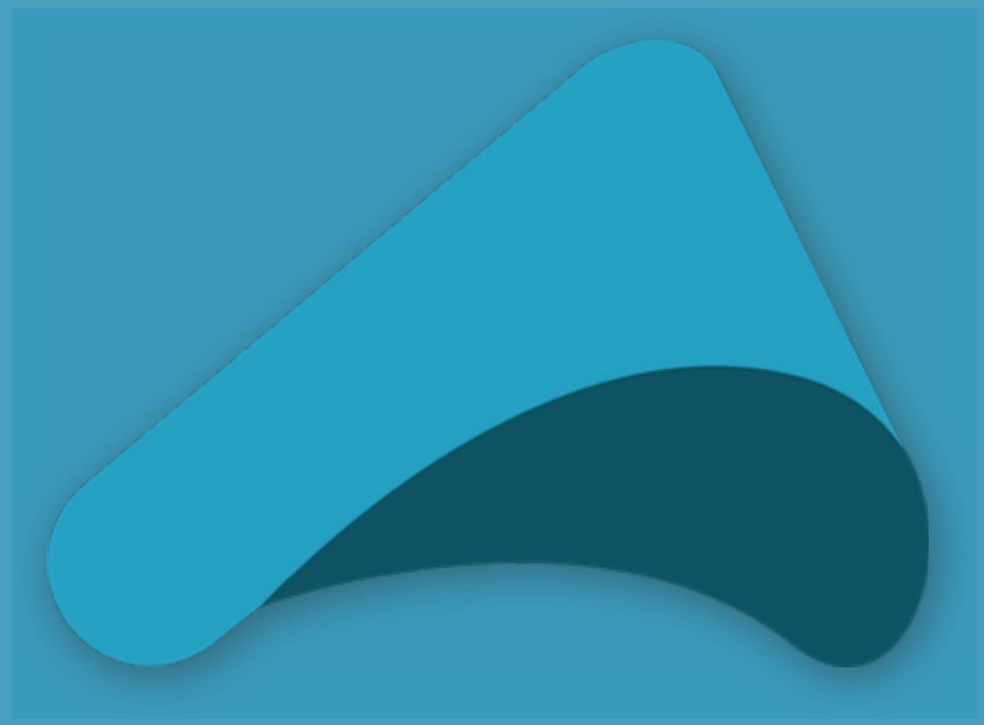
Reactive Streams principles applied in Akka Streams



Eric Loots

Consultant - Trainer @ Lightbend

Botanical Garden - Merano, Italy

Lightbend

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

Agenda:

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

Reactive Streams

What is a *Stream* anyway?

An abundant supply of definitions is available …

So, what's 'our' definition?

A Stream is a controlled flow of Data

Let data flow at just the right rate

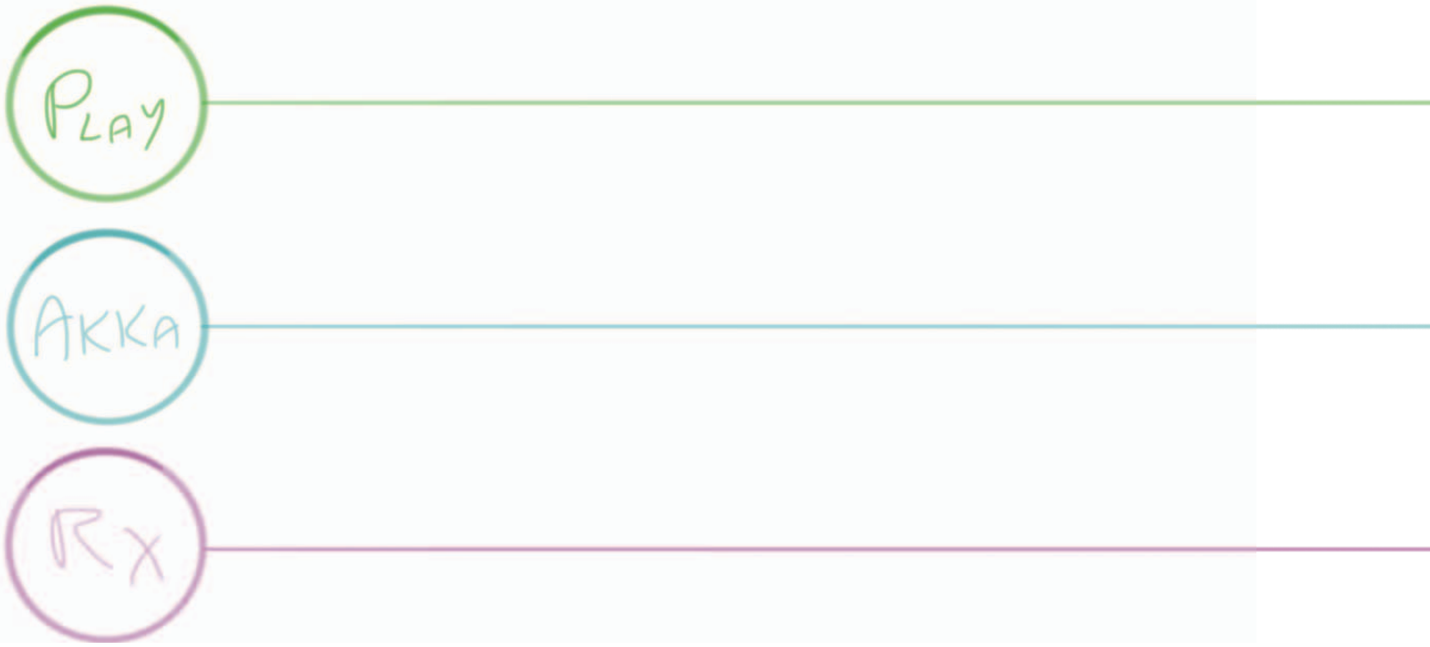
Reactive Streams - story: early FRP

- .NETs' Reactive Extensions (2009)

.NET 3.5

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



~2013:

Reactive Programming
becoming widely adopted on JVM.

- **Play** introduced "**Iteratees**"
- **Akka** (2009) had **Akka-IO** (TCP etc)
- **Ben C.** 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

```
// an iteratee that consumes chunks 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]
```

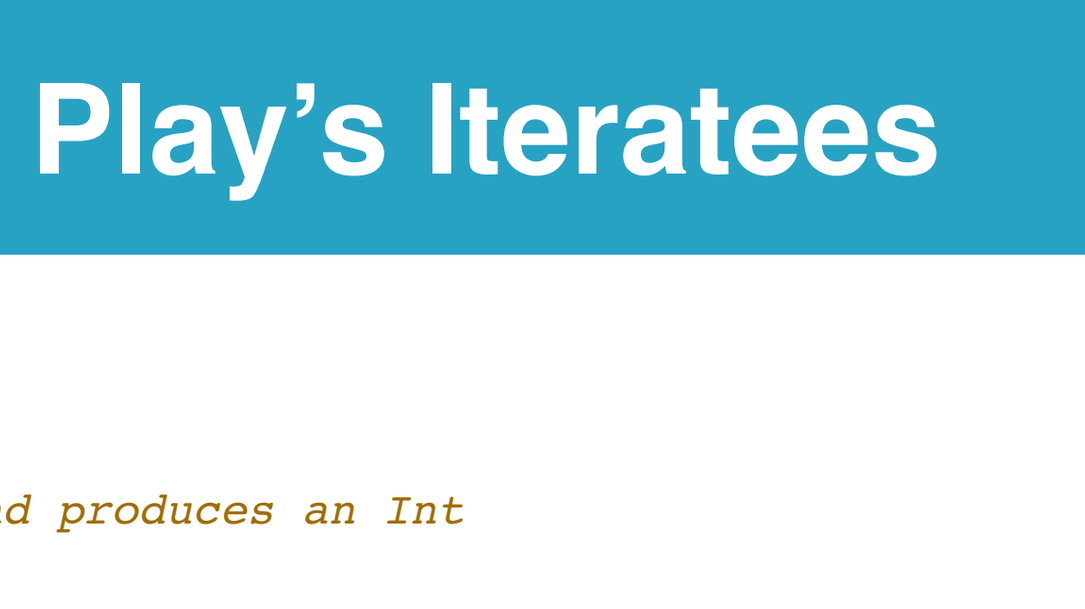
https://www.playframework.com/documentation/2.0/Iteratees

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 nne,
while recording **"F** **era Course**.

Viktor Klang (A **a)**
and Marius Erik

The term "**reacti** **re**" gets coined.

Goals:

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

Also (not shown 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 Konrad Malawski join the effort soon after.

Reactive Streams - expert group founded

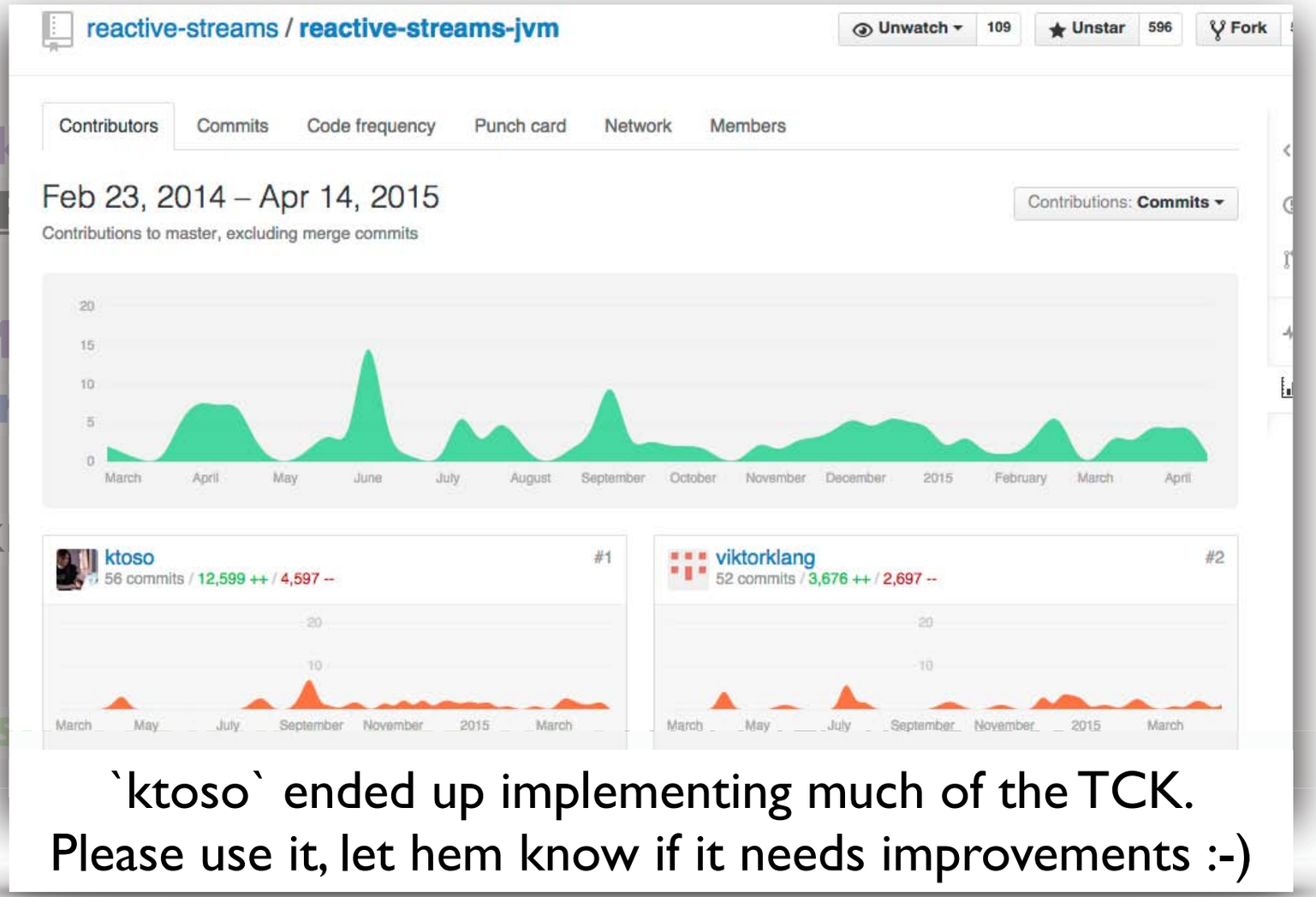
October 2013

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

Viktor Klang (Akka), Erik M
and Marius Eriksen (Twitter

The term "**reactive non-block**

December 2013
Stephane Maldini & Jon Bris



`ktoso` ended up implementing much of the TCK.
Please use it, let hem 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 Konrad M. join the effort soon after.

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

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"**

2017
Reactive Streams included with **JDK9 !**

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

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


    public static interface Subscriber<T> {
      public void onSubscribe(Subscription subscription);
      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> {
    }
}
```

Reactive Streams / Rules + TCK

ID	Rule
1	The total number of `onNext` signals sent 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.
2	A `Publisher` MAY signal less `onNext` than requested and terminate the `Subscription` by calling `onComplete` or `onError`.
3	`onSubscribe`, `onNext`, `onError` and `onComplete` signaled to a `Subscriber` MUST be signaled sequentially (no concurrent notifications).
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 [1] the only legal way to signal failure (or reject the `Subscriber`) is by calling `onError` (after calling `onSubscribe`).
10	`Publisher.subscribe` MAY be called as many times as wanted but MUST be with a different `Subscriber` each time [see 2.12].
11	A `Publisher` MAY support multiple `Subscriber`s and decides whether each `Subscription` is unicast or multicast.

[1] : A stateful Publisher can be overwhelmed, bounded by a f
number of underlying resources, exhausted, shut-down or in a
state.

RS Library A

ID	Rule
1	A `Subscriber` MUST signal demand via `Subscription.request(long n)` to receive `onNext` signals.
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.
3	`Subscriber.onComplete()` and `Subscriber.onError(Throwable t)` MUST NOT call any methods on the `Subscription` or the `Publisher`.
4	`Subscriber.onComplete()` and `Subscriber.onError(Throwable t)` MUST consider the Subscription cancelled after having received the signal.
5	A `Subscriber` MUST call `Subscription.cancel()` on the given `Subscription` after an `onSubscribe` signal if it already has an active `Subscription`.
6	A `Subscriber` MUST call `Subscription.cancel()` if it l longer valid to the `Publisher` without the `Publisher` hav signaled `onError` or `onComplete`.
7	A `Subscriber` MUST ensure that all calls on its `Subscription` take place from the same thread or provide respective external synchronization.
8	A `Subscriber` MUST be prepared to receive one or more `onNext` signals after having called `Subscription.cancel` there are still requested elements pending [see 3.12]. `Subscription.cancel()` does not guarantee to perform t underlying cleaning operations immediately.
9	A `Subscriber` MUST be prepared to receive an `onComplete` signal with or without a preceding `Subscription.request(long n)` call.
10	A `Subscriber` MUST be prepared to receive an `onError` signal with or without a preceding `Subscription.request(long n)` call.
11	A `Subscriber` MUST make sure that all calls on its `onXXX` methods happen-before [1] the processing of the respective signals. I.e. the Subscriber must take care of properly publishing the signal to its processing logic.
12	`Subscriber.onSubscribe` MUST be called at most once for a given `Subscriber` (based on object equality).
	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 only legal way for a `Subscriber` to signal failure

ID	Rule
1	`Subscription.request` and `Subscription.cancel` MUST only be called inside of its `Subscriber` context. A `Subscription` represents the unique relationship between a `Subscriber` and a `Publisher` [see 2.12].
2	The `Subscription` MUST allow the `Subscriber` to call `Subscription.request` synchronously from within `onNext` or `onSubscribe`.
3	`Subscription.request` MUST place an upper bound on possible synchronous recursion between `Publisher` and `Subscriber` [1].
	NOPs.
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.
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.
10	While the `Subscription` is not cancelled, `Subscription.request(long n)` MAY synchronously call `onNext` on this (or other) subscriber(s).

RS library B

Make building powerful concurrent & distributed applications **simple**.

Reactive Streams: goals

1. Avoid Unbounded buffering across Asynchronous Boundaries
2. Inter-op Interfaces between different libraries

Don’t use RS directly!

Use a Library such as Akka Streams

Agenda:

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

Make building powerful concurrent & distributed applications **simple**.

Akka is a toolkit and runtime for building highly concurrent, distributed, and resilient **message-driven** applications on the JVM

Actors – simple & high performance concurrency
Cluster / **Remoting** – location transparency, resilience
Cluster Sharding – and more prepackaged patterns

Streams – back-pressured stream processing

Persistence – Event Sourcing

HTTP – complete, fully async and reactive HTTP Server
Official **Kafka**, **Cassandra**, **DynamoDB integrations**, tons more in the community

Complete **Java** & **Scala APIs** for all features (since day 1)
Akka Typed coming soon…

The Actor Model

”The **actor model** in computer science is a mathematical model of concurrent computation that treats *actors* as the universal primitives of concurrent computation. ”

Wikipedia

An Actor

receives **messages**

and acts on them by:

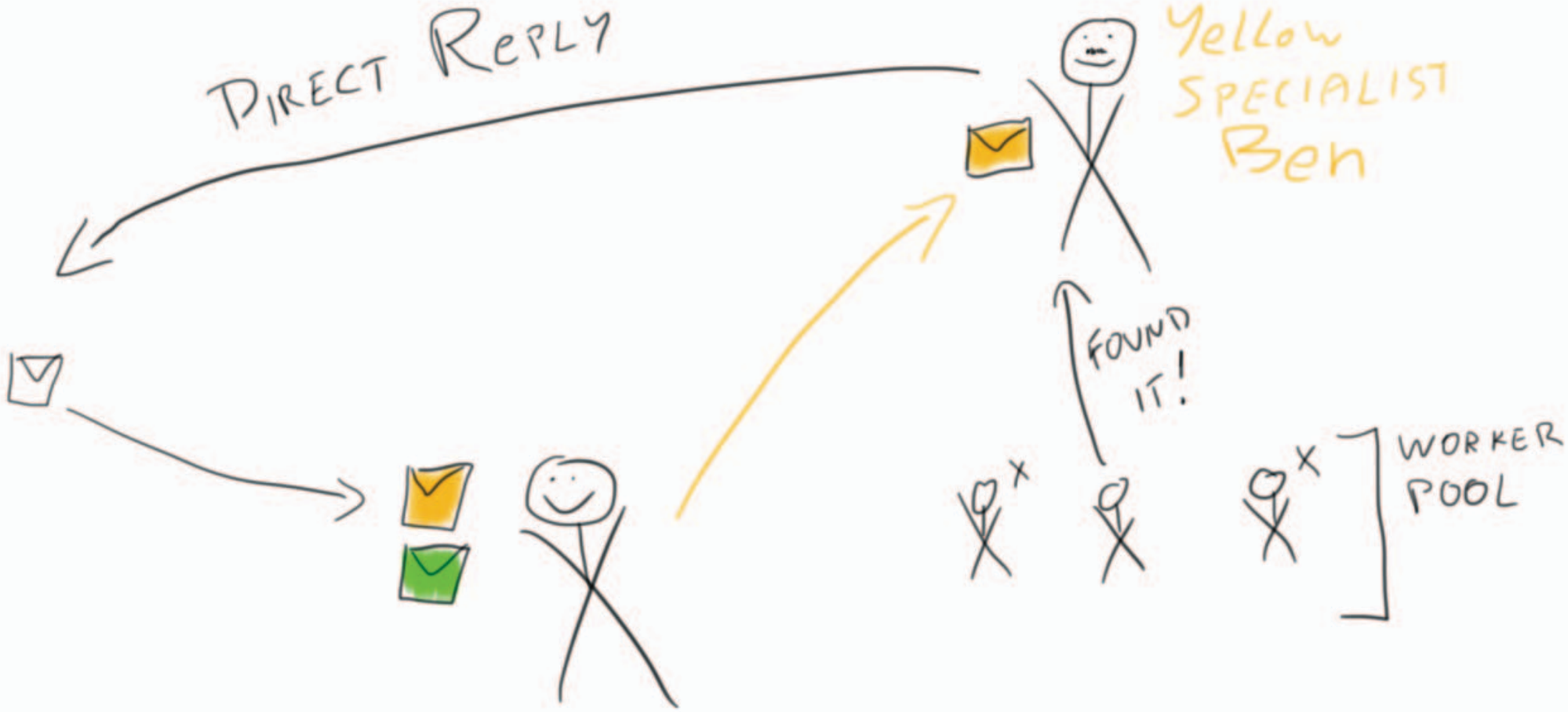
- **Sending messages**
- **Changing its state / behaviour**
- **Creating more actors**

An Actor

A concurrency and distribution construct.

an *addressable, location-transparent, entity*

One actor is no Actor

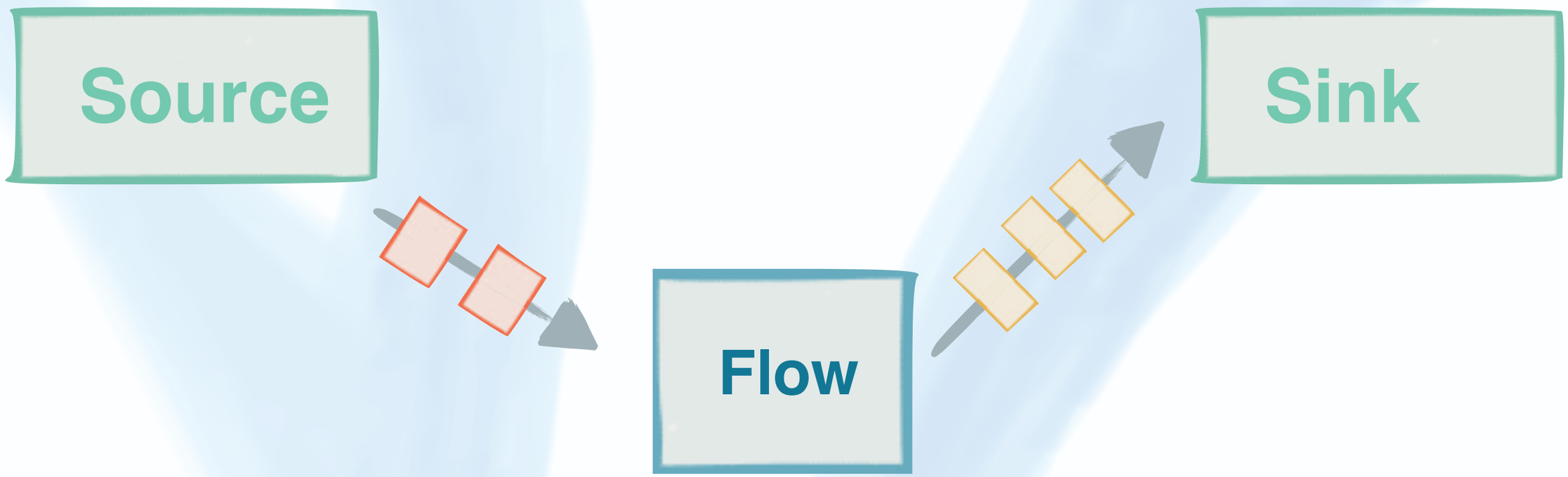


Agenda:

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

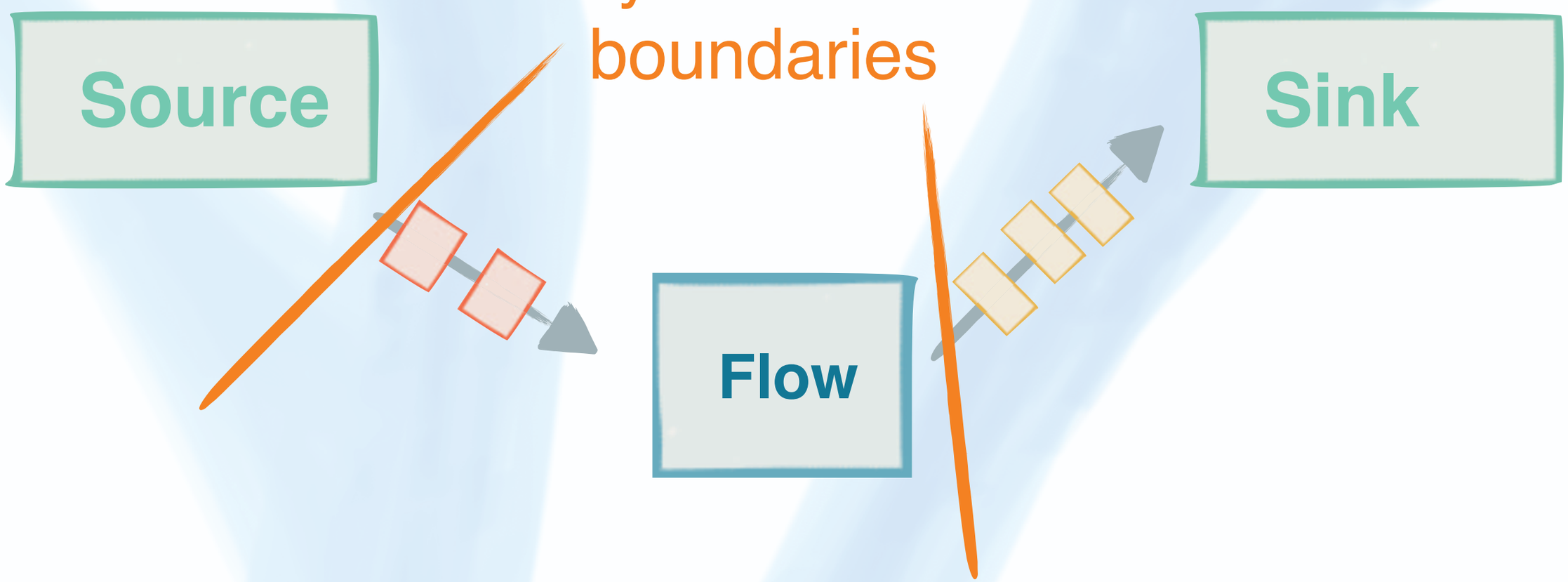
akka streams

Asynchronous back pressured **stream processing**



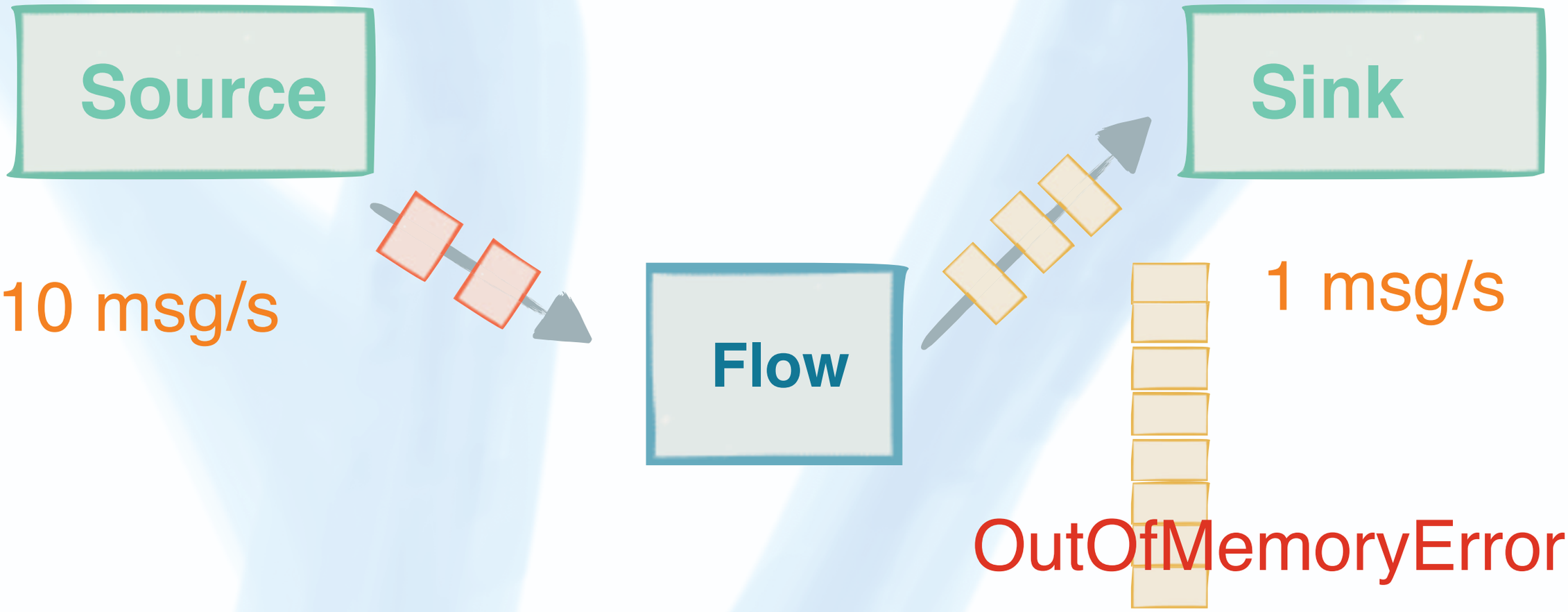
akka streams

Asynchronous back pressured stream processing



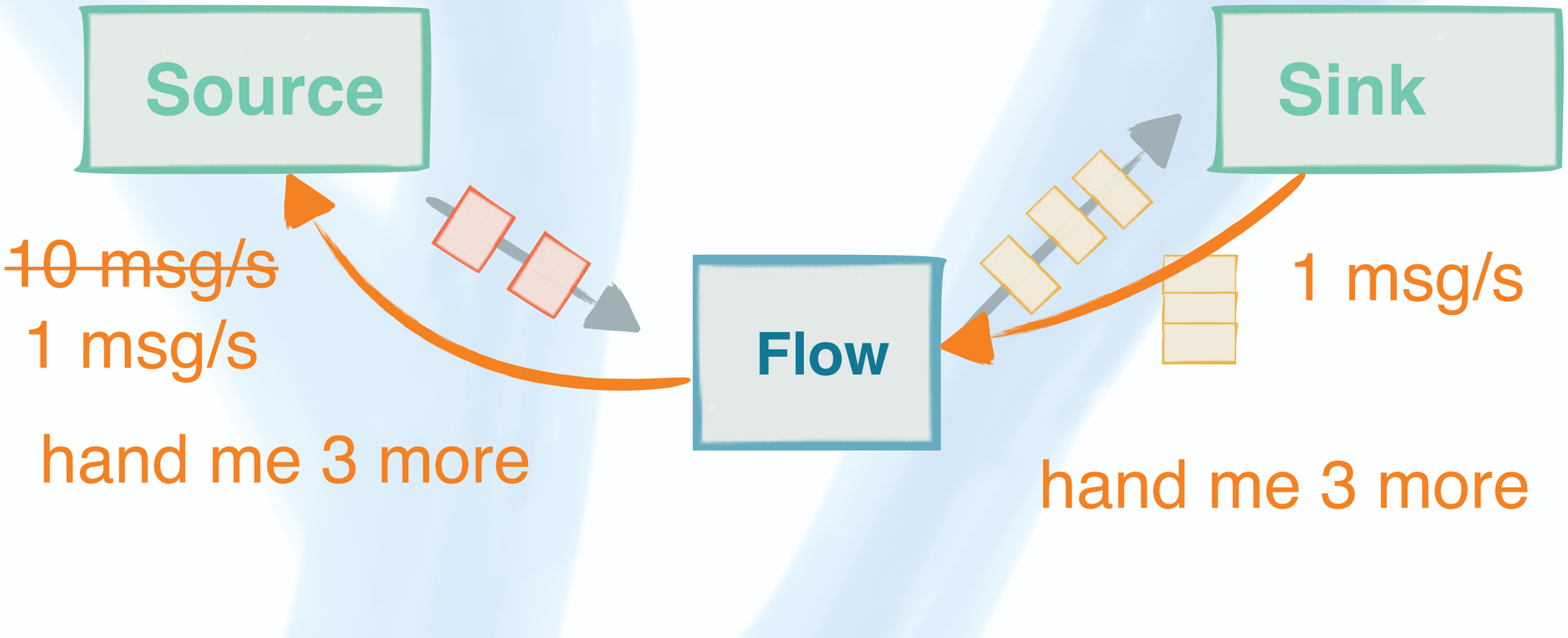
akka streams

Asynchronous **back pressured** stream processing



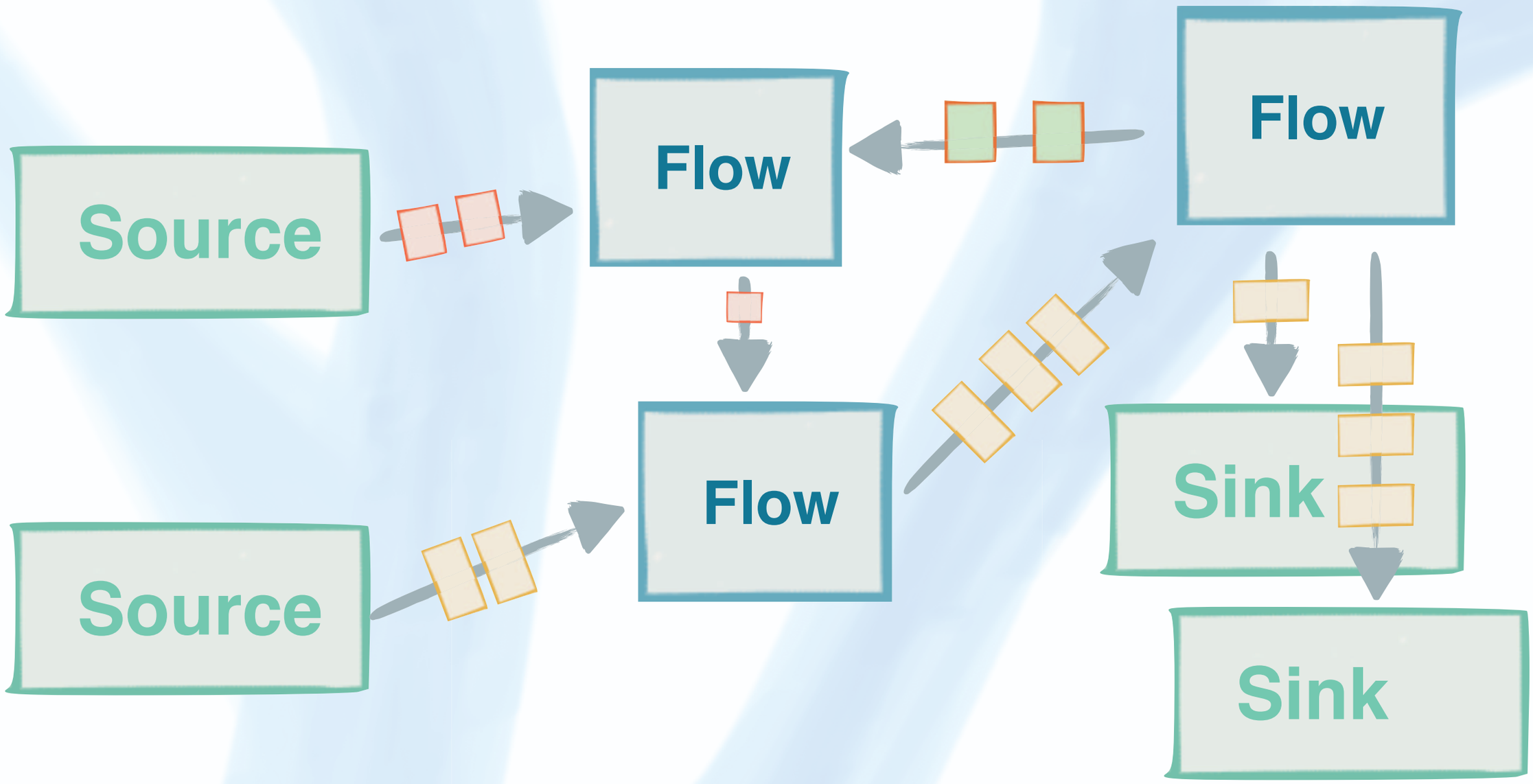
akka streams

Asynchronous **back pressured** stream processing



akka streams

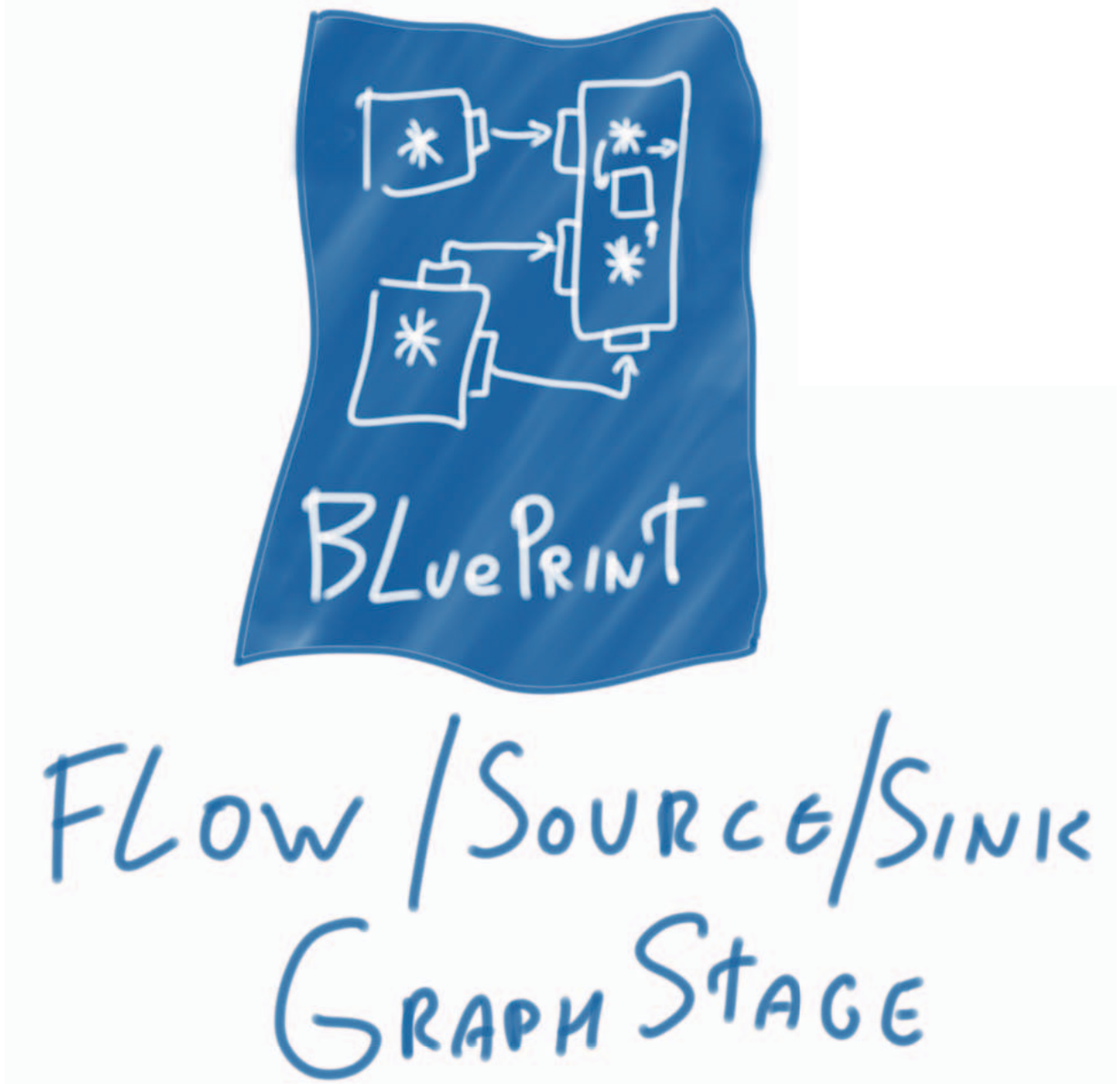
Not only linear streams



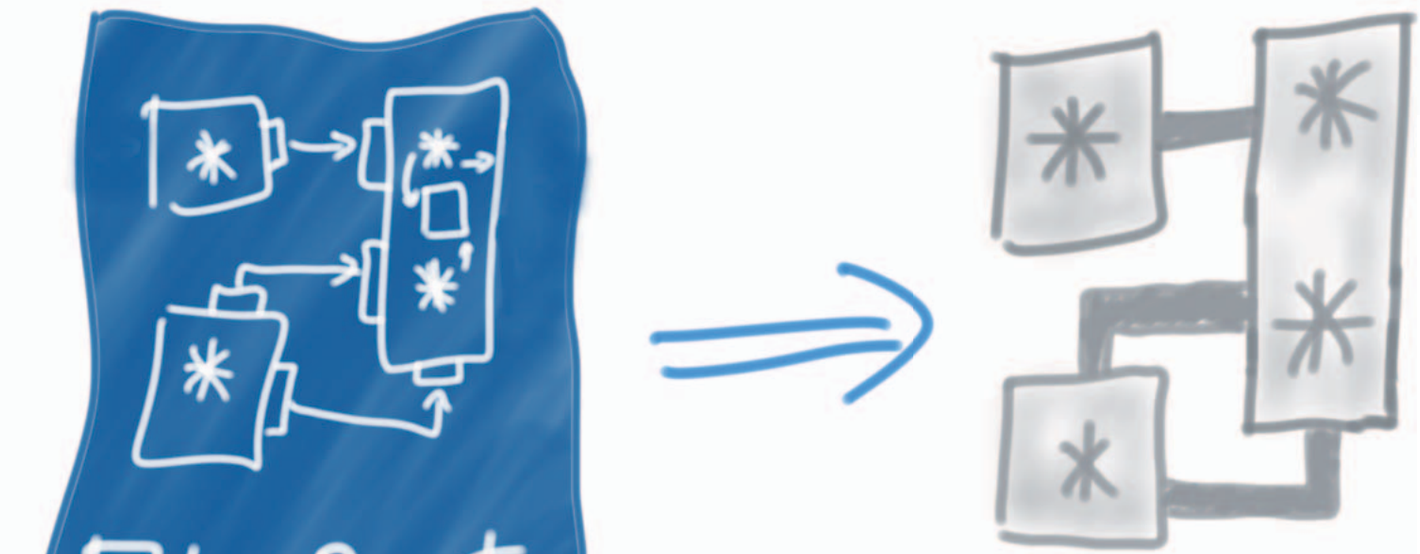
Materialization

Gears from GeeCON.org, did I mention it's an awesome conf?

What is “materialization” really?



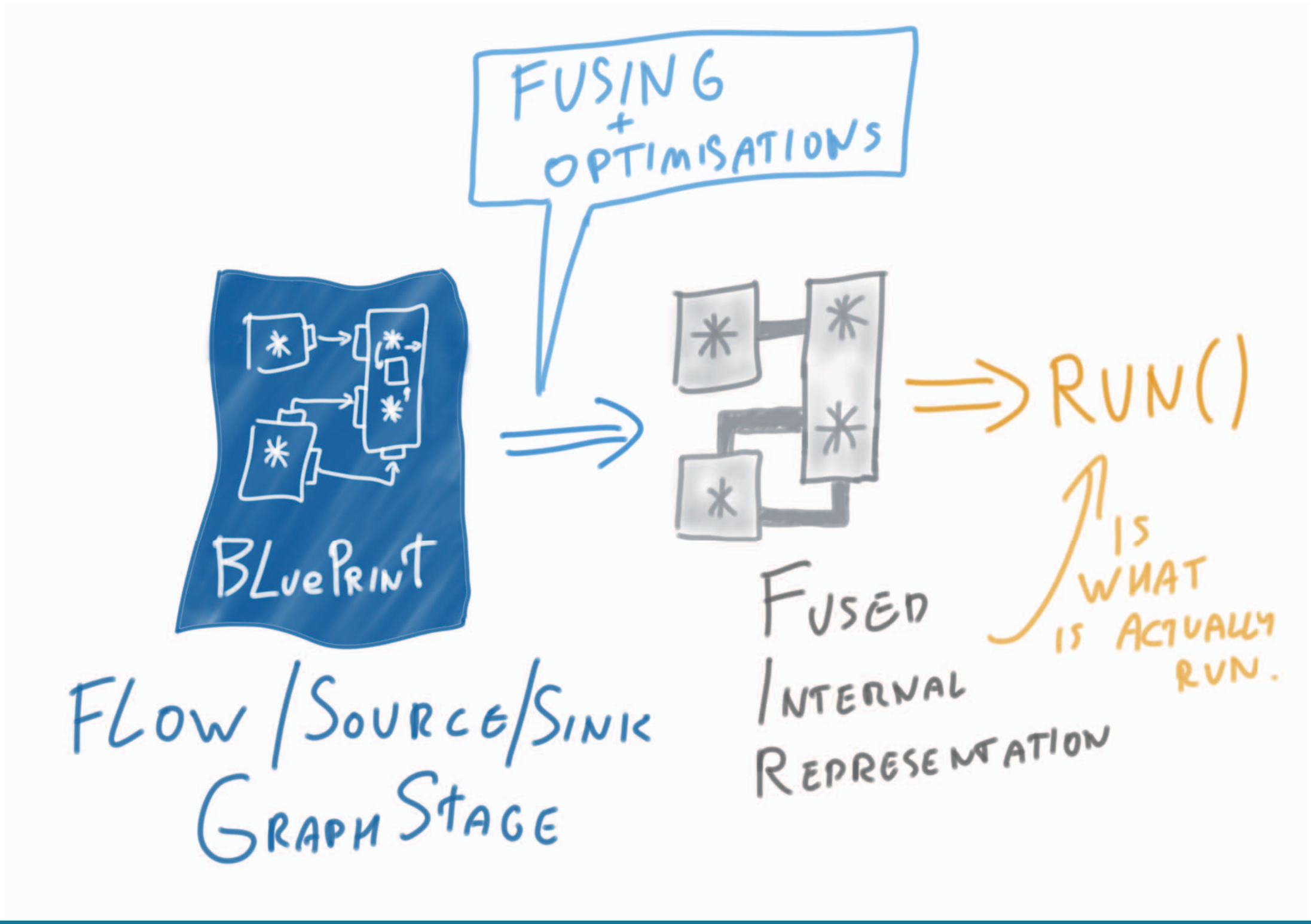
What is “materialization” really?



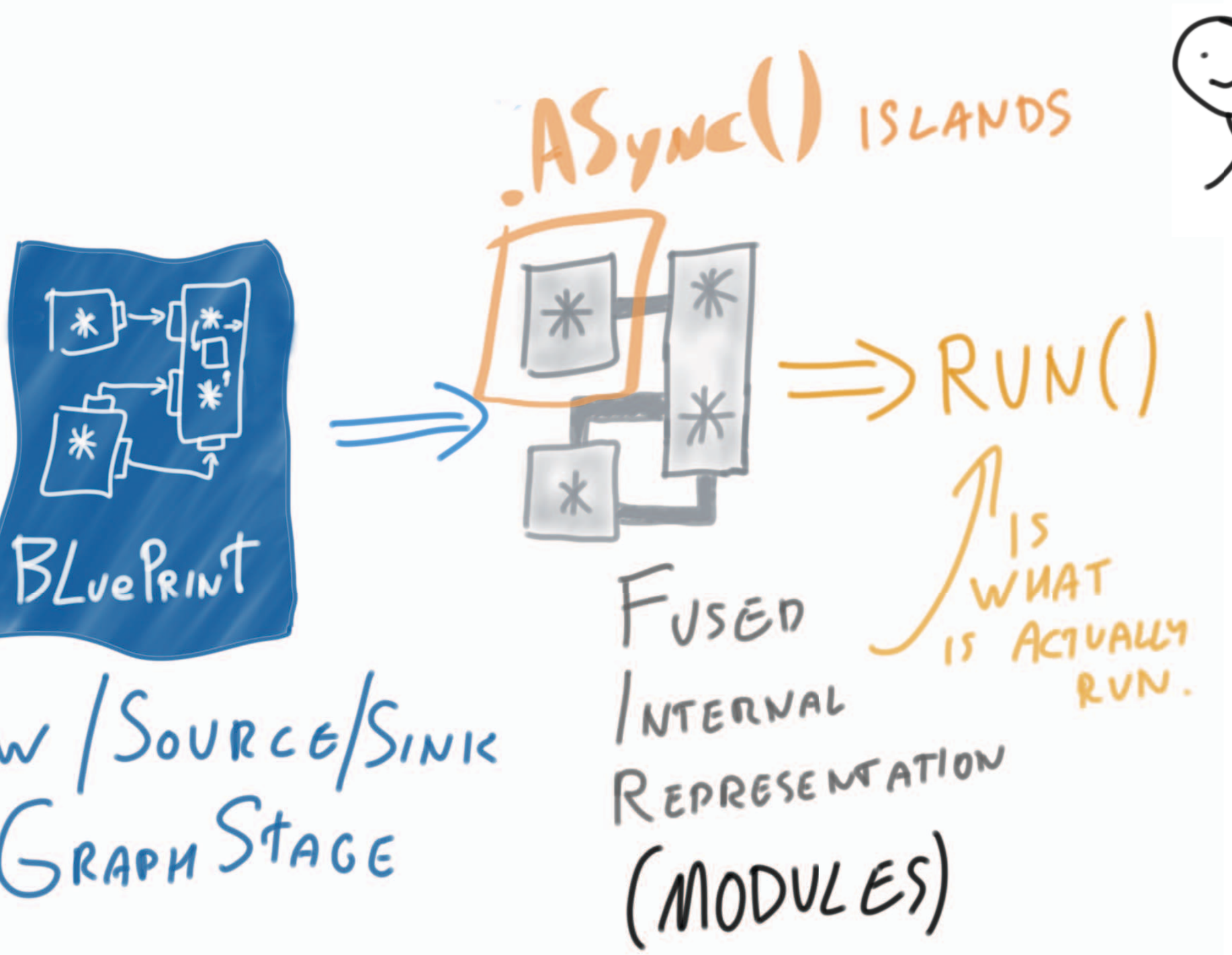
FLOW / SOURCE/SINK
GRAPH STAGE

FUSED
INTERNAL
REPRESENTATION

What is “materialization” really?



What is “materialization” really?



Building an echo Generator

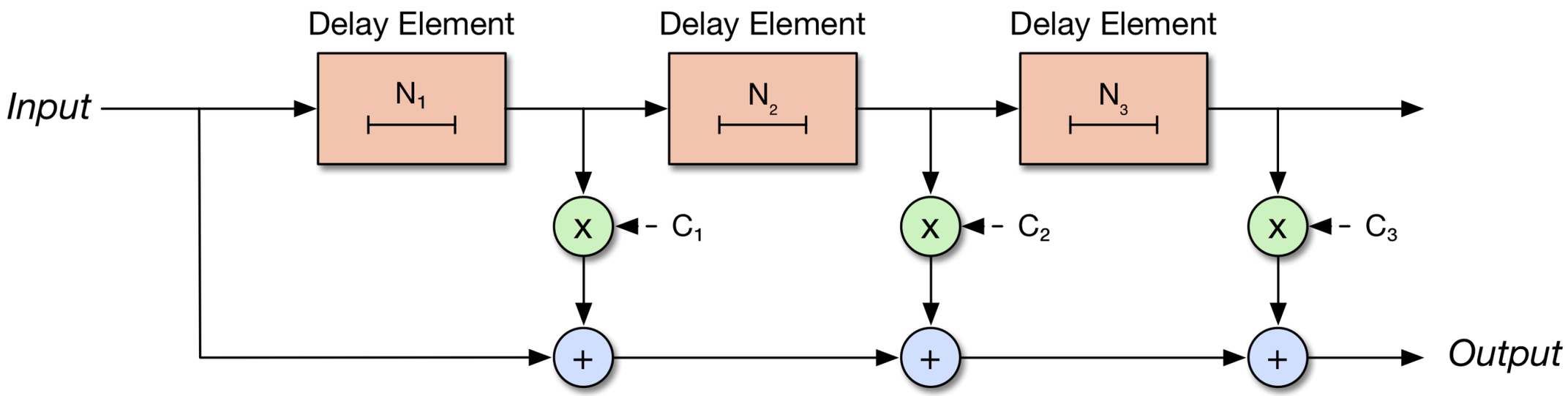
Let's build a sample application!

- Audio echo generator using Akka Streams
  - Use Finite & Infinite Impulse Response Filters
- Let's try to add echo-cancellation too !
- We'll utilise the latest released stuff:
  - Scala 2.12.3 (with Java 8)
  - Akka 2.5.6

Building an echo Generator

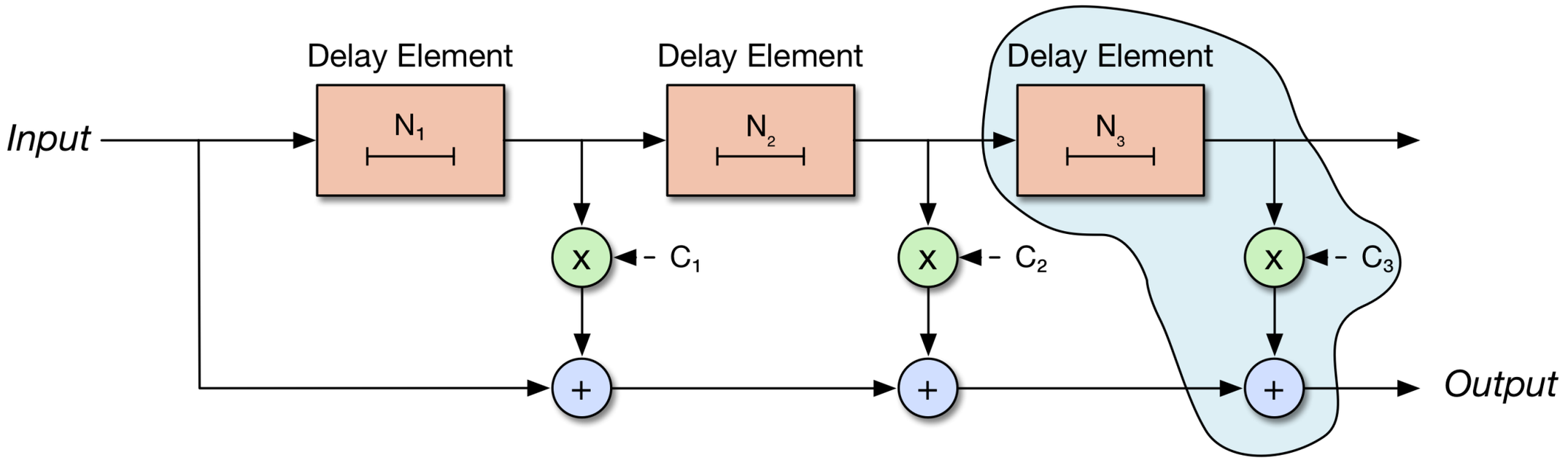
A Finite Impulse Response Filter

- If delays are large enough, 'echos' of the input are a result
- The below example generates three echoes



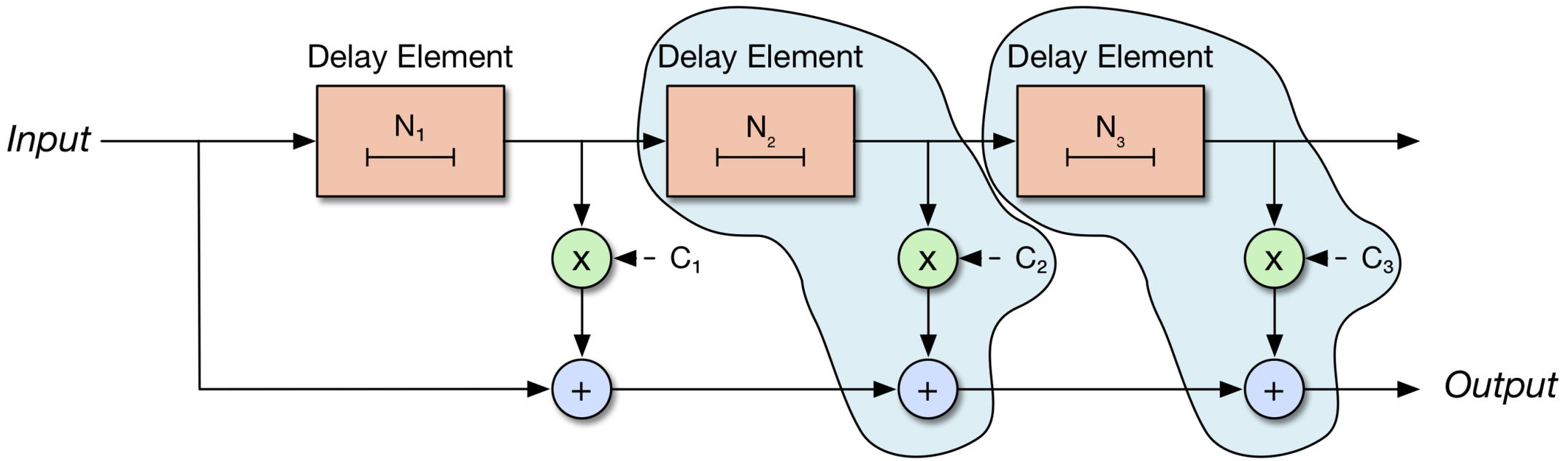
Building an echo Generator

Recurring Building Blocks



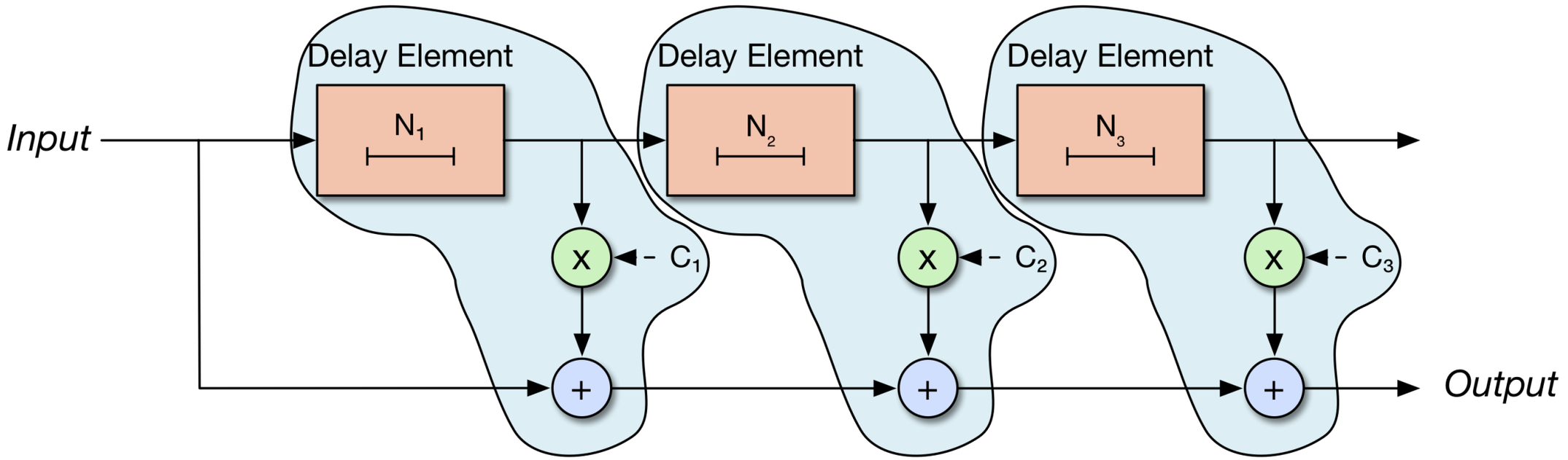
Building an echo Generator

Recurring Building Blocks



Building an echo Generator

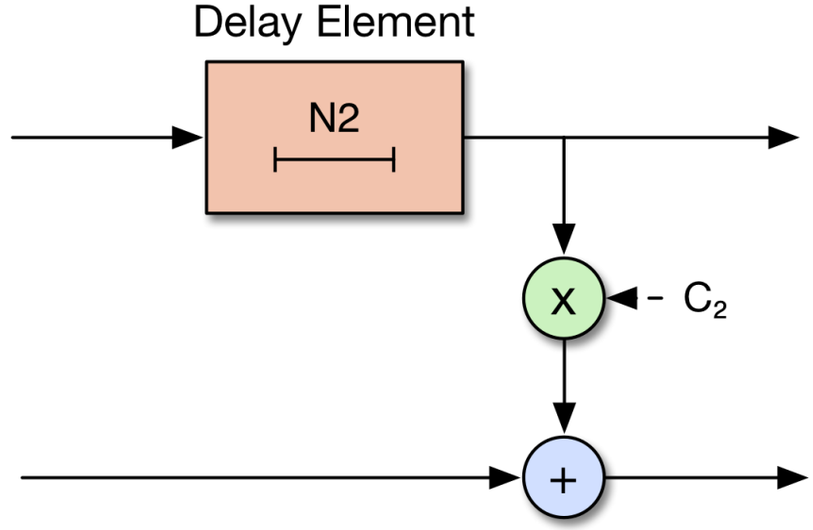
Recurring Building Blocks



Building an echo Generator

The *DelayLine* Building Block

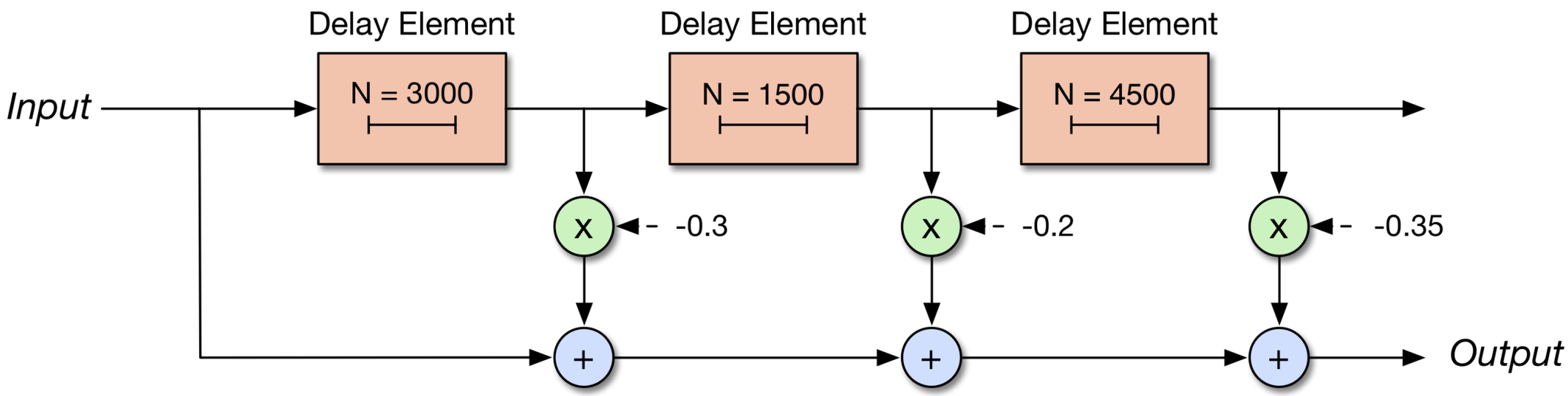
- A delay element
- A Multiplier
- An Adder



Building an echo Generator

Let's implement a (FIR-based) echo generator

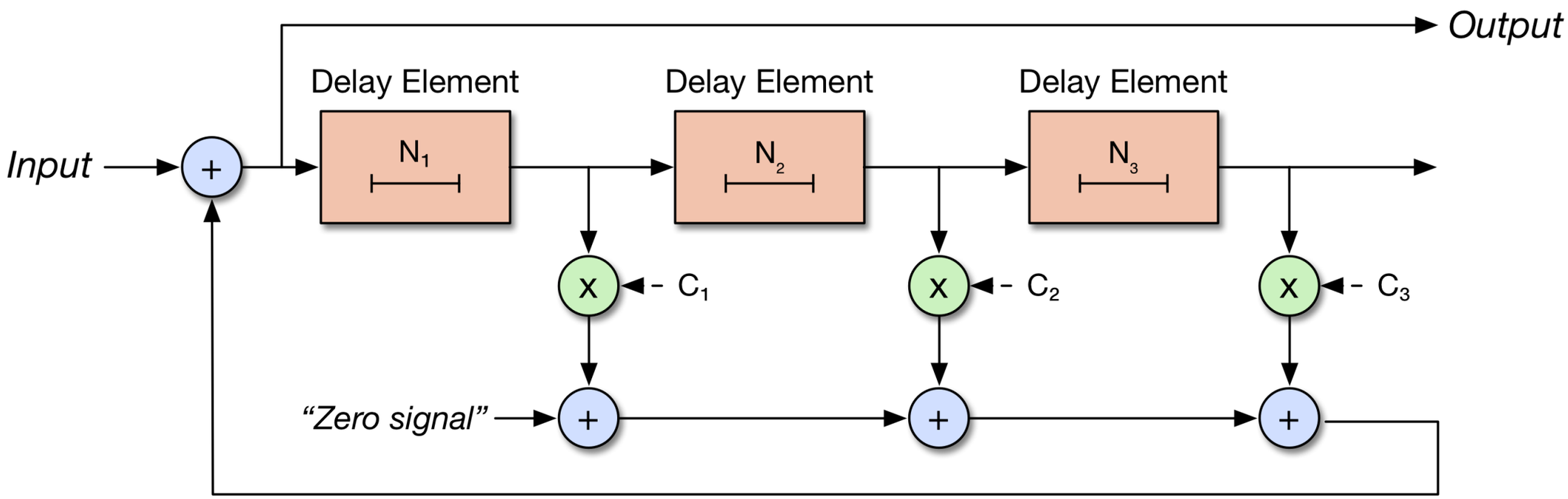
- 3 stages



Building an echo Generator

An Infinite Impulse Response Filter

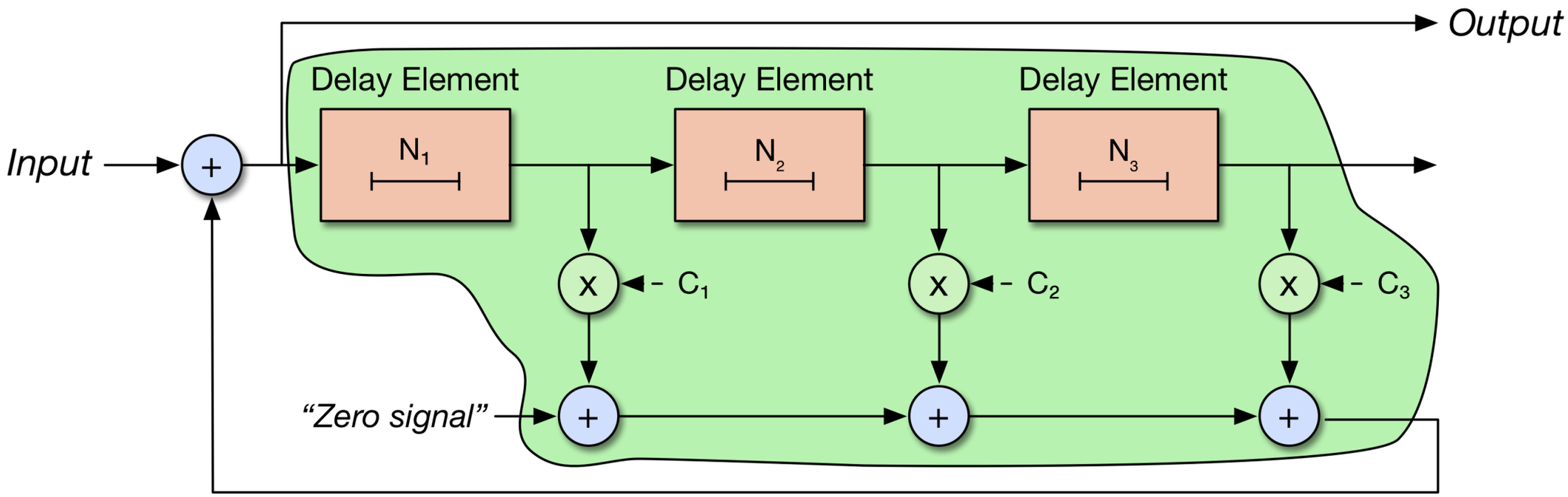
- Similar to an FIR filter, but IIR has feedback
- The below example generates an infinite number of echoes



Building an echo Generator

An Infinite Impulse Response Filter

- Hey, this IIR filter actually contains an FIR filter !



Building an echo Generator

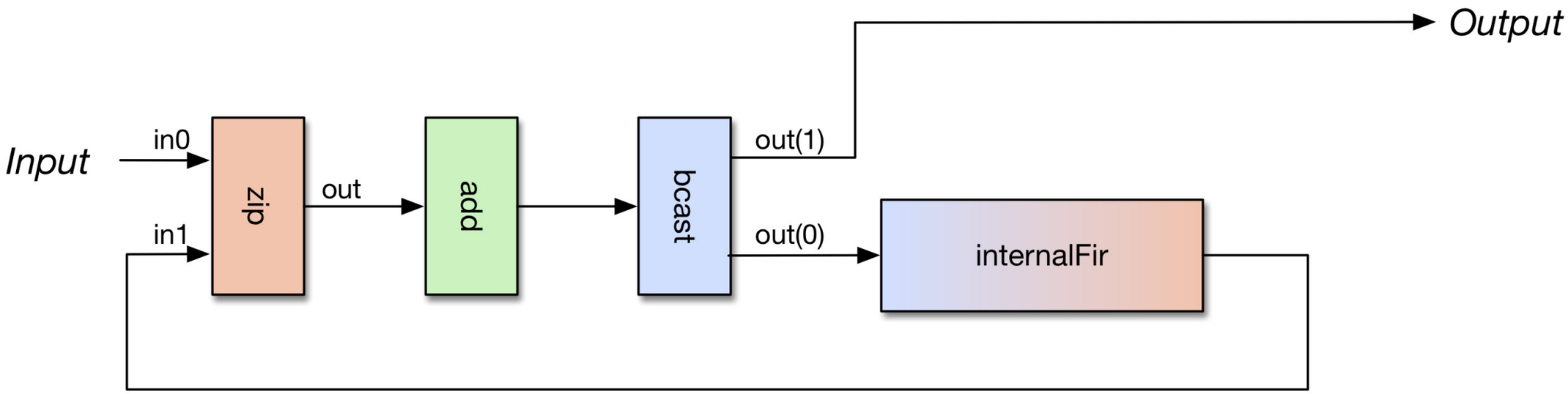
Implementing an Infinite Impulse Response Filter

- We've already got adders and multipliers
- We'll need some extra tools like zippers, broadcasts

Building an echo Generator

An Infinite Impulse Response Filter

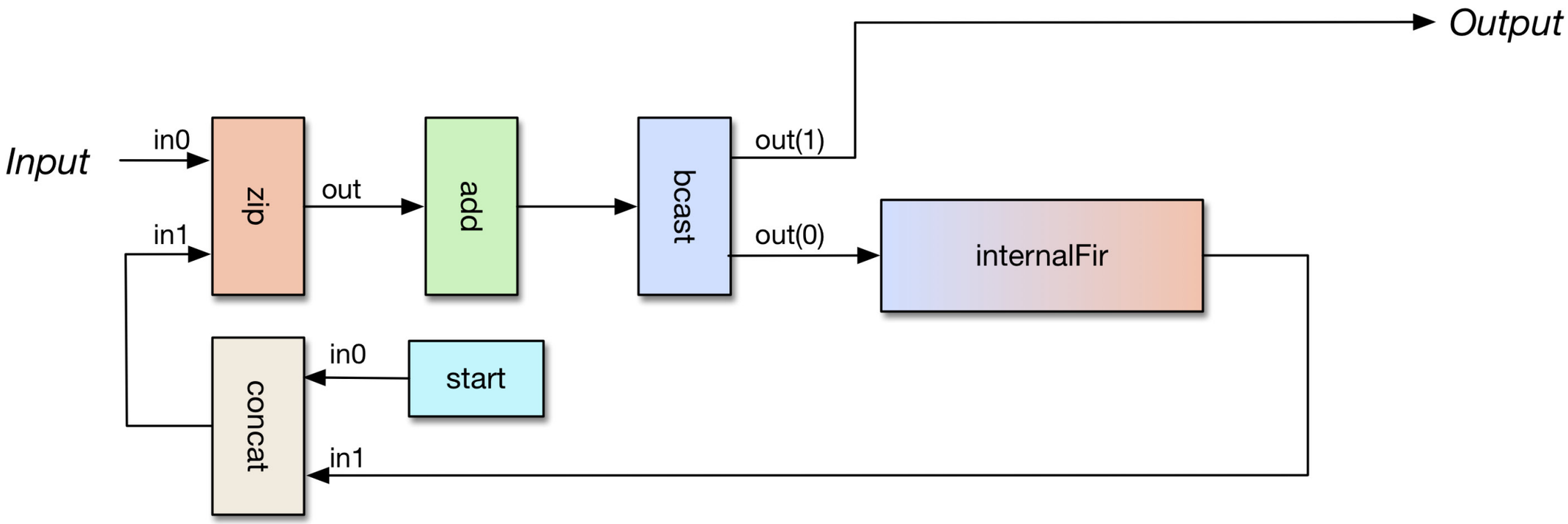
- Translated to an Akka Streams Implementation



Building an echo Generator

An Infinite Impulse Response Filter

- Translated to an Akka Streams Implementation



Further reading:

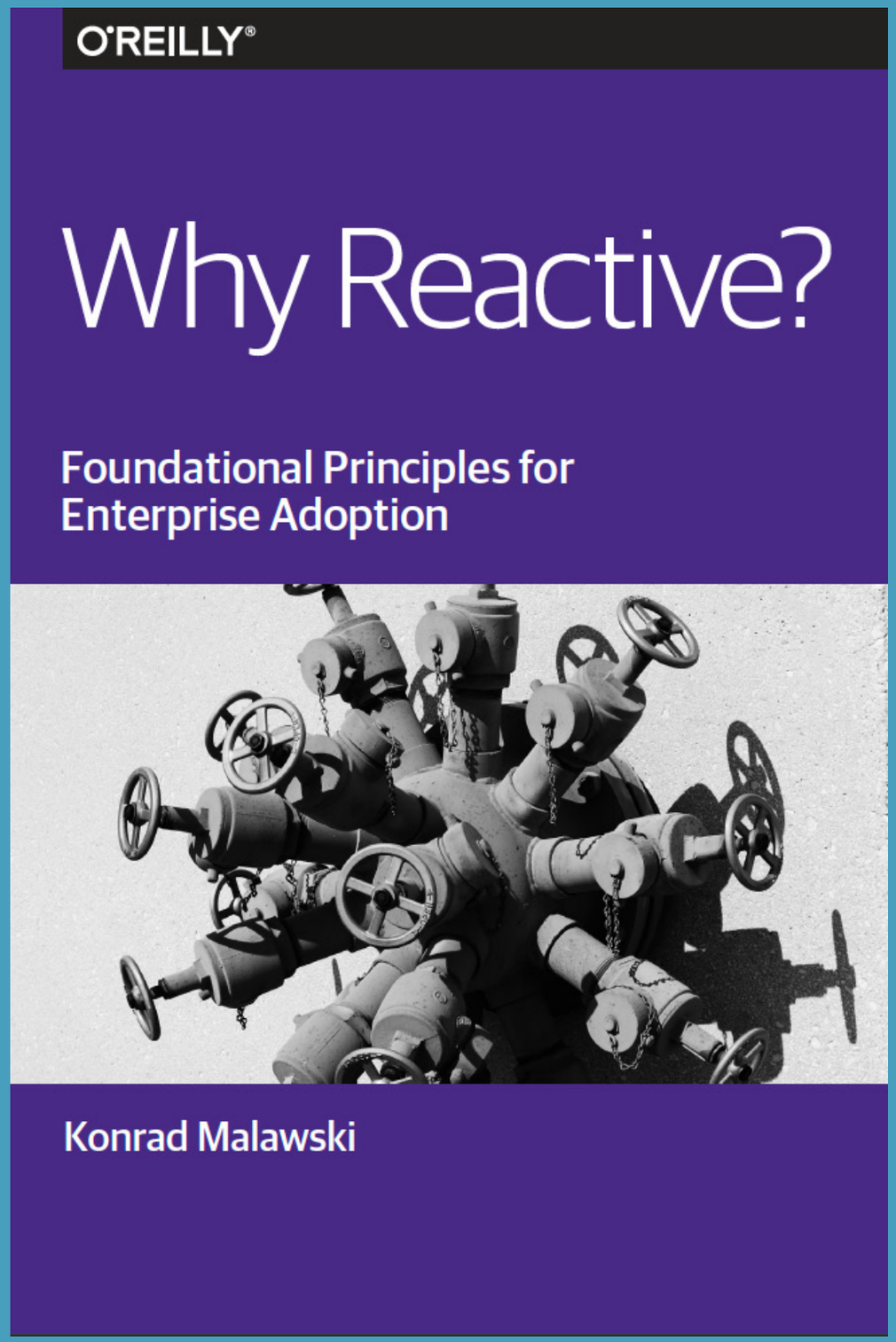
Reactive Streams: reactive-streams.org
Akka documentation: akka.io/docs
Free **O'Reilly** report → → → → → → → →

Get involved:

sources: github.com/akka/akka
mailing list: akka-user @ google groups
gitter channel: https://gitter.im/akka/akka

Contact:

eric.loots@lightbend.com



Questions ?