Programação Reativa com Java e Project Reactor

Sobre o curso

- Conceitos importantes sobre sistemas reativos, programação reativa, fluxo assíncrono e não bloqueante e reactive streams.
- Programação reativa como project reactor.
- Documentações oficiais.
- Exemplos práticos.

Pré requisitos

- Java 11
- Conhecimento Java 8 stream API

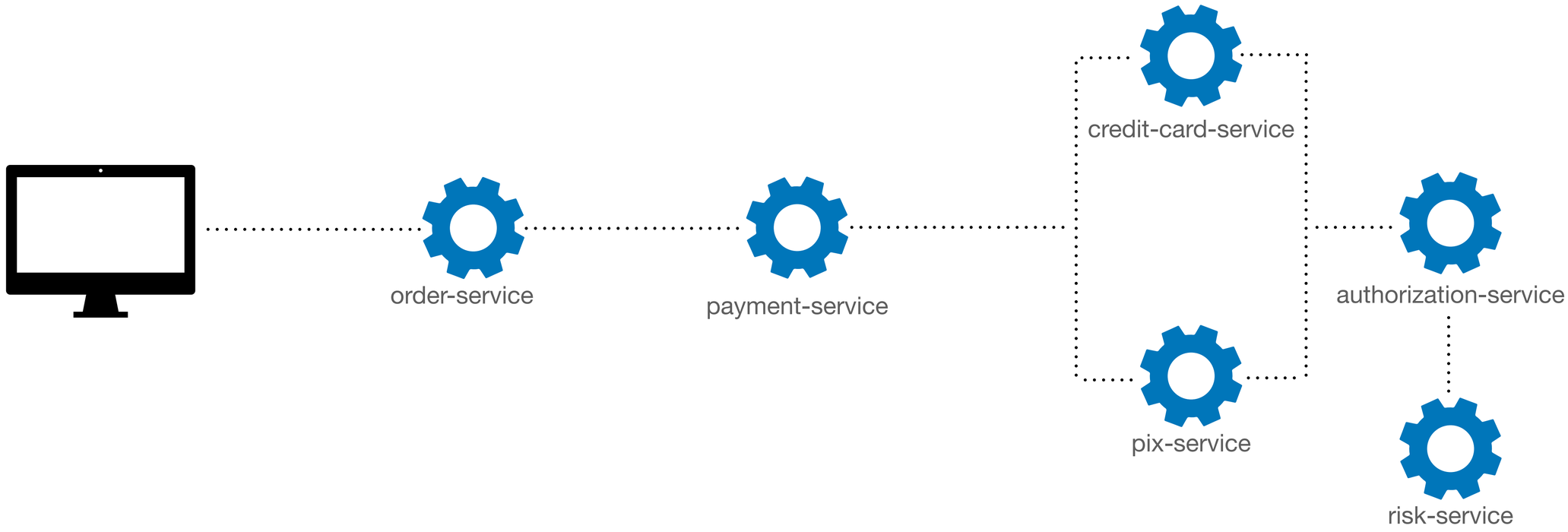
Módulo 2 - Conceitos importantes

Conceitos importantes

- Sistema reativo vs programação reativa
- Fluxo síncrono vs assíncrono vs não bloqueante

Sistema tradicional

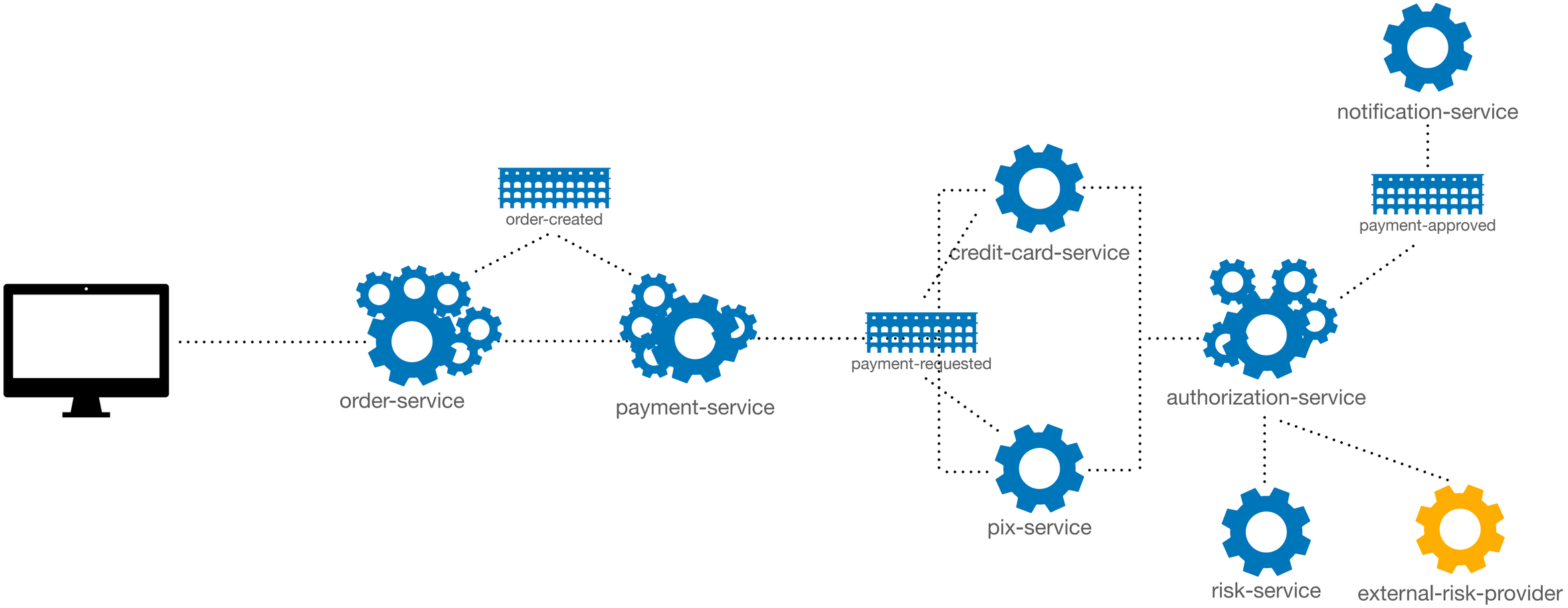
Sistema tradicional



Sistema reativo

É um sistema projetado para ser altamente responsivo, resiliente e elástico lidando de forma eficaz com solicitações e falhas, garantindo um desempenho consistente.

Sistema Reativo



Manifesto reativo

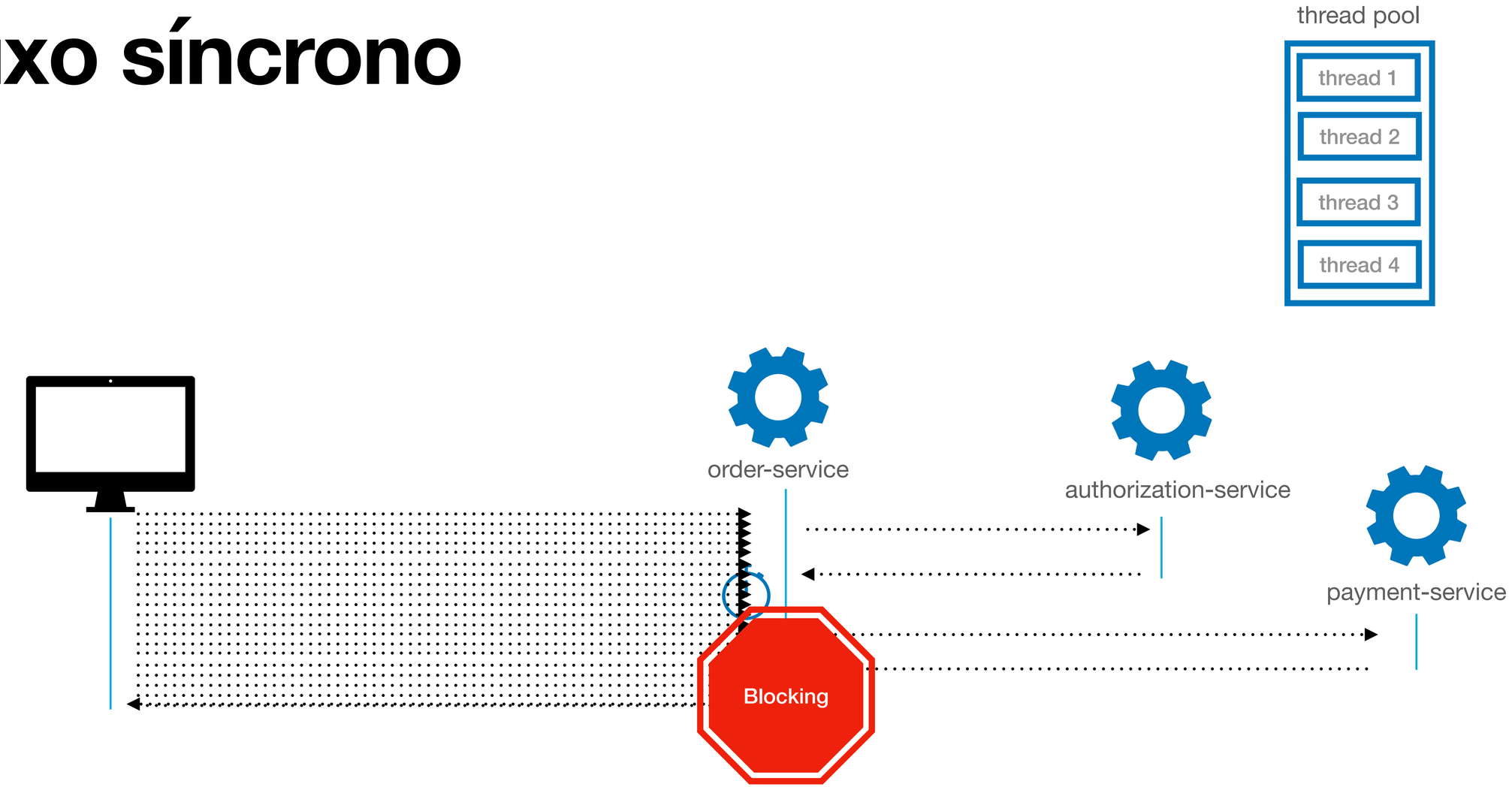
Responsivo	O sistema responde consistentemente, considerando flutuações na carga de trabalho ou condições de rede.
Elástico	O sistema deve ser capaz de se adaptar à demanda. Sistemas reativos geram métricas de desempenho que podem ser usadas para políticas de dimensionamento.
Resiliente	O sistema permanece responsivo em caso de falha. As falhas devem ser tratadas dentro de cada componente ou aplicação, sem comprometer a solução como um todo.
Orientado a mensagem	O sistema é construído usando fluxo de eventos assíncronos que não bloqueiam desnecessariamente qualquer operação.

https://www.reactivemanifesto.org/pt-BR

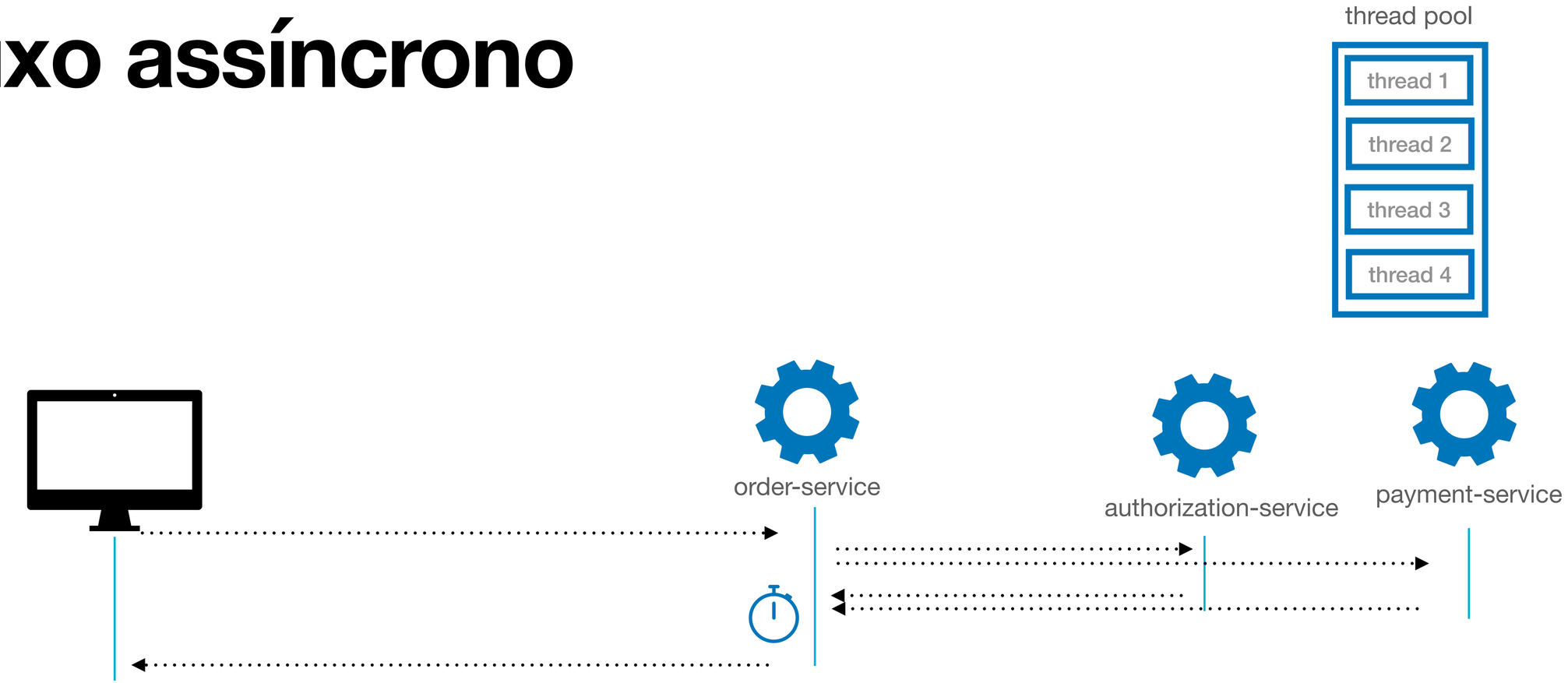
Programação reativa

- Foco em desenvolvimento assíncrono e não bloqueante;
- Programação funcional;
- Propagação de mudanças em um fluxo de dados (data streams);
- Controle de backpressure.

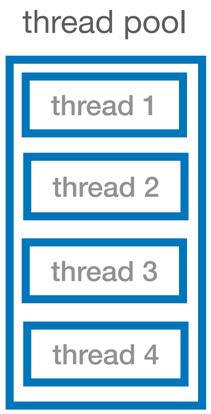
Fluxo síncrono

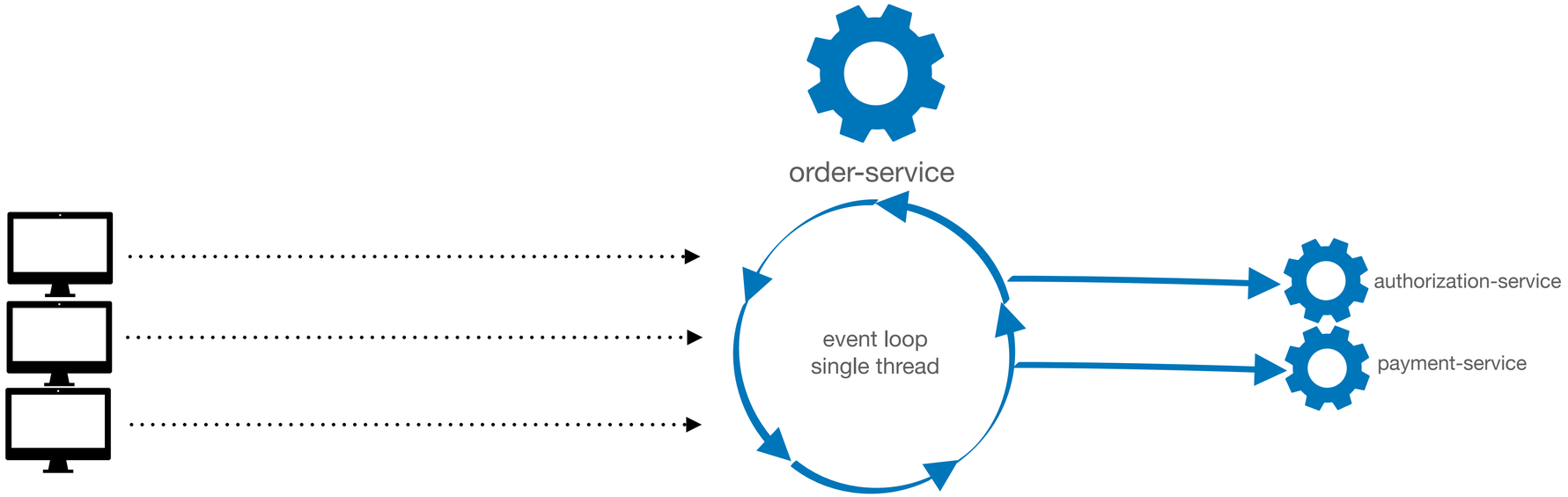


Fluxo assíncrono



Fluxo assíncrono e não bloqueante





Programação funcional

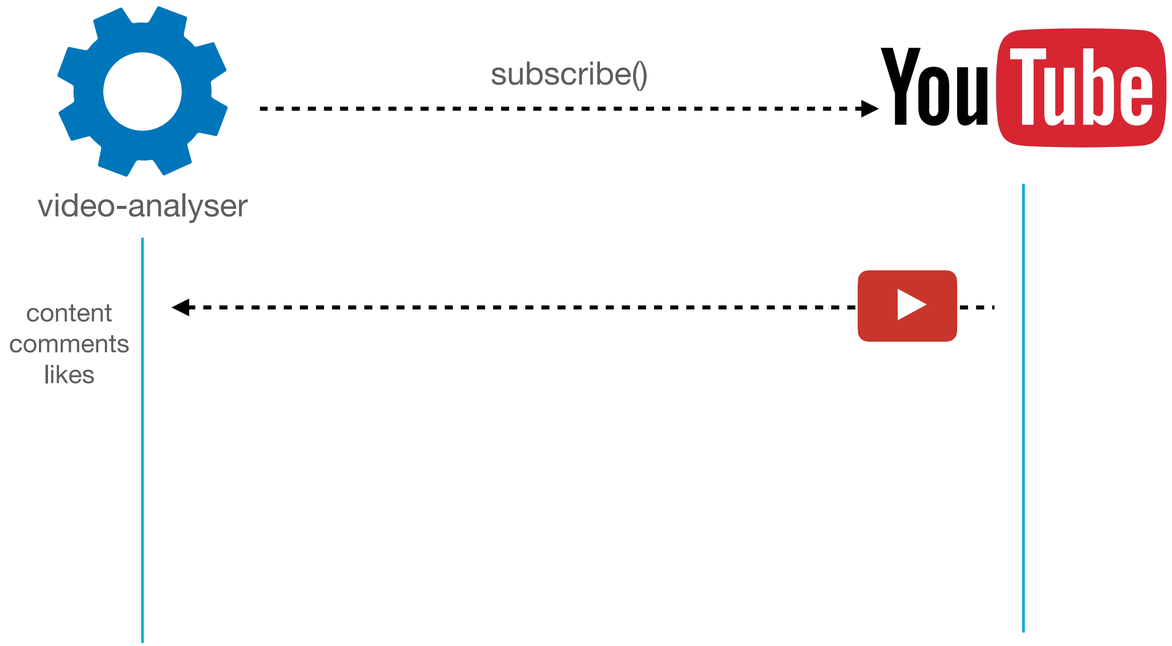
non-funcional

```
List<String> nomes = Arrays.asList("Alice", "Bob", "Charlie", "David");
String nomeEncontrado = null;
for (String nome : nomes) {
    if (nome.startsWith("C")) {
        nomeEncontrado = nome;
        break;
    }
}
if (nomeEncontrado != null) {
    System.out.println("Nome encontrado: " + nomeEncontrado);
}
```

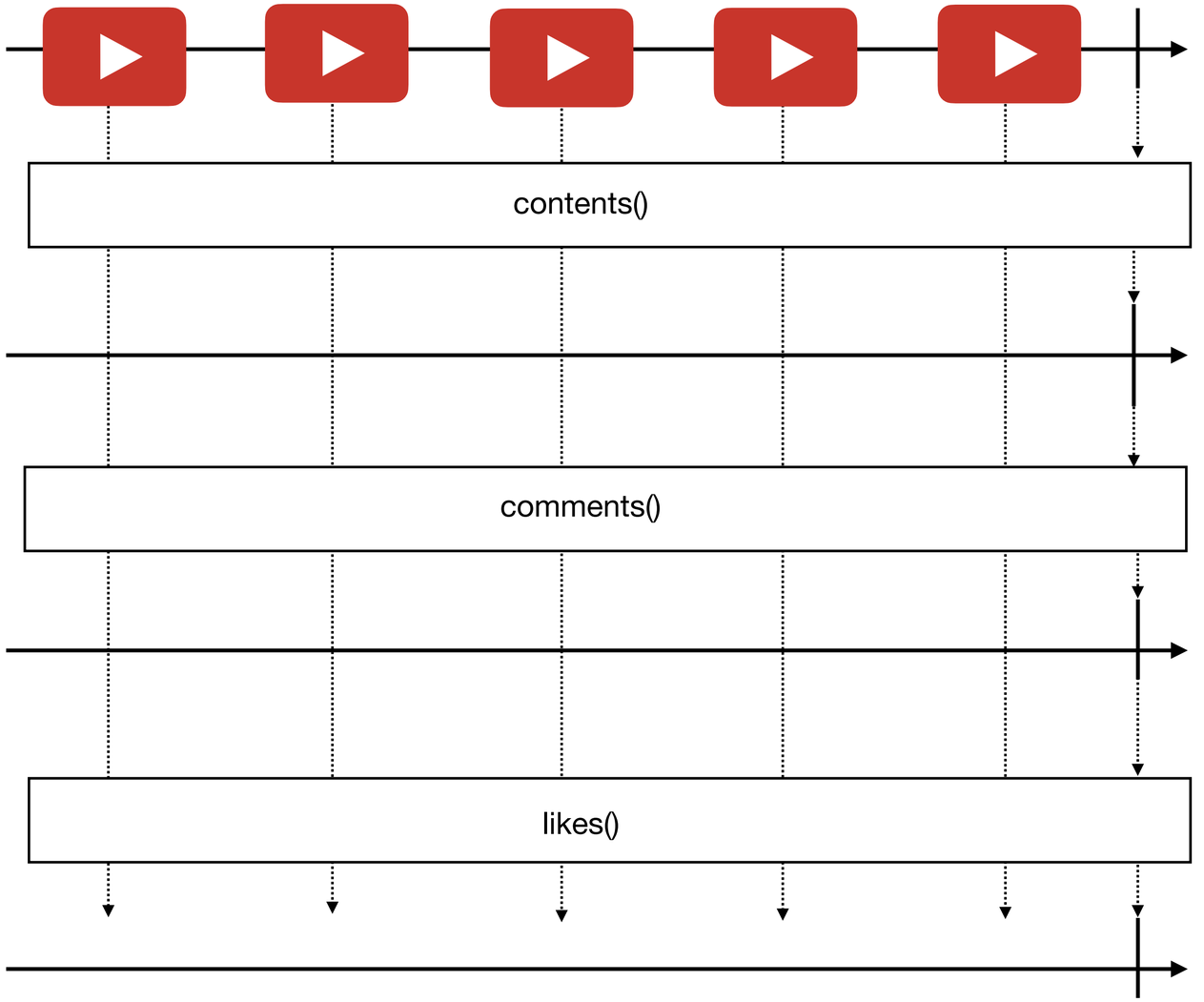
funcional

```
Stream.of("Alice", "Bob", "Charlie", "David")
        .filter(nome -> nome.startsWith("C"))
        .findFirst()
        .ifPresent(nome -> System.out.println("Nome encontrado: " + nome));
```

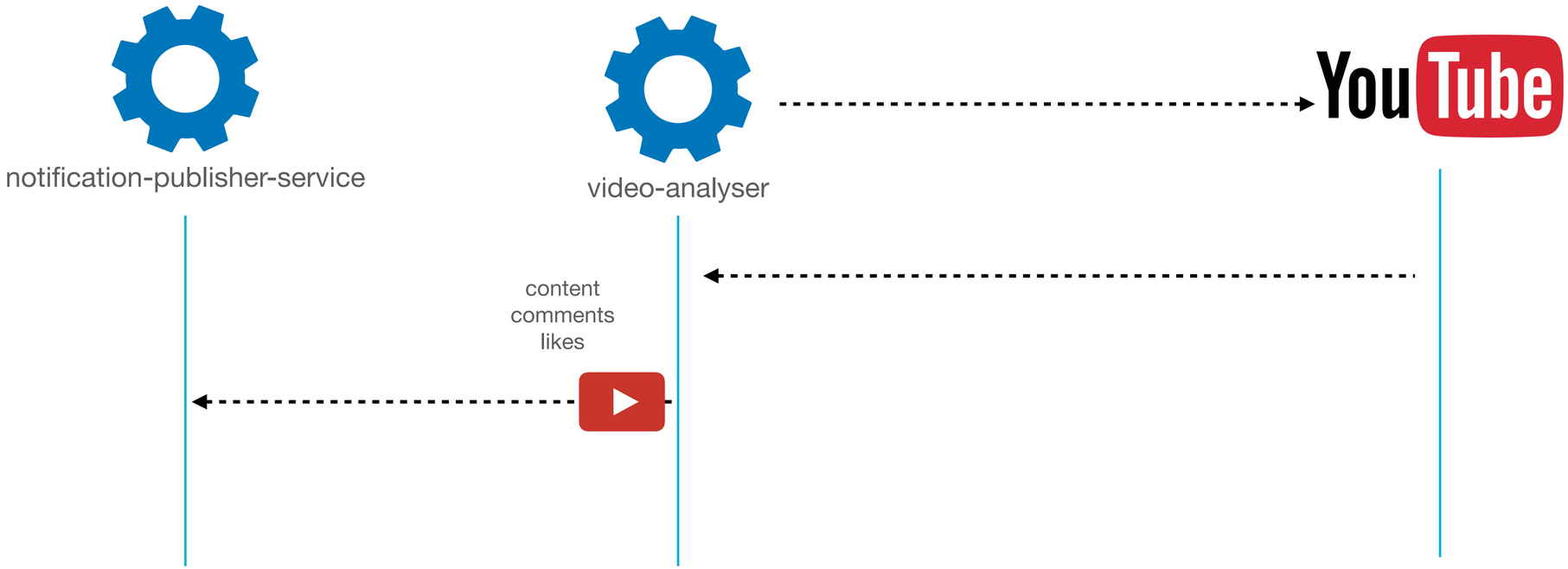
Propagação de mudanças data streams



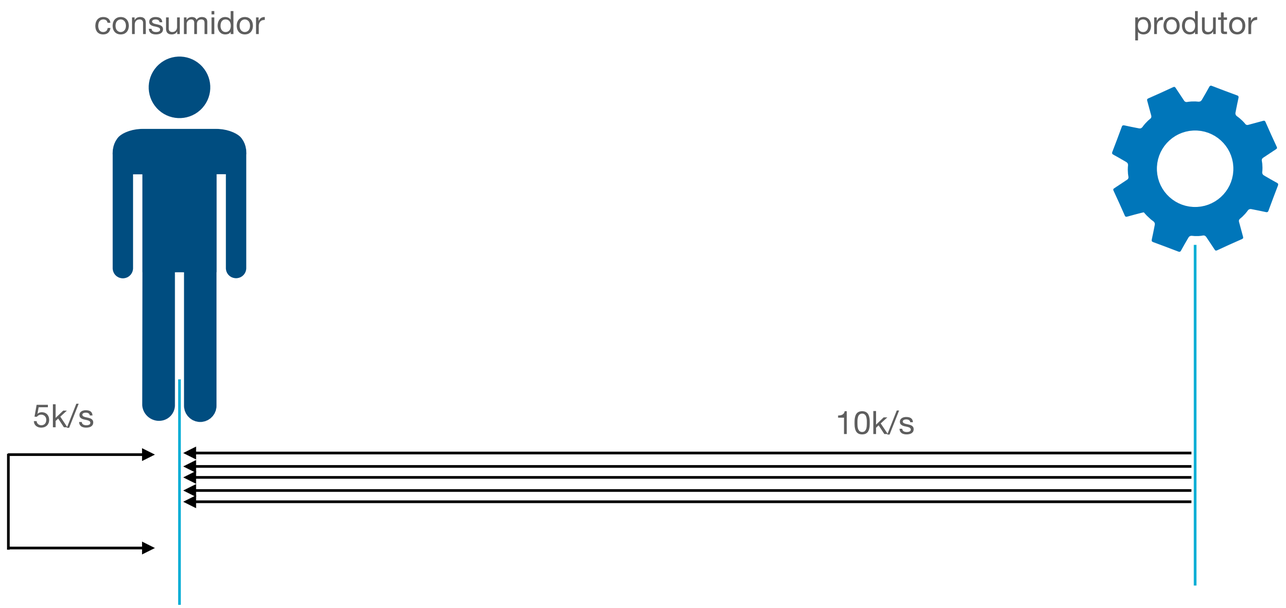
video-analyser internamente



Propagação para outros sistemas



Backpressure



Programação reativa

É uma abordagem no desenvolvimento de software que se concentra em lidar com a **propagação de mudanças** em **fluxos de dados (data streams)** de **forma assíncrona e não bloqueante com mecanismo de backpressure**, permitindo que sistemas **reajam automaticamente a essas mudanças.**

Resumo

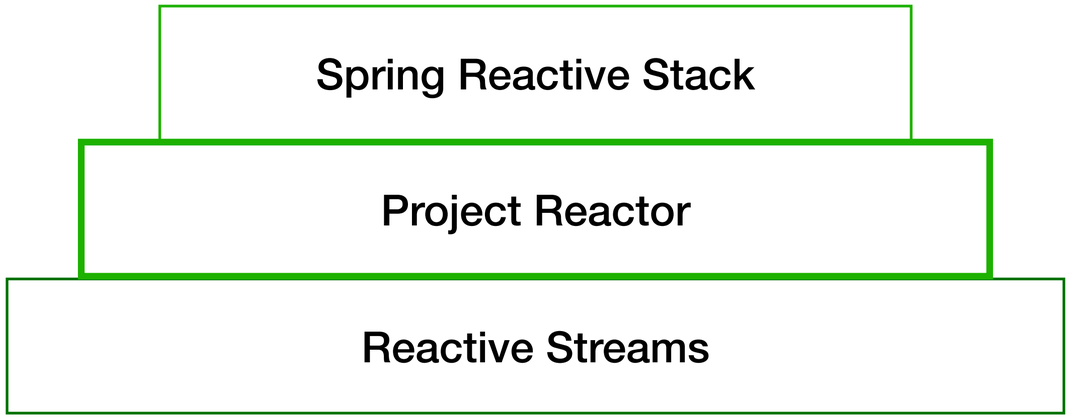
- **Sistema reativo:** Sistemas construídos de forma responsiva, resiliente, escaláveis e orientados a mensagens.
- **Programação reativa**: Paradigma de programação que reage a eventos de forma assíncrona e não bloqueante com backpressure.
- **Fluxo síncrono:** Execução sequencial de tarefas, aguardando a conclusão de cada uma antes de continuar para a próxima.
- **Fluxo assíncrono:** Execução simultânea de tarefas, permitindo paralelismo.
- **Não bloqueante**: Capacidade de um programa continuar a execução sem ser bloqueado por uma outra tarefa.
- **Backpressure**: Mecanismo de controle para lidar com a discrepância de velocidade entre a produção e consumo de dados.

Módulo 3 - Reactive streams

Stack reativa

https://spring.io/reactive/

https://projectreactor.io/



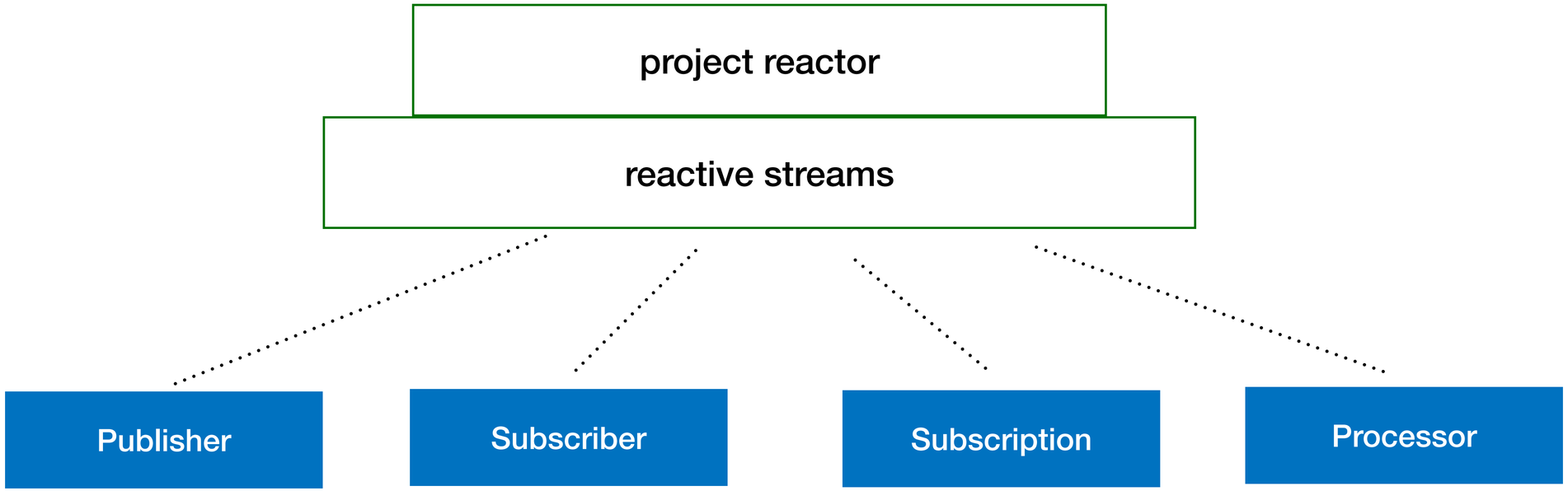
reactive streams

Reactive Streams é um padrão e especificação para **bibliotecas reativas** orientadas a **stream** para a JVM com as seguintes características:

- Processam um número potencialmente ilimitado de elementos em sequência;
- Passam elementos de forma **assíncrona** entre os componentes;
- Executado de forma **não bloqueante** com **backpressure.**

A biblioteca project reactor **segue as especificações do reactive streams.**

Reactive streams - Componentes



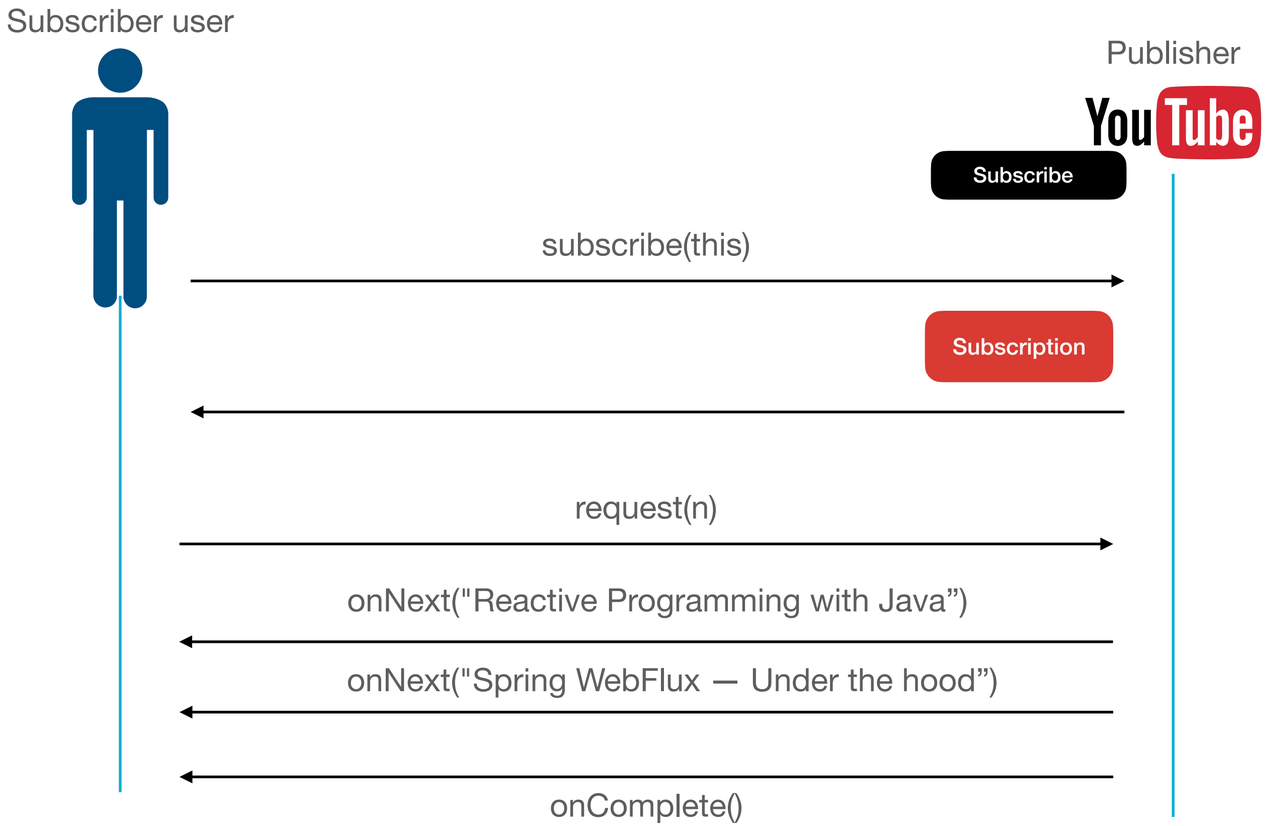
https://github.com/reactive-streams/reactive-streams-jvm

Reactive streams - Componentes

- **Publisher:** Representa a fonte de dados que fornece informações a serem consumidas
- **Subscriber**: Representa o consumidor de dados que recebe e reage às informações fornecidas pelo Publisher
- **Subscription:** Age como intermediário, permitindo que os Subscribers controlem o fluxo de dados recebidos dos Publishers
- **Processor:** Atua como um componente flexível que pode funcionar como um Subscriber ou Publisher

onSubscribe(subscription)

Reactive streams




```java
public interface Publisher<T> {

    public void subscribe(Subscriber<? super T> s);
}
```

```java
public interface Subscriber<T> {

    public void onSubscribe(Subscription s);
    public void onNext(T t);
    public void onError(Throwable t);
    public void onComplete();
}
```

```java
public interface Subscription {

    public void request(long n);
    public void cancel();
}
```

push pull híbrido

Resumo

- Reactive Streams é um padrão e especificação para bibliotecas reativas;
- Composta por 4 componentes: Publisher, Subscriber, Subscription e Processor;
- A biblioteca Project Reactor implementa o Reactive Streams;
- O Project Reactor é a base para stack reativa do Spring.

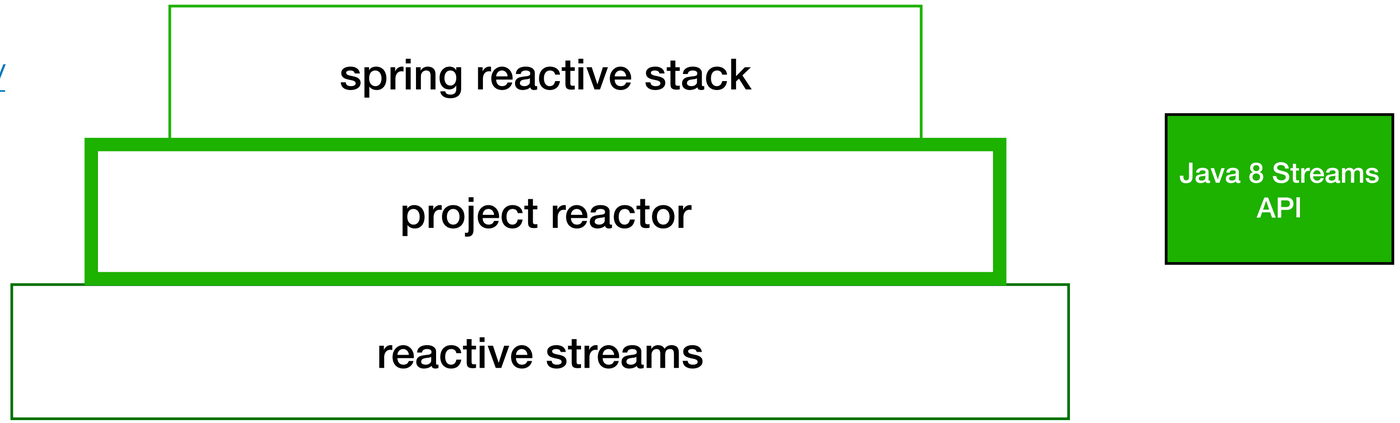
Módulo 4 - Introdução ao project reactor

Project reactor

Biblioteca reativa, baseada na especificação Reactive Streams, para construir aplicações não bloqueantes na JVM.

https://spring.io/reactive/

https://projectreactor.io/

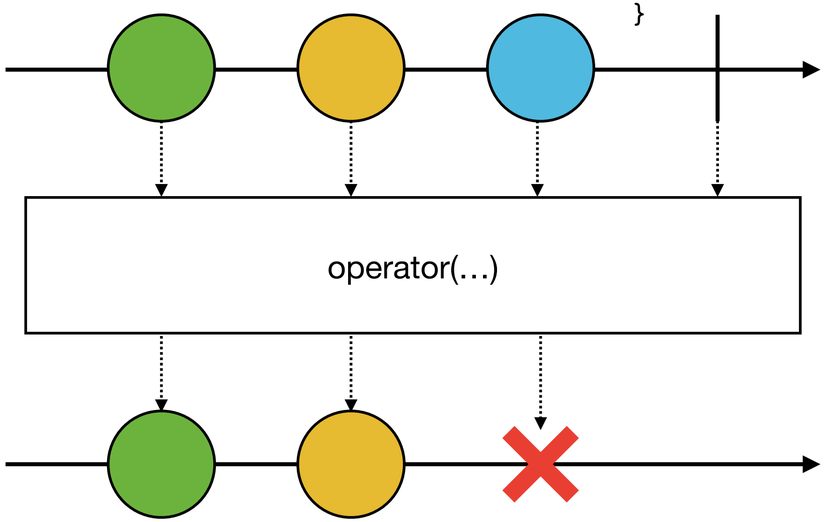


Java 8 Streams API

Project reactor types - Flux

Publisher que emite de 0 a N elementos.

```
public abstract class Flux<T> implements CorePublisher<T>{
…
}
```

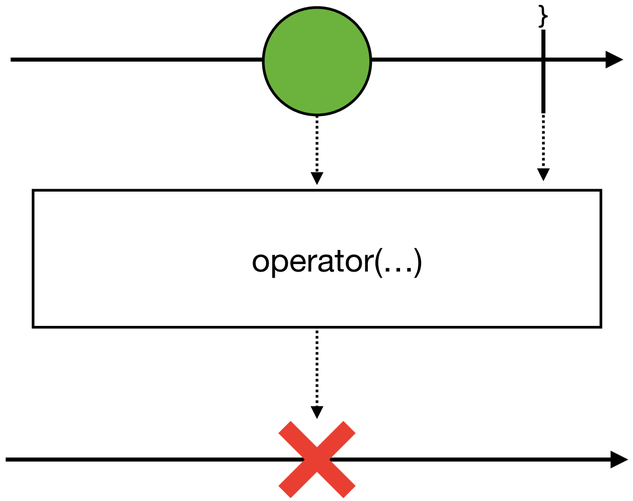


https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Flux.html

Project reactor types- Mono

Publisher que emite de 0 a 1 elemento.

```
public abstract class Mono<T> implements CorePublisher<T>{
...
}
```



https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Mono.html

Resumo

Biblioteca reativa baseada no especificação do Reactive Streams

Base para a stack reativa do Spring

Flux: Representa um fluxo de dados de 0 ou mais elementos

Mono: Representa um 0 ou 1 elemento

O consumo só inicia com o subscribe()

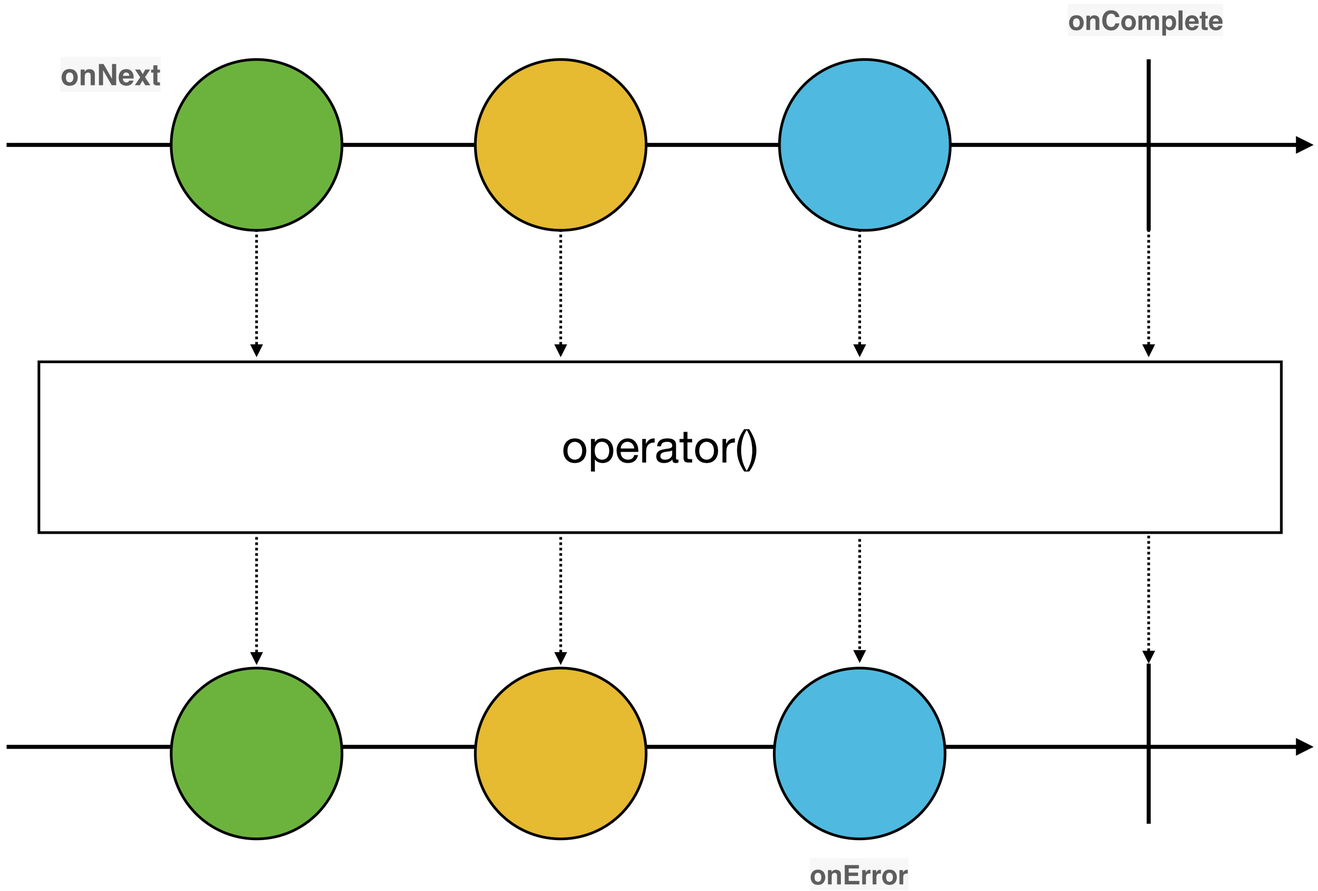
Módulo 5 - Operadores de criação

Operadores

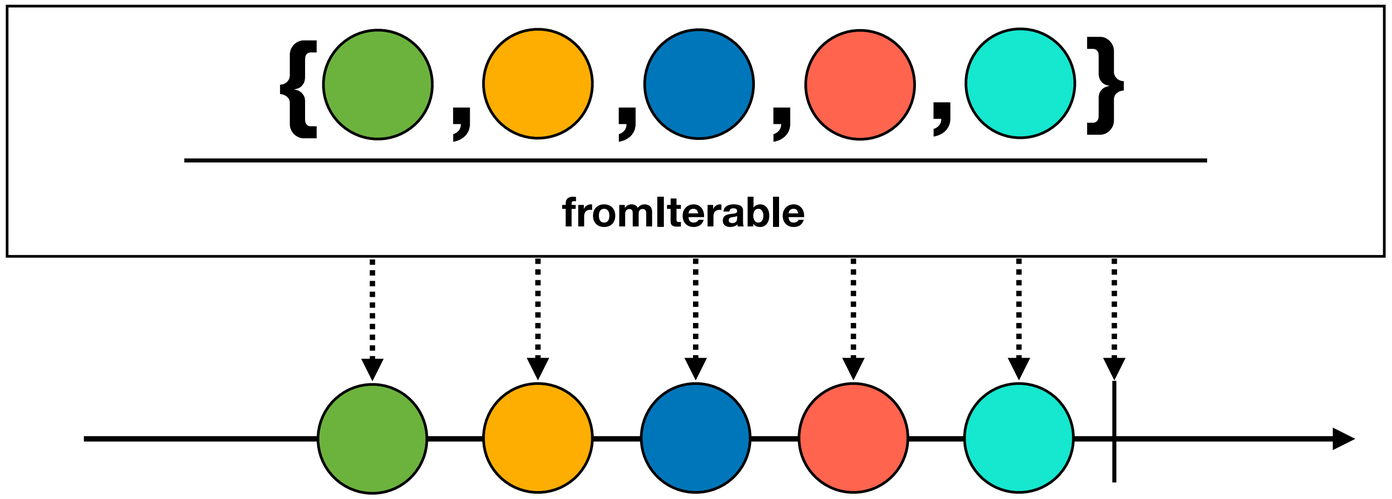
Funções utilizados em programação reativa para manipular, transformar e combinar fluxos de dados assíncronos.

https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Flux.html

Como funciona?



fromIterable()



Flux().create()

Cria um Flux com a capacidade de emitir **múltiplos** elementos de maneira síncrona ou **assíncrona** utilizando **FluxSink**. Isso inclui a emissão de elementos a partir de **múltiplas threads**.

```
Flux<T> create(Consumer<? super FluxSink<T>> emitter)
```

```
Flux.create(emitter -> {
    readFileFluxSink(emitter, file1);
    readFileFluxSink(emitter, file2);
    readFileFluxSink(emitter, file3);
    emitter.complete();
});
```

```
BufferedReader reader = new BufferedReader(new FileReader("file.txt"));
String line;
while ((line = reader.readLine()) != null) {
   emitter.next(line);
}
reader.close();
```

```
emitter.complete();
});
```

```
private void readFileFluxSink(FluxSink emitter, String filePath){
    BufferedReader reader = new BufferedReader(new FileReader(filePath));
    String line;
    while ((line = reader.readLine()) != null) {
        emitter.next(line);
    }
    reader.close();

}
```

Flux().generate()

Cria um Flux com a capacidade de emitir elementos um por um de maneira síncrona utilizando **SynchronousSink.**

```
Flux<T> generate(Callable<S> stateSupplier,
                 BiFunction<S,SynchronousSink<T>,S> generator,
                 Consumer<? super S> stateConsumer)
```

```
Flux.generate(
 () -> {BufferedReader(new FileReader(filePath))},
       (reader, emitter) -> {
               String line = reader.readLine();
                if (line != null) {
                    emitter.next(line);
                } else {
                    emitter.complete();
                }
            return reader;
        },
       reader -> {reader.close()}
);
```

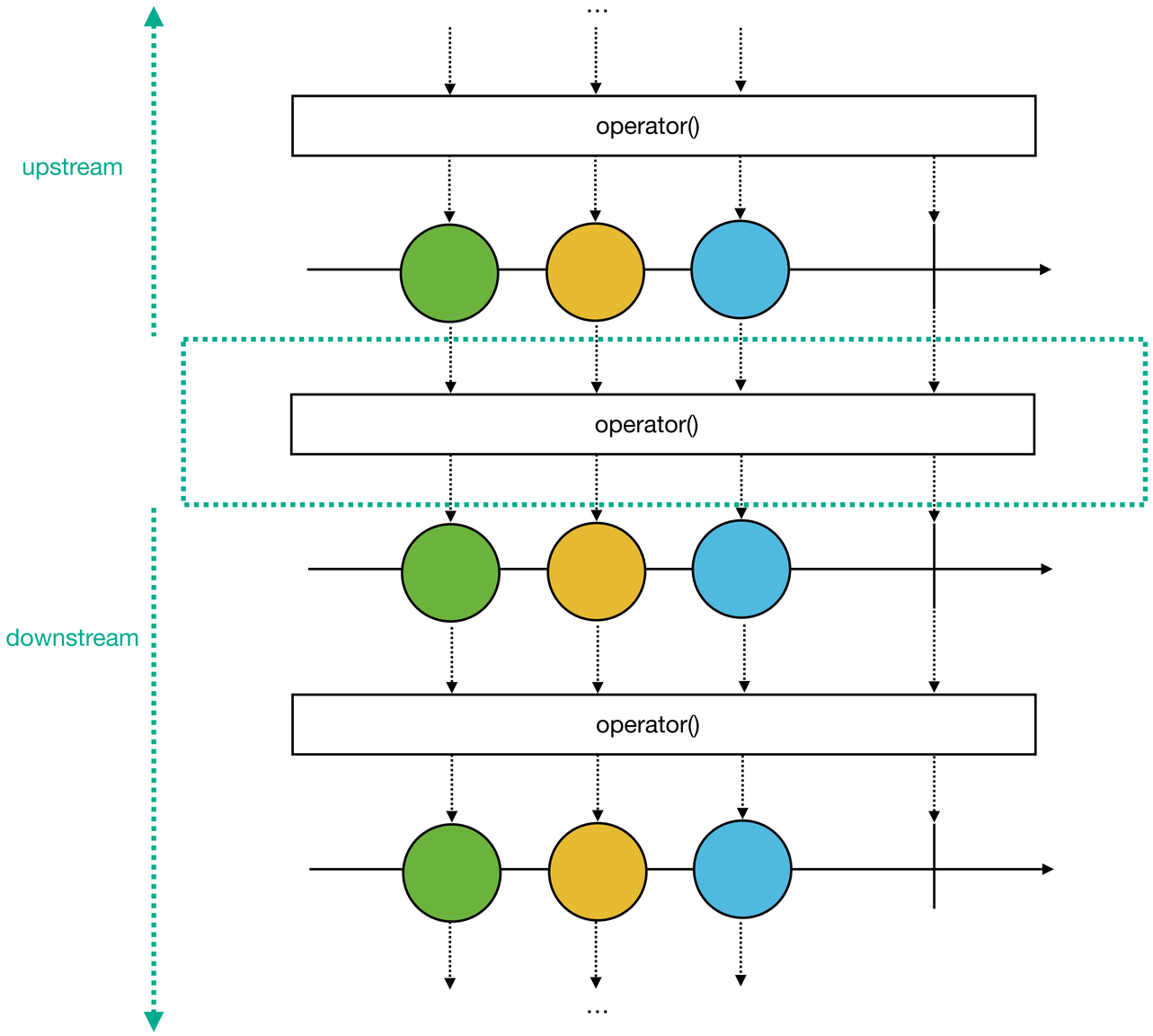
Resumo

- **just()**: Cria um Flux ou Mono de maneira simples;
- **fromIterable()**: Converte uma coleção em um Flux;
- **fromArray()**: Converte um array em um Flux;
- **fromStream()**: Converte um Stream em um Flux;
- **range():** Gera um Flux de números em uma faixa específica.

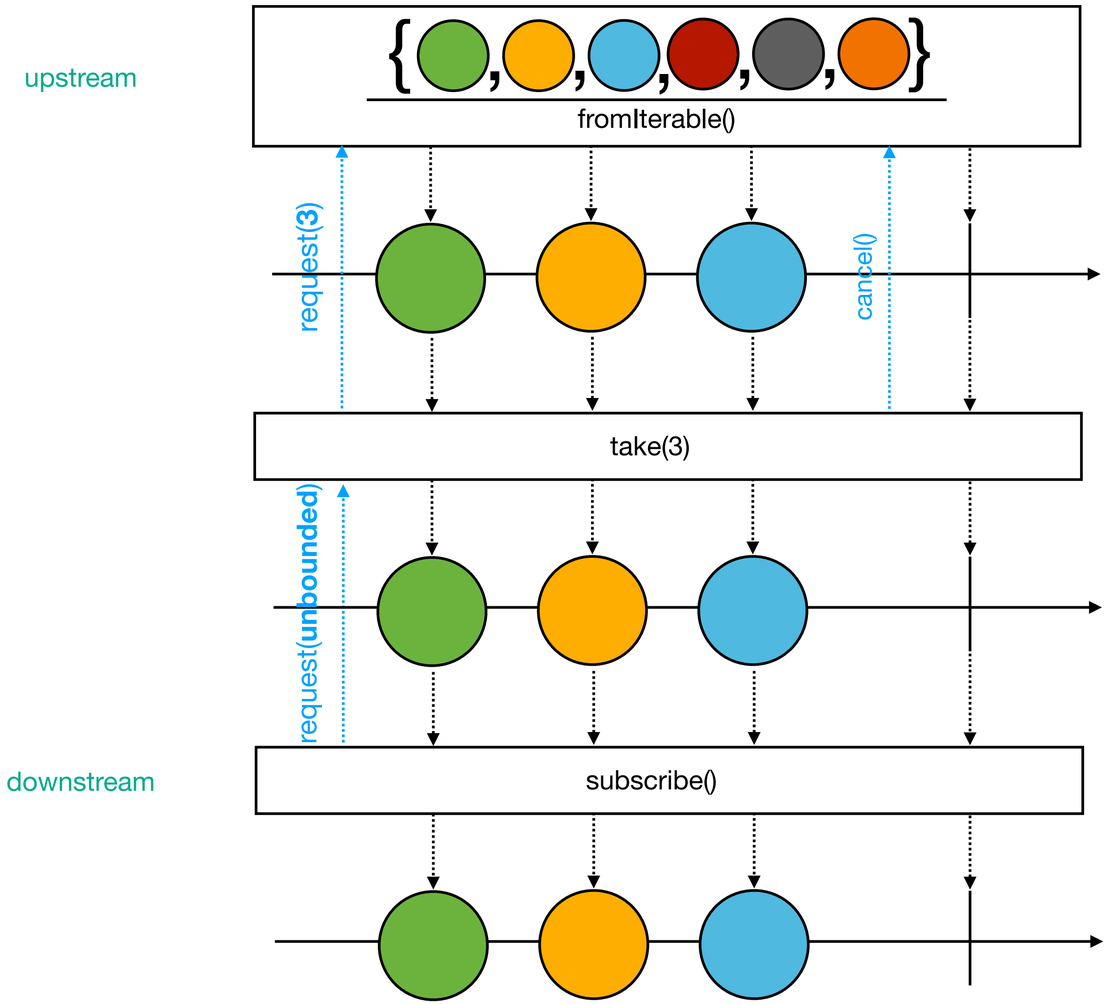
- **Flux.create():** Cria um Flux com a capacidade de emitir **múltiplos** elementos de maneira síncrona ou **assíncrona**. Permite a emissão de elementos a partir de **múltiplas threads**;
- **Flux.generate()**: Cria um Flux com a capacidade de emitir elementos um por um de maneira síncrona**.**

Módulo 6 - Operadores

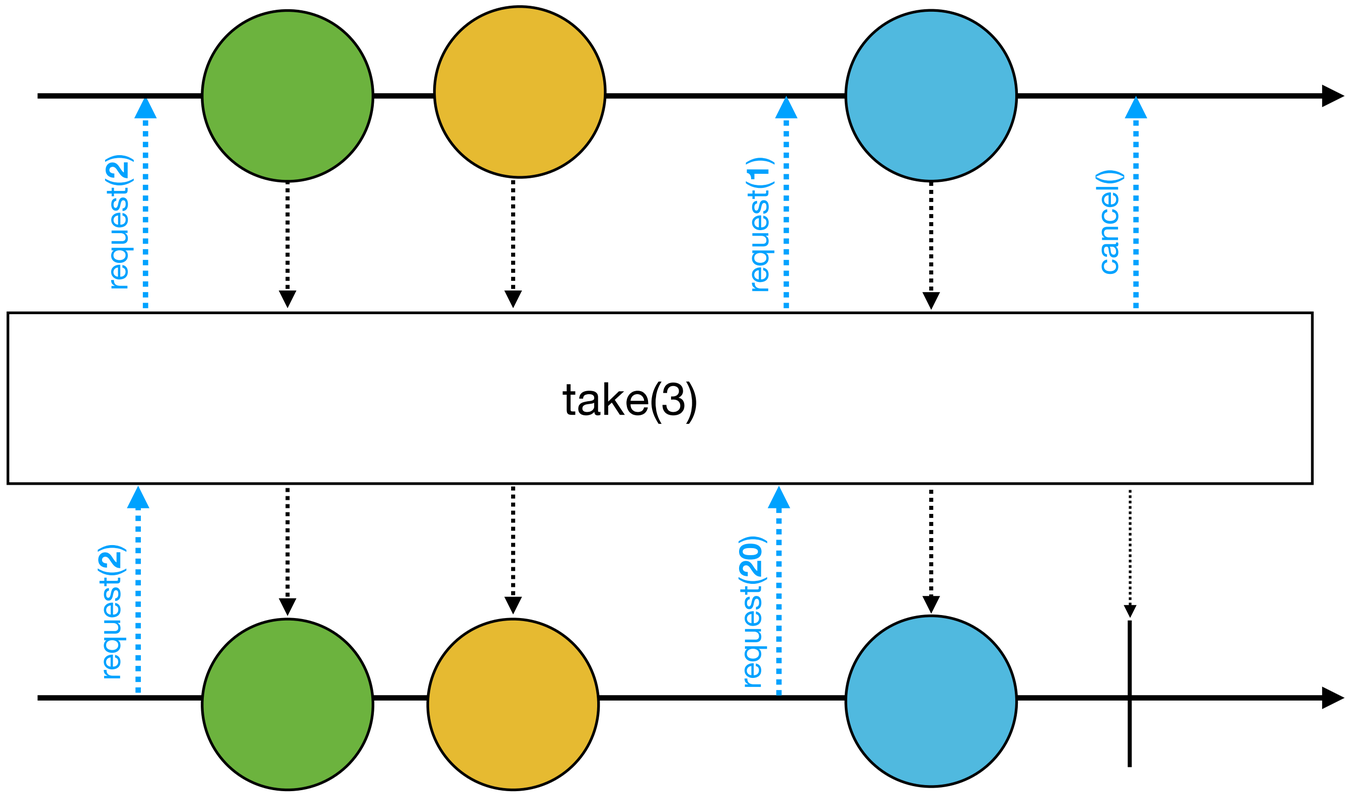
Multiple operators



take()

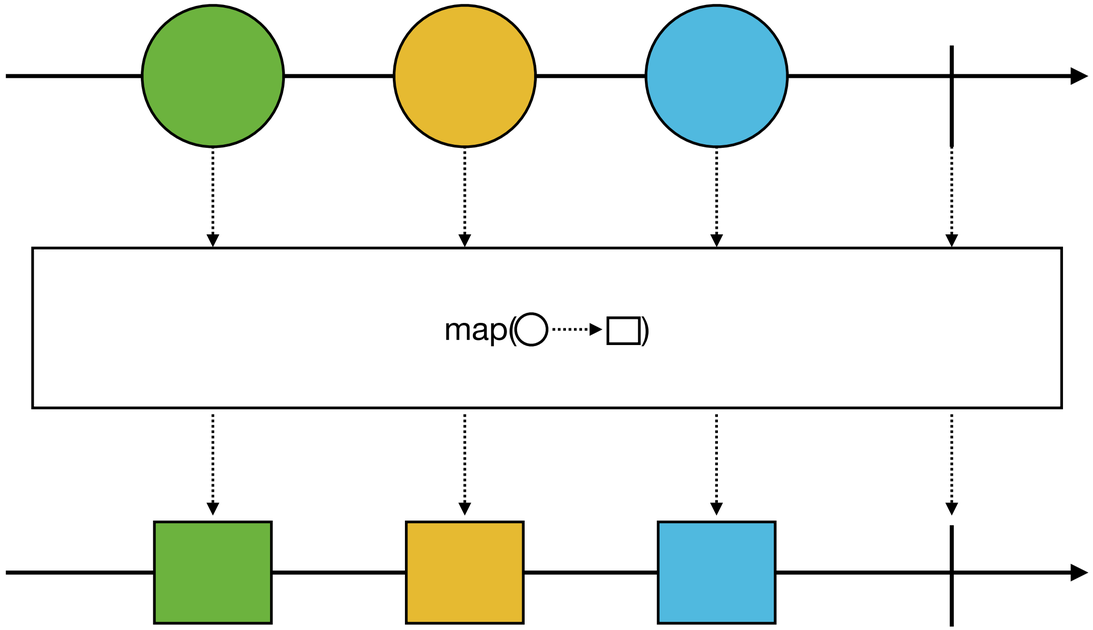

upstream
fromIterable()
request(3)
cancel()
take(3)
request(unbounded)
downstream
subscribe()

take()



map()

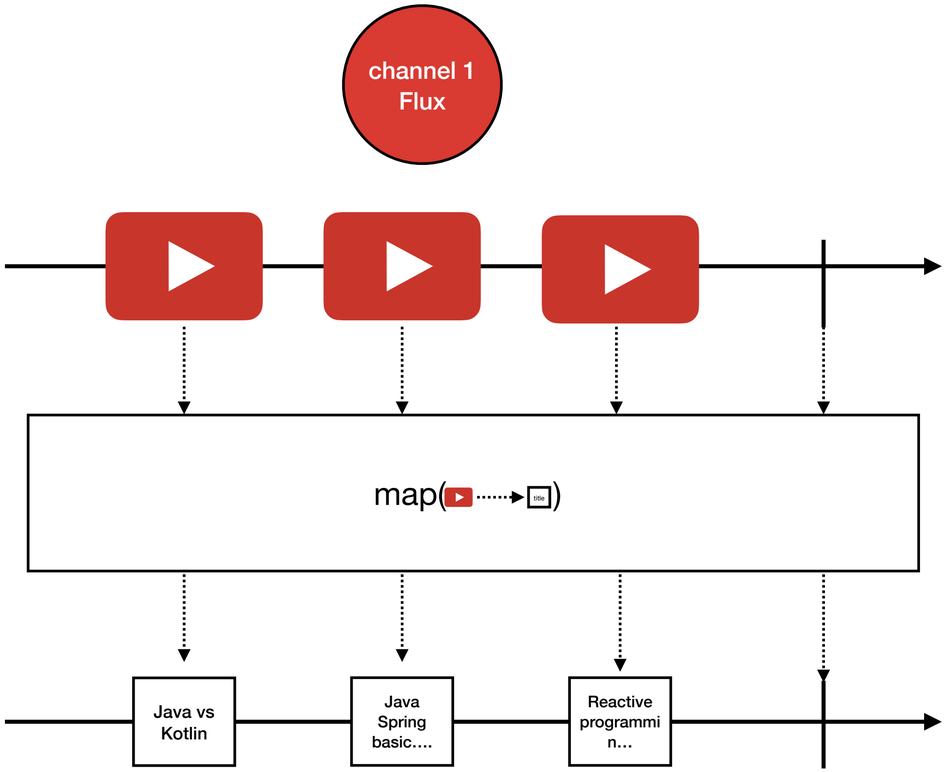
Transforma os itens emitidos



flatmap()

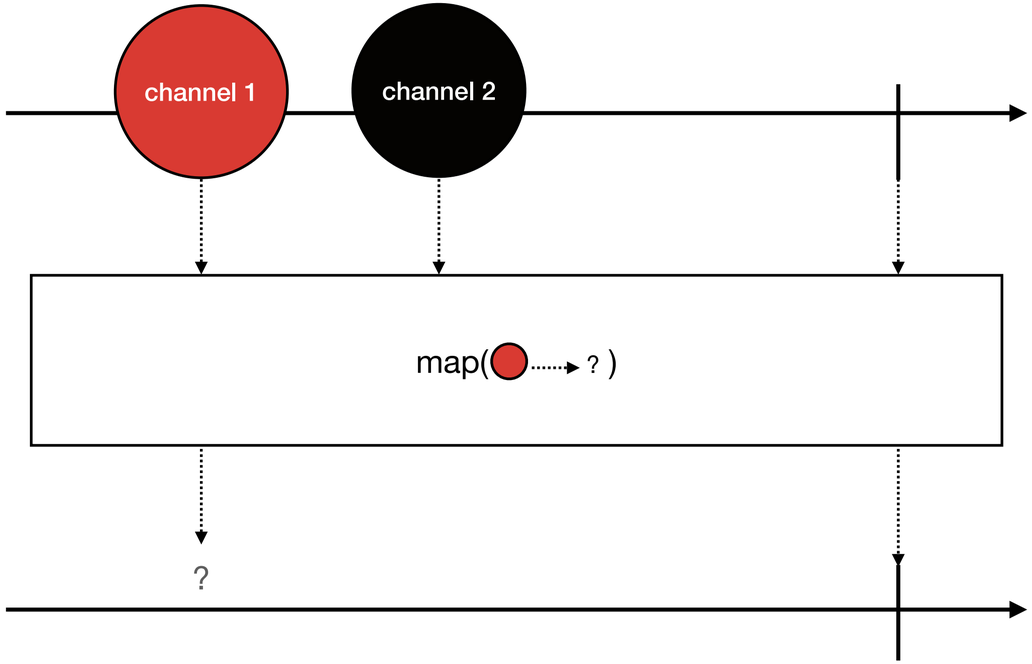
```
YoutubeChannel youtubeChannelFlux = new YoutubeChannel(generateVideos());

StepVerifier
        .create(youtubeChannelFlux.map(Video::getName)
        )
        .expectNext("Reactive Programming with Java",
                "Reactive System vs Reactive programing")
        .verifyComplete();
```



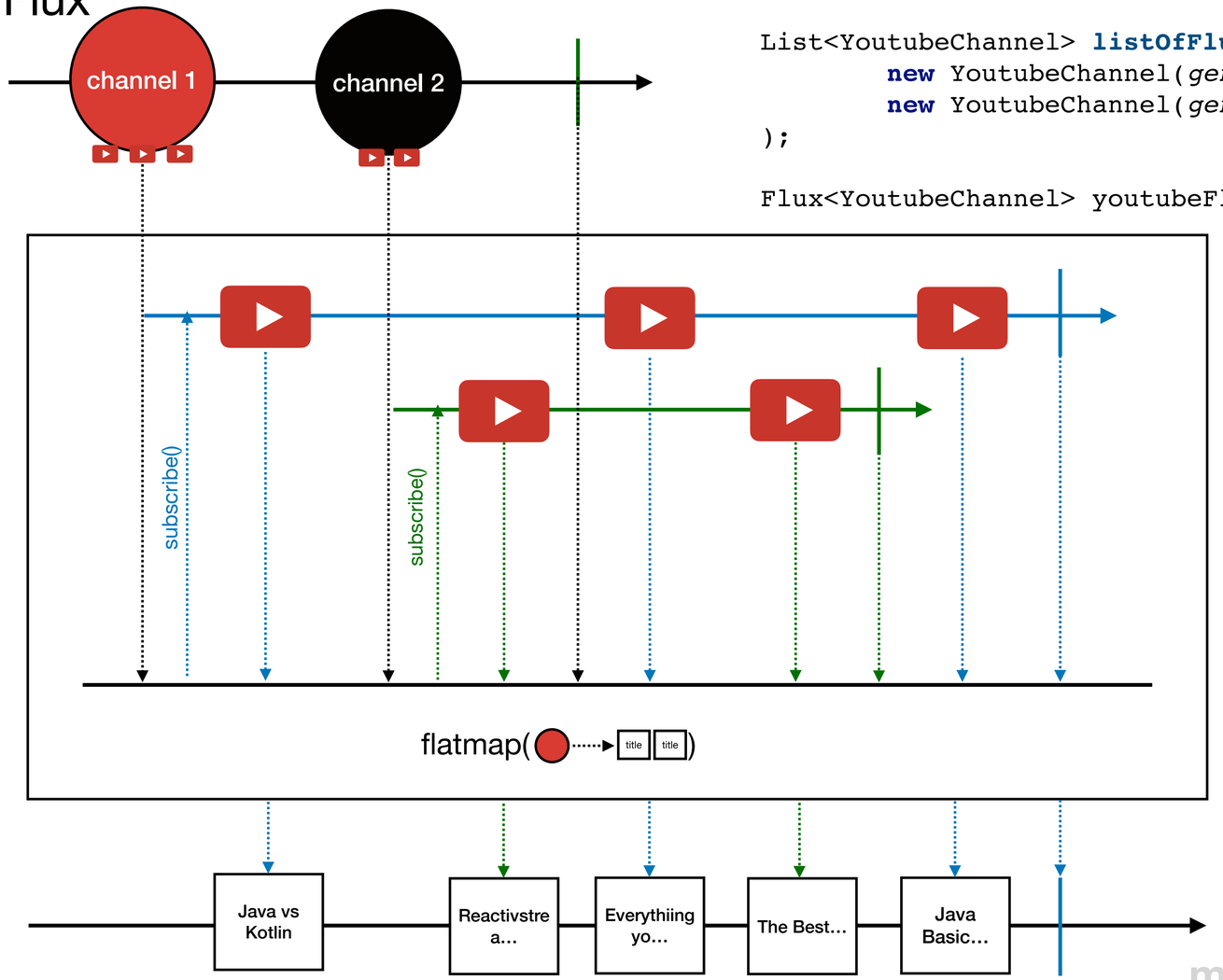

```
List<YoutubeChannel> listOfFlux =  Arrays.asList(
        new YoutubeChannel(generateVideos()),
        new YoutubeChannel(generateVideos2())
);

Flux<YoutubeChannel> youtubeFlux = Flux.fromIterable(listOfFlux);
```



flatmap()

Transformar os elementos emitidos por um Flux em Publishers (inners publishers), e em seguida, mescla-os em um único Flux




```
List<YoutubeChannel> listOfFlux =  Arrays.asList(
        new YoutubeChannel(generateVideos()),
        new YoutubeChannel(generateVideos2())
);

Flux<YoutubeChannel> youtubeFlux = Flux.fromIterable(listOfFlux);
```

transform()

```
Flux.just(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
        .filter(num -> num % 2 == 0)
        .doOnNext(num ->System.out.print("Square of " + num))
        .map(num -> num * num)
        .subscribe(result -> System.out.print(" = " + result));
```

```
Function<Flux<Integer>,Flux<Integer>> squareNumber(){
  return flux -> flux
                .doOnNext(num ->System.out.print("Square of " + num))
                .map(num -> num * num);
}
```

```
Flux.just(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
        .filter(num -> num % 2 != 0)
        .doOnNext(num ->System.out.print("Square of " + num))
        .map(num -> num * num)
        .subscribe(result -> System.out.print(" = " + result));
```

side effects

Permitem adicionar efeitos colaterais sem modificar o fluxo de dados.

exemplo:

Enviar métricas para uma ferramenta de observability no final da execução;

Adicionar logs.

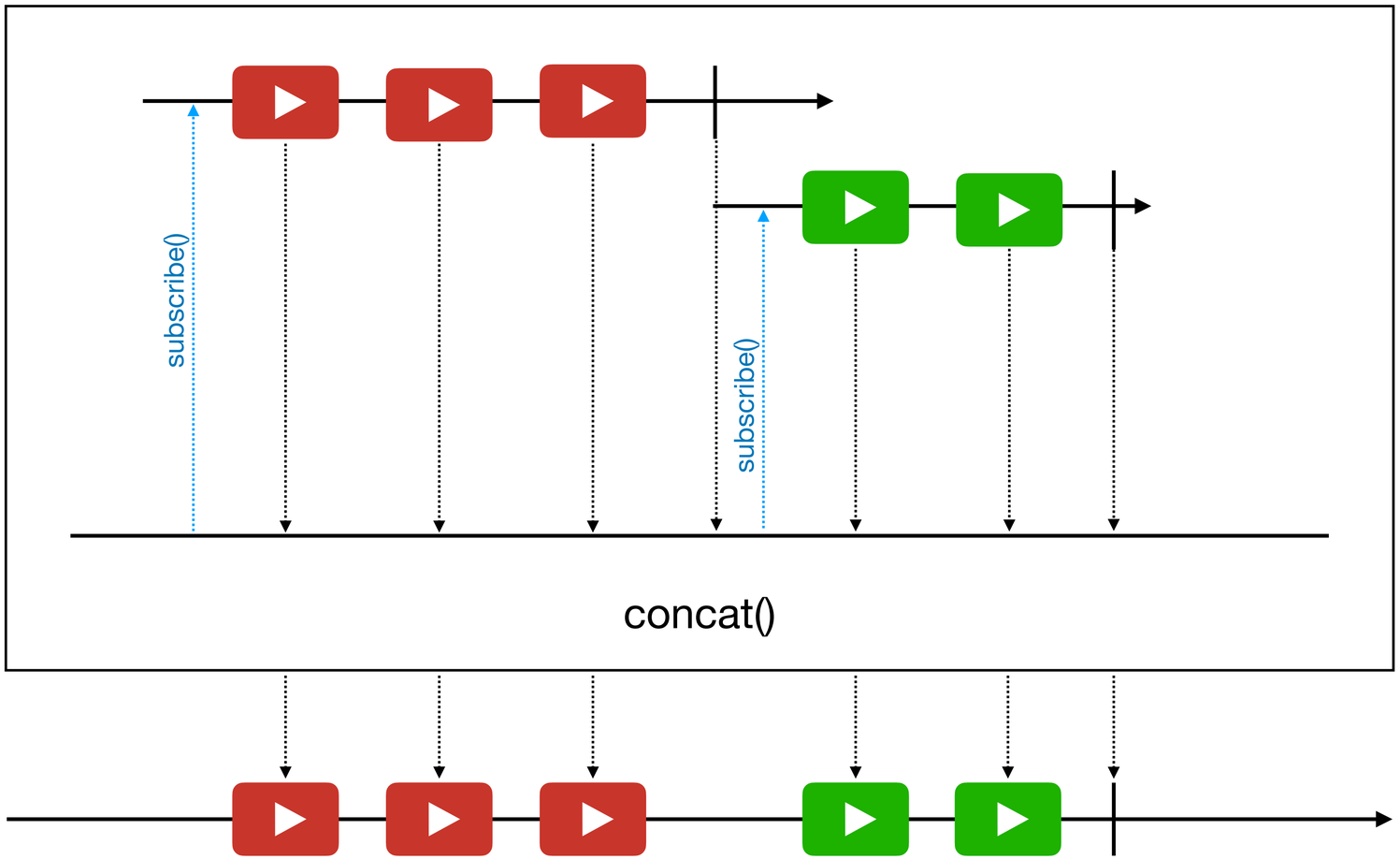
Resumo

- **just(), fromIterable(), fromArray(), fromStream(), range(), Mono.just()**: Criação;
- **take()**: Limita o número de elementos emitidos por um fluxo ou mono;
- **filter()**: Usado para filtrar um valor;
- **map()**:Transforma os itens emitidos;
- **flatMap():** Transforma os elementos emitidos por um Flux em Publishers (inners publishers), e em seguida, mescla-os em um único Flux;
- **delayElements()**: Atrasa a emissão de cada elemento de um fluxo ou mono por um período específico;
- **transform():** Usado para aplicar uma transformação personalizada em um fluxo ou mono;
- **sideEffects**: Permitem adicionar efeitos colaterais sem modificar o fluxo de dados.

Módulo 7 - Combinando fluxos com operadores

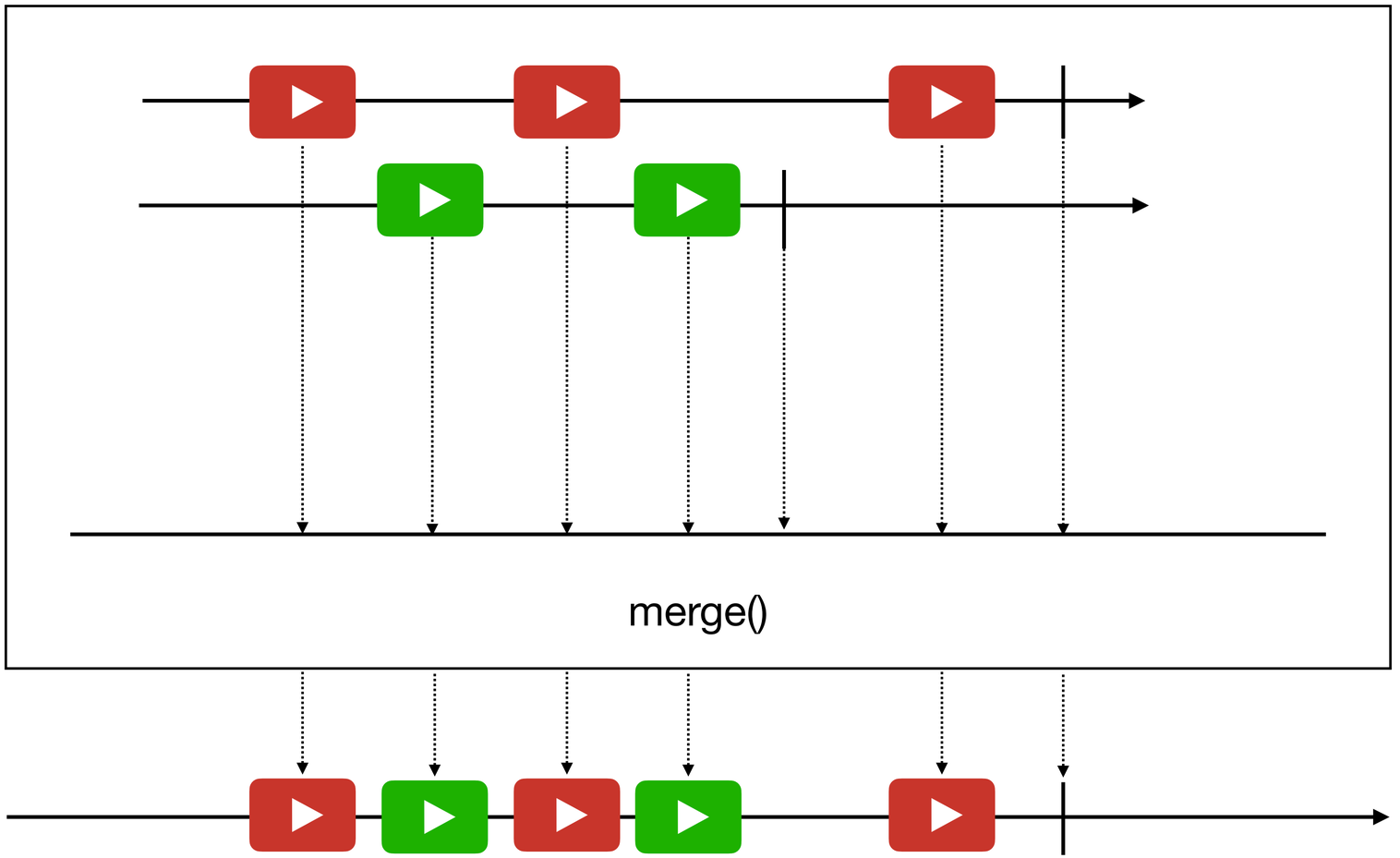
concat()

Concatena os elementos de forma sequencial



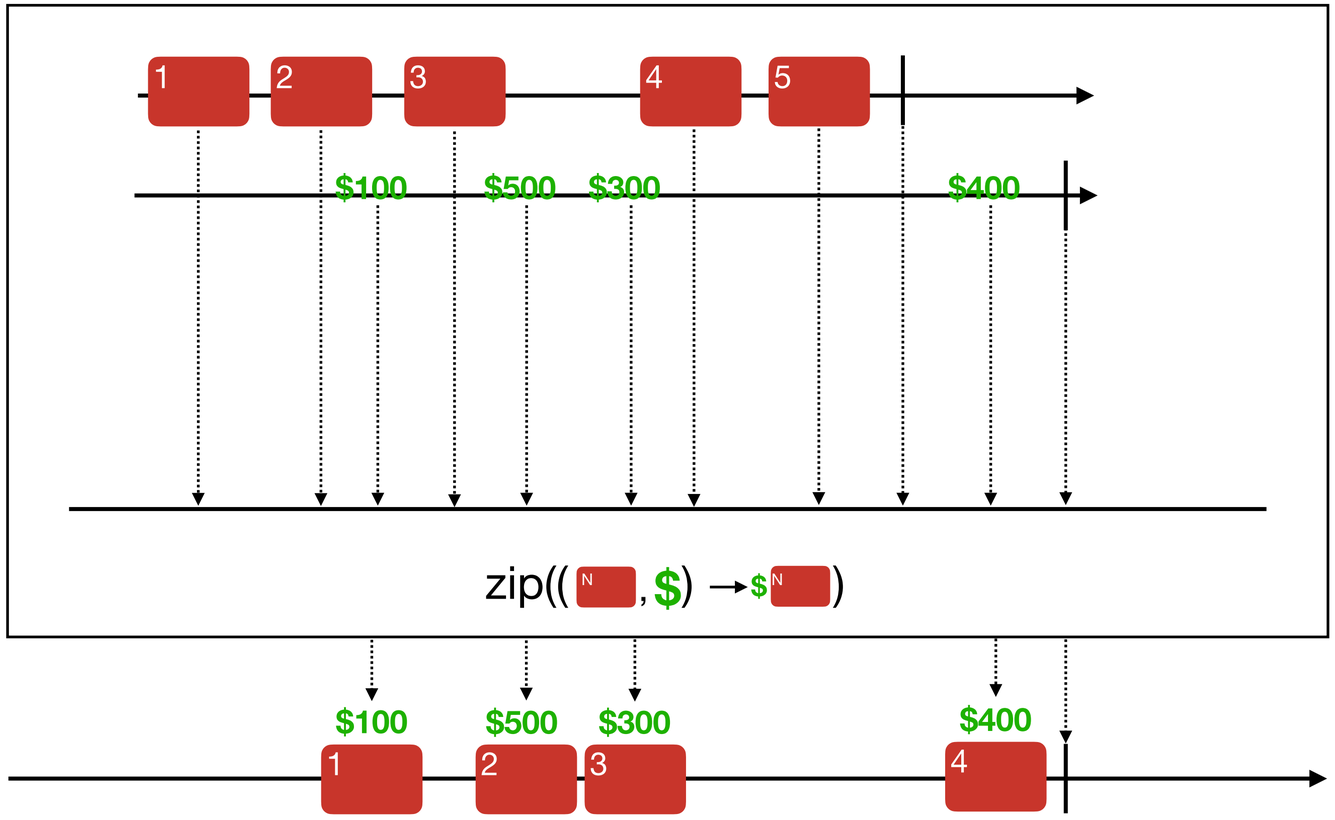
merge()

Faz o merge de múltiplos Publishers em uma sequencia mesclada



zip()

Combina os elementos de múltiplos fluxos em um único fluxo



Resumo

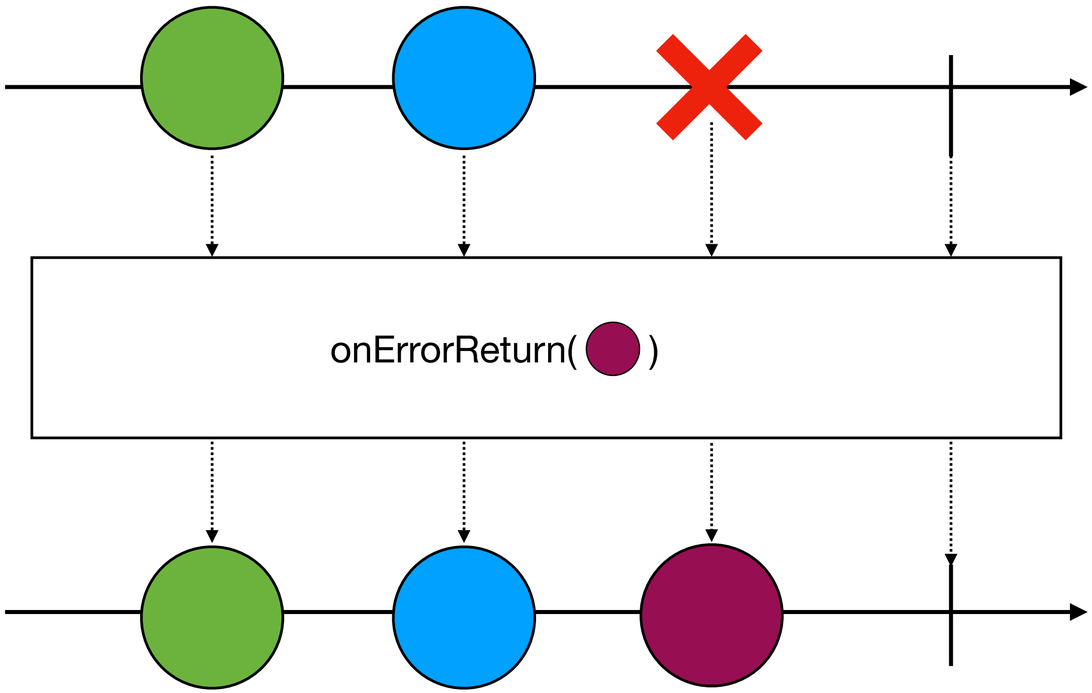
São usados usados para combinar elementos de Publishers:

- **concat()**: Concatena os elementos de forma sequencial;
- **merge()**: Faz o merge de múltiplos Publisher em uma sequencia mesclada;
- **zip()**: Combina os elementos de múltiplos fluxos em um único fluxo.

Módulo 8 - Tratamento de erros

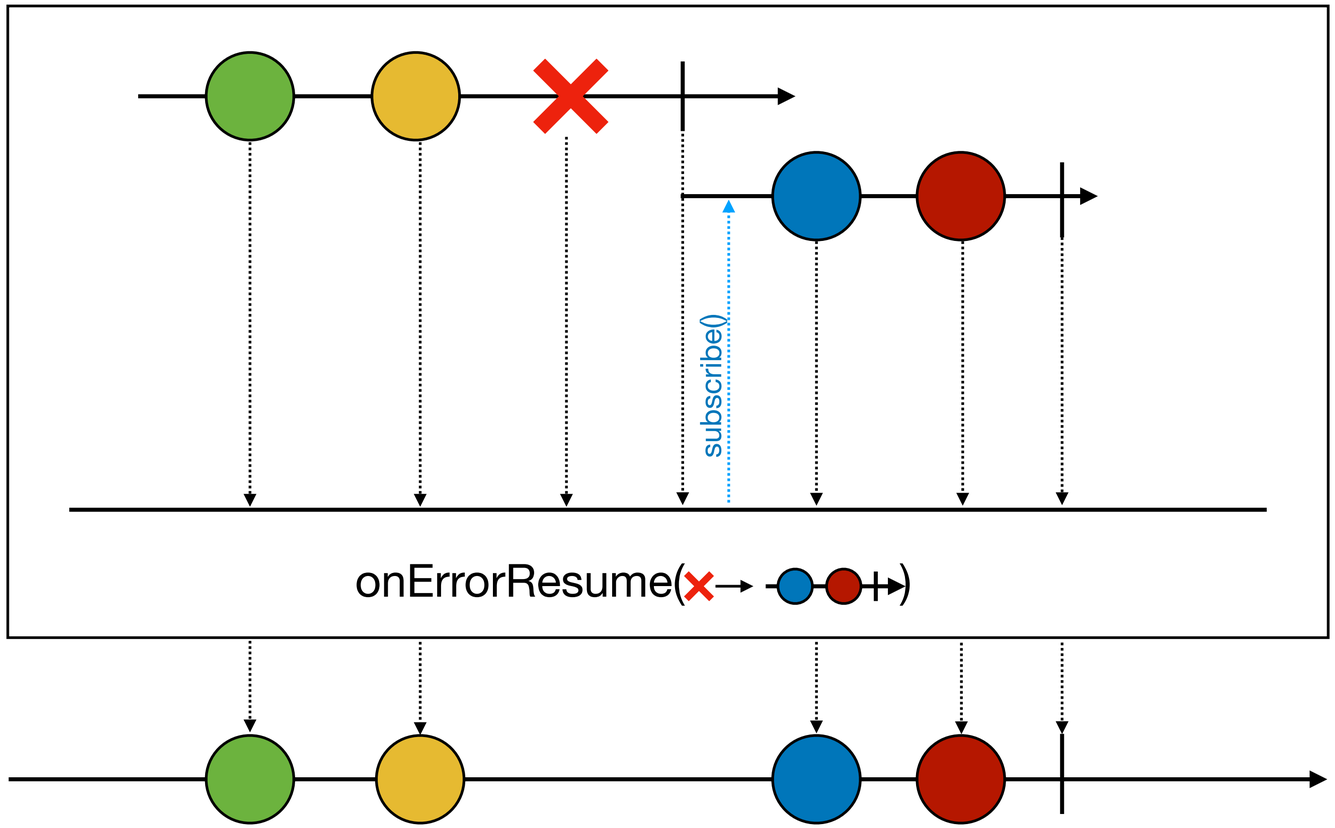
onErrorReturn()

Permite definir uma valor de fallback em caso de erro



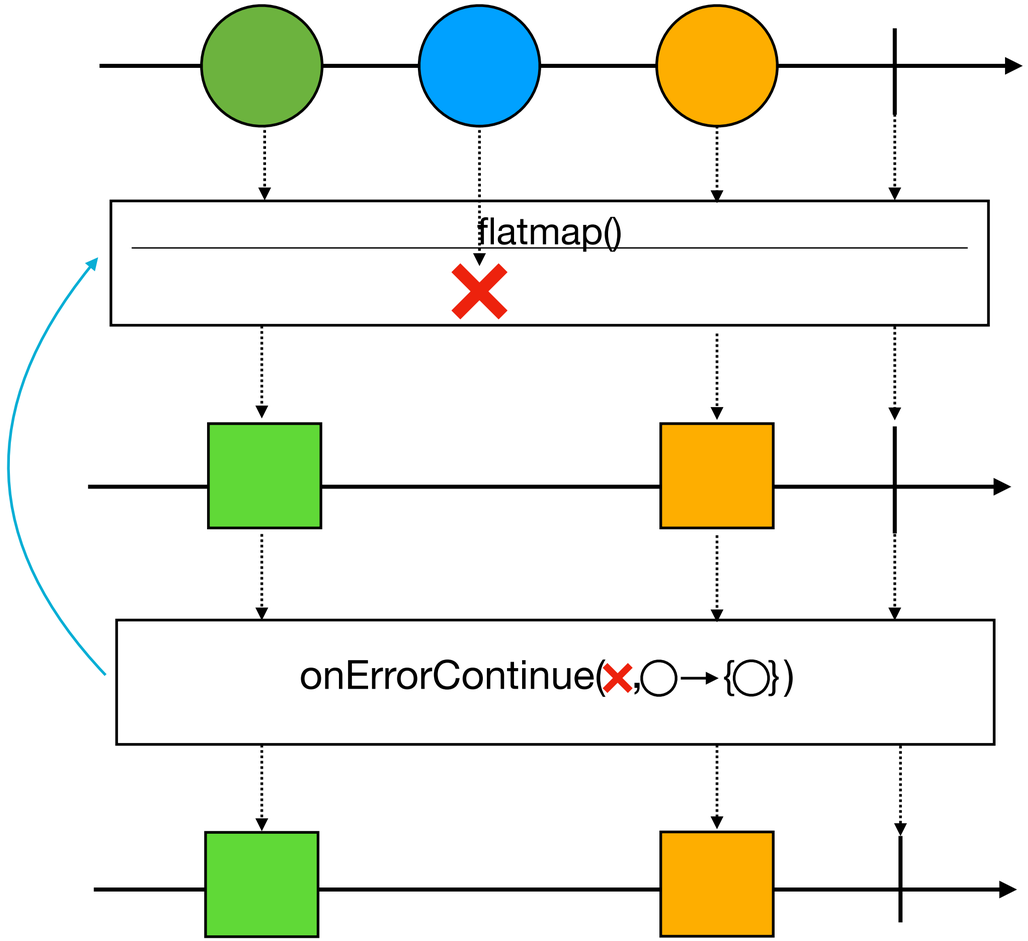
onErrorResume()

Permite definir uma publisher de fallback em caso de erro



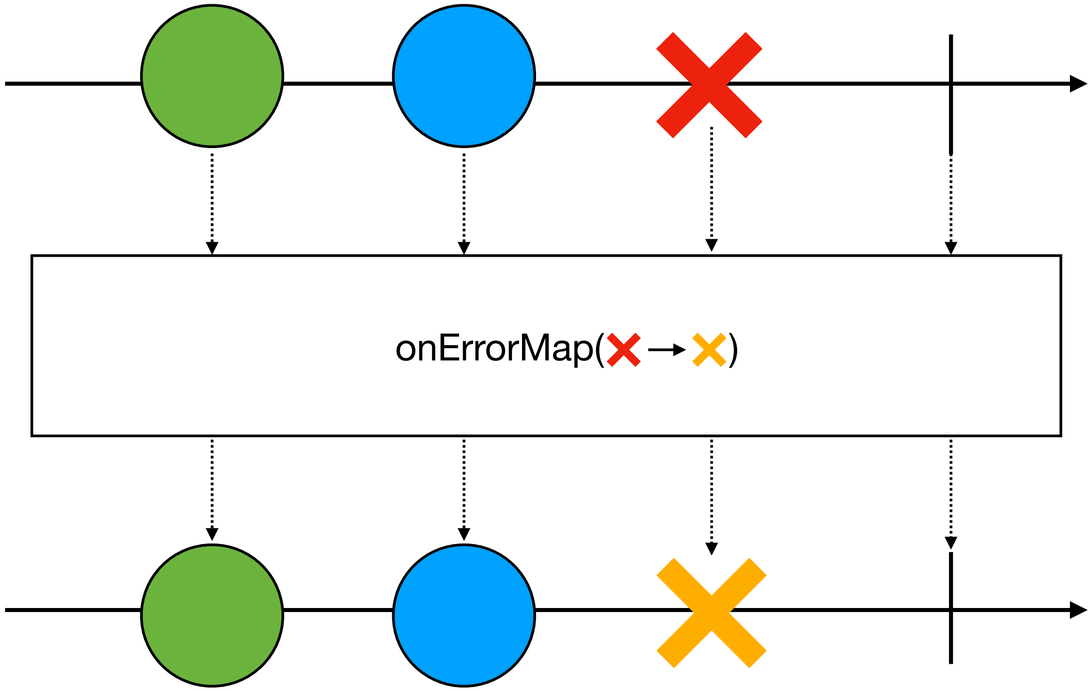
onErrorContinue()

Permita que os operadores upstream compatíveis se recuperem de erros eliminando o elemento com erro da sequência e continuando com os elementos subsequentes



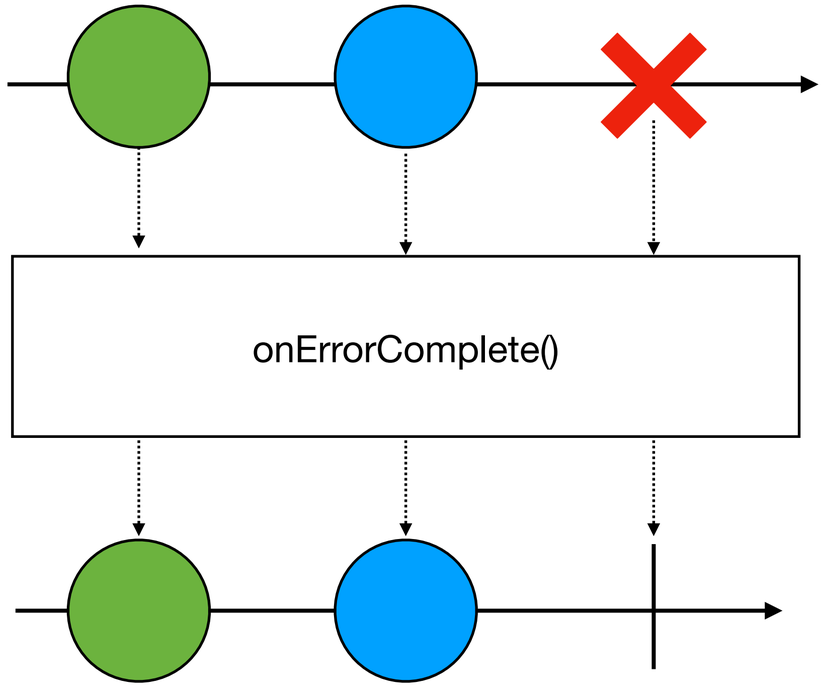
onErrorMap()

Transforme qualquer erro emitido por este Flux aplicando uma função a ele de forma síncrona.



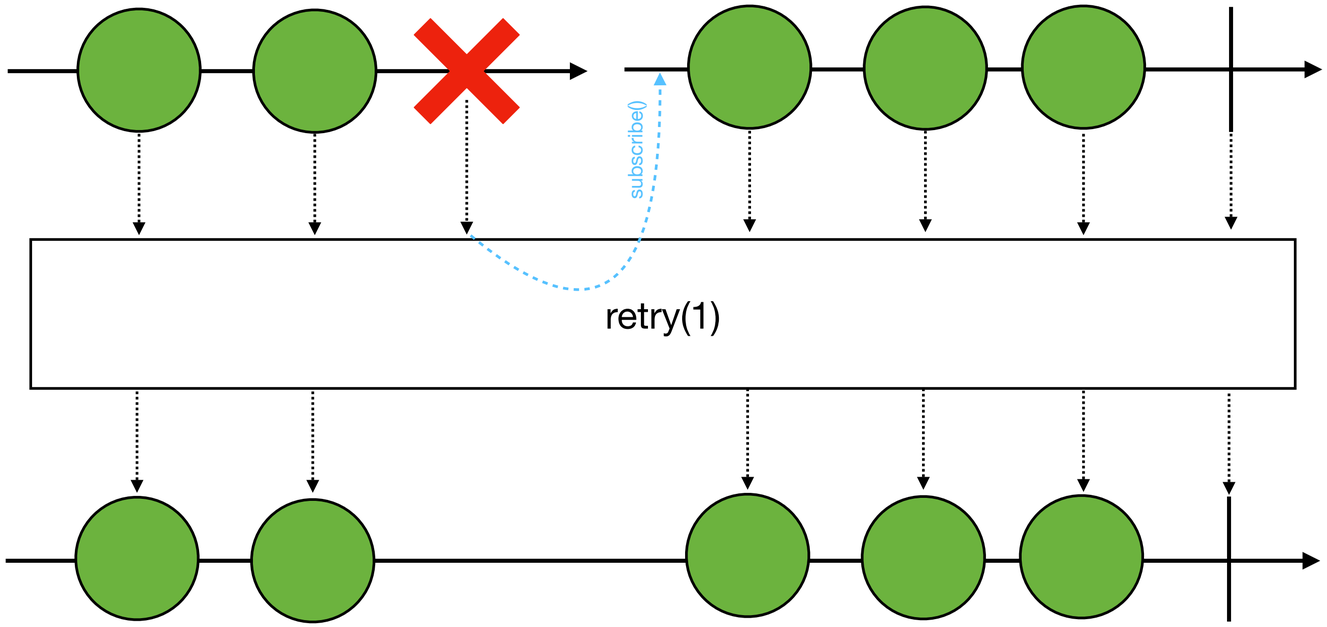
onErrorComplete()

Termina a sequência substituindo um sinal onError por um sinal onComplete



retry()

Faz o “re-subscribe” em um fluxo dado um sinal de erro, por um número fixo de tentativas



Resumo

onErrorReturn: Define um valor padrão para ser retornado em caso de erro;

onErrorResume: Permite definir um Publisher de fallback em caso de erro;

onErrorContinue: Permite continuar a execução mesmo após ocorrer um erro;

onErrorMap: Transforma um tipo de erro em outro;

onErrorComplete: Descarta o erro e simplesmente completa o Flux ou Mono.

defaultIfEmpty: Define um valor padrão para ser emitido caso o Flux ou Mono esteja vazio;

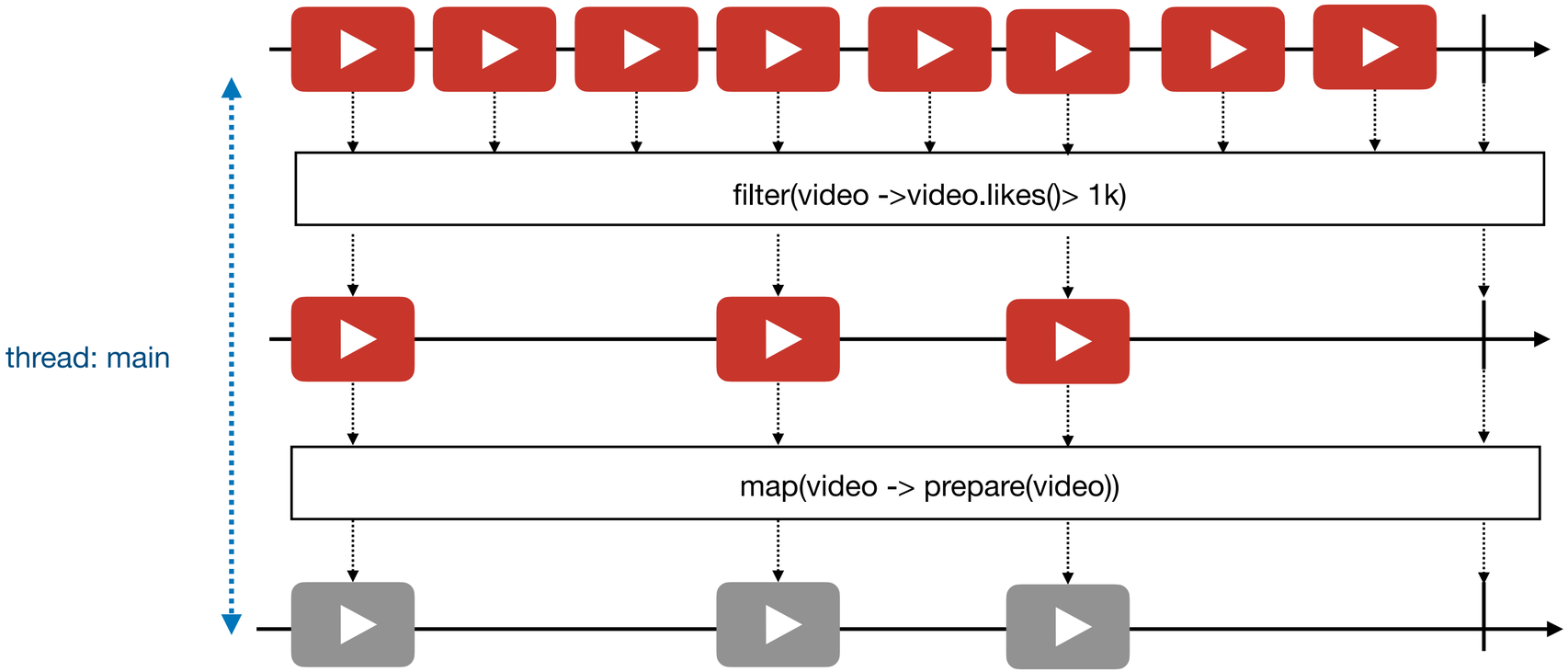
switchIfEmpty: Troca o Flux ou Mono por outro Flux ou Mono alternativo caso esteja vazio.

retry: Tenta novamente a operação em caso de erro;

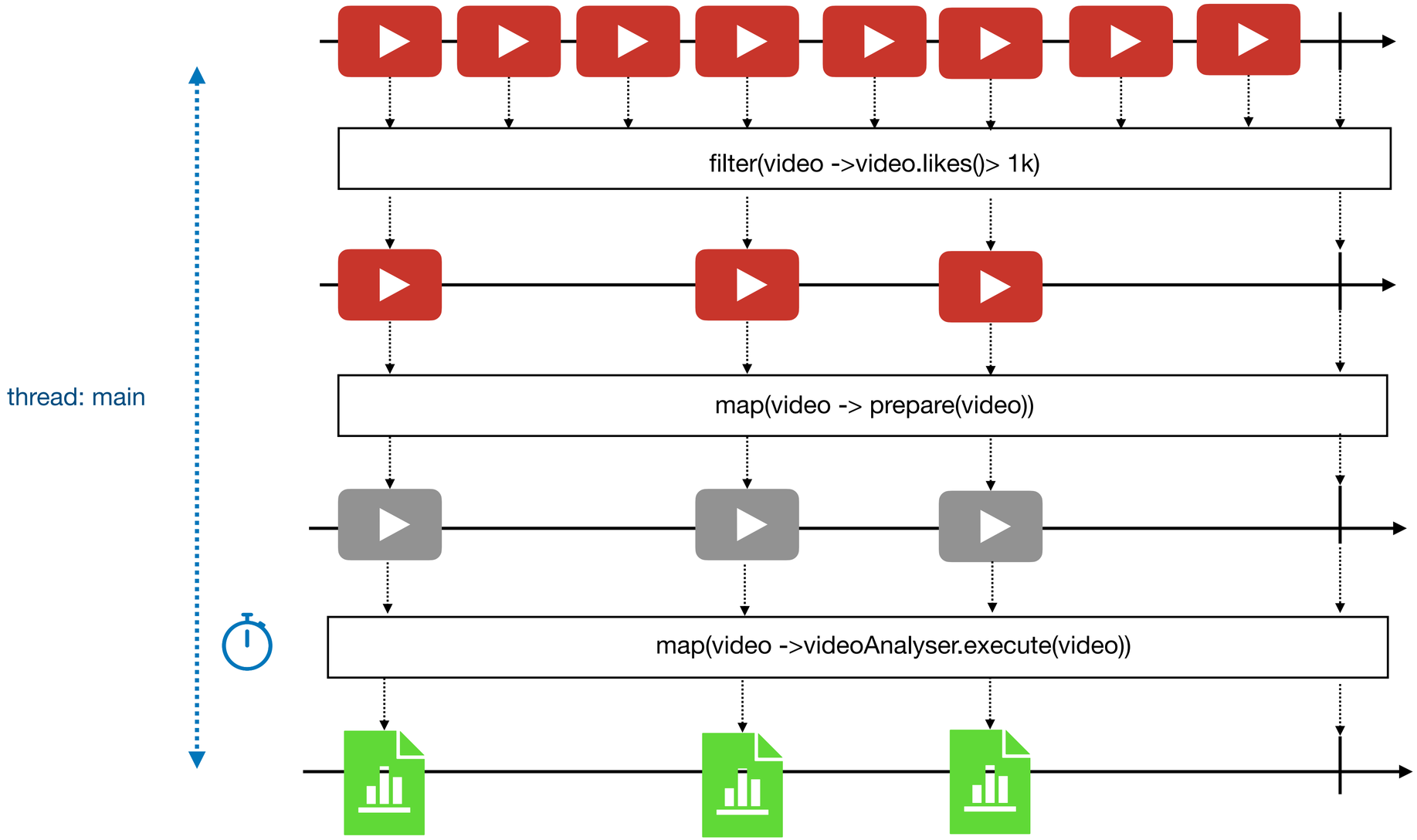
retryWhen: Tenta novamente a operação em caso de erro, de forma customizável.

Módulo 9 - Schedulers

Schedulers


thread: main
filter(video ->video.likes()> 1k)
map(video -> prepare(video))

Schedulers


thread: main
filter(video ->video.likes()> 1k)
map(video -> prepare(video))
map(video ->videoAnalyser.execute(video))

Schedulers

Schedulers são abstrações que oferecem ao usuário maior controle sobre em qual thread ou contexto as tarefas devem ser executadas.

Schedulers

boundedElastic(): Otimizado para execuções mais longas. Número máximo de threads criadas:

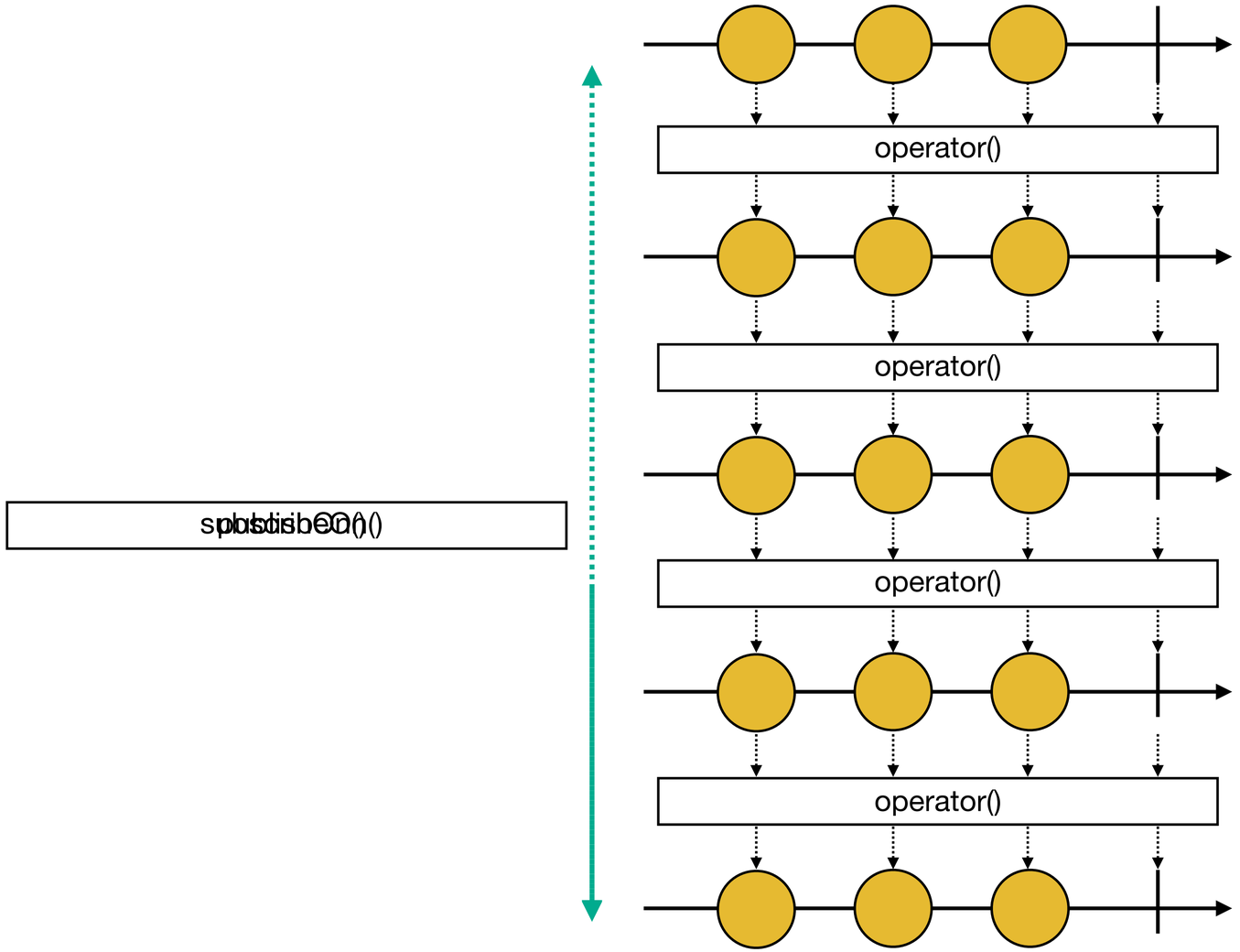
10 vezes o número de CPU cores.

parallel(): Otimizado para operações intensivas de CPU ou cálculos pesados. Número máximo de threads criadas: Número de CPU cores.

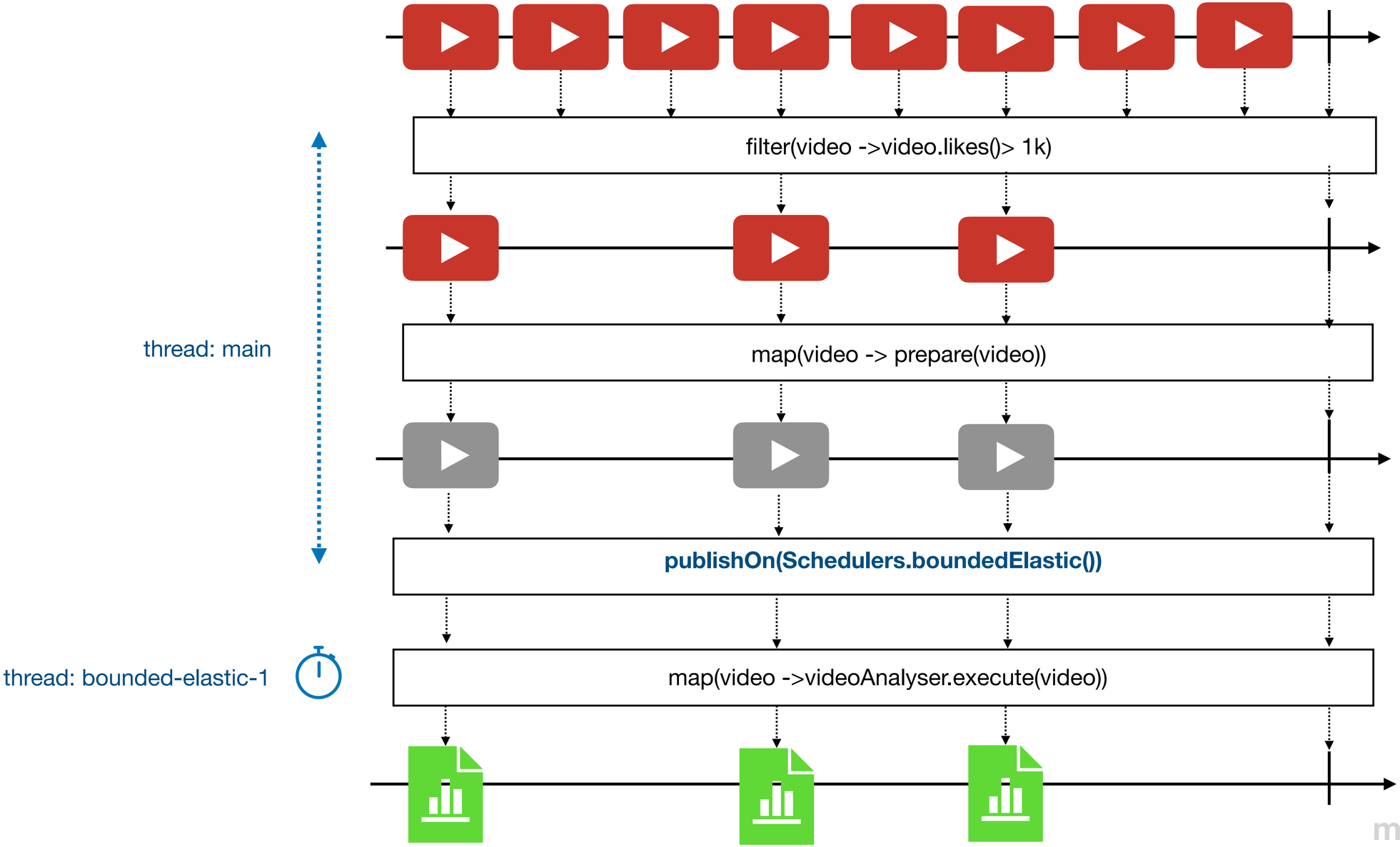
single(): Otimizado para execuções únicas de baixa latência. Única thread criada.

https://projectreactor.io/docs/core/release/api/reactor/core/scheduler/Schedulers.html

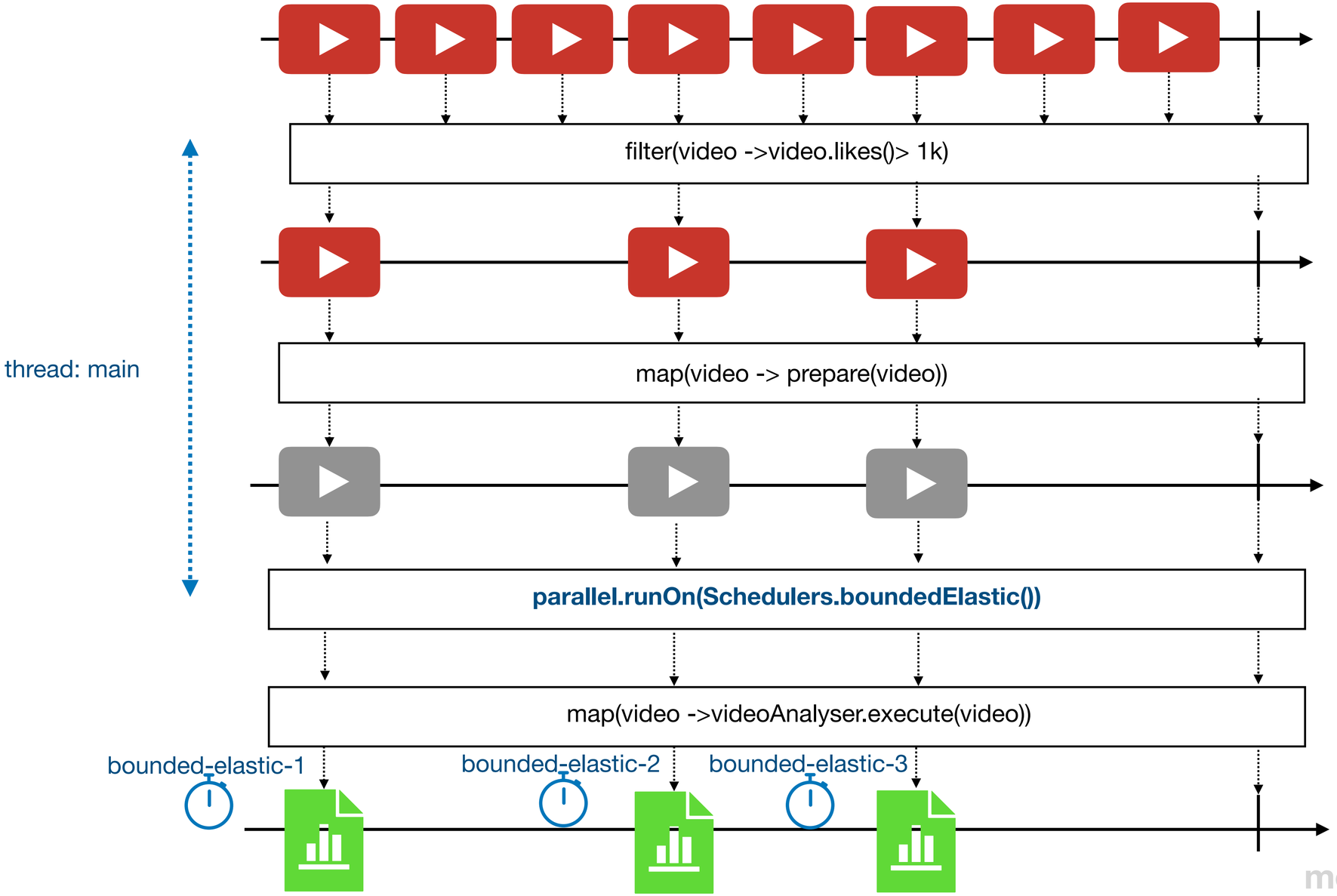
Schedulers - publishOn vs subscribeOn



Schedulers - publishOn



Schedulers - parallel



Resumo

Scheduler é uma abstração que dá ao usuário controle sobre o threading.

Existem 3 tipos de schedulers: **boundedElastic(), parallel() e single().**

publishOn(Scheduler scheduler): Usado para definir o scheduler onde as operações subsequentes de um fluxo (como map, filter, etc.) serão executadas.

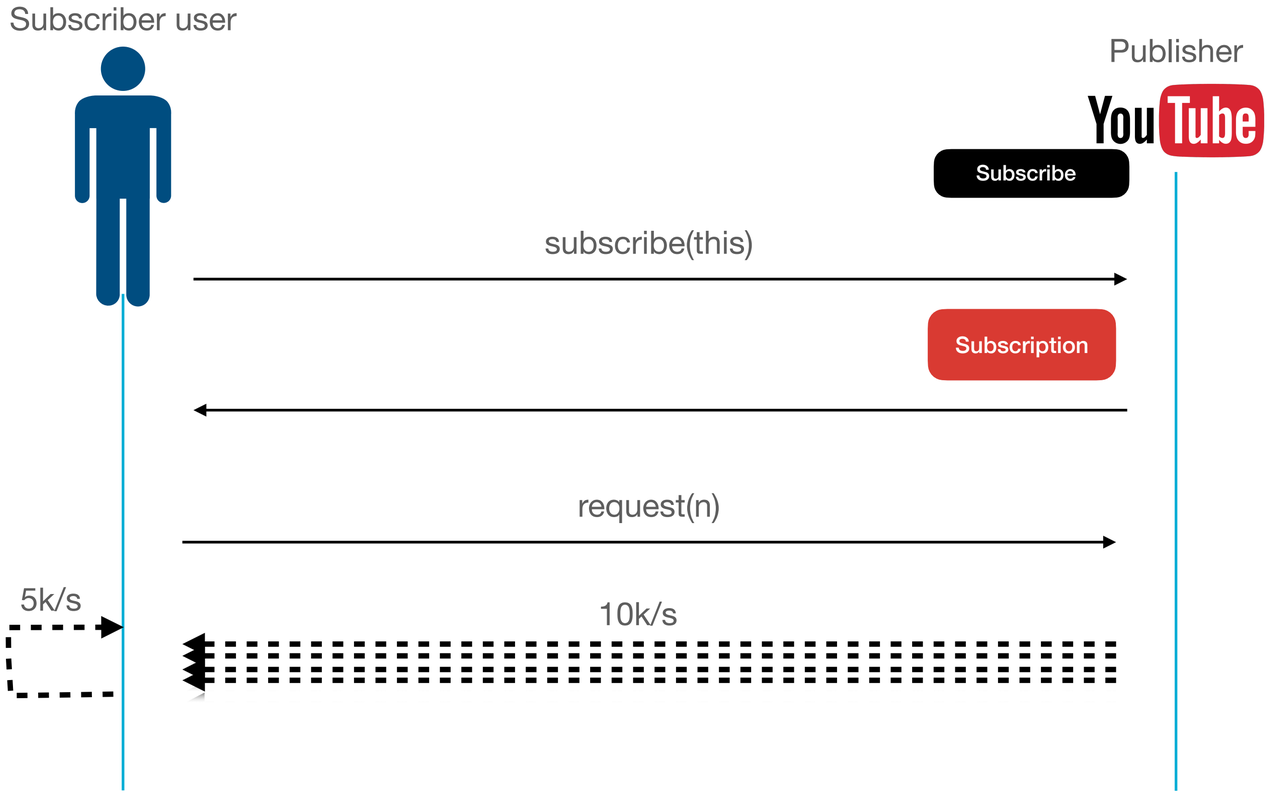
subscribeOn(Scheduler scheduler): Usado para definir o scheduler onde todas as operações serão afetadas (upstream e downstream).

parallel().runOn(Scheduler scheduler): Permite executar operações simultaneamente em paralelo em várias threads.

Mono.fromCallable(() -> function()).publishOn(Scheduler scheduler): Recebe uma função e produz um Mono que é executado em uma thread paralela.

Módulo 10 - Backpressure

Backpressure

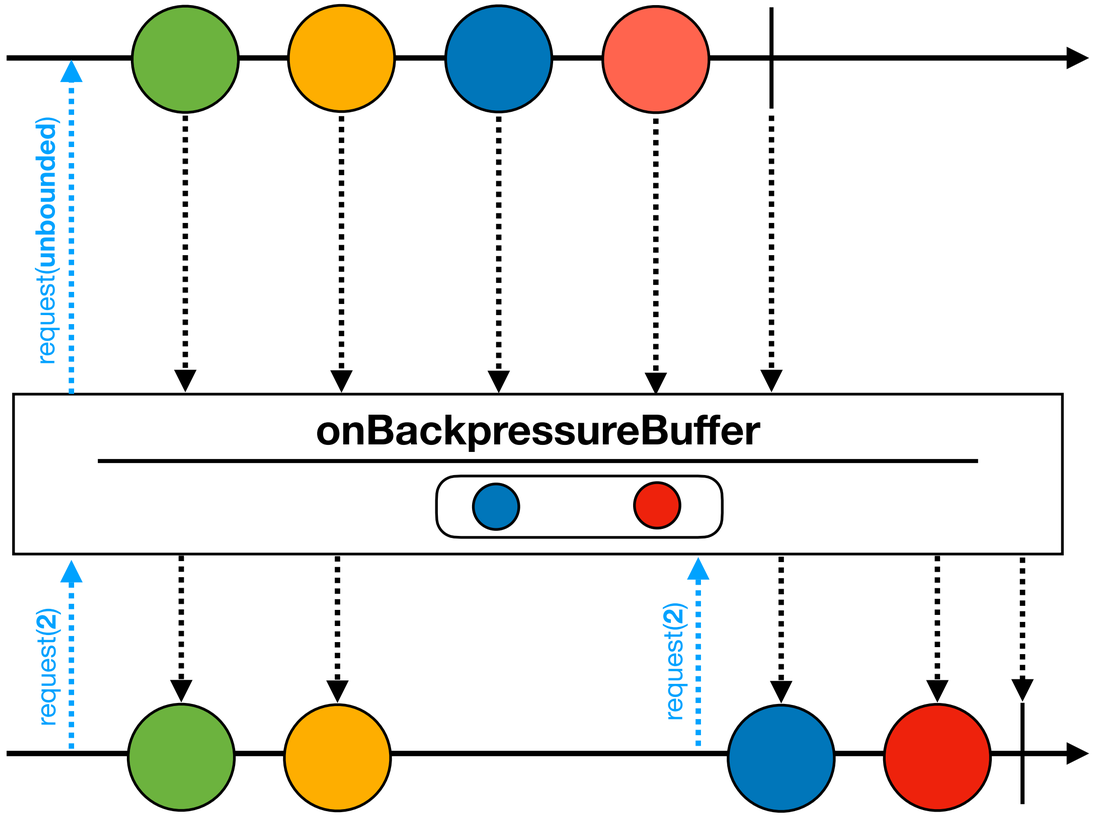


Backpressure

Backpressure é um mecanismo de controle para lidar com a discrepância de velocidade entre a produção e o consumo de dados em fluxos reativos.

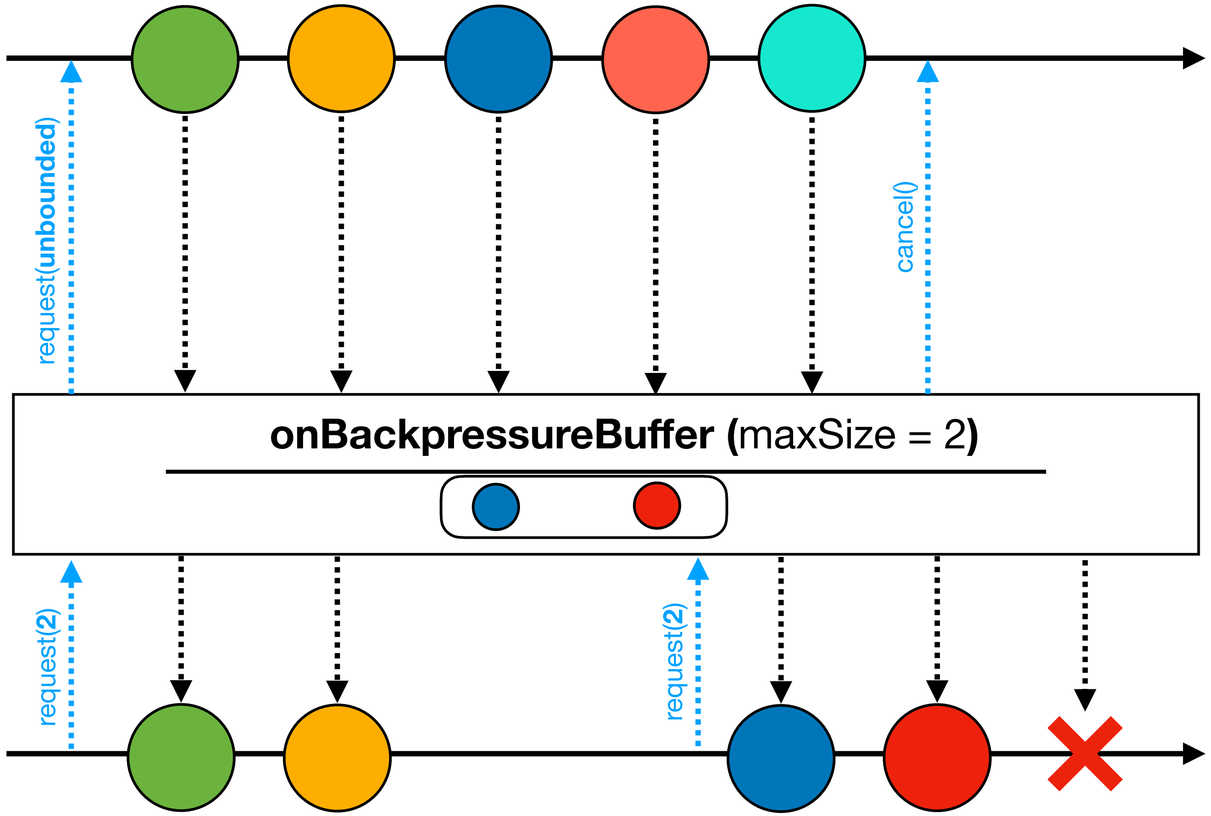
Backpressure strategy - buffer

Armazena os eventos em um buffer. Pode levar ao consumo excessivo de memória se o consumidor não acompanhar o produtor.



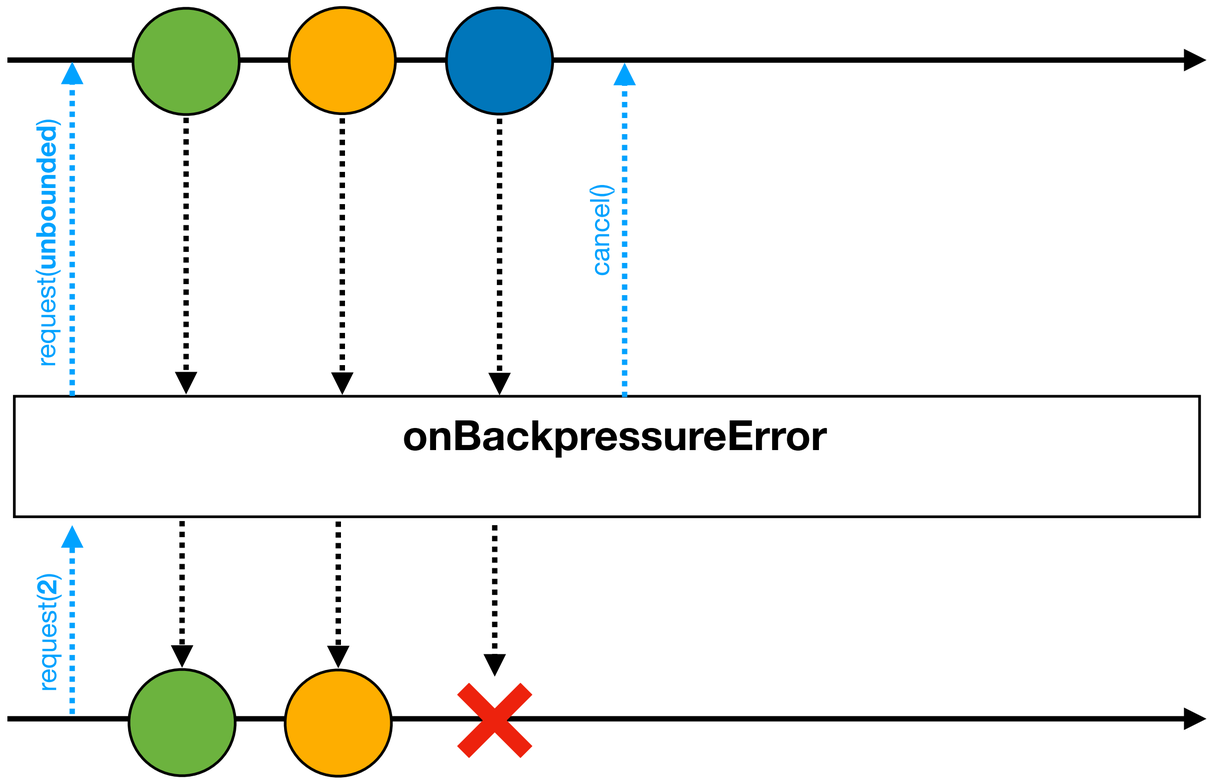
Qual tamanho do buffer?
Limitado a Heap da JVM.

Backpressure strategy - buffer maxSize



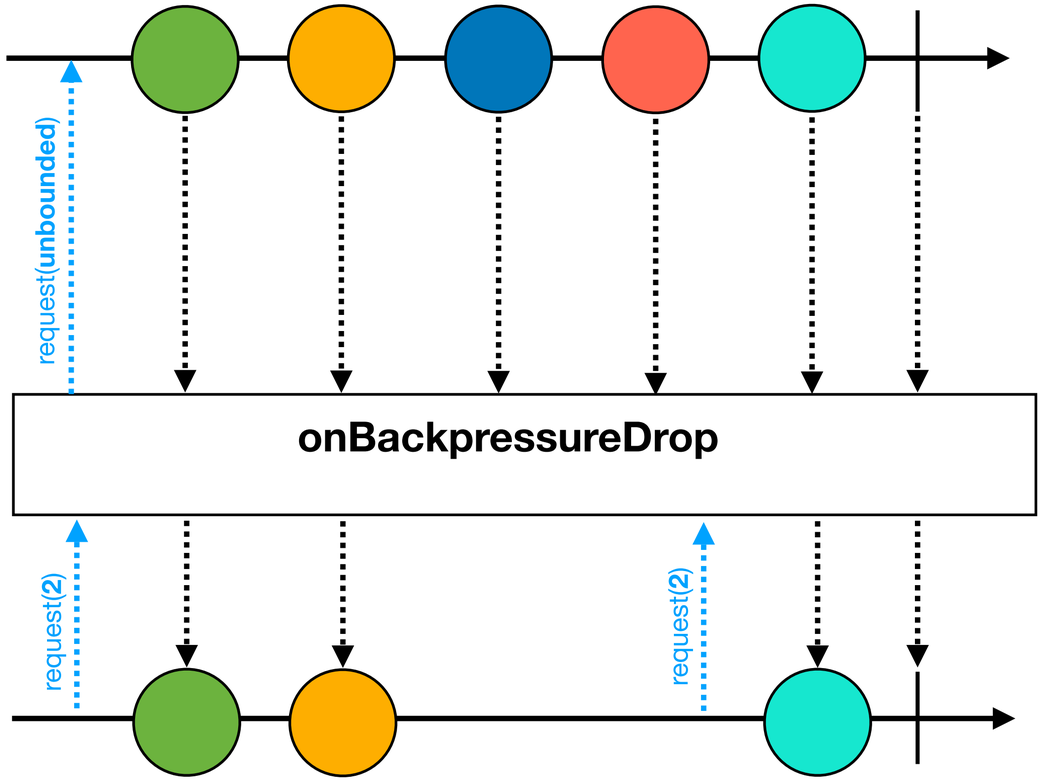
Backpressure strategy - error

Gera um erro quando há mais itens do que o assinante pode processar.



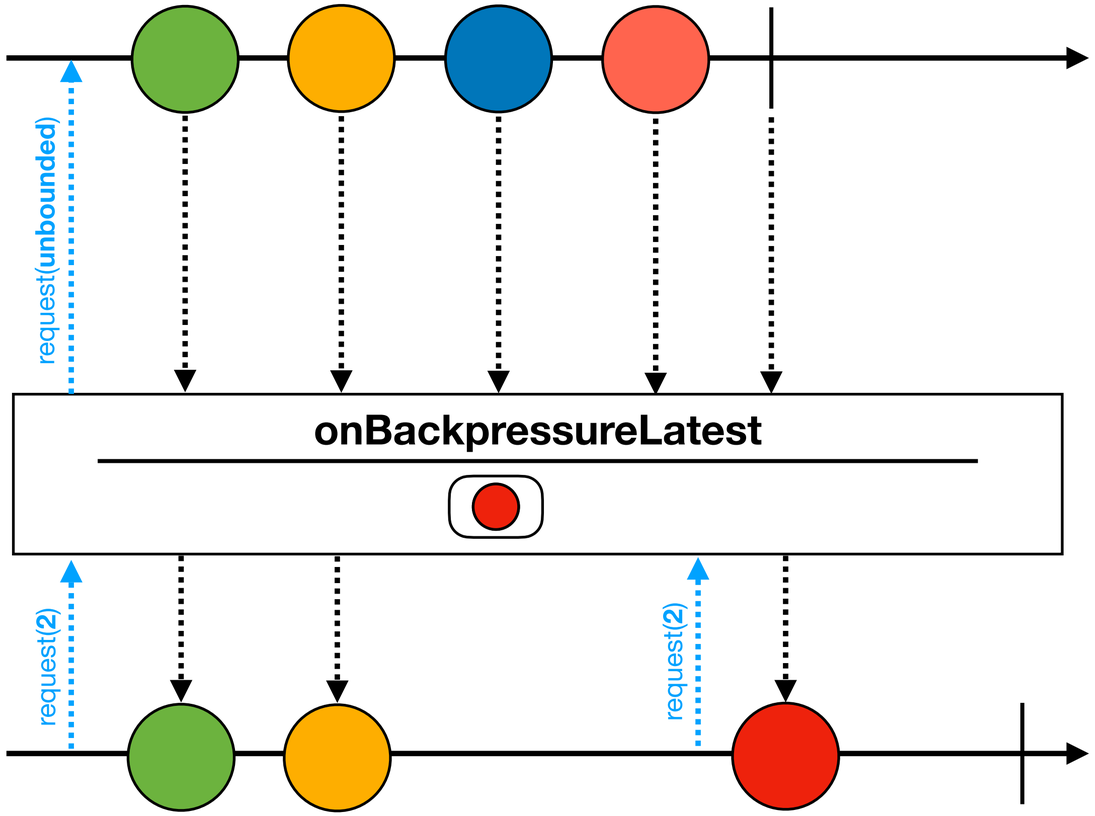
Backpressure strategy - drop

Descarta os itens extras quando o subscriber não consegue acompanhar o fluxo.



Backpressure strategy - latest

Mantém apenas o elemento mais recente no buffer quando cheio, descartando os elementos mais antigos.



Resumo

Backpressure é um mecanismo de controle para lidar com a discrepância de velocidade

entre a produção e o consumo de dados em fluxos reativos.

onBackpressureBuffer(): Armazena os elementos em um buffer temporariamente quando não é solicitado demanda suficiente no downstream.

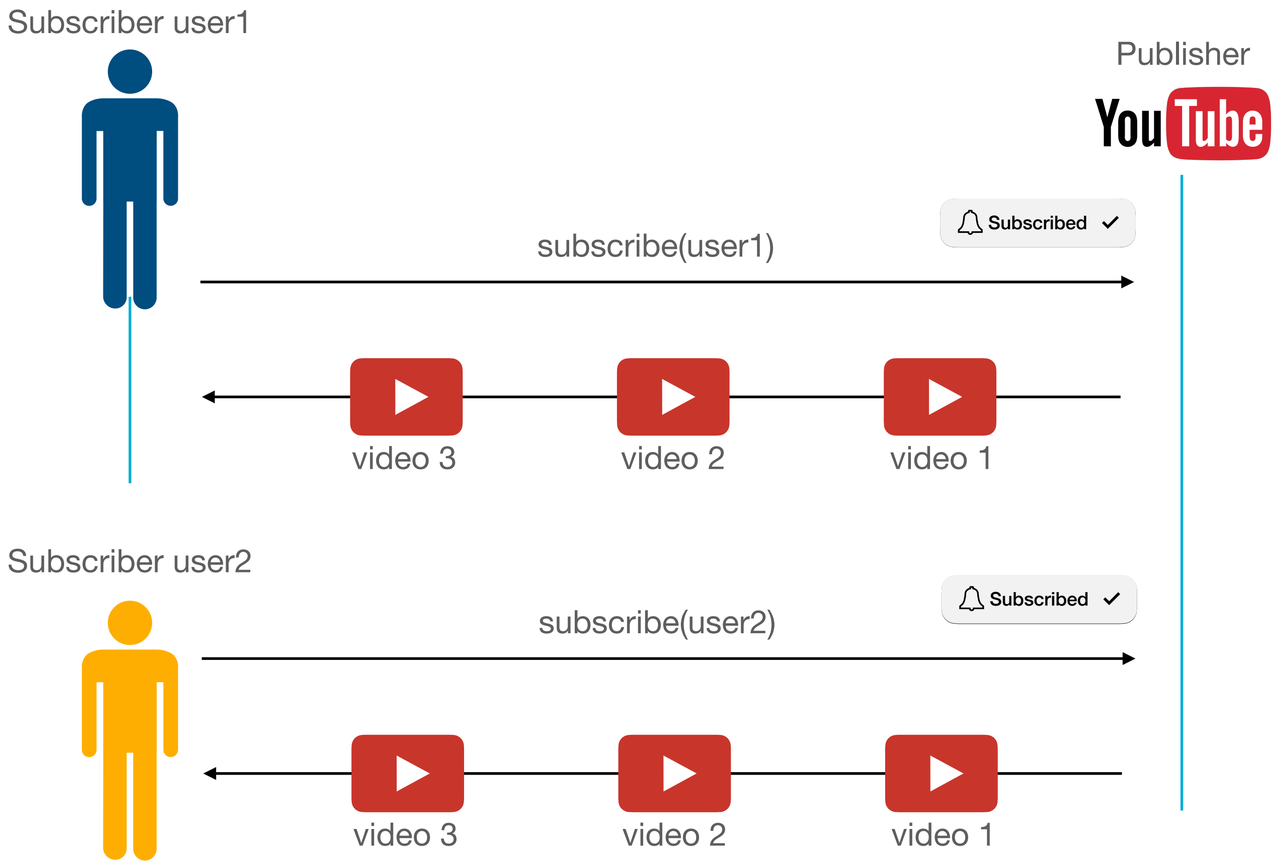
onBackpressureError(): Gera um erro de backpressure imediatamente quando não é solicitado demanda suficiente no downstream.

onBackpressureDrop(): Descarta elementos quando não é solicitado demanda suficiente no downstream.

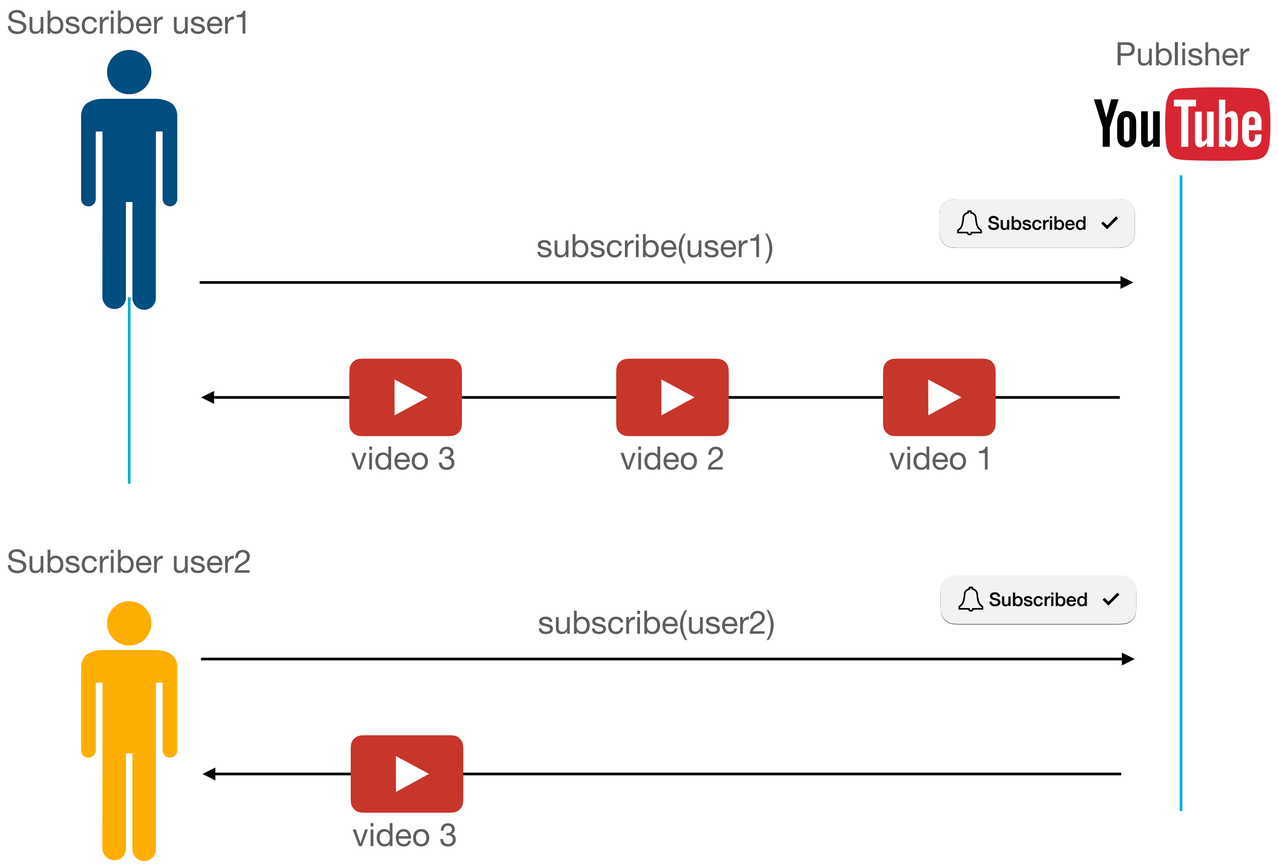
onBackpressureLatest(): Mantém apenas o último elemento emitido no buffer quando não é solicitado demanda suficiente no downstream, permitindo que o mais recente substitua os anteriores.

Módulo 11 - Hot e cold publisher

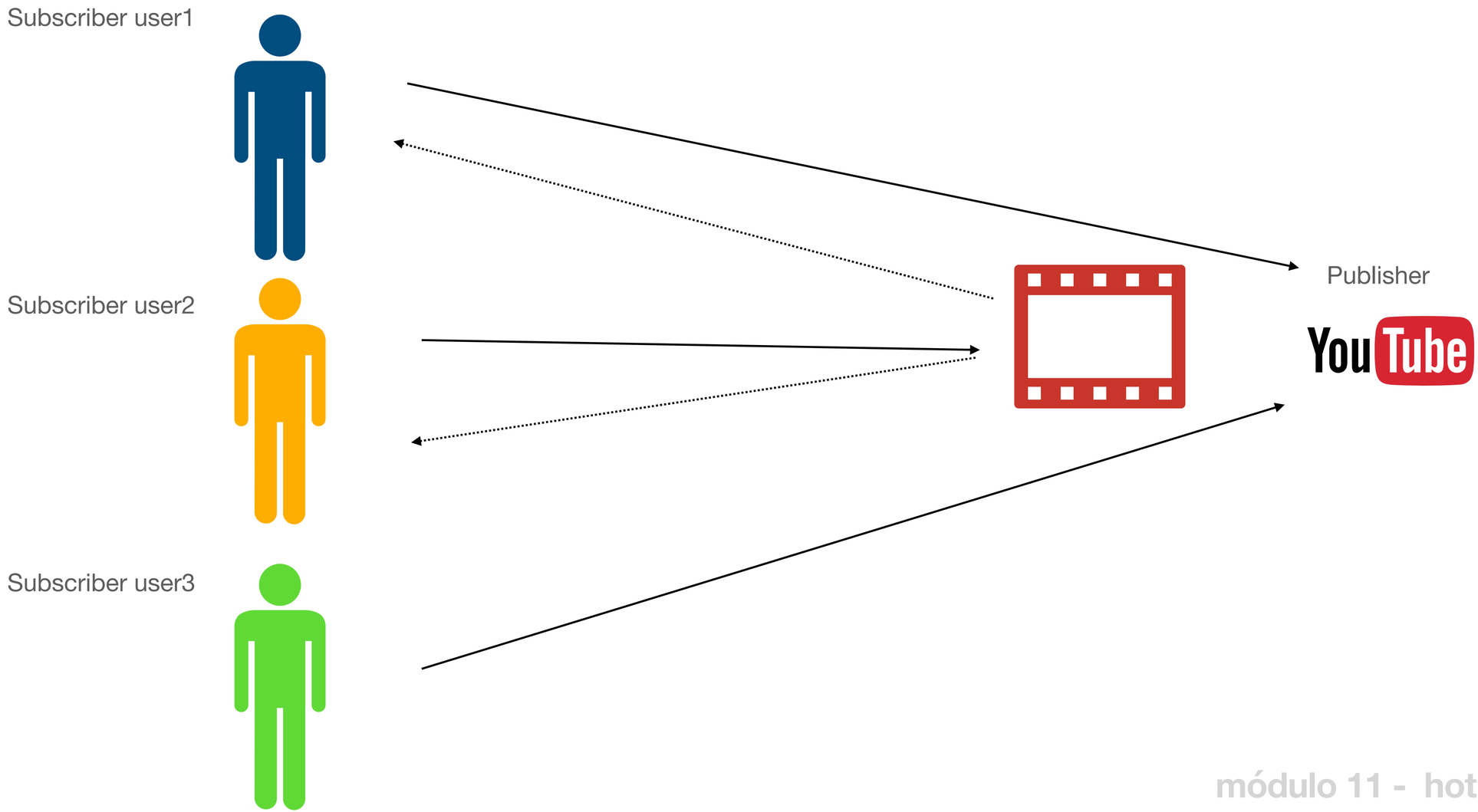
Cold publisher



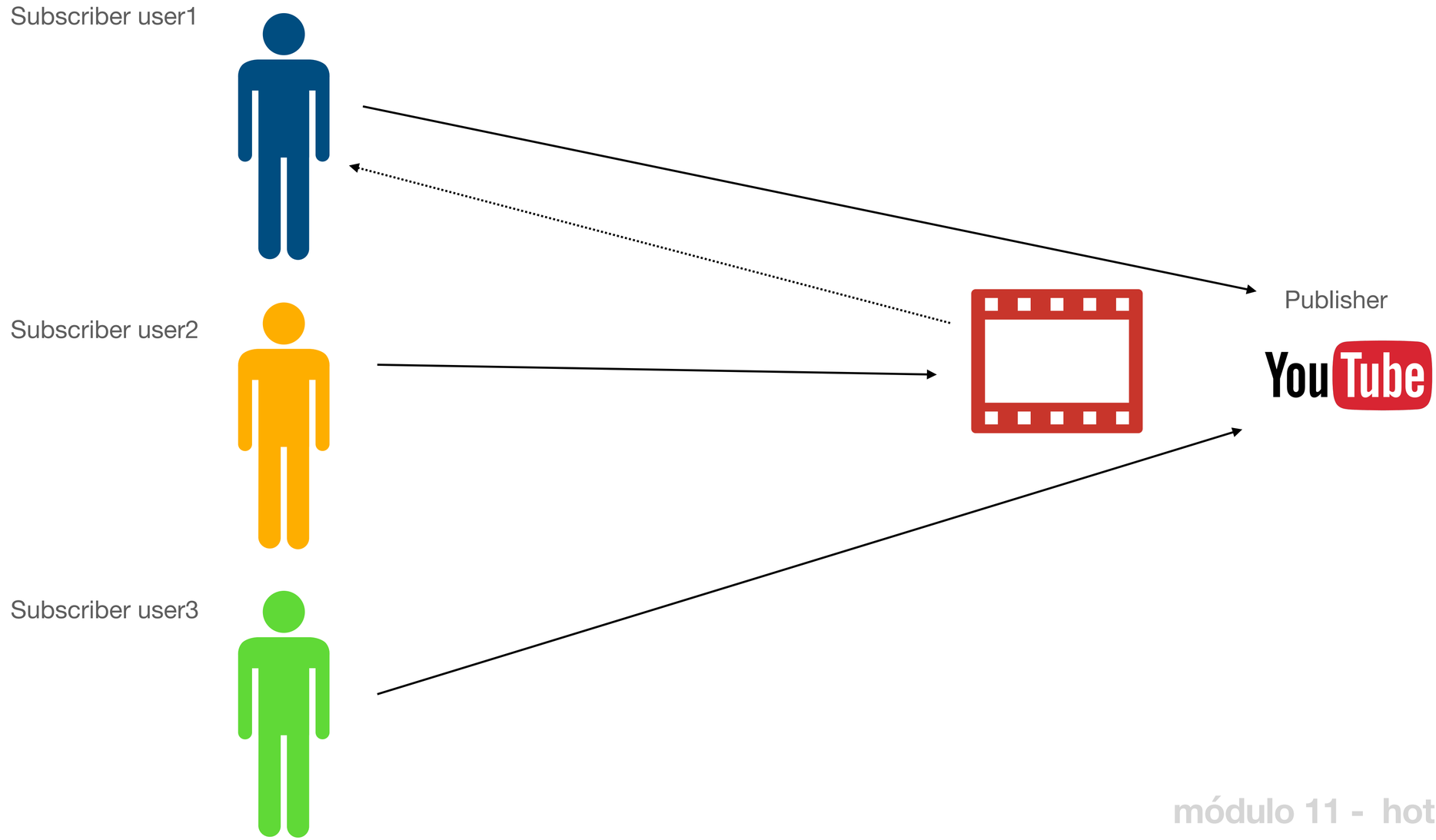
Hot publisher



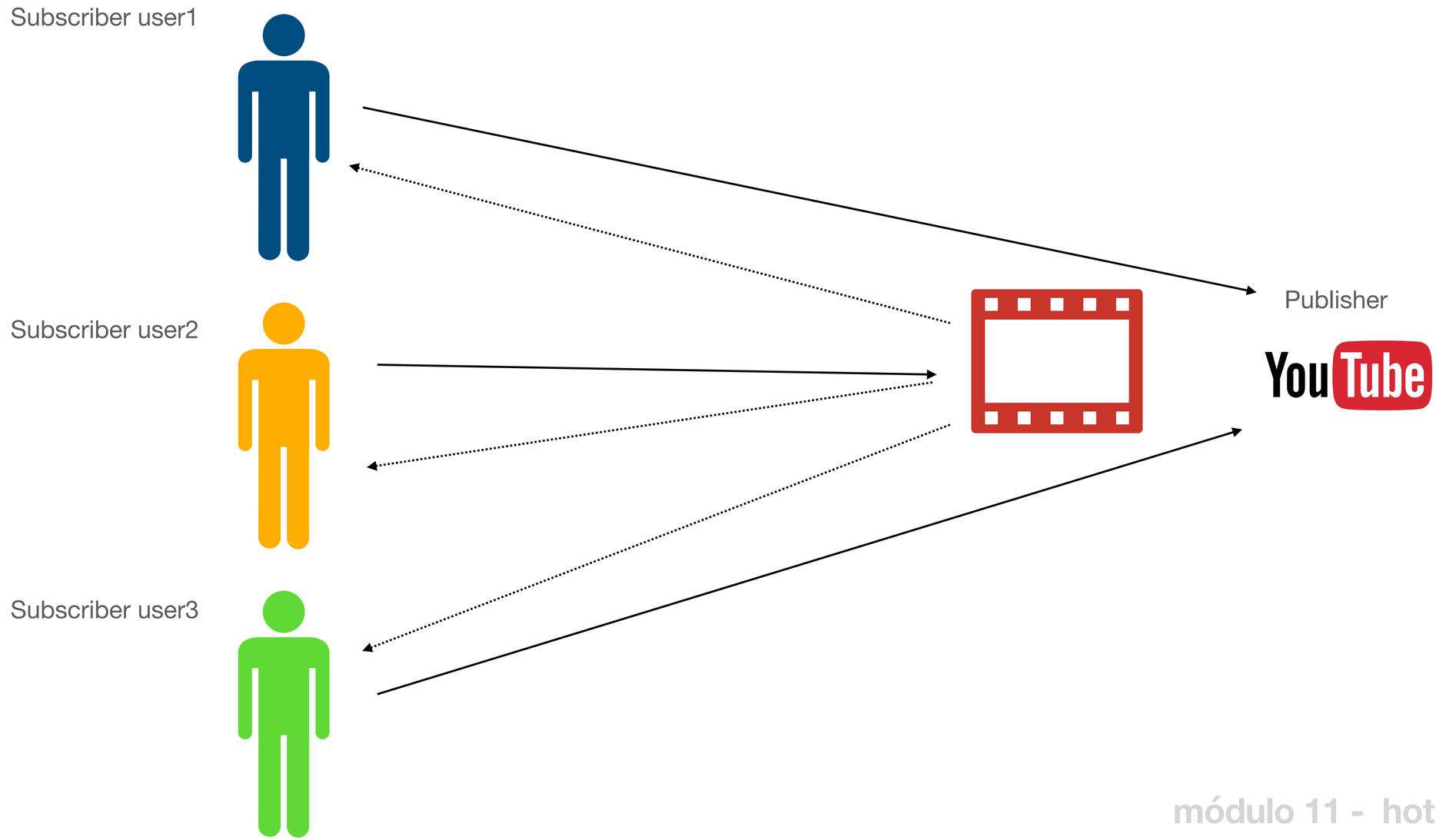
Hot publisher - autoConnect()



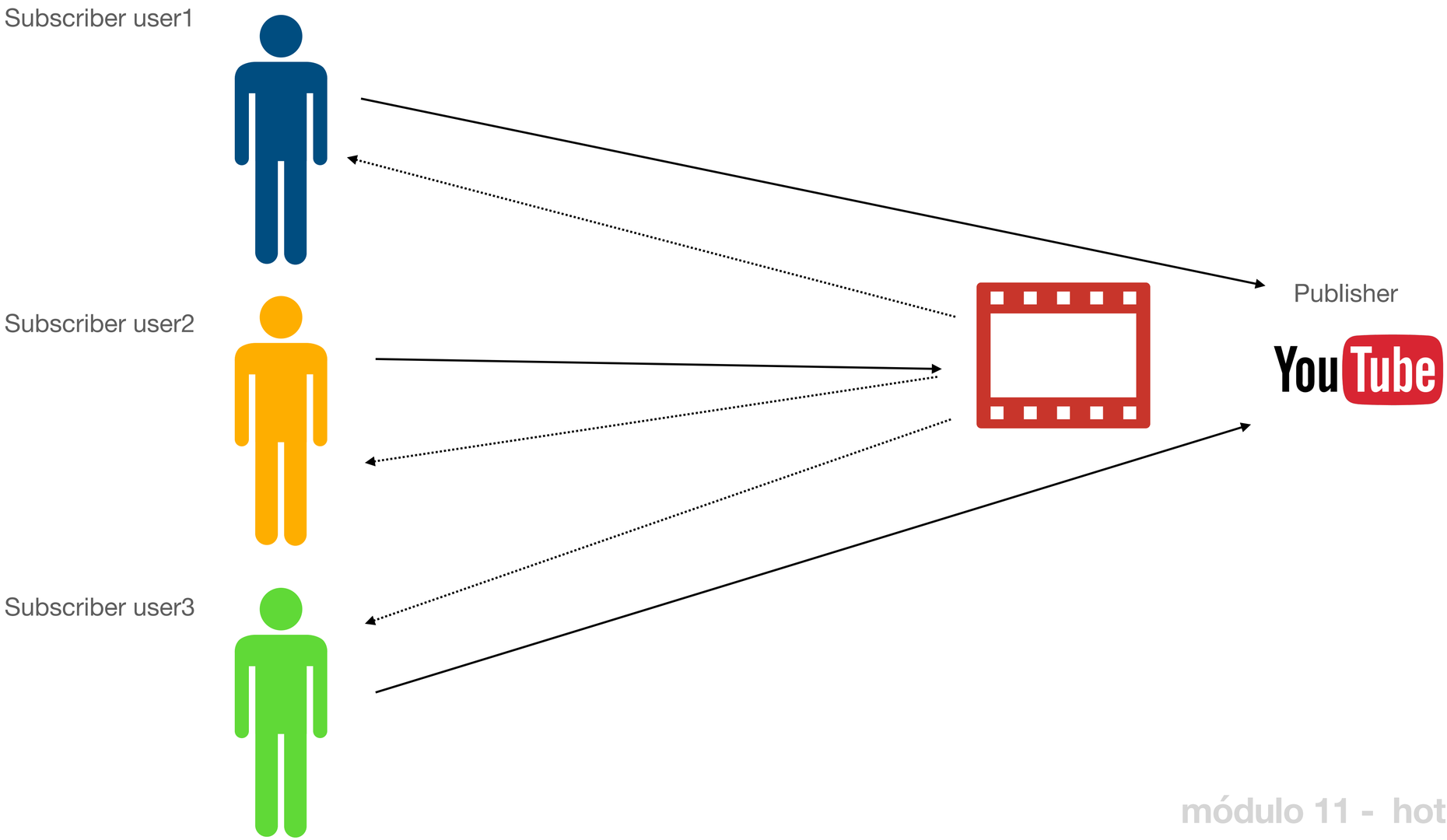
Hot publisher - autoConnect(0)



Hot publisher - autoConnect(2)



Hot publisher - share() ou publish().refCount(n)



Resumo

tipo	método	quantidade mínima de subscriptions para iniciar	depois de terminar, pode recomeçar por uma nova subscription?
cold	-	1	sim
hot	autoConnect()	1	não
hot	autoConnect(N)	N >= 0	não
hot	share()	1	sim
hot	refCount(N)	N > 1	sim