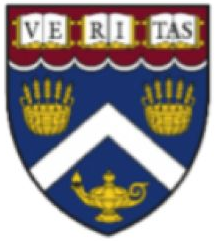
CSCI E-88A Introduction to Functional and Stream Processing for Big Data Systems

Harvard University Extension, Spring 2020

Marina Popova, Edward Sumitra



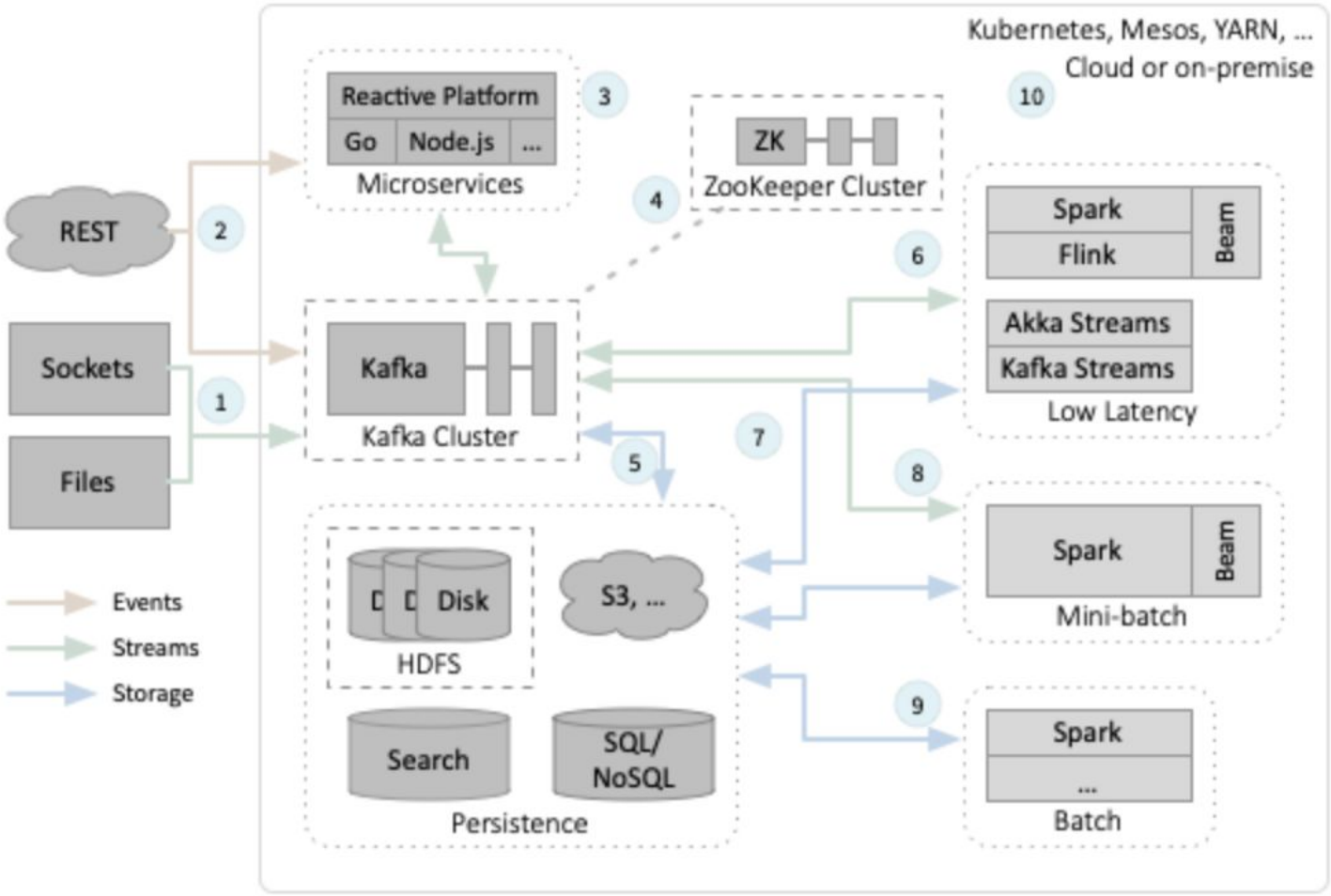
Introduction to Akka Streams

Agenda

- Admin info: Lab
- Streaming Libraries and Frameworks
- Actor Concurrency Model
- Akka Streams Concepts
- Sources, Sinks, Flow
- Demo - Sources, Sinks, Flow
- Runnable Graphs and Materialization
- Building Processing Pipelines
- Demo - Data processing Pipelines

Image credits: All diagrams and images from lightbend.com unless otherwise noted.

Streaming Architecture



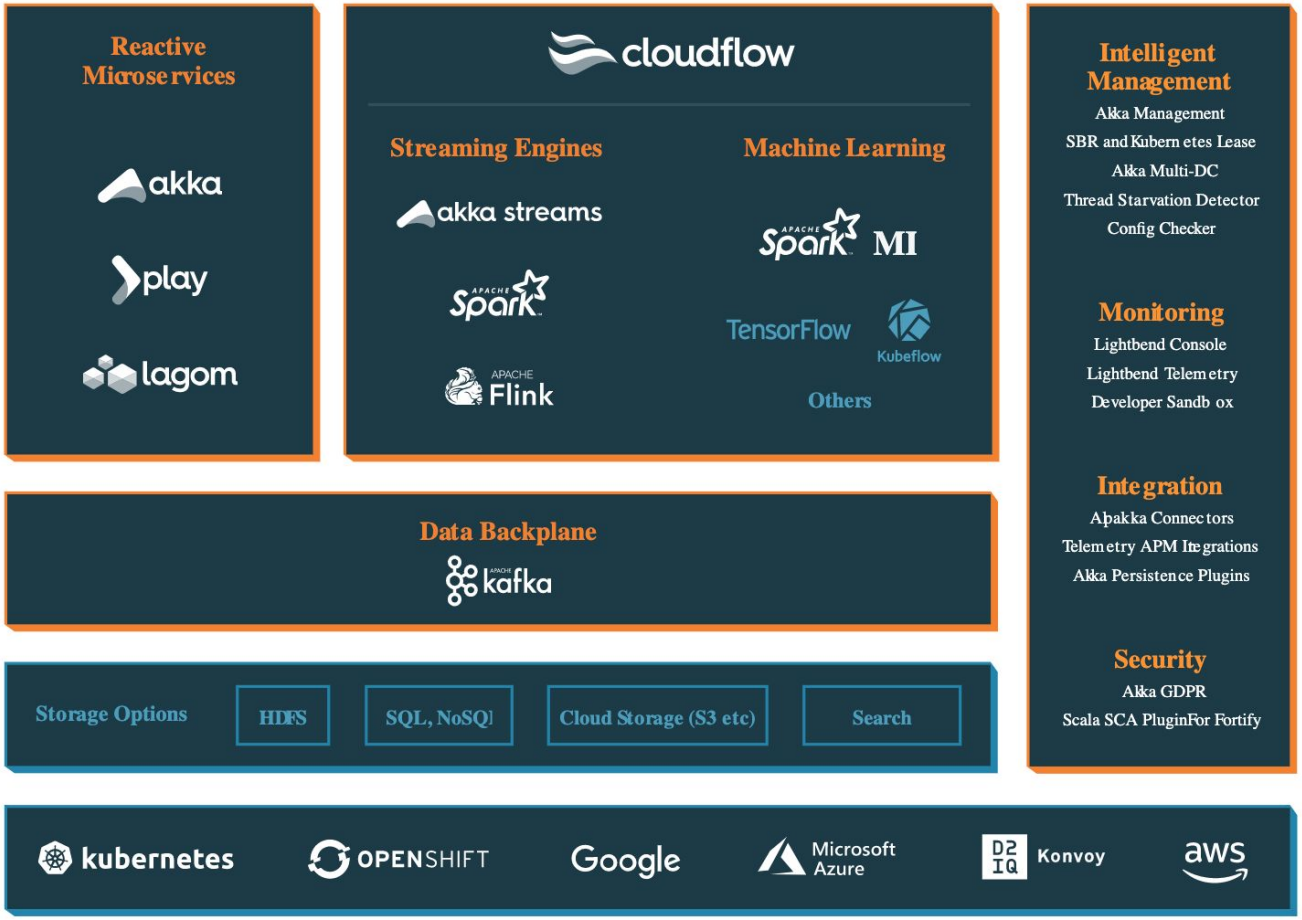
@Marina Popova, Edward Sumitra

Streaming Services and Libraries

Service	**Library**
Spark, Flink, Beam	Akka Streams, Kafka Streams
Large frameworks	Smaller libraries
Submit work to frameworks	Integrate library into processing workflow
Large scale, automatic data partitioning	Smaller scale more flexibility

Akka Streams	**Kafka Streams**
Fine grained control over data flows	Built in support for event windows
Read and write to a wide range of data sources and sinks through Alpakka	Read and write to Kafka Topics
Backpressure aware with reactive streams	Auto scaling with same consumer group

Lightbend Platform



@Marina Popova, Edward Sumitra

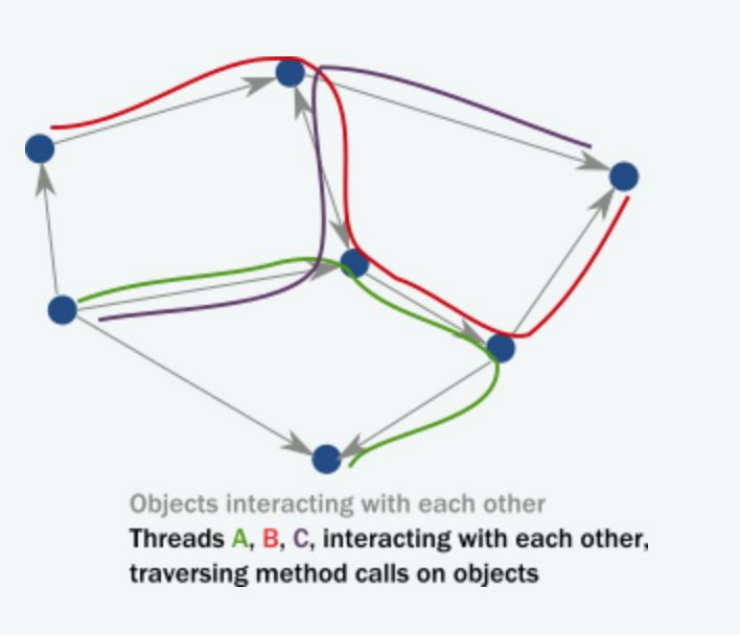
Concurrency Models

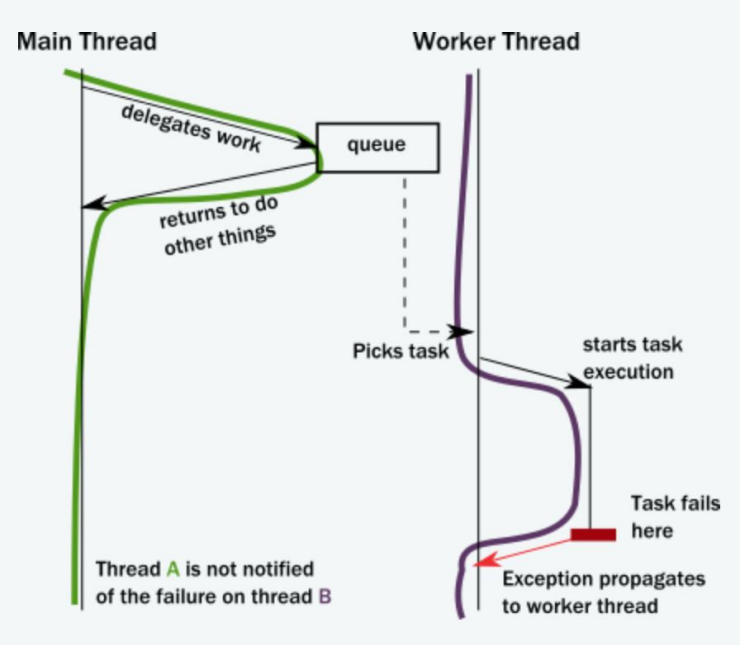
- Concurrency Models
  - Threads
  - Actors
  - Communicating Sequential Processes (CSP)
  - Event-based non-blocking IO
  - Software Transactional memory (STM)

- Actor Concurrency model
  - Popularized in Erlang and Ericsson to build telecom systems
  - Used in akka and all akka based libraries

Limitations of Thread concurrency model

- Encapsulation of models in OOP requires locking when dealing with multiple threads
- Locking limits concurrency
- Failure handling is complex and requires side channels
- Locking only works locally and does not scale across the network



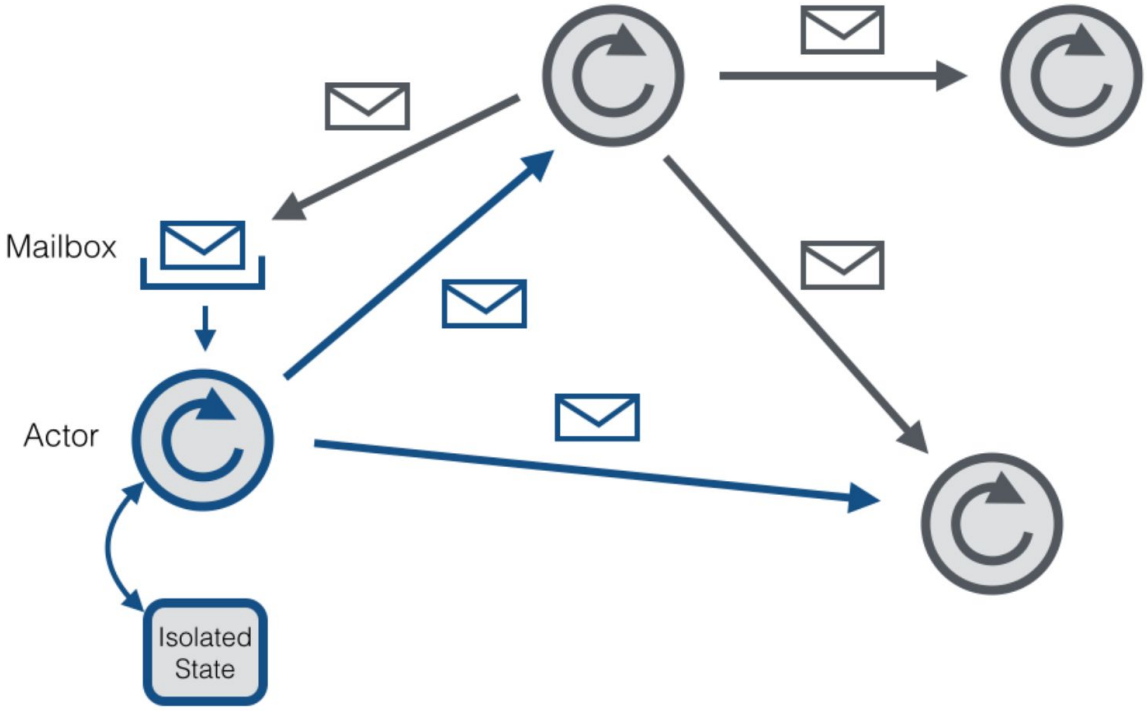


@Marina Popova, Edward Sumitra

Actor Concurrency Model

- Actors can send **asynchronous** messages to other actors
- Actors can create or spawn new actors
- Actors can change local bel
- Actor hierarchy with supervi
- Actors have location Transparency

Ref: [Flink: Akka and Actors]



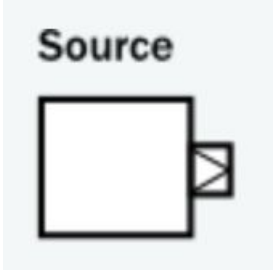
Akka Streams

- Built on top of akka actors
- Provides a flexible and intuitive Streaming API
- Provides bounded resource usage with backpressure
- Interoperates with Reactive Streams initiative in JDK9+
- Reusable and composable building blocks that can used to create complex graphs of streaming pipelines
- Can use Akka Persistence to persist and checkpoint streaming data
- Can use Akka Cluster to provide elasticity and fault-tolerance

Akka Streams Processing Stages

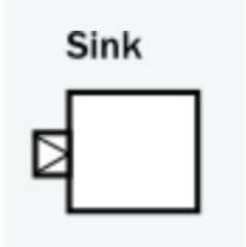
- Source
  - A block with one output

```
final class  Source[+Out, +Mat] extends FlowOpsMat[Out, Mat] with Graph[SourceShape[Out], Mat]
```



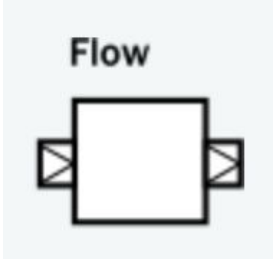
- Sink
  - A block with one input

```
final class  Sink[-In, +Mat] extends Graph[SinkShape[In], Mat]
```



- Flow
  - A block with one input and one output

```
final class  Flow[-In, +Out, +Mat] extends FlowOpsMat[Out, Mat] with Graph[FlowShape[In, Out], Mat]
```



Source - Creation

- Empty and Single value

```
val emp1: Source[Int, NotUsed] = Source.empty

val single1: Source[String, NotUsed] = Source.single("a")
```

- Repeated values

```
val aas: Source[String, NotUsed] = Source.repeat("A")

val countByTwo: Source[Int, NotUsed] = Source.cycle(() => List(1,2).iterator)
```

- From iterables

```
val simpleRange: Source[Int, NotUsed] = Source(1 to 100)

val simpleList: Source[Int, NotUsed] = Source(List(1,2,3,4,5))
```

- From files

```
val csv: Source[ByteString, Future[IOResult]] = FileIO.fromPath(Paths.get("/Users/esumitra/tmp/tsc.log"))

// csv.map(_.utf8String).runWith(Sink.foreach(print(_)))
```

Sources - Operators

- Flow operators
  Map, filter, reduce, drop, take zip, zipWith
  Convenience operators for Flows

- Run methods
  runWith, runFold, runForeach
  Convenience operators for Sinks

```
val sum5b = simpleList.runFold(0)(_ + _)

val run5 = simpleList.runForeach(i => println(s"[$i]"))
```

https://doc.akka.io/api/akka/current/akka/stream/scaladsl/Source.html

Sinks - creation

- For side effects
  foreach, ignore
- To collections
  head, seq,
- To files
  FileIO.toPath

```
val list2 = simpleList.map(_ * 10).runWith(Sink.seq)

val list3Result = simpleList.map(i => ByteString(s"$i")).runWith(FileIO.toPath(Paths.get("list3.txt")))
```

Flow

- A processing stage with one input and one output
- Connects upstream and downstream stages by transforming data elements

- Standalone Flow

```
val upto25: Flow[Int, Int, NotUsed] =
  Flow[Int].takeWhile(_ < 25)
```

- Source + Flow = Source

```
val upto25Source: Source[Int, NotUsed] = simpleRange.via(upto25)
```

- Flow + Sink = Sink

```
val truncatedPrint: Sink[Int, NotUsed] = upto25.to(Sink.seq)
```

Runnable Graphs and Materializing Streams

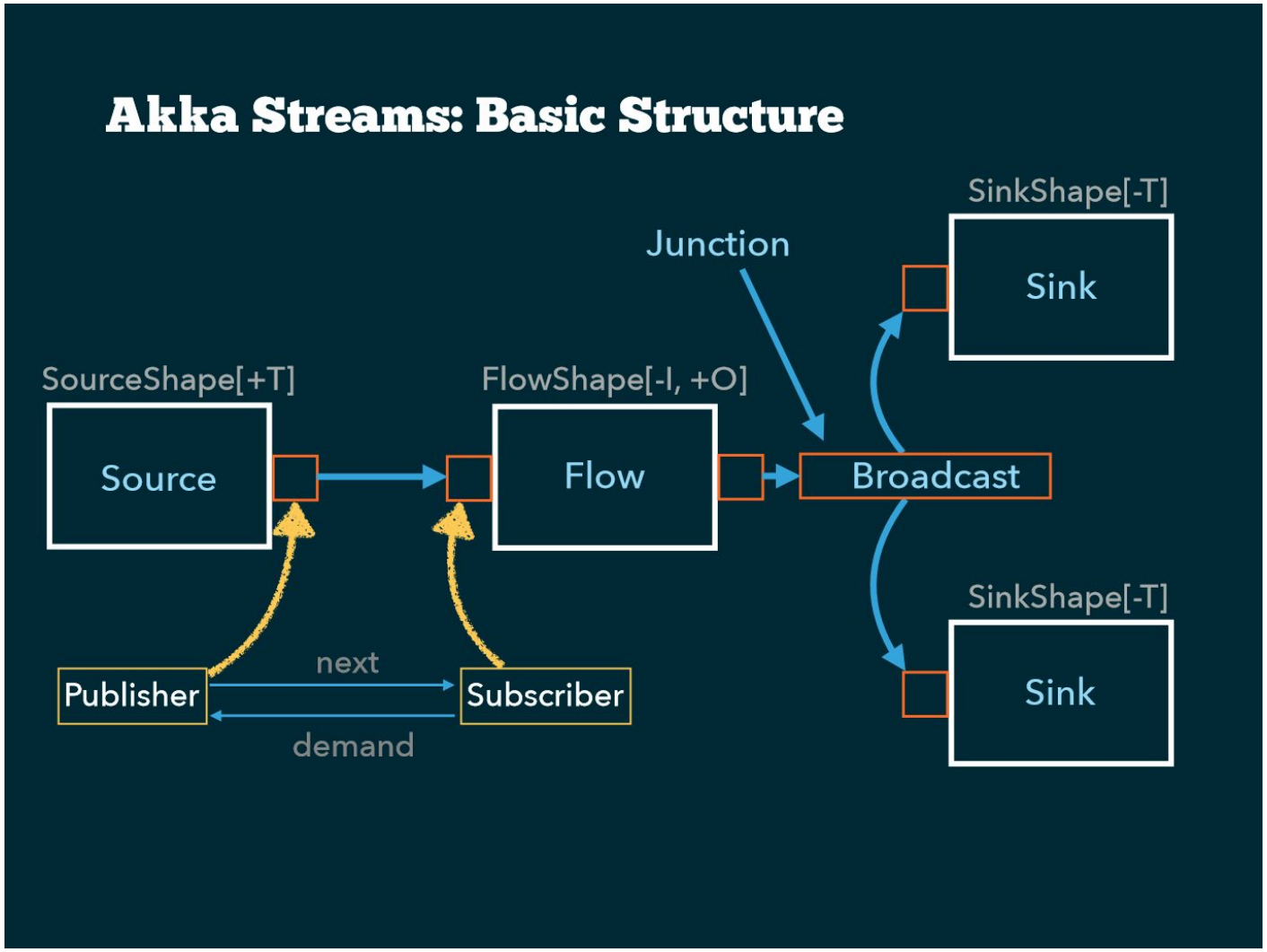
- Runnable Graph
  - a Flow with both ends attached to a Source and a Sink
  - Description of a processing pipeline
  - Reusable Blueprints; Execution Plans that can be run
  - Immutable, Thread-safe and sharable

```
// Graphs
val pipeline1:RunnableGraph[NotUsed] =
  Source(1 to 10).via(Flow[Int].map(_ * 10)).to(Sink.foreach(println(_)))
```

- Materializing a Runnable Graph
  - Allocating all the resources needed to run a graph
  - Creating and allocating actors, opening files, sockets

```
final class Source[+Out, +Mat] extends FlowOpsMat[Out, Mat] with Graph[SourceShape[Out], Mat]
```

Source, Sink, Flow



@Marina Popova, Edward Sumitra

Ref: https://chariotsolutions.com/blog/post/simply-explained-akka-streams-backpressure/

Processing Pipelines

- Simple pipeline with source, flow and sink

```
// Graphs
val pipeline1:RunnableGraph[NotUsed] =
  Source(1 to 10).via(Flow[Int].map(_ * 10)).to(Sink.foreach(println(_)))
```

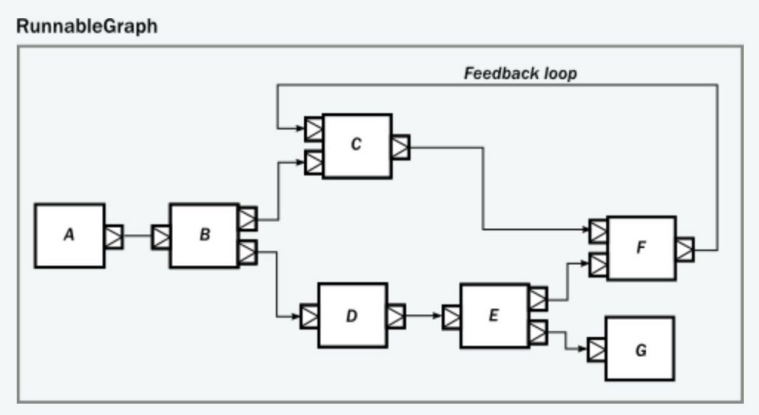
- Composing new processing pipelines from stages

```
val upto25: Flow[Int, Int, NotUsed] =
  Flow[Int].takeWhile(_ < 25)

val truncatedPrint: Sink[Int, NotUsed] =
  upto25.to(Sink.foreach(println))

val pipeline2: RunnableGraph[NotUsed] =
  simpleRange.to(truncatedPrint)
```

Building Data Pipelines




```
import GraphDSL.Implicits._
RunnableGraph.fromGraph(GraphDSL.create() { implicit builder =>
  val A: Outlet[Int]                  = builder.add(Source.single(0)).out
  val B: UniformFanOutShape[Int, Int] = builder.add(Broadcast[Int](2))
  val C: UniformFanInShape[Int, Int]  = builder.add(Merge[Int](2))
  val D: FlowShape[Int, Int]          = builder.add(Flow[Int].map(_ + 1))
  val E: UniformFanOutShape[Int, Int] = builder.add(Balance[Int](2))
  val F: UniformFanInShape[Int, Int]  = builder.add(Merge[Int](2))
  val G: Inlet[Any]                   = builder.add(Sink.foreach(println)).in

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

  ClosedShape
})
```

Akka Streams Reference

Akka Streams Reference Guide

https://doc.akka.io/docs/akka/2.5.2/scala/stream/index.html

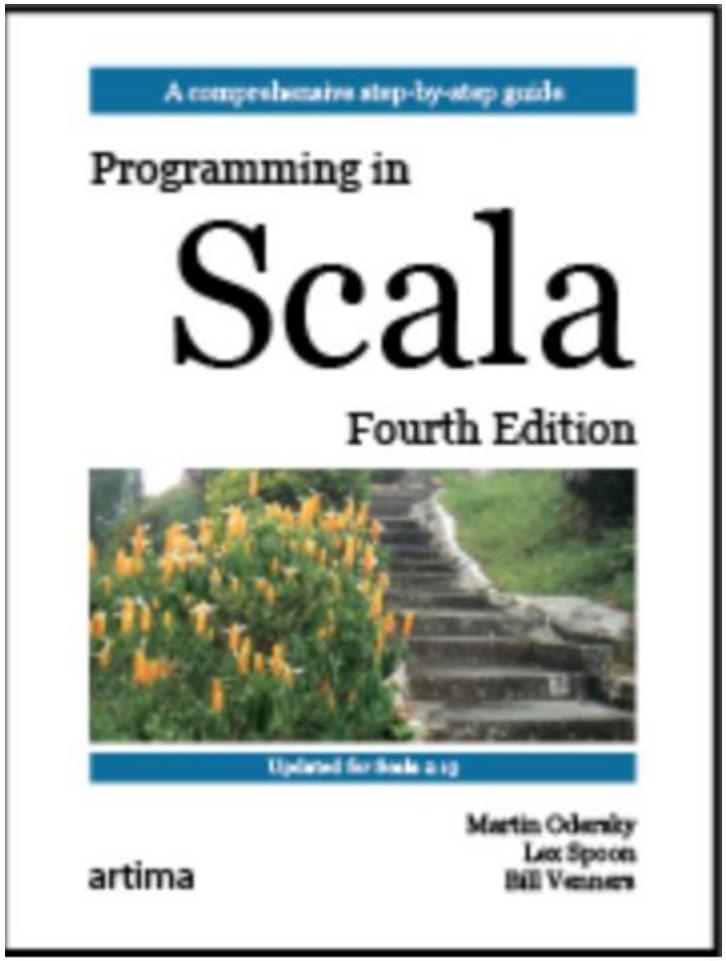
Akka Streams DSL API Docs

https://doc.akka.io/api/akka/2.5.23/akka/stream/scaladsl/index.html

Scala Reference

Programming in Scala

[Programming in Scala, First Edition]



@Marina Popova, Edward Sumitra