Dr Streamlove

or: How I Learned to Stop Worrying and Love the Flow

Viktor Klang
Chief Architect
@viktorklang

Typesafe
Typesafe

• Founded ~3.5 years ago
  • fusion of Scalable Solutions and Scala Solutions
• Offices in Uppsala, Lausanne and San Francisco
• 60+ employees all over the world
• Main projects
  • Play, Akka, Scala, Slick
My office door
View from my office
Agenda

• What is a Stream?
• Live demo
• What is Reactive?
• Reactive Streams
• Akka Streams
• Live demo
• What’s next / Opportunities
• Outro
What do we mean by “Stream”?
“You cannot step twice into the same stream. For as you are stepping in, other waters are ever flowing on to you.”

- Heraclitus
What is a Stream?

- Ephemeral flow of data
- Possibly unbounded in length
- Focused on describing transformation
- Can be formed into processing networks
What is a Collection?

- Oxford Dictionary:
  - “a group of things or people”
- wikipedia:
  - “a grouping of some variable number of data items”
- backbone.js:
  - “collections are simply an ordered set of models”
- java.util.Collection:
  - definite size, provides an iterator, query membership
User Expectations

• an Iterator is expected to visit all elements (especially with immutable collections)
• \texttt{x.head ++ x.tail == x}
• the contents does not depend on who is processing the collection
• the contents does not depend on when the processing happens (especially with immutable collections)
Unexpected: observed sequence depends on

• when the subscriber subscribed to the stream
• whether the subscriber can process fast enough
• whether the streams flows fast enough
java.util.stream

• Stream is not *derived* from Collection
  “Streams differ from Collections in several ways”
  • no storage
  • functional in nature
  • laziness seeking
  • possibly unbounded
  • consumable
Streams vs. Collections

• a collection can be streamed
• a stream processor can create a collection
• … but saying that a Stream is just a lazy Collection evokes the wrong associations

Streams are not Collections!
Live Demo
The Four Horsemen of Reactive

http://reactivemanifesto.org/
The Problem:

Getting Data across an Async Boundary
Possible Solutions

• the Traditional way: blocking calls
Possible Solutions

• the Push way: buffering and/or dropping
Possible Solutions

- the Reactive way:
  non-blocking & non-dropping & bounded
Reactive Streams Initiative
“Reactive Streams is an initiative to provide a standard for asynchronous stream processing with non-blocking back pressure on the JVM.”

- reactive-streams.org
Collaboration between Engineers

• Björn Antonsson – Typesafe Inc.
• Gavin Bierman – Oracle Inc.
• Jon Brisbin – Pivotal Software Inc.
• George Campbell – Netflix, Inc.
• Ben Christensen – Netflix, Inc.
• Mathias Doenitz – spray.io
• Marius Eriksen – Twitter Inc.
• Tim Fox – Red Hat Inc.
• Viktor Klang – Typesafe Inc.
• Dr. Roland Kuhn – Typesafe Inc.
• Doug Lea – SUNY Oswego
• Stephane Maldini – Pivotal Software Inc.
• Norman Maurer – Red Hat Inc.
• Erik Meijer – Applied Duality Inc.
• Todd Montgomery – Kaazing Corp.
• Patrik Nordwall – Typesafe Inc.
• Johannes Rudolph – spray.io
• Endre Varga – Typesafe Inc.
Motivation

• all participants face the same basic problem
• all are building tools for their community
• a common solution benefits everybody
• interoperability to make best use of efforts
  • e.g. use Reactor data store driver with Akka transformation pipeline and Rx monitoring to drive a vert.x REST API (purely made up, at this point)
• propose to include in future JDK

See also: Jon Brisbin’s post on “Tribalism as a Force for Good”
Recipe for Success

- minimal interfaces—essentials only
- rigorous specification of semantics
- TCK for verification of implementation
- complete freedom for many idiomatic APIs
Reactive Streams

• asynchronous & non-blocking
  • flow of data
  • flow of demand
• minimal coordination and contention
• message passing allows for distribution across
  • applications, nodes, CPUs, threads, actors
A Data Market using Supply & Demand

- data elements flow downstream
- demand flows upstream
- data elements flow only when there is demand
  - data in flight is bounded by signaled demand
  - recipient is in control of maximal incoming data rate
Dynamic Push–Pull

- “push”—when consumer is faster
- “pull”—when producer is faster
- switches automatically between these
- batching demand allows batching data
Explicit Demand: One-to-many

Splitting the data means *merging the demand*
Explicit Demand: Many-to-one

Merging the data means splitting the demand
trait Publisher[T] {
  def subscribe(sub: Subscriber[T]): Unit
}

trait Subscription {
  def request(n: Int): Unit
  def cancel(): Unit
}

trait Subscriber[T] {
  def onSubscribe(s: Subscription): Unit
  def onNext(e: T): Unit
  def onError(t: Throwable): Unit
  def onComplete(): Unit
}
trait Processor[T, R] extends Subscriber[T] with Publisher[R]
public interface Publisher<T> {
    public void subscribe(Subscriber<T> s);
}

public void Subscription {
    public void request(Int n);
    public void cancel();
}

public interface Subscriber<T> {
    public void onSubscribe(Subscription s);
    public void onNext(T t);
    public void onError(Throwable t);
    public void onComplete();
}
The dessert: Java

```java
public interface Processor<T, R>
    extends Subscriber<T>, Publisher<R> {
}
```
How does it Connect?

Publisher

subscribe

Subscription

Subscriber

onSubscribe
How does it Flow?

Publisher

- request
- Elements
  - request
  - Elements
  - request

Subscriber

- onNext
- onNext
How does it Complete?

Publisher

request

Elements

Subscriber

onNext

onComplete
How does it Fail?

Publisher

request

Elements

request

Subscriber

onNext

onError

☠
Akka Streams
WAIT! What is akka?
Akka

- Akka's unit of computation is called an Actor
- Akka Actors are purely reactive components:
  - an address
  - a mailbox
  - a current behavior
  - local storage
- Scheduled to run when sent a message
- Each actor has a parent, handling its failures
- Each actor can have 0..N “child” actors
Akka Actors

• An actor processes a message at a time
  • Multiple-producers & Single-consumer
• The overhead per actor is about ~450 bytes
  • Run millions of actors on commodity hardware
• Akka Cluster currently handles ~2500 nodes
  • 2500 nodes \times \text{millions of actors} = "ought to be enough for anybody"
Actor model fundamentals

• **CREATE**(behavior)
  • Creates a new actor

• **BECOME**(behavior)
  • Changes the actors behavior for the next message

• **SEND**(message)
  • Sends a message asynchronously and non-blocking to an actor
Actor model augmentations

- SUPERVISE(actor)
  - Lets an actor handle the failure(s) of another actor
- WATCH(actor)
  - Lets an actor observe the termination of another actor
Actor Messages vs Failures

- Parent
- Child
- Messages
- Replies
- Failures / Recovery
Actor Hierarchies

Guardian System Actor

```
system.actorOf(fooProps, "Foo")
```

```
context.actorOf(aProps, "A")
```

Diagram:

```
Foo
  /   \
A     C
  |
B   E
  |
D
```

```
Bar
  /   \
A     C
  |
B   C
  |
E
```

Typesafe

48
Actor Paths

Guardian System Actor

- Foo
  - /Foo
  - /Foo/A
    - /Foo/A/D
  - /Bar/A/B
- Bar
- A
  - B
  - E
  - D
Canonical papers

• Carl Hewitt; Peter Bishop; Richard Steiger (1973). A Universal Modular Actor Formalism for Artificial Intelligence. IJCAI.

Akka Streams

• powered by Akka Actors because
  • execution
  • distribution
  • resilience

• Typesafe (pun intended) streaming through Actors with bounded buffering

• Flow & Duct DSL is a lifted representation
  • Uses pluggable materialisation
Akka HTTP Server Overview

TCP Connection

user handler

rendering

bypass

parsing
val requestProducer = 
Flow(tcpConn.inputStream)
.transform(rootParser)
.splitWhen(_.isInstanceOf[MessageStart])
.headAndTail // Flow[(Start, Producer[...])]
.tee(bypassConsumer)
.collect {
  case (x: RequestStart, entityParts) =>
    HttpServerPipeline.constructRequest(x, entityParts) }
.toProducer(materializer)
val (bypassConsumer, bypassProducer) = Duct[(RequestOutput, Producer[RequestOutput])].collect[MessageStart with RequestOutput] { case (x: MessageStart, _) => x }.build(materializer)
val responseConsumer = 
  Duct[HttpResponse]
    .merge(bypassProducer)
    .transform(applyApplicationBypass)
    .transform(rendererFactory.newRenderer)
    .flatten(concat)
    .transform(logErrors)
    .toProducer(materializer)
    .produceTo(tcpConn.outputStream)
val logErrors =
    new Transformer[ByteString, ByteString] {
      def onNext(element: ByteString) =
        element :: Nil
      override def onError(cause: Throwable) =
        log.error(cause, "Response stream error")
    }
Advanced Live Demo
What’s next for Akka Streams?
Opportunity: API

• Current API is minimal
  • Establish core functionality and take it from there

• Naming: Use established terminology or simplified?

• Both Scala and Java APIs
  • Allows for use by other JVM-hosted languages
Opportunity: Self-tuning back pressure

• Each processing stage can know
  • Latency between requesting more and getting more
  • Latency for internal processing
  • Behavior of downstream demand
    • Latency between satisfying and receiving more
    • Trends in requested demand (patterns)
      • Lock-step
      • N-buffered
      • N + X-buffered
      • “chaotic”
Opportunity: Operation Fusion

• Compile-time, using Scala Macros
  • filter ++ map == collect
  • map ++ filter == collect?

• Run-time, using intra-stage simplification
  • Rule: `<any> ++ identity == `<any`
  Rule: identity ++ `<any> == `<any`
  • filter ++ dropUntil(cond) ++ map
  • filter ++ identity ++ map == collect
Opportunity: Operation Elision

• Compile-time, using Scala Macros
  • fold ++ take(n where n > 0) == fold
  • drop(0) == identity
  • <any> concat identity == <any>

• Run-time, using intra-stage simplification
  • map ++ dropUntil(cond) ++ take(N)
  • map ++ identity ++ take(N)
  • map ++ take(N)
Opportunity: Execution optimizations

- synchronous intra-stage execution N steps then trampoline and/or give control to other Thread / Flow
- We already do inter-stage execution reduction
Opportunity: Distributed Streams

• Encode Reactive Streams as a transport protocol
  • Possibility to run over
    • TCP
    • UDP
  • … essentially any bidirectional channel
• MUX-ing streams
• Materialize a Flow on a cluster of Akka nodes
Outro: How do I get my hands on this?

- [http://reactive-streams.org/](http://reactive-streams.org/)
- [https://github.com/reactive-streams](https://github.com/reactive-streams)
- Early Preview is available:
  "org.reactivestreams" % "reactive-streams-spi" % "0.3"
  "com.typesafe.akka" %% "akka-stream-experimental" % "0.3"
- check out the Activator template
  "Akka Streams with Scala!"