimeGroupingStrategy<Context> where Context : Scheduler {
    case byTime(Context, Context.SchedulerTimeType.Stride)
    case byTimeOrCount(Context, Context.SchedulerTimeType.Stride, Int)
}
```

flatMap using Queue-Drain

```
extension Publisher {
    func flatMap<T, P>(
        maxPublishers: Subscribers.Demand = .unlimited,
        _ transform: @escaping (Self.Output) -> P
    ) -> Publishers.FlatMap<P, Self>
      where T == P.Output, P : Publisher,
            Self.Failure == P.Failure
}
```

(Almost) Same API as RxJava's

```
flatMap(mapper, maxConcurrency, bufferSize)
```

```
// Queue-Drain pseudocode, inspired from RxJava
struct FlatMap<NewPublisher, Upstream> : Publisher where … {
    let upstream: Upstream
    let maxPublishers: Subscribers.Demand
    let transform: (Upstream.Output) -> NewPublisher

    func receive<S: Subscriber>(subscriber: S) where … {
        let mergeSubscriber = MergeSubscriber(
            upstream: upstream,
            maxPublishers: maxPublishers,
            transform: transform,
            downstream: subscriber
        )
        upstream.subscribe(mergeSubscriber)
    }
}
```

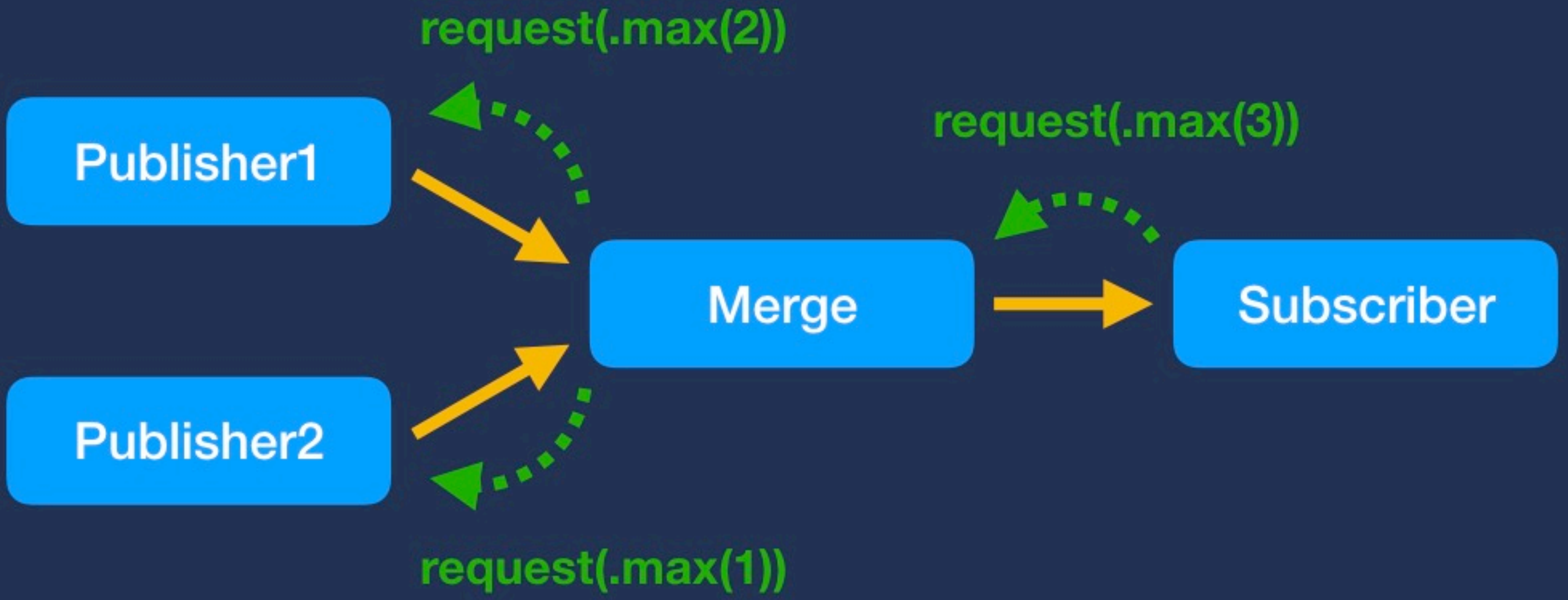
```
private final class MergeSubscriber<...> : Subscriber, Subscription, … {
    @Atomic var remaining: Demand = .none
    @Atomic var drainCount: Int = 0
    @Atomic var queue: Queue<Output> = []
    @Atomic var innerSubscribers: [InnerSubscriber] = []

    func receive(subscription: Subscription) {
        self.subscription = subscription
        downstream.receive(subscription: self)
        subscription.request(maxPublishers)
    }

    func receive(_ input: Upstream.Output) -> Subscribers.Demand {
        queue.append(input) // enqueue value
        let innerSubscriber = InnerSubscriber(parent: self)
        innerSubscribers.append(innerSubscriber)
        transform(input).subscribe(innerSubscriber)
    }
}
```

```swift
private final class InnerSubscriber: Subscriber {
    let parent: MergeSubscriber<...>
    var subscription: Subscription?

    func receive(subscription: Subscription) {
        self.subscription = subscription
        parent.drainLoop()
    }

    func receive(_ input: Upstream.Output) -> Subscribers.Demand {
        parent.drainLoop()
    }
}
```

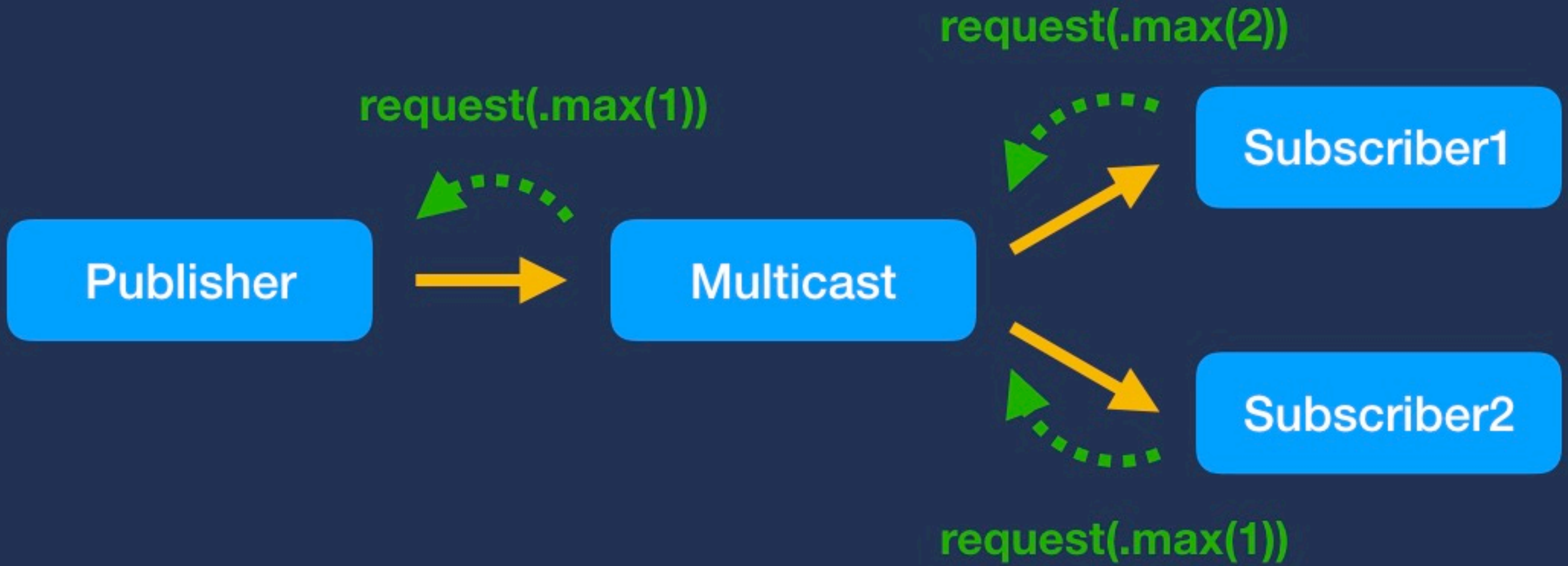
```
extension MergeSubscriber {
    func drainLoop(subscription: Subscription) {
        guard $drainCount.modify { $0 + 1 } == 0 else { return }
        while true {
            var replenishCount = 0
            while true {
                var emittedCount = 0
                while remaining > .none {
                    let value = queue.pop()   // dequeue value
                    downstream.receive(value) // send
                    replenishCount += 1
                    emittedCount += 1
                    remaining -= 1
                }
                remaining -= emittedCount
            }
            for inner in innerSubscribers { /* loop for inner queues polling */ }
            if replenishCount != 0 && !isCancelled {
                subscription.request(replenishCount)
    }}}
```

Queue-Drain `request` handlings for multiple `Publishers`

Splitted `request` for combined publisher



Minimum `request` for broadcasting



Recap

- **Rx Operator Fusion**
  - Clever technique to optimize stream pipeline at compile time with the help of Swift type system
- **Backpressure**
  - A mechanism for slow subscriber to talk to fast publisher
  - Conforms to Reactive Streams specification
  - Difficult to implement Queue-Drain model

References (Rx Operator Fusion)

- Why Combine has so many Publisher types | Thomas Visser
- Advanced Reactive Java: Operator-fusion (Part 1)
- Advanced Reactive Java: Operator fusion (part 2 - final)

References (Backpressure)

- https://www.reactive-streams.org
- Backpressure · ReactiveX/RxJava Wiki
- RxJava/Backpressure-(2.0).md
- RxJava/Implementing-custom-operators-(draft).md
- RxJava/Writing-operators-for-2.0.md at 3.x · ReactiveX/RxJava
- Reactive Systems と Back Pressure

Thanks!

Yasuhiro Inami
@inamiy