How reactive streams & akka streams change the JVM Ecosystem

Typesafe

Konrad `@ktosopl` Malawski @ LinkedIn 2015

Typesafe *(we're renaming soon!)*

Konrad `ktoso` Malawski

Akka Team,
Reactive Streams TCK,
Maintaining Akka Http

Typesafe *(we're renaming soon!)*

Konrad `@ktosopl` Malawski

akka.io
typesafe.com
geecon.org
Java.pl / KrakowScala.pl
sckrk.com / meetup.com/Paper-Cup @ London
GDGKrakow.pl
lambdakrk.pl

Nice to meet you!
Who are you guys?

Agenda for today:

- Story & landscape
- The Reactive Streams Protocol
- Akka Streams / Demo
- Akka Http / Demo
- Q/A?

Reactive Streams - story: early FRP

- .NETs’ Reactive Extensions

http://blogs.msdn.com/b/rxteam/archive/2009/11/17/announcing-reactive-extensions-rx-for-net-silverlight.aspx
http://infoscience.epfl.ch/record/176887/files/DeprecatingObservers2012.pdf - Ingo Maier, Martin Odersky
https://github.com/ReactiveX/RxJava/graphs/contributors
https://github.com/reactor/reactor/graphs/contributors
https://medium.com/@viktorklang/reactive-streams-1-0-0-interview-faaca2c00bec#.69st3rndy

Reactive Streams - story: 2013's impls

PLAY
AKKA
RX

~2013:

Reactive Programming
becoming widely adopted on JVM.

- Play introduced "Iteratees"
- Akka (2009) had Akka-IO (TCP etc.)
- Ben starts work on RxJava

} Teams discuss need for back-pressure in simple user API.
Play's Iteratee / Akka's NACK in IO.

http://blogs.msdn.com/b/rxteam/archive/2009/11/17/announcing-reactive-extensions-rx-for-net-silverlight.aspx
http://infoscience.epfl.ch/record/176887/files/DeprecatingObservers2012.pdf - Ingo Maier, Martin Odersky
https://github.com/ReactiveX/RxJava/graphs/contributors
https://github.com/reactor/reactor/graphs/contributors
https://medium.com/@viktorklang/reactive-streams-1-0-0-interview-faaca2c00bec#.69st3rndy

Reactive Streams - story: 2013’s impls

Play Iteratees – pull back-pressure, difficult API

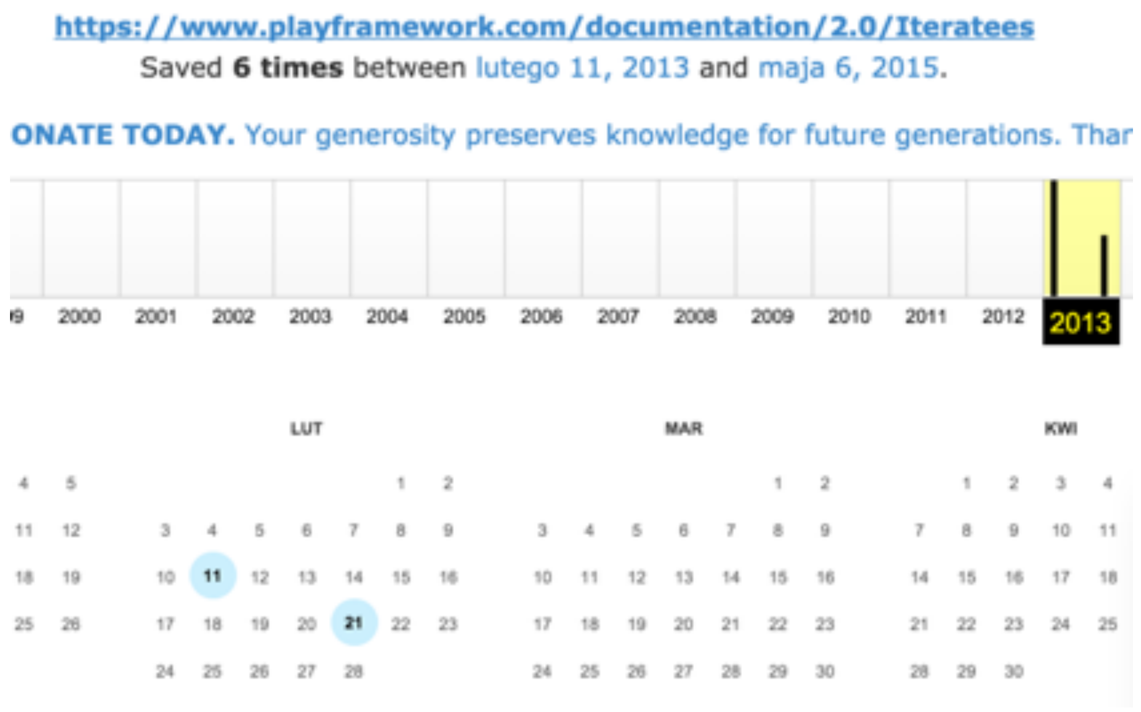
Akka-IO – NACK back-pressure; low-level IO (Bytes); messaging API

RxJava – no back-pressure, nice API

http://blogs.msdn.com/b/rxteam/archive/2009/11/17/announcing-reactive-extensions-rx-for-net-silverlight.aspx
http://infoscience.epfl.ch/record/176887/files/DeprecatingObservers2012.pdf - Ingo Maier, Martin Odersky
https://github.com/ReactiveX/RxJava/graphs/contributors
https://github.com/reactor/reactor/graphs/contributors
https://medium.com/@viktorklang/reactive-streams-1-0-0-interview-faaca2c00bec#.69st3rndy

Reactive Streams - Play’s Iteratees

```scala
// an iteratee that consumes chunkes of String and produces an Int
Iteratee[String,Int]

def fold[B](
  done: (A, Input[E]) => Promise[B],
  cont: (Input[E] => Iteratee[E, A]) => Promise[B],
  error: (String, Input[E]) => Promise[B]
): Promise[B]
```


https://www.playframework.com/documentation/2.0/Iteratees

Saved **6 times** between lutego 11, 2013 and maja 6, 2015.

ONATE TODAY. Your generosity preserves knowledge for future generations. Thar

Feb 2013

Iteratees solved the back-pressure problem, but were hard to use.

Iteratee & Enumeratee – Haskell inspired.

Play / Akka teams looking for common concept.

https://www.playframework.com/documentation/2.0/Iteratees

Reactive Streams - expert group founded

October 2013

Roland Kuhn (Akka) and **Erik Meijer (Rx .NET)** meet in Lausanne,
while recording **“Principles of Reactive Programming” Coursera Course**.

Viktor Klang (Akka), Erik Meijer, Ben Christensen (RxJava)
and Marius Eriksen (Twitter) meet at Twitter HQ.

The term “**reactive non-blocking asynchronous back-pressure**” gets coined.

Reactive Streams - expert group founded

October 2013

Roland Kuhn (A ,
while recording " **Course**.

Viktor Klang (A
and Marius Erik

The term "**reacti** ets coined.

Goals:

- asynchronous
- never block (waste)
- safe (back-threads pressured)
- purely local abstraction
- allow synchronous impls.

Also, for our examples today:

- compatible with TCP

Reactive Streams - expert group founded

October 2013

Roland Kuhn (Akka) and Erik Meijer (Rx .NET) meet in Lausanne,
while recording "Principles of Reactive Programming" Coursera Course.

Viktor Klang (Akka), Erik Meijer, Ben Christensen (RxJava)
and Marius Eriksen (Twitter) meet at Twitter HQ.

The term "reactive non-blocking asynchronous back-pressure" gets coined.

December 2013
Stephane Maldini & Jon Brisbin (Pivotal Reactor) contacted by Viktor.

Reactive Streams - expert group founded

October 2013

Roland Kuhn (Akka) and **Erik Meijer (Rx .NET)** meet in Lausanne,
while recording **"Principles of Reactive Programming" Coursera Course**.

Viktor Klang (Akka), Erik Meijer, Ben Christensen (RxJava)
and Marius Eriksen (Twitter) meet at Twitter HQ.

The term "**reactive non-blocking asynchronous back-pressure**" gets coined.

December 2013
Stephane Maldini & Jon Brisbin (Pivotal Reactor) contacted by **Viktor.**

Soon after, the "Reactive Streams" expert group is formed.

Also joining the efforts: Doug Lea (Oracle), Endre Varga (Akka), Johannes Rudolph & Mathias Doenitz (Spray), and many others, including myself join the effort soon after.

Reactive Streams - expert group founded

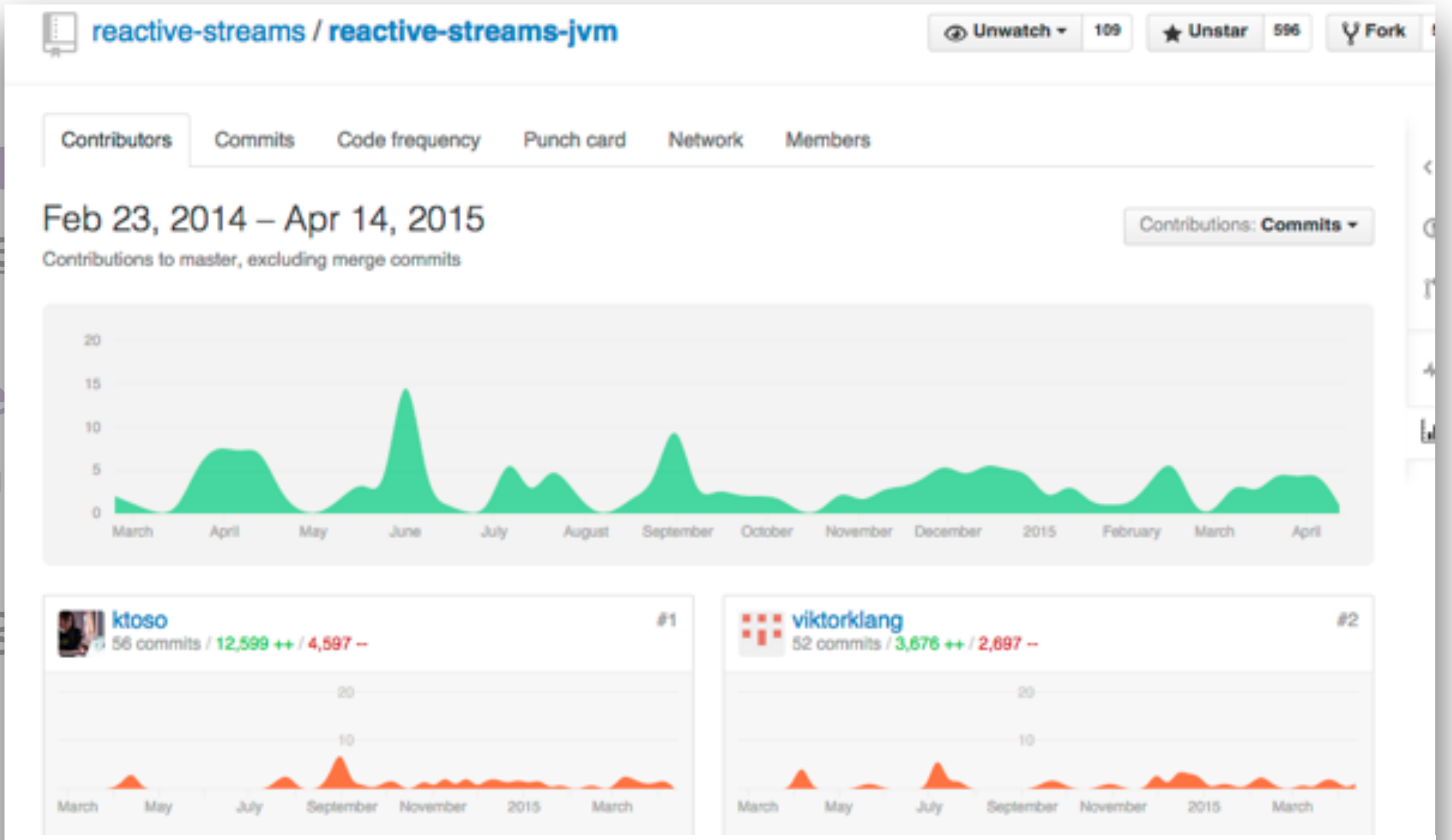
October 2013

Roland Kuhn (Akka) and Erik M
while recording "Principles of Re

Viktor Klang (Akka), Erik Meije
and Marius Eriksen (Twitter) m

The term "reactive non-blocking

December 2013
Stephane Maldini & Jon Brisbin


reactive-streams / reactive-streams-jvm

Feb 23, 2014 – Apr 14, 2015
Contributions to master, excluding merge commits

ktoso #1
56 commits / 12,599 ++ / 4,597 --

viktorklang #2
52 commits / 3,676 ++ / 2,697 --

I ended up implementing much of the TCK.
Please use it, let me know if it needs improvements :-)

Soon after, the "Reactive Streams" expert group is formed.

Also joining the efforts: Doug Lea (Oracle), Endre Varga (Akka), Johannes Rudolph & Mathias Doenitz (Spray), and many others, including myself join the effort soon after.

Typesafe

Reactive Streams - story: 2013's impls

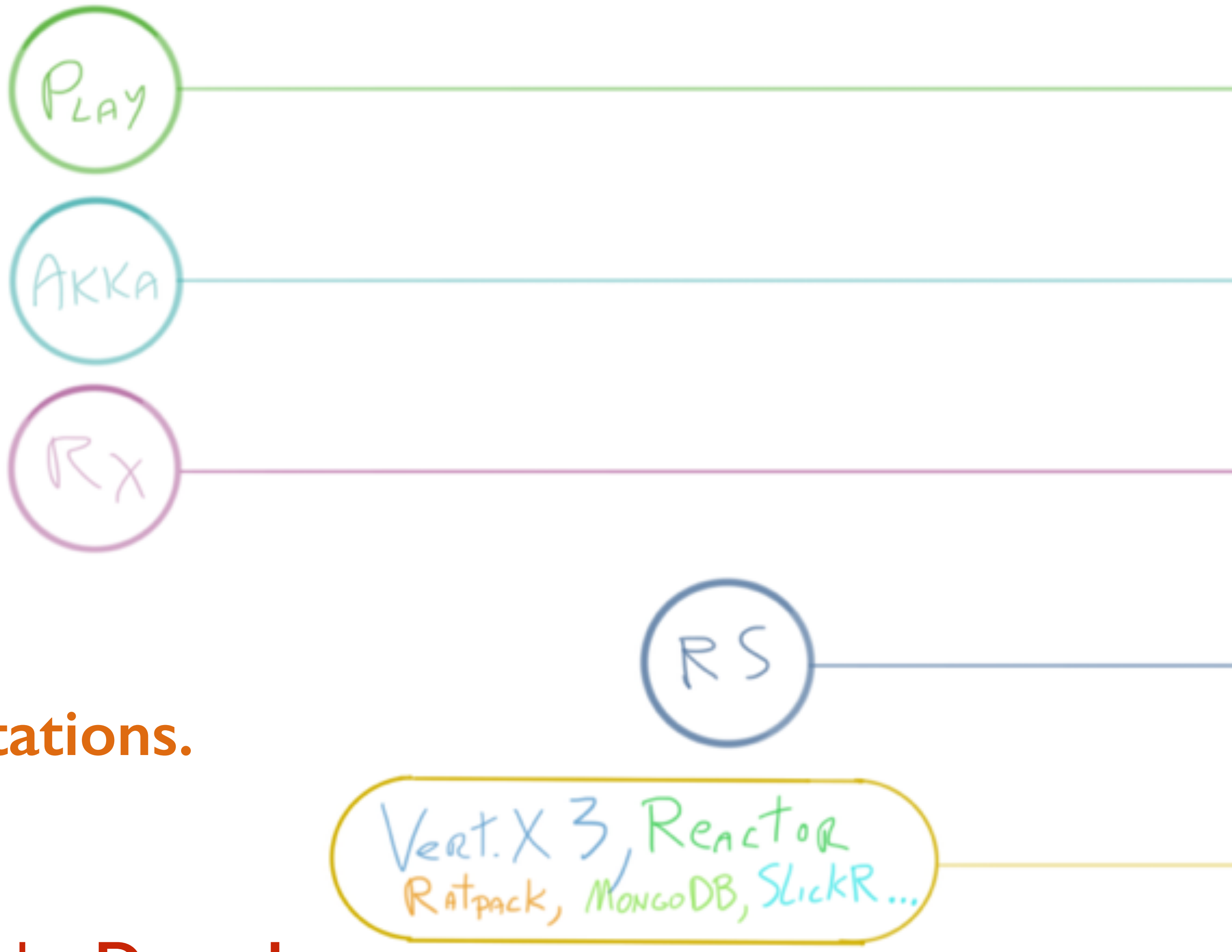
2014–2015:

Reactive Streams Spec & TCK
development, and implementations.

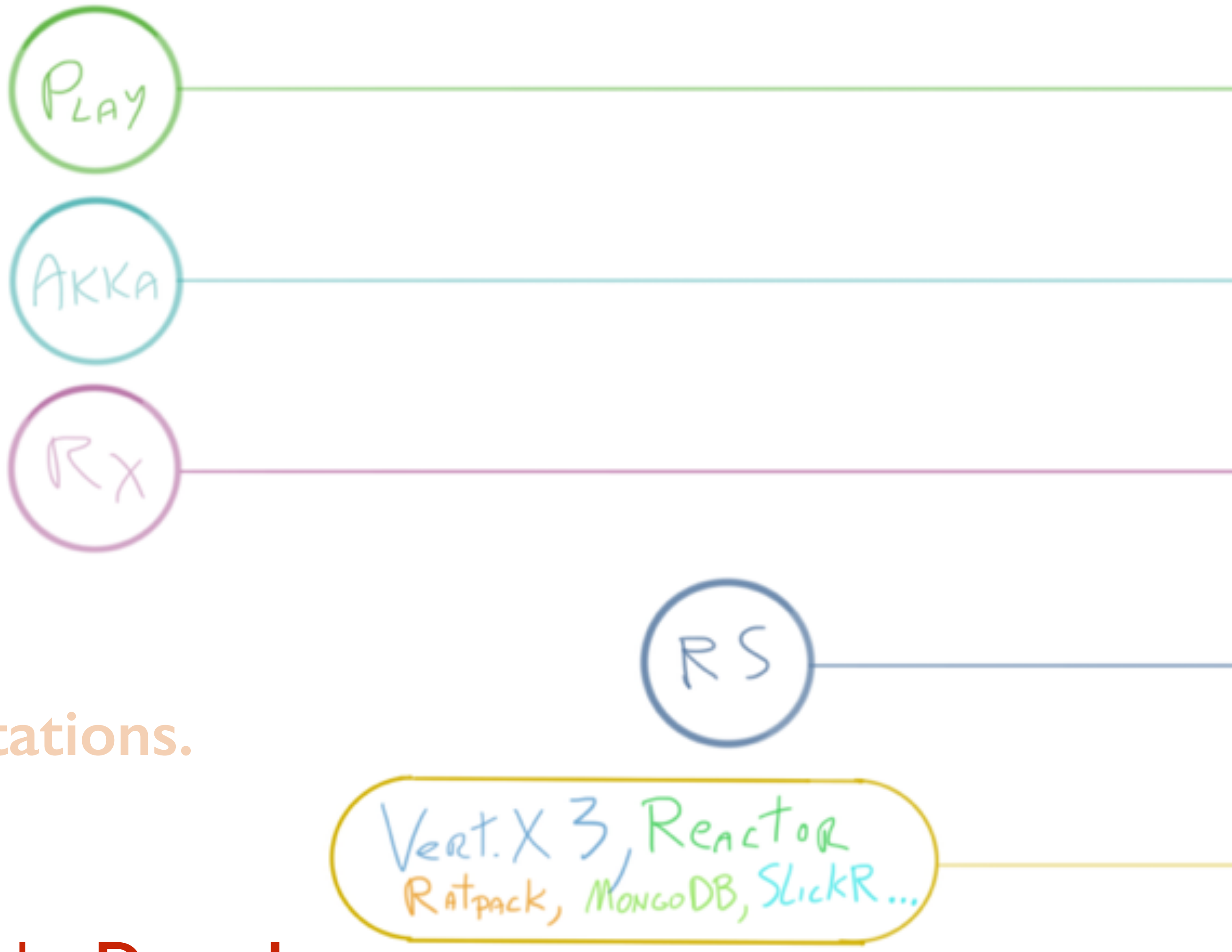
**1.0 released on April 28th 2015,
with 5+ accompanying implementations.**

2015
Proposed to be included with **JDK9** by **Doug Lea**
via **JEP-266 "More Concurrency Updates"**

http://hg.openjdk.java.net/jdk9/jdk9/jdk/file/6e50b992bef4/src/java.base/share/classes/java/util/concurrent/Flow.java



Reactive Streams - story: 2013's impls



2014–2015:

Reactive Streams Spec & TCK development, and implementations.

1.0 released on April 28th 2015, with 5+ accompanying implementations.

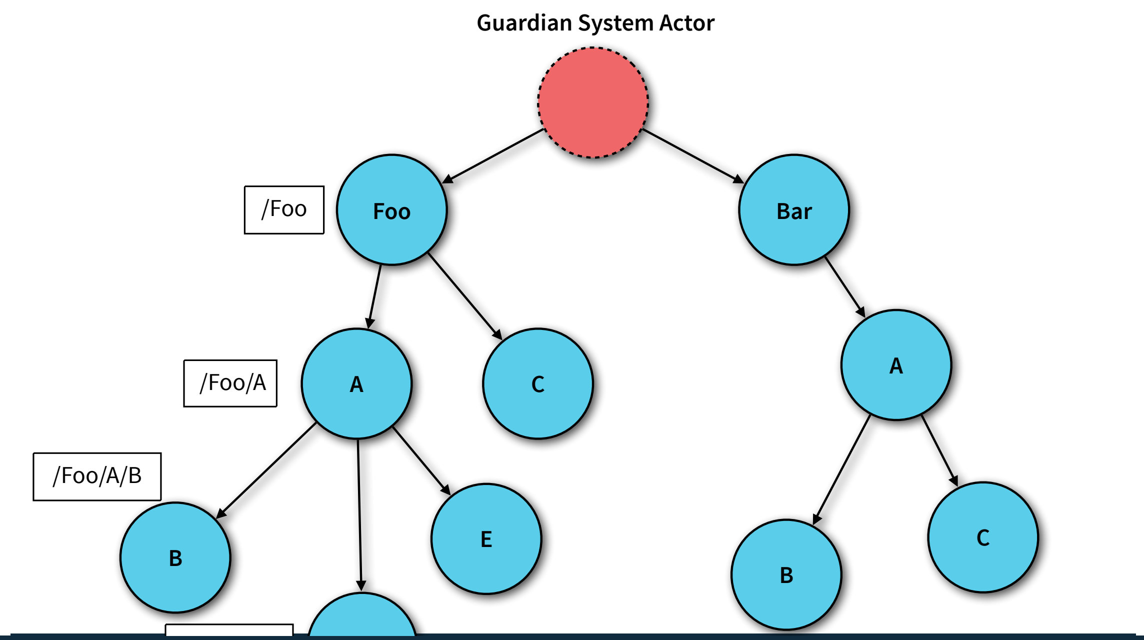
2015

Proposed to be included with **JDK9** by **Doug Lea** via **JEP-266 "More Concurrency Updates"**

http://hg.openjdk.java.net/jdk9/jdk9/jdk/file/6e50b992bef4/src/java.base/share/classes/java/util/concurrent/Flow.java

akka in a few words:

- Toolkit for building **scalable distributed / concurrent apps.**
- High Performance Actor Model implementation
  - "share nothing" – messaging instead of sharing state
  - millions of msgs, per actor, per second
- Supervision trees – built-in and mandatory
- **Clustering** and **Http** built-in



Why back-pressure?

?

Why back-pressure?

So you've built your app and it's awesome.

Why back-pressure?

Let's not smash it horribly **under load.**

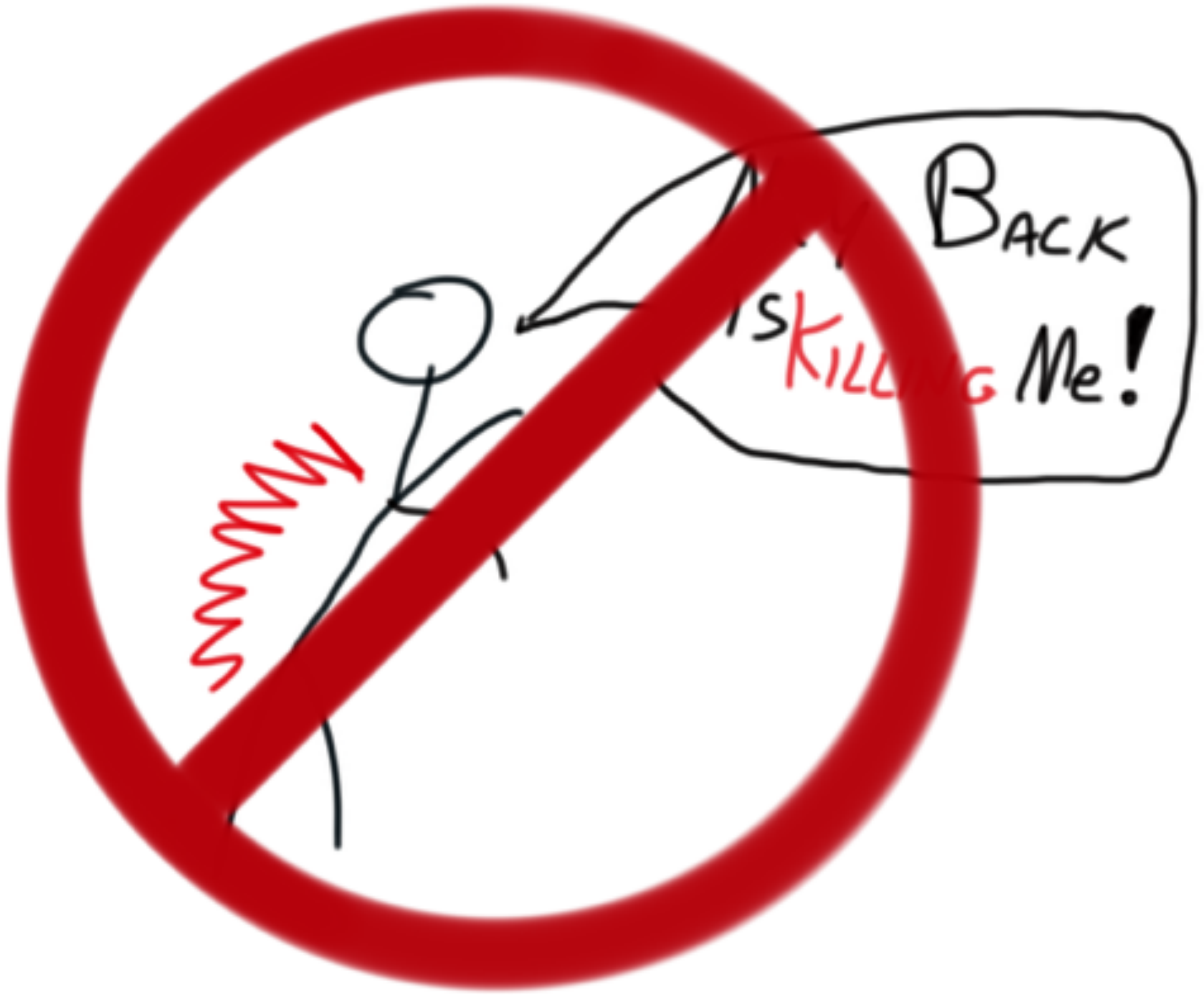
What is back-pressure?

?

What is back-pressure?

What is back-pressure?

No no no…!
Not THAT Back-pressure!



Typesafe

Back-pressure explained

Publisher[T]

Subscriber[T]

What if…?

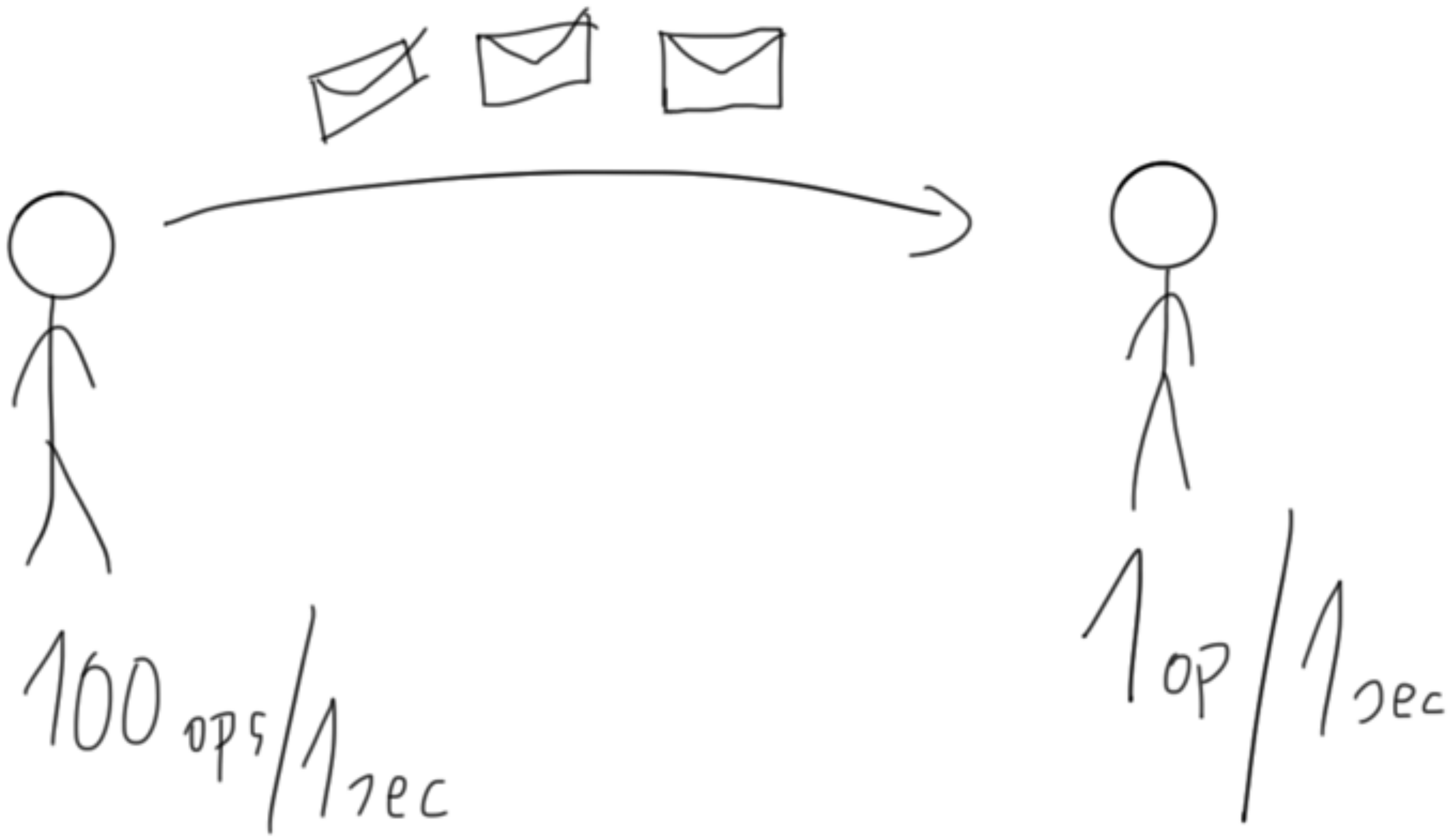
Fast Publisher

Slow Subscriber

100 ops / 1 sec

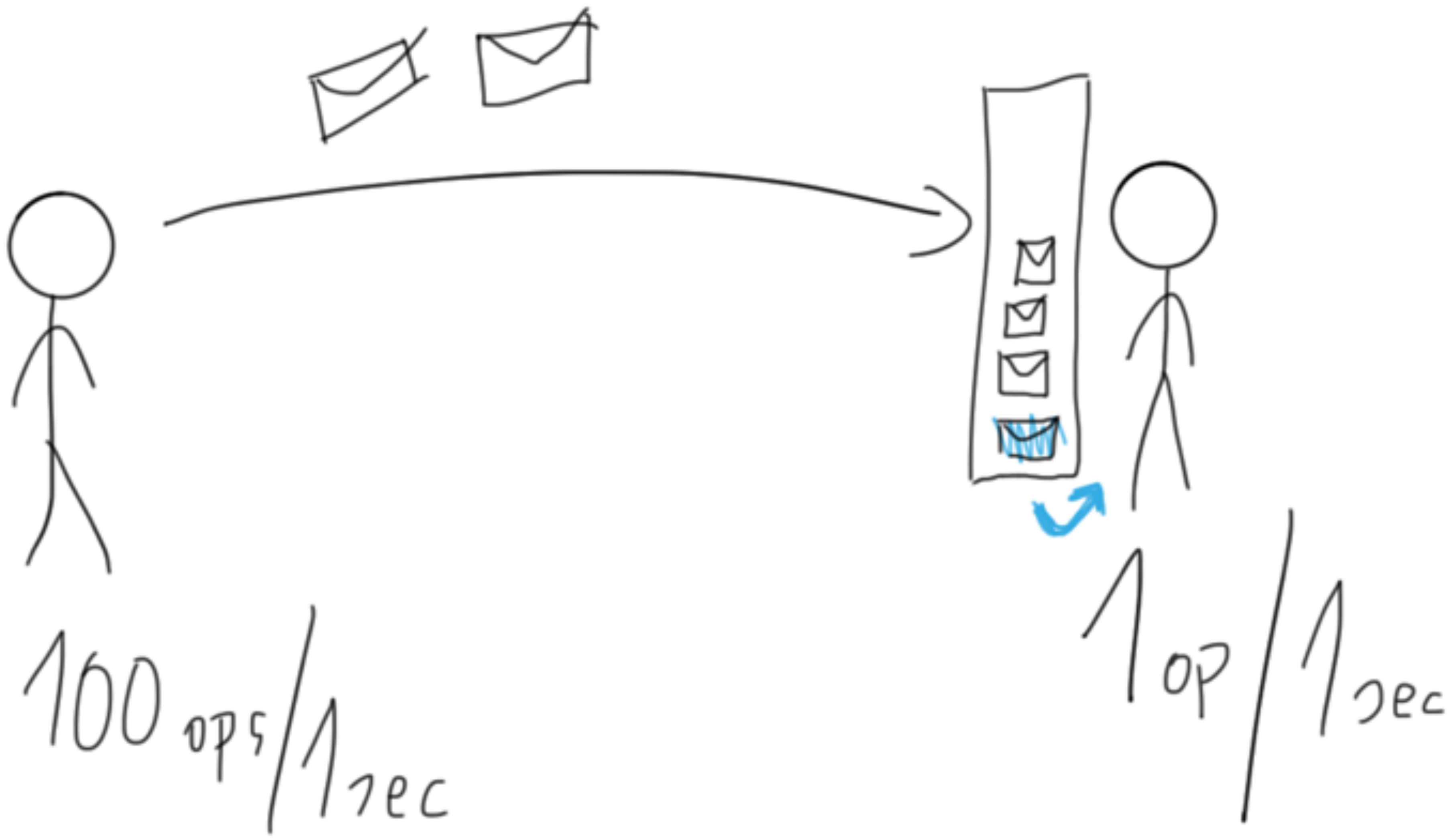
Typesafe

Push + NACK model

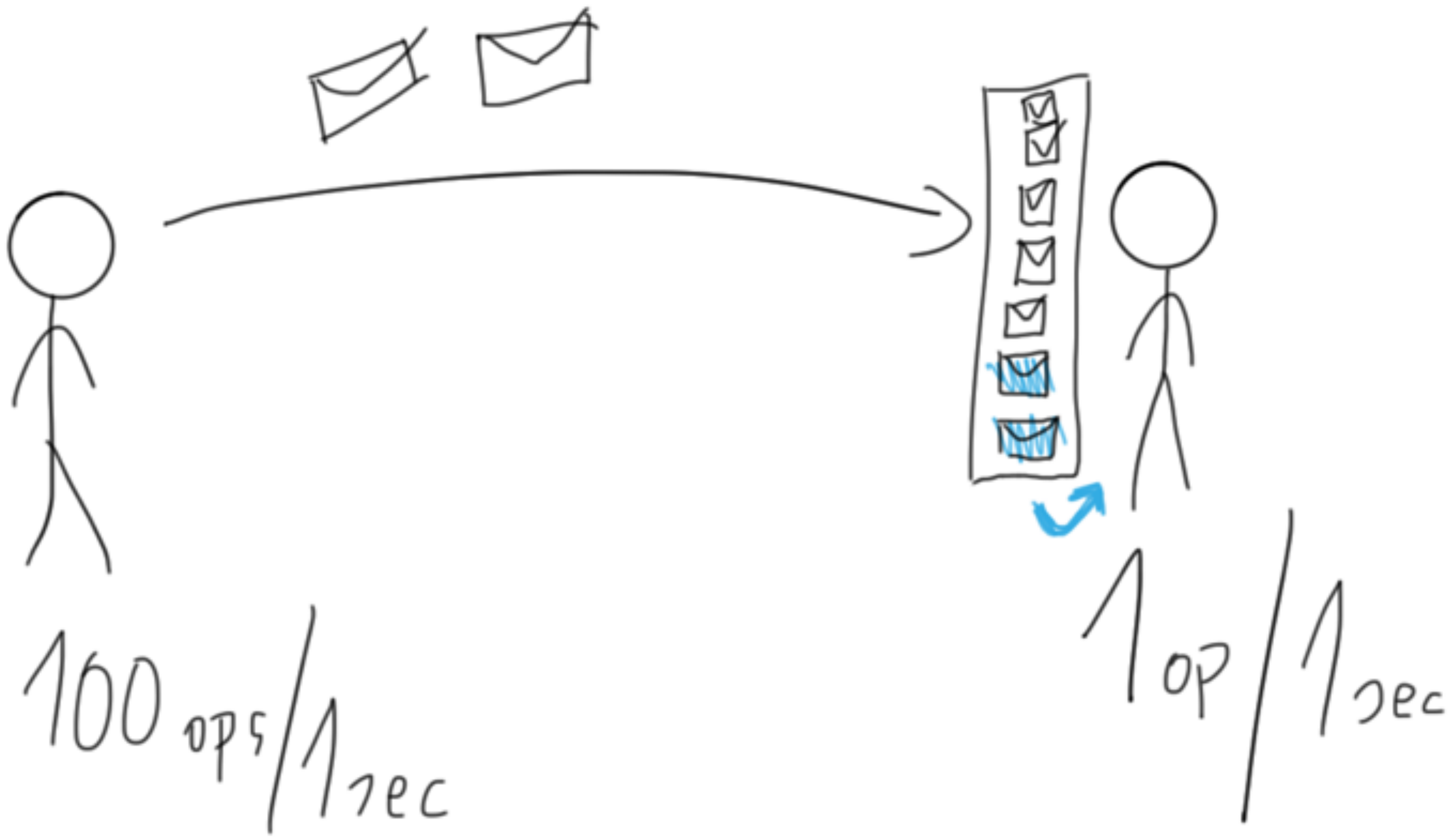


Push + NACK model

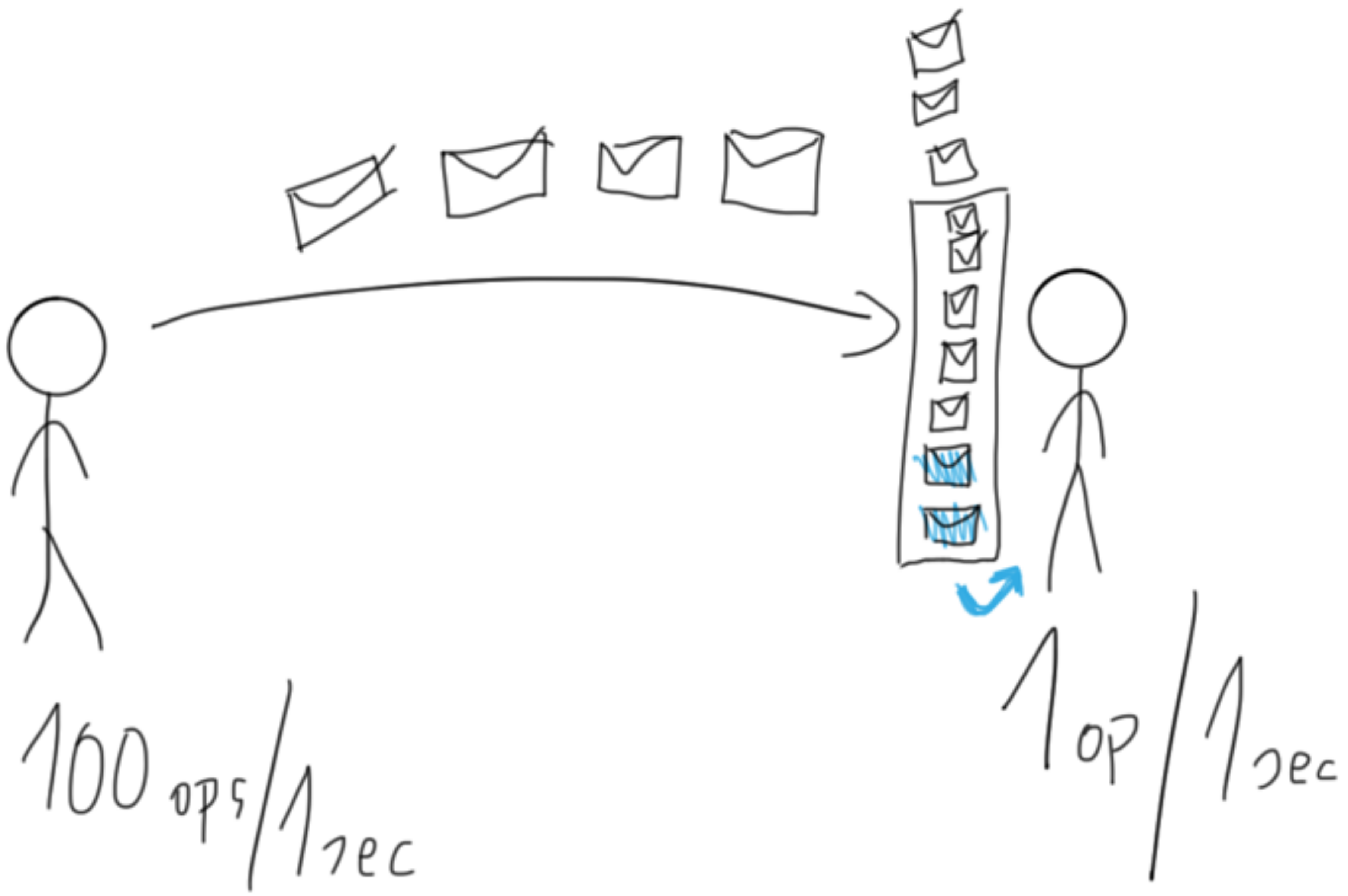
Subscriber usually has some kind of buffer.



Push + NACK model



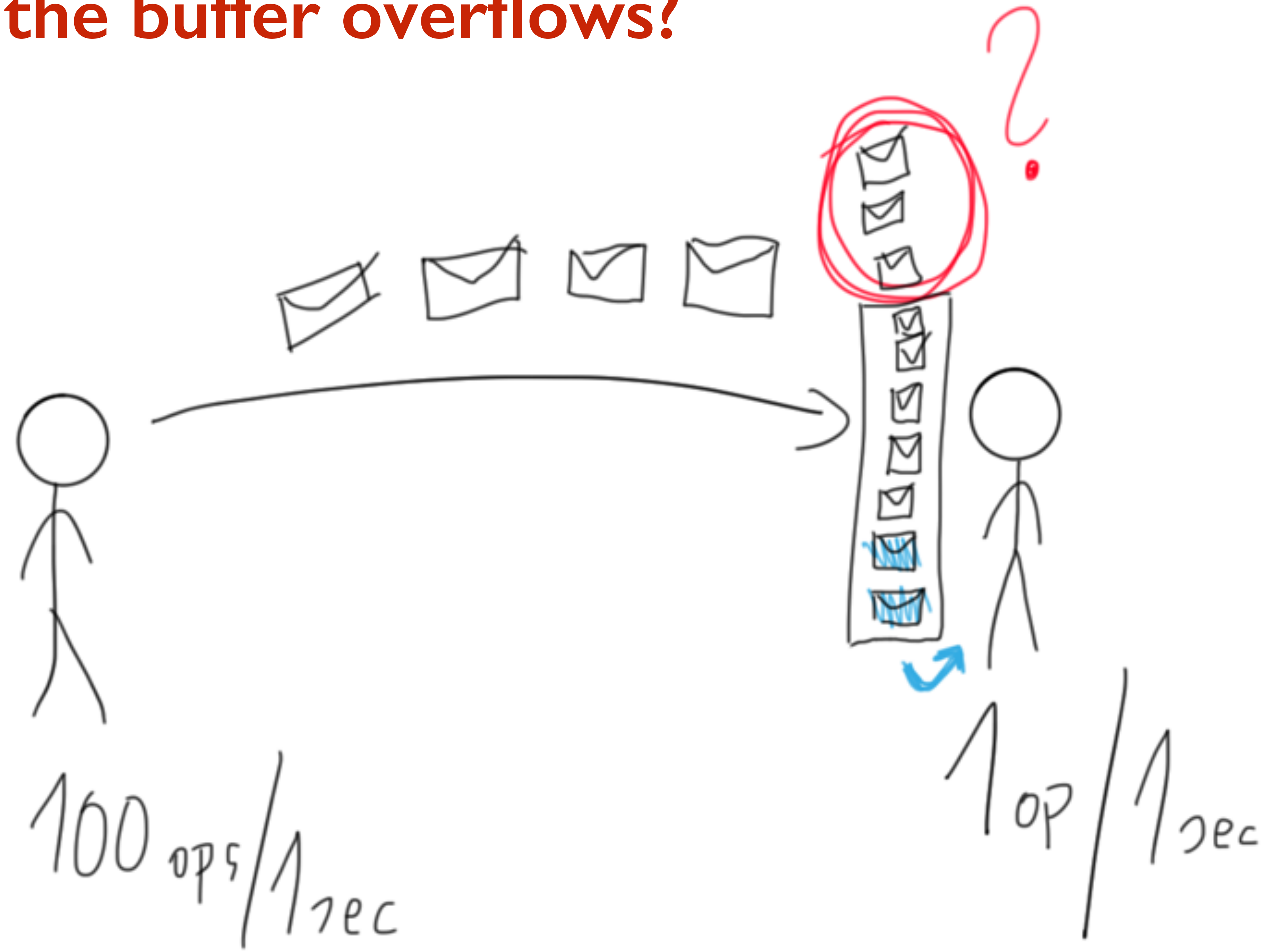
Push + NACK model



Typesafe

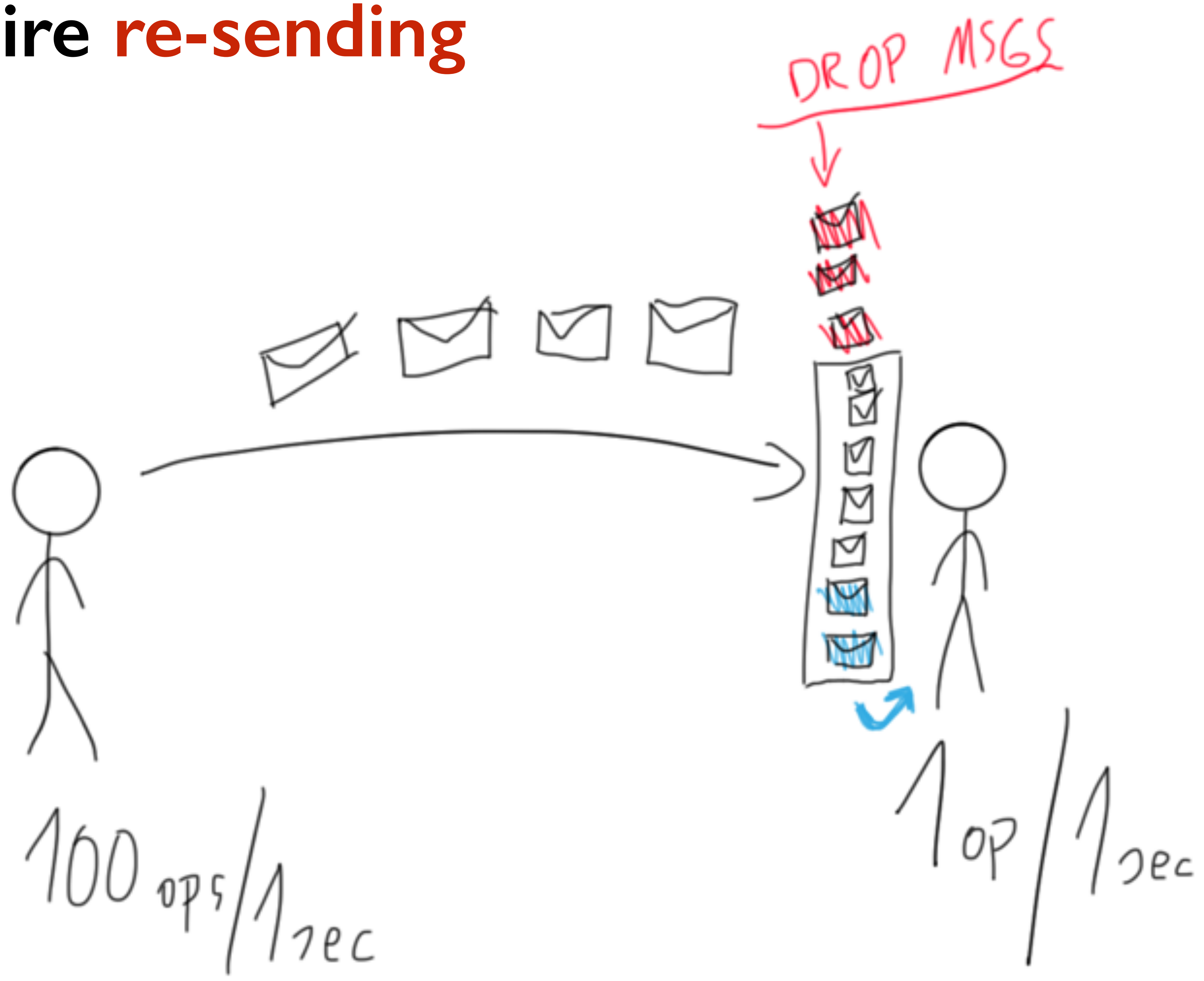
Push + NACK model

What if the buffer overflows?



Push + NACK model

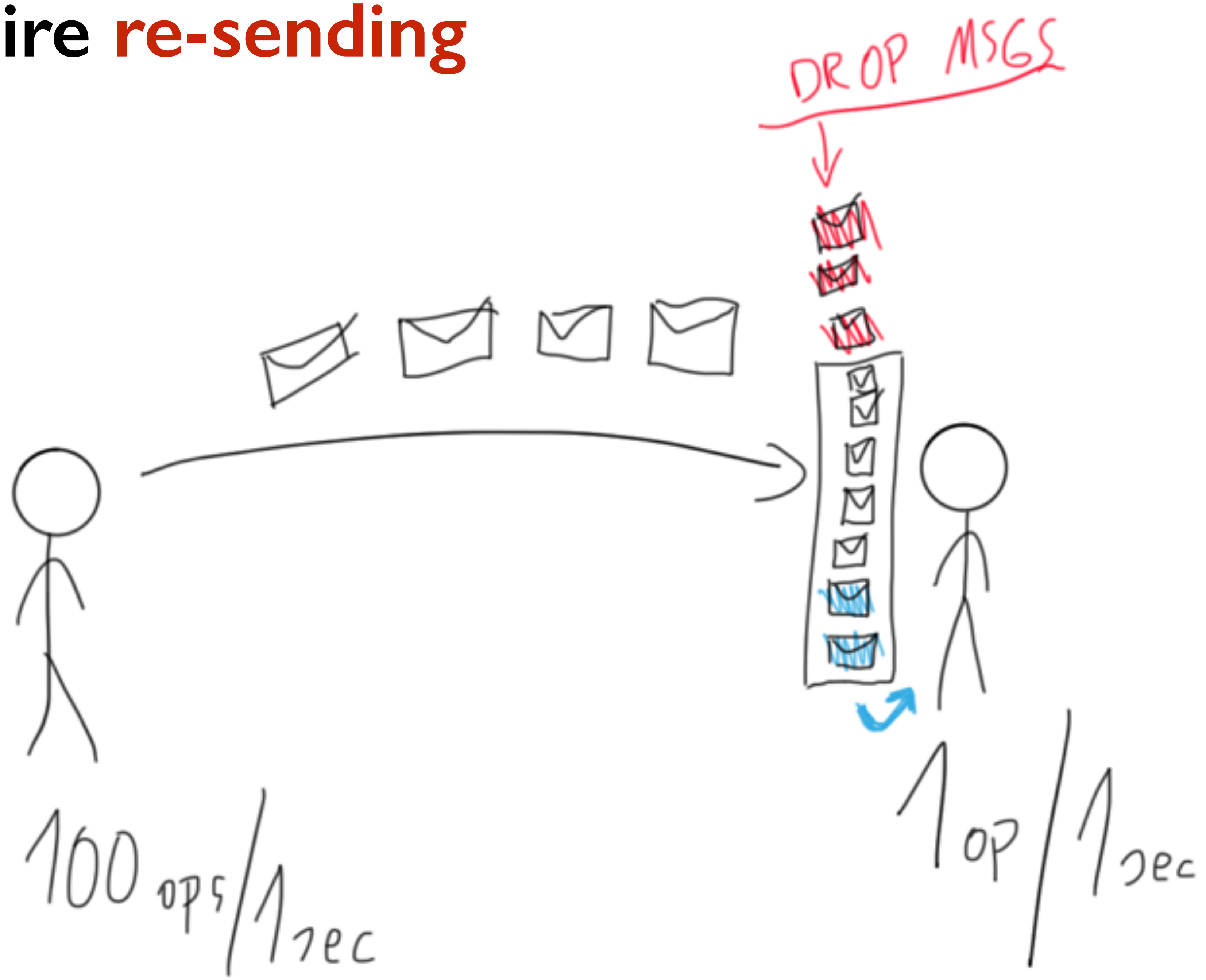
Use bounded buffer,
drop messages + require re-sending



Push + NACK model

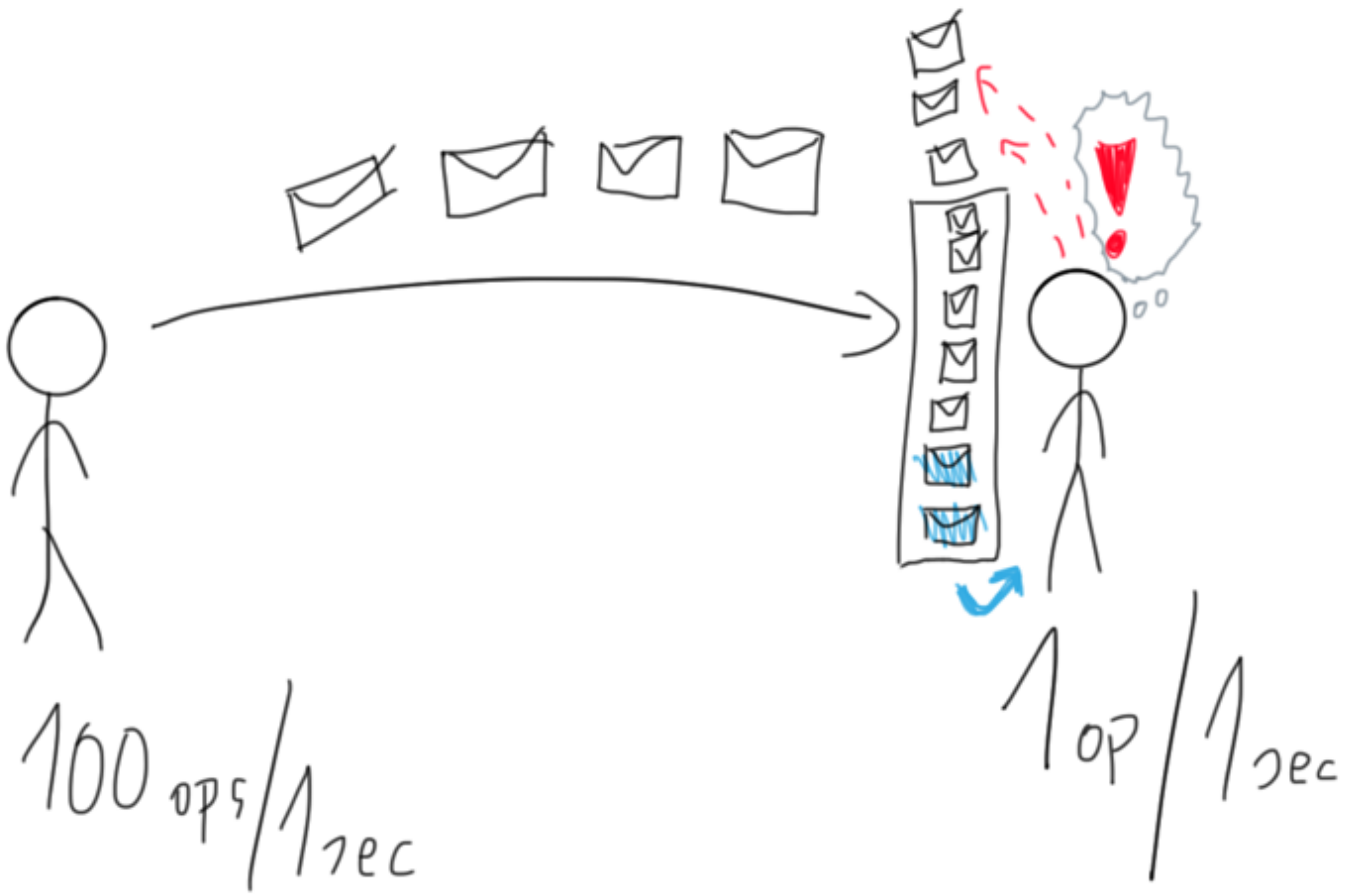
Use bounded buffer,
drop messages + require re-sending

Kernel does this!
Routers do this!
(TCP)

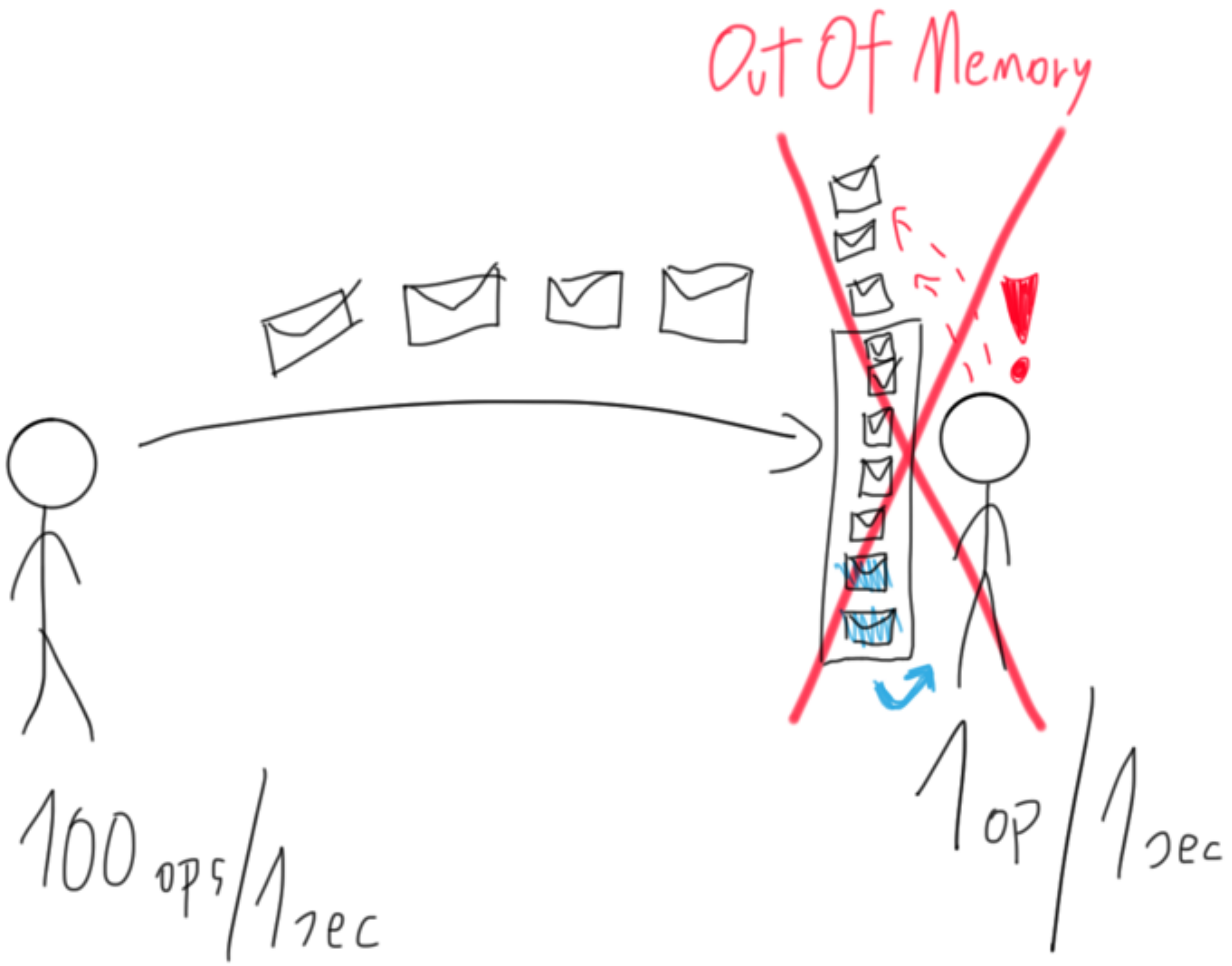


Push + NACK model

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



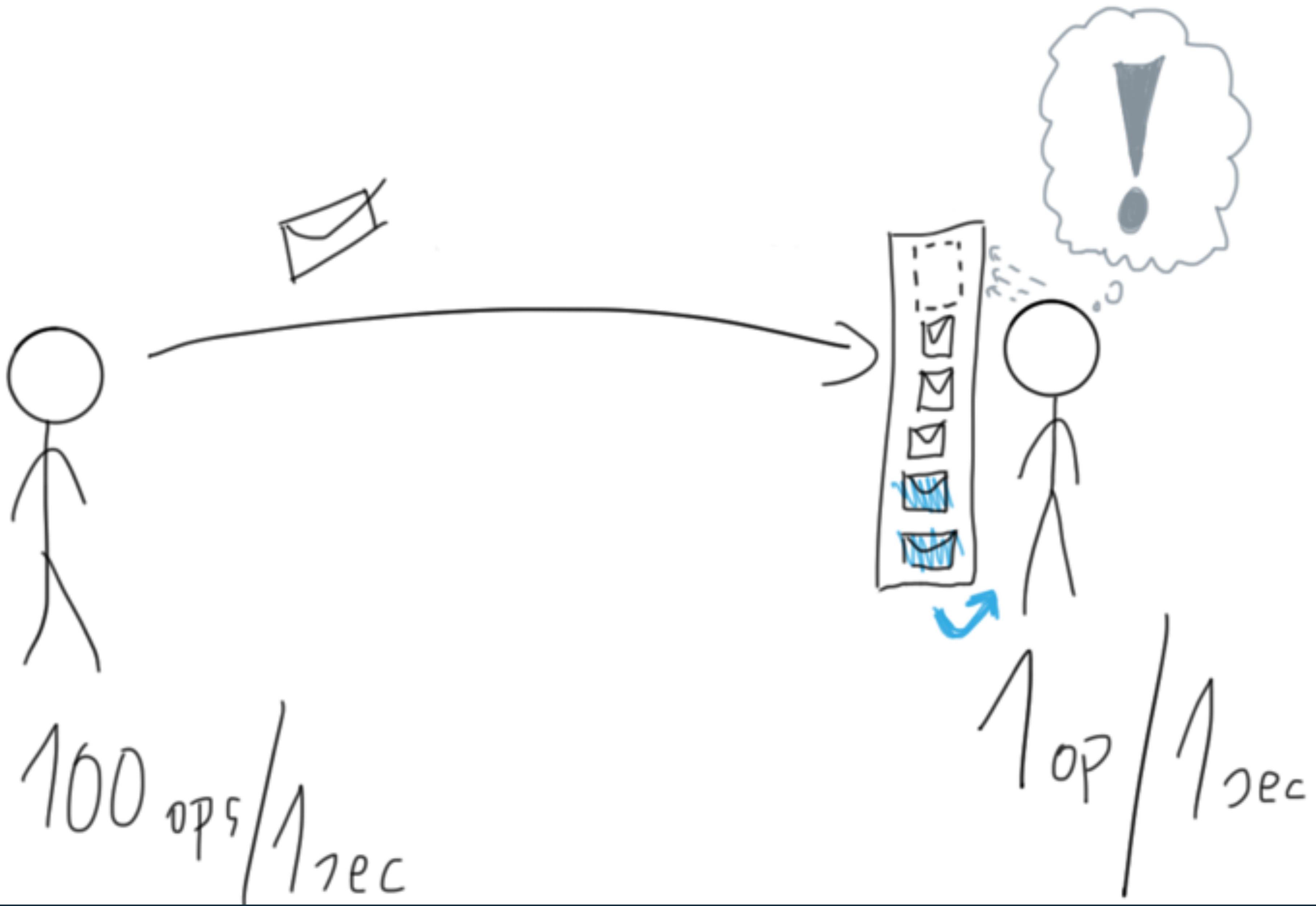
Push + NACK model



Negative **ACK**nowledgement

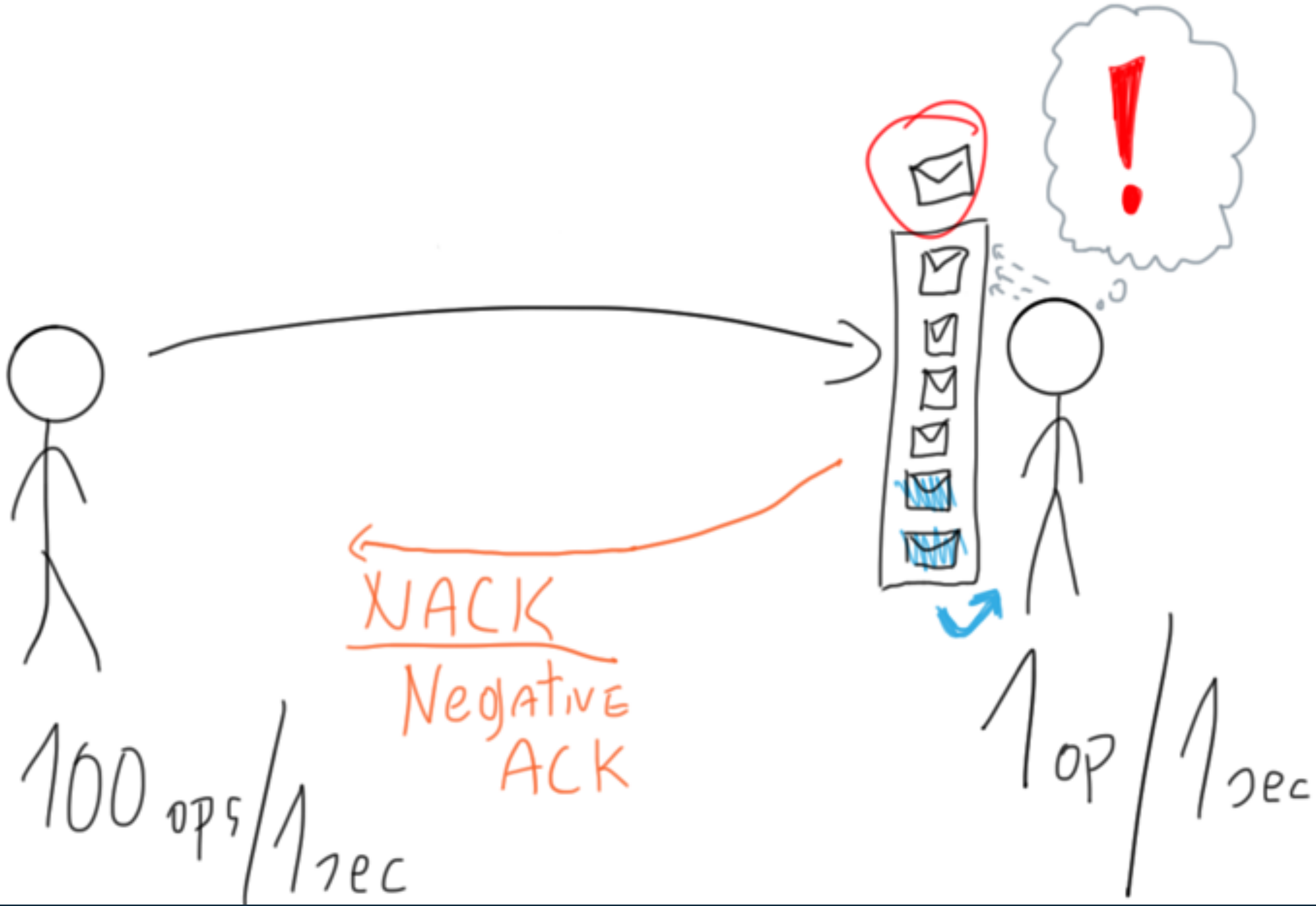
NACKing

Buffer overflow is imminent!



NACKing

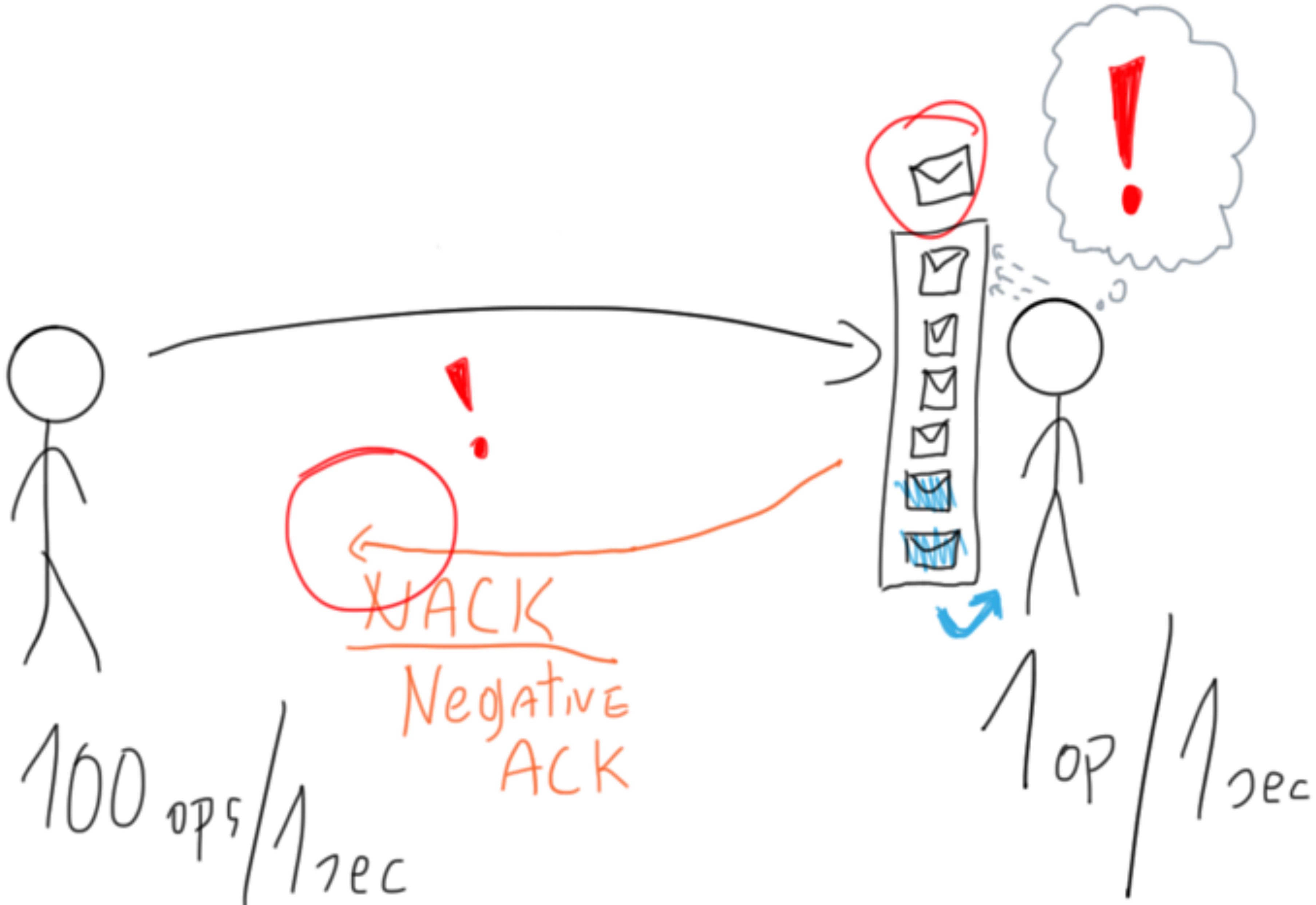
Telling the Publisher to slow down / stop sending…



NACKing

NACK did not make it in time,

because M was in-flight!



We need low-overhead for “happy case”

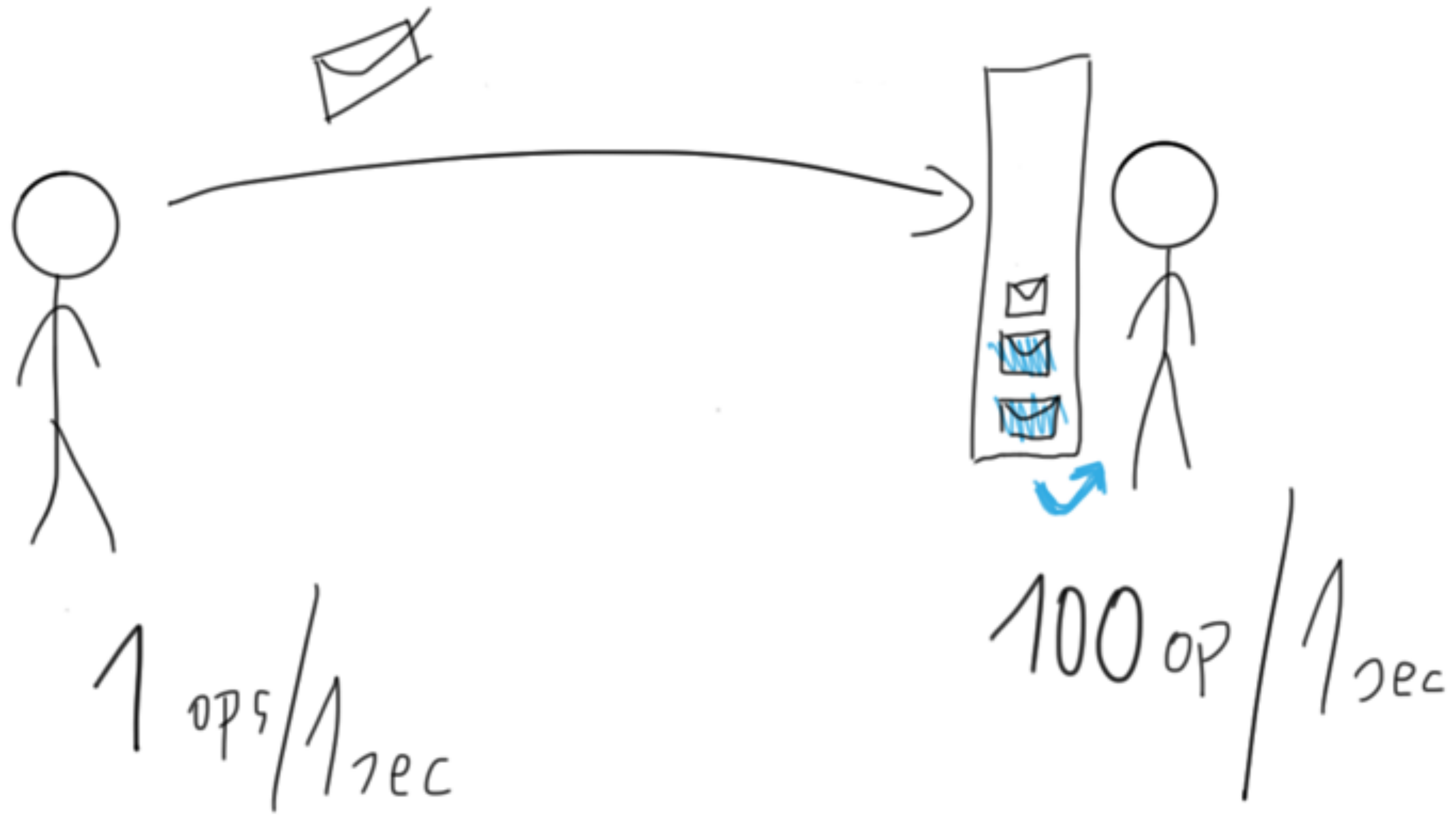
What if…

We don’t need to back-pressure, because:

speed(publisher) < speed(subscriber)

Fast Subscriber => no problem

No problem!



Typesafe

Fast Subscriber => no problem

Back-pressure?

Reactive-Streams

=

“Dynamic Push/Pull”

Reactive Streams: “dynamic push/pull”

Just push – not safe when Slow Subscriber

Just pull – too slow when Fast Subscriber

Reactive Streams: “dynamic push/pull”

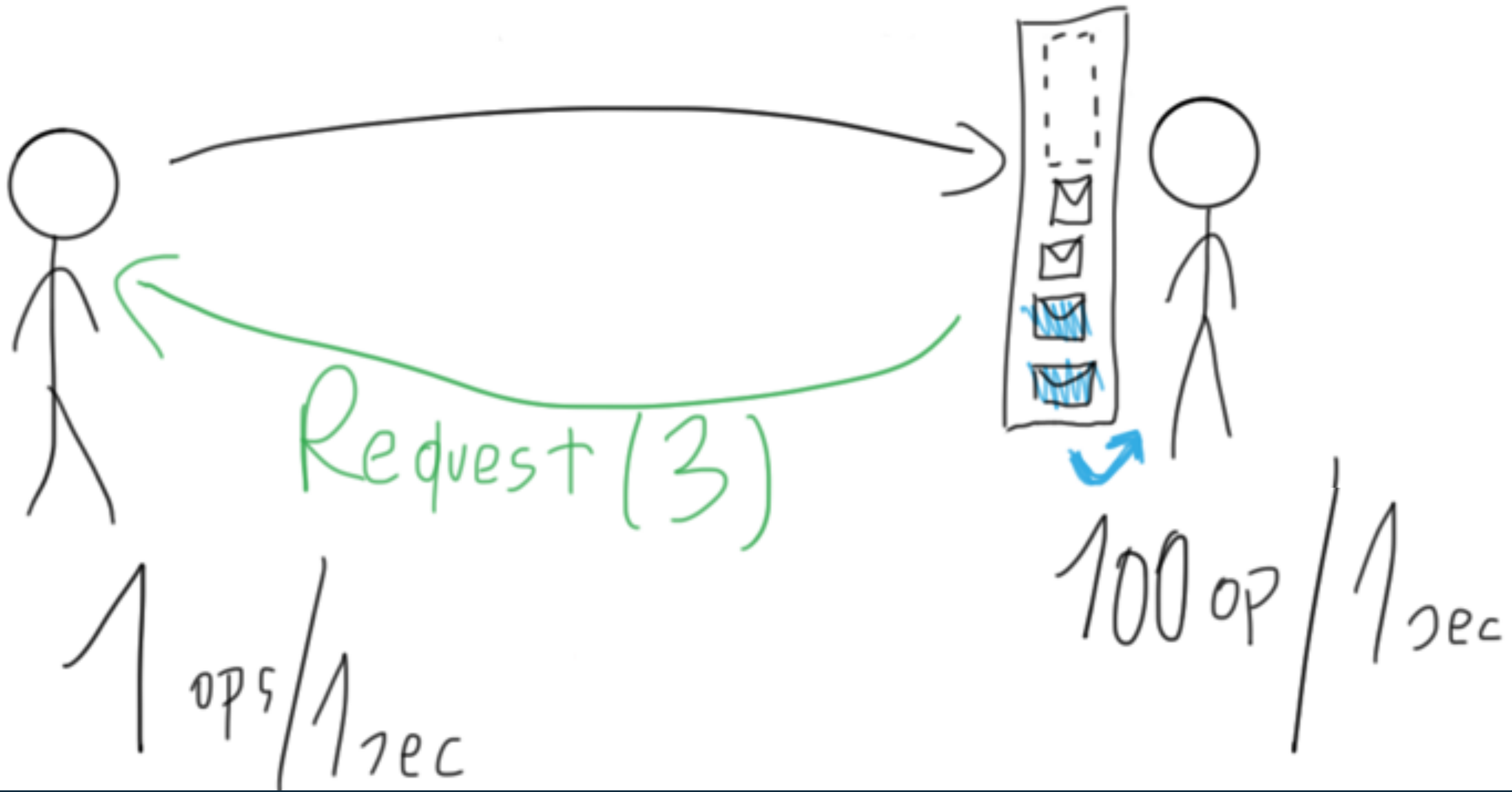
Just push – not safe when Slow Subscriber

Just pull – too slow when Fast Subscriber

Solution:
Dynamic adjustment

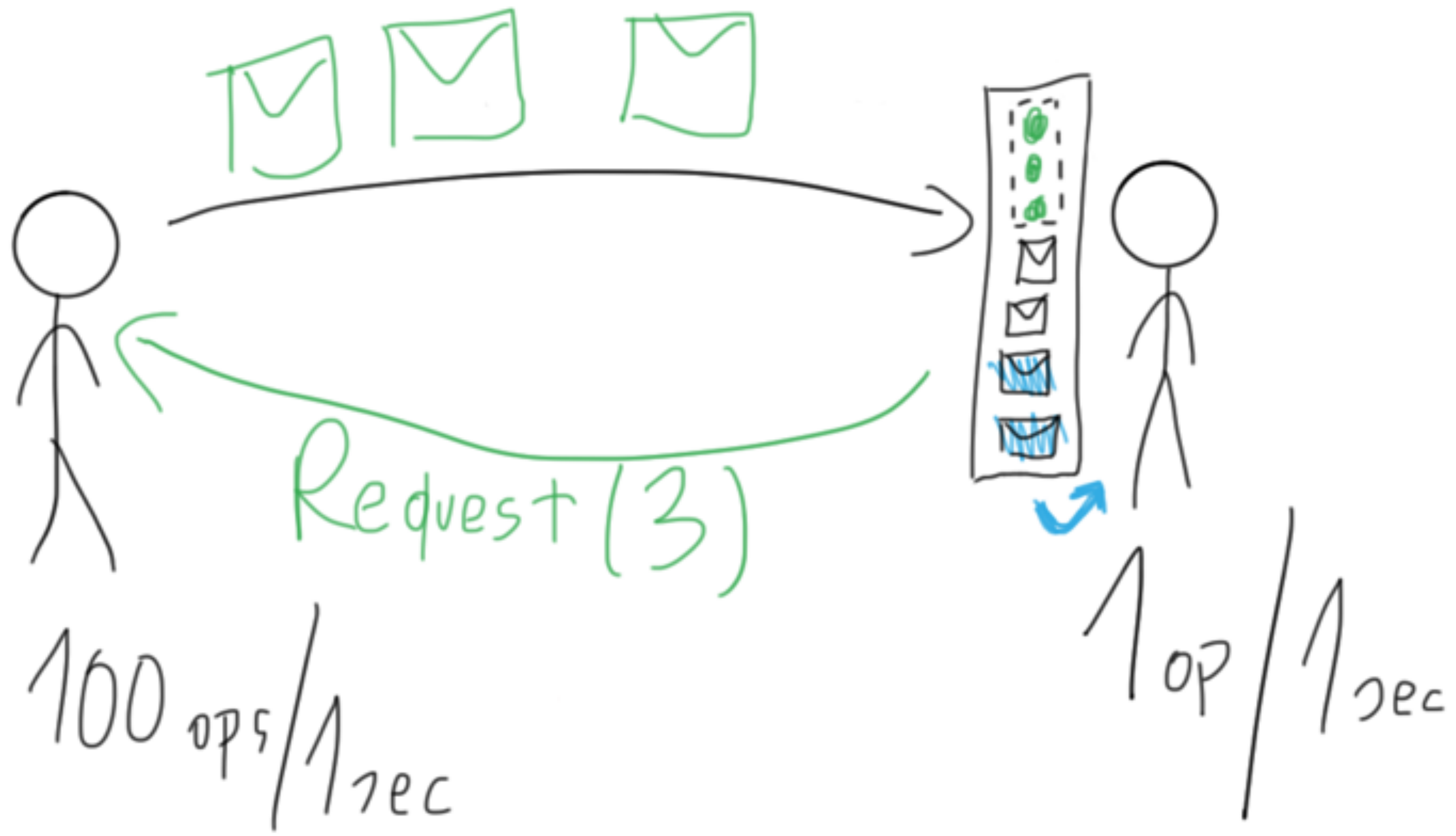
Reactive Streams: “dynamic push/pull”

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



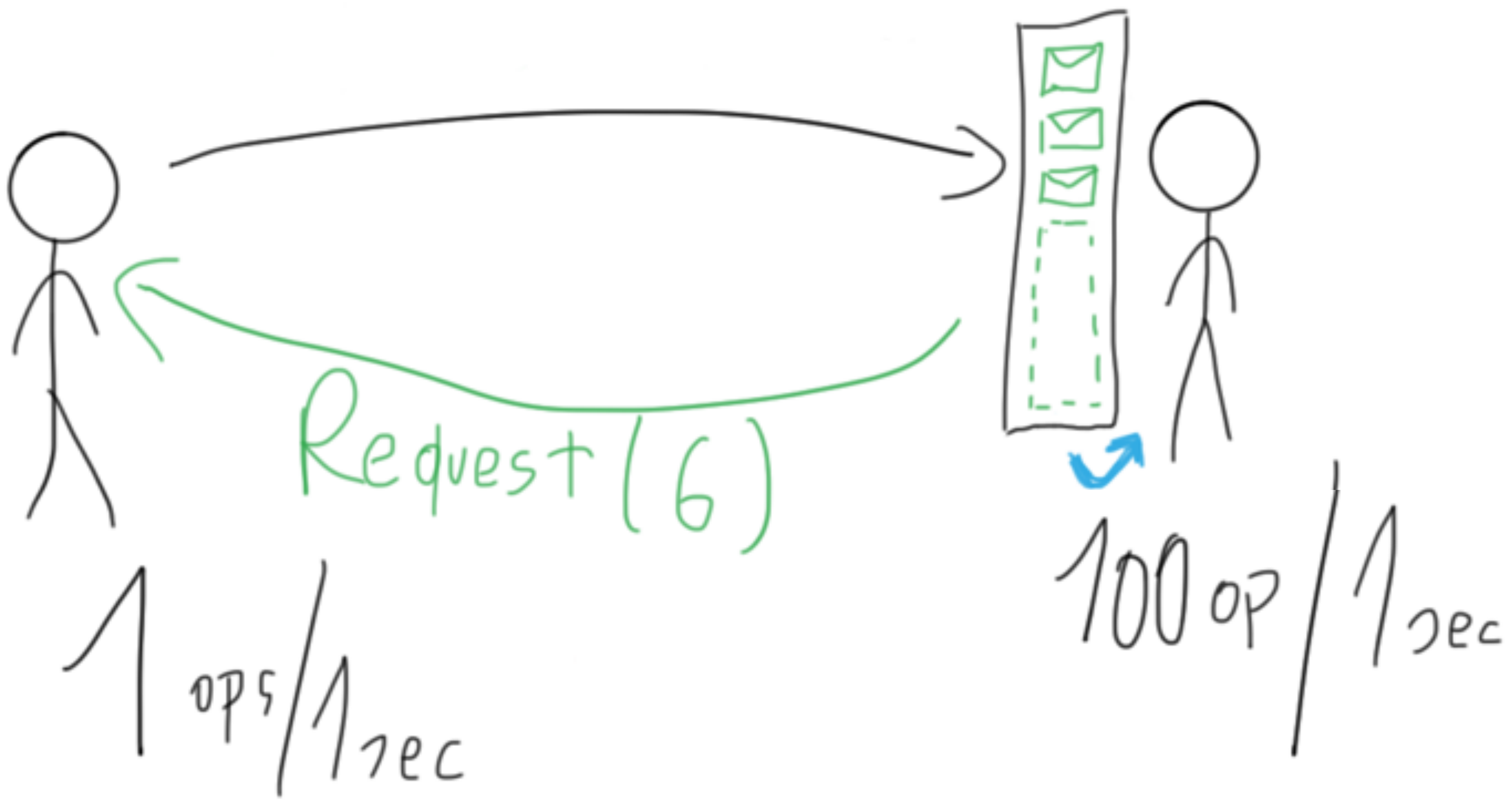
Reactive Streams: “dynamic push/pull”

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



Reactive Streams: “dynamic push/pull”

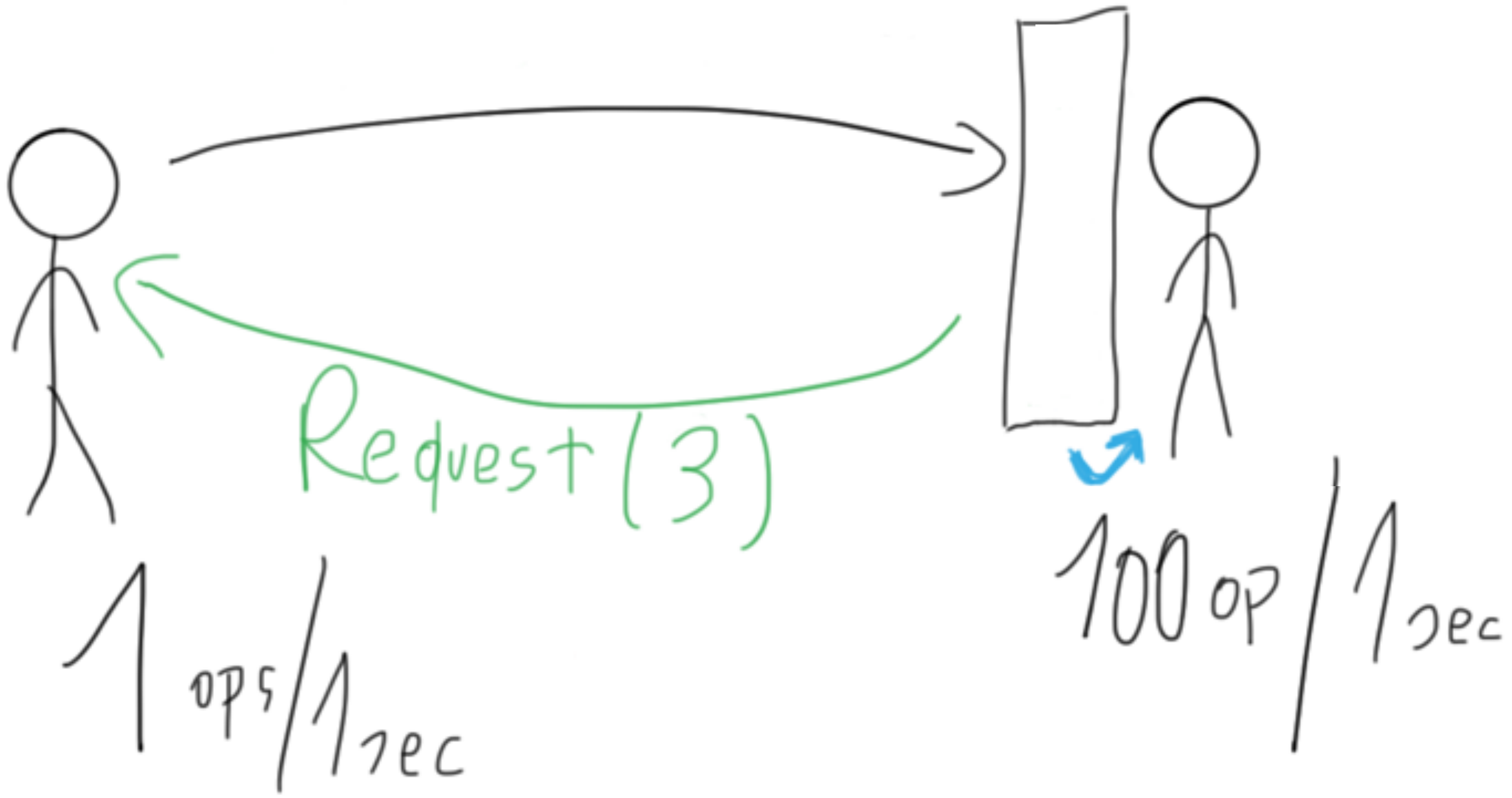
Fast Subscriber can issue more Request(n), before more data arrives!



Reactive Streams: “dynamic push/pull”

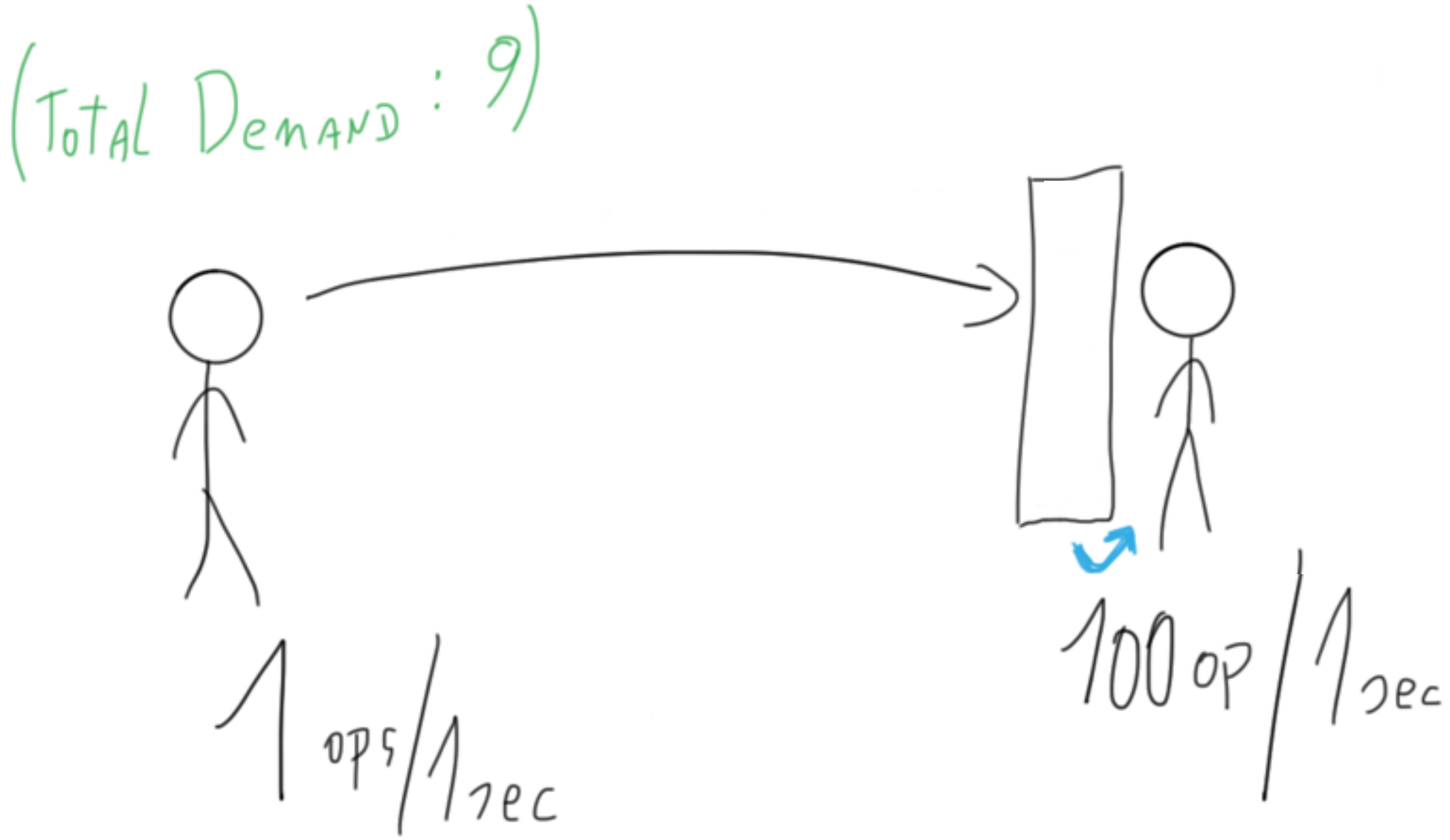
Fast Subscriber can issue more Request(n), before more data arrives.

Publisher can accumulate demand.



Reactive Streams: accumulate demand

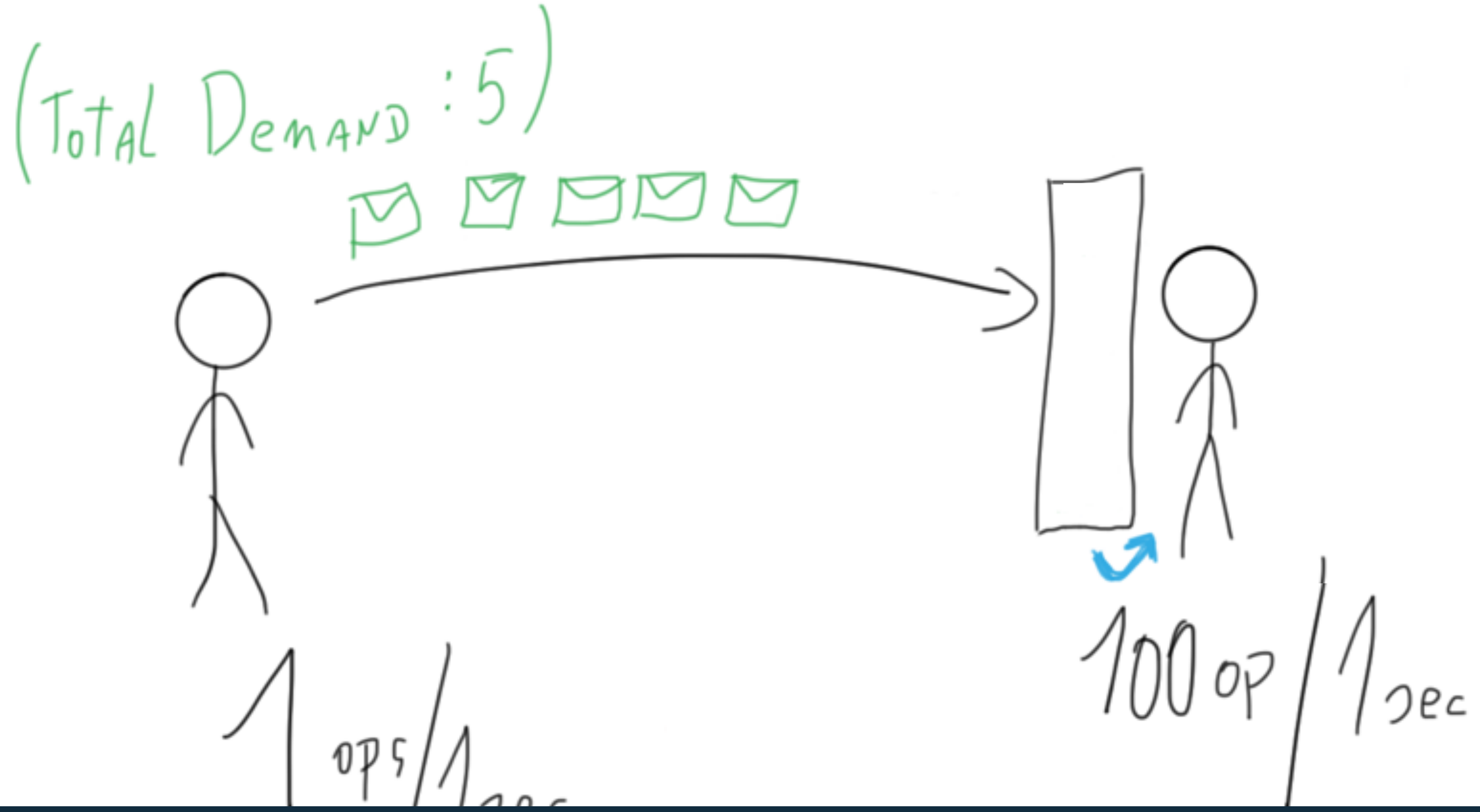
Publisher accumulates total demand per subscriber.



Reactive Streams: accumulate demand

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

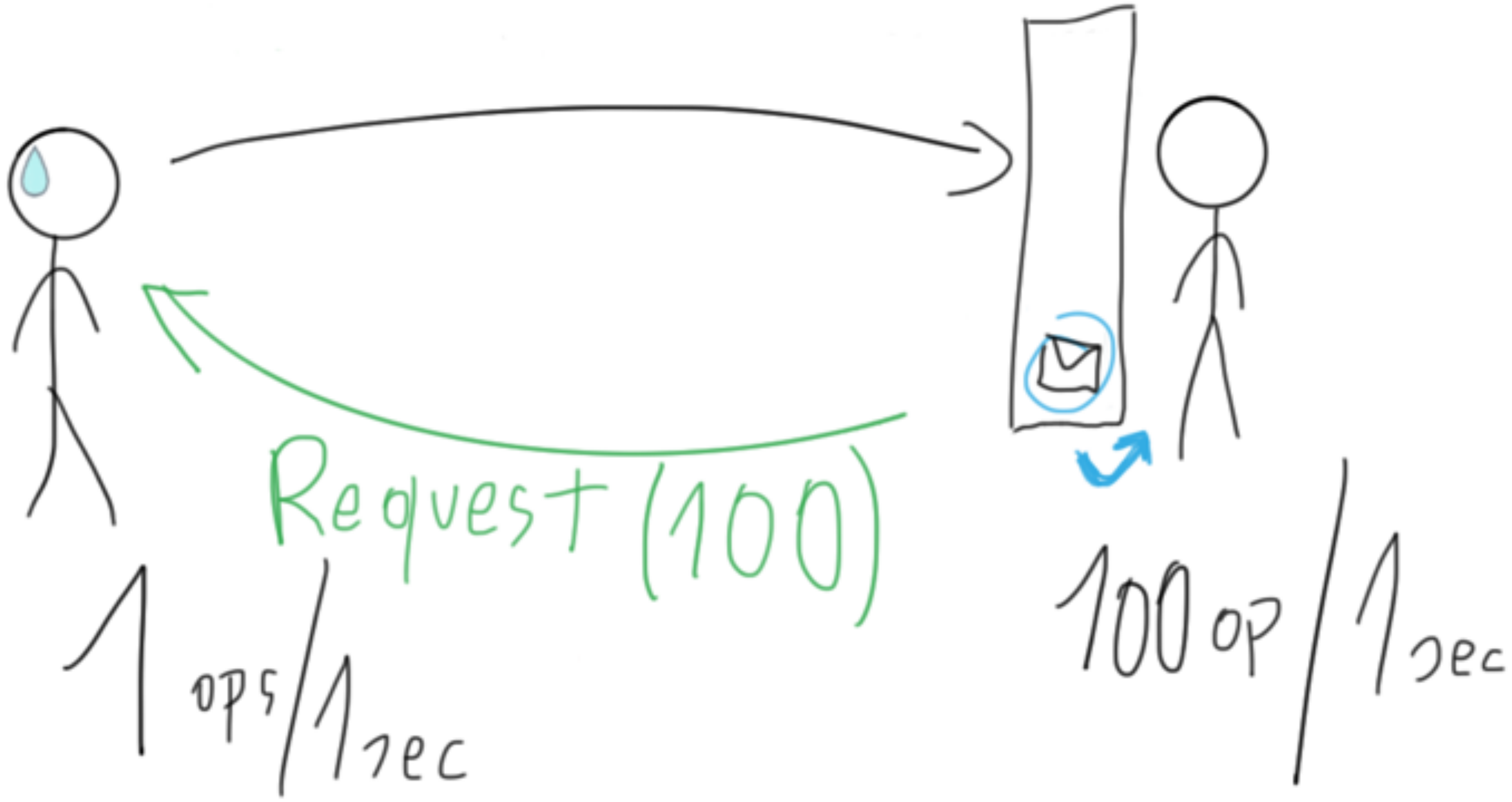
Subscriber's buffer will not overflow.



Typesafe

Reactive Streams: requesting “a lot”

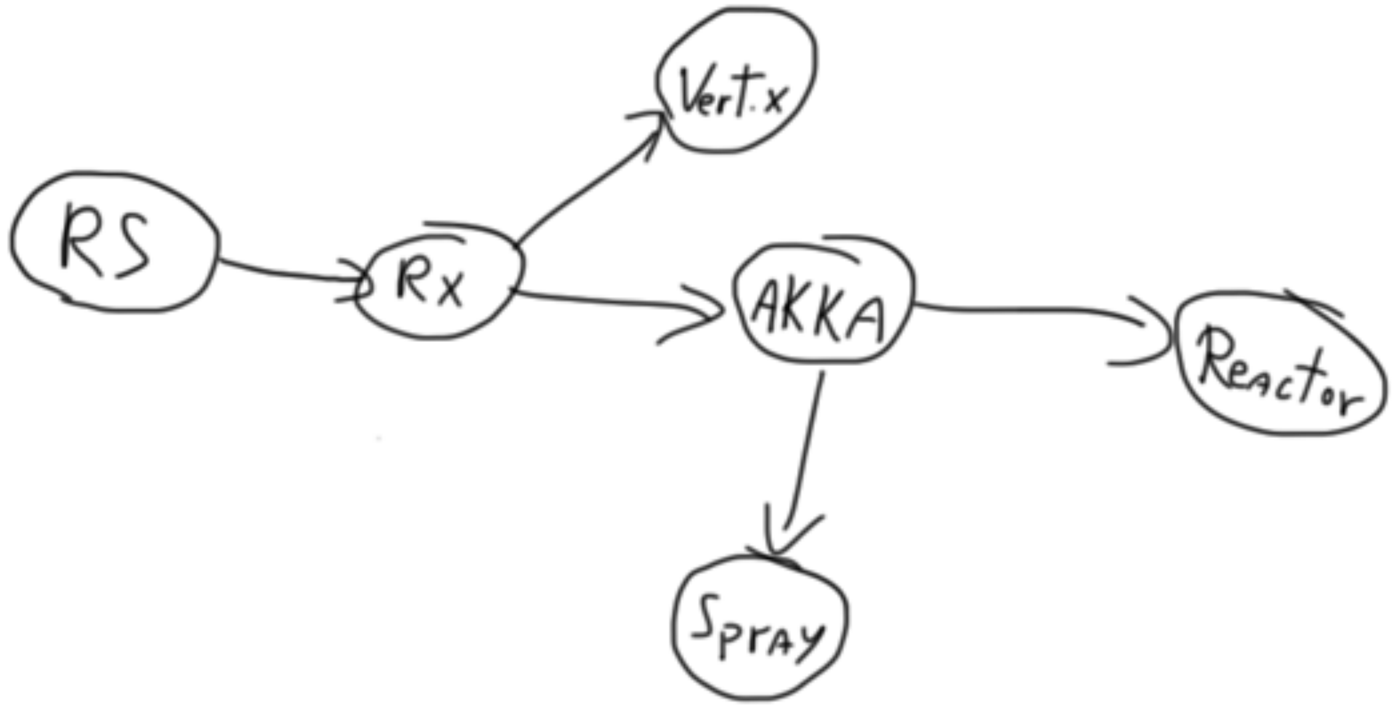
Fast Subscriber can issue arbitrary large requests, including “gimme all you got” (Long.MaxValue)



Typesafe

Reactive Streams: Inter Op

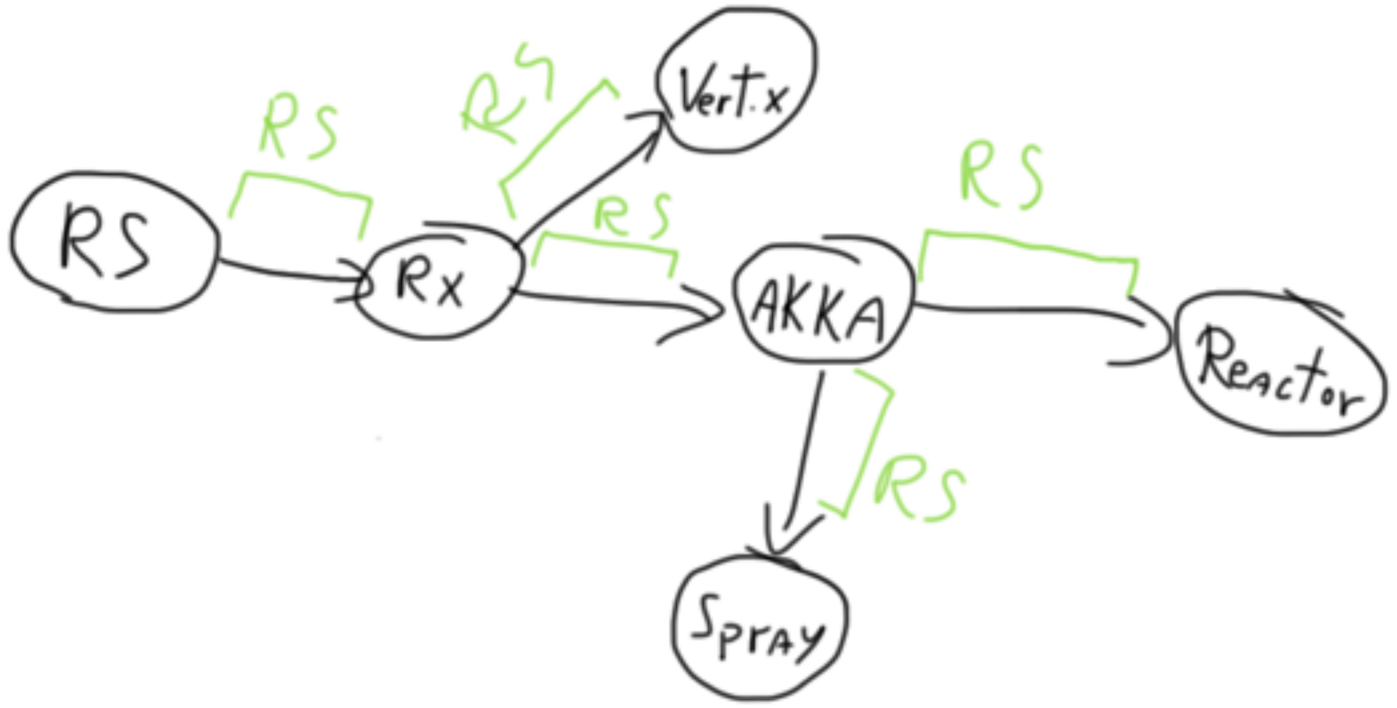
We want to make **different implementations co-operate** with each other.



http://reactive-streams.org

Reactive Streams: Inter Op

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



http://reactive-streams.org

Reactive Streams: Inter-Op

RS is NOT a "daily use", "end-user" API.

It's an SPI - Service Provider Interface.

Service Provider Interface (SPI) is an **API intended** to be **implemented or extended by a third party**.

Reactive Streams: Inter-Op

```
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))
```

https://en.wikipedia.org/wiki/Service_provider_interface

Reactive Streams: Inter-Op

```
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))
```

https://en.wikipedia.org/wiki/Service_provider_interface

Akka Streams

akka streams

Akka Streams & HTTP

akka streams
& HTTP

Akka Streams in 20 seconds:

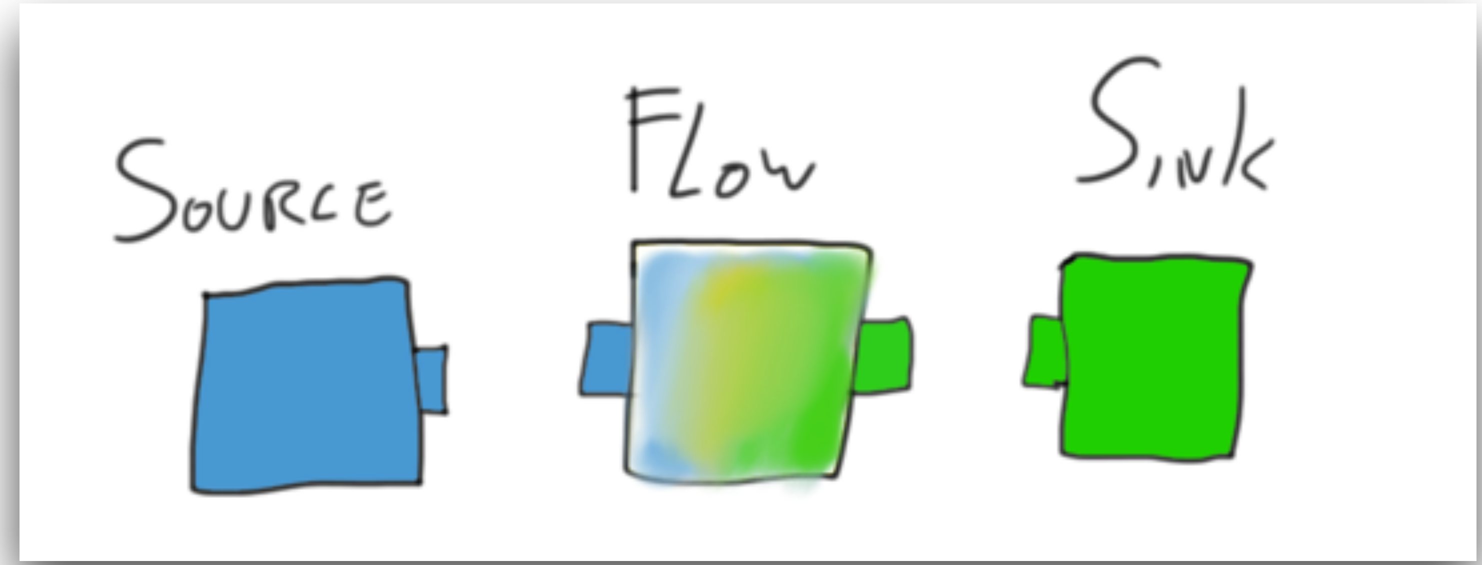
```
// types:
Source[Out, Mat]
Flow[In, Out, Mat]
Sink[In, Mat]

// generally speaking, it's always:
val ready = Source(???).via(flow).map(_ * 2).to(sink)

val mat: Mat = ready.run()

// the usual example:
val f: Future[String] =
  Source.single(1).map(_.toString).runWith(Sink.head)
```

Proper static typing!

Akka Streams in 20 seconds:

```
Source.single(1).map(_.toString).runWith(Sink.head)
```

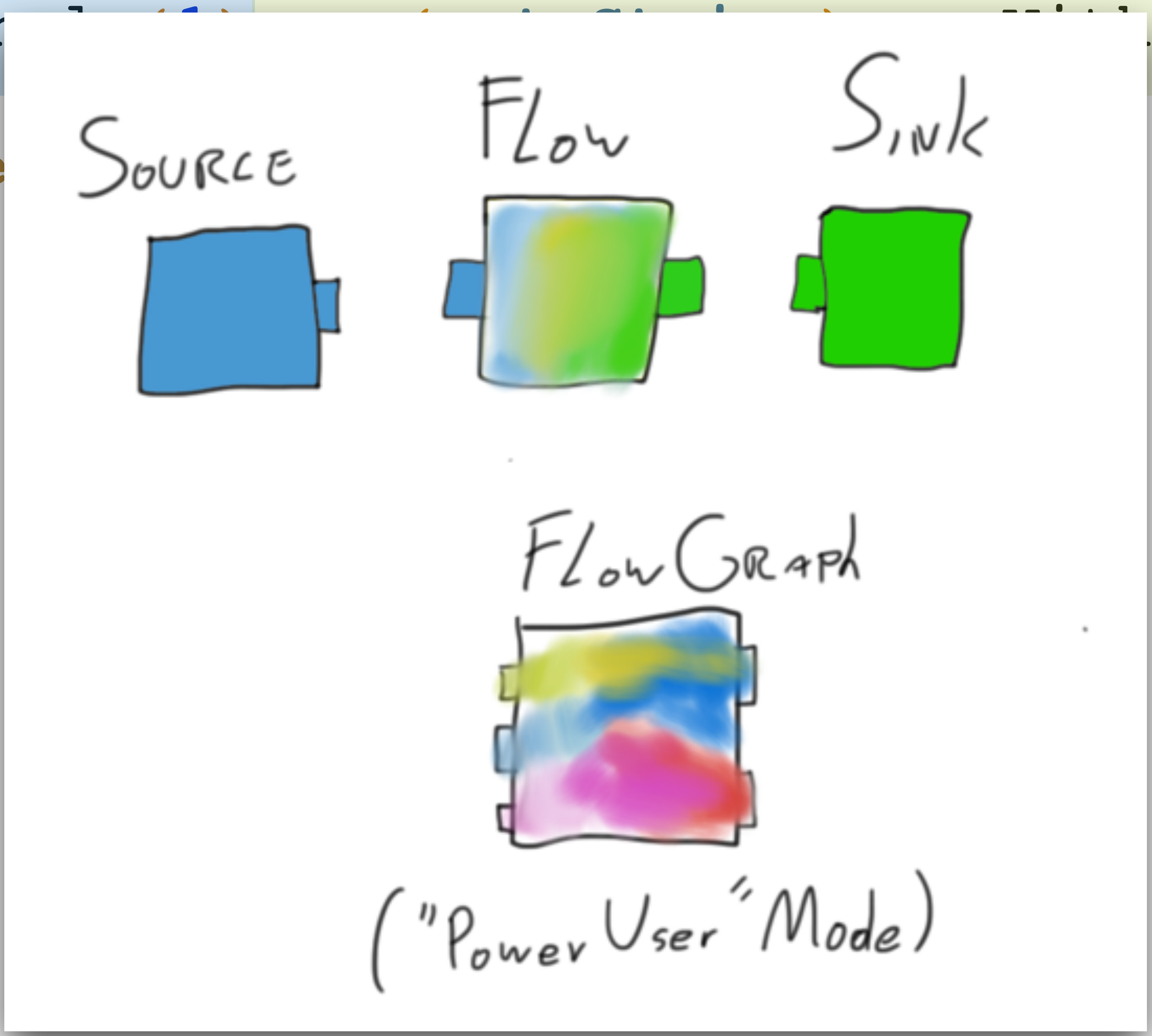
```
// types:
Source[Int, Unit]
  Flow[Int, String, Unit]
       Sink[String, Future[String]]
```



Akka Streams in 20 seconds:

```
Source.si                                              (Sink.head)
```

```
    // type
    Source[
      Flow[
```



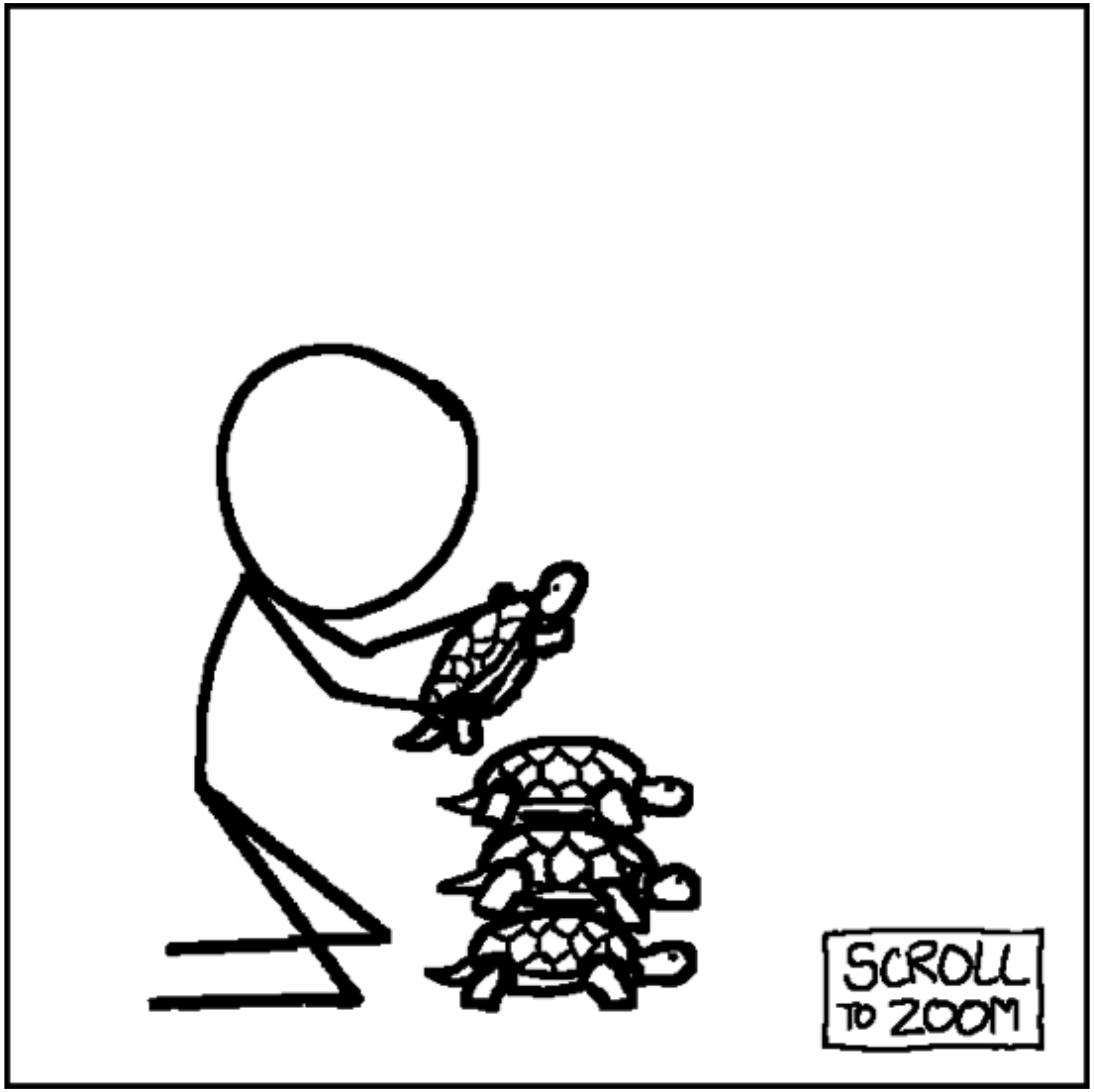
Akka HTTP

Joint effort of Spray and Akka teams.
Complete HTTP Server/Client implementation.

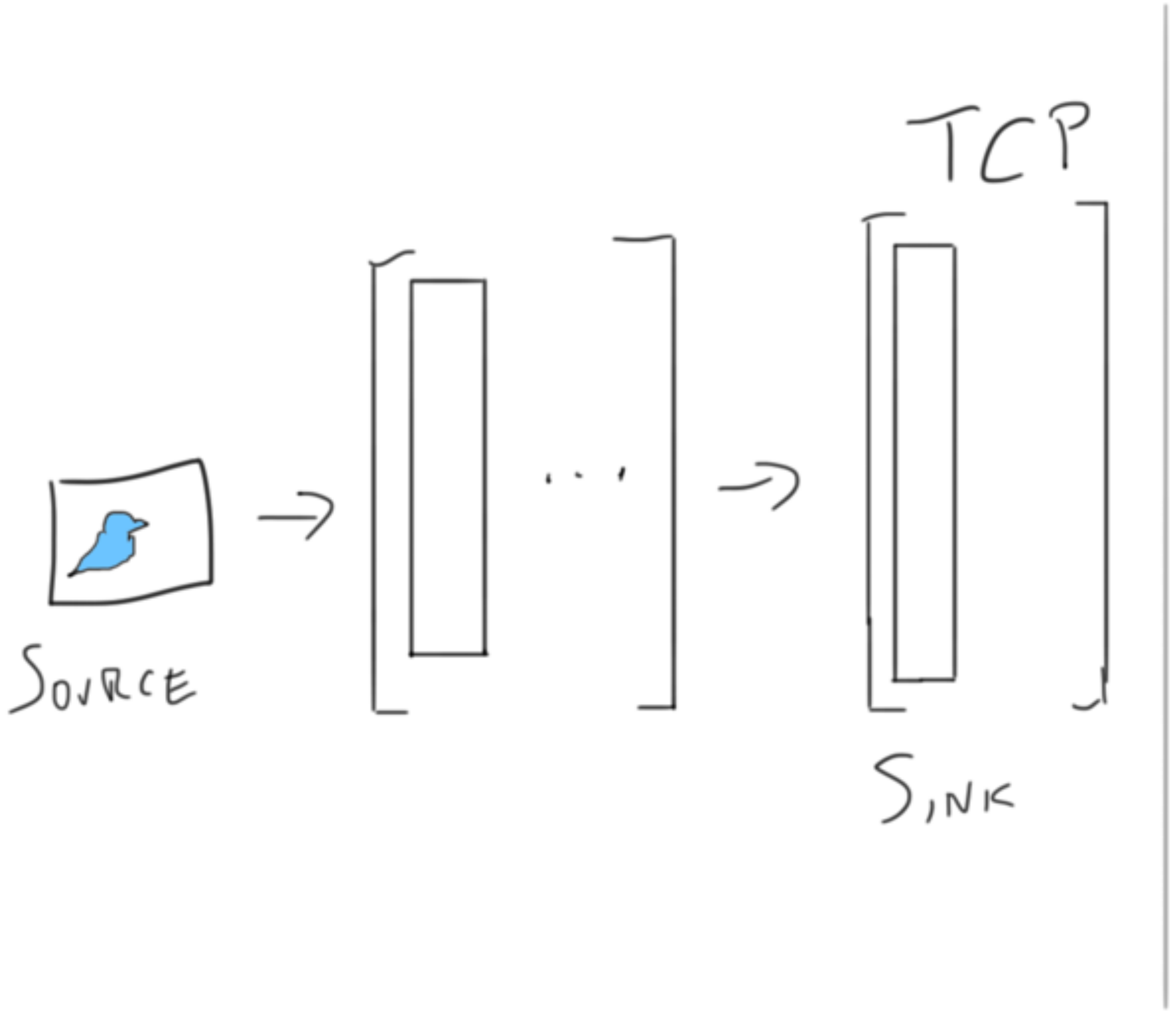
Soon prod ready, developed ~1.5 years.
Learns from Spray's 3-4 years history.

Since the beginning with
streaming as first class citizen.

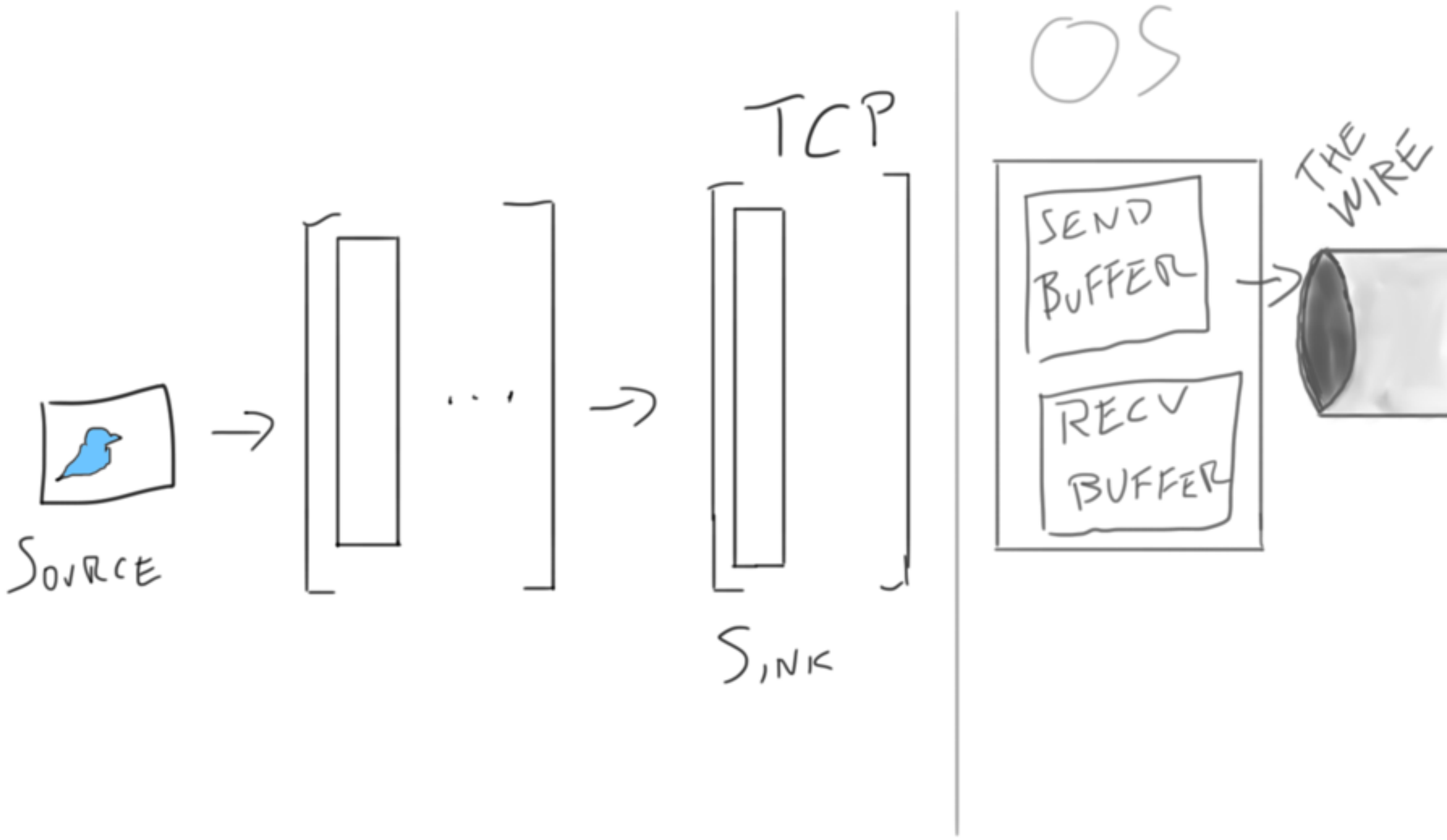
It’s ~~turtles~~ buffers all the way down!



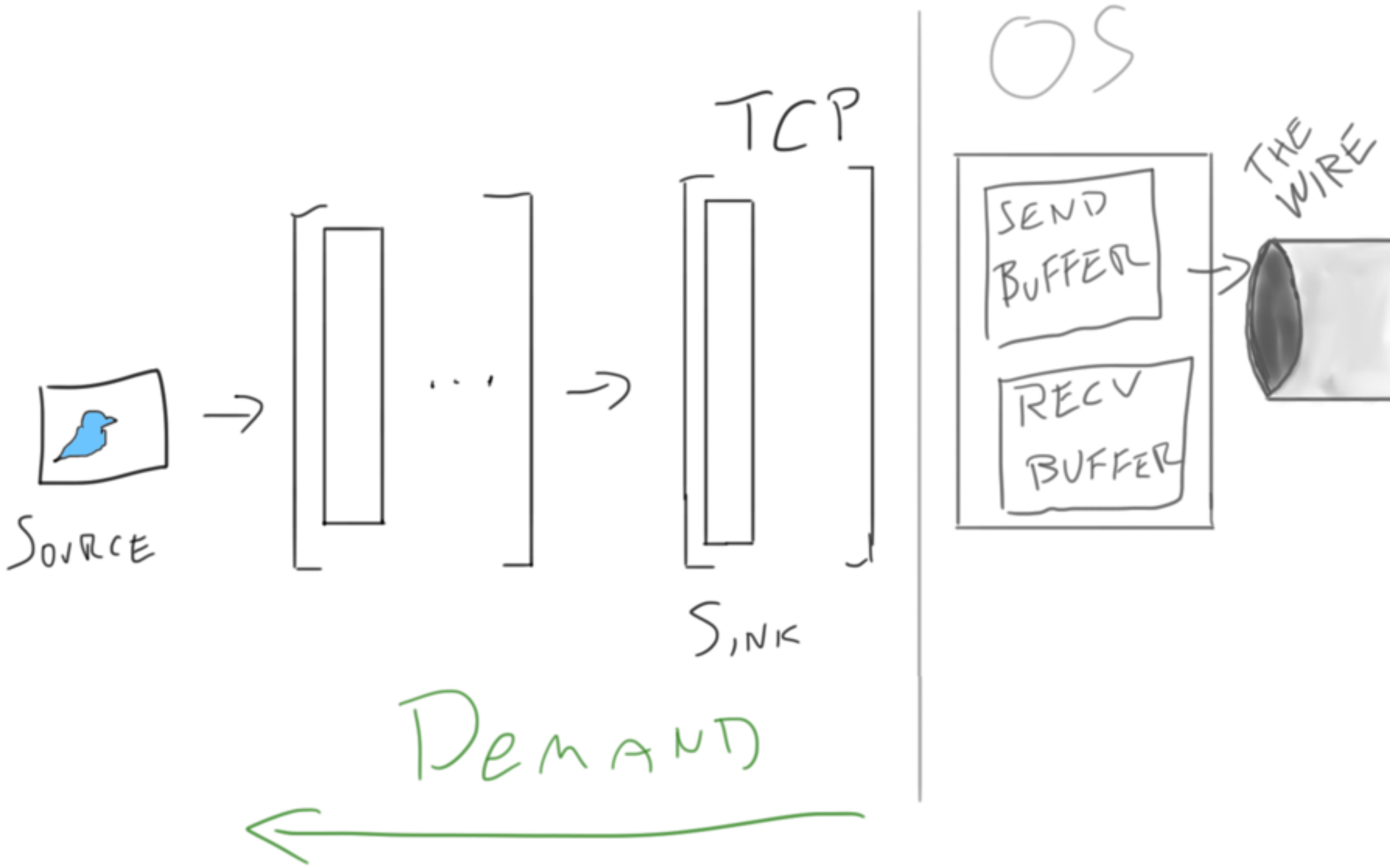
Streaming from Akka HTTP



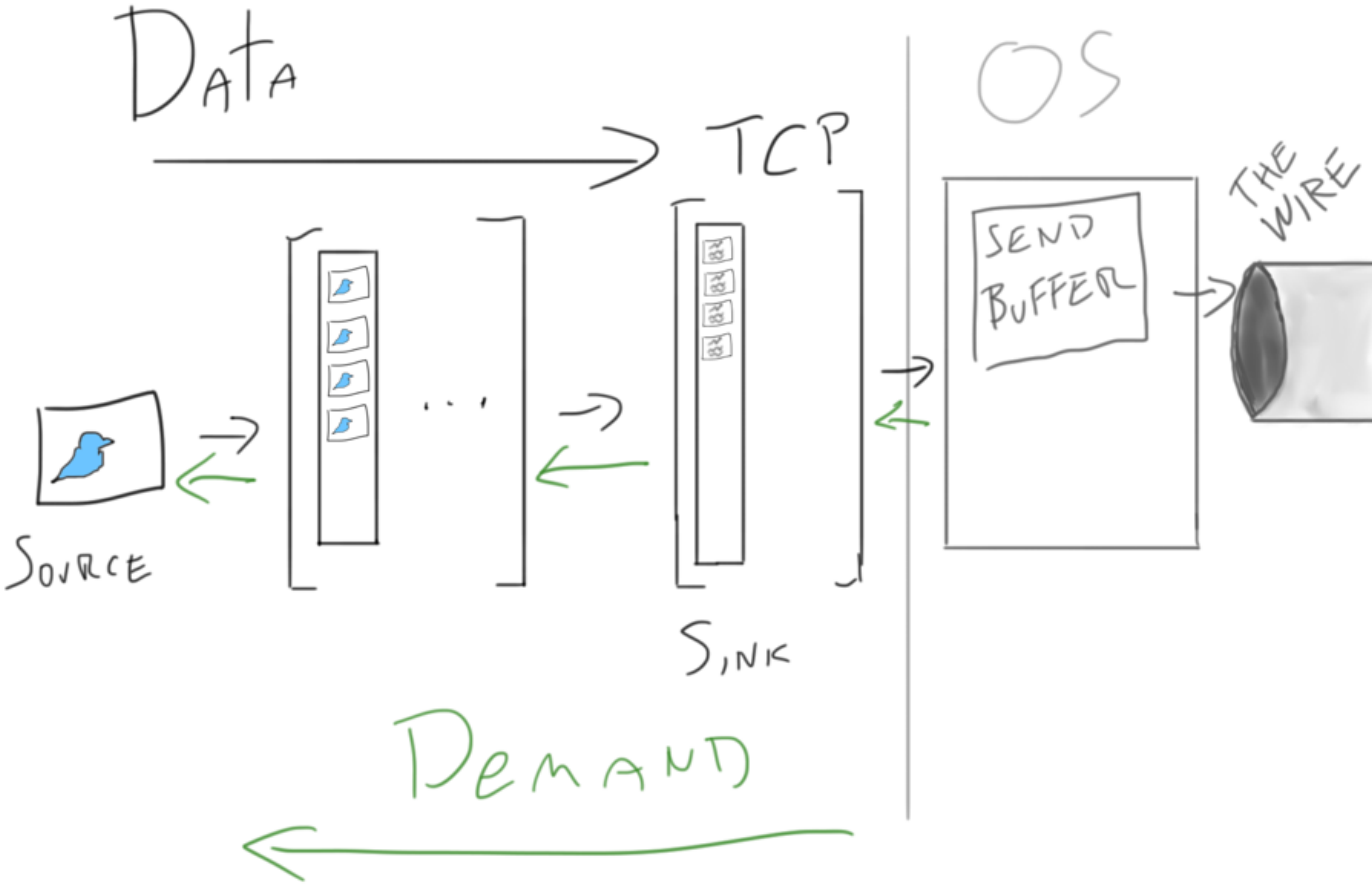
Streaming from Akka HTTP



Streaming from Akka HTTP



Streaming from Akka HTTP



Typesafe

Streaming from Akka HTTP

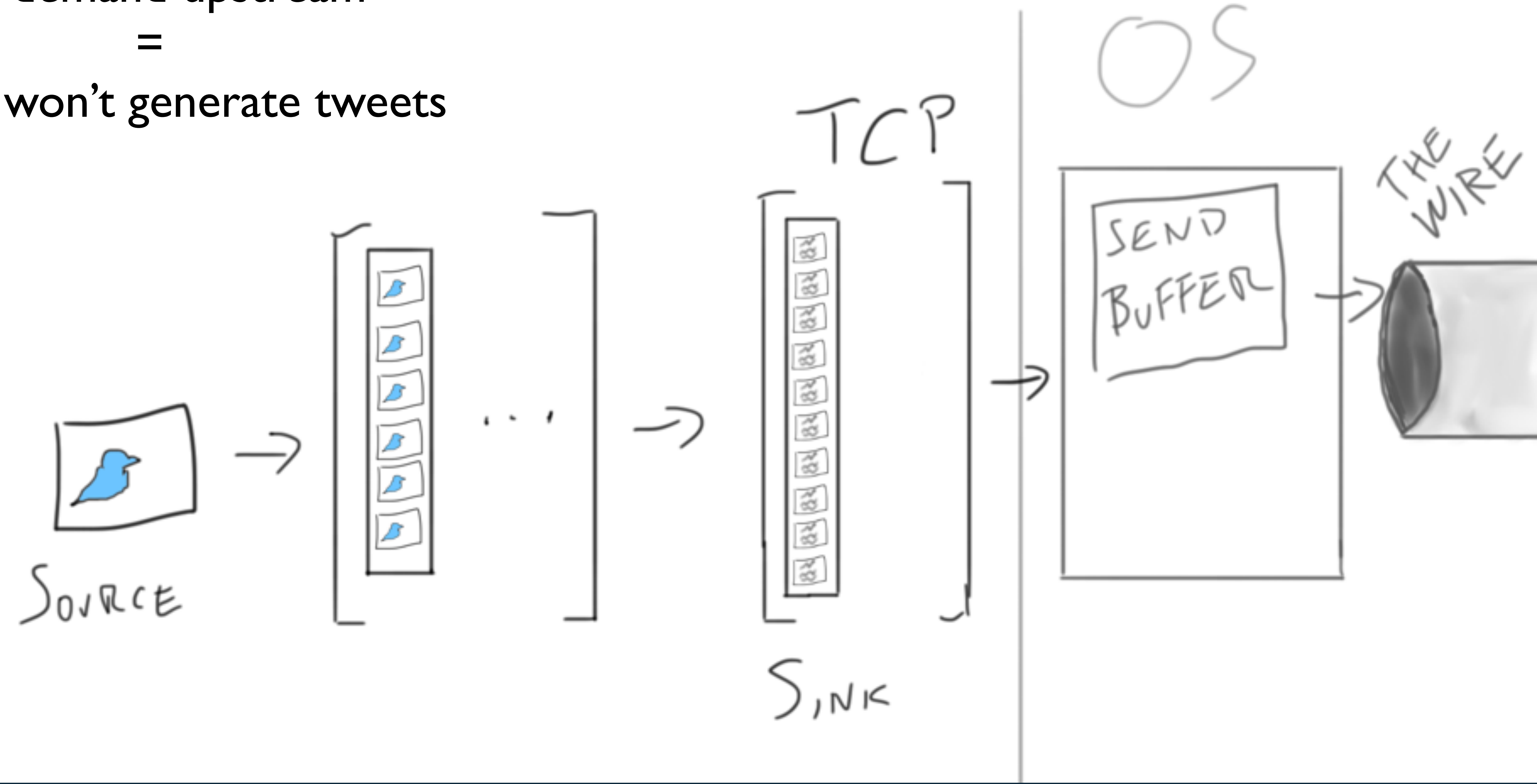
No demand from TCP

=

No demand upstream

=

Source won't generate tweets



Streaming from Akka HTTP

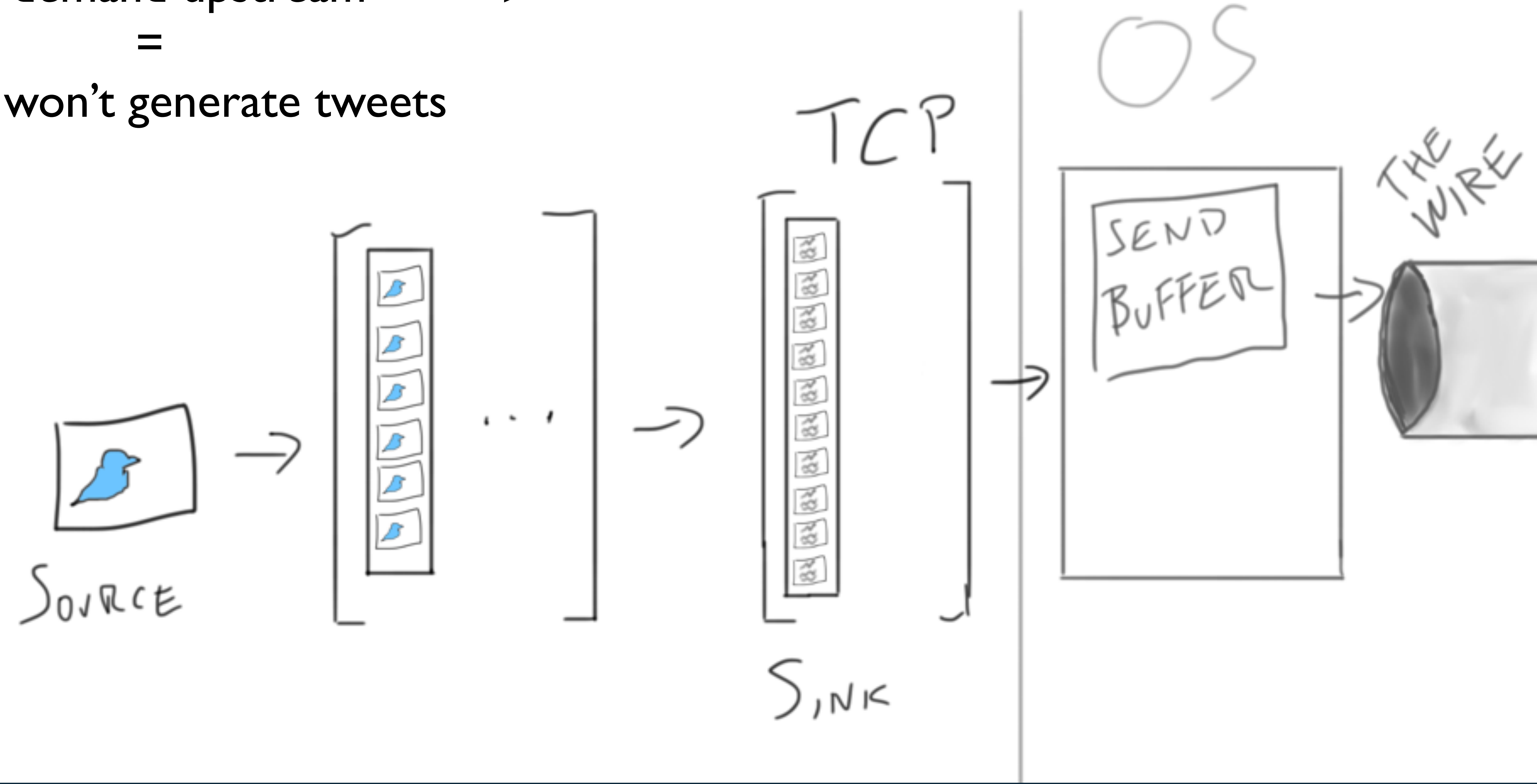
No demand from TCP

=

No demand upstream **=>**

=

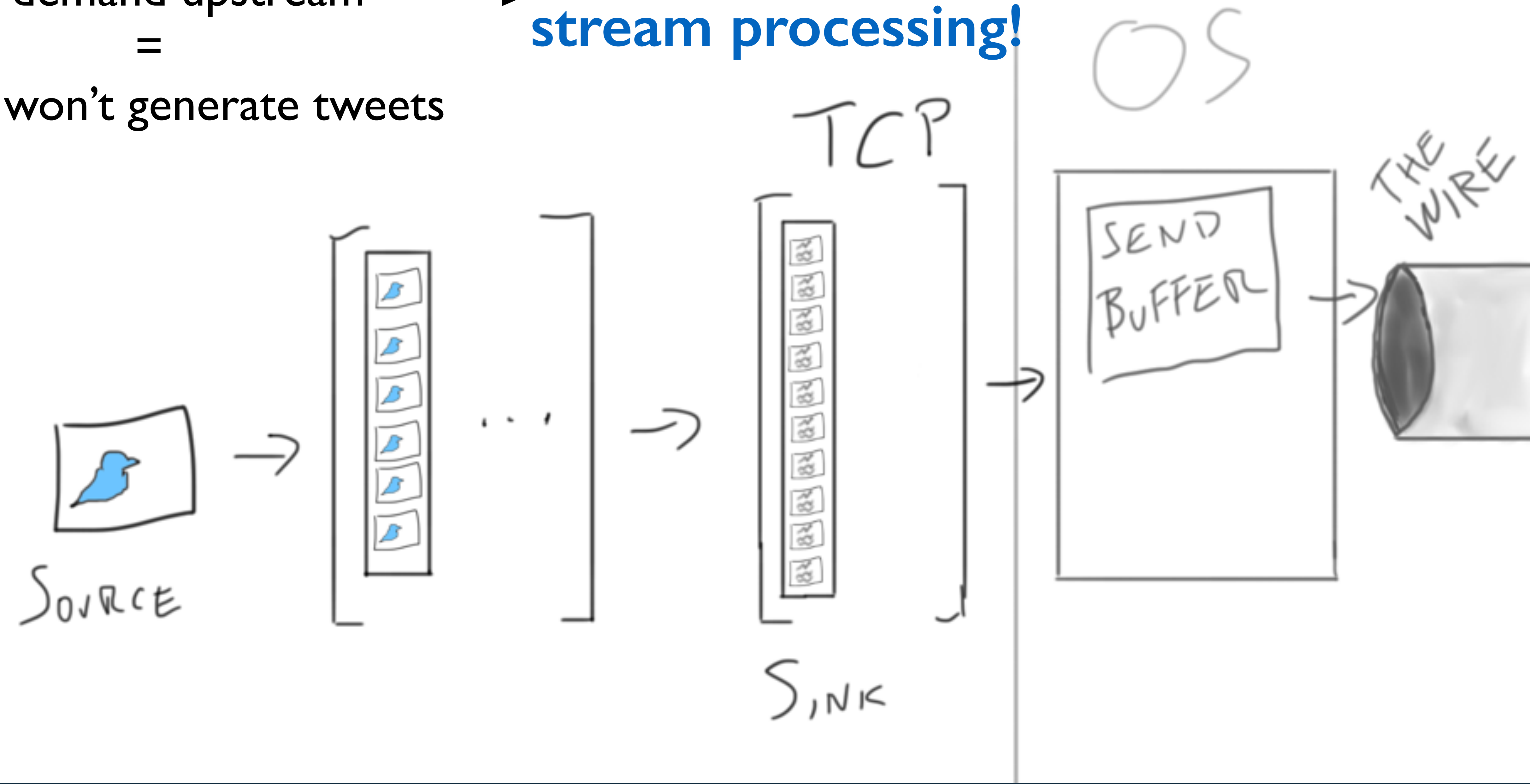
Source won't generate tweets



Streaming from Akka HTTP

No demand from TCP
=
No demand upstream
=
Source won't generate tweets

=> **Bounded memory stream processing!**



Client / Server “JSON Streaming” demo

Demo time

Akka Streams

Hidden powers:

Parallelism

&&

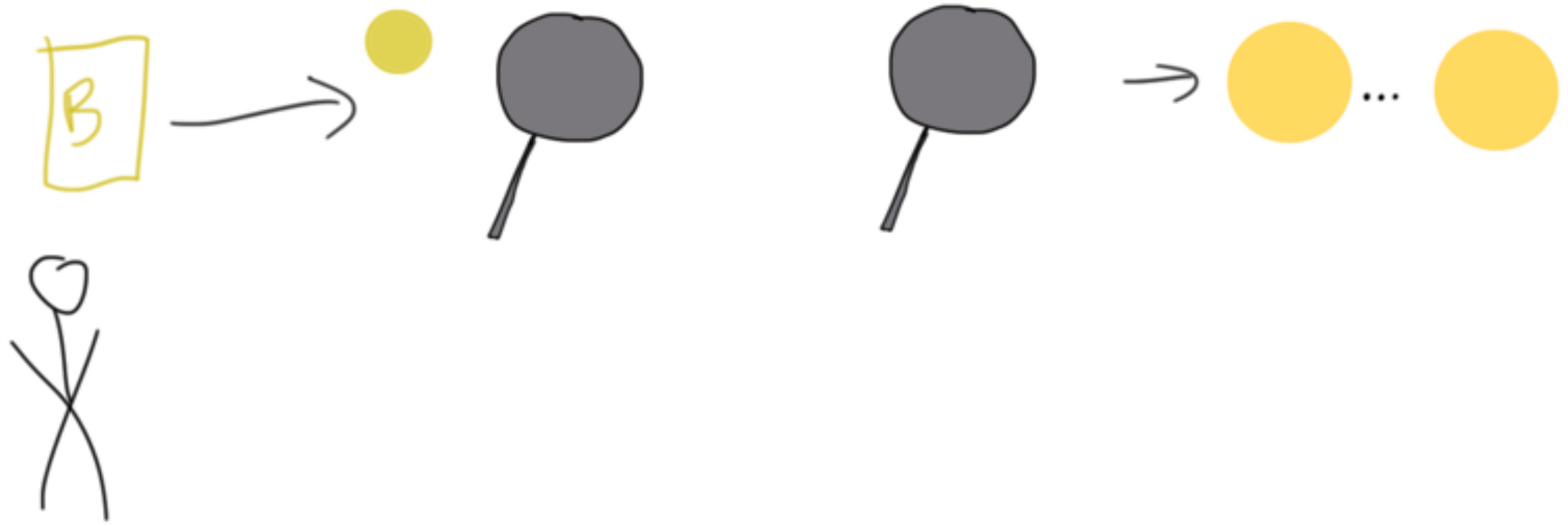
Pipelining

Pipelining Pancakes

http://doc.akka.io/docs/akka-stream-and-http-experimental/1.0/scala/stream-parallelism.html

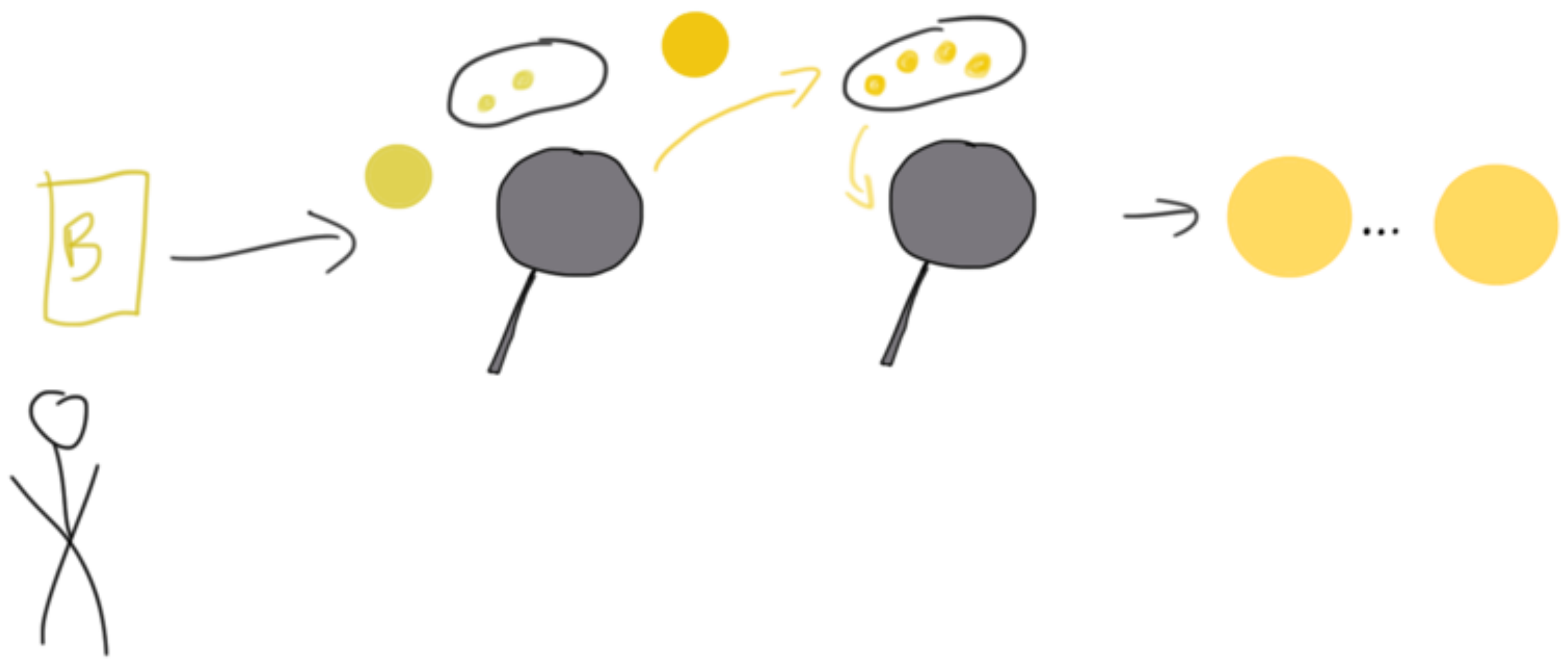
Pipelining

Flow[Scoop, Pancake,]



Pipelining

Flow[Scoop, Pancake,   ]

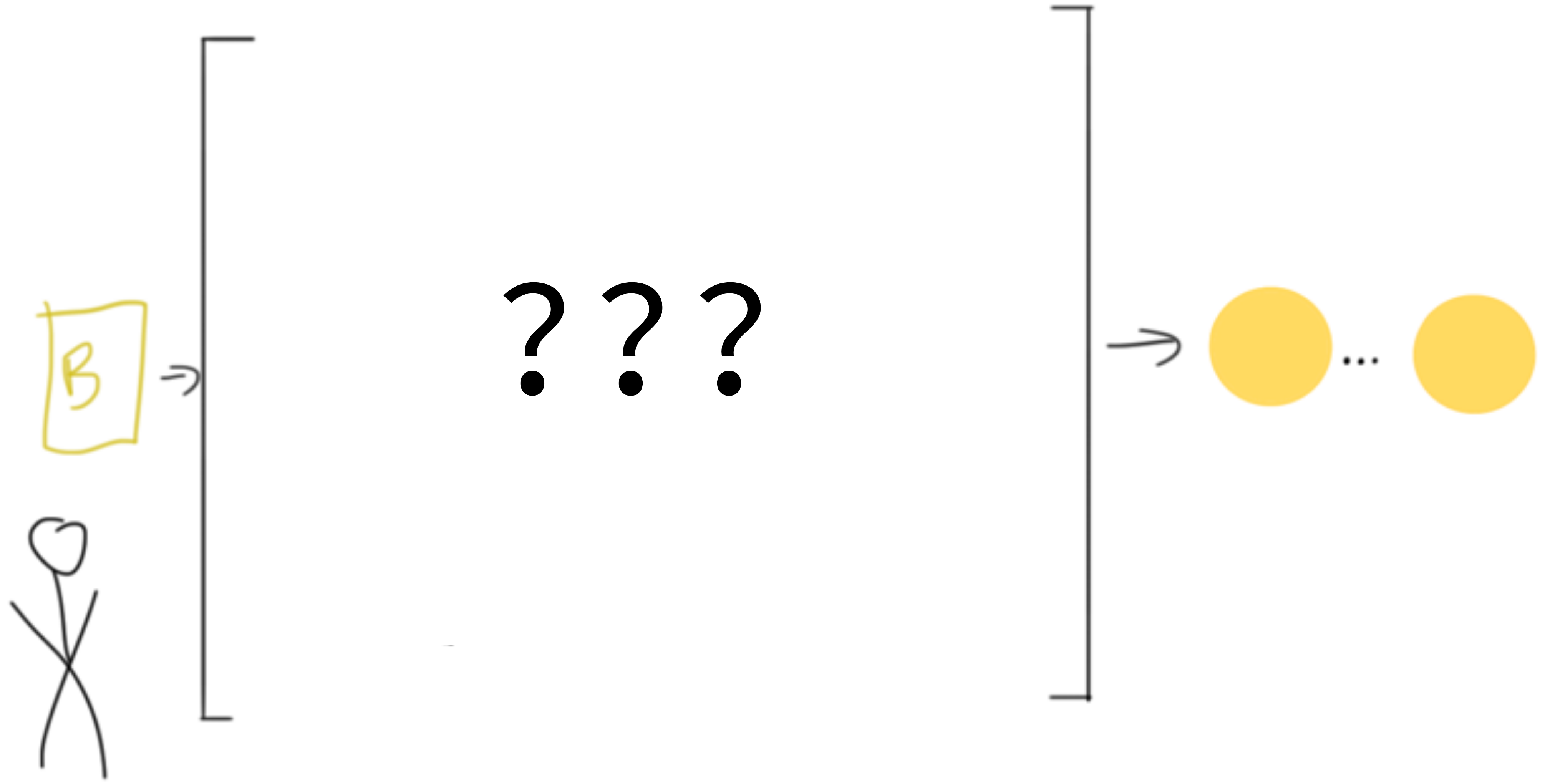


Pipelining

```
// Takes a scoop of batter and creates a pancake with one side cooked
val fryingPan1: Flow[ScoopOfBatter, HalfCookedPancake, Unit] =
  Flow[ScoopOfBatter].map { batter => HalfCookedPancake() }

// Finishes a half-cooked pancake
val fryingPan2: Flow[HalfCookedPancake, Pancake, Unit] =
  Flow[HalfCookedPancake].map { halfCooked => Pancake() }

// With the two frying pans we can fully cook pancakes
val pancakeChef: Flow[ScoopOfBatter, Pancake, Unit] =
  Flow[ScoopOfBatter].via(fryingPan1).via(fryingPan2)
```

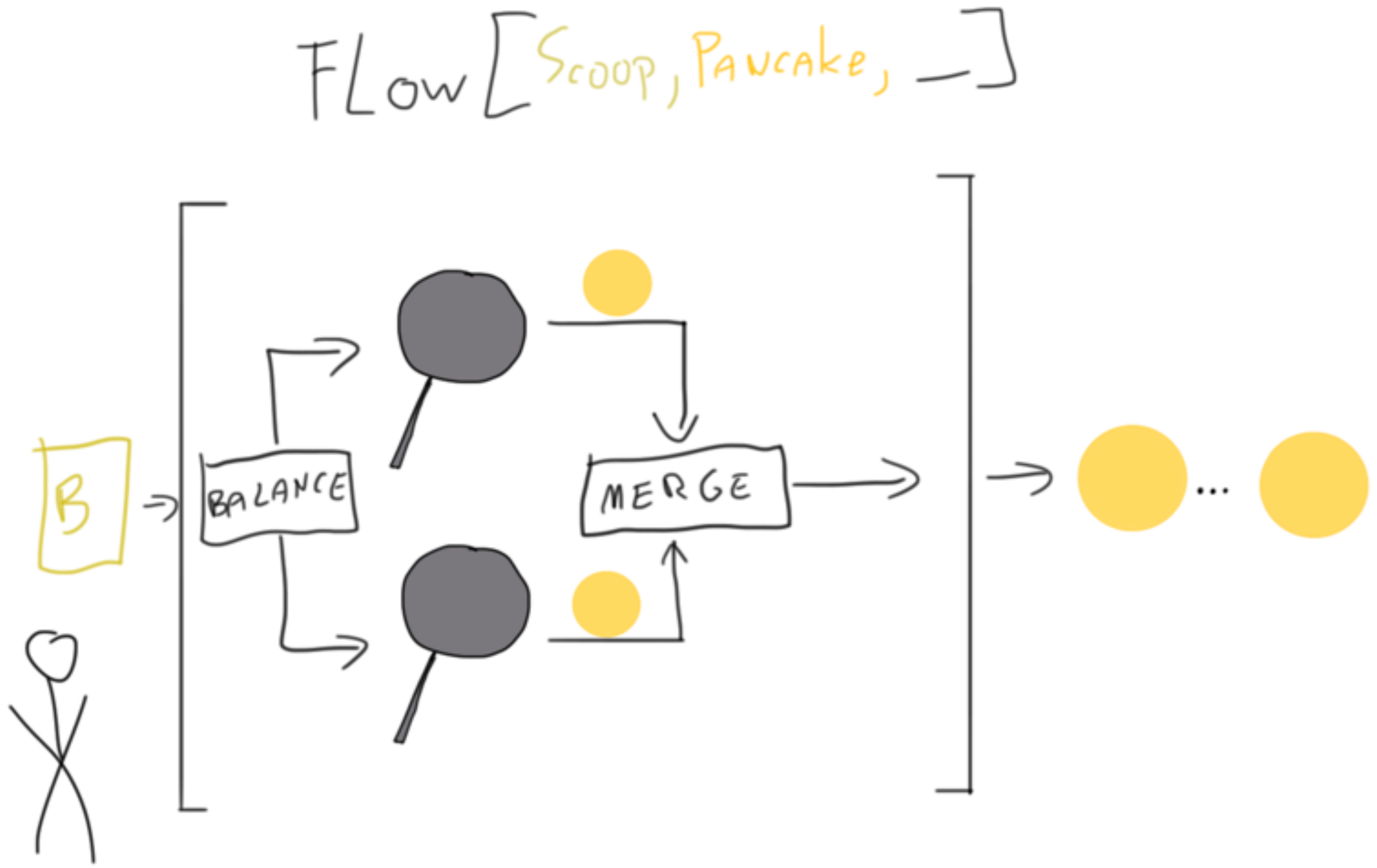
Parallelism

Flow[Scoop, Pancake, _]



Typesafe

Parallelism



Parallelism

```scala
val fryingPan: Flow[ScoopOfBatter, Pancake, Unit] =
  Flow[ScoopOfBatter].map { batter => Pancake() }

val pancakeChef: Flow[ScoopOfBatter, Pancake, Unit] = Flow() {
  implicit builder =>

  val dispatchBatter = builder.add(Balance[ScoopOfBatter](2))
  val mergePancakes = builder.add(Merge[Pancake](2))

  dispatchBatter.out(0) ~> fryingPan ~> mergePancakes.in(0)
  dispatchBatter.out(1) ~> fryingPan ~> mergePancakes.in(1)

  (dispatchBatter.in, mergePancakes.out)
}
```

Parallelism

Or simply “mapAsync”:

```
val fryingPanFun: ScoopOfBatter ⇒ Future[Pancake] =
   batter ⇒ Future.successful(Pancake())

 val pancakeChef: Flow[ScoopOfBatter, Pancake, Unit] =
   Flow[ScoopOfBatter].mapAsync(parallelism = 2)(fryingPanFun)
```

Parallelism

```scala
val fryingPan: Flow[ScoopOfBatter, Pancake, Unit] =
  Flow[ScoopOfBatter].map { batter => Pancake() }

val pancakeChef: Flow[ScoopOfBatter, Pancake, Unit] = Flow() {
  implicit builder =>

  val dispatchBatter = builder.add(Balance[ScoopOfBatter](2))
  val mergePancakes = builder.add(Merge[Pancake](2))

  dispatchBatter.out(0) ~> fryingPan ~> mergePancakes.in(0)
  dispatchBatter.out(1) ~> fryingPan ~> mergePancakes.in(1)

  (dispatchBatter.in, mergePancakes.out)
}
```

Pipelining && Parallelism

Parallelism

&&

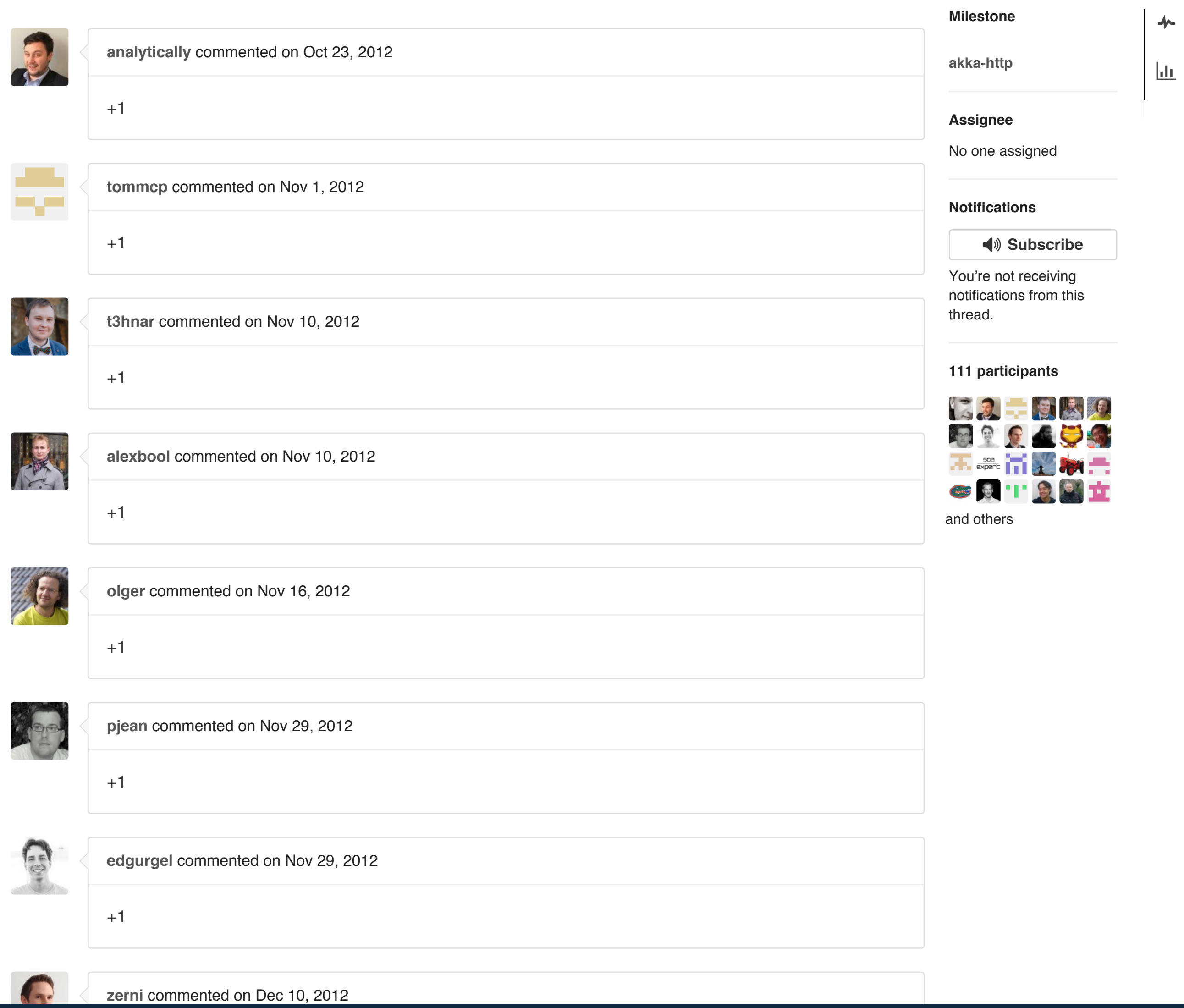
Pipelining

do the heavy-work for you.

WebSockets

A.K.A.

“Spray’s single most upvoted feature request ever”



analytically commented on Oct 23, 2012

+1

tommcp commented on Nov 1, 2012

+1

t3hnar commented on Nov 10, 2012

+1

alexbool commented on Nov 10, 2012

+1

olger commented on Nov 16, 2012

+1

pjean commented on Nov 29, 2012

+1

edgurgel commented on Nov 29, 2012

+1

zerni commented on Dec 10, 2012

Milestone

akka-http

Assignee

No one assigned

Notifications

Subscribe

You're not receiving notifications from this thread.

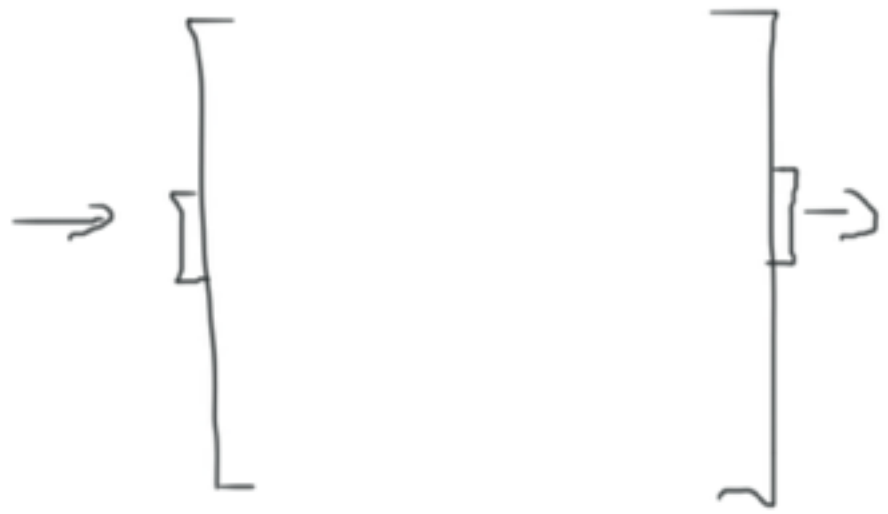
111 participants

and others

98 * “+1”

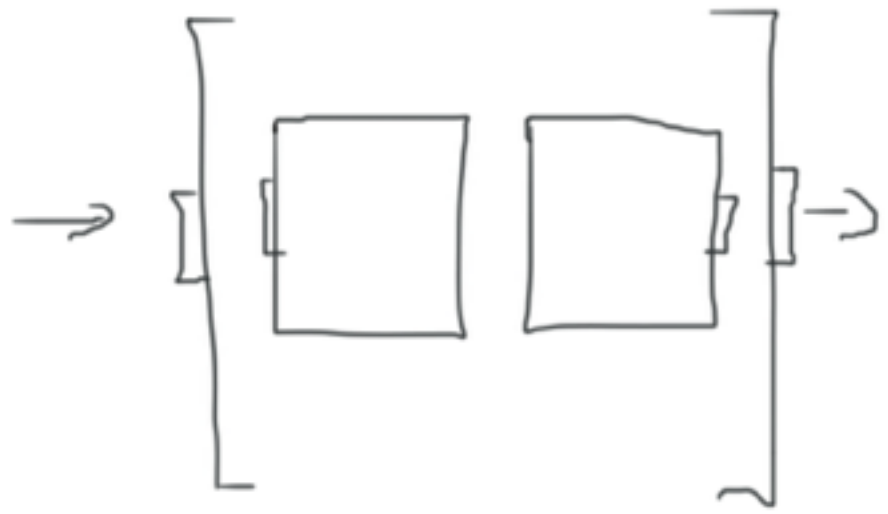
Spray's most requested feature ever: WebSockets

```
path("ws") {
  val handler: Flow[Message, Message] = ???

  handleWebsocketMessages(handler)
}
```



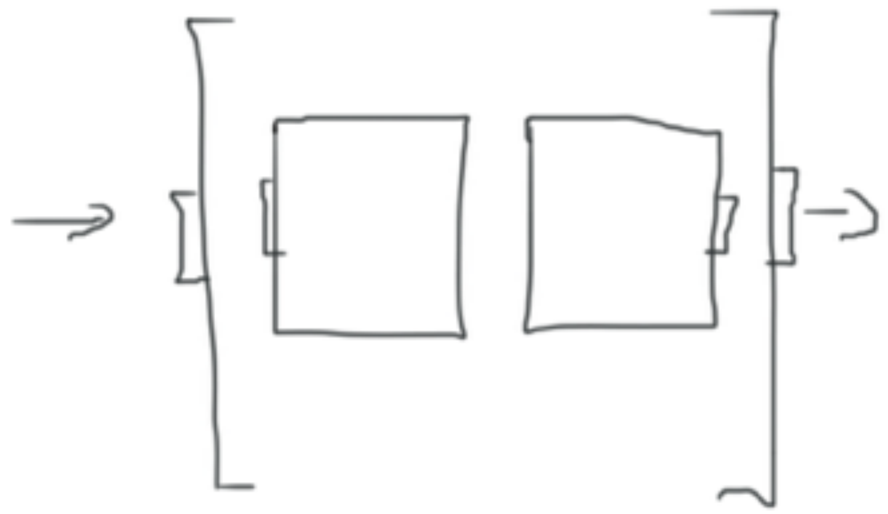
Spray's most requested feature ever: WebSockets

```
path("ws") {
  val handler: Flow[Message, Message] = ???

  handleWebsocketMessages(handler)
}
```



Spray's most requested feature ever: WebSockets

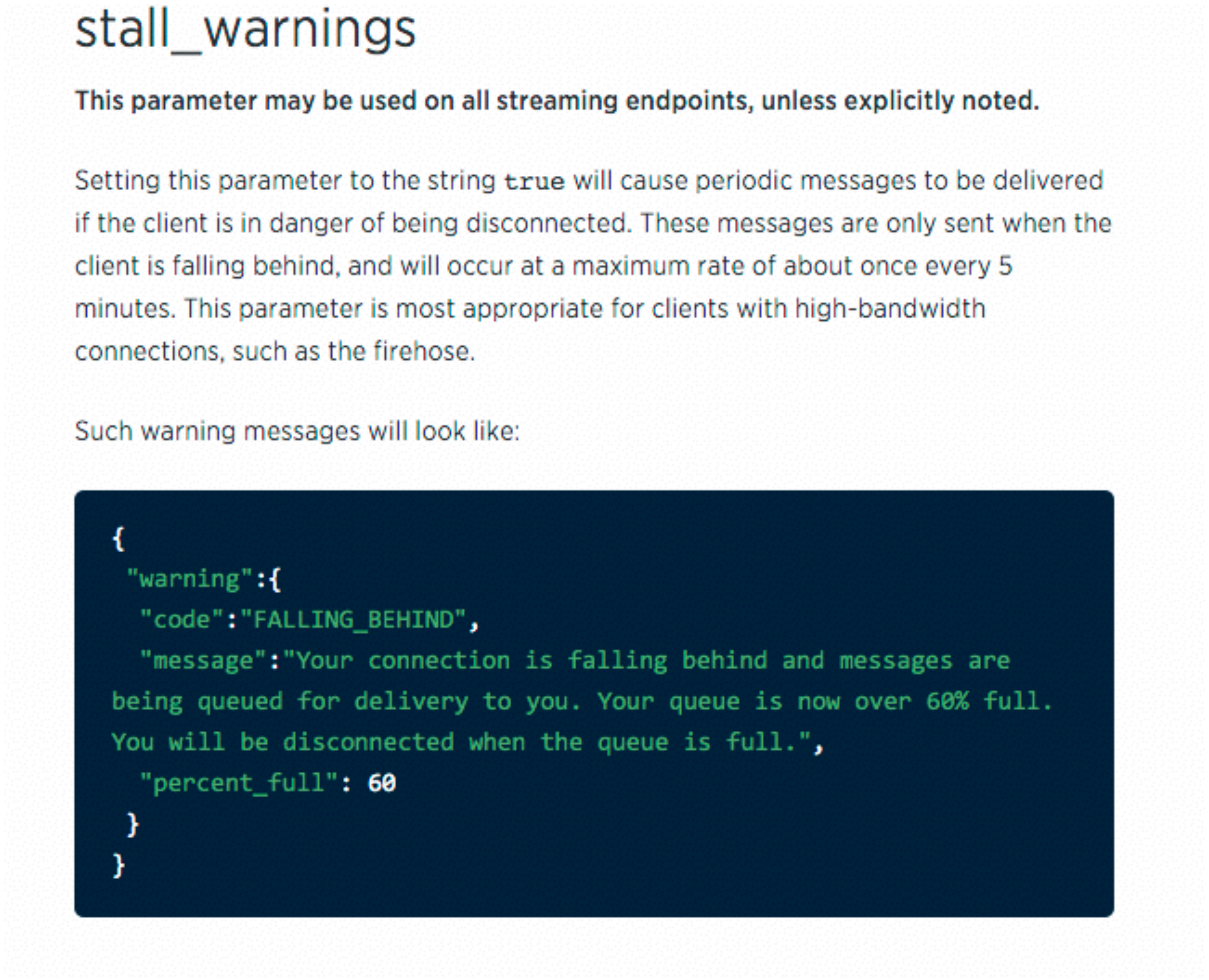
```
path("ws") {
  val handler = Flow.fromSinkAndSource(
    Sink.ignore,
    Source.single(TextMessage("Hello World!”)))

  handleWebsocketMessages(handler)
}
```



Summing up…

Summing up…

buffers, buffers everywhere!



stall_warnings

This parameter may be used on all streaming endpoints, unless explicitly noted.

Setting this parameter to the string `true` will cause periodic messages to be delivered if the client is in danger of being disconnected. These messages are only sent when the client is falling behind, and will occur at a maximum rate of about once every 5 minutes. This parameter is most appropriate for clients with high-bandwidth connections, such as the firehose.

Such warning messages will look like:

```
{
 "warning":{
  "code":"FALLING_BEHIND",
  "message":"Your connection is falling behind and messages are
being queued for delivery to you. Your queue is now over 60% full.
You will be disconnected when the queue is full.",
  "percent_full": 60
 }
}
```

https://dev.twitter.com/streaming/overview/request-parameters#stallwarnings

JEP-266 – soon…!

```
public final class Flow {
    private Flow() {} // uninstantiable

  @FunctionalInterface
    public static interface Publisher<T> {
       public void subscribe(Subscriber<? super T
    }


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


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

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

Roadmap Update: Streams & HTTP

Already pretty mature and complete implementation.
WebSockets!

Play 2.5 (2.5.M1) uses Akka Streams.
(Scala || Java) DSL == same power.

Last phases of polishing up APIs and features.
1.1 release in coming weeks.

After 1.1, merging with Akka 2.4 (experimental module).

Akka 2.4 requires JDK8.
(that's about time to do so!)

Roadmap Update: Akka

- Reactive Platform
  - Remoting / Cluster: **Docker networking support**
  - Cluster: **Split Brain Resolver** (beta)
  - Akka Persistence: **Cross-Scala-version snapshot deserializer**
  - Java 6: **Extended LTS**

- Akka 2.4.0 (released this month, binary compatible with 2.3)
  - Cluster Tools *promoted to stable!*
  - Persistence *promoted to stable!*
  - Persistence Queries (experimental)
  - Akka Typed (experimental)
  - Distributed Data (experimental)
  - Akka Streams (currently 1.0, will be included in 2.4.x eventually)

Links

- **The projects:**
  - **akka.io**
  - **typesafe.com/products/typesafe-reactive-platform**
  - reactive-streams.org

- Viktor Klang's interview with all RS founding members
- Akka HTTP in depth with Mathias and Johannes @ Scala.World

- **Akka User** - mailing list:
  - https://groups.google.com/group/akka-user
- **Community chat:**
  - http://gitter.im/akka/akka

akka

SAMSUNG

Fidelity INVESTMENTS

Scotiabank

HSBC

IBM

BBC

intel

verizon

ebay

THE HUFFINGTON POST

NBC

Time Warner Cable

comcast

workday.

Walmart

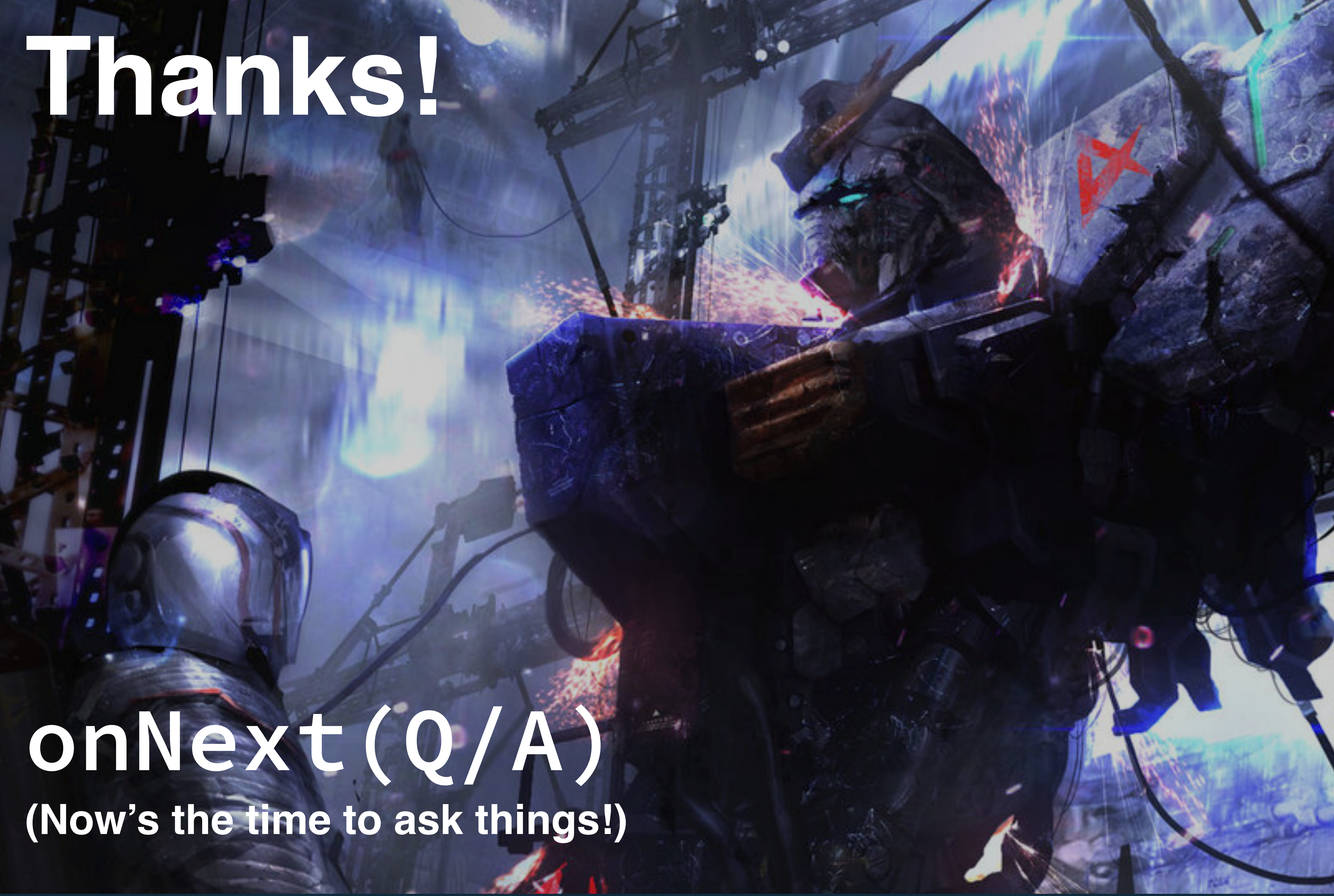
dropcam

CISCO

amazon web services

hp

Typesafe



Thanks!

onNext(Q/A)

(Now's the time to ask things!)

Typesafe

ktoso @ typesafe.com
twitter: ktosopl

github: ktoso
team blog: letitcrash.com

home: akka.io

Typesafe