Akka Streams

Type-safe computation pipelines for streaming data

Andrea Zito
@nivox

Who are you?

Andrea Zito
@nivox

Senior Software Engineer @ www.thinkin.io

- Solving problems on streaming data
- Akka contributor

Reactive Streams

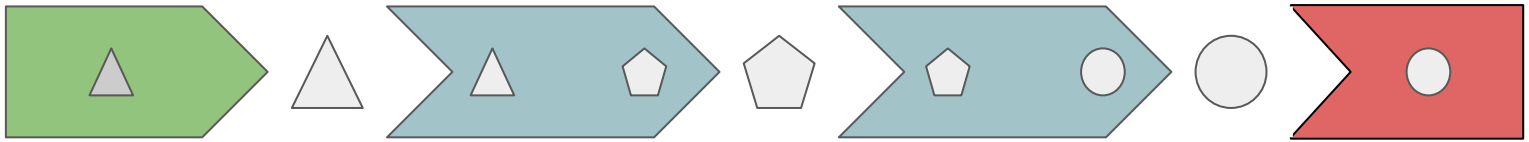
Reactive Streams is an initiative to provide a standard for asynchronous stream processing with non-blocking back pressure.

- Asynchronous:
  enable parallel use of computational resources
- Back pressure:
  allows use of bounded queues between async boundaries
- Non-blocking:
  if blocking were used, asynchronicity would suffer

Multiple (interconnectable) implementations:

Anatomy of a Stream

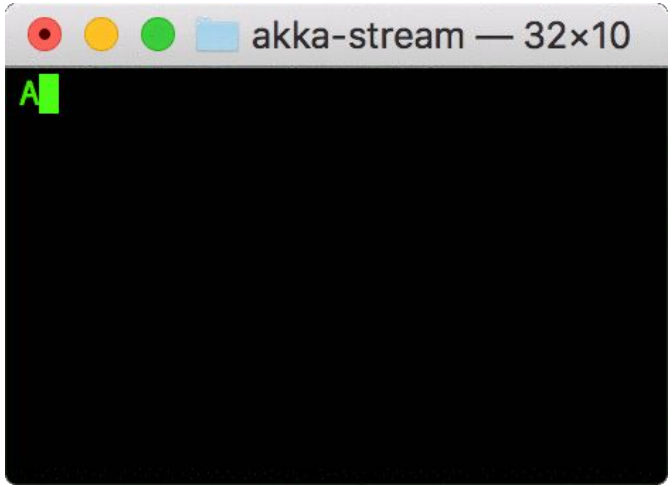
1. **Source:**
   generates elements and emit them into the stream
2. **Flow:**
   accepts elements in input and emit them transformed
3. **Sink:**
   consumes incoming elements



Once the stream is fully connected you get a *runnable graph* which represent the ***blueprint*** of the stream and can be run (or ***materialized***) as many times as needed.

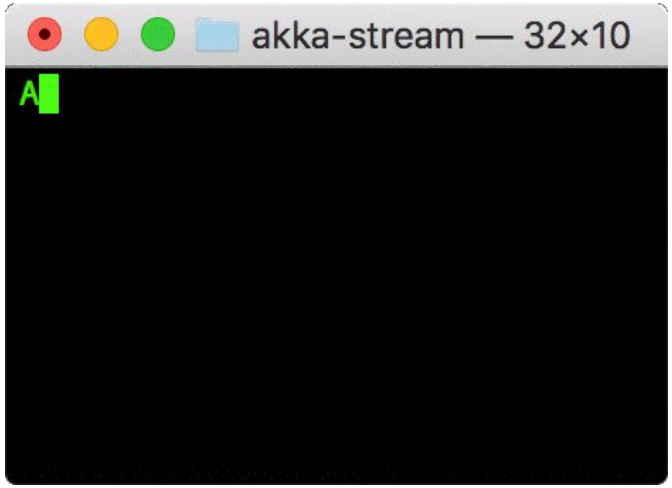
Talk is cheap. Show me the code

```
1   implicit val system: ActorSystem = ActorSystem("akka-stream-demo")
2
3   val source = Source.repeat("Akka stream for the win!")
4
5   val flow = Flow[String].zipWithIndex
6     .map { case (s, idx) => " " * idx.toInt + s }
7     .take(3)
8     .mapConcat(_.toList :+ '\n')
9     .throttle(1, 50.millis, 1, ThrottleMode.shaping)
10
11  val sink = Sink.foreach(print)
12
13  val blueprint = source.via(flow).toMat(sink)(Keep.right)
14
15  val doneF = blueprint.run()
```



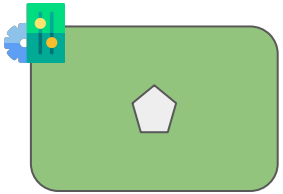
Talk is cheap. Show me the code

```scala
implicit val system: ActorSystem = ActorSystem("akka-stream-demo")

val source: Source[String, NotUsed] =
  Source.repeat("Akka stream for the win!")

val flow: Flow[String, Char, NotUsed] = Flow[String].zipWithIndex
  .map { case (s, idx) => " " * idx.toInt + s }
  .take(3)
  .mapConcat(_.toList :+ '\n')
  .throttle(1, 50.millis, 1, ThrottleMode.shaping)

val sink: Sink[Char, Future[Done]] = Sink.foreach(print)

val blueprint: RunnableGraph[Future[Done]] =
  source.via(flow).toMat(sink)(Keep.right)

val doneF: Future[Done] = blueprint.run()
```

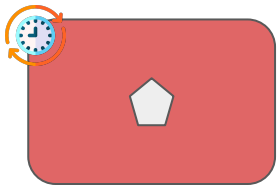


The mystical materialized value

Source[ , Control]

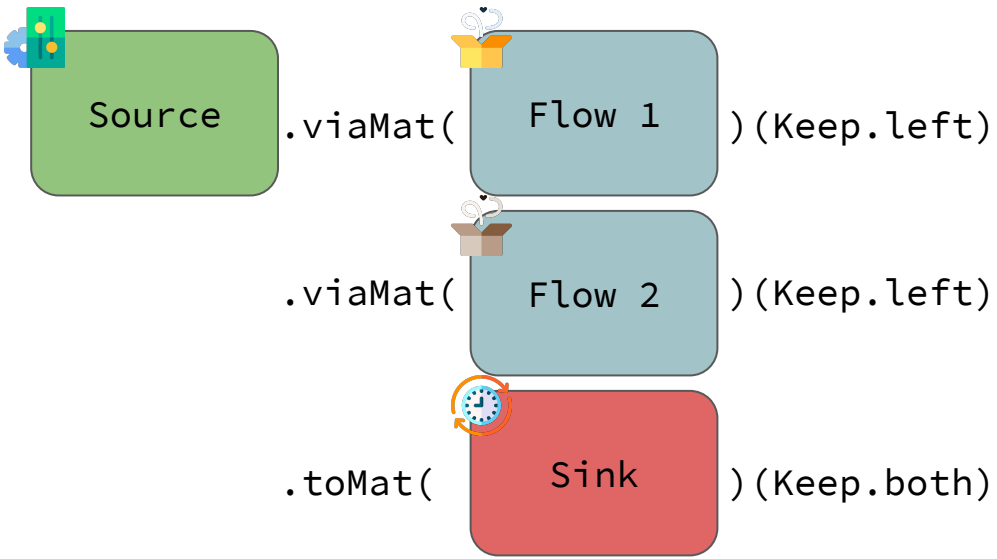


Sink[ , Future[Done]]



- All stream stages have a value that they generate during the materialization phase
- When connecting 2 stages we need to specify how to combine these values
- The materialized value of the stream is the result of the chained application of the combiner functions

The mystical materialized value



Keep.left(,) =>

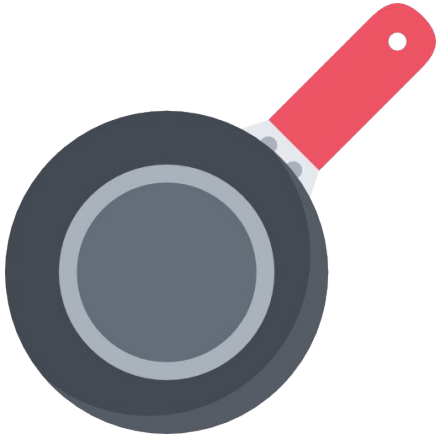
Keep.left(,) =>

Keep.both(,) => (,)

Materialized value: (,)

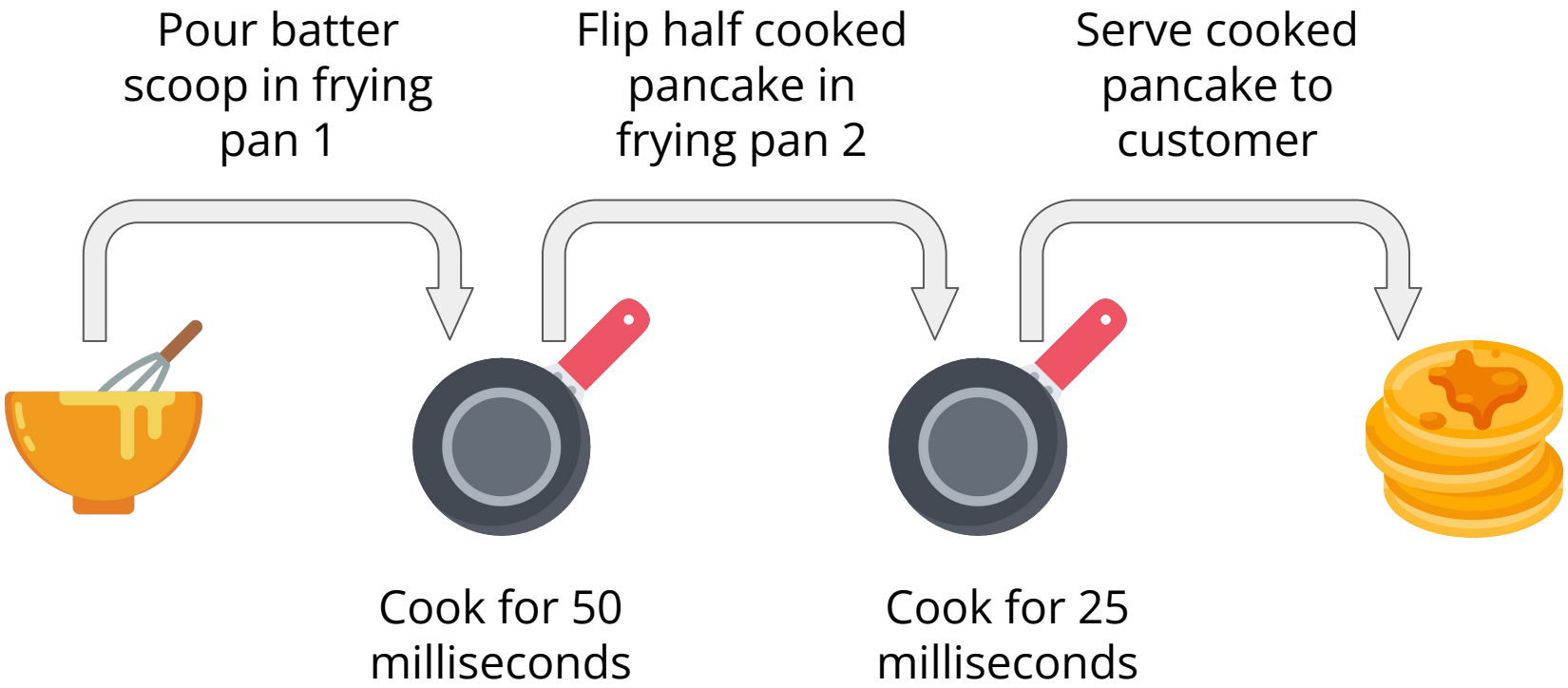
Let’s talk pancakes

- We’re assigned to the express pancake kiosk
- Suddenly a bunch of customers arrive at the same time and they are ***hungry***!

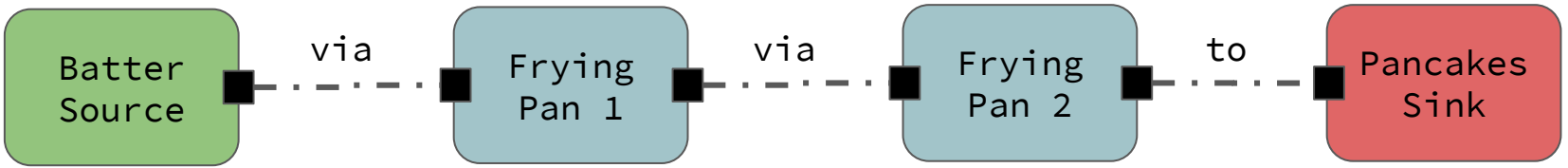


- Cooking one pancake at a time takes too much time…
- Fortunately we have 2 frying pan we can use to speed the process up!

Let’s talk pancakes: pipelining



Let’s talk pancakes: pipelining



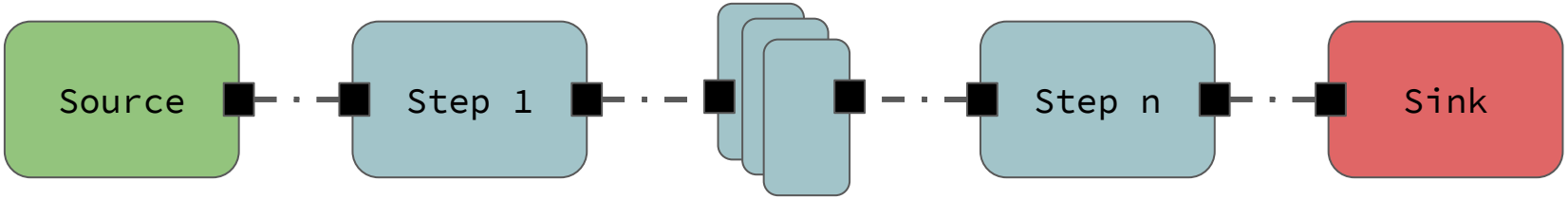
We can express the logical flow by simply connecting the various stages with simple operators!

Let’s talk pancakes: pipelining

```scala
val fryingPan1: Flow[BatterScoop, HalfCookedPancake, NotUsed] = {
  Flow[BatterScoop].mapAsync(1) { scoop =>
    println(s"Cooking batter scoop [${scoop.id}] on frying pan 1")
    scoop.cookFor(50.millis)
  }
}

val fryingPan2: Flow[HalfCookedPancake, CookedPancake, NotUsed] = {
  Flow[HalfCookedPancake].mapAsync(1) { hcPancake =>
    println(s"Cooking pancake [${hcPancake.id}] on frying pan 2")
    hcPancake.cookFor(25.millis)
  }
}

val doneF = Source(1.to(10))
  .map(id => BatterScoop(id))
  .via(fryingPan1.async)
  .via(fryingPan2.async)
  .runWith(Sink.foreach { pancake =>
    println(s"Pancake [${pancake.id}] is done!")
  })
```

```
Cooking batter scoop [1] on frying pan 1
Cooking batter scoop [2] on frying pan 1
Cooking pancake [1] on frying pan 2
Pancake [1] is done!
Cooking batter scoop [3] on frying pan 1
Cooking pancake [2] on frying pan 2
Pancake [2] is done!
Cooking batter scoop [4] on frying pan 1
Cooking pancake [3] on frying pan 2
Pancake [3] is done!
Cooking batter scoop [5] on frying pan 1
Cooking pancake [4] on frying pan 2
Pancake [4] is done!
Cooking batter scoop [6] on frying pan 1
Cooking pancake [5] on frying pan 2
Pancake [5] is done!
...
```

Let’s talk pancakes: pipelining



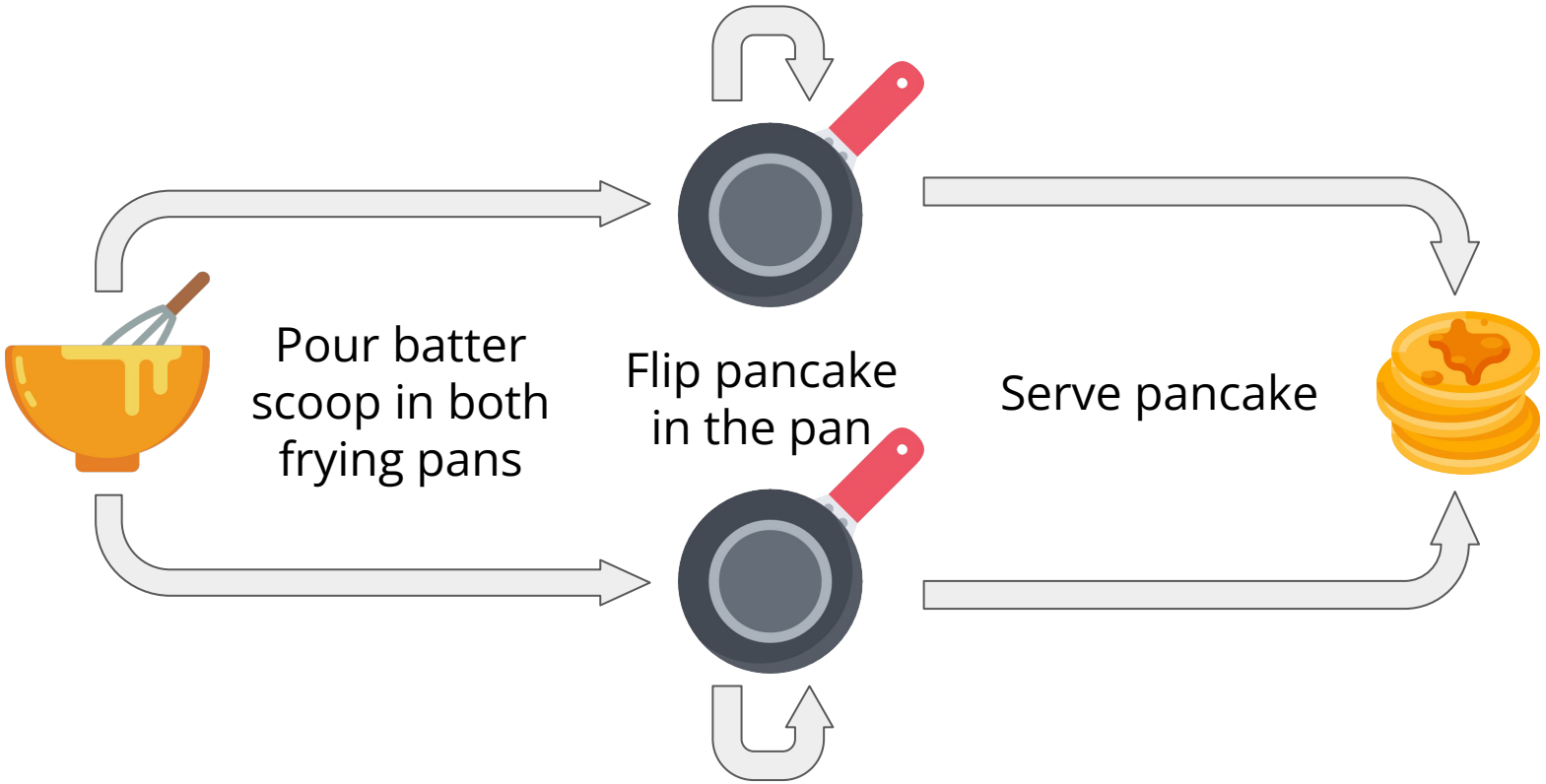
Pros:

- Trivial to implement
- Applicable to almost any process (also those not parallelizable)

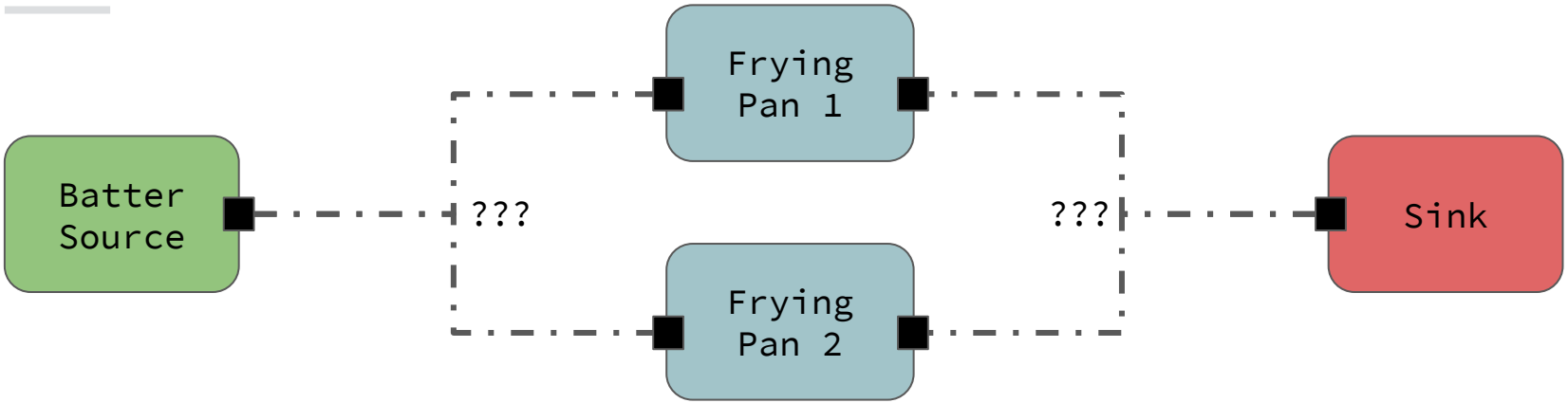
Cons:

- Limited to the number of step of the actual process
- all the steps need to take roughly the same time otherwise some resources will be wasted waiting for data to process

Let’s talk pancakes: parallelism



Let’s talk pancakes: parallelism



We are mostly there but we seem to be missing some pieces of the puzzle that allow us to connect the various stages.

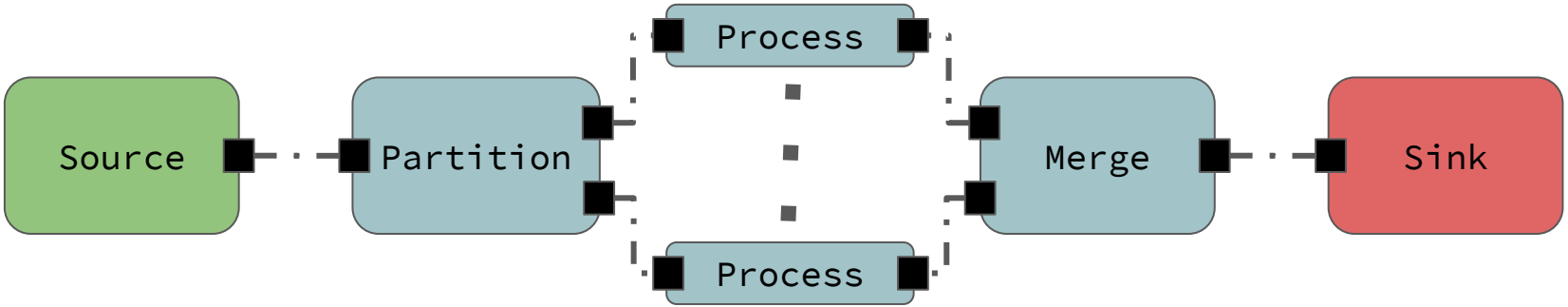
But fear not! When the built-in operators aren’t enough we have a richer APIs we can tap into: GraphDSL.

Let’s talk pancakes: parallelism

```scala
def fryingPanFlow(fryingPanId: Int)
  : Flow[BatterScoop, CookedPancake, NotUsed] = ???

val cookFlow: Flow[BatterScoop, CookedPancake, NotUsed] = {
  Flow.fromGraph(GraphDSL.create() { implicit builder =>
    import GraphDSL.Implicits._

    val dispatchBatter = builder.add(Balance[BatterScoop](2))
    val retrievePancakes = builder.add(Merge[CookedPancake](2))

    dispatchBatter.outlets.zip(retrievePancakes.inlets)
    .zipWithIndex.map {
      case ((batterOut, pancakeIn), idx) =>
        batterOut ~> fryingPanFlow(idx).async ~> pancakeIn
    }

    FlowShape(dispatchBatter.in, retrievePancakes.out)
  })
}

val doneF = Source(1.to(10)).map(id => BatterScoop(id))
  .via(cookFlow).runWith(Sink.foreach { pancake =>
    println(s"Pancake [${pancake.id}] is done!")
  })
```

```
Cooking batter [1] on frying pan [1]
Cooking batter [2] on frying pan [2]
Cooking pancake [1] on frying pan [1]
Cooking pancake [2] on frying pan [2]
Cooking batter [4] on frying pan [2]
Cooking batter [3] on frying pan [1]
Pancake [2] is done!
Pancake [1] is done!
Cooking pancake [4] on frying pan [2]
Cooking pancake [3] on frying pan [1]
Cooking batter [6] on frying pan [2]
Cooking batter [5] on frying pan [1]
Pancake [3] is done!
Pancake [4] is done!
Cooking pancake [5] on frying pan [1]
Cooking pancake [6] on frying pan [2]
...
```

Let’s talk pancakes: parallelism



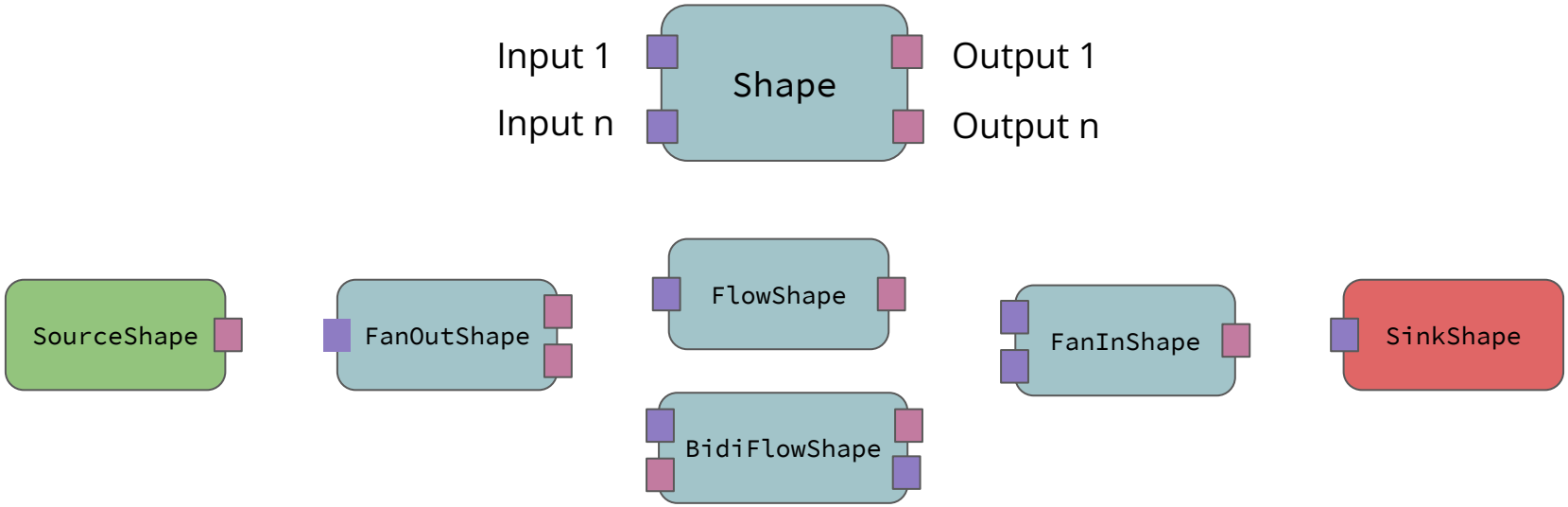
Pros:

- Easy to scale
- Optimal usage of resources not tied to duration of sub-tasks

Cons:

- Not all processes are parallelizable
- Order of data is not maintained

So I lied! It's all about shapes...



Shapes are the primitive that enable the composability of Akka Stream and allows you to extend it with the same power and first class support as the built-in operators.

… and you can create your own

```
1 class CustomStage[T] extends GraphStage[FlowShape[T, (T, Int)]] {
2   val in: Inlet[T] = Inlet("CustomStage.in")
3   val out: Outlet[(T, Int)] = Outlet("CustomStage.out")
4
5   override def shape: FlowShape[T, (T, Int)] = FlowShape(in, out)
6
7   override def createLogic(inheritedAttributes: Attributes): GraphStageLogic = new GraphStageLogic(shape) {
8     var counter: Int = 0
9
10    setHandler(in, new InHandler {
11      override def onPush(): Unit = {
12          val input = grab(in)
13          val output = (input, counter)
14          counter += 1
15          push(out, output)
16      }
17    })
18
19    setHandler(out, new OutHandler {
20      override def onPull(): Unit = pull(in)
21    })
22  }
23 }
```

That was a lot. Let’s recap

- **Source/Flow/Sink operators**
  Highest level (and most used) API: simple and safe

- **GraphDSL**
  Connect shapes wrapping them into another (usually simpler) shape

- **GraphStage**
  Lowest level API, used to create complex stages with custom logic.
  Offers the most power but exposes you to (some) of the reactive stream protocol handling.

Connect all the things

The Alpakka project offers connectors (i.e. sources and sinks) for a plethora of different technologies. It's the Apache Camel of streams!

... and more

Takeaways

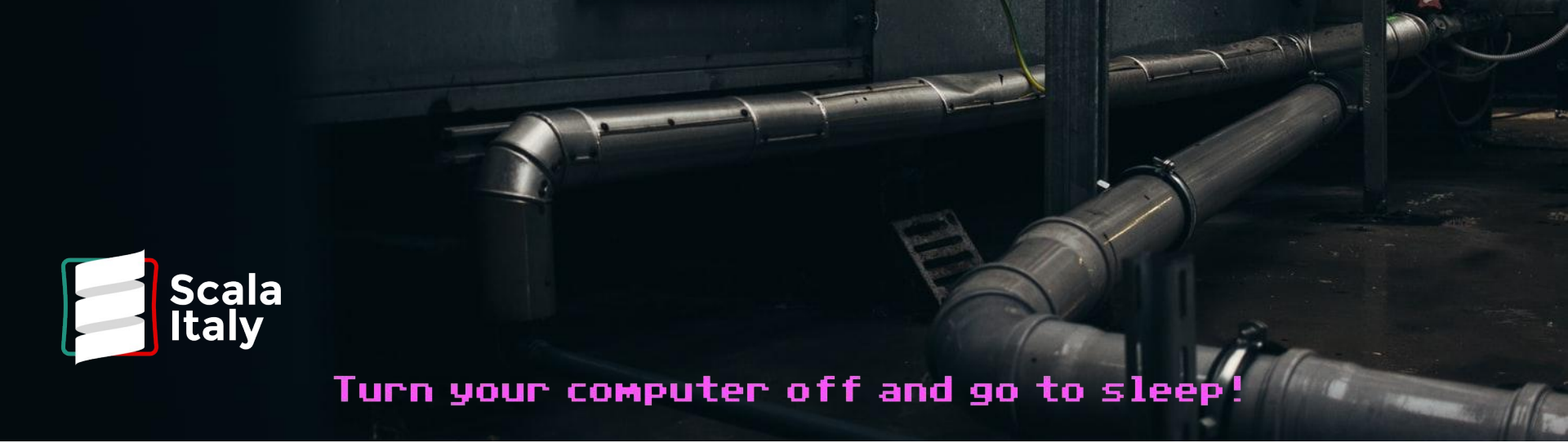
- natural fit to model streaming data problems
- stages are modular and composable
- Backpressure is key!
- type safe: let the compiler work for you!
- battery included
- simple (as in simple concepts)
- extremely extensible

Is this it? Well... no

- Remote Akka Streams
- Interop with Akka Actors
- Event sourcing with Akka Persistence
- High Availability with Akka Cluster

Pointers:

- documentation: https://doc.akka.io/docs/akka/current/stream/index.html
- Kuhn, Roland. Reactive Design Patterns. Manning, 2017



Akka Streams

Type-safe computation pipelines for streaming data

Andrea Zito
@nivox

Example 2: order preserving

```
def orderPreserving[IN, OUT, CTX, M](
    flow: Flow[(IN, OrderPreservingCtx[CTX]), (OUT, OrderPreservingCtx[CTX]), M]
): Flow[(IN, CTX), (OUT, CTX), M]
```