Rhein-Main Scala Enthusiasts

Reactive Streams

control flow, back-pressure, akka-streams

About me - Alexey Novakov

- Working at dataWerks
- 10 years with JVM, 3 years with Scala
- Focusing on distributed systems
- Did online courses for learning Java language

What is Reactive Stream?

- It is an initiative to provide a standard for **asynchronous** stream processing with **non-blocking back pressure.** (JVM & JavaScript)

- Started by Lightbend, Pivotal, Netflix and others

http://www.reactive-streams.org

JVM Interfaces

at Maven Central: API + Technology Compatibility Kit

```
"org.reactivestreams" % "reactive-streams" % "1.0.1”

"org.reactivestreams" % "reactive-streams-tck" % "1.0.1" % "test"
```

Now is in JDK 9 as

`java.util.concurrent.Flow`

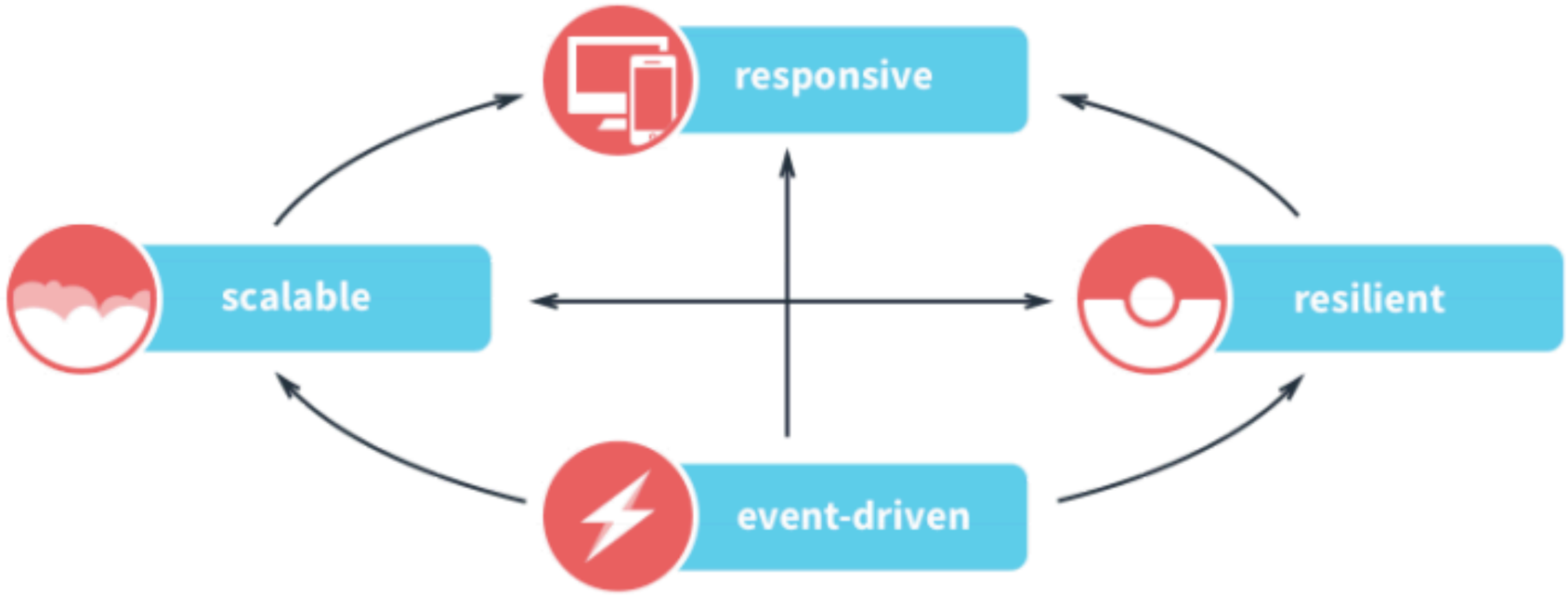
`It is a copy of RS API`

Content

- Keywords:

  - ➔ publisher, subscriber, processor, subscription
  - ➔ data stream processing
  - ➔ synchronous / asynchronous
  - ➔ back-pressure

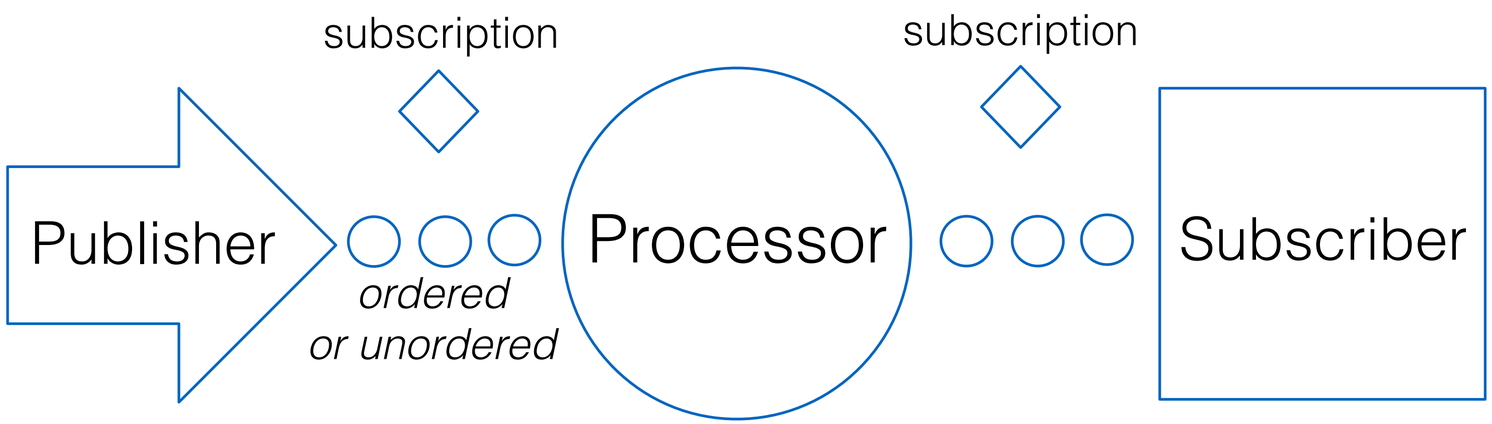
Reactive Manifesto



http://www.reactivemanifesto.org

… Reactive Streams are also related to Reactive Manifesto

Stream parts



Message Queue
Http con.
Database con.
File
etc.

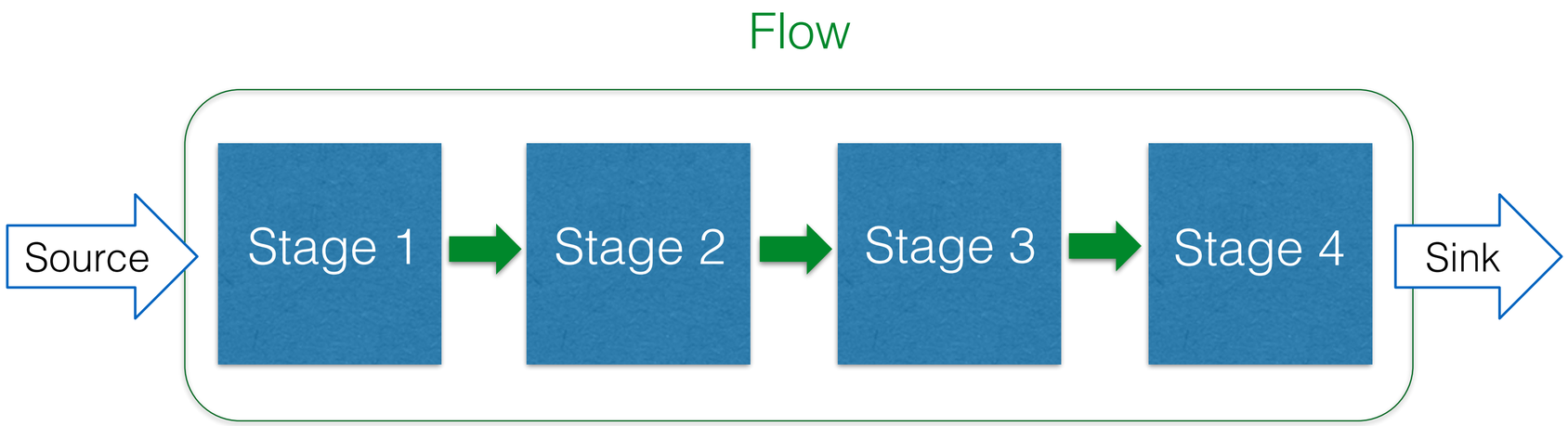
some data transformation
function

Console
TCP
<can be the same as Source>
etc.

Typical Scenarios with unbounded data processing

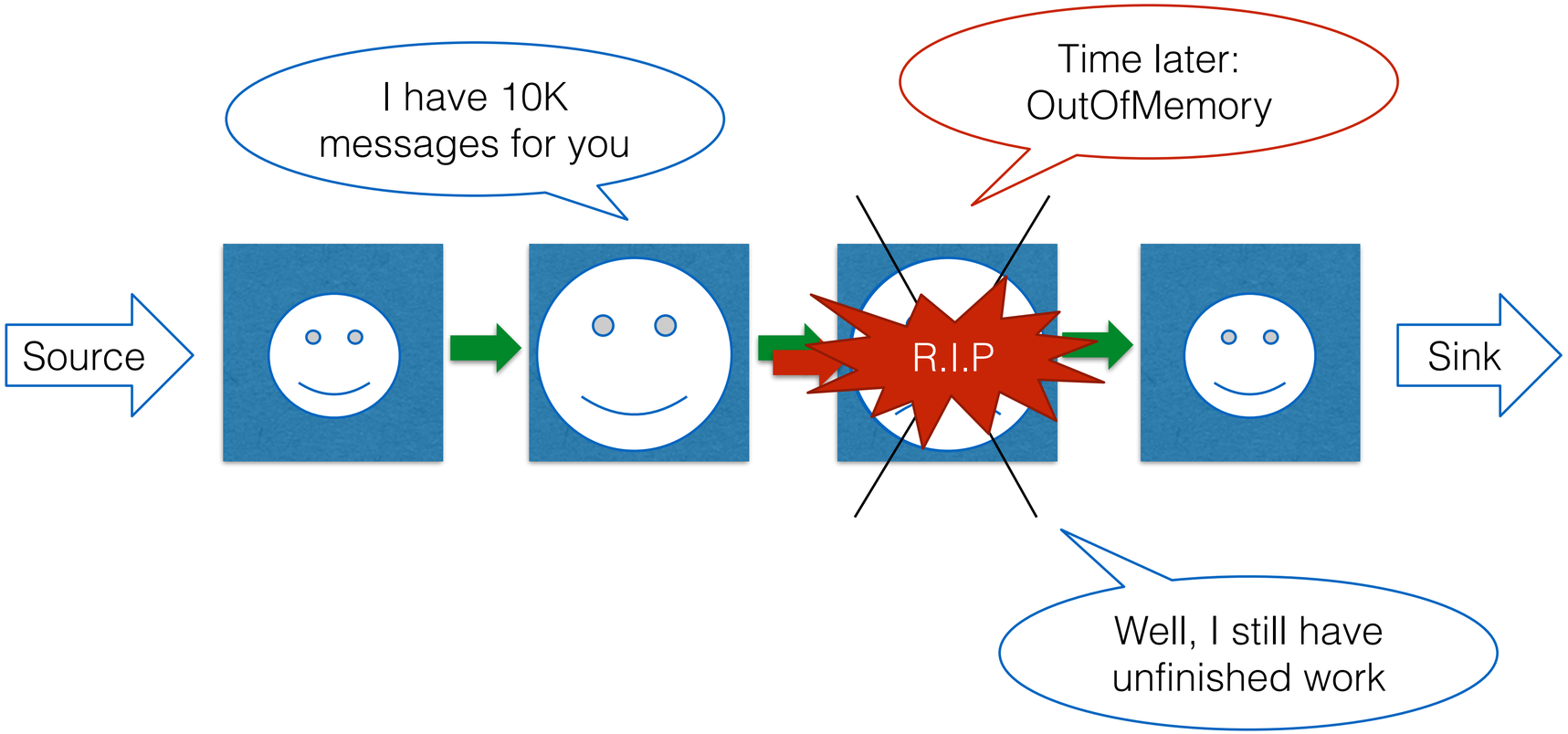
Publisher & Subscriber

Data constantly is moving from Source to Sink



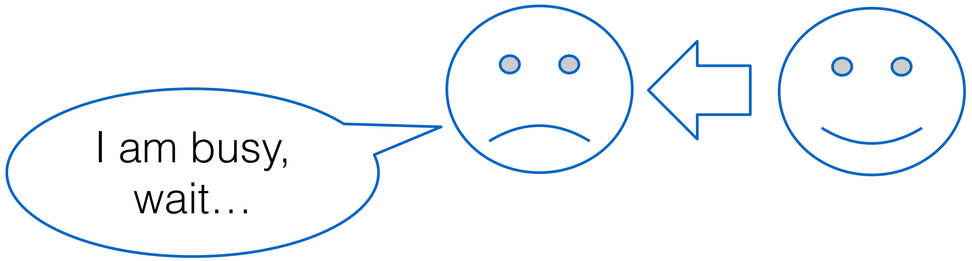
Each flow stage can be sync or async

Publisher & Subscriber



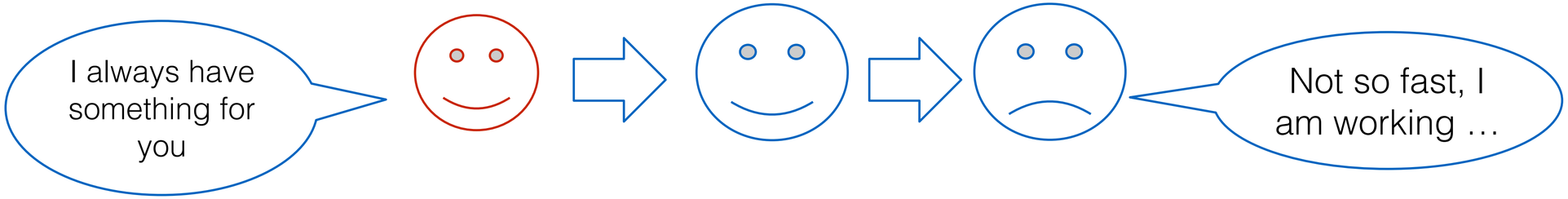
Problem situations

1) Slow Publisher, Fast Subscriber

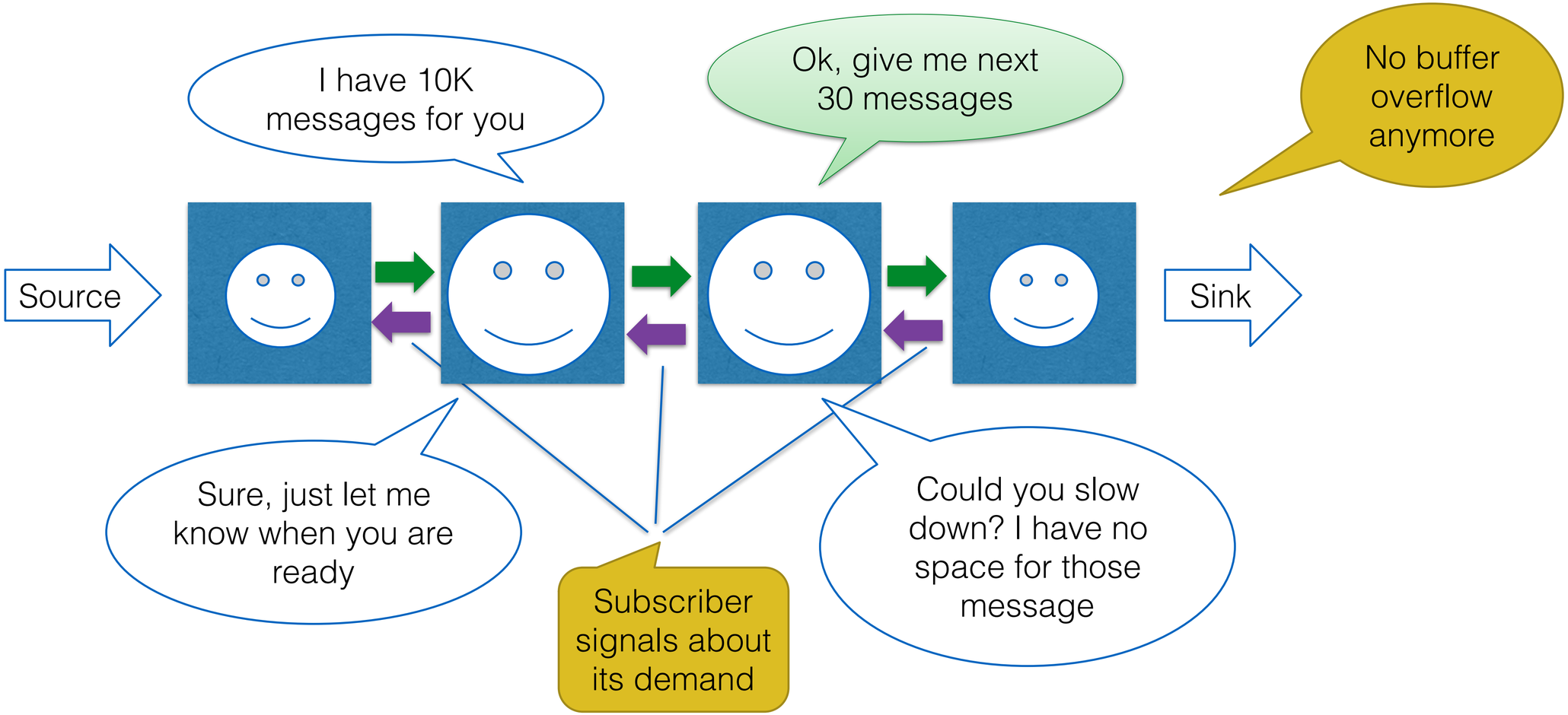


2) Fast Publisher, Slow Subscriber

Publisher also has to deal with its own back-pressure.



Stream w/ back-pressure



Interfaces

Publisher:

void subscribe(subscriber)

Subscriber:

onSubscribe(s)
onNext*(e)
onError(t) | onComplete ?

Subscription:

request(n)
cancel

Back-pressure

- Subscriber tells **number** messages it can process 

- Publisher sends that **requested** amount 

- It is simple protocol to enable dynamic *"push-pull"* communication

  - ➢ Propagated through the entire stream (Source -> Sink)
  - ➢ Enables bounded queue/buffer

Implementation

- Akka Streams
- MongoDB
- Ratpack
- Reactive Rabbit
- Reactor
- RxJava
- Slick
- Vert.x 3.0
- Monix

Streams Implementation

- appeared around 2014
- uses Actors behind the scene
- provides Scala and Java DSL
- driven by Lightbend Akka Team
- simplifies usage Actors in some sense

Example 1

Source	A .. Z	(Publisher)
Flow	A + B + C + D…	(Processor)
Sink	print(ABCD…)	(Subscriber)

Example 1

```
implicit val system = ActorSystem("Example1")
implicit val materializer = ActorMaterializer()

val source = Source('A' to 'Z')
val fold = Flow[Char].fold(" ")(_ + _)
val sink = Sink.foreach[String](println)
source.via(fold).to(sink).run
```

These guys need to be around

Stream Parts

Bind and execute in a separate thread

```
Output: ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Get a value back

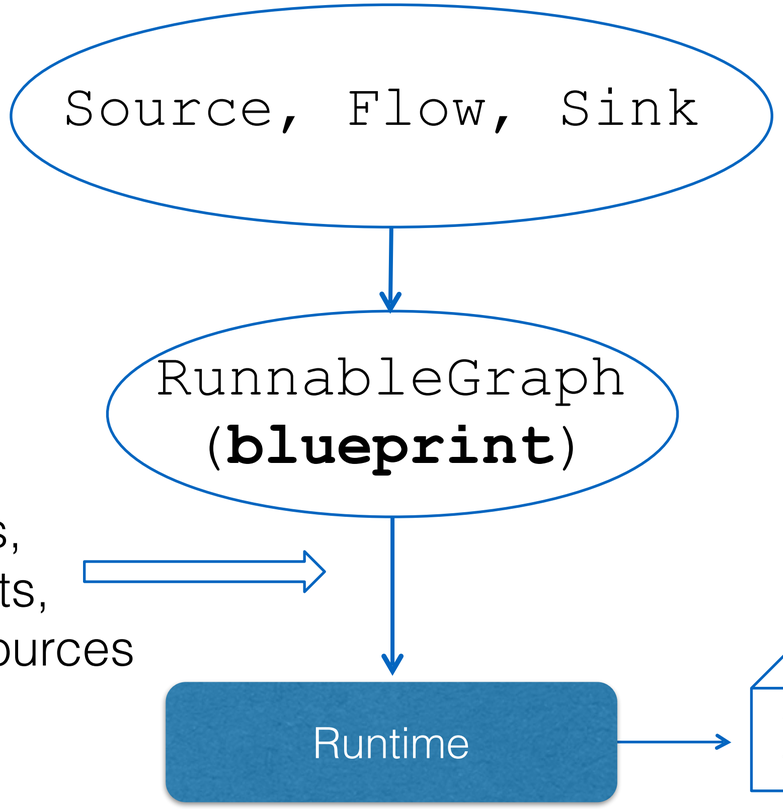
- Sometimes you need to run short-term stream and get a side-value as its result
- It can be some metric
- Or last element of the executed stream, etc.
- Akka-Streams calls this process - **Materialization**

Example of mat value

```
val source = Source(1 to 100)
val concat = Flow[Int].filter(_ % 2 == 0)
val sink = Sink.fold[Int, Int](0)(_ + _)

val g: RunnableGraph[Future[Int]] =
          source.via(concat).toMat(sink)(Keep.right)

val sum: Future[Int] = g.run
sum.foreach(print)

Output: 2550
```

Akka-Streams

API reminds
Scala Collections
std library



Order of streaming
elements is preserved

Starting Actors,
Opening Sockets,
Providing other resources

A graph **may** produce some
value as the result of stream
execution

Materialized values

Stream Materialization

- By default processing stages are fused:
  - only one Actor will be used
  - single-threaded processing

```
Source(List (1, 2, 3))
    .map(_ + 1)
    .map(_ * 2)
    .to(Sink.ignore)
```

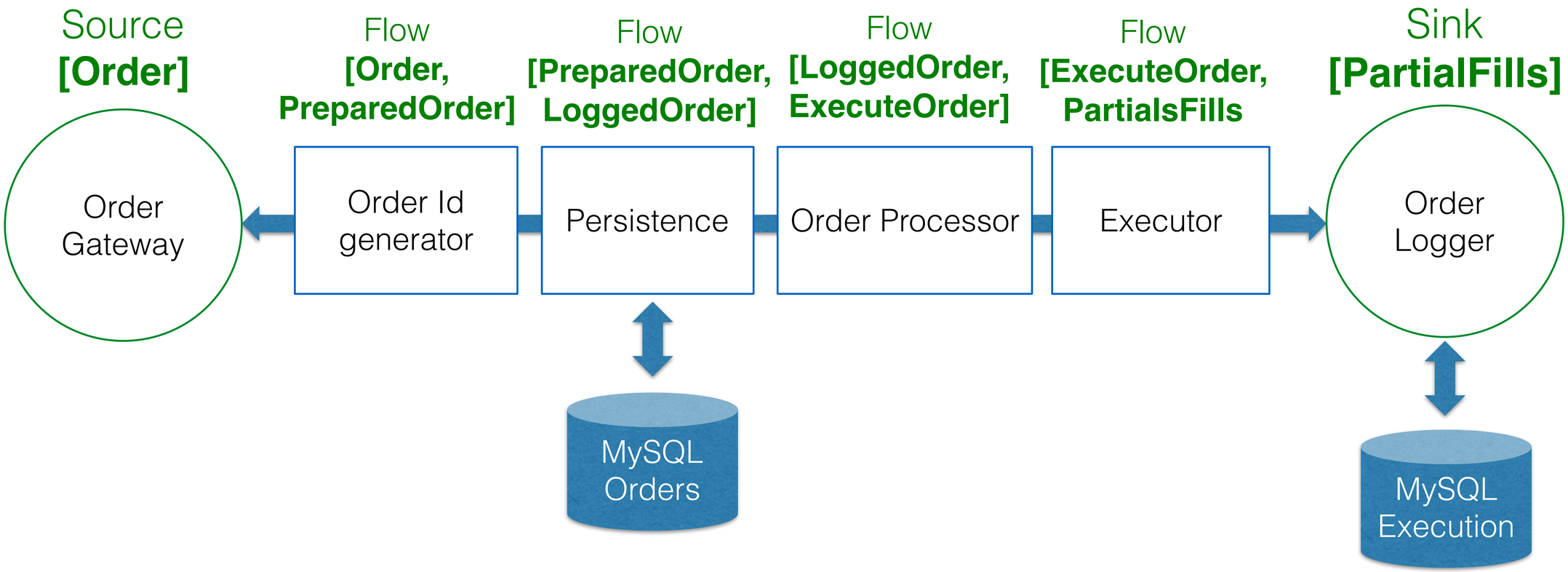
Stream Materialization

- Use “async” combinator to run on multiple actors

Async boundaries

```
Source(List (1, 2, 3))
    .map(_ + 1).async
    .map(_ * 2)
    .to(Sink.ignore)
```

Example 2 – Stock Exchange Stream



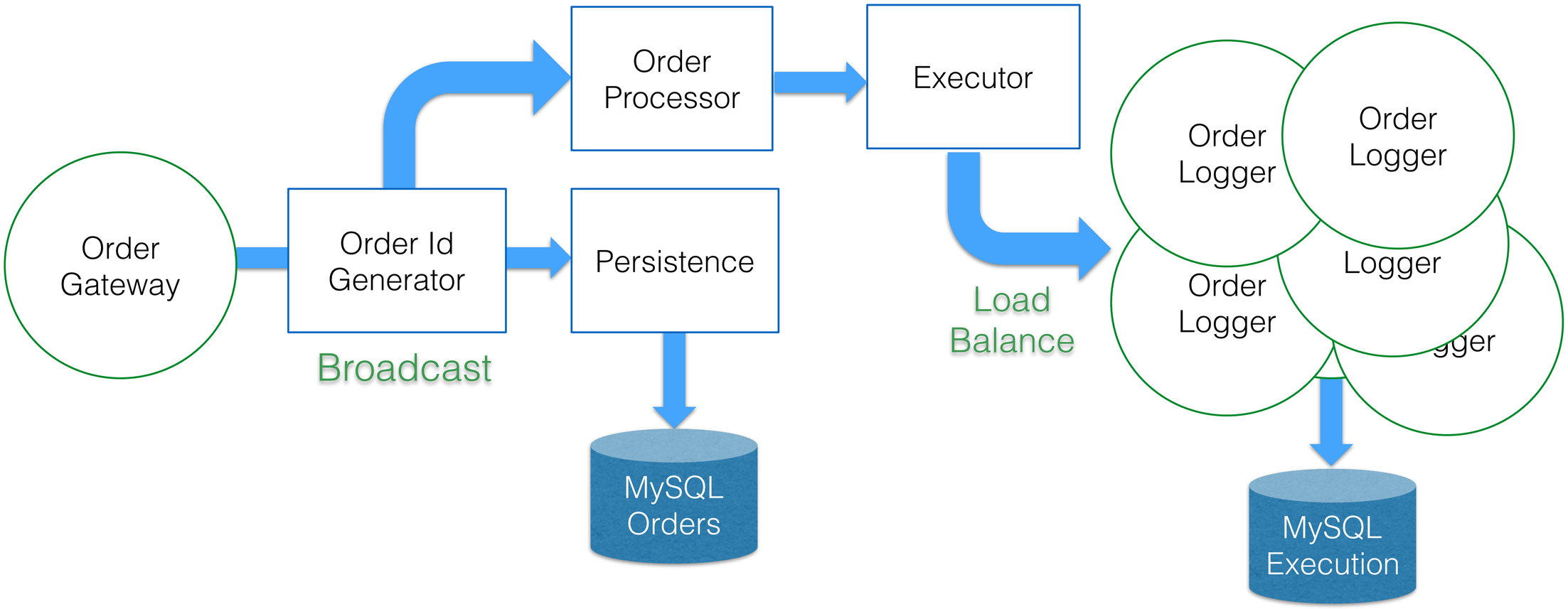
```scala
val orderPublisher = ActorPublisher[Order](orderGateway)

Source.fromPublisher(orderPublisher)
  .via(OrderIdGenerator())
  .via(OrderPersistence(orderDao))
  .via(OrderProcessor())
  .via(OrderExecutor())
  .runWith(Sink.actorSubscriber(orderLogger))

// testing: send some orders to publisher actor
1 to 1000 foreach {
    _ => orderGateway ! generateRandomOrder
}
```

Is an ActorRef

It is not aware
about back-pressure

```scala
object OrderIdGenerator {
  def apply(): Flow[Order, PreparedOrder, NotUsed] = {
    var seqNo: Long = 0

    def nextSeqNo(): Long = {
      seqNo += 1
      seqNo
    }

    Flow.fromFunction(o => PreparedOrder(o, nextSeqNo()))
  }
}
```

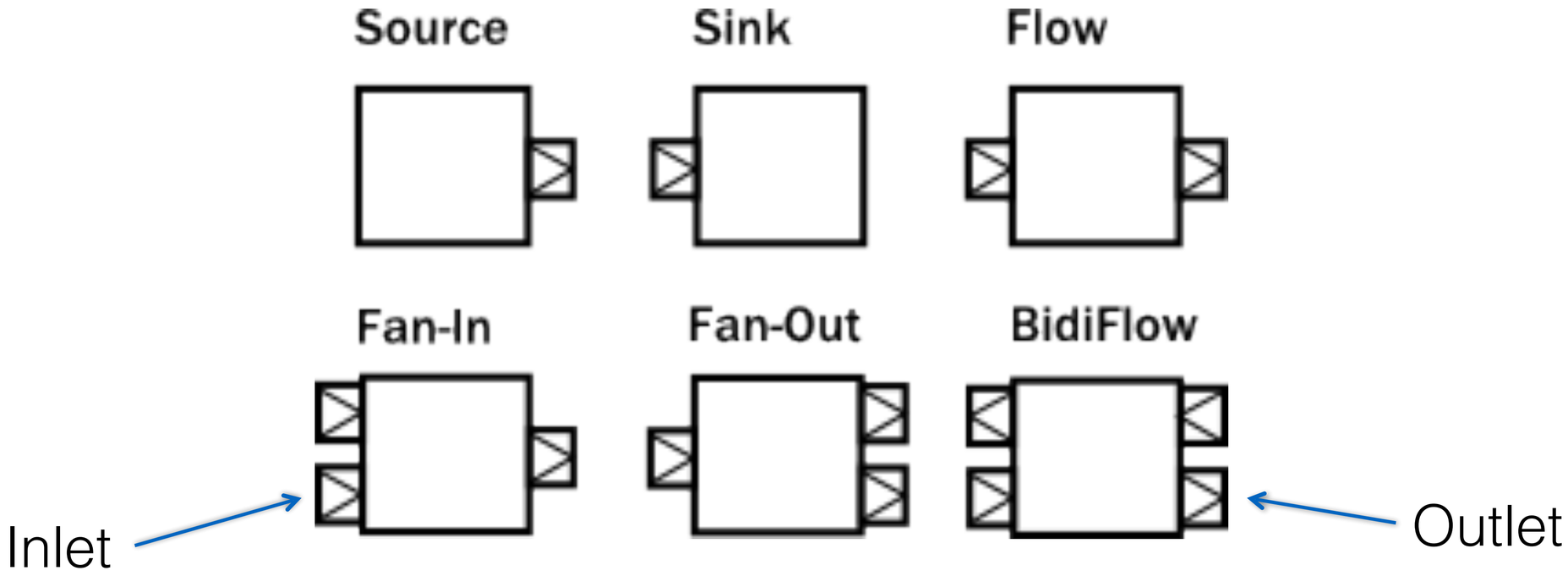
Example 2 alt. Fan Out



Example 2 alt.: Graph DSL

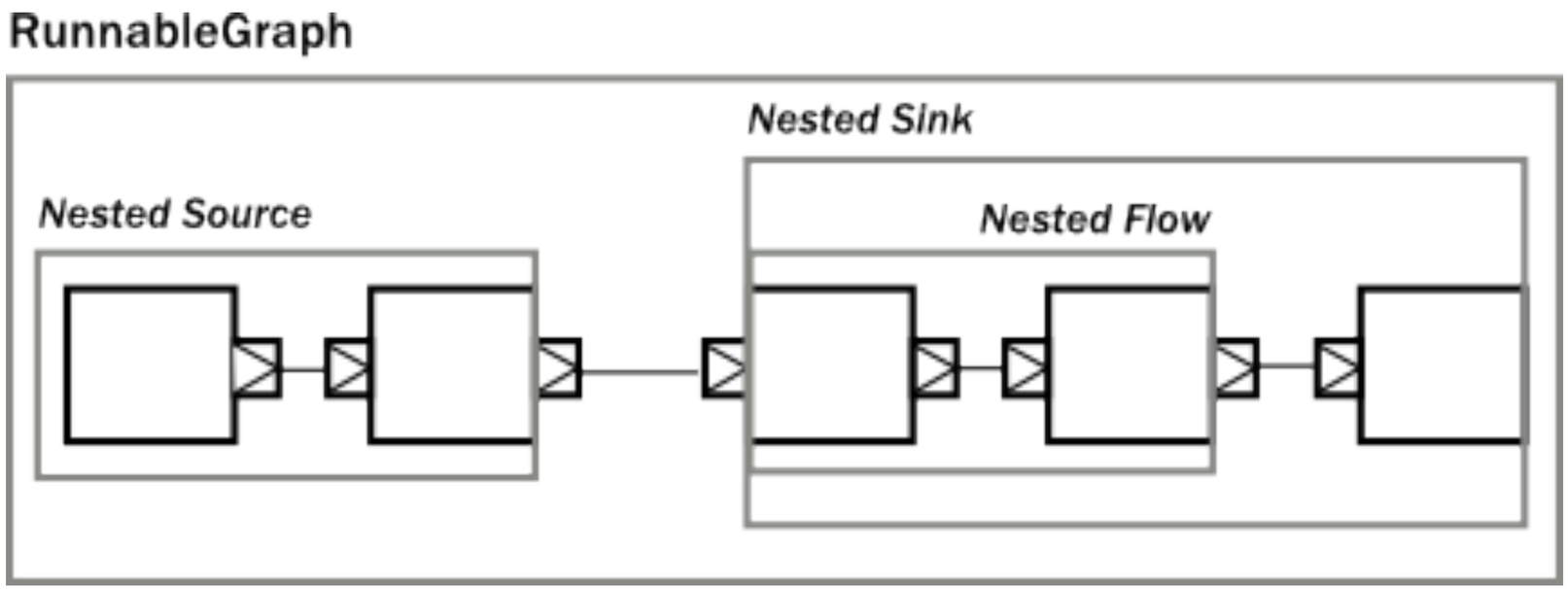
```
val bcast = b.add(Broadcast[PreparedOrder](2))
val balancer = b.add(Balance[PartialFills](workers))

val S = b.add(Source.fromGraph(orderSource))
val IdGen = b.add(OrderIdGenerator())
val A = b.add(OrderPersistence(orderDao).to(Sink.ignore))
val B = b.add(OrderProcessor2())
val C = b.add(OrderExecutor())

S   ~>          IdGen      ~> bcast
                bcast      ~> A
                bcast      ~> B             ~> C      ~> balancer

for (i <- 0 until workers)
  balancer ~> b.add(Sink.fromGraph(orderLogger).named(s"logger-$i"))
```

Building Blocks

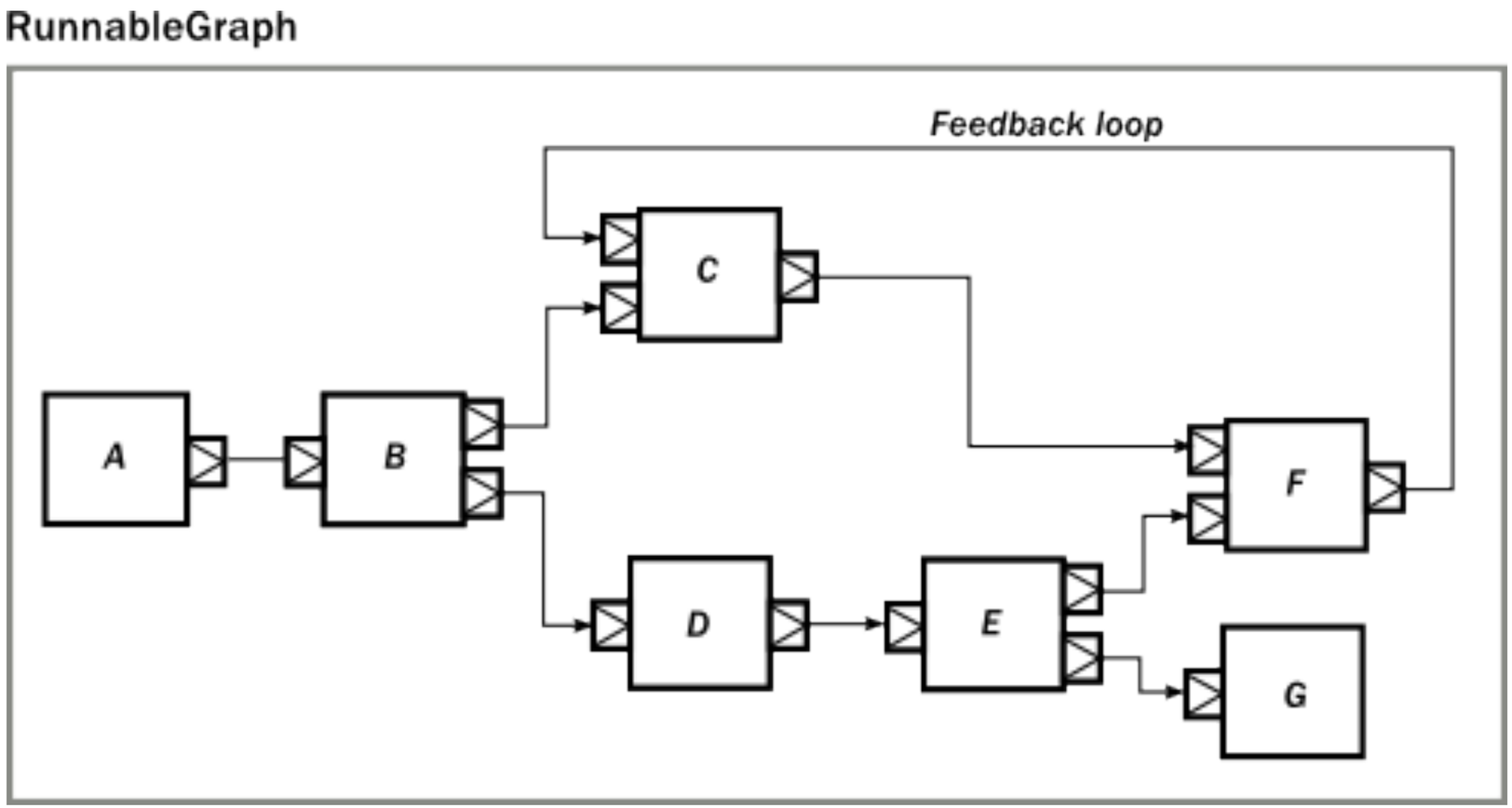


```
/**
 * A bidirectional flow of elements
 * that consequently has two inputs and two
 * outputs, arranged like this:
 *
 * {{{
 *        +------+
 *  In1 ~>|      |~> Out1
 *        | bidi |
 * Out2 <~|      |<~ In2
 *        +------+
 * }}}
 */
```

Nesting

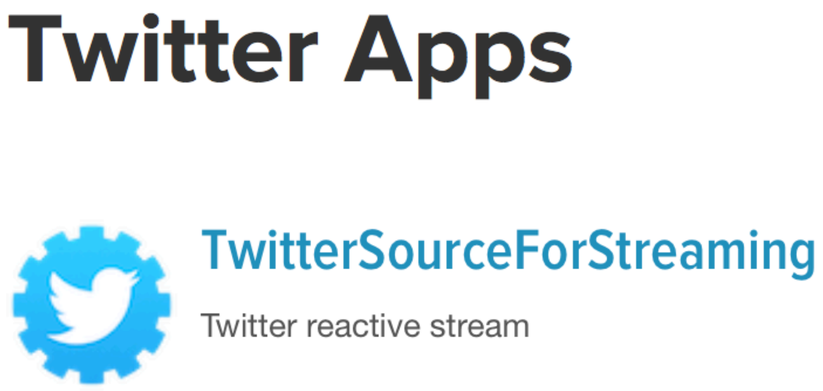


Cycling Graph



Example 3 – Twitter Stream

- Let's implement a WordCount over the infinite Twitter Stream
- We can use free API: **Filter Real-time Tweets**
  - https://stream.twitter.com/1.1/statuses/filter.json
  - HTTP chunked response
- Just register your Twitter app to get a consumer key



Example 3 – Twitter Stream

scan[ByteString] filter[String] map[Tweet] scan[String] forEach[String]

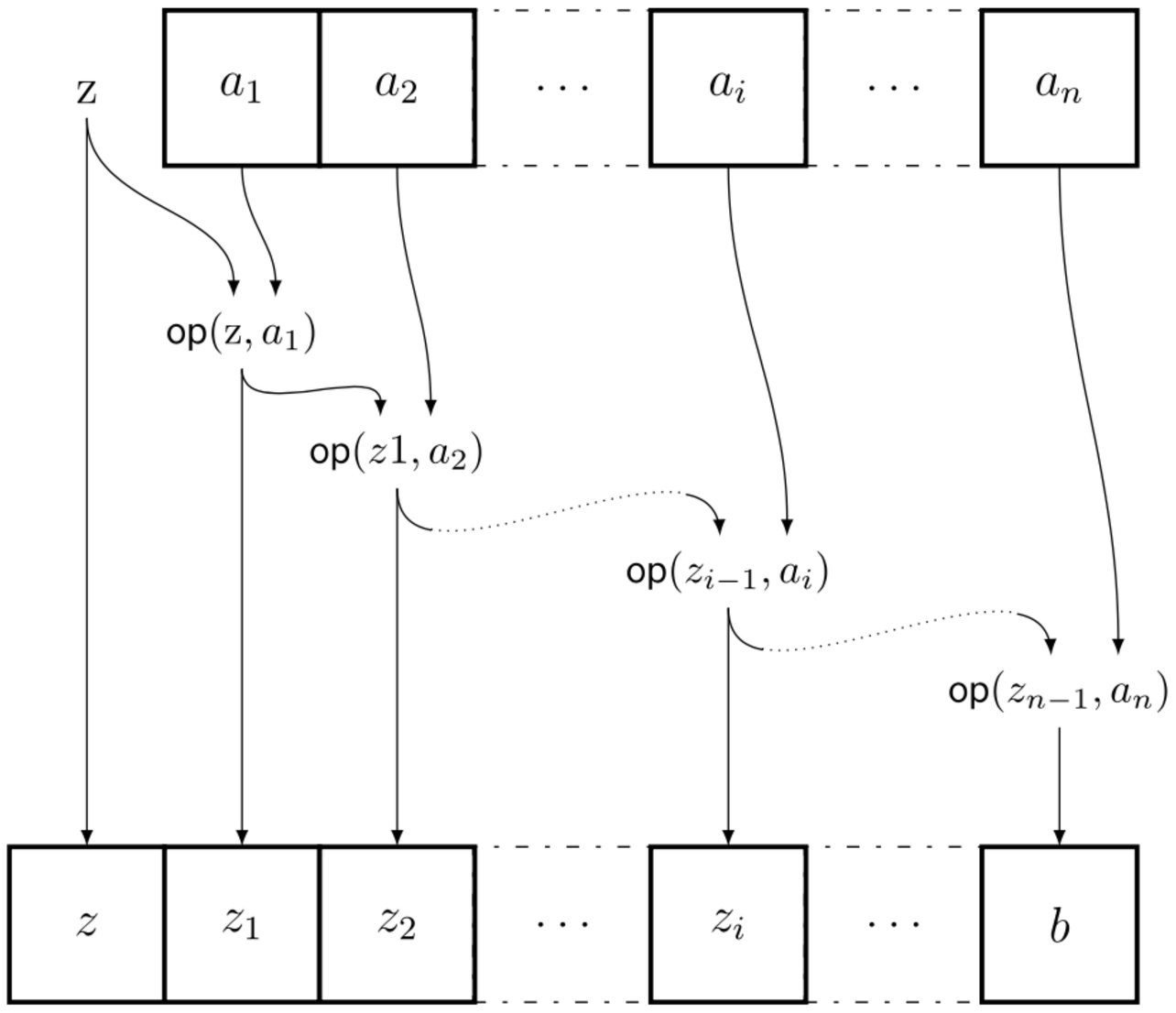
```
val response = Http().singleRequest(httpRequest)

response.foreach { resp =>
  resp.status match {
    case OK =>
      val source: Source[ByteString, Any] =
                 resp.entity.withoutSizeLimit().dataBytes

    ...

  }
}
```

Hi, akka-http :-)

```
source
  .scan("")((acc, curr) =>
    if (acc.contains("\r\n")) curr.utf8String
    else acc + curr.utf8String
  )

  .filter(_.contains("\r\n")).async
```

Reseting accumulator here

scanLeft



https://superruzafa.github.io/visual-scala-reference/index.html#scanLeft

```
.scan(Map.empty[String, Int]) {
  (acc, text) => {
    val wc = tweetWordCount(text)
    ListMap(
       (acc combine wc).toSeq
       .sortBy(- _._2)
       .take(uniqueBuckets): _*
    )
  }
}
```

Starting from this stage, flow is concurrent

```scala
def tweetWordCount(text: String): Map[String, Int] = {
  text.split(" ")
    .filter(s => s.trim.nonEmpty && s.matches("\\w+"))
    .map(_.trim.toLowerCase)
    .filterNot(stopWords.contains)
    .foldLeft(Map.empty[String, Int]) {
      (count, word) => count |+| Map(word -> 1)
    }
}
```

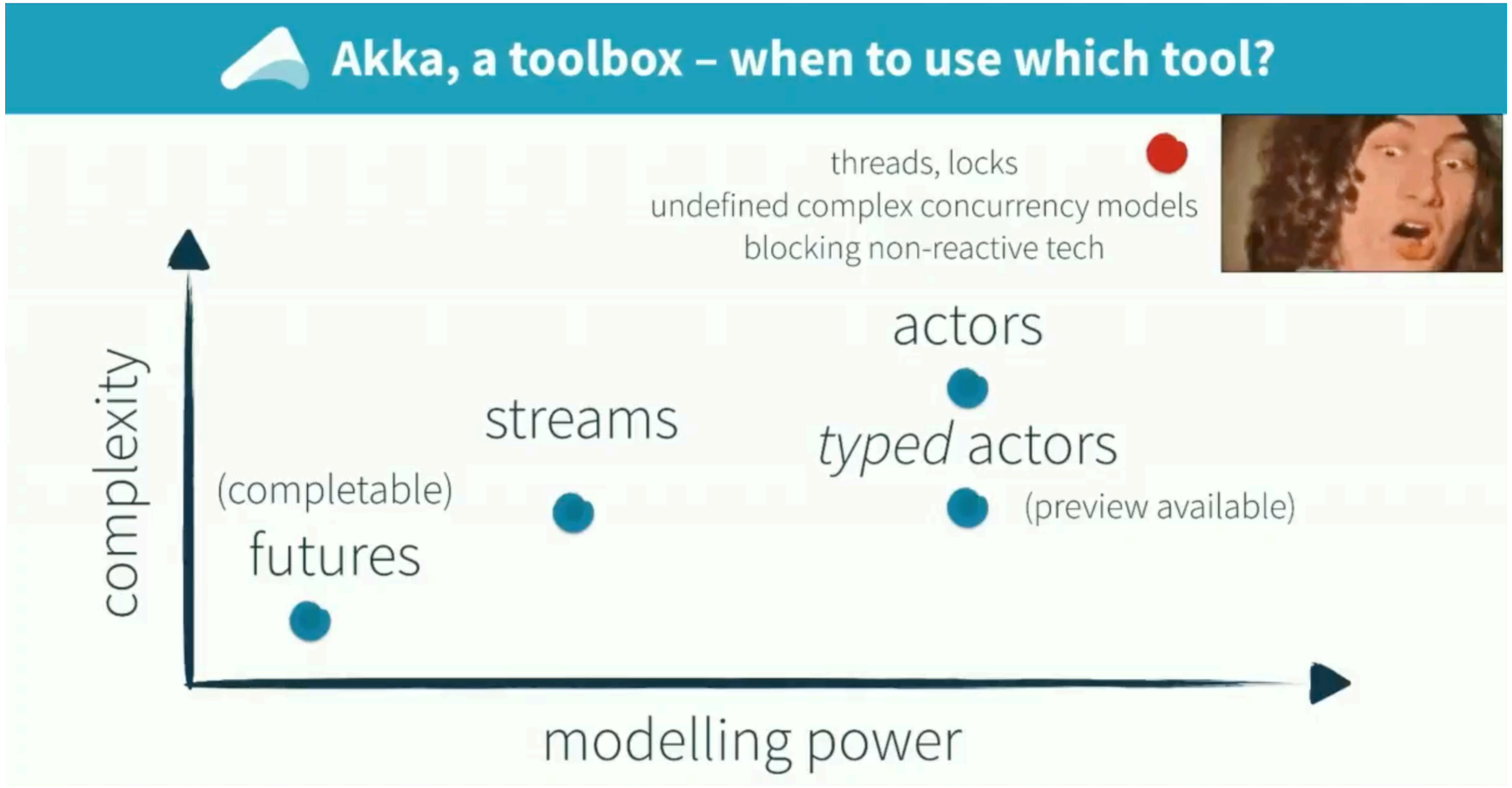
```
.runForeach { wc =>

  val stats = wc.take(topCount)
    .map{case (k, v) => k + ":" + v}.mkString("  ")

  print("\r" + stats)
}
```

Project Alpakka

Source/Flow/Sink implementation for many popular data sources

- AMQP Connector
- Apache Geode connector
- AWS DynamoDB Connector
- AWS Kinesis Connector
- AWS Lambda Connector
- AWS S3 Connector
- AWS SNS Connector
- AWS SQS Connector
- Azure Storage Queue Connector
- Cassandra Connector
- Elasticsearch Connector
- File Connectors
- FTP Connector
- Google Cloud Pub/Sub
- HBase connector
- IronMq Connector
- JMS Connector
- MongoDB Connector
- MQTT Connector
- Server-sent Events (SSE) Connector
- Slick (JDBC) Connector
- Spring Web
- File IO
- Azure
- Camel
- Eventuate
- FS2
- HTTP Client
- MongoDB
- Kafka
- TCP

Konrad Molawski at JavaOne 2017



https://www.youtube.com/watch?v=KbZ-psFJ-fQ

Going to Production

- Configure your ExecutionContext
- Set Supervision strategy to react on failures
- Think/test which stage can be fused and which can be done concurrently
- Think on using grouping of the elements for better throughput
- Set Overflow strategy
- Think on rate limiter using throttle combinator

Thank you! Questions?

More to learn:

- https://doc.akka.io/docs/akka/2.5.6/scala/stream/
  Official documentation

- https://github.com/reactive-streams/reactive-streams-jvm
  Reactive Streams specification

- https://blog.redelastic.com/diving-into-akka-streams-2770b3aeabb0
  Kevin Webber, Diving into Akka Streams

- http://blog.colinbreck.com/patterns-for-streaming-measurement-data-with-akka-streams/
  Colin Breck: Patterns for Streaming Measurement Data with Akka Streams

- https://github.com/novakov-alexey/meetup-akka-streams
  Examples Source Code