Lunatech Labs

March, 2021

Applied Akka Streams

Agenda

Agenda

- Introduction
- SourceWithContext
- Substreams
- Stateful streams
- Flow from Sink and Source
- Async boundaries
- GraphDSL with internal feedback
- Stream rate adaptation
- Things we haven’t covered
- Wrap-up

Introduction

- We will cover more advanced topics by
  - Running through example code
  - Doing exercises
- Feel free to jump in if you can comment about your personal experience with Akka Streams

SourceWithContext

Stateful (sub-)streams

Keeping state

- We saw stateful sources (eg. Source.fromIterator) but Flows can do this too

```
statefulMapConcat[T](f: () => O => IterableOnce[T]): Flow[I, T, _]
```

Exercise: using stateful streams

- We will write a simple data analysis stream
- Read the exercise descriptions to get started.
  NB. Don’t miss the part about pulling in the templates.

Substreams

- Split a stream into many identical substreams
  - But: each with individual state and async runtime

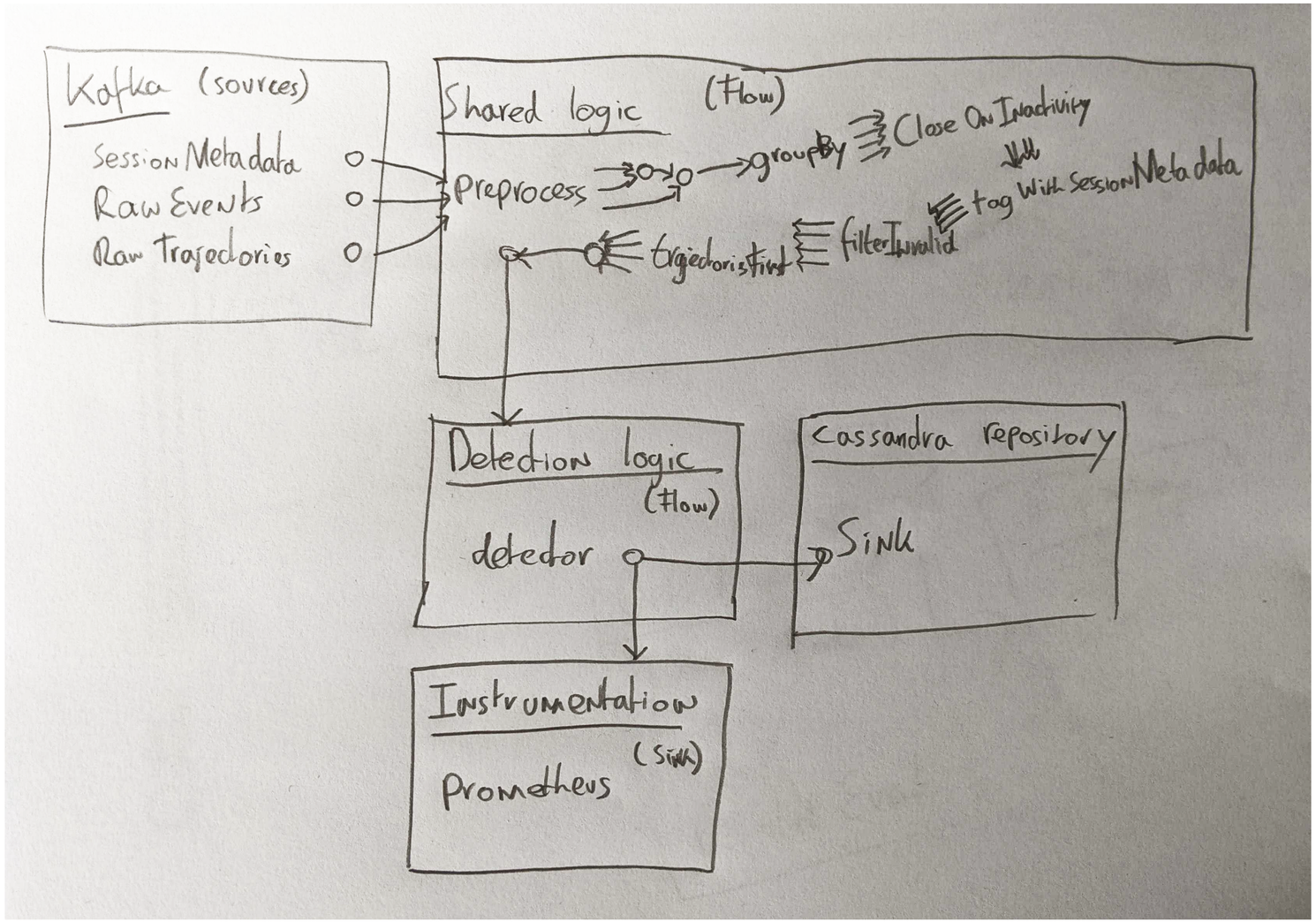
```
groupBy[K](
       maxSubstreams: Int,
       f:(O) => K,
       allowClosedSubstreamRecreation: Boolean = false
     ): SubFlow[O, Mat, Flow[I, O, NotUsed], Closed]
```

- Substreams are merged at some stage

```
mergeSubstreams: Flow[I, O, NotUsed]
```

Substreams

- Application of Substreams in a real-life application



Exercise: apply substreams

- Transform the data analysis stream to work with multiple vehicles
- Bonus exercise: take limits and long-running processes into account. What happens if a car stops sending messages? What should happen?

Scanning for state

- statefulMapConcat is nice but relies on mutability
- Alternative of equal expressiveness:

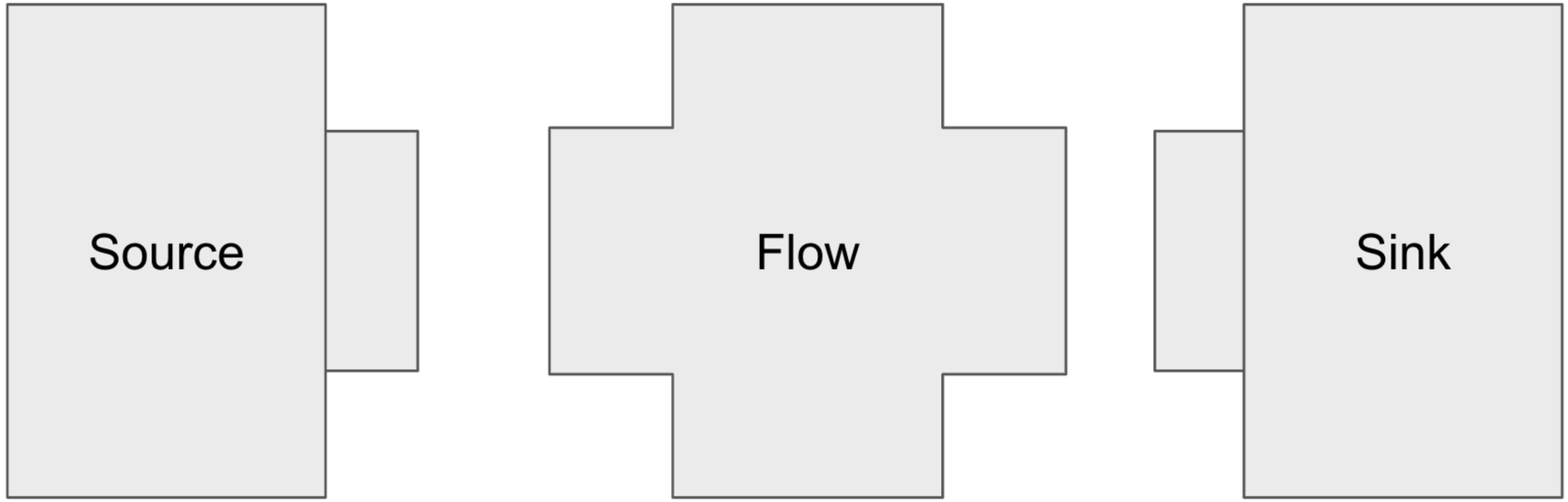
```
scan[T](zero: T)(f: (T, O) => T): Flow[I, T, _]
```

Exercise: use scan

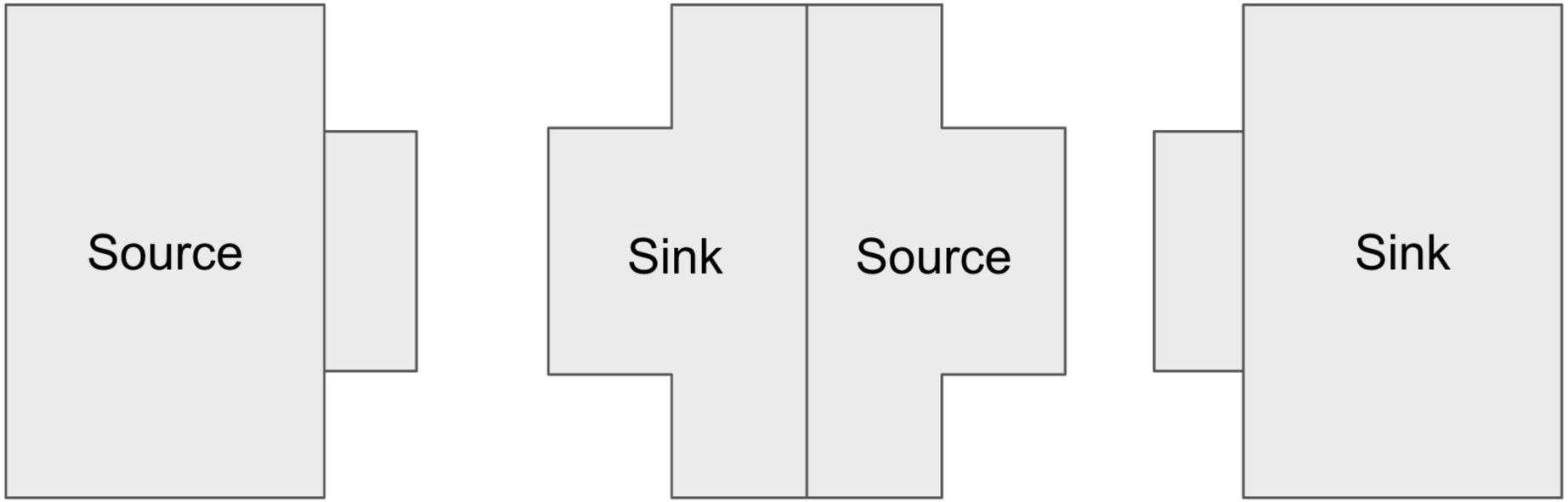
- Refactor the analysis logic, remove statefulMapConcat and use scan

Flow from Sink and Source

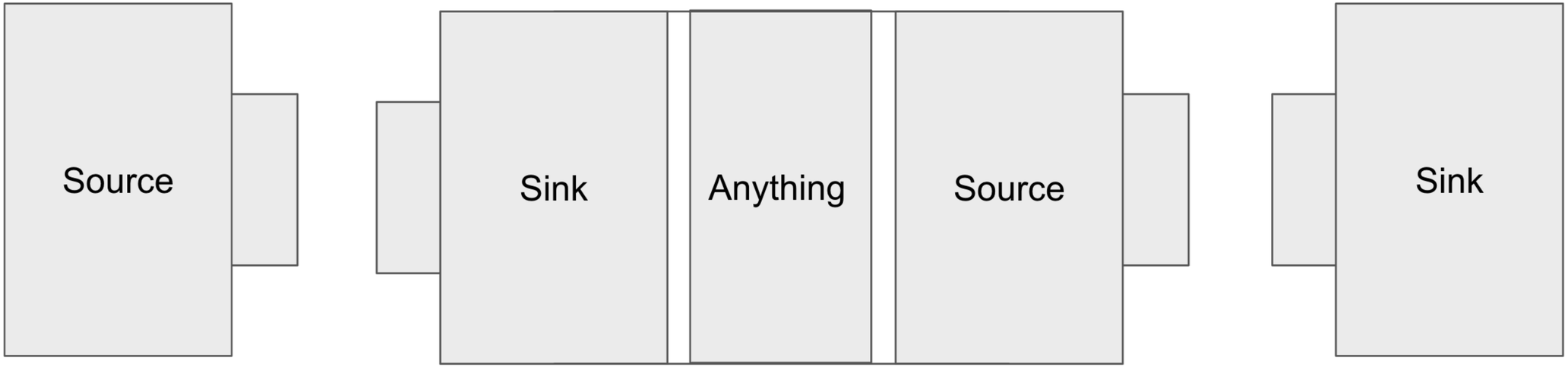
Flow from Sink and Source



Flow from Sink and Source



Flow from Sink and Source

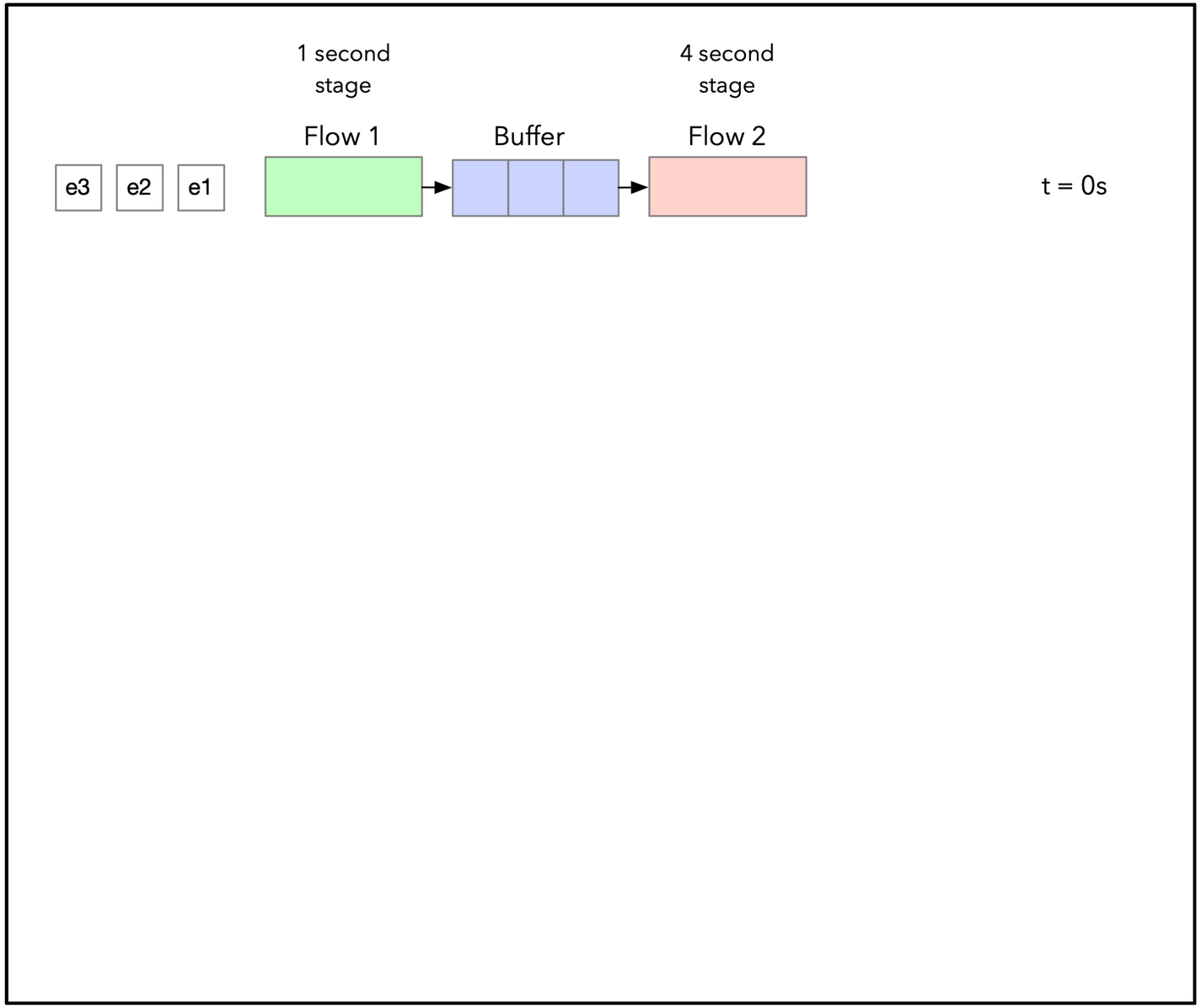


Async Boundaries

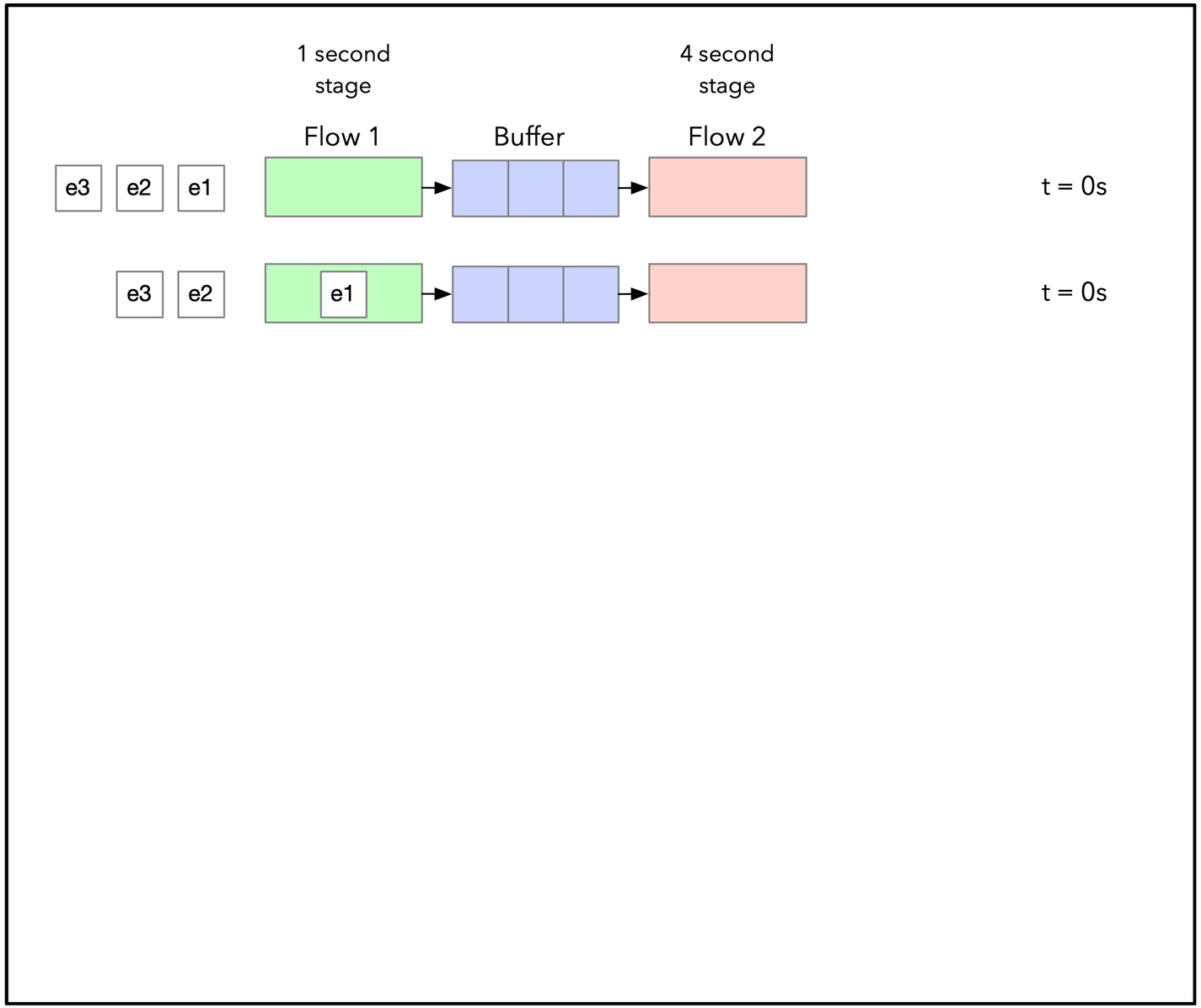
Async Boundaries

- An asynchronous boundary introduces an extra actor that can run subsequent Stream Components asynchronously
- A buffer is introduced too (default size = 16)
- Throughput may increase because of pipelining of processing

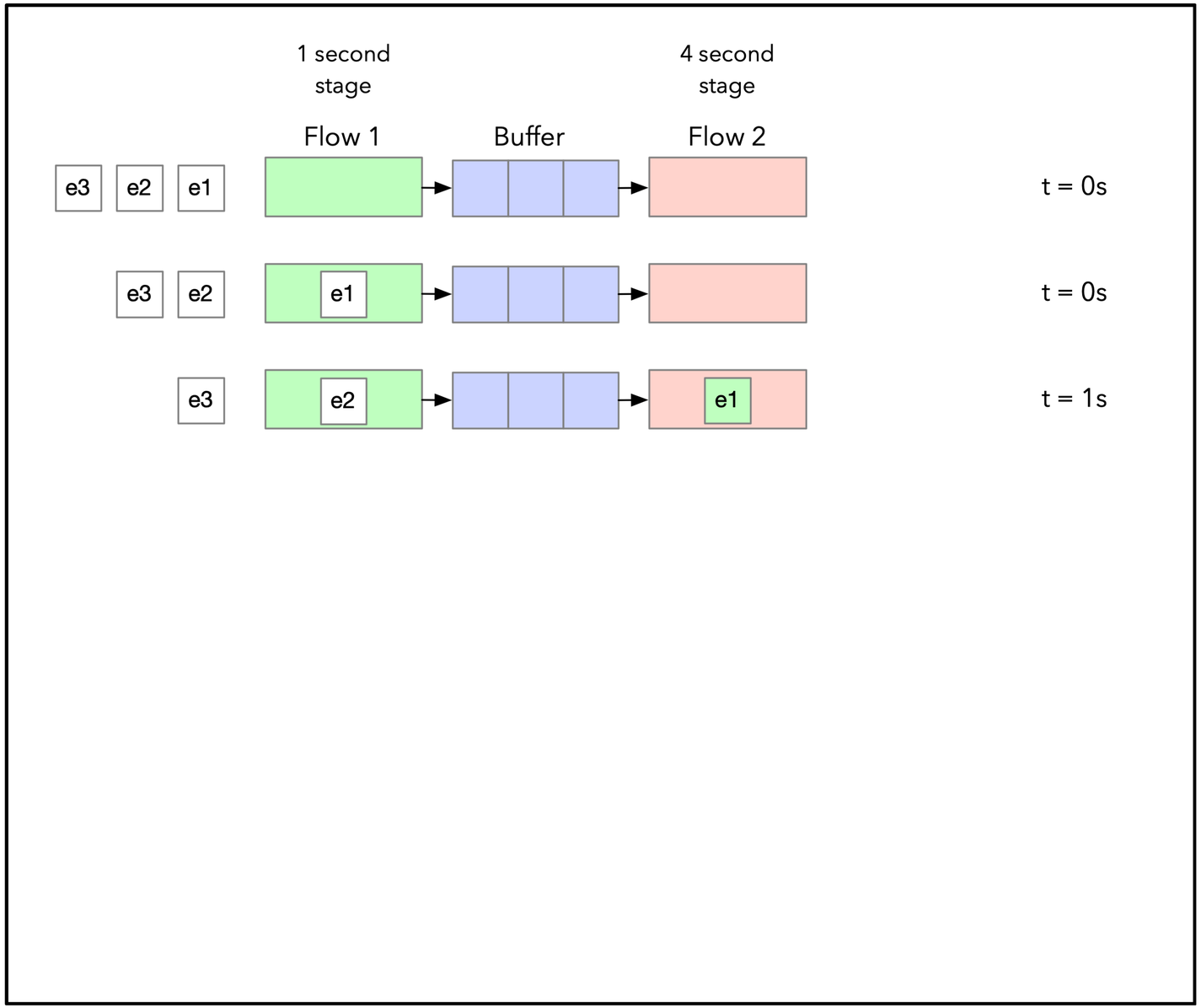
Async Boundaries - Pipelining



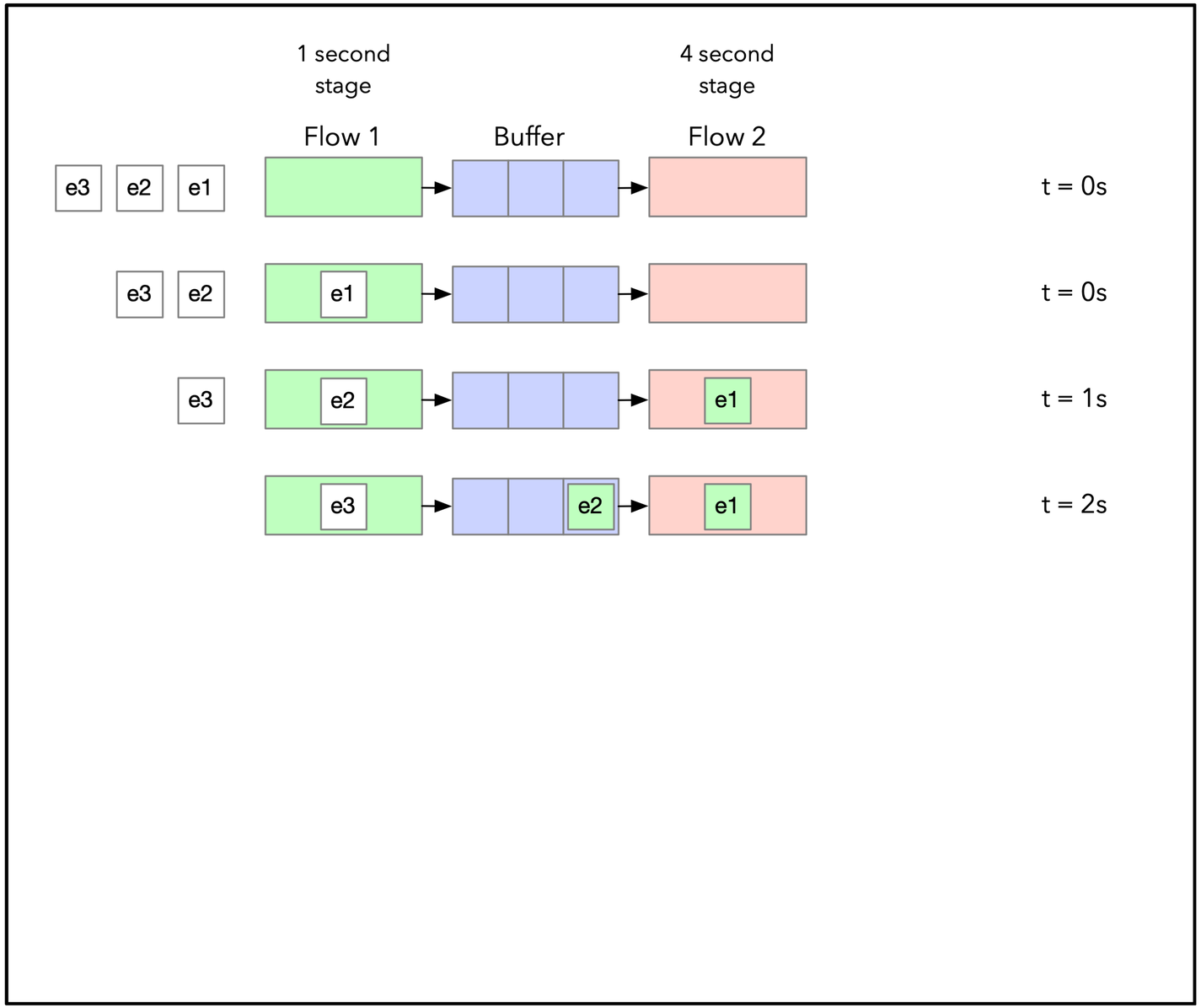
Async Boundaries - Pipelining



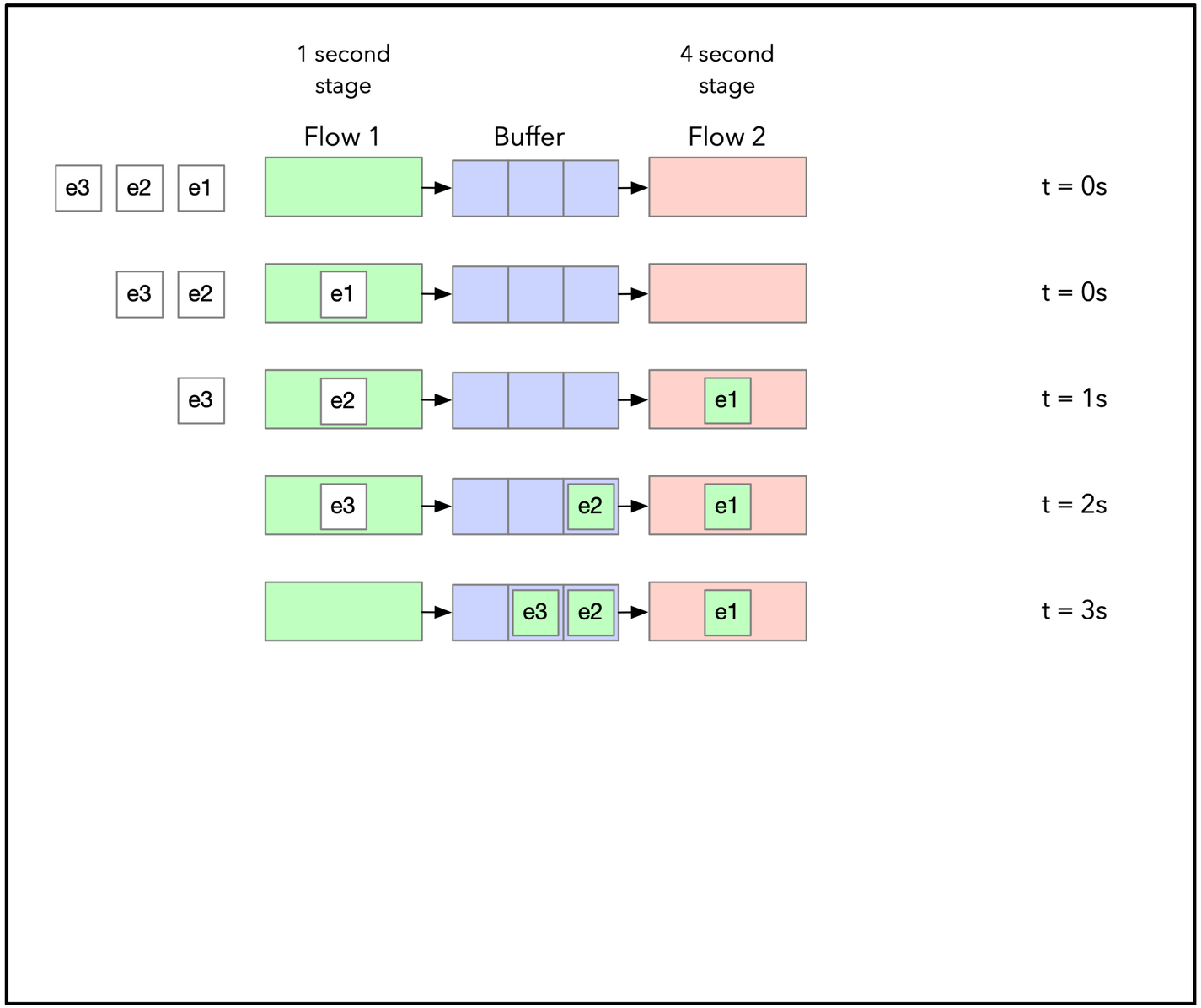
Async Boundaries - Pipelining



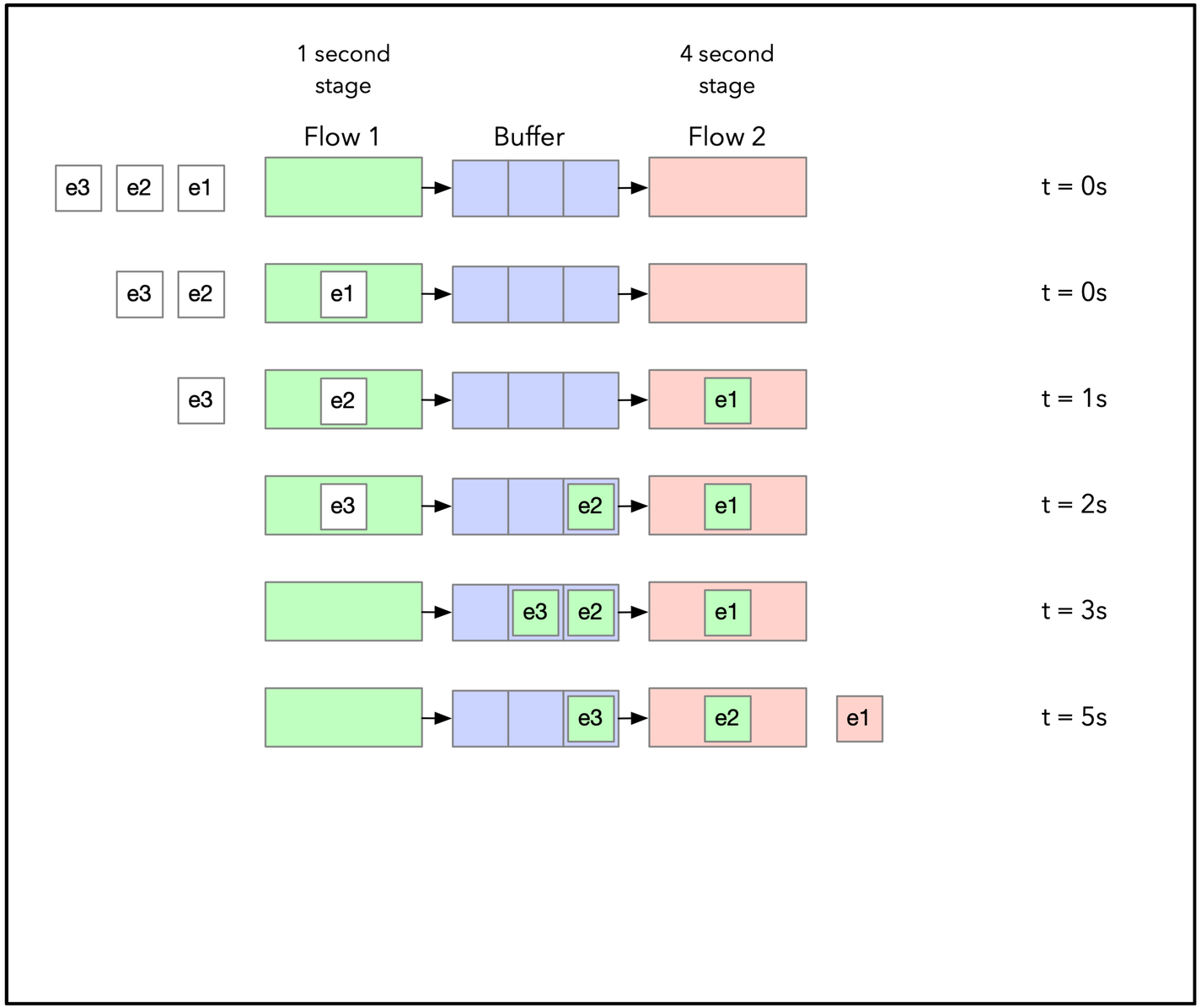
Async Boundaries - Pipelining



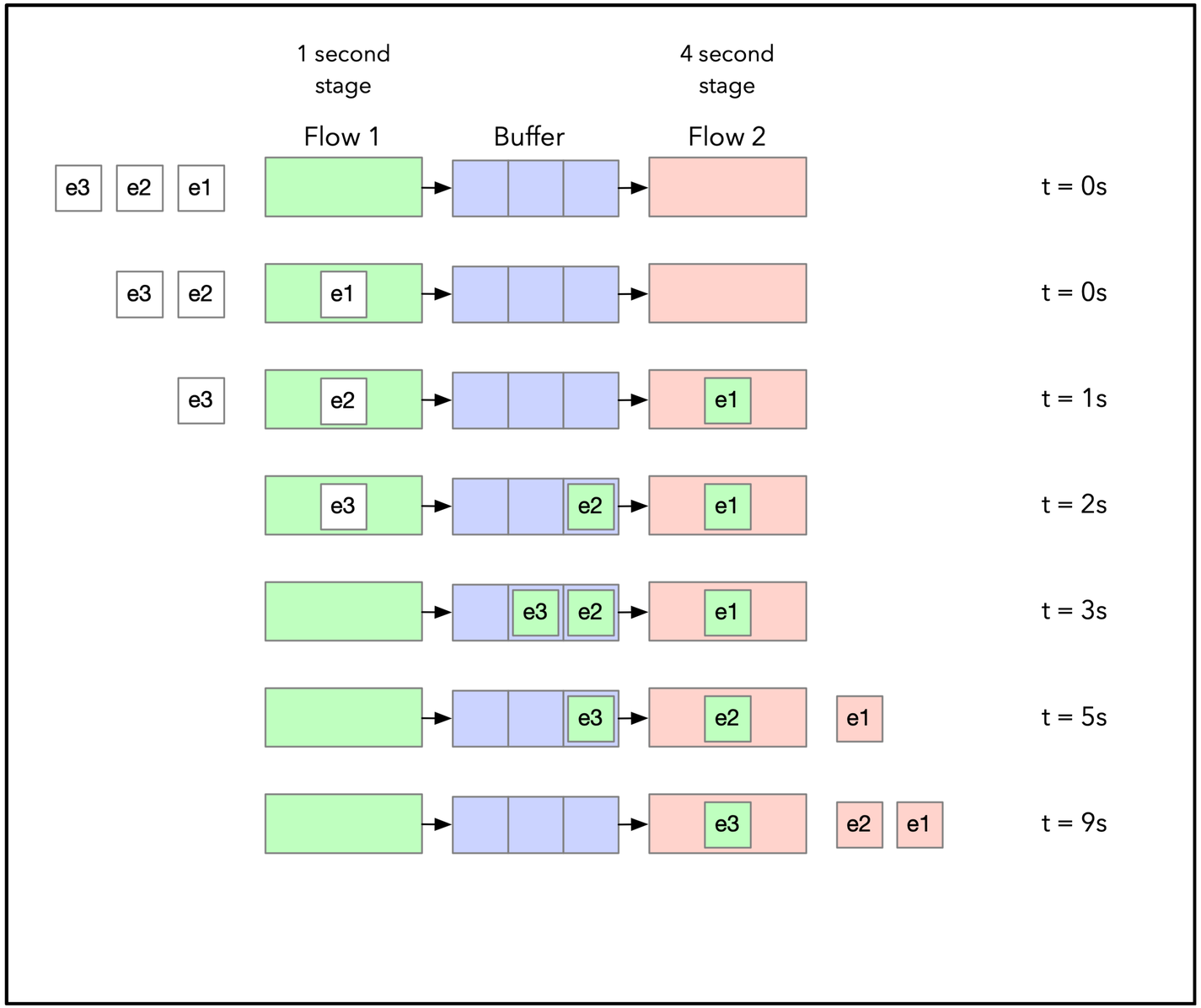
Async Boundaries - Pipelining



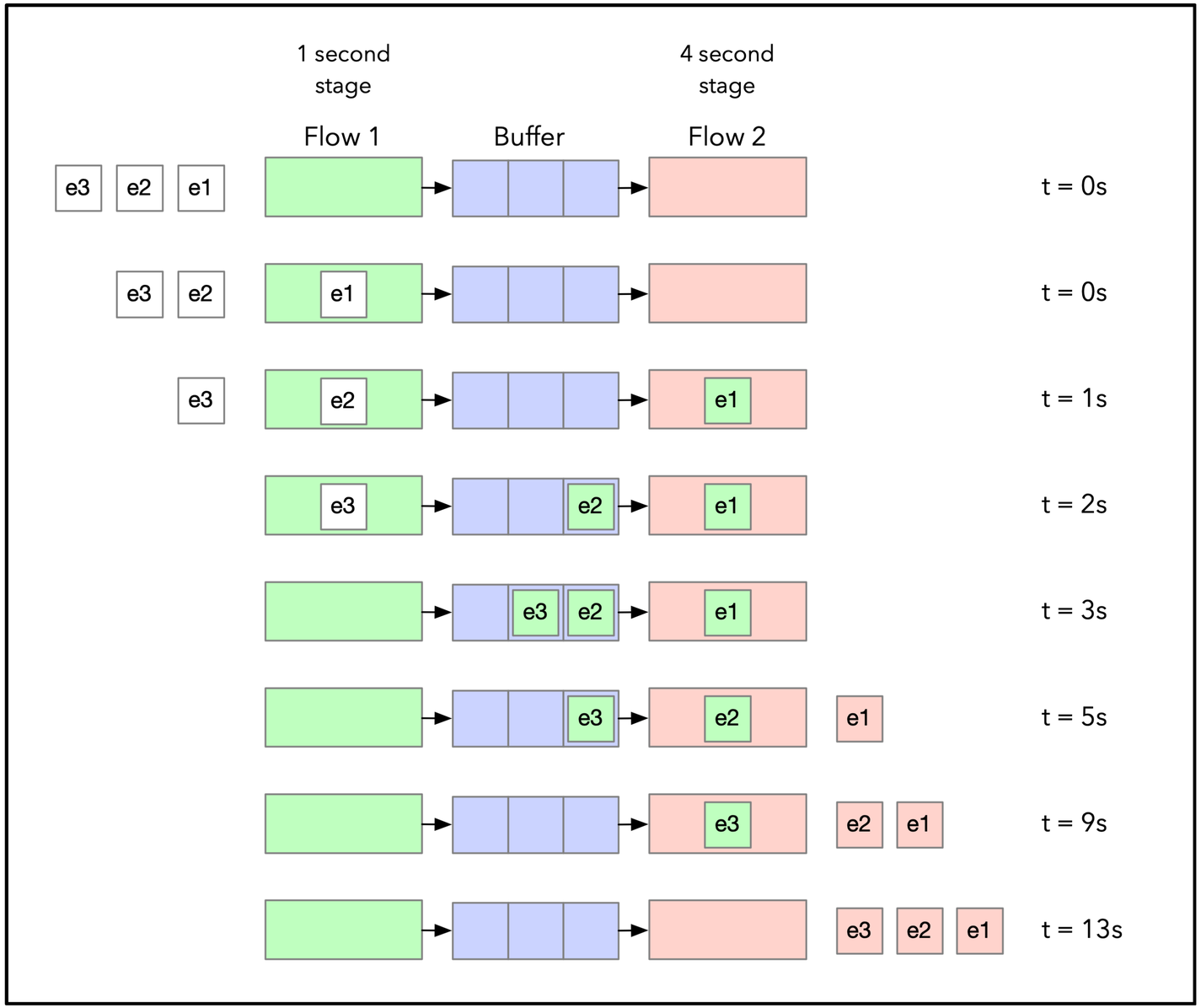
Async Boundaries - Pipelining



Async Boundaries - Pipelining



Async Boundaries - Pipelining



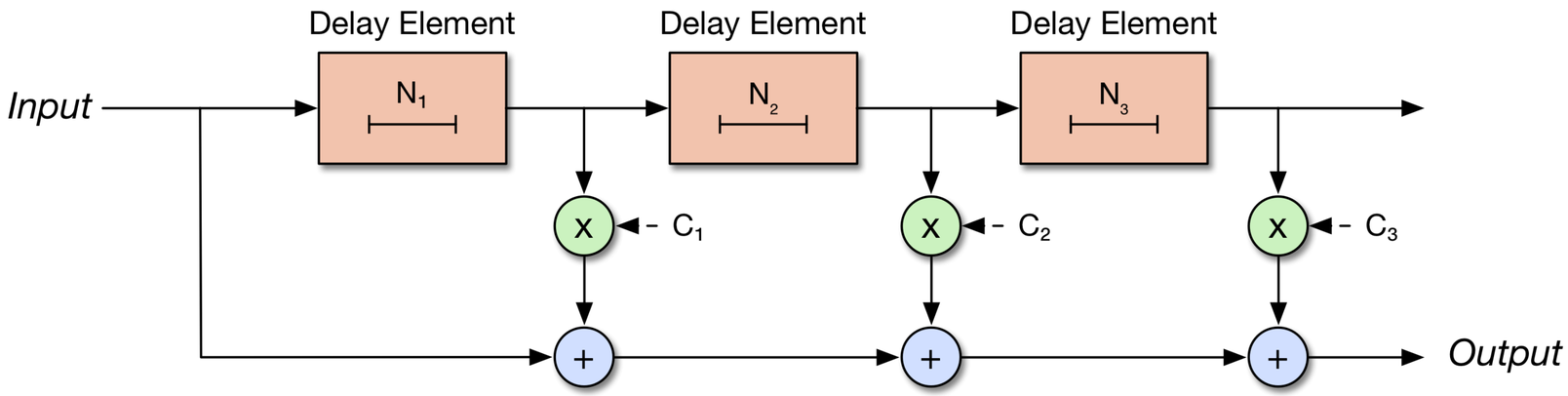
Using Akka Streams to perform some Signal Processing

Building an echo Generator

- Audio echo generator using Akka Streams
  - Use Finite & Infinite Impulse Response Filters
- Let’s try to add echo-cancellation too !

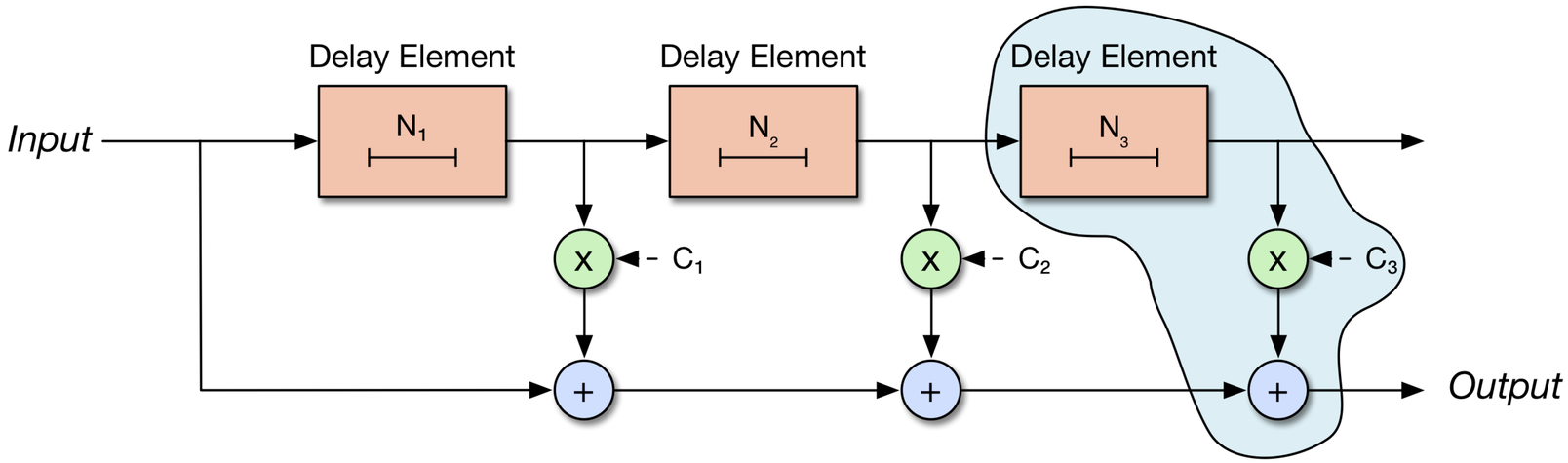
A Finite Impulse Response Filter (FIR)

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



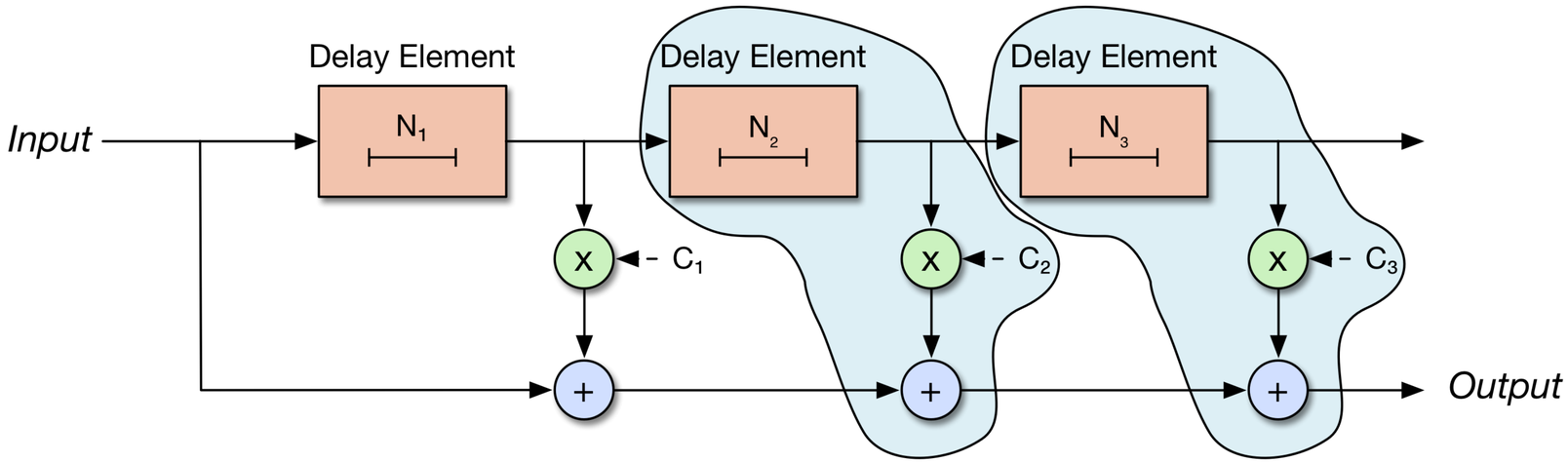
A Finite Impulse Response Filter (FIR)

- Recurring building blocks



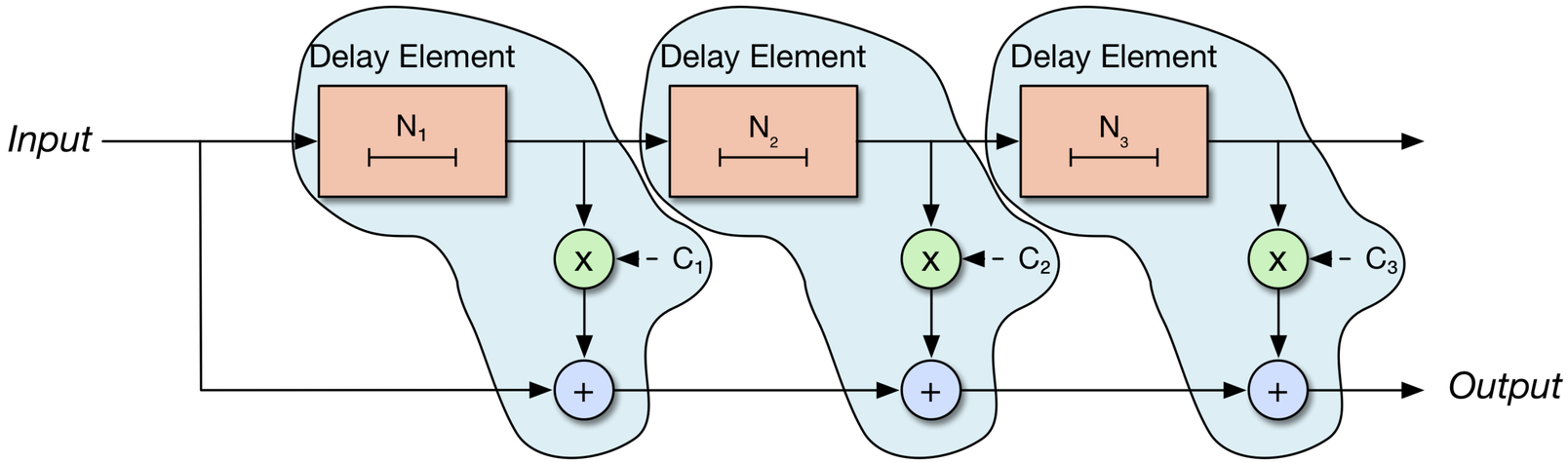
A Finite Impulse Response Filter (FIR)

- Recurring building blocks



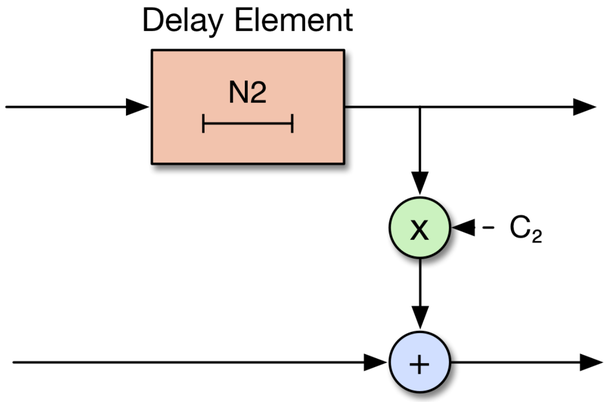
A Finite Impulse Response Filter (FIR)

- Recurring building blocks



The *Delay Line*

- The ***DelayLine*** Building Block
  - A delay element
  - A Multiplier
  - An Adder



Building a Delay Line stream component

In this exercise, we will build a basic delay line component using Akka Streams

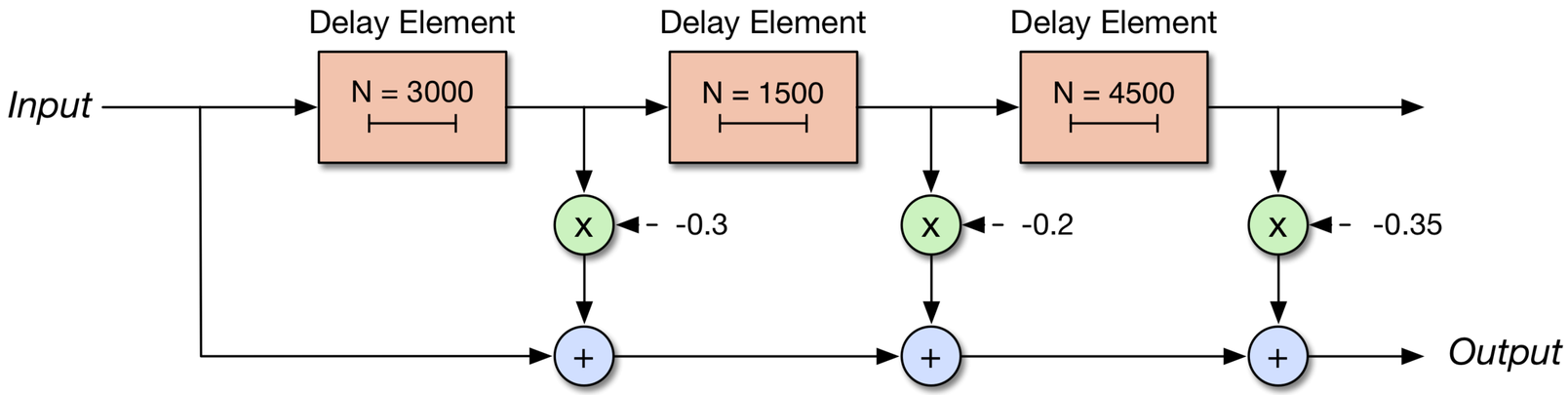
- Start an sbt session in the root folder of the project
- Make sure your sbt prompt should look as follows:

```
man [e] > Lunatech Akka Streams with Scala > implement a delay element >
```

- Follow the exercise instructions by executing the `man e` command in the sbt prompt

Building an echo generator

- With the Delay Line in place, we can take the next step
- Let's implement a 3-stage (FIR-based) echo generator



Building an FIR based echo generator - I

In this exercise, we will build a 3-stage echo generator using the DelayLine component we built before

- Start an sbt session in the root folder of the project
- Make sure your sbt prompt should look as follows:

```
man [e] > Lunatech Akka Streams with Scala > implement fir manually >
```

- Follow the exercise instructions by executing the `man e` command in the sbt prompt

Building an FIR based echo generator - II

This is an optional exercise

- Start an sbt session in the root folder of the project
- Make sure your sbt prompt should look as follows:

```
man [e] > Lunatech Akka Streams with Scala > implement fir streamlined >
```

- Follow the exercise instructions by executing the `man e` command in the sbt prompt

Building an IIR based echo generator - I

In this exercise, we set the stage for the next exercise in which we will implement an IIR based echo generator

- Start an sbt session in the root folder of the project
- Make sure your sbt prompt should look as follows:

```
man [e] > Lunatech Akka Streams with Scala > implement iir set stage >
```

- Follow the exercise instructions by executing the `man e` command in the sbt prompt

Building an IIR based echo generator - II

In this exercise, we will implement an IIR based echo generator

- Start an sbt session in the root folder of the project
- Make sure your sbt prompt should look as follows:

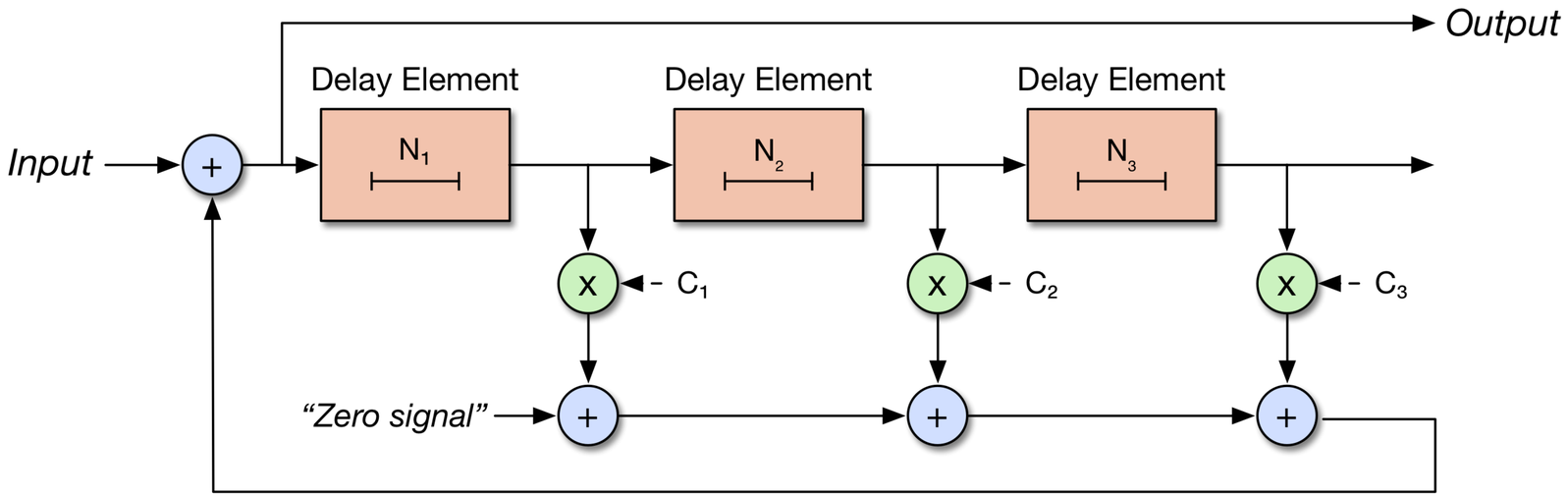
```
man [e] > Lunatech Akka Streams with Scala > implement iir >
```

- Follow the exercise instructions by executing the `man e` command in the sbt prompt

An Infinite Impulse Response Filter (IIR)

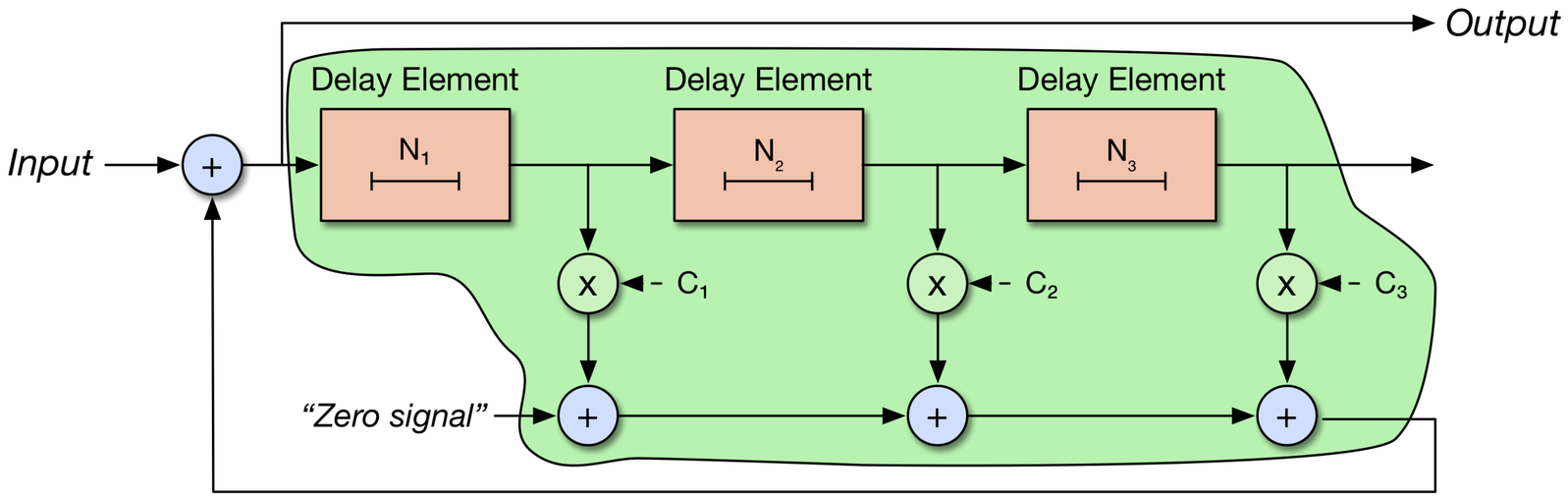
- Similar to an FIR filter, but IIR has feedback

  The below example generates an infinite number of echoes



The link between FIR and IIR

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



Do we have the right tools?

Do we have the right tools?

We already have adders

Do we have the right tools?

We already have adders

… and multi-pliers

Do we have the right tools?

We already have adders

We just need to add

Zippers

… and multi-pliers

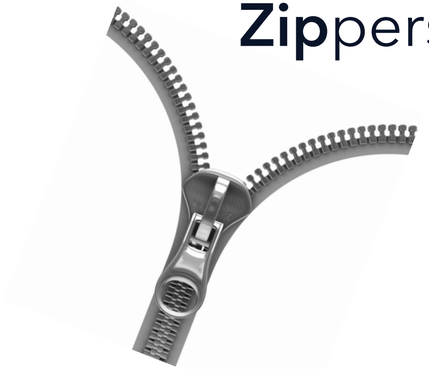
Do we have the right tools?

We already have adders

… and multi-pliers

We just need to add

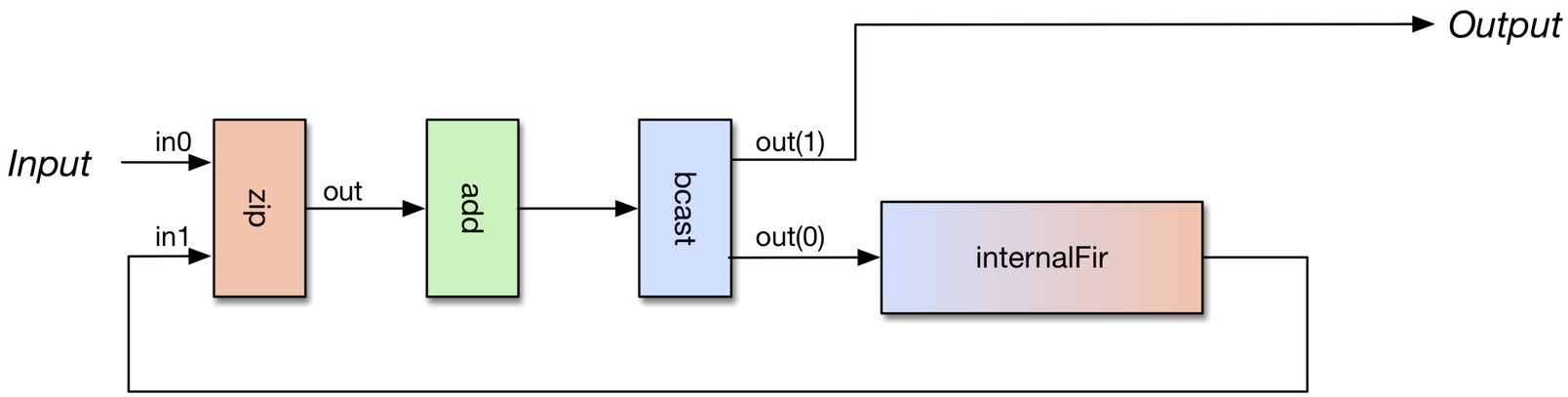
Zippers



… and **Broadcast**ers

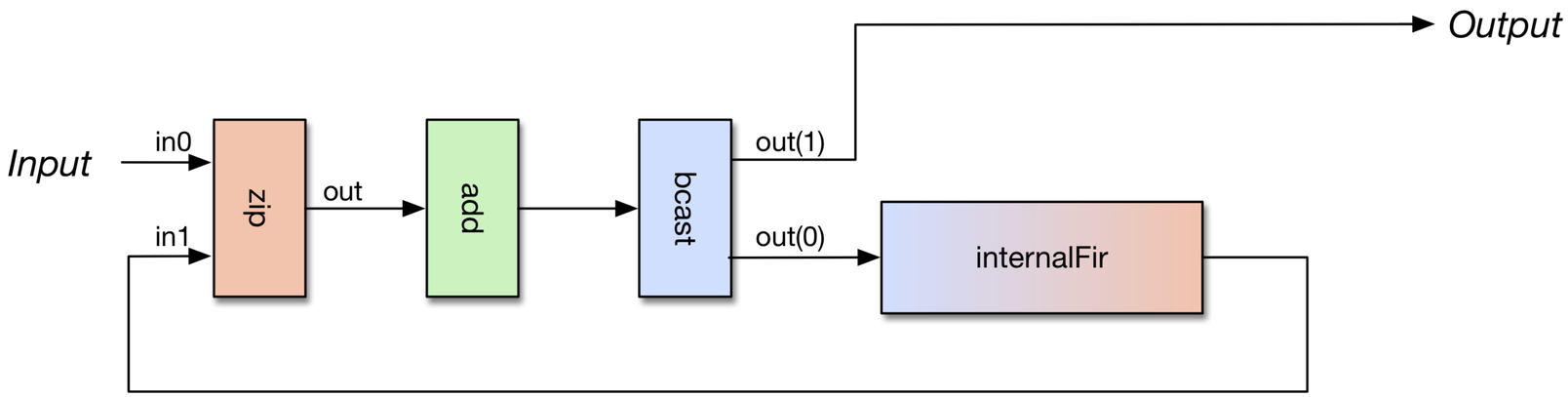
An Akka Streams based IIR Filter

- Translated to an Akka Streams Implementation



An Akka Streams based IIR Filter

- Translated to an Akka Streams Implementation

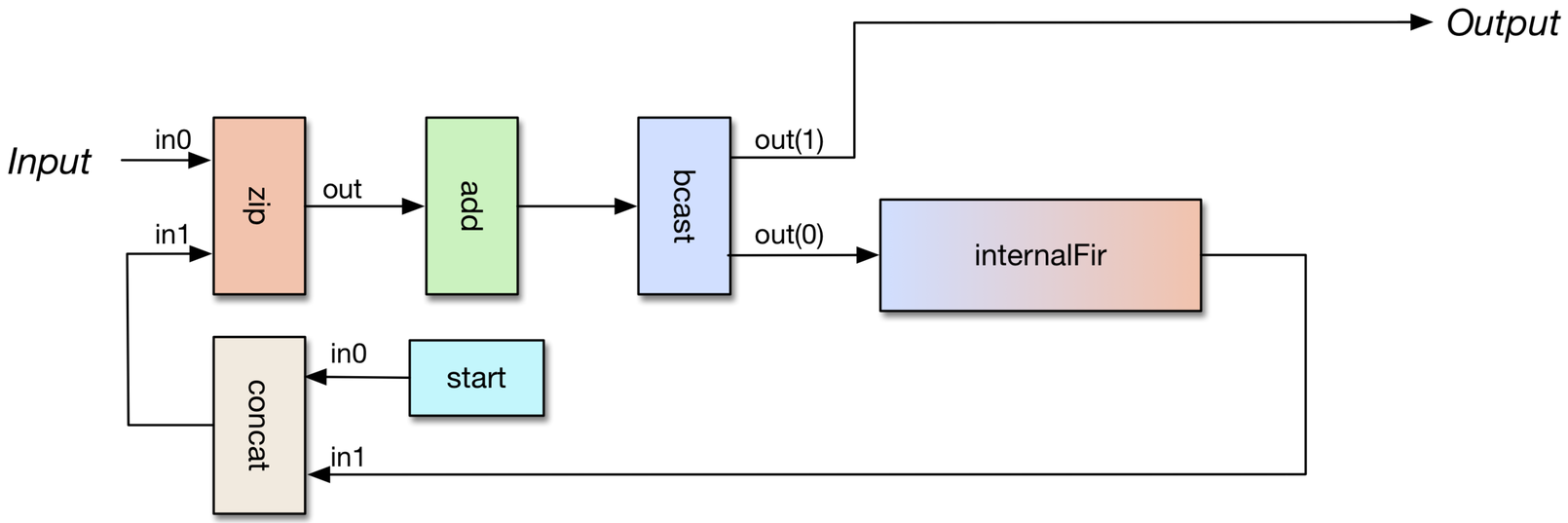


This implementation doesn't produce *any* output…

Can you spot the problem?

An Akka Streams based IIR Filter

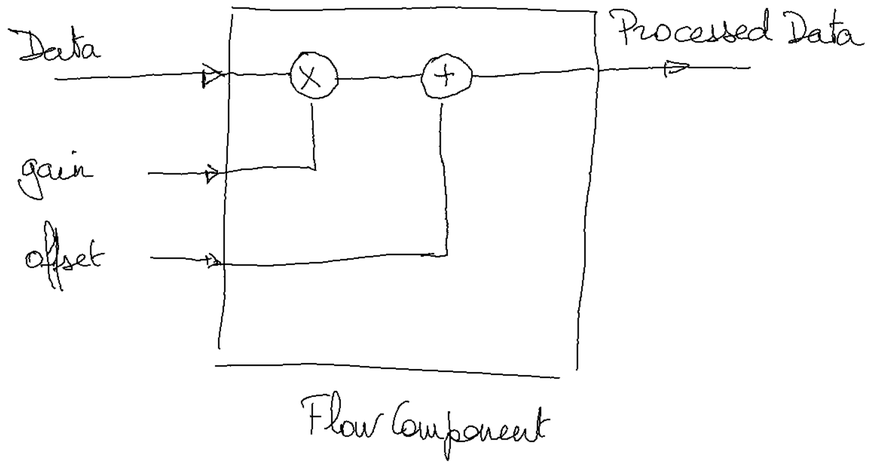
- Solving the catch-22 situation



Adapting stream rate between different streams

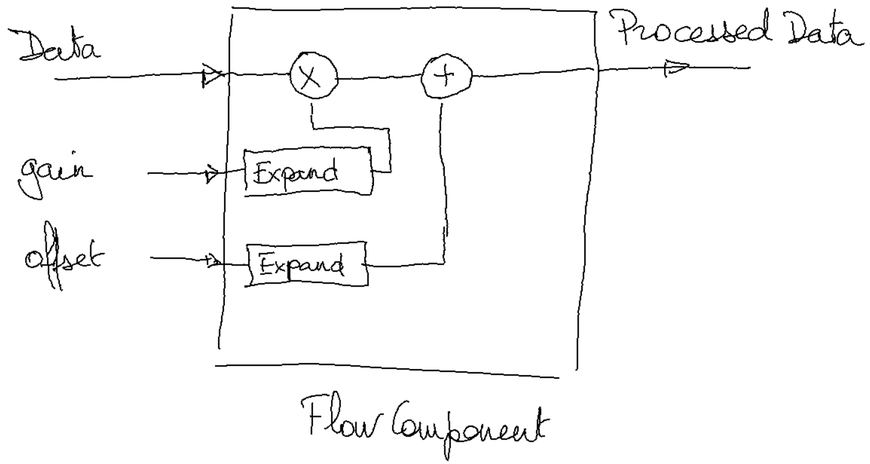
Adapting stream rate - use case

- Imagine two streams:
  - Stream 1: data coming from some source
  - Stream 2: parameters that determine how the data is processed
- The rate at which parameters change will in general be much lower than the data streaming rate



Adapting stream rate - use case

- Imagine two streams:
  - Stream 1: data coming from some source
  - Stream 2: parameters that determine how the data is processed
- The rate at which parameters change will in general be much lower than the data streaming rate



- One solution is to adapt the streaming rate of the slower stream to match that of the faster one

Wrap-up

Things we didn’t cover

- Restartable Source with exponential backoff
- killswitches
- Alpakka connectors
- StreamRefs
- Akka HTTP
  - uses Akka Streams under the hood
  - exposes an Akka Streams API

The End