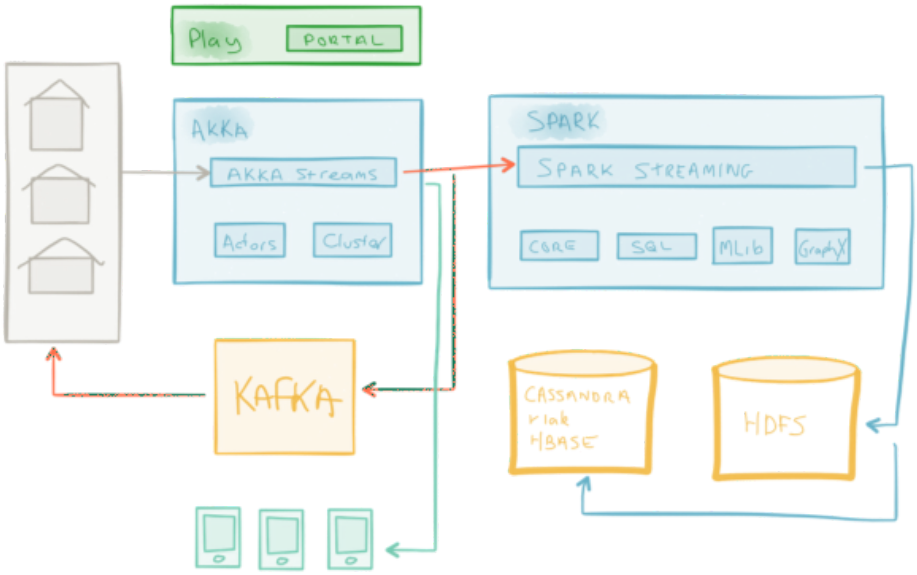
Diving into Akka Streams

Streaming is the ultimate game changer for data-intensive systems. In a world where every second counts, the batch jobs of yesteryear with overnight processing times measured in hours are nothing but an expense. Per-event streaming puts the emphasis on recent data that can be actioned **right now**.

We've lost our way. We've become enamoured with the *size* of data rather than the *value* of data. *Big* data. *Huge* data. *Massive* data. *Too much data!* We've become hoarders, hiding away data that we don't need in long forgotten dusty corners to process that data with some magical algorithm at an unknown time in the future.

Our big, huge, massive data will make us rich one day! All we need to do is lease petabytes of storage, choose an enterprise big data platform, staff up a big data team, hire security experts to ensure all of our data is secure (so we conform to the many industry standards around sensitive data such as HIPAA), and… you get the idea. Or we can come to the collective realization that data is inventory. If data is not generating revenue today it's an *expense*, not an asset.



A fast data architecture with the Lightbend Reactive Platform, Spark, and Kafka.

Fast data is what matters most. Let the others horde. We want to build real-time systems. Data that is recent is relevant. Data that is stale is history. Historical data is good for generating insights that may influence long-term strategic thinking—or perhaps cranking out a few infographics—but when it comes to the tactical behaviour of running a business, *immediacy* is the primary concern. We don't want to restock inventory or decline a credit card transaction based on what happened yesterday (or last month or last year), we want to make critical decisions based on what's happening *right now*. Yes, we must use a certain amount of historical data for business context—for instance, we may decline a credit card application based on a previous bankruptcy. But the ultimate decision we make in any given moment must primarily be influenced by events happening *now.*

Thinking in events enables us to rethink our systems in a purely reactive way —ingesting *events,* issuing *commands* which emit new *events*. We can feed those events back into our system, or emit them for further downstream processing. Forward thinking organizations are using event-driven architectures like this to transform batch-mode systems into real-time systems, reducing latencies from hours to seconds.

While the rest of the article expects some familiarity with Akka, being an Akka expert isn't required to explore Akka Streams. In fact, the Akka Streams API provides a higher level of abstraction than Akka. It makes Akka

Streams well suited to build event-driven systems using a very simple dialect.

Introduction

Akka Streams is a toolkit for *per-event* processing of streams. It's an implementation of the Reactive Streams specification, which promises interoperability between a variety of compliant streaming libraries.

> *Akka Streams provides a way to express and run a chain of asynchronous processing steps acting on a sequence of elements.*

If you're new to the world of stream processing, I recommend reading the first part of this series, A Journey into Reactive Streams, before continuing. The rest of this article assumes some familiarity with the content outlined in that post, as well as a high-level understanding of Akka.

The Basics

Let's start by describing a simple linear data flow. A linear flow is a chain of processing steps each with only one input and one output, connected to a data source and a data sink. In this example we'll use Akka Streams to ingest a CSV file which contains records of all flight data in the US for a single year, process the flight data, and emit an ordered list of average flight delays per carrier in a single year.

We start with an *Outlet*, that's our *Source*. A source of data can be a queue (such as Kafka), a file, a database, an API endpoint, and so on. In this example, we're reading in a CSV file and emitting each line of the file as a String type.

We connect the Outlet to a *FlowShape*—a processing step that has exactly one input and one output. This FlowShape ingests *String* types and emits *FlightEvent* types—a custom data type that represents a row of data in the CSV file. The FlowShape uses a higher-order function called *csvToFlightEvent* to convert each String to a FlightEvent.

Let's explore the source code below. Akka Streams supports both Scala and Java—our examples will be in Scala.

```scala
import akka.actor.ActorSystem
import akka.stream._
import akka.stream.scaladsl._

// implicit actor system
implicit val system = ActorSystem("Sys")

// implicit actor materializer
implicit val materializer = ActorMaterializer()

def main(args: Array[String]): Unit = {

  val g: RunnableGraph[_] = RunnableGraph.fromGraph(GraphDS
    implicit builder =>

      // Source
      val A: Outlet[String] = builder.add(Source.fromIterat
      val B: FlowShape[String, FlightEvent] = builder.add(c
      val C: Inlet[Any] = builder.add(Sink.ignore).in

      import GraphDSL.Implicits._ // allows us to build our

      // Graph
      A ~> B ~> C

      ClosedShape // defines this as a "closed" graph, not
  })

  g.run() // materializes and executes the blueprint

}
```

LinearFlow.scala hosted with by GitHub view raw

First, we define the overall blueprint, called a graph, using the *GraphDSL,* and assign it to a value, *g*.

Blueprints can be created, assigned to values, shared, and composed to make larger graphs.

Next we need both an *ActorSystem* and an *ActorMaterializer* in scope to *materialize* the graph. Materialization is a step that provisions Akka Actors under the hood to do the actual work for us. We kick off materialization by calling *run()* on the graph, which provisions actors, attaches to the source of data, and begins the data flow. At runtime the materializer walks the graph, looks at how it's defined, and creates actors for all of the steps. If it sees opportunity for optimization, it can put multiple operations into one actor or create multiple actors for parallelization. The key concept is that the hardest work is done for you—all of this optimization would need to be done manually if we dropped down from Akka Streams to Akka Actors.

Digging deeper into the blueprint, we notice a third step, an *Inlet*. This is a generalized type that defines our *Sink*. There are a number of different Sink types which we will explore in more detail shortly. A runnable graph must have all flows connected to an inlet and an outlet, so for this initial example we'll use an *ignore sink* to wire everything together but not actually do anything with the results of our first flow step. The ignore sink will act as a black hole until we decide what to do with the events it consumes.

Let's pause. If we created a system like this before Akka Streams, we would have needed to create an actor for each step in the flow and manually wired them together for message passing. That adds a significant amount of cognitive overhead for the relatively simple task of localized data flow processing.

You'll perhaps also notice another significant difference between Streams and Actors—*types*. We can specify the types of each step in our flow rather than relying on pattern matching as we would do with Akka Actors.

Graph shapes

There are four main building blocks of an Akka Streams application:

- **Source**—A processing stage with exactly one output
- **Sink**— A processing stage with exactly one input
- **Flow**—A processing stage which has exactly one input and output
- **RunnableGraph**—A Flow that has both ends attached to a Source and Sink

A basic linear flow.

That's a good starting point, but without anything else we could only create *linear* flows—flows made up entirely of steps with single inputs and single outputs.

Linear flows are nice… but a bit boring. At a certain point we'll want to split apart streams using *fan-out* functions and join them back together using *fan-in* functions. We'll explore a few of the more useful fan-in and fan-out functions below.

Fan out

Fan out operations give us the ability to split a stream into substreams.

Broadcast ingests events from one input and emits duplicated events across more than one output stream. An example usage for a broadcast would be to create a side-channel, with events from one stream being persisted to a data store, while the duplicated events are sent to another graph for further computation.

Balance signals one of its output ports for any given signal, but not both. According to the documentation, “each upstream element is emitted to the first available downstream consumer”. This means events are not distributed in a deterministic fashion, with one output being signalled and then the other, but rather whichever downstream subscriber happens to be available. This is a very useful fan-out function for high-throughput streams, enabling graphs to be split apart and multiple instances of downstream subscribers replicated to handle the volume.

Fan in

Fan in operations give us the ability to join multiple streams into a single output stream.

Merge picks a signal randomly from *one* of *many* inputs and emits the event downstream. Variants are available such as *MergePreferred*, which favours a particular inlet if events are available to consume on multiple incoming streams, and *MergeSorted*, which assumes the incoming streams are sorted and therefore ensures the outbound stream is also sorted.

Concat is similar to Merge, except it works with exactly *two* inputs rather than *many* inputs.

Materialization

The separation of blueprints from the underlying runtime resources required to execute all of the steps is called *materialization.* Materialization enables developers to separate the *what* from the *how*; developers focus on the higher-level task of defining blueprints, while the materializer provisions actors under the hood to turn those blueprints into runnable code.

The materializer can be further configured, with an error handling strategy for example.

```scala
val decider: Supervision.Decider = exc => exc match {
  case _: ArithmeticException => Supervision.Resume
  case _                      => Supervision.Stop
}
// ActorFlowMaterializer takes the list of transformations
// and materializes them in the form of org.reactivestreams
implicit val mat = ActorFlowMaterializer(
  ActorFlowMaterializerSettings(system).withSupervisionStra
val source = Source(0 to 5).map(100 / _)
val result = source.runWith(Sink.fold(0)(_ + _))
```

Decider.scala hosted with by GitHub view raw

In the example above, we've defined a *decider* that ignores arithmetic exceptions but stops if any other exceptions occur. This is because division-by-zero is expected but doesn't prevent successful completion. In the example above the result will be a *Result* will be a *Future* completed with *Success(228)*.

Considerations

Akka enables *asynchrony*—different steps within the flow can be parallelized because they aren't tied to a single thread. Using Akka Actors enables local message passing, remote message passing, and clustering of actors, providing a powerful arsenal for distributed computing. Akka Streams provides a subset of this flexibility, limiting distribution to threads and thread pools. In the future it's entirely possible that Akka Streams will support distribution of steps remotely over the network, but in the meantime multiple Akka Stream applications can be connected together manually and chained for more complex, distributed stream processing. The advantage of using Streams over Actors is how much complexity Streams removes for applications that don't require advanced routing and distribution over network connections.

Conclusion

Our solution has come together. We've chained together processing steps using the *GraphDSL* that removes the complexity of manually passing messages between Akka Actors. Akka Streams also handles much of the complexity of timeouts, failure, backpressure, and so forth, freeing us to think about the bigger picture of how events flow through our systems.

We perform the following computation on the CSV file:

- Convert each line from a String into a FlightEvent
- Filter out non-delayed flights from the stream
- Broadcast the stream across two distinct flows—one to capture raw events, another to capture aggregate flight delay data
- Emit a substream per airline carrier, accumulating the total number of delayed flights and the minutes of each delay for each airline, then merging the streams together with the totals
- Print flight delay information to the console

```scala
// @formatter:off
val g = RunnableGraph.fromGraph(GraphDSL.create() {
  implicit builder =>
    import GraphDSL.Implicits._

    // Source
    val A: Outlet[String] = builder.add(Source.fromIterator

    // Flows
    val B: FlowShape[String, FlightEvent] = builder.add(csv
    val C: FlowShape[FlightEvent, FlightDelayRecord] = buil
    val D: UniformFanOutShape[FlightDelayRecord, FlightDela
    val F: FlowShape[FlightDelayRecord, (String, Int, Int)]

    // Sinks
    val E: Inlet[Any] = builder.add(Sink.ignore).in
    val G: Inlet[Any] = builder.add(Sink.foreach(averageSin

    // Graph
    A ~> B ~> C ~> D
              E <~ D
         G <~ F <~ D

    ClosedShape
})
// @formatter:on

g.run()
```

FlightDelays.scala hosted with by GitHub view raw

We've crunched through a 600MB CSV file in ~1 minute and output meaningful results with ~100 lines of code. This program could be improved to do a number of things, such as stream raw events to a dashboard, emit events to a database (with backpressure!), feed aggregate data into Kafka for processing by Spark—the sky is the limit.

```
Delays for carrier HA: 18 average mins, 18736 delayed fligh
Delays for carrier DL: 27 average mins, 209018 delayed flig
Delays for carrier FL: 31 average mins, 117632 delayed flig
Delays for carrier 9E: 32 average mins, 90601 delayed fligh
Delays for carrier OH: 34 average mins, 96154 delayed fligh
Delays for carrier B6: 42 average mins, 83202 delayed fligh
Delays for carrier EV: 35 average mins, 122751 delayed flig
Delays for carrier AQ: 12 average mins, 1908 delayed flight
Delays for carrier MQ: 35 average mins, 205765 delayed flig
Delays for carrier CO: 34 average mins, 141680 delayed flig
Delays for carrier AS: 27 average mins, 62241 delayed fligh
Delays for carrier YV: 37 average mins, 111004 delayed flig
Delays for carrier AA: 35 average mins, 293277 delayed flig
Delays for carrier NW: 28 average mins, 158797 delayed flig
Delays for carrier OO: 31 average mins, 219367 delayed flig
Delays for carrier WN: 26 average mins, 469518 delayed flig
Delays for carrier US: 28 average mins, 167945 delayed flig
Delays for carrier UA: 38 average mins, 200470 delayed flig
Delays for carrier XE: 36 average mins, 162602 delayed flig
Delays for carrier F9: 21 average mins, 46836 delayed fligh
```

flight-delay-output.txt hosted with by **GitHub** **view raw**

Full source code can be obtained here:
https://github.com/rocketpages/flight_delay_akka_streams

Flight data can be obtained here:
http://stat-computing.org/dataexpo/2009/the-data.html

In a future post we will demonstrate the big-picture architectural possibilities of Akka Streams and use it as glue between inbound data (API endpoints, legacy back-end systems, and so forth) and distributed computation using Spark. At Lightbend we call this architecture *fast data*, which enables a whole new type of creativity when working the volume of data that flows through a modern organization. It elevates data to the heart of the organization, enabling us to build systems that deliver real-time *actionable* insights.

If you're interested in a deeper conceptual overview of fast data platforms, I invite you to read a whitepaper, Fast Data: Big Data Evolved, by Dean Wampler, member of the OCTO team and "Fast Data Wrangler" at Lightbend. And if you're as excited as I am about transforming batch jobs into real-time systems, stay tuned for the next post in this series!

About The Author

Kevin Webber is Enterprise Advocate at Lightbend. He's passionate about helping large organizations transition from heritage architectures to real-time distributed systems that embrace the principles of reactive programming. In his spare time he organizes ReactiveTO and Programming Book Club Toronto. He rarely writes about himself in the third-person, but this is one of those moments.