Intro to Akka Streams

Mitch McCuiston

Topics

- Background on Akka and Akka Streams
- Akka Streams concepts
- API examples

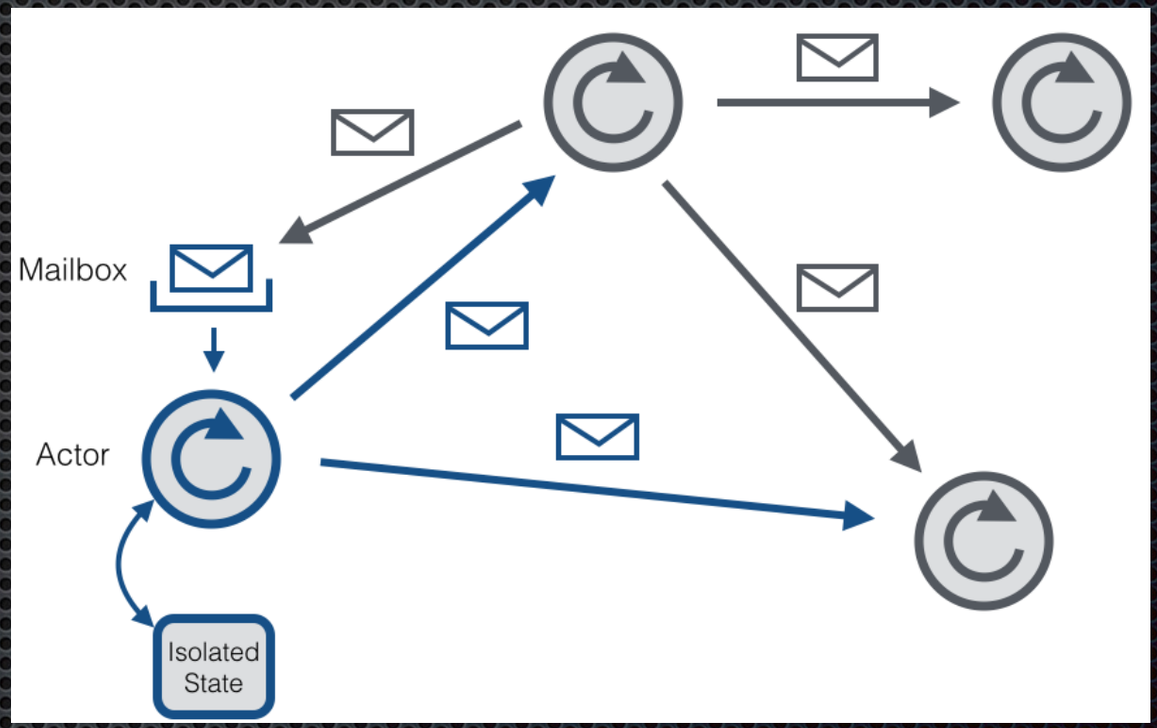
Akka

- Actor model for the JVM
- http://akka.io

Type to enter a caption.

Actor

- The only way to interact with an actor is via message
- Sending a message is fire and forget - non blocking
- No built-in ACK



Actors for streaming

- Actors seem to be a natural fit
  - Everything is a message
  - Easy to distribute
  - Async + non-blocking = good concurrency
- There is one gotcha!

Producer-consumer problem

- Actors buffer inbound messages - mailbox
- What if a producer is faster than consumer?
  - OutOfMemoryException

Back-pressure

- Feedback loop of ‘demand’ from consumer to producer
- Possible to roll your own solution
  - Need custom protocol using ACKs
  - Time consuming and error prone

Reactive Streams

- An initiative between engineers at Netflix, Pivotal, and Lightbend to standardize (2013)
- Standardize streaming interfaces
  - Asynchronous
  - Non-blocking
  - **Back-pressure**
- Incorporated into JDK 9 (2017)

Reactive Streams

- Subscriber / Publisher
- Subscriber communicates demand (amount of data) via Subscription
- If there is no demand, no message is pushed

Reactive Streams

- Low-level streaming primitives
- Even moderately complex streams become challenging to implement

Akka Streams

- Uses Reactive Streams interfaces internally
- Interoperable with other Reactive Streams implementations
  - RxJava
  - Project Reactor
- End-user focused DSL for building streams

Akka Streams

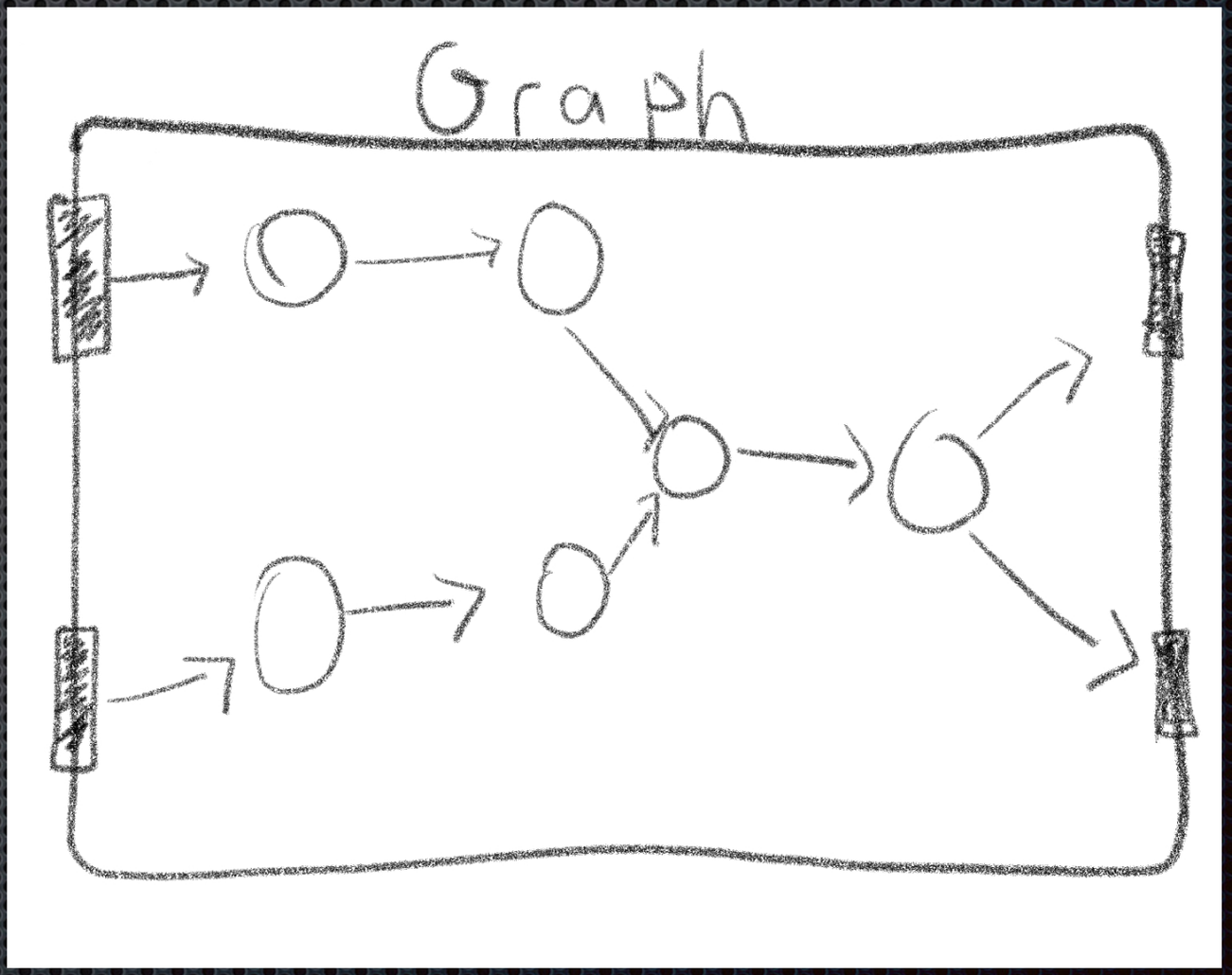
- Streams are expressed as a graph of operators
- Operators apply back pressure to upstream operators
- Back-pressure semantics are customizable

Getting Started

- Define data flow as a **Graph** having **Inlets/Outlets**
- Connect **Inlets/Outlets** making the **Graph** runnable
- Run it!

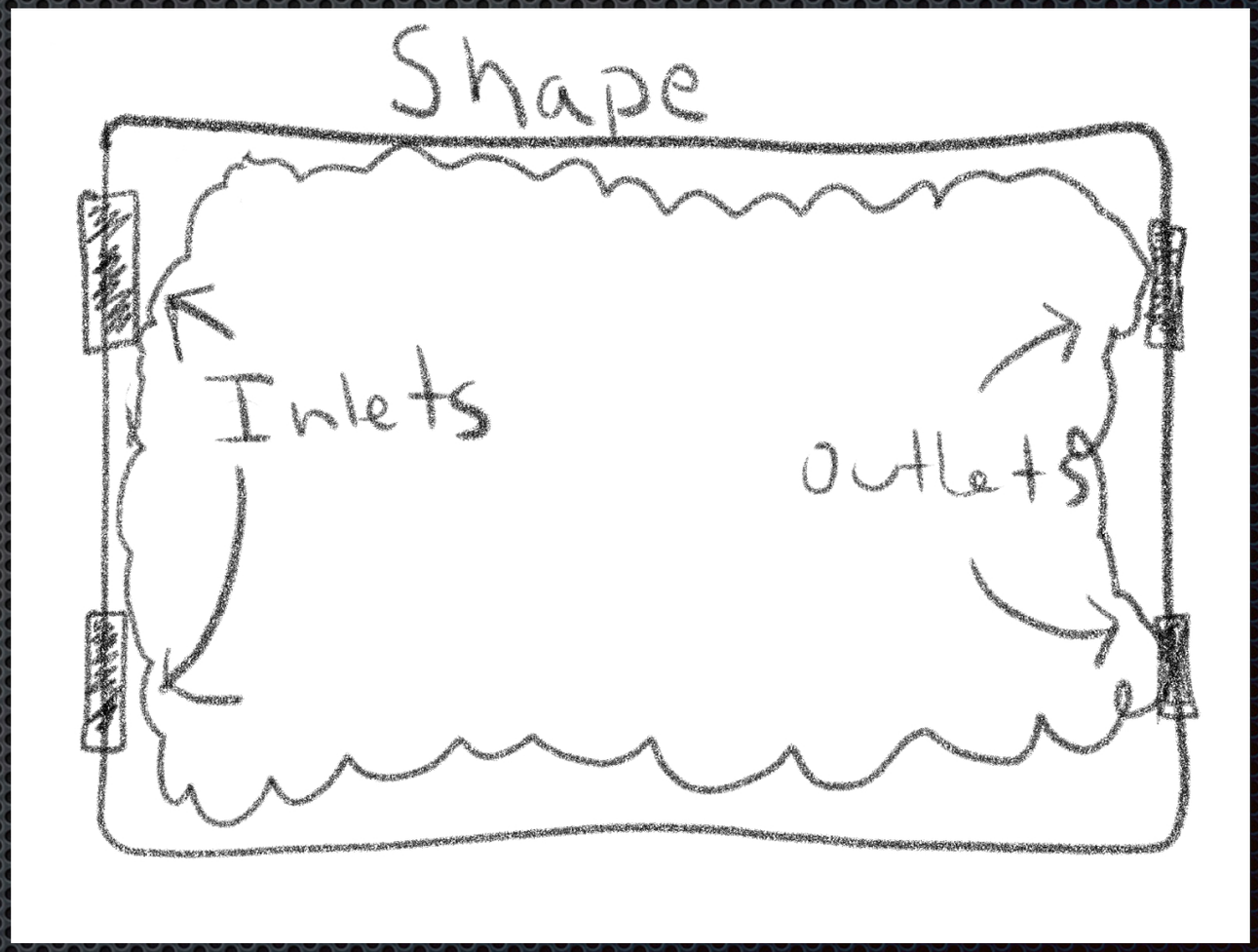
Graph

- Immutable “execution plan” of a stream
- Builder APIs
  - Fluent DSLs
  - GraphDSL



Shape

- trait Graph[Shape , _]
- Graph “Interface”
- Inlets and Outlets

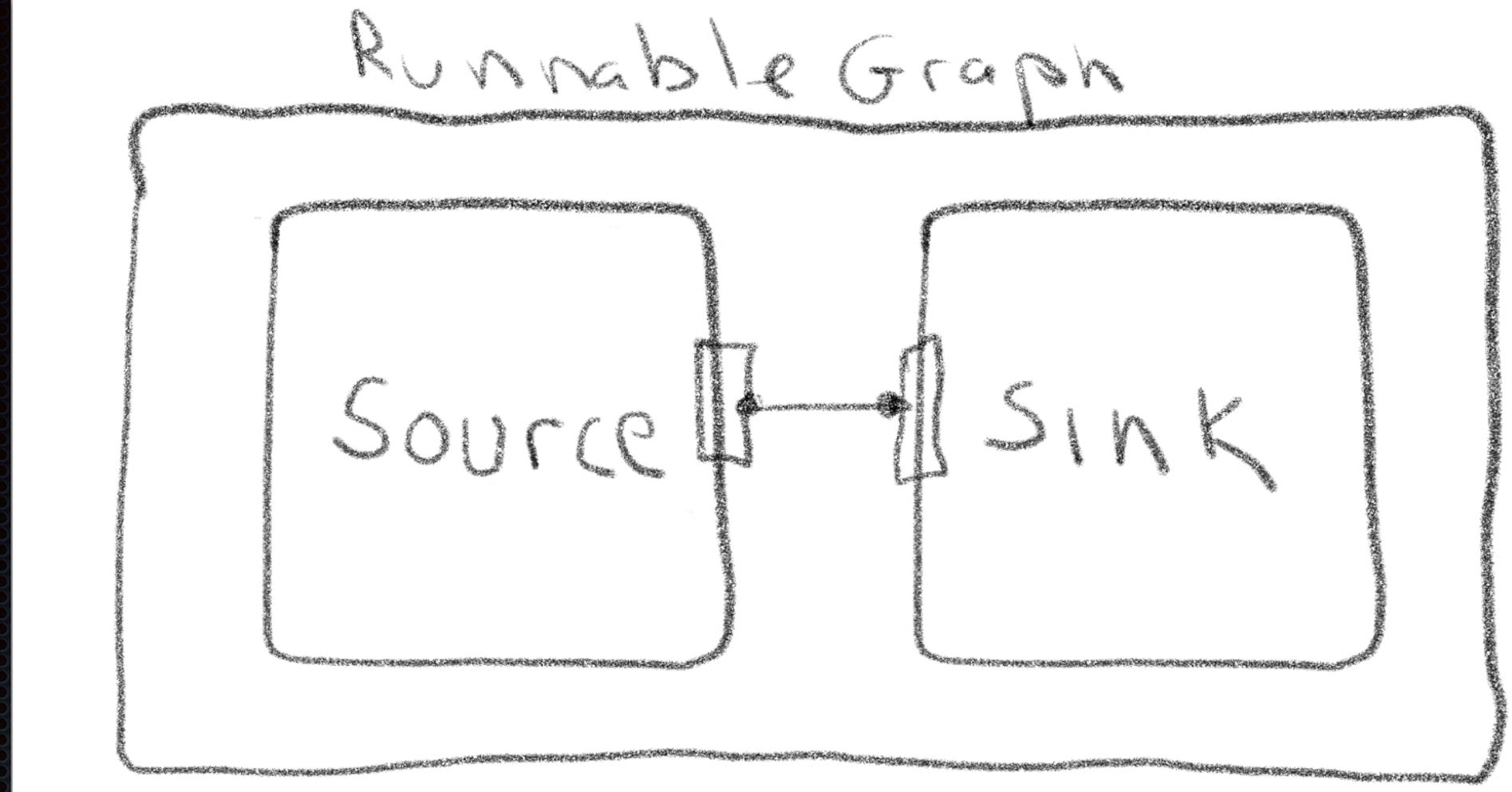


Common Graphs

	InletCount	Outlet Count
Source	0	1
Sink	1	0
Flow	1	1
Runnable Graph	0	0

Combing Graphs

- Graphs composable into larger Graphs
- Outlets connect to Inlets
- For a graph to run - ALL Inlets/Outlets must be connected



Combining Graphs

Combining Sink and Source

Example

Flow

- Graph [FlowShape, _]
- 1 Outlet
- 1 Inlet
- Independent from Source and Sink

Flow Example

FlowOps

- Operations available on Flow and Source

map	filter	mapAsync	mapAsyncUnordered
collect	recover	wireTap	buffer
throttle	sliding	concat	divertTo
to	via	take	limit
alsoTo	takeWhile	merge	flatMapMerge
sliding	fold	groupBy	flatMapConcat

mapAsync

- def mapAsync[T](par: Int)(f: Out => Future[T]): Flow[T]
- Executes async function for each element
- Future result emitted
- Perfect for DAO/Webservice calls

throttle

- def throttle(elems: Int, per: FiniteDuration): Flow[Out]

- Limits flow over time

Materialization

- Materializer interprets Graph into running stream
- Triggered by ‘terminal operations’
  - run, runWith, runForEach
- May result in “Materialized Value”

Materialized Value

- Graph[_ , **+Mat**]
- “Side channel” to interact with running stream
- Some Graphs have no materialized value - Unused
- Materialized values can be combined when Graphs are combined

Materialized Value Example

Combining Materialized Values

- ‘to’ and ‘via’ operators
  - Returns LHS Materialized value
  - Ignores RHS
- ‘toMat’ and ‘viaMat’
  - Must specify which materialized value to keep
  - Keep.{left, right, both, none}

Combining Materialized Values

KillSwitch Example

Alpakka

- Everything as a Stream!
  - Kafka
  - HTTP
  - DynamoDB
  - …

Resources

- https://github.com/mmccuiston/akka-stream-demo

Resources

Thanks!