Demystification Of Reactive Streams

by Vlas Shatokhin

What is a Stream

“You cannot step twice into the same stream. For as you are stepping in, other waters are ever flowing on to you.”
- Heraclitus

A stream is a sequence of objects, which made available **over time** and can be accessed in sequential order.

This sequence potentially has **no beginning and no end**.

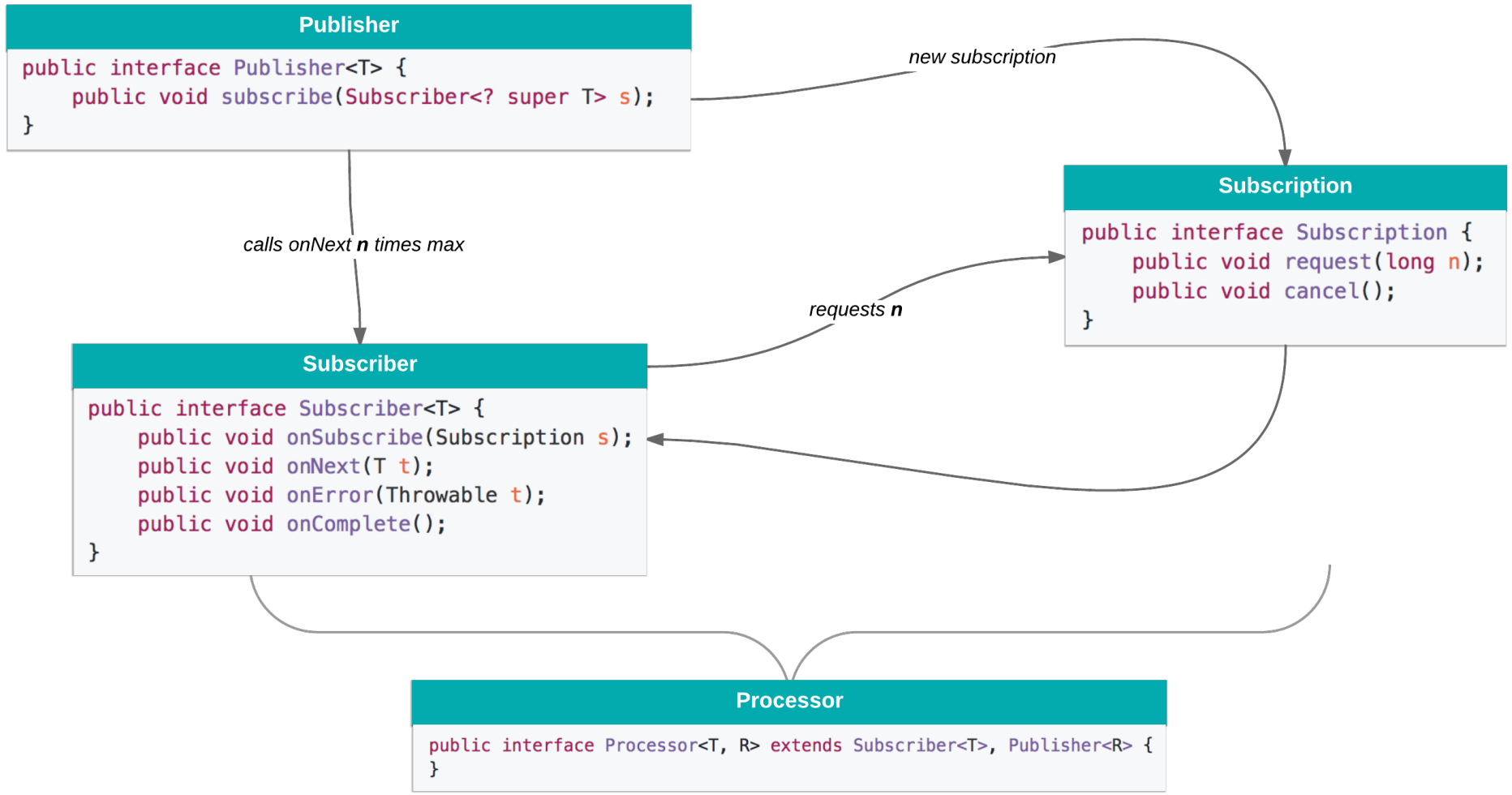
Reactive Streams | Story

- 2009 - .NET's Reactive Extensions
- 2013 - Widely adopted on JVM
  - Play Iteratees – pull back-pressure, clumsy API
  - Akka-IO – NACK back-pressure, low-level IO (bytes)
  - RxJava – no back-pressure, nice API
- Oct 2013 - "reactive non-blocking asynchronous back-pressure"
- 2014-2015 - Reactive Streams spec

Reactive Streams | Goals

- Govern the **exchange** of stream data across an asynchronous boundary
- Allow the creation of many conforming implementations, which by virtue of abiding by the rules will be able to **interoperate** smoothly
- Highlight the benefits of asynchronous processing, which would be negated if the communication of back pressure were synchronous
- Stream manipulations (transformation, splitting, merging, etc.) is **not covered** by specification

Reactive Streams API chart



Looks simple, isn’t it?

Reactive Streams Specification | Implementing

2. Subscriber (Code)

```
public interface Subscriber<T> {
    public void onSubscribe(Subscription s);
    public void onNext(T t);
    public void onError(Throwable t);
    public void onComplete();
}
```

ID	Rule
1	A `Subscriber` MUST signal demand via `Subscription.request(long n)` to receive `onNext` signals.
	The intent of this rule is to establish that it is the responsibility of the Subscriber to signal when, and how many, elements it is able and willing to receive.
2	If a `Subscriber` suspects that its processing of signals will negatively impact its `Publisher` 's responsivity, it is RECOMMENDED that it asynchronously dispatches its signals.
	The intent of this rule is that a Subscriber should not obstruct the progress of the Publisher from an execution point-of-view. In other words, the Subscriber should not starve the Publisher from CPU cycles.
3	`Subscriber.onComplete()` and `Subscriber.onError(Throwable t)` MUST NOT call any methods on the `Subscription` or the `Publisher` .
	The intent of this rule is to prevent cycles and race-conditions—between Publisher, Subsription and Subscriber—during the processing of completion signals.
4	`Subscriber.onComplete()` and `Subscriber.onError(Throwable t)` MUST consider the Subscription cancelled after having received the signal.
	The intent of this rule is to make sure that Subscribers respect a Publisher's terminal state signals. A Subscription is simply not valid anymore after an onComplete or onError signal has been received.
5	A `Subscriber` MUST call `Subscription.cancel()` on the given `Subscription` after an `onSubscribe` signal if it already has an active `Subscription` .
	The intent of this rule is to prevent that two, or more, separate Publishers from thinking that they can interact with the same Subscriber. Enforcing this rule means that resource leaks are prevented since extra Subscriptions will be cancelled.
6	A `Subscriber` MUST call `Subscription.cancel()` if it is no longer valid to the `Publisher` without the `Publisher` having signaled `onError` or `onComplete` .
	The intent of this rule is to establish that Subscribers cannot just throw Subscriptions away when they are no longer needed, they have to call `cancel` so that resources held by that Subscription can be safely, and timely, reclaimed.
7	A `Subscriber` MUST ensure that all calls on its `Subscription` take place from the same thread or provide for respective external synchronization.
	The intent of this rule is to establish that external synchronization must be added if a Subscriber will be using a Subscription concurrently by two or more threads.
8	A `Subscriber` MUST be prepared to receive one or more `onNext` signals after having called `Subscription.cancel()` if there are still requested elements pending [see 3.12]. `Subscription.cancel()` does not guarantee to perform the underlying cleaning operations immediately.
	The intent of this rule is to highlight that there may be a delay between calling `cancel` the Publisher seeing that.
9	A `Subscriber` MUST be prepared to receive an `onComplete` signal with or without a preceding `Subscription.request(long n)` call.
	The intent of this rule is to establish that completion is unrelated to the demand flow—this allows for streams which complete early, and obviates the need to poll for completion.
10	A `Subscriber` MUST be prepared to receive an `onError` signal with or without a preceding `Subscription.request(long n)` call.
	The intent of this rule is to establish that Publisher failures may be completely unrelated to signalled demand. This means that Subscribers do not need to poll to find out if the Publisher will not be able to fulfill its requests.
11	A `Subscriber` MUST make sure that all calls on its signal methods happen-before the processing of the respective signals. I.e. the Subscriber must take care of properly publishing the signal to its processing logic.
	The intent of this rule is to establish that it is the responsibility of the Subscriber implementation to make sure that asynchronous processing of its signals are thread safe. See JMM definition of Happens-Before in section 17.4.5.
12	`Subscriber.onSubscribe` MUST be called at most once for a given `Subscriber` (based on object equality).
	The intent of this rule is to establish that it MUST be assumed that the same Subscriber can only be subscribed at most once. Note that `object equality` is `a.equals(b)` .
13	Calling `onSubscribe` . `onNext` . `onError` or `onComplete` MUST return normally except when any provided parameter is `null` in which case it MUST throw a `java.lang.NullPointerException` to the caller, for all other situations the rais 17 A `Subscription` MUST support an unbounded number of calls to `request` and MUST support a demand up to 2^63-1 (`java.lang.Long.MAX_VALUE`). A demand equal or greater than 2^63-1 (`java.lang.Long.MAX_VALUE`) MAY be considered by the `Publisher` as "effectively unbounded". *The intent of this rule is to establish that the Subscriber can request an unbounded number of elements, in any increment above 0 [see 3:9], in any number of invocations of `request` . As it is not feasibly reachable with current or foreseen hardware within a reasonable amount of time (1 element per nanosecond would take 292 years) to fulfill a demand of 2^63-1, it is allowed for a Publisher to stop tracking demand beyond this point.*

A `Subscription` is shared by exactly one `Publisher` and one `Subscriber` for the purpose of mediating the data exchange between this pair. This is the reason why the `subscribe()` method does not return the created `Subscription` , but instead returns `void` ; the `Subscription` is only passed to the `Subscriber` via the `onSubscribe` callback.

4.Processor (Code)

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

ID	Rule
1	A `Processor` represents a processing stage—which is both a `Subscriber` and a `Publisher` and MUST obey the contracts of both.
	The intent of this rule is to establish that Processors behave, and are bound by, both the Publisher and Subscriber specifications.
2	A `Processor` MAY choose to recover an `onError` signal. If it chooses to do so, it MUST consider the `Subscription` cancelled, otherwise it MUST propagate the `onError` signal to its Subscribers immediately.
	The intent of this rule is to inform that it's possible for implementations to be more than simple transformations.

While not mandated, it can be a good idea to cancel a `Processors` upstream `Subscription` when/if its last `Subscriber` cancels their `Subscription` , to let the cancellation signal propagate upstream.

3. Subscription (Code)

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

ID	Rule
1	`Subscription.request` and `Subscription.cancel` MUST only be called inside of its `Subscriber` context.
	The intent of this rule is to establish that a Subscription represents the unique relationship between a Subscriber and a Publisher [see 2.12]. The Subscriber is in control over when elements are requested and when more elements are no longer needed.
2	The `Subscription` MUST allow the `Subscriber` to call `Subscription.request` synchronously from within `onNext` or `onSubscribe` .
	The intent of this rule is to make it clear that implementations of `request` must be reentrant, to avoid stack overflows in the case of mutual recursion between `request` and `onNext` (and eventually `onComplete` / `onError`). This implies that Publishers can be `synchronous` , i.e. signalling `onNext` 's on the thread which calls `request` .
3	`Subscription.request` MUST place an upper bound on possible synchronous recursion between `Publisher` and `Subscriber` .
	The intent of this rule is to complement [see 3.2] by placing an upper limit on the mutual recursion between `request` and `onNext` (and eventually `onComplete` / `onError`). Implementations are RECOMMENDED to limit this mutual recursion to a depth of `1` (ONE)—for the sake of conserving stack space. An example for undesirable synchronous, open recursion would be Subscriber.onNext -> Subscription.request -> Subscriber.onNext -> ..., as it otherwise will result in blowing the calling Thread 's stack.
4	`Subscription.request` SHOULD respect the responsivity of its caller by returning in a timely manner.
	The intent of this rule is to establish that `request` is intended to be a non-obstructing method, and should be as quick to execute as possible on the calling thread, so avoid heavy computations and other things that would stall the caller 's thread of execution.
5	`Subscription.cancel` MUST respect the responsivity of its caller by returning in a timely manner, MUST be idempotent and MUST be thread-safe.
	The intent of this rule is to establish that `cancel` is intended to be a non-obstructing method, and should be as quick to execute as possible on the calling thread, so avoid heavy computations and other things that would stall the caller 's thread of execution. Furthermore, it is also important that it is possible to call it multiple times without any adverse effects.
6	After the `Subscription` is cancelled, additional `Subscription.request(long n)` MUST be NOPs.
	The intent of this rule is to establish a causal relationship between cancellation of a subscription and the subsequent non-operation of requesting more elements.
7	After the `Subscription` is cancelled, additional `Subscription.cancel()` MUST be NOPs.
	The intent of this rule is superseded by 3.5.
8	While the `Subscription` is not cancelled, `Subscription.request(long n)` MUST register the given number of additional elements to be produced to the respective subscriber.
	The intent of this rule is to make sure that `request` -ing is an additive operation, as well as ensuring that a request for elements is delivered to the Publisher.
9	While the `Subscription` is not cancelled, `Subscription.request(long n)` MUST signal `onError` with a `java.lang.IllegalArgumentException` if the argument is <= 0. The cause message MUST include a reference to this rule and/or quote the full rule. *The intent of this rule is to prevent faulty implementations to proceed operation without any exceptions being raised. Requesting a negative or 0 number of elements, since requests are additive, most likely to be the result of an erroneous calculation on the behalf of the Subscriber.*
10	While the `Subscription` is not cancelled, `Subscription.request(long n)` MAY synchronously call `onNext` on this (or other) subscriber(s).
	The intent of this rule is to establish that it is allowed to create synchronous Publishers, i.e. Publishers who execute their logic on the calling thread.
11	While the `Subscription` is not cancelled, `Subscription.request(long n)` MAY synchronously call `onComplete` or `onError` on this (or other) subscriber(s).
	The intent of this rule is to establish that it is allowed to create synchronous Publishers, i.e. Publishers who execute their logic on the calling thread.
12	While the `Subscription` is not cancelled, `Subscription.cancel()` MUST request the `Publisher` to eventually stop signaling its `Subscriber` . The operation is NOT REQUIRED to affect the `Subscription` immediately.
	The intent of this rule is to establish that the desire to cancel a Subscription is eventually respected by the Publisher, acknowledging that it may take some time before the signal is received.
13	While the `Subscription` is not cancelled, `Subscription.cancel()` MUST request the `Publisher` to eventually drop any references to the corresponding subscriber. *The intent of this rule is to make sure that Subsribers can be properly garbage-collected after their subscription no longer being valid. Re-subscribing with the same Subscriber object is discouraged [see 2.12], but this specification does not mandate that it is disallowed since that would mean having to store previously cancelled subscriptions indefinitely.*
14	While the `Subscription` is not cancelled, calling `Subscription.cancel` MAY cause the `Publisher` , if stateful, to transition into the `shut-down` state if no other `Subscription` exists at this point [see 1.9].
	The intent of this rule is to allow for Publishers to signal `onComplete` or `onError` following `onSubscribe` for new Subscribers in response to a cancellation signal from an existing Subscriber.
15	Calling `Subscription.cancel` MUST return normally.
	The intent of this rule is to disallow implementations to throw exceptions in response to `cancel` being called.
16	Calling `Subscription.request` MUST return normally.

SPECIFICATION

1. Publisher (Code)

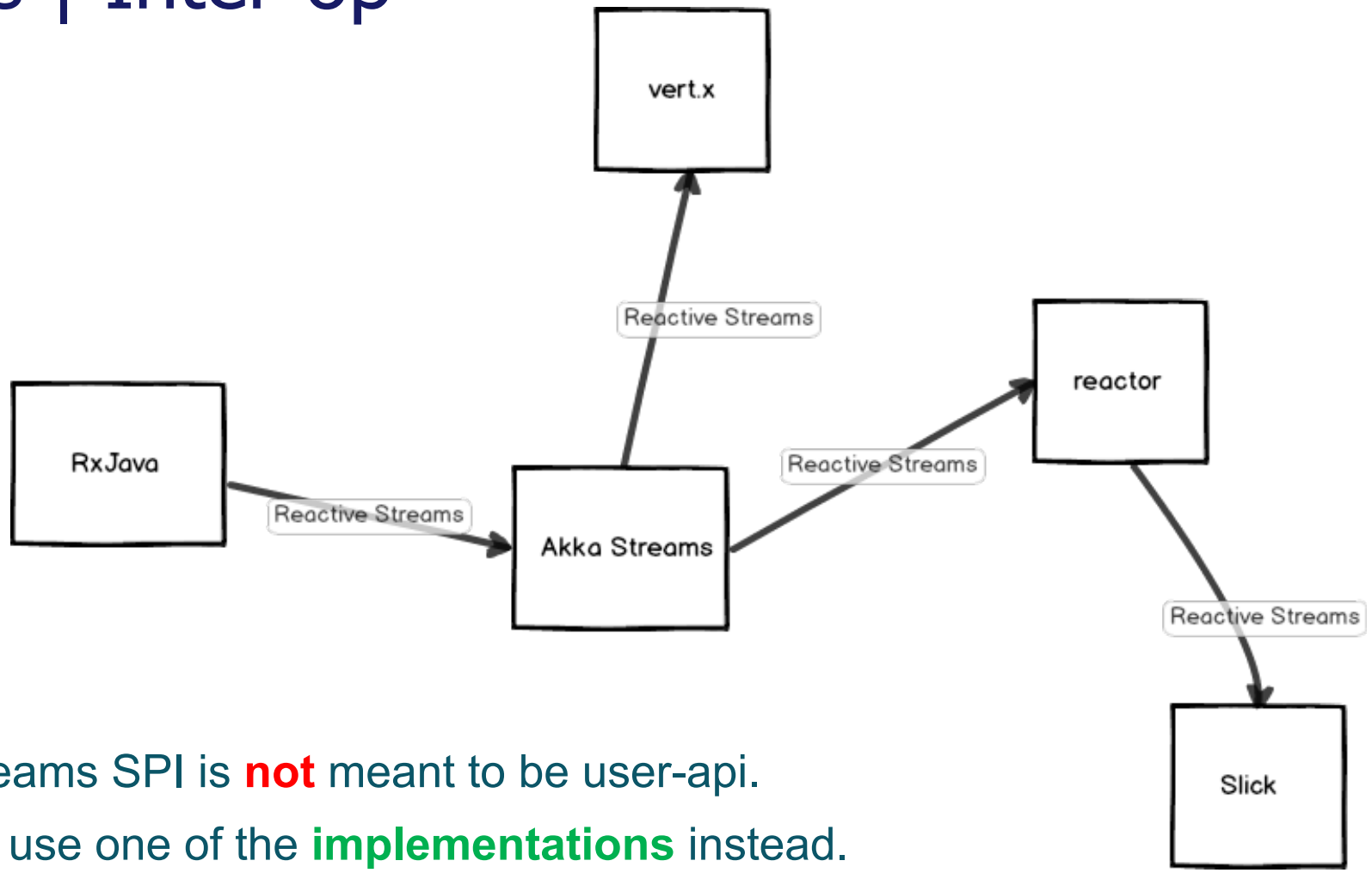
```
public interface Publisher<T> {
    public void subscribe(Subscriber<? super T> s);
}
```

ID	Rule
1	The total number of `onNext` 's signalled by a `Publisher` to a `Subscriber` MUST be less than or equal to the total number of elements requested by that `Subscriber` 's `Subscription` at all times.
	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.
2	A `Publisher` MAY signal fewer `onNext` than requested and terminate the `Subscription` by calling `onComplete` or `onError` .
	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 elements requested; it simply might not be able to produce them all; it may be in a failed state; it may be empty or otherwise already completed.
3	`onSubscribe` , `onNext` , `onError` and `onComplete` signaled to a `Subscriber` MUST be signaled in a `thread-safe` manner—and if performed by multiple threads—use `external synchronization` .
	The intent of this rule is to make it clear that `external synchronization` must be employed if the Publisher intends to send signals from multiple/different threads.
4	If a `Publisher` fails it MUST signal an `onError` .
	The intent of this rule is to make it clear that a Publisher is responsible for notifying its Subscribers if it detects that it cannot proceed—Subscribers must be given a chance to clean up resources or otherwise deal with the Publisher 's failures.
5	If a `Publisher` terminates successfully (finite stream) it MUST signal an `onComplete` .
	The intent of this rule is to make it clear that a Publisher is responsible for notifying its Subscribers that it has reached a terminal state—Subscribers can then act on this information; clean up resources, etc.
6	If a `Publisher` signals either `onError` or `onComplete` on a `Subscriber` , that `Subscriber` 's `Subscription` MUST be considered cancelled.
	The intent of this rule is to make sure that a Subscription is treated the same no matter if it was cancelled, the Publisher signalled onError or onComplete.
7	Once a terminal state has been signaled (`onError` , `onComplete`) it is REQUIRED that no further signals occur.
	The intent of this rule is to make sure that onError and onComplete are the final states of an interaction between a Publisher and Subscriber pair.
8	If a `Subscription` is cancelled its `Subscriber` MUST eventually stop being signaled.
	The intent of this rule is to make sure that Publishers respect a Subscriber's request to cancel a Subscription when Subscription.cancel() has been called. The reason for eventually is because signals can have propagation delay due to being asynchronous.
9	`Publisher.subscribe` MUST call `onSubscribe` on the provided `Subscriber` prior to any other signals to that `Subscriber` and MUST return normally, except when the provided `Subscriber` is `null` in which case it MUST throw a `java.lang.NullPointerException` to the caller, for all other situations the only legal way to signal failure (or reject the `Subscriber`) is by calling `onError` (after calling `onSubscribe`). *The intent of this rule is to make sure that `onSubscribe` is always signalled before any of the other signals, so that initialization logic can be executed by the Subscriber when the signal is received. Also `onSubscribe` MUST only be called at most once, [see 2.12]. If the supplied `Subscriber` is `null` , there is nowhere else to signal this but to the caller, which means a `java.lang.NullPointerException` must be thrown. Examples of possible situations: A stateful Publisher can be overwhelmed, bounded by a finite number of underlying resources, exhausted, or in a terminal state.*
10	`Publisher.subscribe` MAY be called as many times as wanted but MUST be with a different `Subscriber` each time [see 2.12].
	The intent of this rule is to have callers of `subscribe` be aware that a generic Publisher and a generic Subscriber cannot be assumed to support being attached multiple times. Furthermore, it also mandates that the semantics of `subscribe` must be upheld no matter how many times it is called.
11	A `Publisher` MAY support multiple `Subscriber` s and decides whether each `Subscription` is unicast or multicast.
	The intent of this rule is to give Publisher implementations the flexibility to decide how many, if any, Subscribers they will support, and how elements are going to be distributed.

softserve experience matters

github.com/reactive-streams/reactive-streams-jvm/blob/master/README.md#specification

Reactive Streams | Inter-op

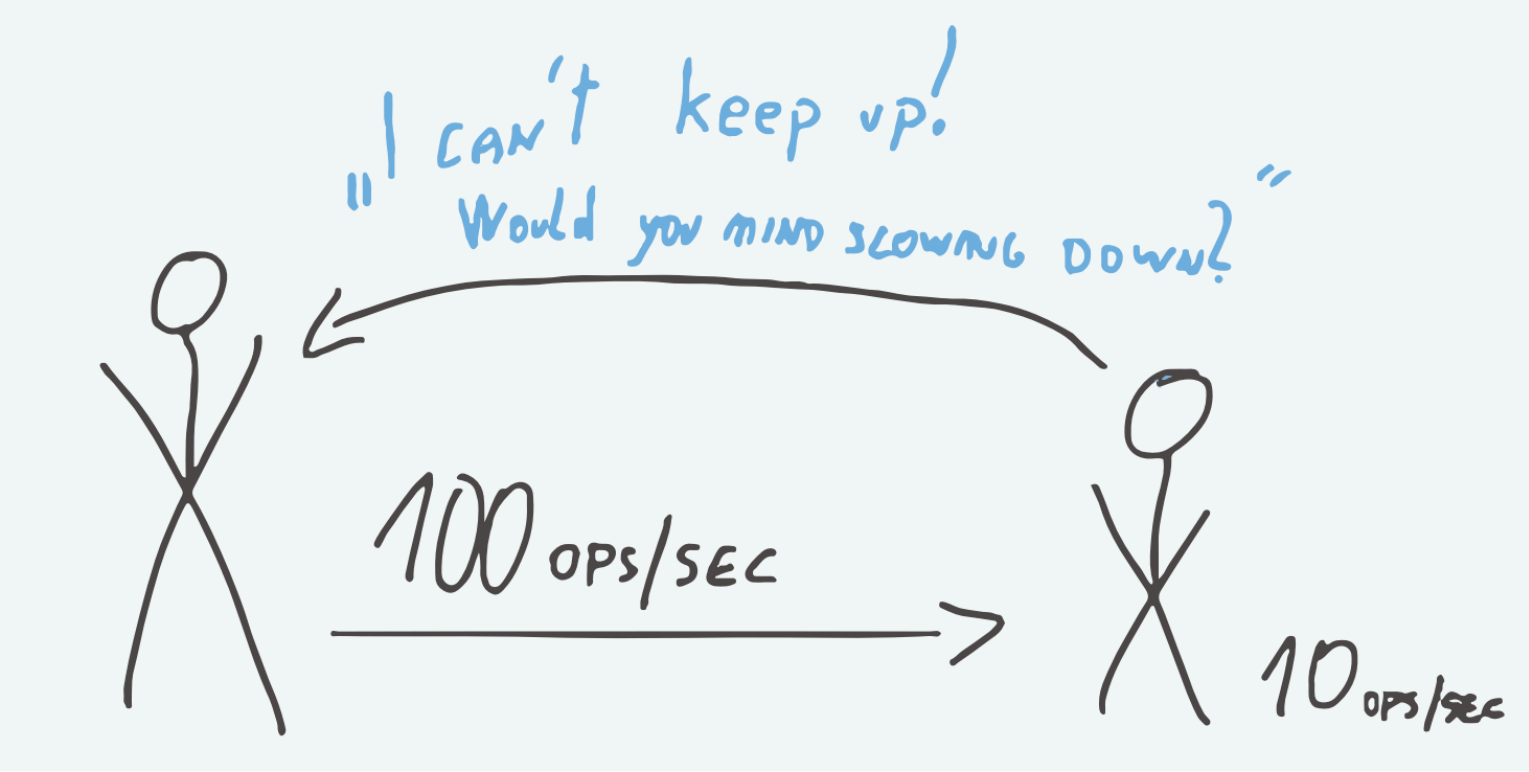


The Reactive Streams SPI is **not** meant to be user-api.

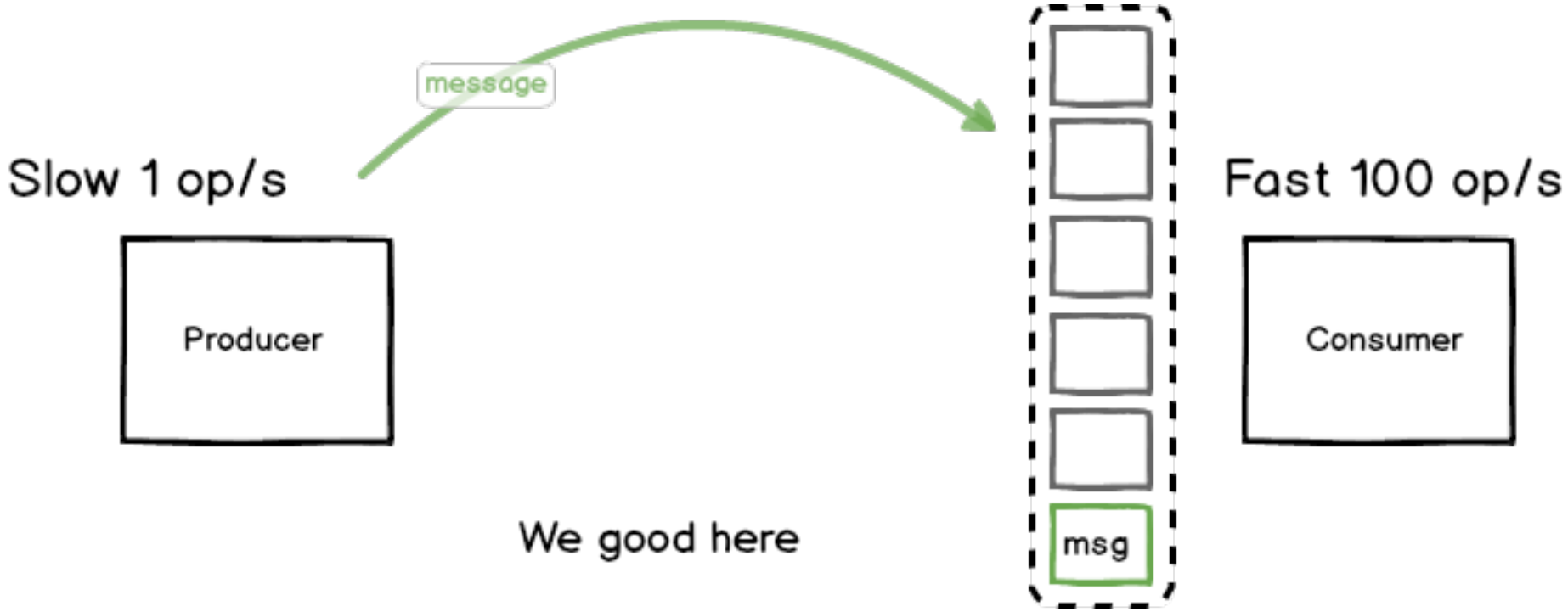
That's why we should use one of the **implementations** instead.

Back-pressure

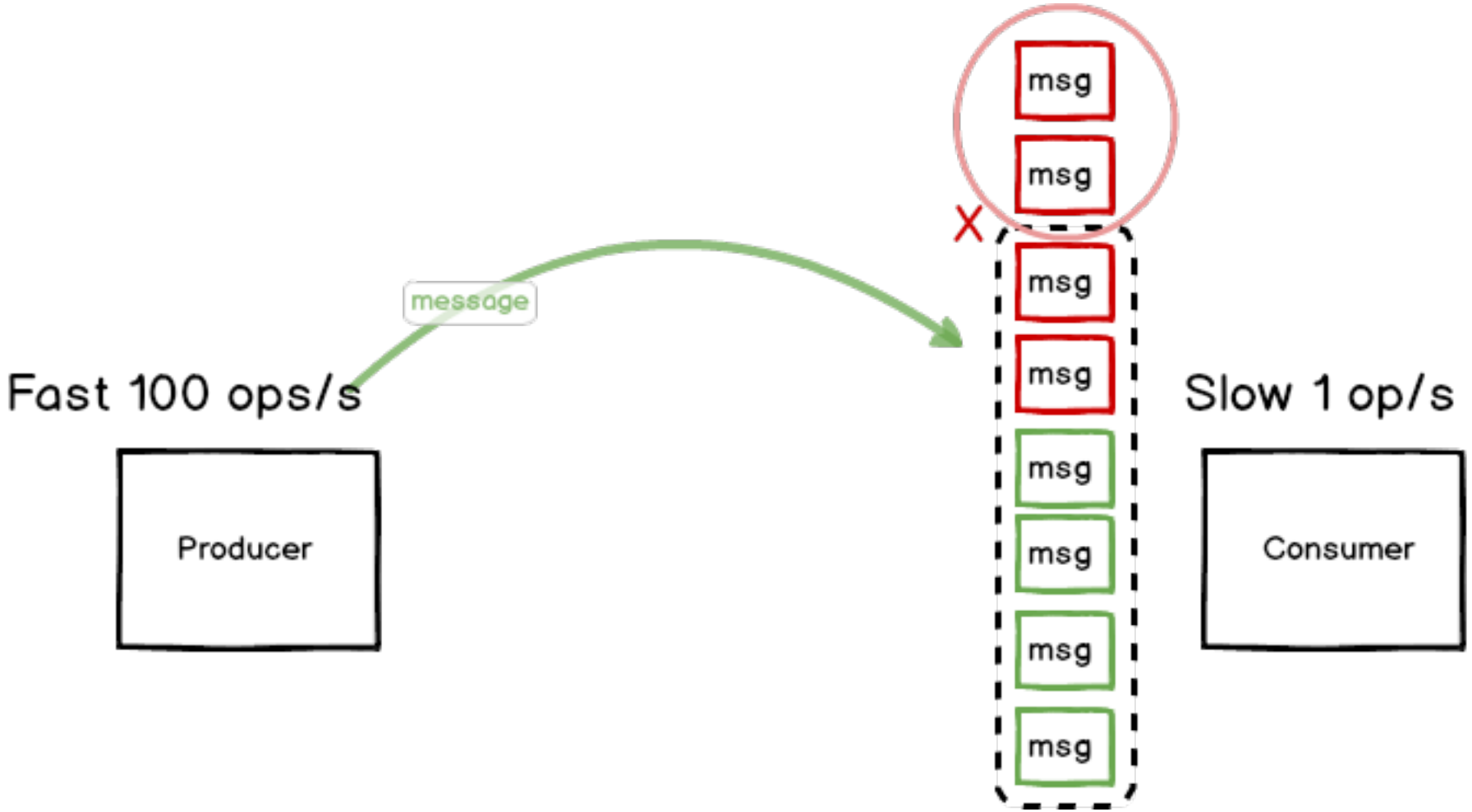
Back-pressure communicates workload levels so that data passing never faces bottlenecks on either side.



Back-pressure | Slow producer, fast consumer

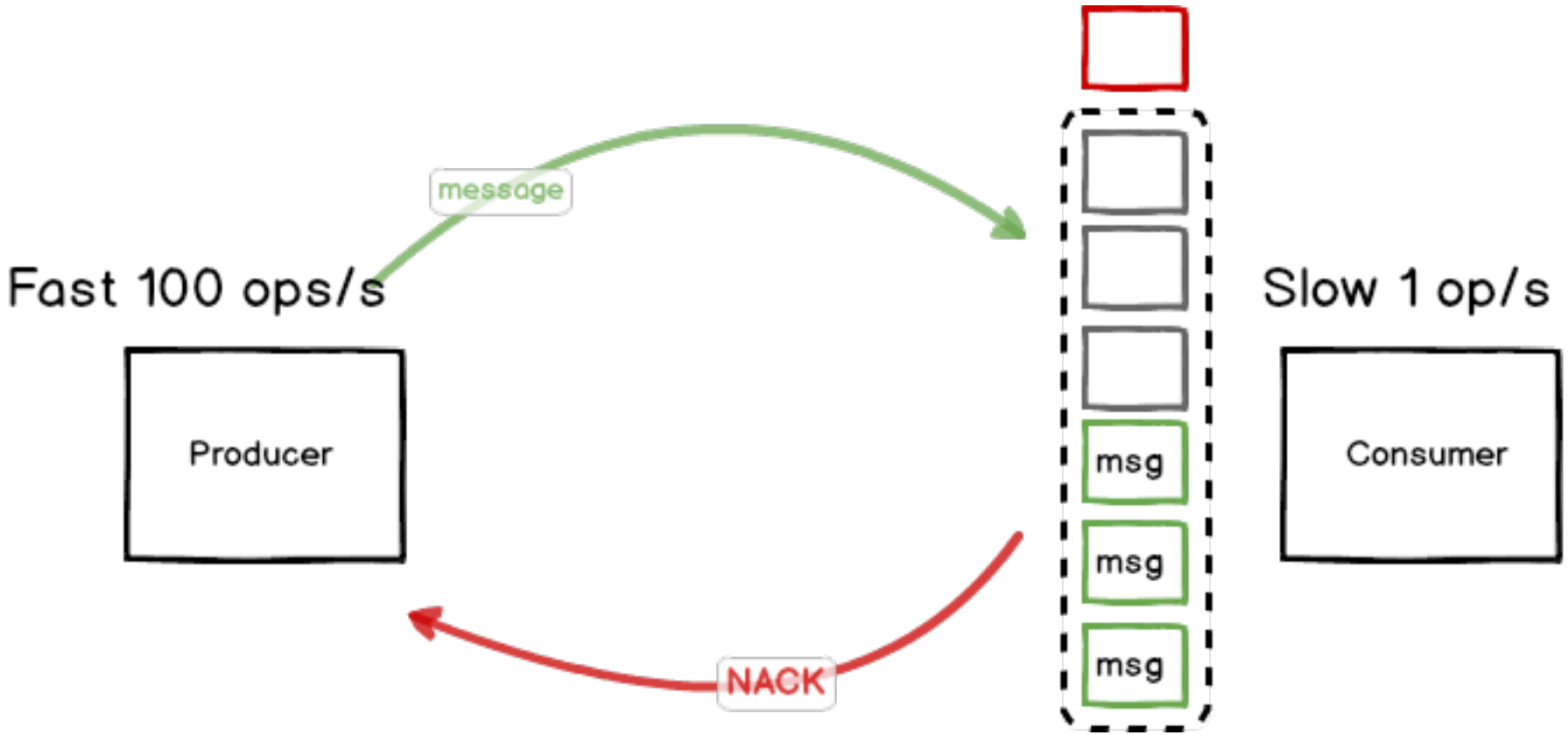


Back-pressure | Fast producer, slow consumer



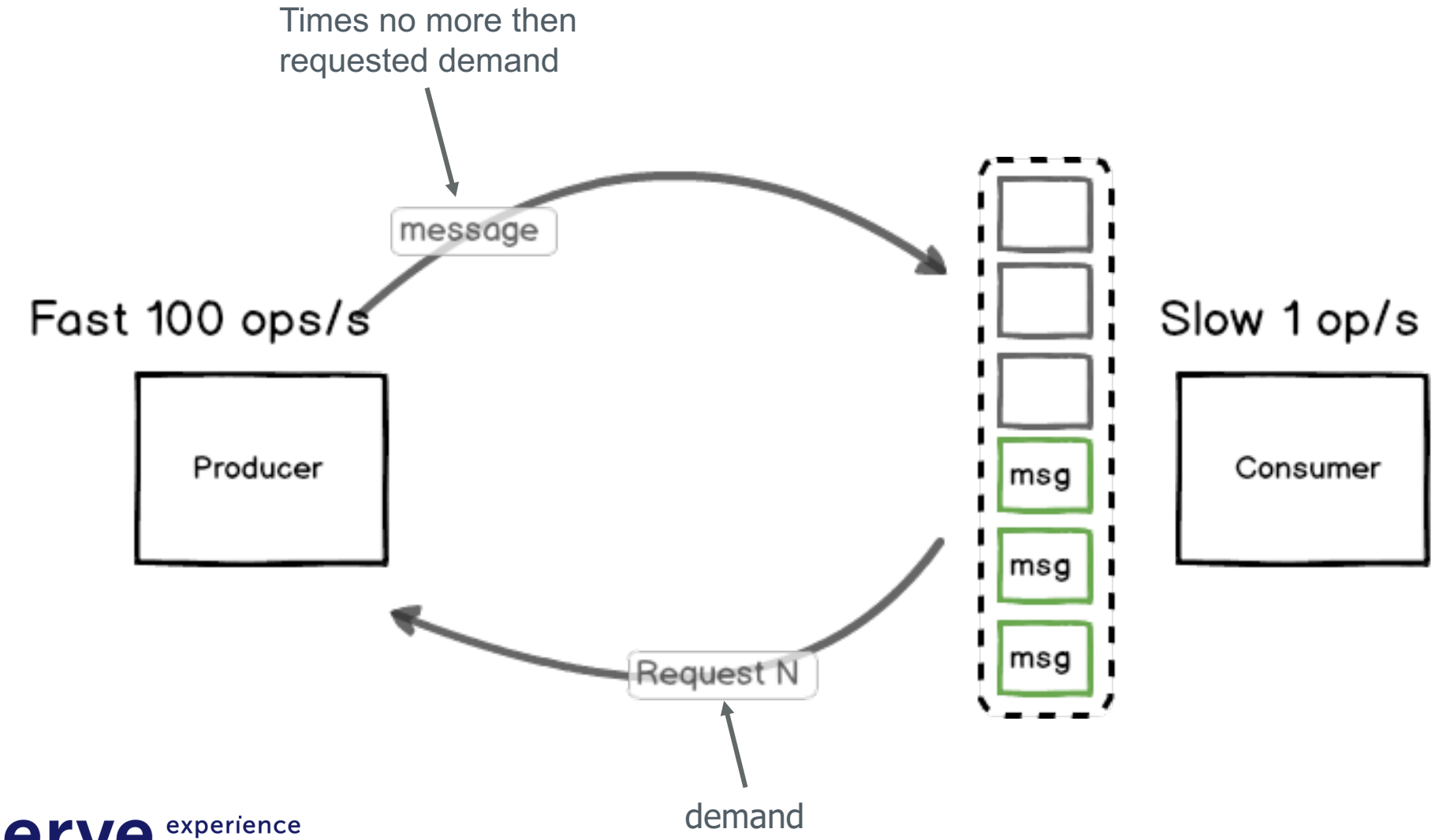
- Drop the messages (data loss)
- Require re-send (+ roundtrips)
- Increase buffer size (good as far as we have some memory)
- Send a NACK..

Back-pressure | NACKing



What if consumer sends a NACK, but there're several messages in flight?

Back-pressure



So Reactive Streams is …

.. a standard and specification for Stream-oriented libraries for the JVM that

- process a potentially unbounded number of elements
- in sequence,
- asynchronously passing elements between components,
- with mandatory non-blocking backpressure.

softserve experience matters

- www.reactive-streams.org
- github.com/reactive-streams/reactive-streams-jvm/blob/v1.0.0/README.md

softserve experience matters

https://github.com/VlasShatokhin/reactive-community-1

Why Akka Streams?

- Drove foundation of Reactive Streams specification
- Seamless integration with Akka HTTP
- Works on top of Akka Actors
- Designed from the ground up for back-pressure
- Provides concise DSL for complex graphs creating
- Encourages reuse of the actual stream blueprints instead of creating them again for every use
- Parallel out of the box - runs every processing step on a different Actor

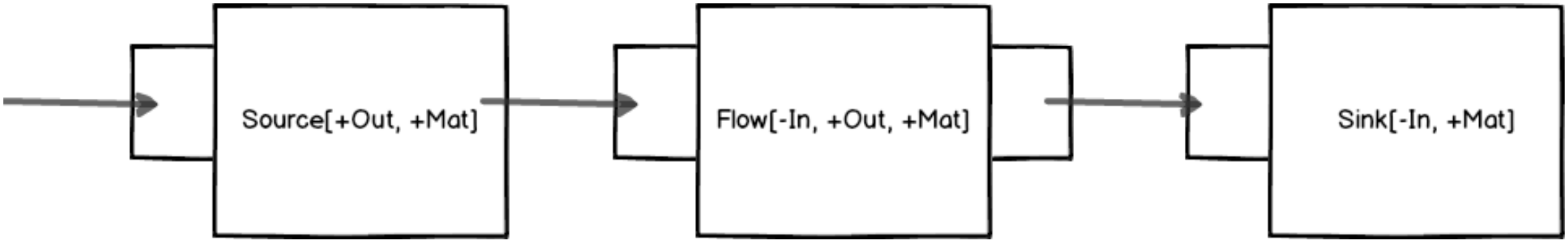
Akka Streams | Components

Demo

Basic components:

- `Source[+Out, +Mat]`
- `Flow[-In, +Out, +Mat]`
- `Sink[-In, +Mat]`

- `RunnableGraph[+Mat]`



Ok now when we saw Akka Stream basics,
can we see where that back-pressure is?

Demo

Akka Streams | Asynchrony and fusion

Each processing step will be running on a separate Actor

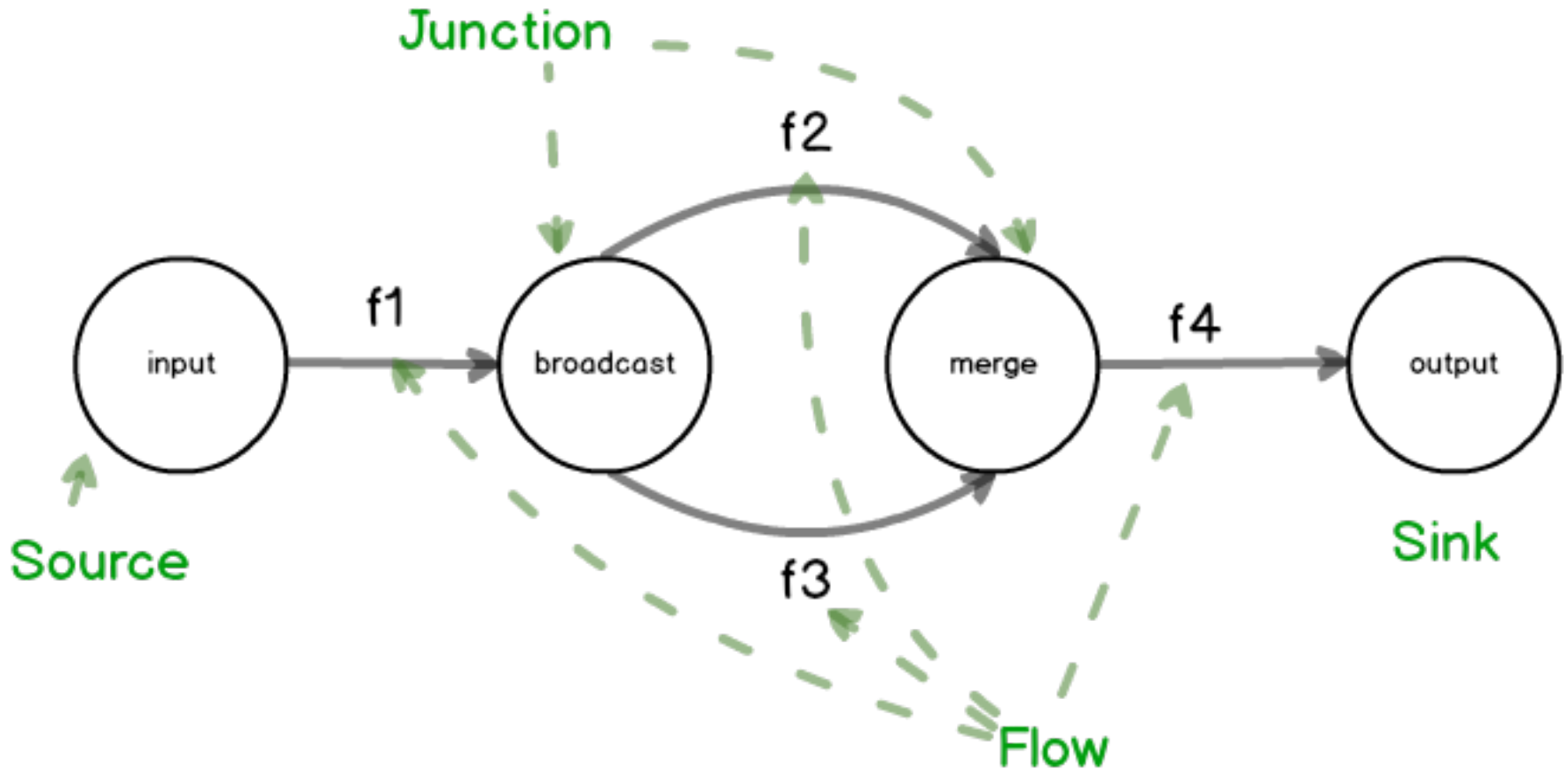
```
val flow = Flow[Int].map(_ * 2).filter(_ > 500)
```

Usually small operations are useful to run within same Actor. Fusion comes to rescue!

```
val fused = Fusing.aggressive(flow)
```

Akka Graphs | Whiteboard

What if we have multiple inputs (merge) / fan-out (broadcast) situation?



softserve experience matters

If only our code would look like our drawings…

Akka Graphs | In code

Demo

One of the goals of the Graph DSL is to look similar to how one would draw a graph on a whiteboard:

```
GraphDSL.create() { implicit builder =>

  val in = Source(1 to 10)
  val out = Sink.foreach(println)

  val bcast = builder.add(Broadcast[Int](2))
  val merge = builder.add(Merge[Int](2))

  val f1, f2, f3, f4 = Flow[Int] map {
    _ + 10
  }

  in ~> f1 ~> broadcast ~> f2 ~> merge ~> f3 ~> out
              broadcast ~> f4 ~> merge

  ClosedShape
}
```

Reactive Streams Implementations

Some of the popular implementations:

- Akka Streams
- Project Reactor
- RxJava
- Vert.x 3

Complete list is here:
http://www.reactive-streams.org/announce-1.0.0

Interoperability of implementations

Demo

Conforming implementations work together easily:

```
// RxScala Publisher
val rxPub: Publisher[Int] = Flowable.fromIterable((1 to 10) asJava)

// Akka Streams Source
val akkaSource: Source[Int, NotUsed]#Repr[String] = Source.fromPublisher(rxPub).map(_.toString)

// Akka Streams Publisher
val akkaPub: Publisher[String] = akkaSource.runWith(Sink.asPublisher(true))

// Reactor Publisher
val reactorPub = Flux.from(akkaPub).map[String](_ + "\n")

reactorPub.subscribe(print(_))
```

Some user-facing classes already implement Reactive Streams role, some of them need be transformed.

References

Readings:

- Official page: www.reactive-streams.org
- Reactive Streams source: github.com/reactive-streams/reactive-streams-jvm/blob/master/README.md
- Akka Streams documentation: doc.akka.io/docs/akka/2.4.17/scala/stream/
- Lightbend activator templates: github.com/typesafehub/activator-akka-stream-scala/tree/master/src/main/scala/sample/stream
- Reactive Programming vs Reactive Systems: www.lightbend.com/reactive-programming-versus-reactive-systems

Videos:

Great talks on Reactive Streams with Akka by Konrad Malawski

- youtu.be/x62K4ObBtw4
- youtu.be/bP0q0kbYYkA

Thank you!

softserve experience matters

fb: @vshatokhin
github: @vlasshatokhin