Introduction to Akka Streams

Reactive Stream Processing with Back Pressure

Christian Simon illucIT Software GmbH

Outline

- Reactive Streams
  - Motivation
  - Concepts
  - Back-Pressure
  - Use Cases
- Akka Streams
  - Materializing and running
  - Stream completion
  - Example
    - Sources
    - Sinks
    - Flows
  - Combinators
  - File I/O
  - Akka Http
    - Server
    - Client
  - GraphDSL

Reactive Streams

Motivation

Goal:

Processing of data streams …

Challenges:

- … of unknown length?
- … with memory constraints?
- … with flow capacity constraints?
- … non-blocking / asynchronously?
- “Big Data”?

Motivation (2)

Solution:

- Standard API for stream processing
- Flow control through “Back-Pressure”
- Implementation with Akka actors
- Functional composition of processing stages
- Bounded buffers of processing elements
- Reusability of defined flows

Concepts

Stream:

Active process that involves moving and transforming data from “upstream” to “downstream” (= materialized graph)

Graph:

Connected pathway in directed graph, defining the topology/blueprint of a stream

Graph Stage:

Building block of graph,

Concepts (2)

Source:

Graph element with 1 output, emitting elements into the stream

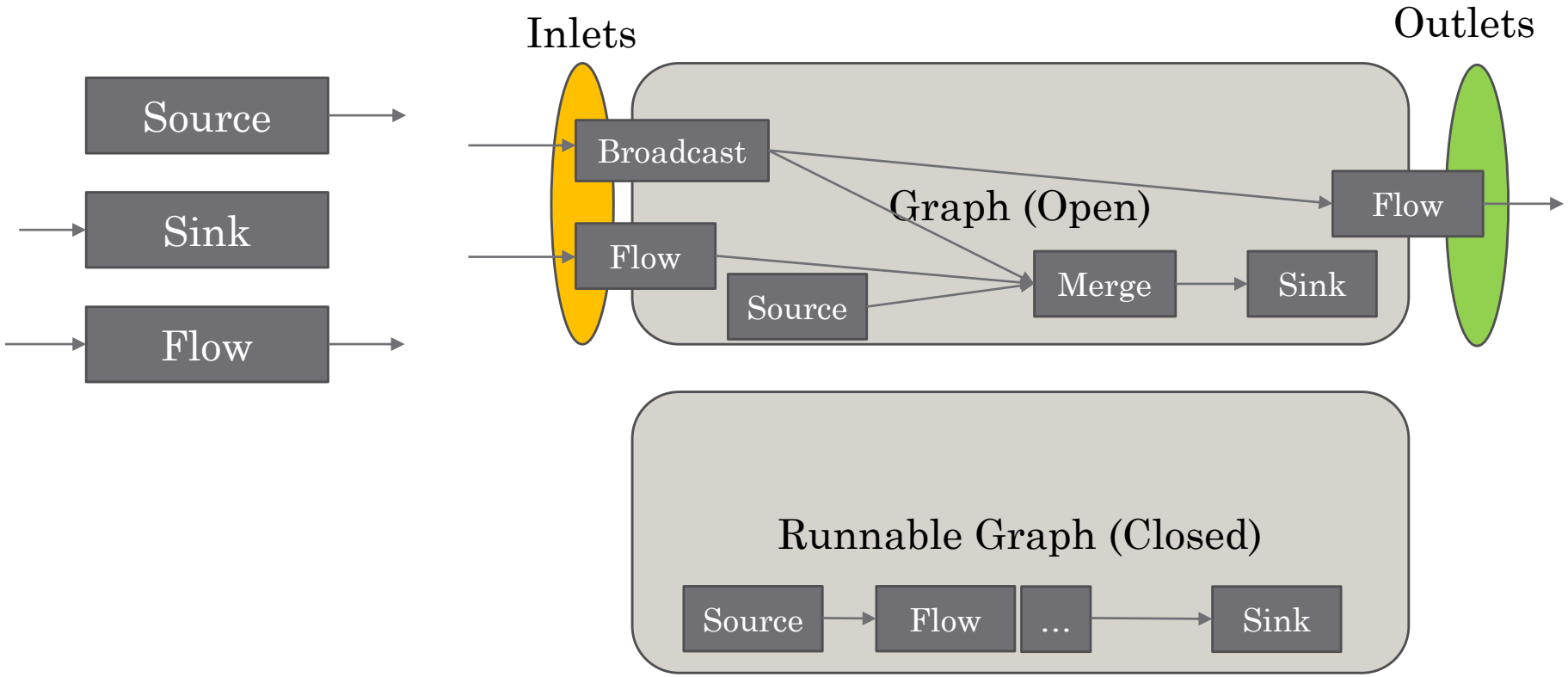
Sink:

Graph element with 1 input, consuming elements from the stream

Flow:

Graph element with 1 input and 1 output

Concepts (3)


Source
Sink
Flow
Inlets
Outlets
Broadcast
Flow
Source
Graph (Open)
Merge
Sink
Flow
Runnable Graph (Closed)
Source
Flow
...
Sink

Back-Pressure

Problem:

Fast Publisher, Slow Subscriber:

Publisher emits more elements into a stream than the Subscriber can handle

➔Buffer overflows, elements lost (bounded)

➔Out of Memory Errors (unbounded)

Solution:

Subscriber signals the Publisher the rate of elements it can consume

Use Cases

- I/O Streaming
  - Reading/writing large files
  - Processing/transforming data from files while reading
  - Streaming data from/to sockets
- Handling of Requests (as Flow)
  - HTTP requests (Akka Http)
  - WebSockets
- Complex stream processing by composition of processing stages
  - Custom Graphs/GraphStages
  - “GraphDSL”

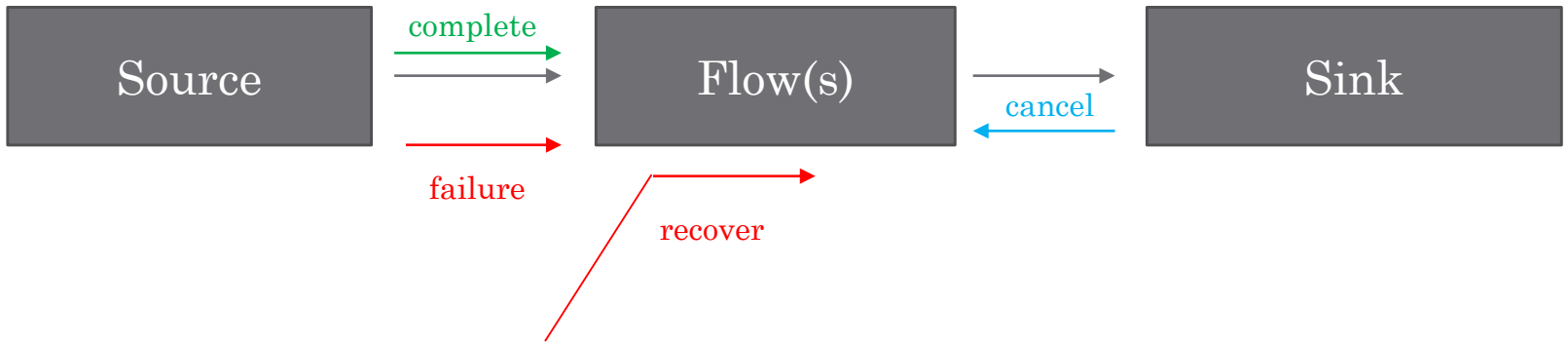
Akka Streams

Materializing and running

- Streams need to be materialized from graphs (blueprint → stream)
- Akka Actor System, Actor Materializer required
- Stream processing happens in dedicated threadpool
- Each graph stage gets materialized to runtime value, to access the status **while the stream is running**
  - Usually `Future[_]` when graph stage contains
  - Value "`NotUsed`" for graph stages where the materialized is irrelevant (e.g. `map()`)
  - Combining
  - With `map(), filter(), via()` … the materialized value of the source is kept by default
  - With `runWith()` the materialized value of the sink is kept by default

Graph completion

- A stream finishes:
  - when the source signals no more elements to downstream (“complete”)
  - when the sink requests no more elements from upstream (“cancel”)
  - when the stream is failed (e.g. Exception in flow) and no recovery is done



Example

```
object SimpleExample extends App {

  implicit val system: ActorSystem = ActorSystem("example")
  implicit val materializer: Materializer = ActorMaterializer()

  val source = Source(List("apple", "pear", "orange"))

  val sink = Sink.ignore

  // Materialize and run stream
  val streamResult = source.runWith(sink)

  // Handle future
  Await.ready(streamResult, atMost = 10 seconds)

}
```


Example: Sources

```
class Sources {

  // Empty
  val sourceEmpty = Source.empty[String]

  // From Iterable
  val sourceIterable = Source(1 to 1000)
  val sourceIterable2 = Source(List(1,2,3,99))
  val sourceIterableInfinite = Source(Stream.from(1))

  // From File
  val sourceFile = FileIO.fromPath(Paths.get("/my/file"))

  // Repeat same value
  val sourceRepeat = Source.repeat("Hello World")

  // From Future
  val promise = Promise[Int]()
  val sourceFuture = Source.fromFuture(promise.future)
  promise.success(1337)

}
```


Example: Sinks

```
class Sinks {

  // Ignore values (drain stream)
  val sinkIgnore = Sink.ignore

  // head, headOption, last, lastOption
  val sinkHead = Sink.head[String]
  val sinkHeadOption = Sink.headOption[Int]
  val sinkLast = Sink.last[String]
  val sinkLastOption = Sink.lastOption[String]

  // fold (e.g. count chars)
  val sinkFoldCount = Sink.fold[Int, String](zero = 0)((numChars, nextString) => numChars + nextString.length)
  val sinkFoldConcat = Sink.fold[ByteString, ByteString](ByteString.empty)(_ ++ _)

  // collect all values
  val sinkSeq = Sink.seq[String]

}
```

Example: Flows

```
class Flows {

  Source(1 to 100000)
    .filter(_ % 2 == 0 /* Even numbers only */)
    .drop(15 /* drop first 15 elements */)
    .map(nr => ByteString(s"[$nr] Line $nr\n") /* Convert each to ByteString as "[33] Line 33\n" */)
    .to(FileIO.toPath(Paths.get("out.txt"))) /* Write to file */
    .run()

  val fruit = Source(List("apple", "banana", "orange"))
  val prices = Source(List("1,33", "2,44", "0,99")).map(_ + "€")

  fruit
    .zipWith(prices)("The " + _ + " costs " + _)
    .runForeach(println)
  // The apple costs 1,33€
  // The banana costs 2,44€
  // The orange costs 0,99€

}
```


Combinators

Merge:

Combine multiple sources to a single, combined source (Fan-in)

Broadcast:

Emit to multiple sinks (Fan-out)

Zip:

Combine each 1st, 2nd, … elements from multiple streams

Concat:

Pull from 1st stream, and then from 2nd stream after the first finished

FileIO

```
class FileIOExample {

  // File source: Read data (ByteString) from file
  val fileSource = FileIO.fromPath(Paths.get("in.dat"), chunkSize = 8192)

  // File sink: Store data (ByteString) into file
  val fileSink = FileIO.toPath(Paths.get("out.dat"))

  // Stream from one file to another
  fileSource.runWith(fileSink)

}
```


Akka Http: Server

```
val route: Route = {
  path("url") {
    get {
      complete {
        val source = FileIO.fromPath(Paths.get("myfile.txt"))
        // Stream download from file or other source
        HttpEntity(ContentTypes.`application/octet-stream`, source)
      }
    } ~
    post {
      extractRequestEntity { entity =>
        val sink = FileIO.toPath(Paths.get("myfile.txt"))
        // Stream upload into file or other sink
        onSuccess(entity.dataBytes.runWith(sink)) { streamResult =>
          complete {
            if (streamResult.wasSuccessful) {
              "upload complete"
            } else {
              StatusCodes.InternalServerError -> s"error storing file: ${streamResult.getError.getMessage}"
            }
          }
        }
      }
    }
  }
}
```


Akka Http: Client

```scala
val httpFlow: Flow[HttpRequest, HttpResponse, Any] = Http().outgoingConnectionHttps("illucit.com", port = 443)

// Single Request

val request = Get("https://www.illucit.com/impressum/")

val singleResponse: Future[String] =
  Source.single(request)
    .map(_ ~> addHeader(Cookie("qtrans_front_language" -> "de")))
    .via(httpFlow)
    .mapAsync(1)(Unmarshal(_).to[String])
    .runWith(Sink.head)

// Flow

val responseFlow: Flow[Uri, String, NotUsed] = Flow[Uri]
  .map(uri => Get(uri))
  .map(_ ~> addHeader(Cookie("qtrans_front_language" -> "de")))
  .via(httpFlow)
  .mapAsync(1)(Unmarshal(_).to[String])

val responses: Future[Seq[String]] =
  Source(List(Uri("https://www.illucit.com/"), Uri("https://www.illucit.com/robots.txt"), Uri("https://www.illucit.com/sitemap_index.xml")))
    .via(responseFlow)
    .runWith(Sink.seq)
```


GraphDSL

- Domain specific language to defined graphs from basic building blocks
- Inner graph stages define behavior of graph
- Non-connected graph stages define the “shape of the graph”
  - Closed Shape → Graph has no unconnected ports → RunnableGraph
  - FlowShape: One Input and One Output can be connected
  - SourceShape: One Output
  - SinkShape: One Input

GraphDSL: Example

```
RunnableGraph.fromGraph(GraphDSL.create() {
  implicit builder: GraphDSL.Builder[NotUsed] =>
    import GraphDSL.Implicits._

    val in = Source(1 to 10)
    val out = Sink.ignore

    val bcast = builder.add(Broadcast[Int](2))
    val merge = builder.add(Merge[Int](2))

    val f1, f2, f3, f4 = Flow[Int].map(_ + 10)

    in ~> f1 ~> bcast ~> f2 ~> merge ~> f3 ~> out
    bcast ~> f4 ~> merge
    ClosedShape
})
```


GraphDSL: Example (2)

```
Flow.fromGraph(GraphDSL.create() {
  implicit builder: GraphDSL.Builder[NotUsed] =>
    import GraphDSL.Implicits._

    val in = builder.add(Flow[Int])
    val out = builder.add(Flow[String])

    val bcast = builder.add(Broadcast[Int](2))
    val merge = builder.add(Merge[Int](2))

    val f1, f2, f3, f4 = Flow[Int].map(_ + 10)

    val toStr = Flow[Int].map(_.toString)

    in ~> f1 ~> bcast ~> f2 ~> merge ~> f3 ~> toStr ~> out
    bcast ~> f4 ~> merge
    FlowShape(in.in, out.out)
})
```


ILLUCIT
SOFTWARE

Thank you for your attention !

github.com/metaxmx

simon@illucit.com

www.illucit.com