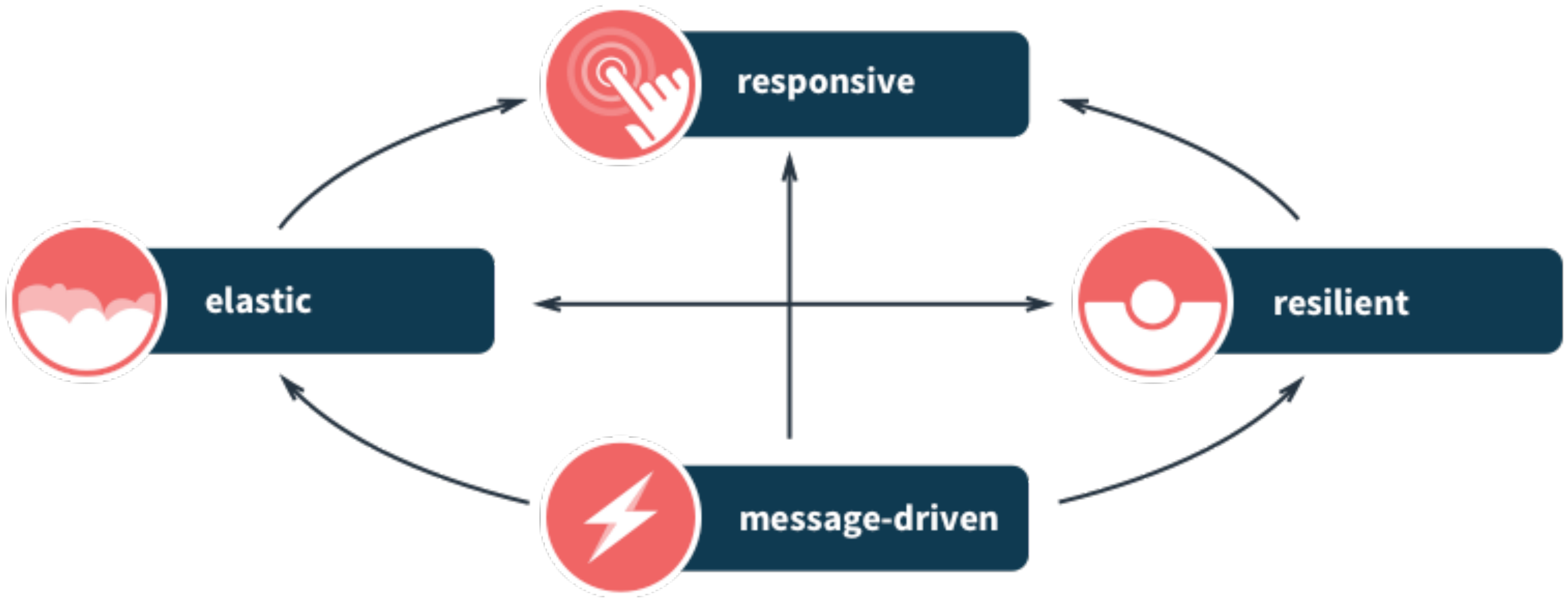
Akka Streams

Asynchronous non-blocking streaming made easy

Mirco Dotta
@mircodotta

Typesafe

The Four Reactive Traits



http://reactivemanifesto.org/

Why Reactive?

Why Reactive?

- Users expectations have changed
  - Services must be always up.
  - Must be fast.
- Billions of internet connected devices.
- Data is *transformed* and *pushed* continuously.

Reactive Streams

An initiative for providing

Standardised(!)

Back-pressured

Asynchronous

Stream processing

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

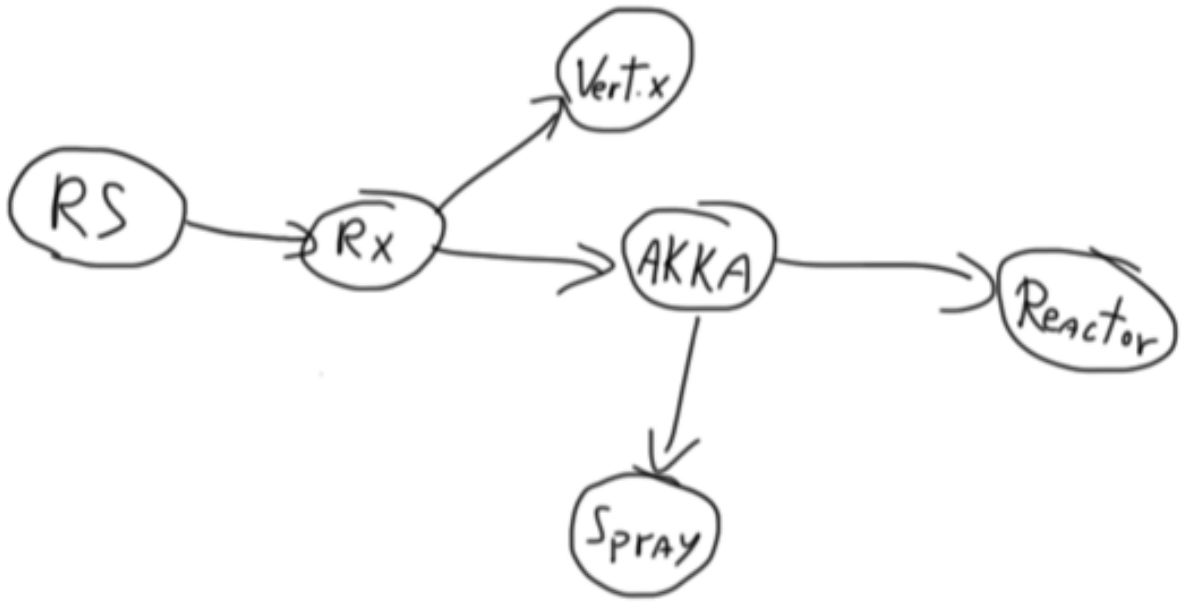
Reactive Streams: Who?

- Kaazing
- Netflix (rxJava)
- Pivotal (reactor)
- RedHat (vert.x)
- Twitter
- Typesafe (akka-streams & slick)
- Doug Lea proposed an implementation for JDK9!

Standardised!

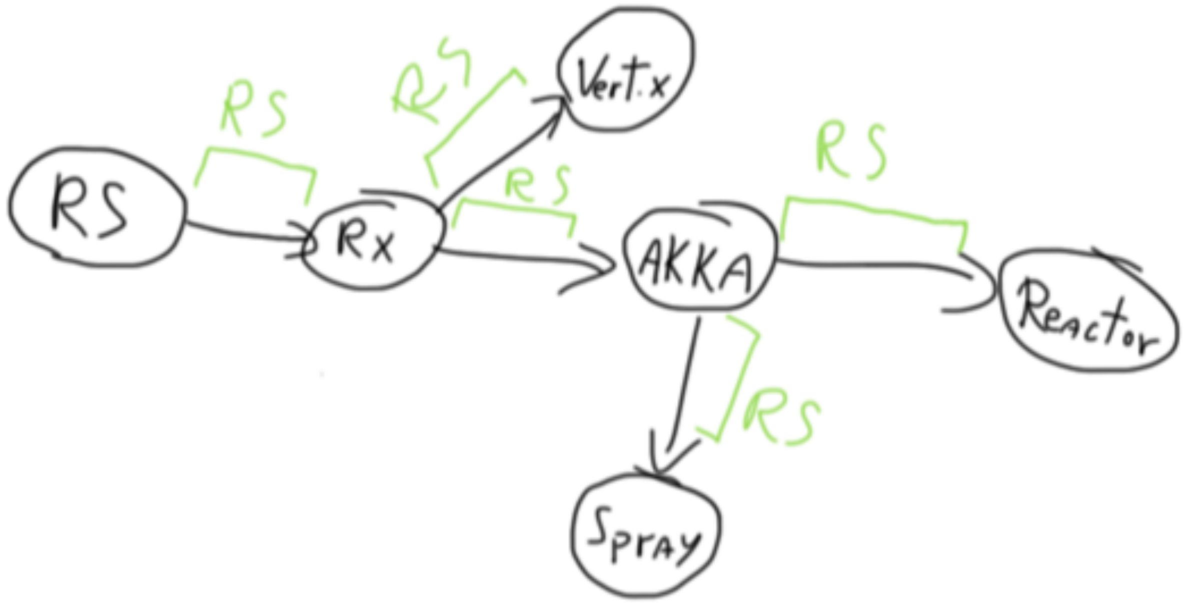
Reactive Streams: Inter-op

We want to make different implementations co-operate with each other.



Reactive Streams: Inter-op

The different implementations "talk to each other" using the Reactive Streams protocol.



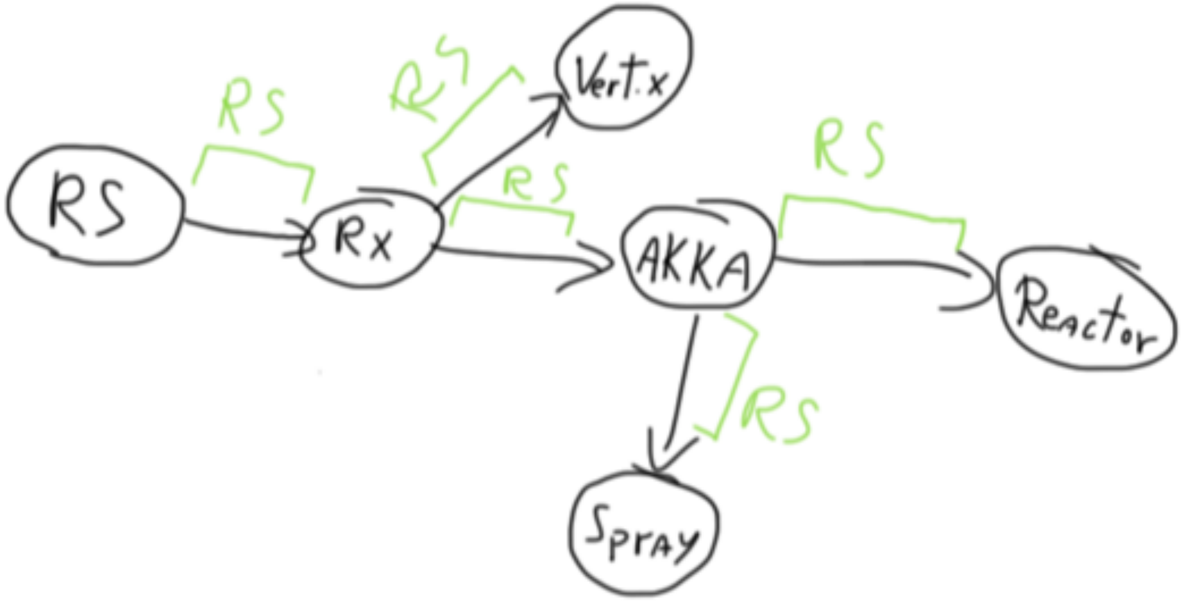
Reactive Streams: Inter-op

```scala
// here are a few imports that you are not seeing
object ScalaMain extends App {
  EmbeddedApp.fromHandler(new Handler {
    override def handle(ctx: Context): Unit = {
      // RxJava Observable
      val intObs = Observable.from((1 to 10).asJava)

      // Reactive Streams Publisher
      val intPub = RxReactiveStreams.toPublisher(intObs)

      // Akka Streams Source
      val stringSource = Source(intPub).map(_.toString)

      // Reactive Streams Publisher
      val stringPub = stringSource.runWith(Sink.fanoutPublisher(1, 1))

      // Reactor Stream
      val linesStream = Streams.create(stringPub).map[String](new reactor.function.Function[String, String] {
        override def apply(in: String) = in + "\n"
      })

      // and now render the HTTP response (RatPack)
      ctx.render(ResponseChunks.stringChunks(linesStream))
    }

  }).test(new Consumer[TestHttpClient] {
    override def accept(client: TestHttpClient): Unit = {
      val text = client.getText()
      println(text)
      system.shutdown()
    }
  })
}
```

https://github.com/rkuhn/ReactiveStreamsInterop

Reactive Streams: Inter-op

The Reactive Streams SPI is **NOT** meant to be user-api. You should use one of the implementing libraries.

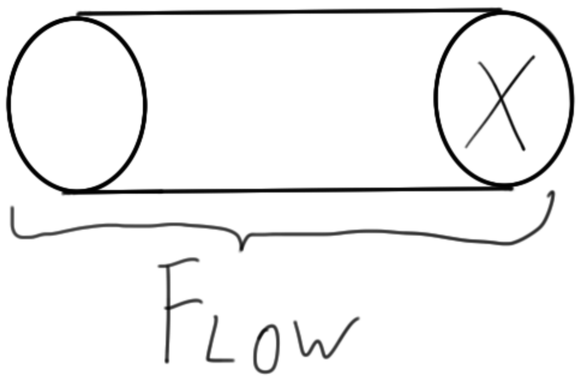


Akka Streams

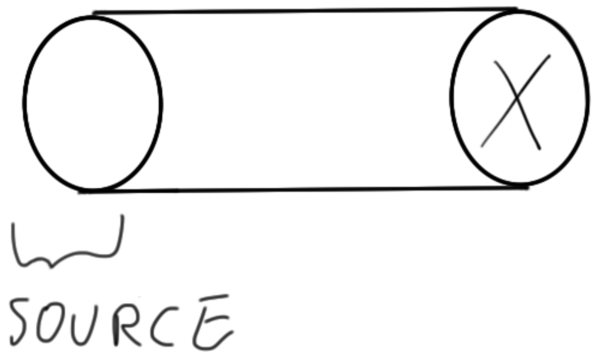
Akka Streams: Basics

- DSL for the formulation of *transformations* on *data streams*.
- Basic building blocks:
  - Source - something with exactly one output stream.
  - Flow - something with exactly one input and one output stream.
  - Sink - something with exactly one input stream.
  - RunnableFlow - A Flow that has both ends "attached" to a Source and Sink respectively, and is ready to be run() .

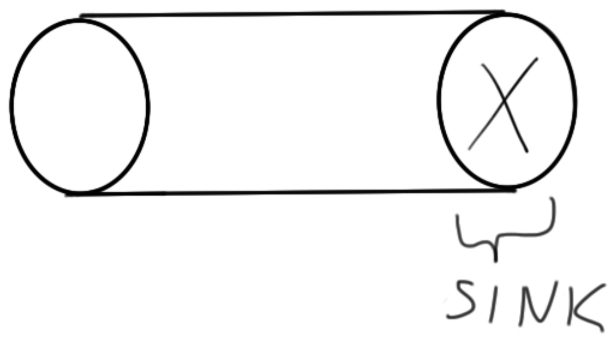
Akka Streams: Basics



Akka Streams: Basics



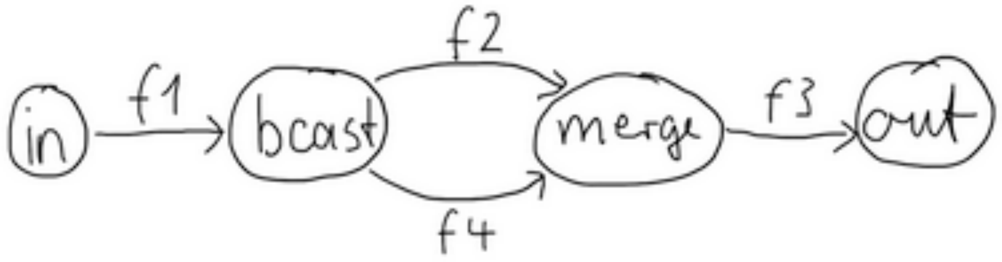
Akka Streams: Basics



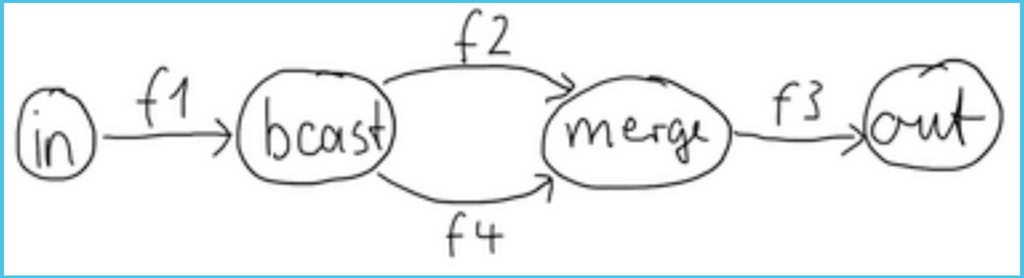
Demo 1

Akka Streams: Graph

- Source, Flow, and Sink are good for expressing linear computations.
- But how to express a computation *graph?*



Demo 2



Akka Streams: Fan-out

- `Broadcast` - given an input element emits to **each** output.
- `Balance` - given an input element emits to **one** of its output ports.
- `UnZip` - splits a stream of (A,B) tuples into two streams, one of type A and on of type B.
- `FlexiRoute` - enables writing custom fan out elements using a simple DSL.

Akka Streams: Fan-in

- `Merge` - picks **randomly** from inputs pushing them one by one to its output.
- `MergePreferred` - like Merge but if elements are available on **preferred** port, it picks from it, otherwise randomly from others.
- `ZipWith`(f_n) - takes a function of N inputs that given a value for each input emits 1 output element.

Akka Streams: Fan-in cont’d

- `Zip` - is a `ZipWith` specialised to zipping input streams of A and B into an (A,B) tuple stream.
- `Concat` - concatenates two streams (first consume one, then the second one).
- `FlexiMerge` - enables writing custom fan-in elements using a simple DSL.

Demo 3

What is back-pressure?

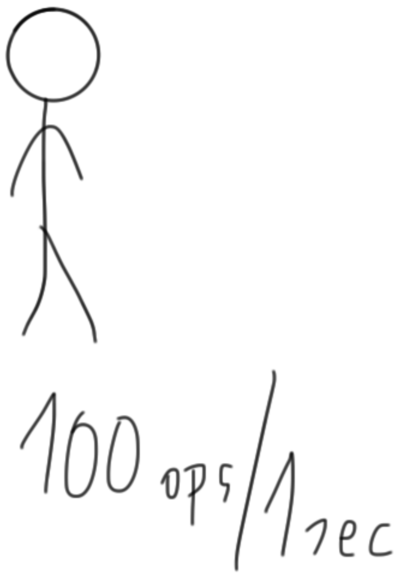
Back-pressure? Example Without

Publisher[T]

Subscriber[T]

Back-pressure? Example Without

Fast Publisher



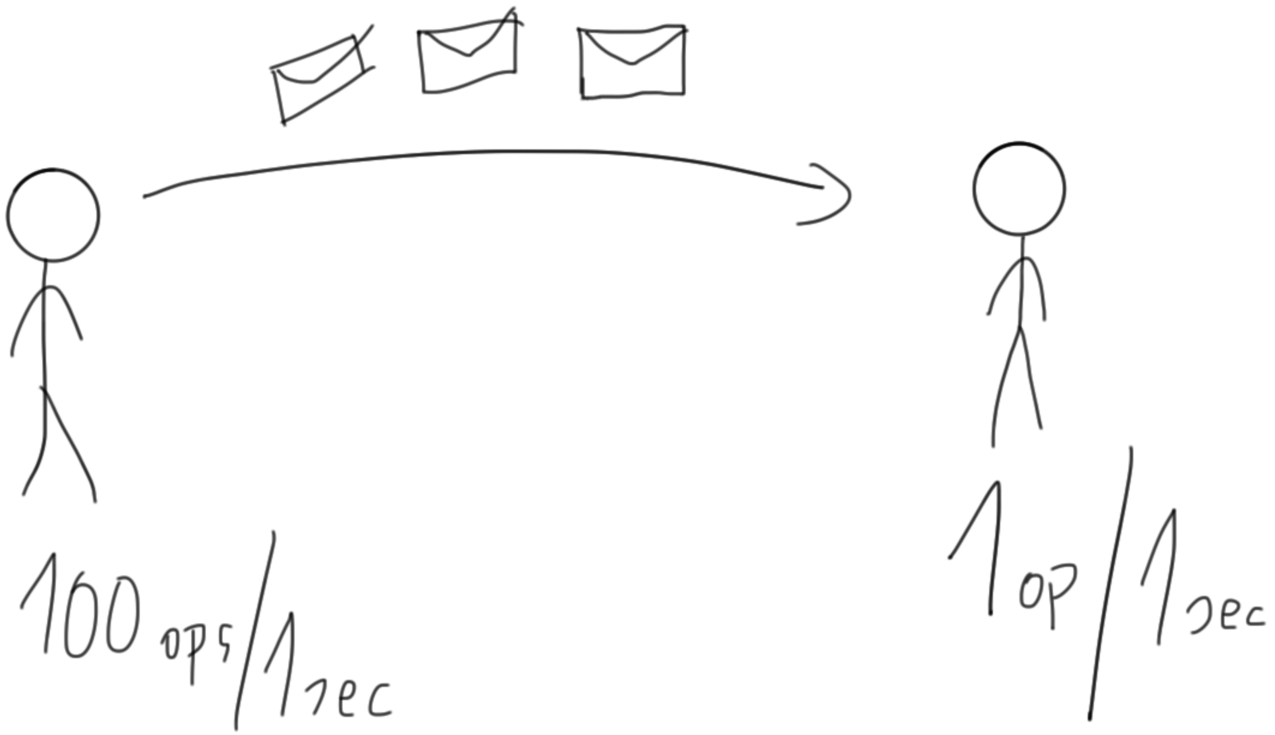
Slow Subscriber

Typesafe

Back-pressure?

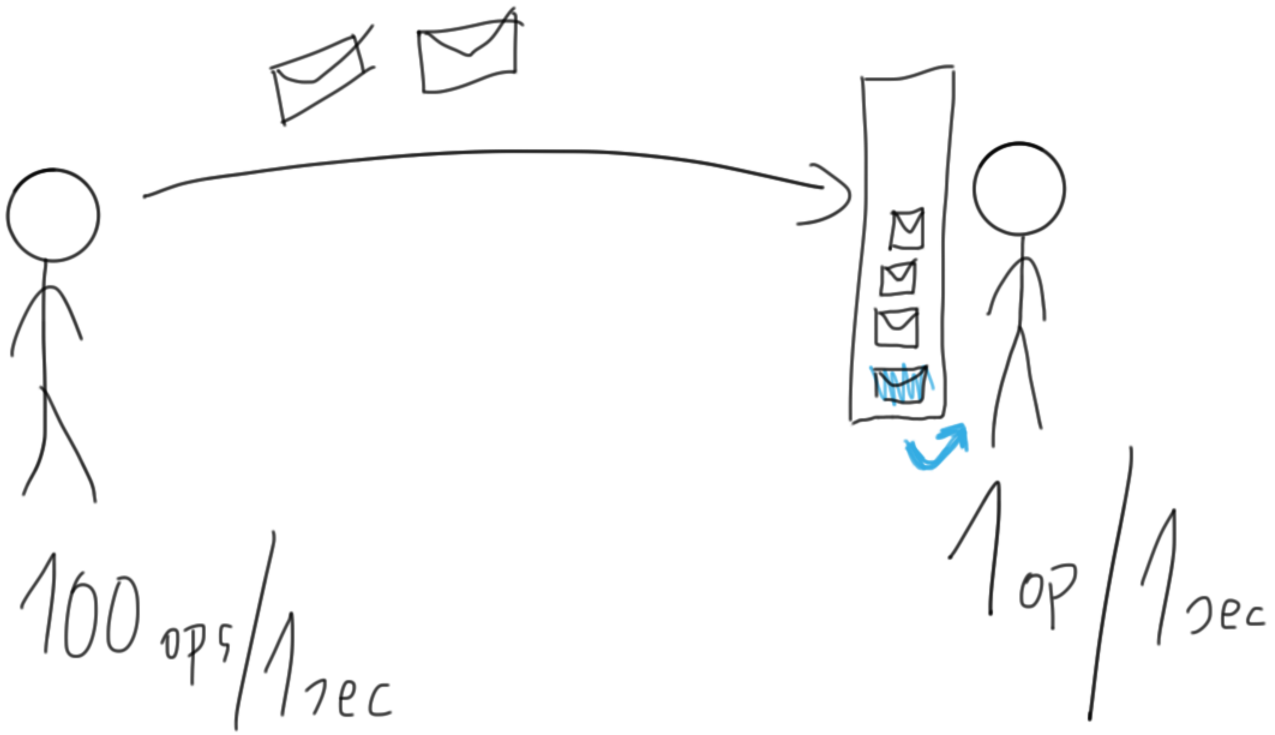
“Why would I need that!?”

Back-pressure? Push + NACK model

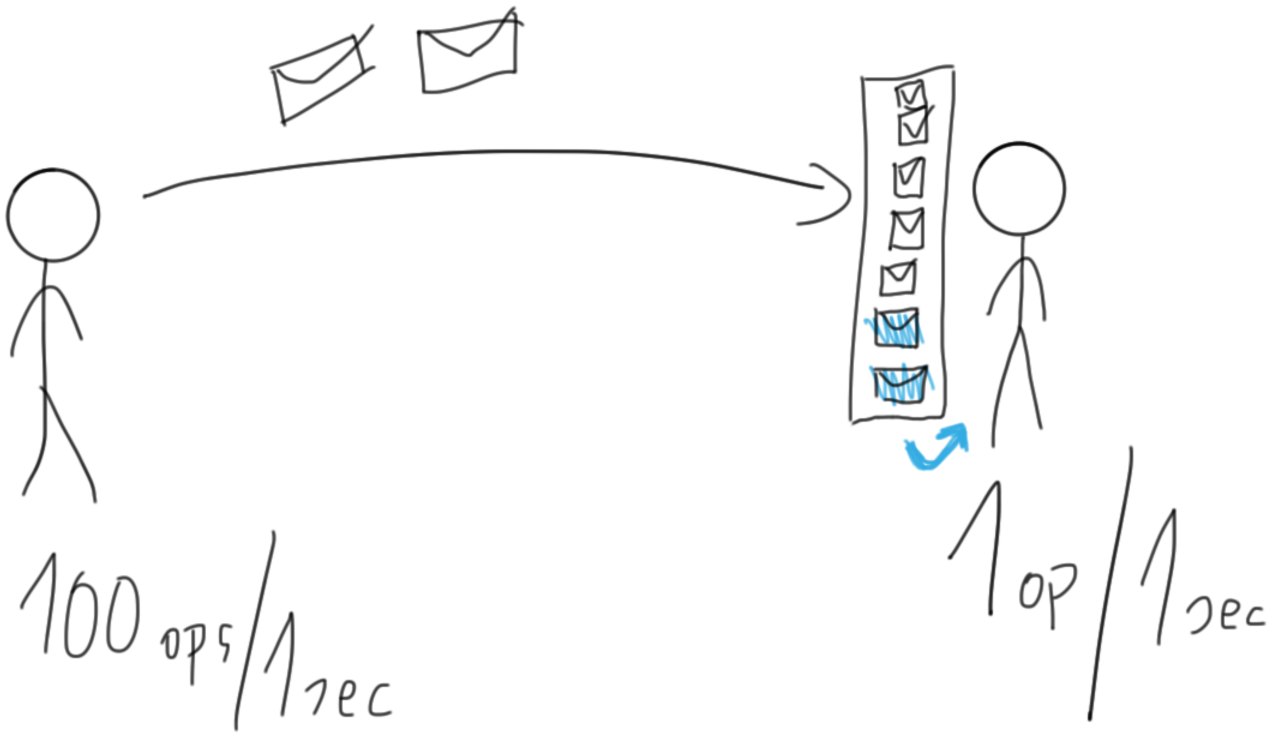


Back-pressure? Push + NACK model

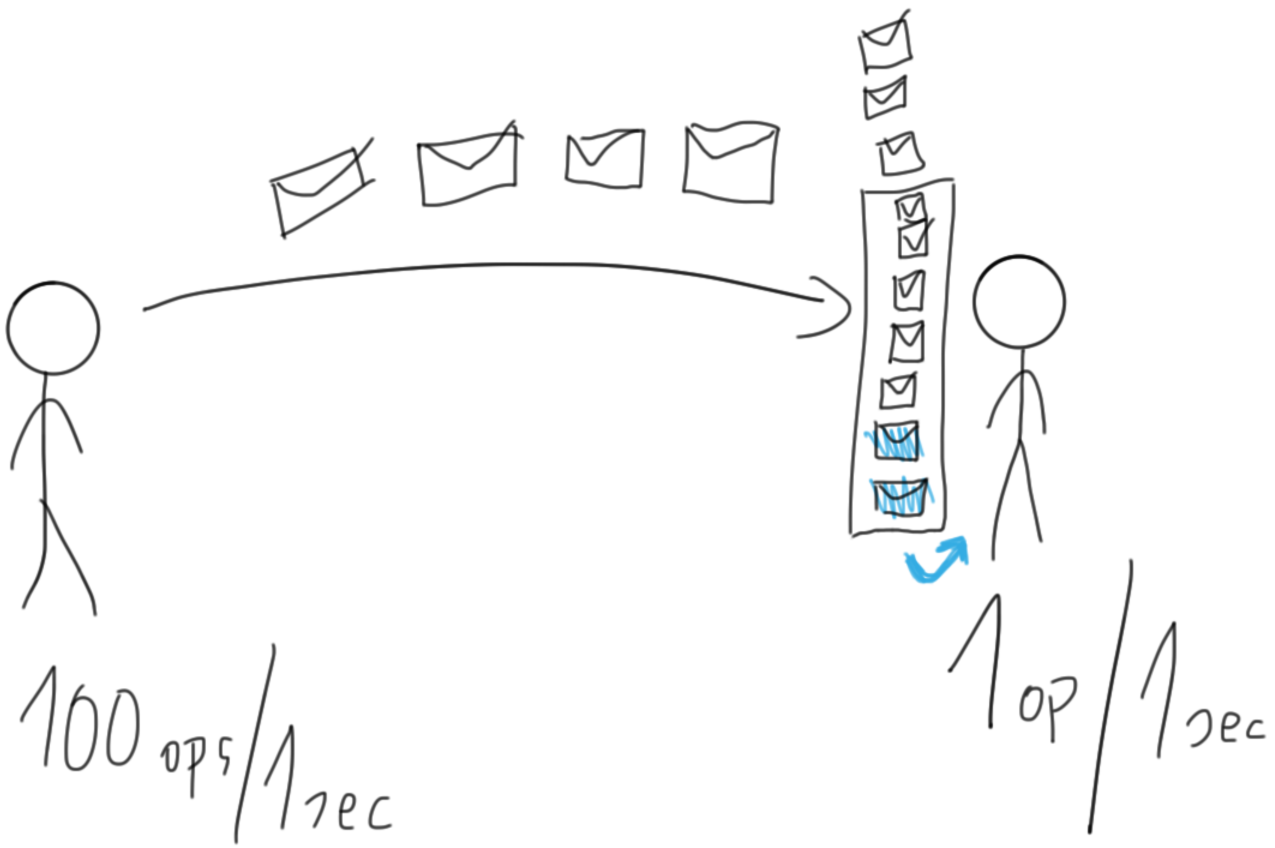
Subscriber usually has some kind of buffer.



Back-pressure? Push + NACK model

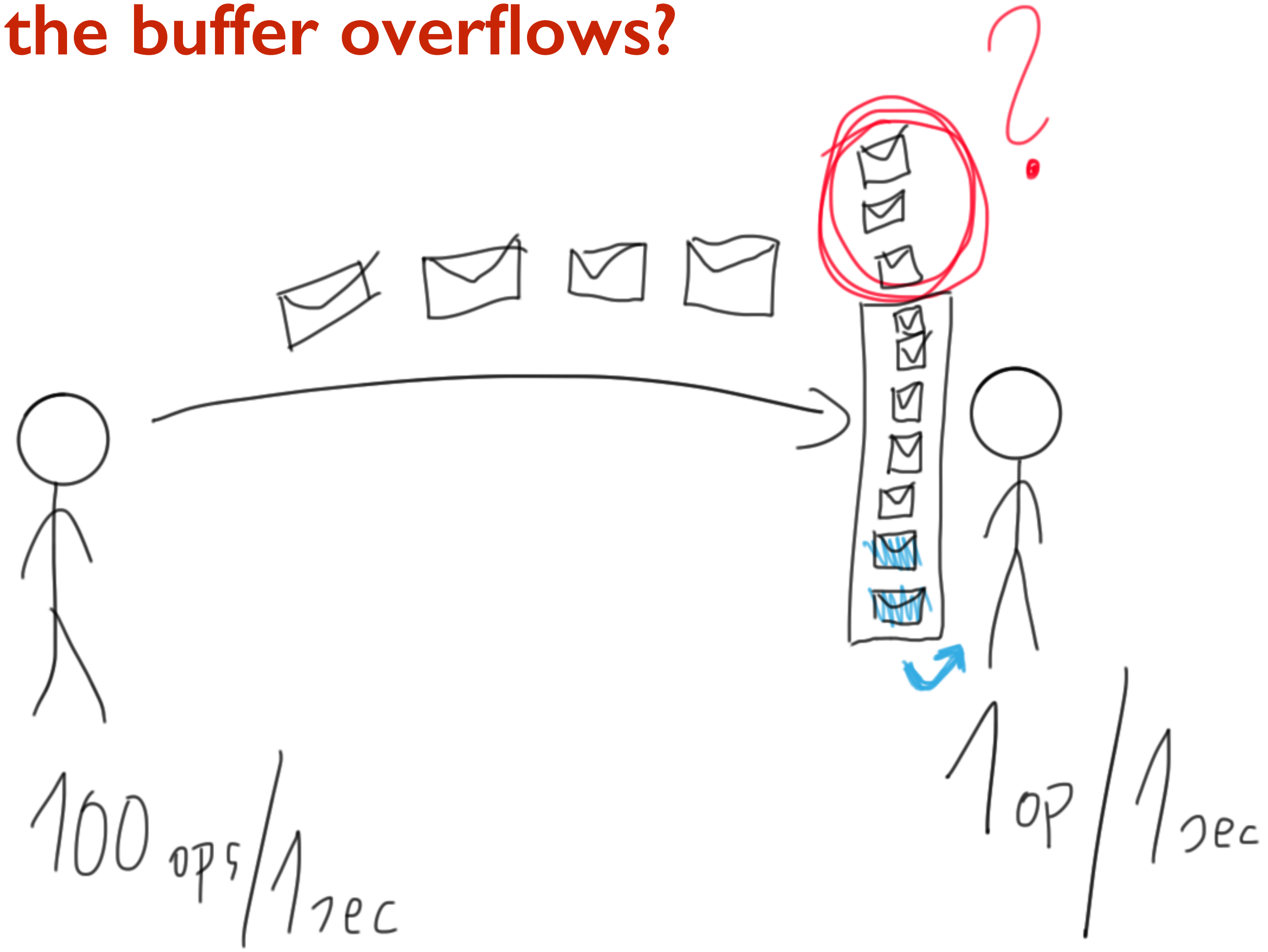


Back-pressure? Push + NACK model



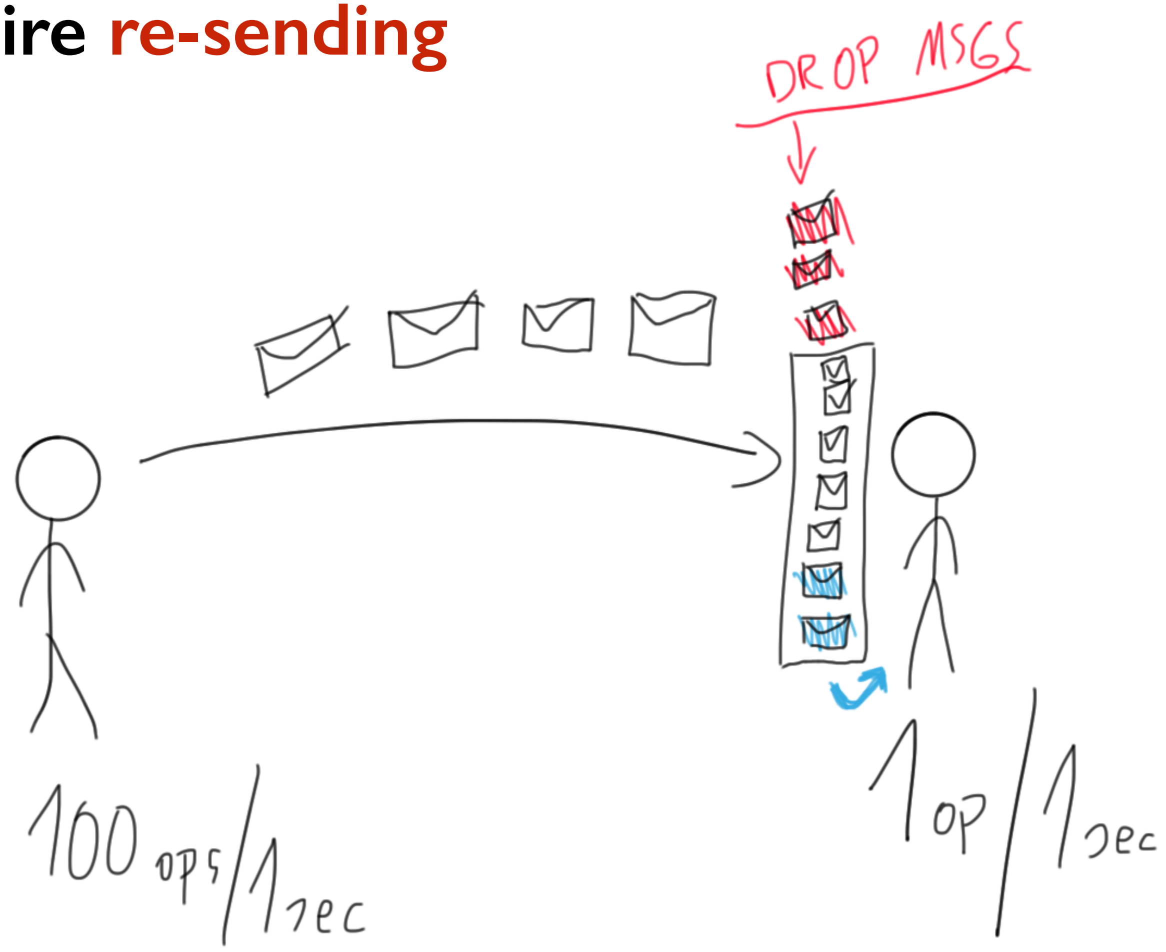
Back-pressure? Push + NACK model

What if the buffer overflows?



Back-pressure? Push + NACK model (a)

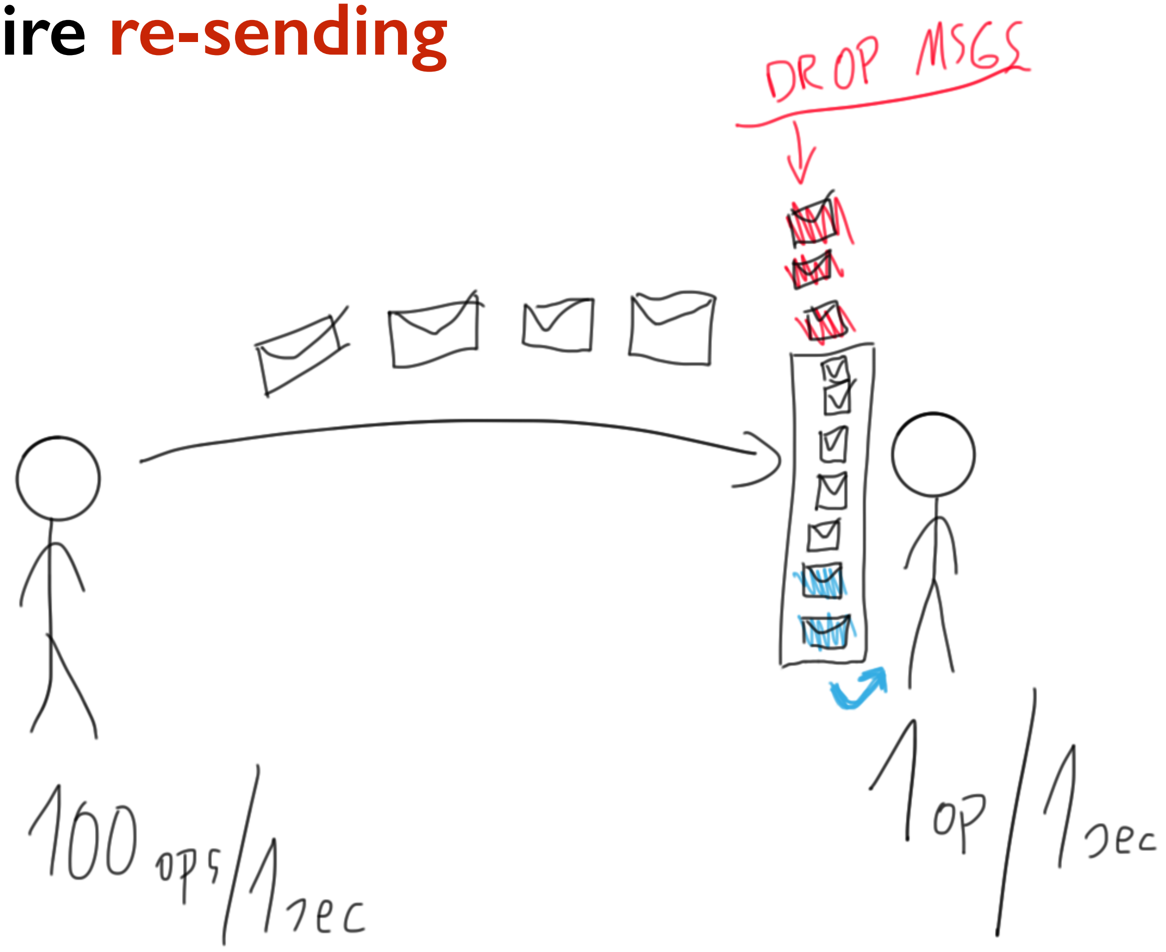
Use bounded buffer,
drop messages + require re-sending



Back-pressure? Push + NACK model (a)

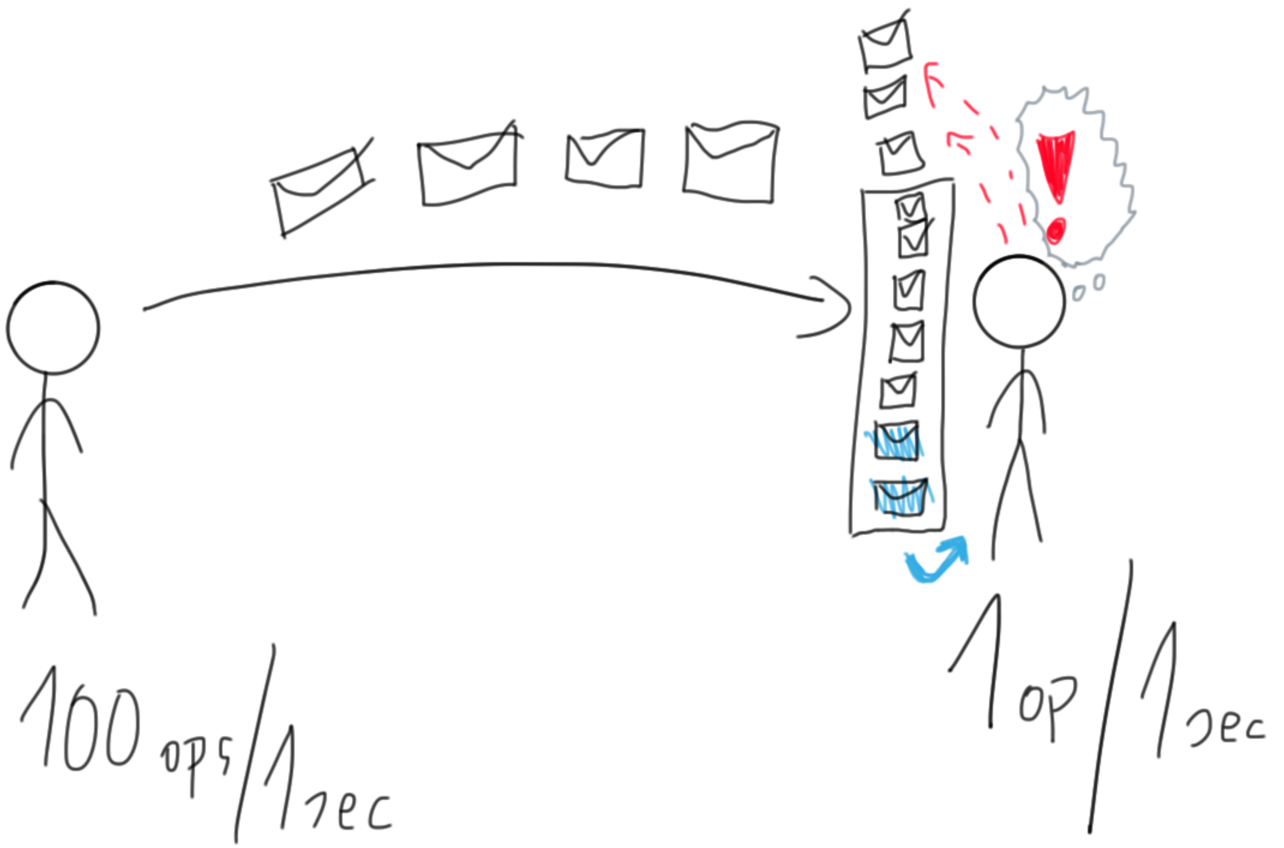
Use bounded buffer,
drop messages + require re-sending

Kernel does this!
Routers do this!
(TCP)

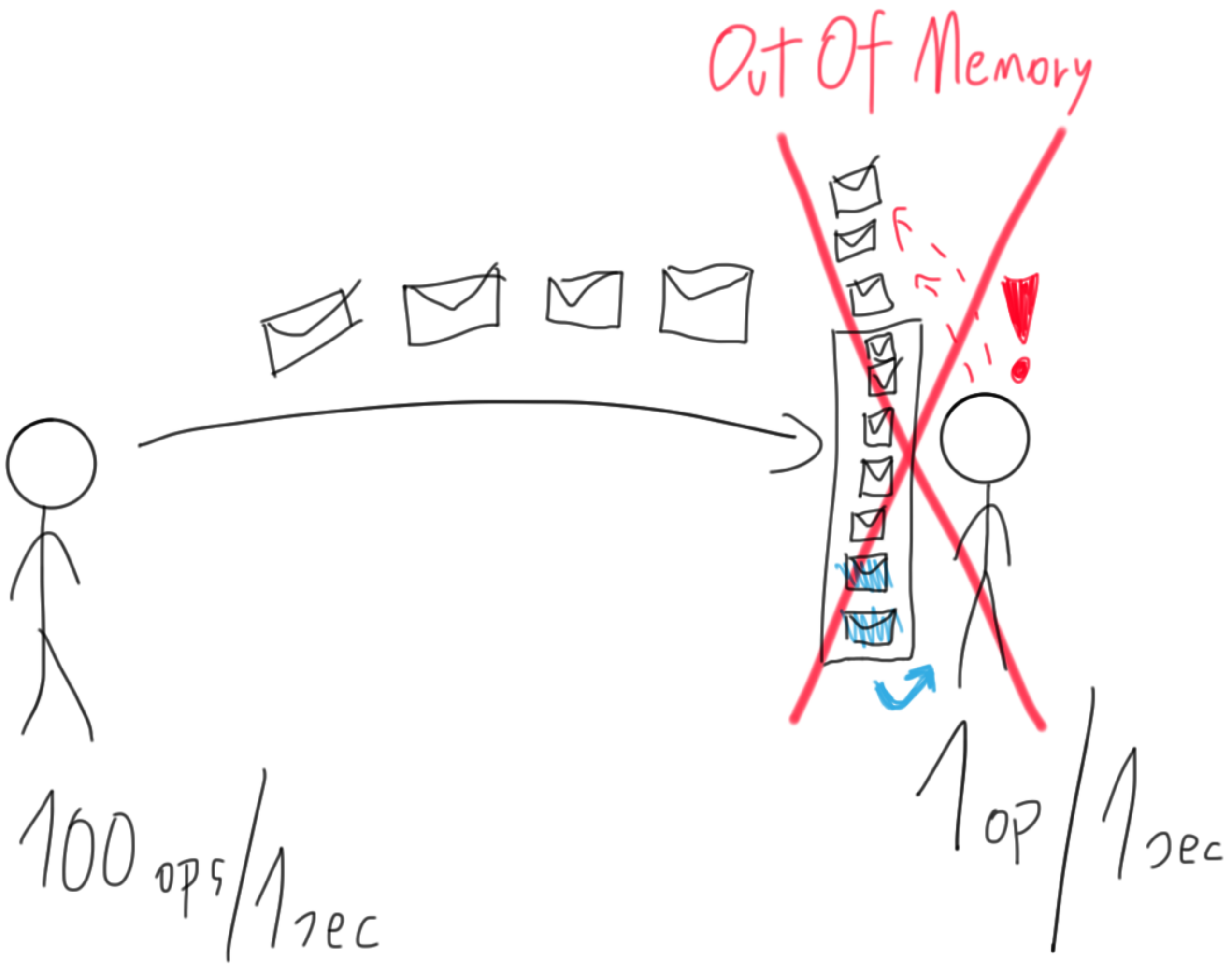


Back-pressure? Push + NACK model (b)

Increase buffer size…
Well, while you have memory available!



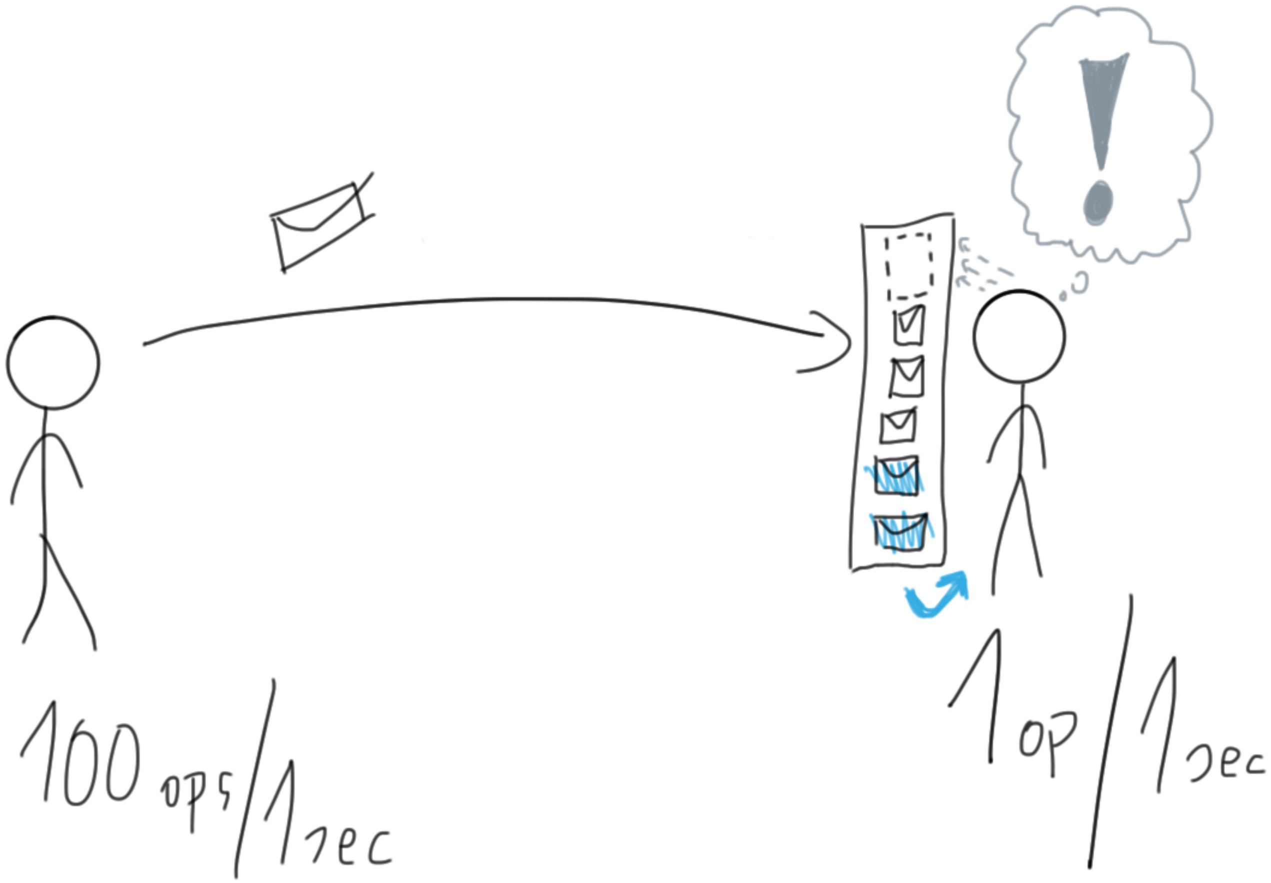
Back-pressure? Push + NACK model (b)



Negative **ACK**nowledgement

Back-pressure? Example NACKing

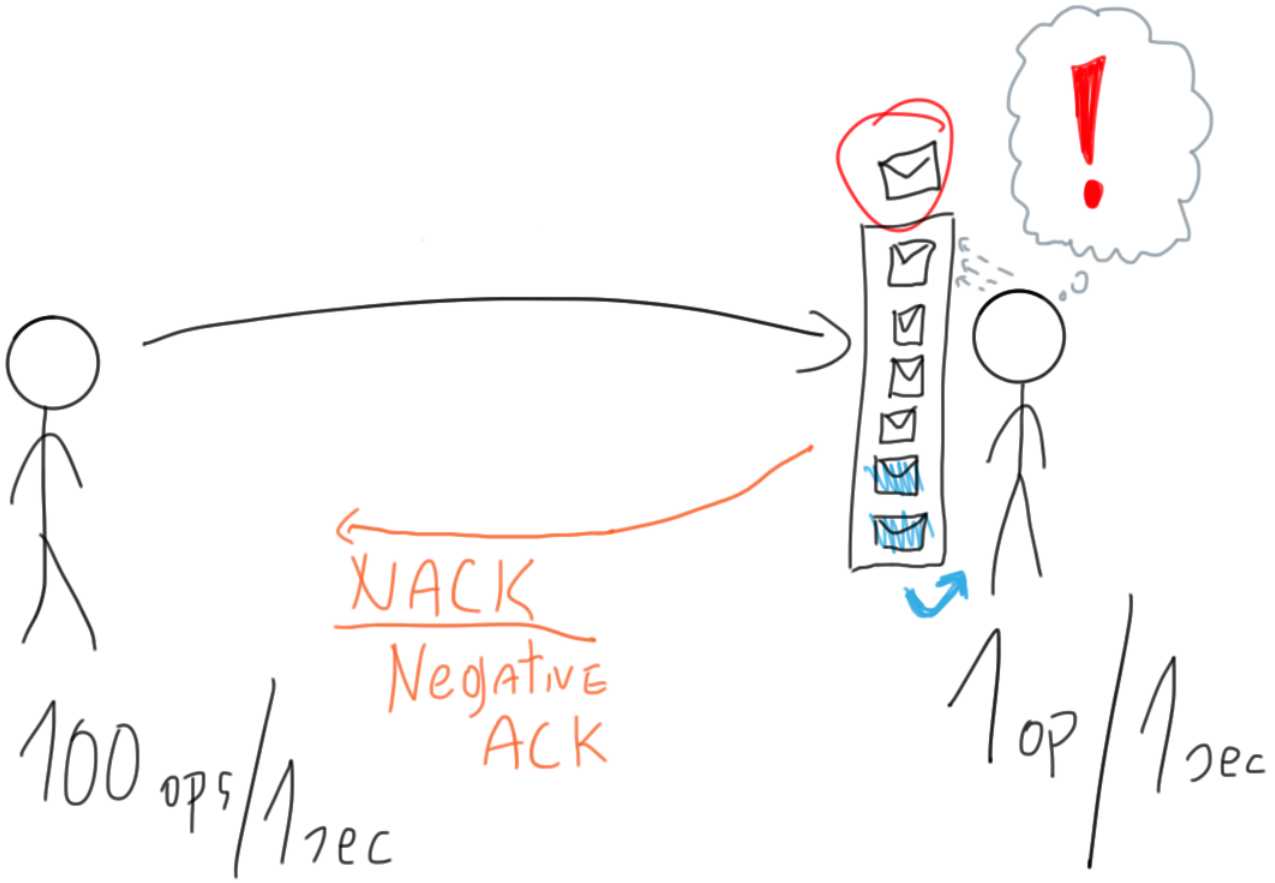
Buffer overflow is imminent!



Typesafe

Back-pressure? Example NACKing

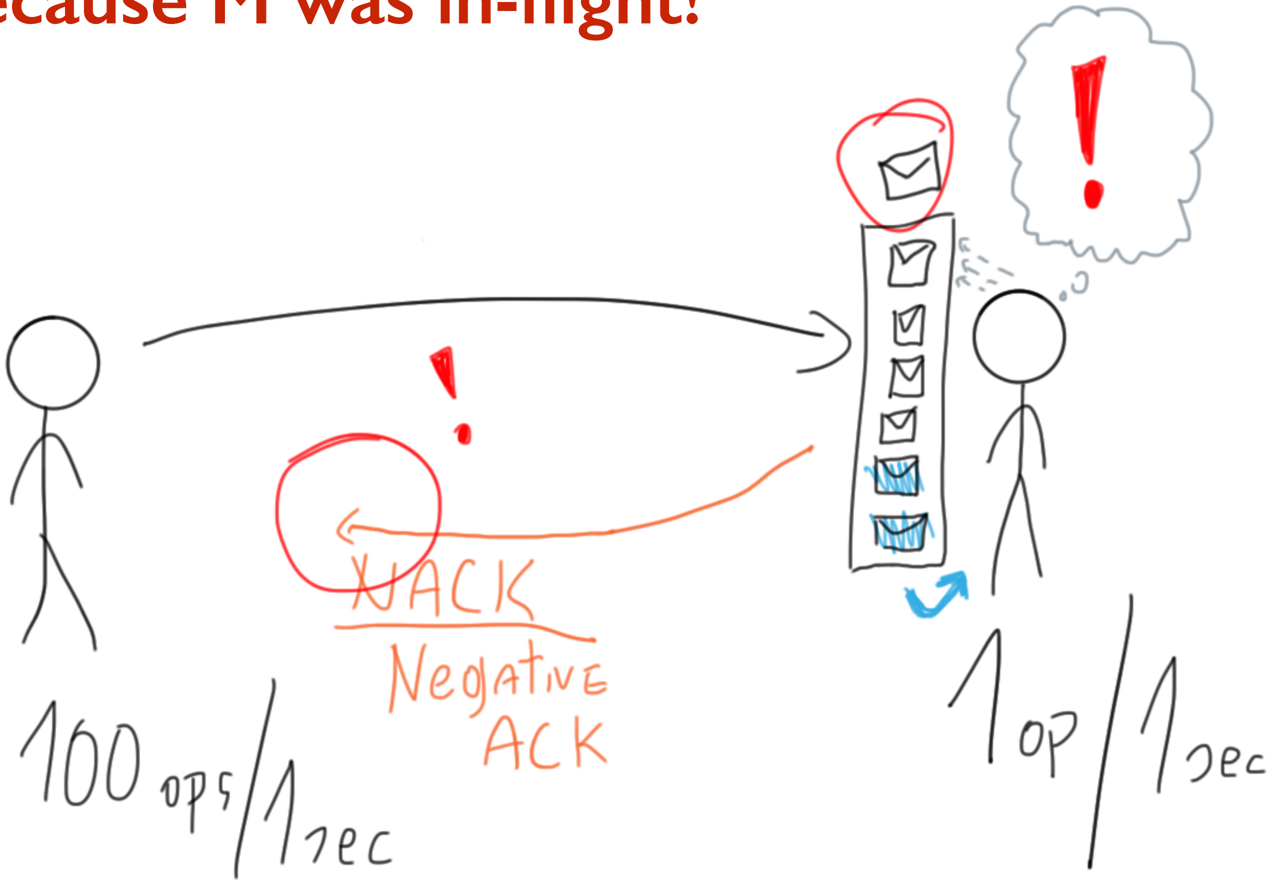
Telling the Publisher to slow down / stop sending…



Back-pressure? Example NACKing

NACK did not make it in time,

because M was in-flight!



Back-pressure?

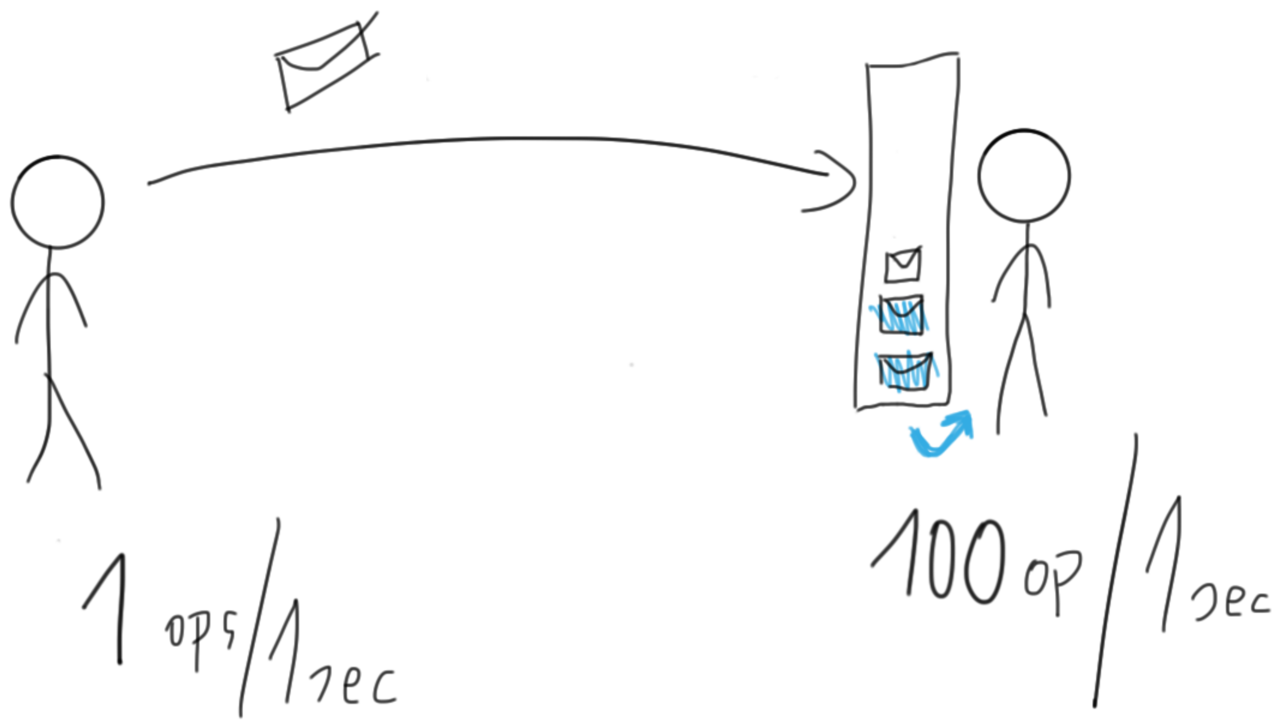
NACKing is NOT enough.

Back-pressure?

speed(publisher) < speed(subscriber)

Back-pressure? Fast Subscriber, No Problem

No problem!



Typesafe

Back-pressure?
Reactive-Streams
=

Back-pressure? RS: Dynamic Push/Pull

Just push – not safe when Slow Subscriber

Just pull – too slow when Fast Subscriber

Back-pressure? RS: Dynamic Push/Pull

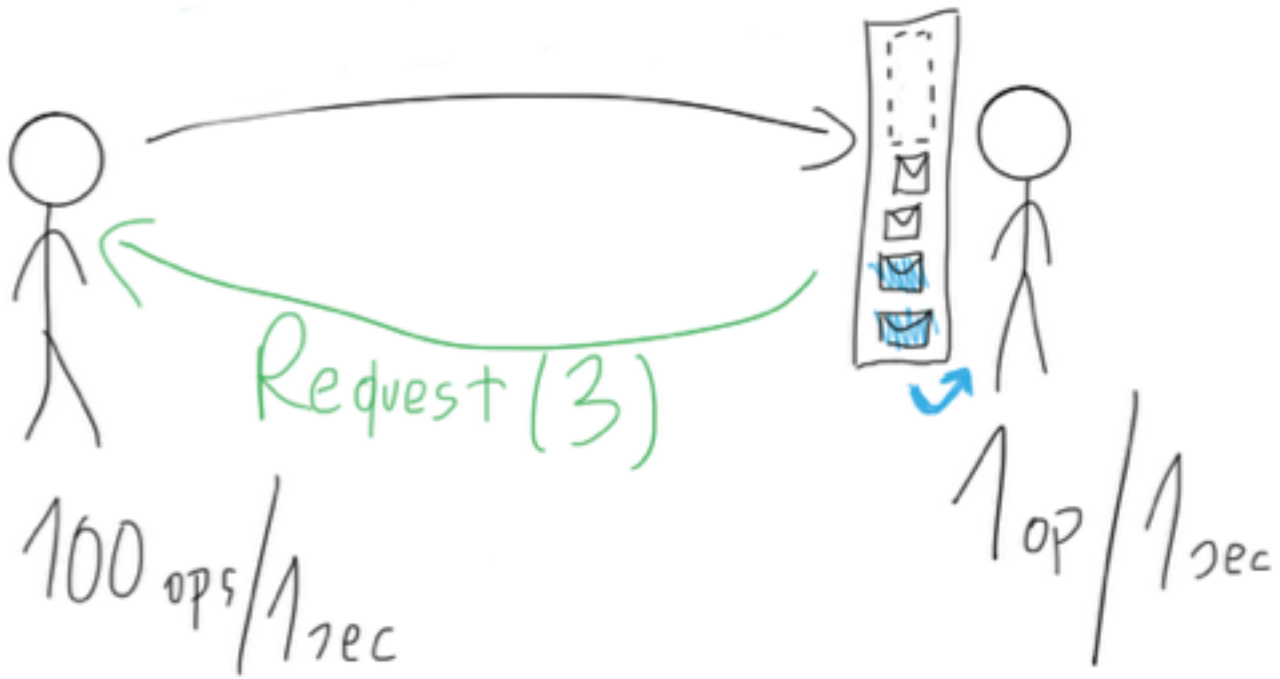
Just push – not safe when Slow Subscriber

Just pull – too slow when Fast Subscriber

Solution:
Dynamic adjustment

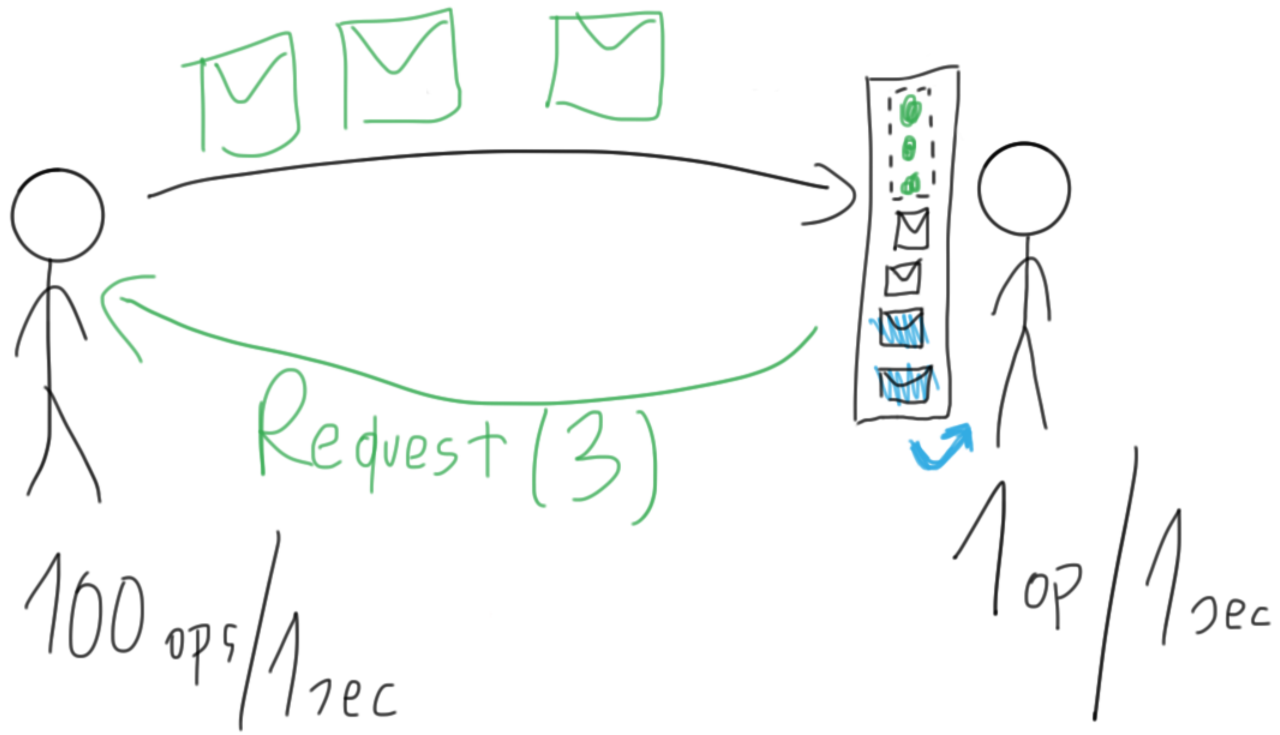
Back-pressure? RS: Dynamic Push/Pull

Slow Subscriber sees it's buffer can take 3 elements. Publisher will never blow up it's buffer.



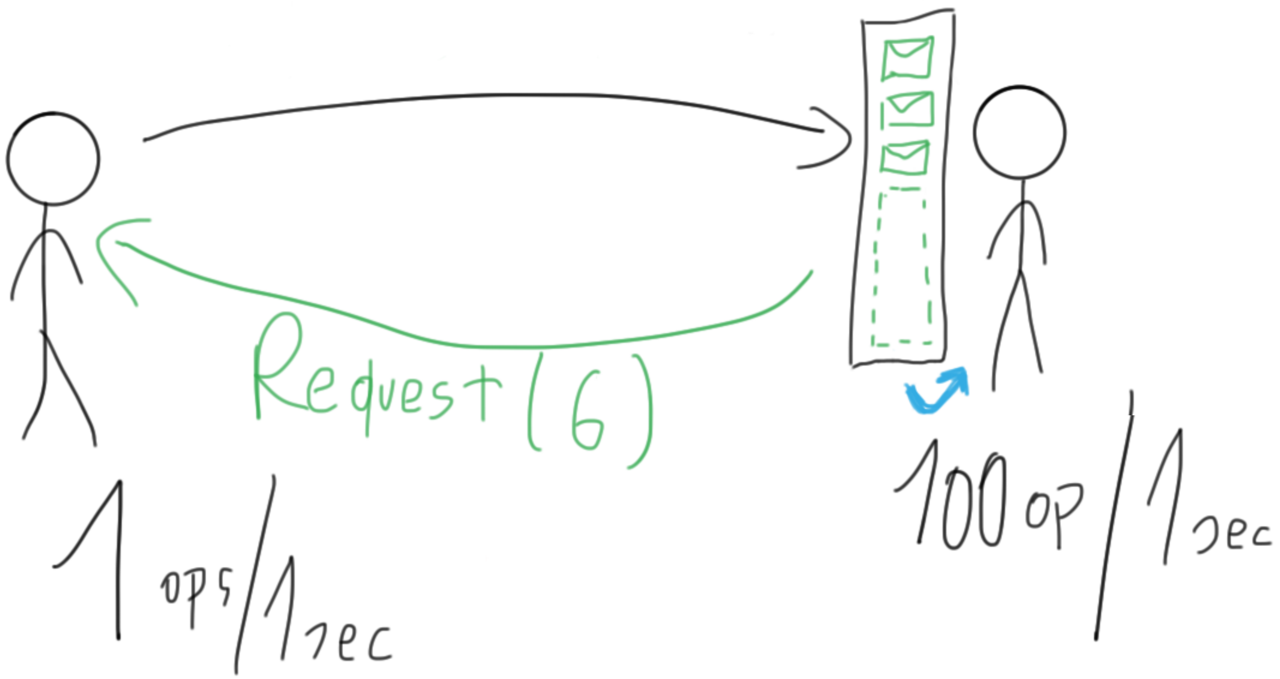
Back-pressure? RS: Dynamic Push/Pull

Fast Publisher will send at-most 3 elements. This is pull-based-backpressure.



Back-pressure? RS: Dynamic Push/Pull

Fast Subscriber can issue more Request(n), before more data arrives!

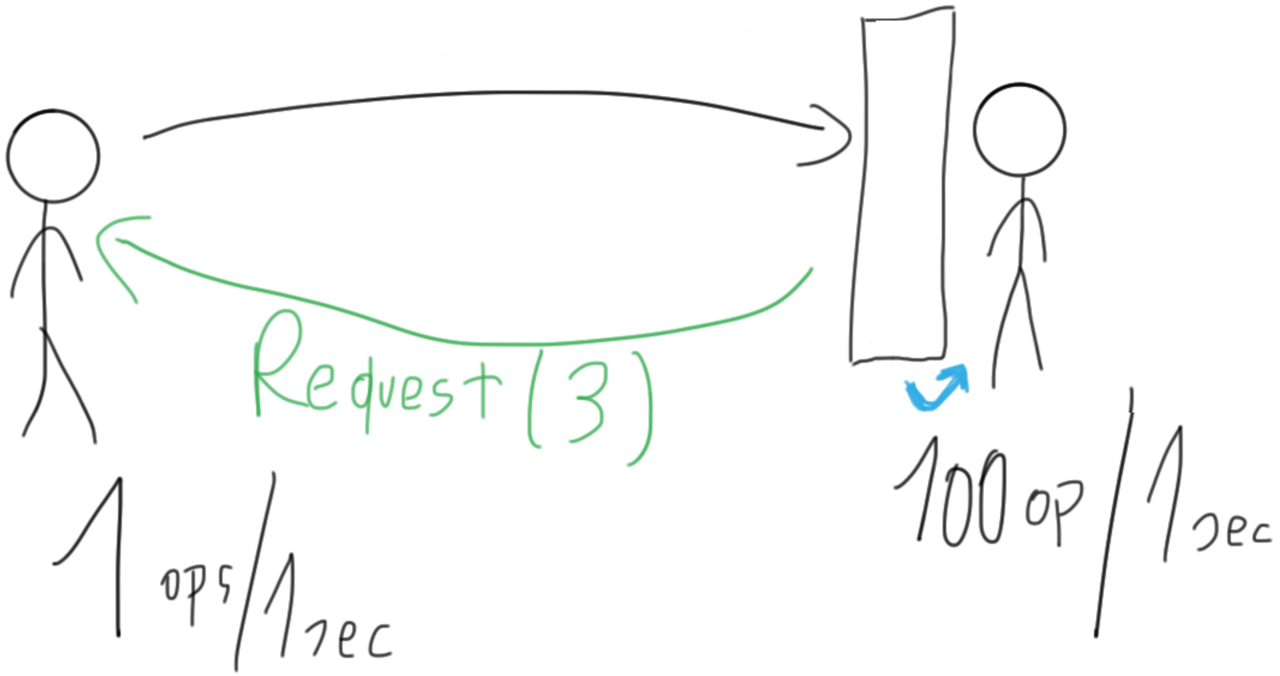


Typesafe

Back-pressure? RS: Dynamic Push/Pull

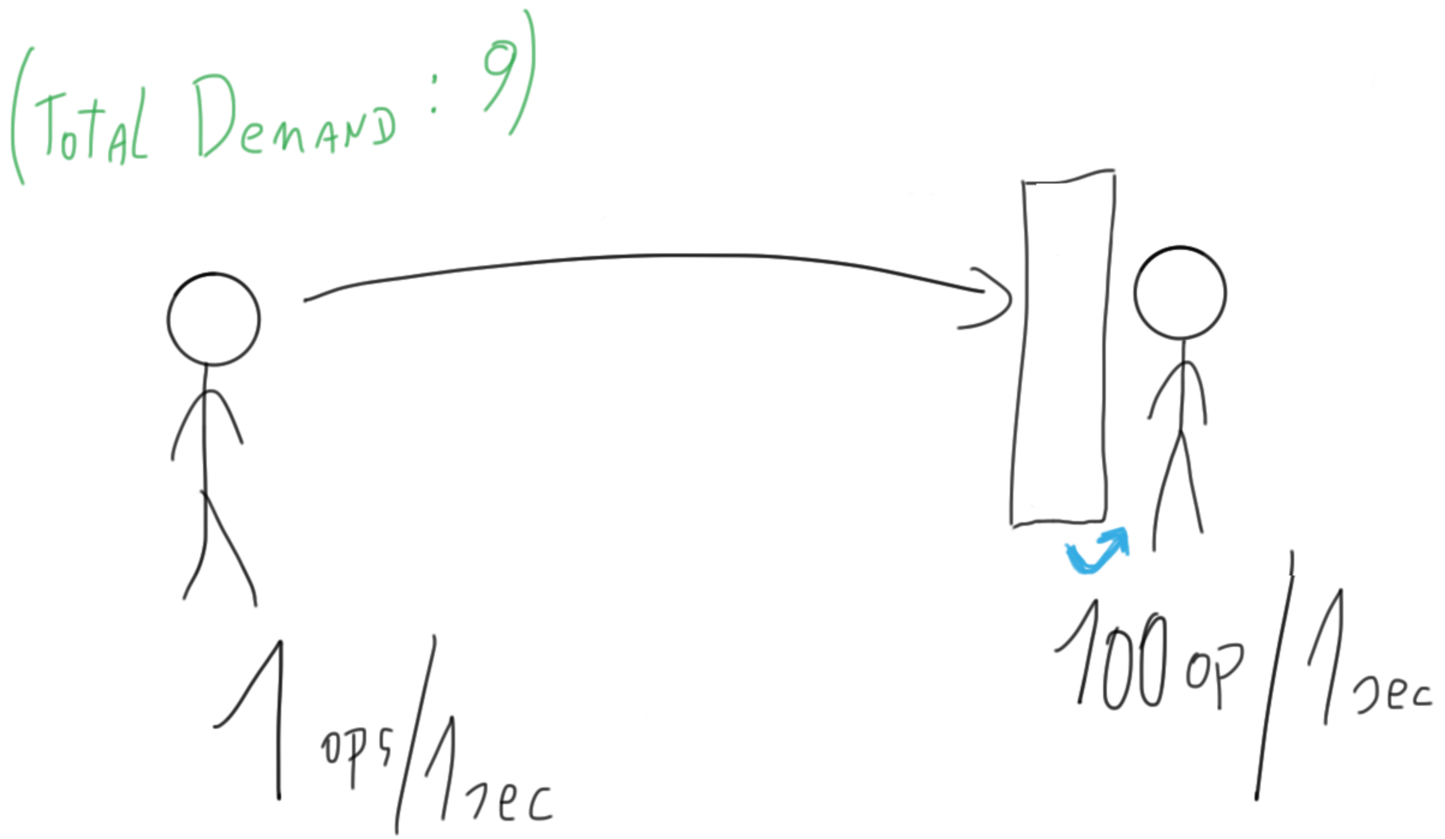
Fast Subscriber can issue more Request(n), before more data arrives.

Publisher can accumulate demand.



Back-pressure? RS: Accumulate demand

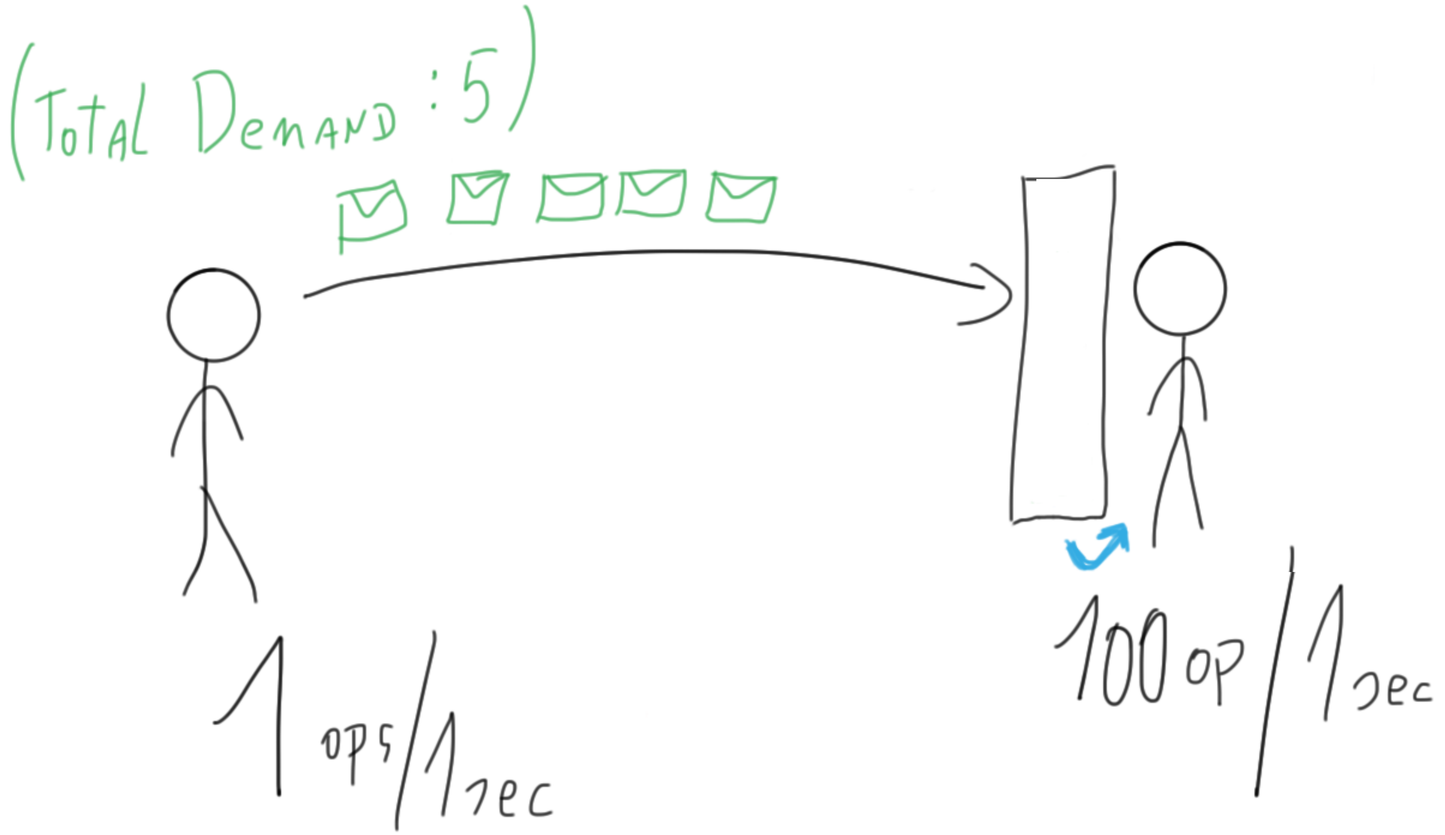
Publisher accumulates total demand per subscriber.



Back-pressure? RS: Accumulate demand

Total demand of elements is **safe to publish.**

Subscriber's buffer will not overflow.



Demo 4

Is that really all there is to know?

- Naaaa, there is a lot more for you to explore!
  - If the existing building blocks are not enough, define your owns.
  - Use `mapAsync/mapAsyncUnordered` for integrating with external services.
  - Streams Error Handling.
  - Handling TCP connections with Streams.
  - Integration with Actors.

What now?

- Use it:
  `"com.typesafe.akka" %% "akka-stream-experimental" % "1.0-RC2"`
- Check out the Activator template
  Akka Streams with Java8 or Scala.
- Akka Streams API doc and user guide for both Java8 and Scala.
- Code used for the demos https://github.com/dotta/akka-streams-demo/releases/tag/v02

Next Steps

- Akka Streams 1.0 final soon.
- Inclusion in future JDK (shooting for JDK9)
- We aim at polyglot standard (JS, wire proto)
- Try it out and give feedback!
- http://reactive-streams.org/
- https://github.com/reactive-streams

Typesafe Reactive Platform

A Unified Platform for Building Modern Apps

play akka Scala Java™

Typesafe