



ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Introducing Reactive Streams semantics

Programming Reactive Systems

Konrad Malawski, Julien Richard-Foy

Flow-control and Reactive Streams

In the previous sections we already talked about asynchronous systems.

We did omit one very important point that raises in such systems: the need of *back-pressure* (or *flow-control*).

Back-pressure / Flow control

Flow control is a mechanism to control the *rate* at which an upstream publisher is emitting data to a downstream subscriber in proportion to the subscribers's receiving capabilities.

Motivating example

In general the need of back-pressure can be exemplified in any asynchronous system in which the upstream is producing faster than the downstream consumer is able to process it.

You will start noticing this everywhere once you become aware of it;

These situations happen when:

- ▶ adding elements to queues asynchronously
- ▶ issuing HTTP requests to a slow service,
- ▶ or even submitting your coding tasks to the grading system during these courses (!).

Back-pressure / Flow control

The motivating example consists of any pair of such stages that:

- ▶ the “upstream” produces data at a *fast rate*
- ▶ the “downstream” consumes data at a *slow rate*

```
val slowConsumer = context.spawn(SlowConsumer)
val fastProducer = context.spawn(FastProducer(slowConsumer))
```

Reactive Streams

The Reactive Streams initiative provides a standard for exchanging streams of data.

*Reactive Streams is an initiative to provide a standard for **asynchronous** stream processing with **non-blocking back-pressure**.*

Reactive Streams components

<http://www.reactive-streams.org>

Reactive Streams was initially designed and developed in Java, however their core abstractions are language agnostic.

Reactive Streams define:

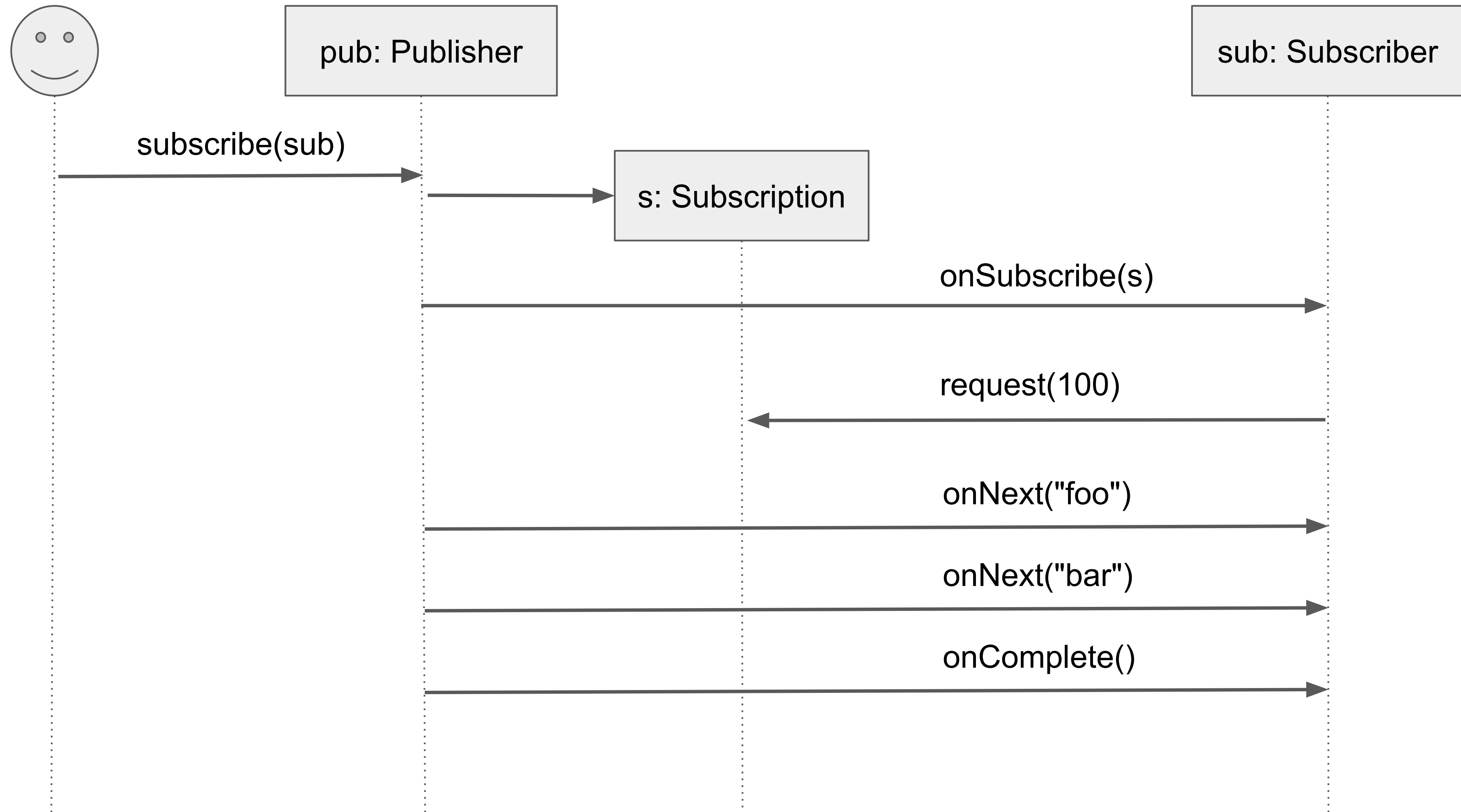
- ▶ a set of interaction Rules (the *Specification*),
- ▶ a set of types (the *SPI*),
- ▶ a Technology Compliance (test) Kit (*TCK*).

Reactive Streams ecosystem

Reactive Streams have since:

- ▶ part of the Java standard, since Java 9 (`java.util.concurrent.Flow`)
- ▶ been ported to .NET () Reactive Streams are part of Java SE since Java 9, and have been ported to .NET
- ▶ as well as heavily inspired the GenStage library in Elixir.

Canonical Example



Reactive Streams methods as Signals (Messages)

Methods' return type is `void` (in Scala: `Unit`), this is the same idea here as with Actor messaging – lack of return type enables us, and leads us to, building the system as asynchronous signals.


By forcing the signatures to be `... => Unit` we allow implementations to dispatch the work asynchronously as quickly as possible, without enforcing much overhead (except for checking validity of the call (i.e. `nulls` are not allowed)).

Reactive Streams Specification & TCK

SPECIFICATION

1. Publisher (Code)

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

ID	Rule
1	The total number of <code>onNext</code> 's signalled by a <code>Publisher</code> to a <code>Subscriber</code> MUST be less than or equal to the total number of elements requested by that <code>Subscriber</code> 's <code>Subscription</code> at all times.
	<i>The intent of this rule is to make it clear that Publishers cannot signal more elements than Subscribers have requested. There's an implicit, but important, consequence to this rule: Since demand can only be fulfilled after it has been received, there's a happens-before relationship between requesting elements and receiving elements.</i>
2	A <code>Publisher</code> MAY signal fewer <code>onNext</code> than requested and terminate the <code>Subscription</code> by calling <code>onComplete</code> or <code>onError</code> .
	<i>The intent of this rule is to make it clear that a Publisher cannot guarantee that it will be able to produce the number of</i>

Reactive Streams Specification

And freely available TCK test suite: Reactive Streams TCK

Challenge: Resource Efficiency

The Reactive Streams *implementors* are responsible of managing resources.

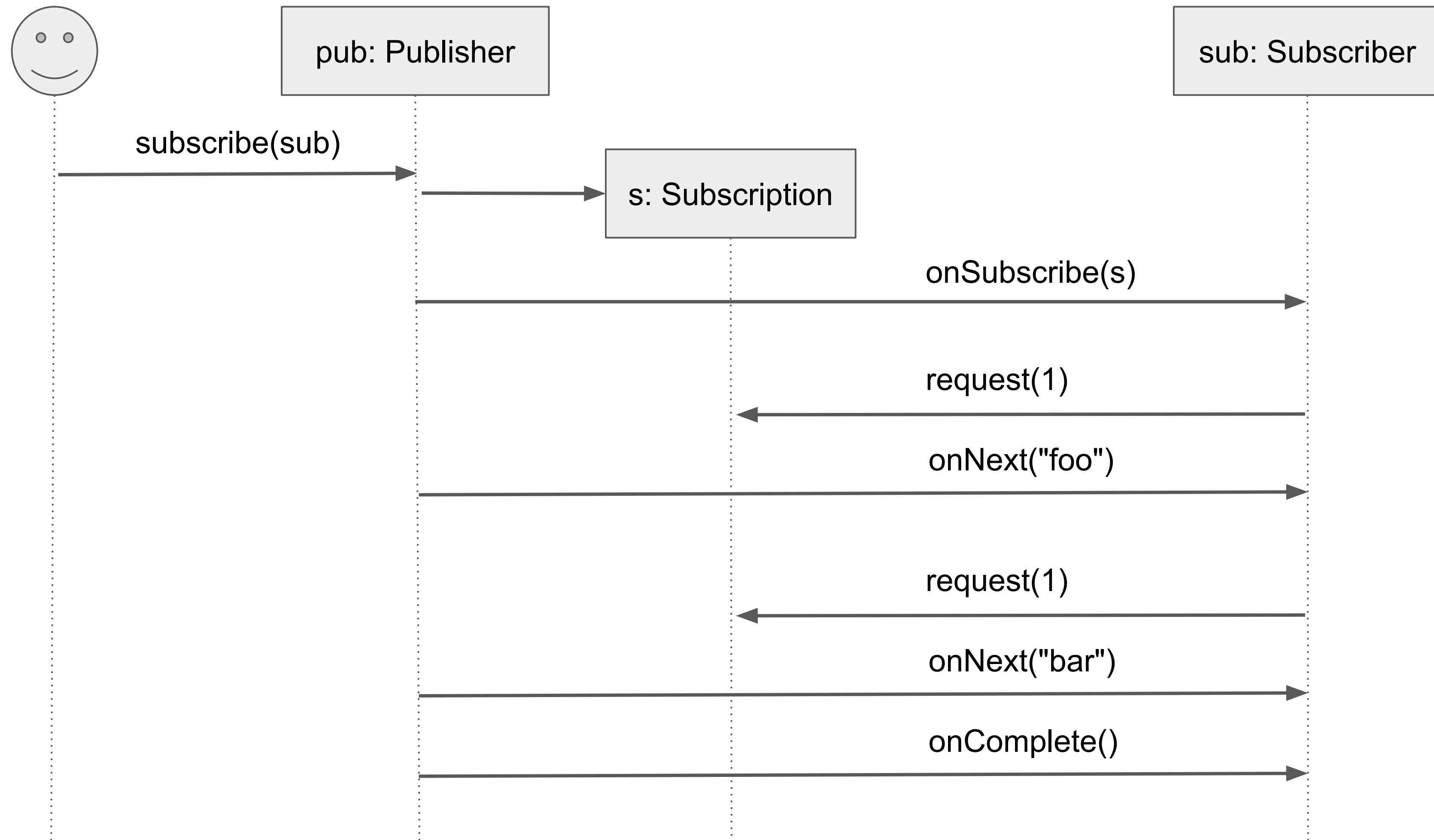
Nevertheless:

Care has been taken to mandate fully non-blocking and asynchronous behavior of all aspects of a Reactive Streams implementation

(Reactive Streams specification)

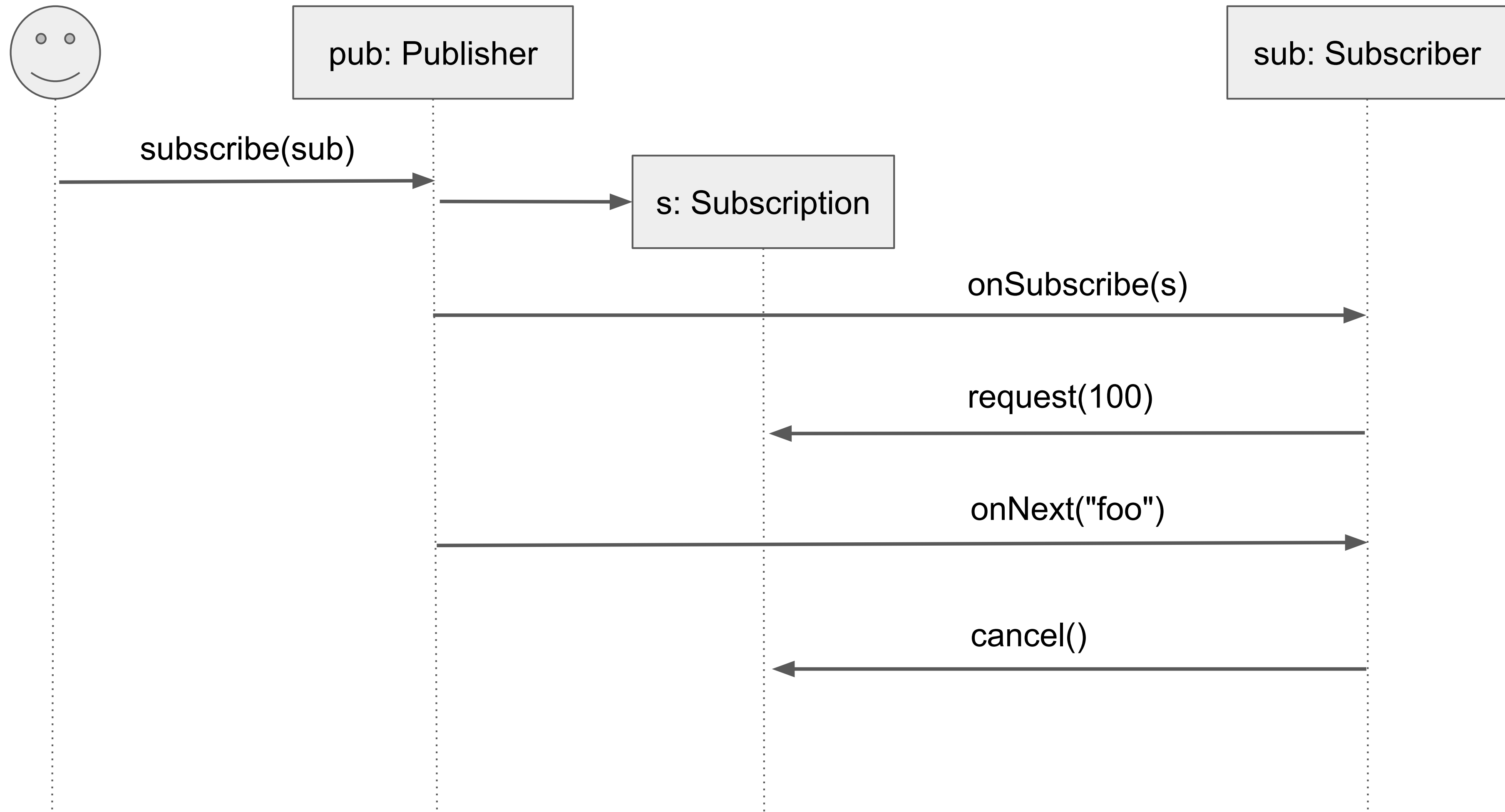
Challenge: Producer And Consumer Synchronization

Flow of data constrained by consumer demand (back pressure)



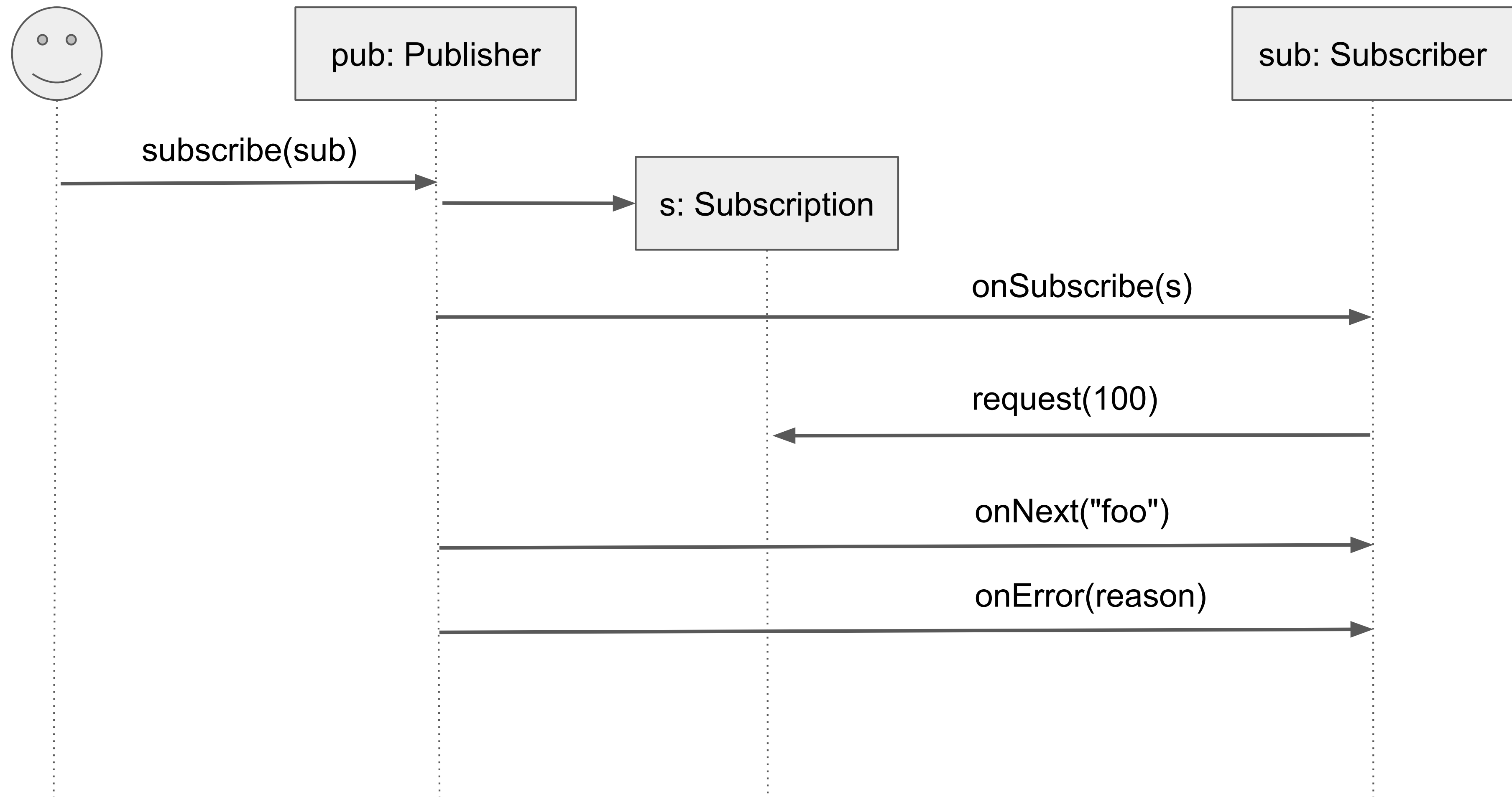
Challenge: Resource Management

The Publisher can release resources when a subscription is cancelled



Challenge: Failure Handling

onError message notifies a consumer about an upstream failure



Challenge: Separating business logic from plumbing

- ▶ Out of scope of Reactive Streams – it is an *SPI*
- ▶ Left to libraries implementing the Reactive Streams protocol

[Reactive Stream's] primary purpose is to define interfaces such that different streaming implementations can interoperate; it is not the purpose of Reactive Streams to describe an end-user API.

(Akka Streams documentation)

Challenge: Interoperability

It is the intention of this specification to allow the creation of many conforming implementations, which by virtue of abiding by the rules will be able to interoperate smoothly, preserving the aforementioned benefits and characteristics across the whole processing graph of a stream application.

Reactive Streams Specification

Summary

- ▶ Reactive Streams is an initiative to provide a standard for asynchronous stream processing with non-blocking back pressure.
- ▶ Reactive Streams mandate a “permit” based way of handling flow-control
- ▶ Akka Streams is one of the many implementations of Reactive Streams

<https://github.com/reactive-streams/reactive-streams-jvm/blob/v1.0.2/README.md#specification>