akka streams

streaming data transformation á la carte

√

Deputy CTO

Typesafe

#protip

Think of “the concept of streams” as

- ephemeral, time-dependent, sequences of elements
- possibly unbounded in length
- in essence: transformation & transportation of data

«You cannot step twice into ***the same*** *stream.*
For as you are stepping in, other waters are ever
flowing on to you.» — Heraclitus

- Simple **message-oriented** programming model for building **Reactive** applications
- Usable from both **Java** and **Scala**
- Raised abstraction levels
  - Never think in terms of shared state, memory visibility, threads, locks, concurrent collections, thread notifications
  - High CPU utilization, low latency, high throughput, and elasticity as result
- Applications are made resilient through supervisor hierarchies

akka actors

- 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 ~450bytes
  - Run millions of actors on commodity hardware
- Akka Cluster currently handles ~2500 nodes

> *« 2500 nodes × millions of actors per GB RAM = a lot»*
>
> *— √*

akka
streams

immutable

REUSABLE

composable

coordinated

asynchronous

transformations

Flows

akka streams: Linear transformations

- Time-Agnostic
  - map, mapConcat, filter, collect, grouped, drop, take, groupBy, …
- Time-Sensitive
  - takeWithin, dropWithin, groupedWithin, …
- Rate-Detached
  - expand, conflate, buffer, …
- Asynchronous
  - mapAsync, mapAsyncUnordered, …

Sources

akka streams: Sources

- `org.reactivestreams.Publisher[T]`
- `() => Iterator[T] / immutable.Iterable[T]`
- `scala.concurrent.Future[T]`
- `actorPublisher / subscriber / actorRef`
- `single/empty/failed/timer/…`
- …or create your own!

Sinks

akka streams: Sinks

- org.reactivestreams.Subscriber[T]
- foreach / fold / onComplete
- actorSubscriber / actorRef /
- ignore / publisher / fanoutPublisher / head / cancelled / …
- … or create your own!

Fan-In

&

Fan-Out

akka streams: Nonlinear transformations

- `merge`
- `mergePreferred`
- `concat`
- `zip & zipWith`
- ... or create your own!

- `broadcast`
- `route`
- `balance`
- `unzip`
- ... or create your own!

Fan-tastic!

akka streams: Nonlinear transformations

- BidiFlow
- FlowGraph.Builder
- Custom Stages
- Coming: Octopus (“Kraken”) / N:M-way
- … and more!

OI

akka streams: Output & Input

- `Akka Http`
- `Akka Tcp Stream`
- `InputStreamSource & OutputStreamSink`
- `Reactive Streams interop`
- … create some of your own!

Materialization

akka streams: Materialization

- Akka Streams separate the ***what*** from the ***how***
  - declarative Source/Flow/Sink DSL to create a **blueprint**
  - **ActorFlowMaterializer** turns this into running Actors
- enables customizable materialization strategies
  - optimization
  - verification / validation
  - distributed deployment
- only Akka Actors (for now)

live

demo

time

Klang's

conjecture

«If you cannot solve a problem **without** programming;
you cannot solve a problem **with** programming.»

Getting **data** *across* an **asynchronous** boundary

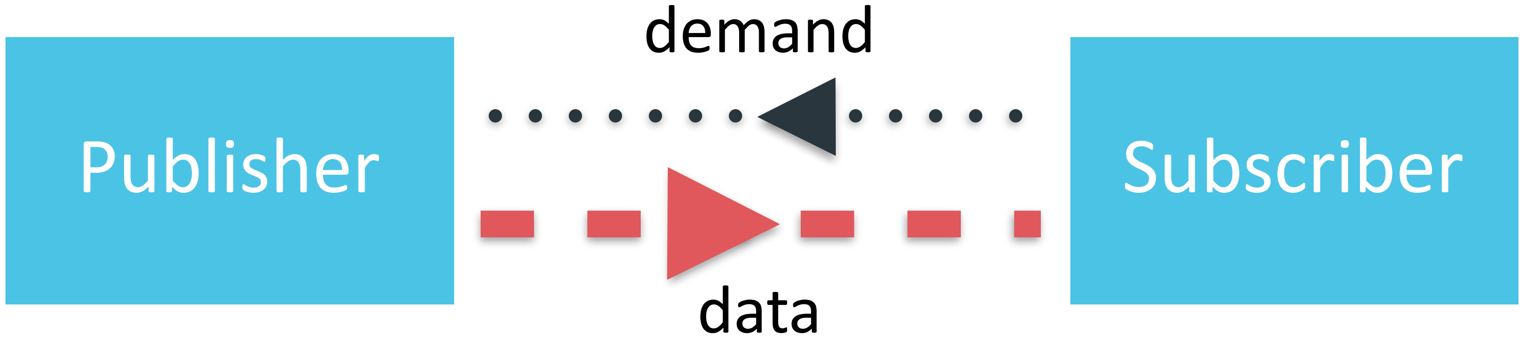
AL
1999

Getting **data** across
an **asynchronous**
boundary
with ***non***-*blocking*
back pressure

Comparing Push vs Pull

Requirements	Push	Pull
support potentially unbounded sequences	:)	:)
sender runs separately from receiver	:)	:)
rate of reception may vary from rate of sending	:)	:)
dropping elements should be a choice and not a necessity	:(!	:)
minimal (if any) overhead in terms of latency and throughput	:)	:(!

Supply & Demand


demand
Publisher
Subscriber
data

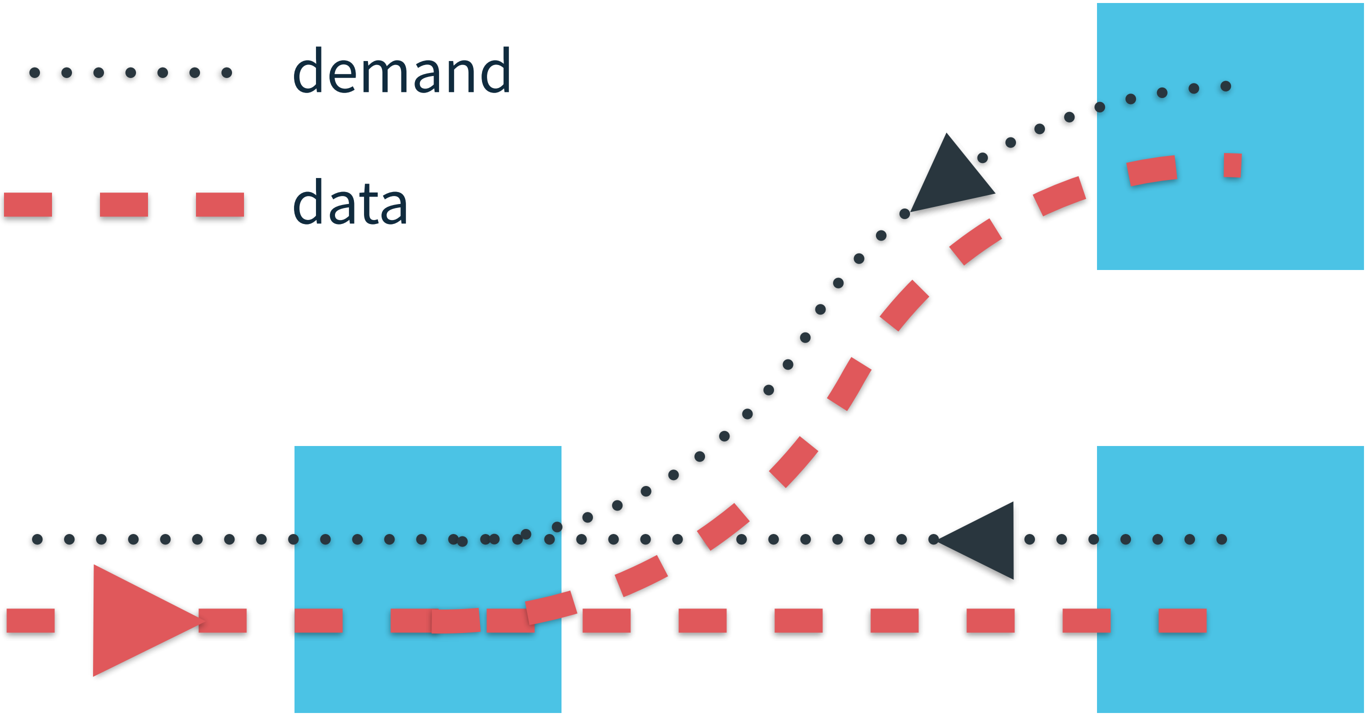
- “*push*” when *subscriber* is faster
- “*pull*” when *publisher* is faster
- switches **automatically** between both
- batching demand allows batching ops

Dynamic Push–Pull

Comparing Push vs Pull vs Both

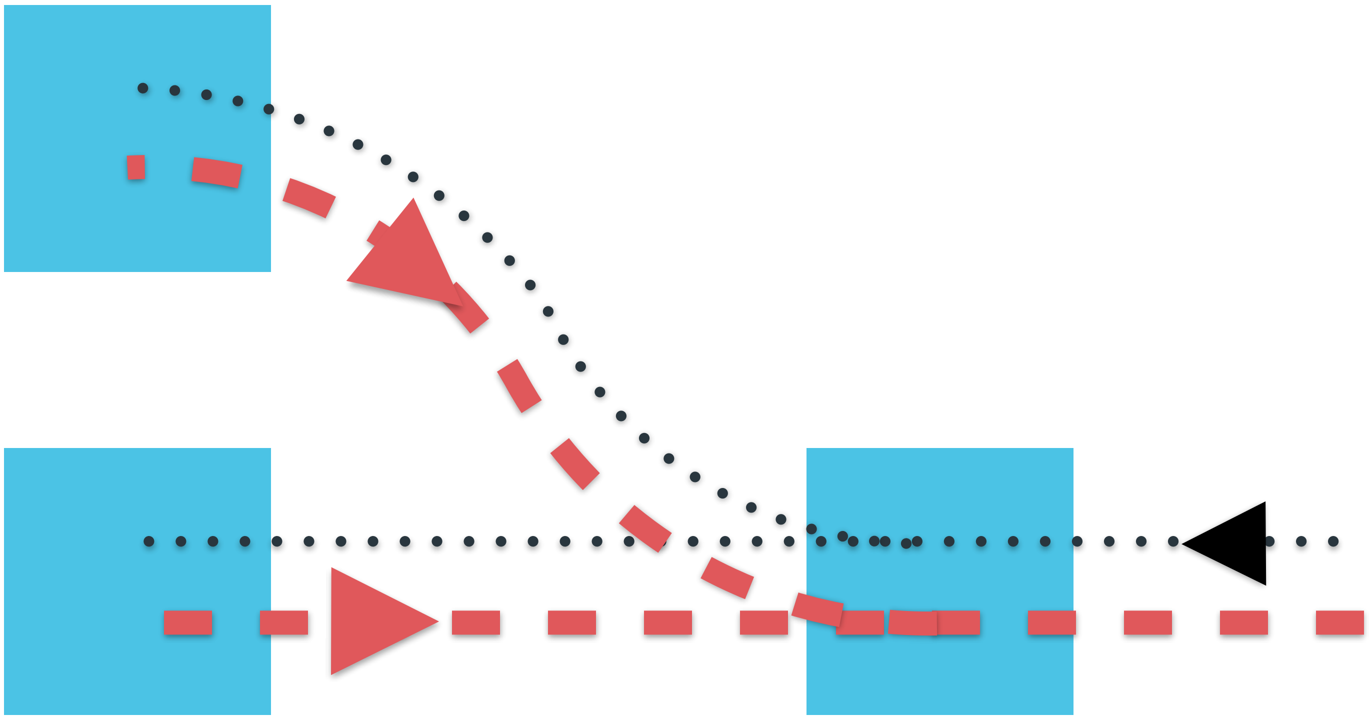
Requirements	Push	Pull	Both
support potentially unbounded sequences	:)	:)	:)
sender runs separately from receiver	:)	:)	:)
rate of reception may vary from rate of sending	:)	:)	:)
dropping elements should be a choice and not a necessity	:(	:)	:)
minimal (if any) overhead in terms of latency and throughput	:)	:(	:)

Stream splitting



splitting the data means merging the demand

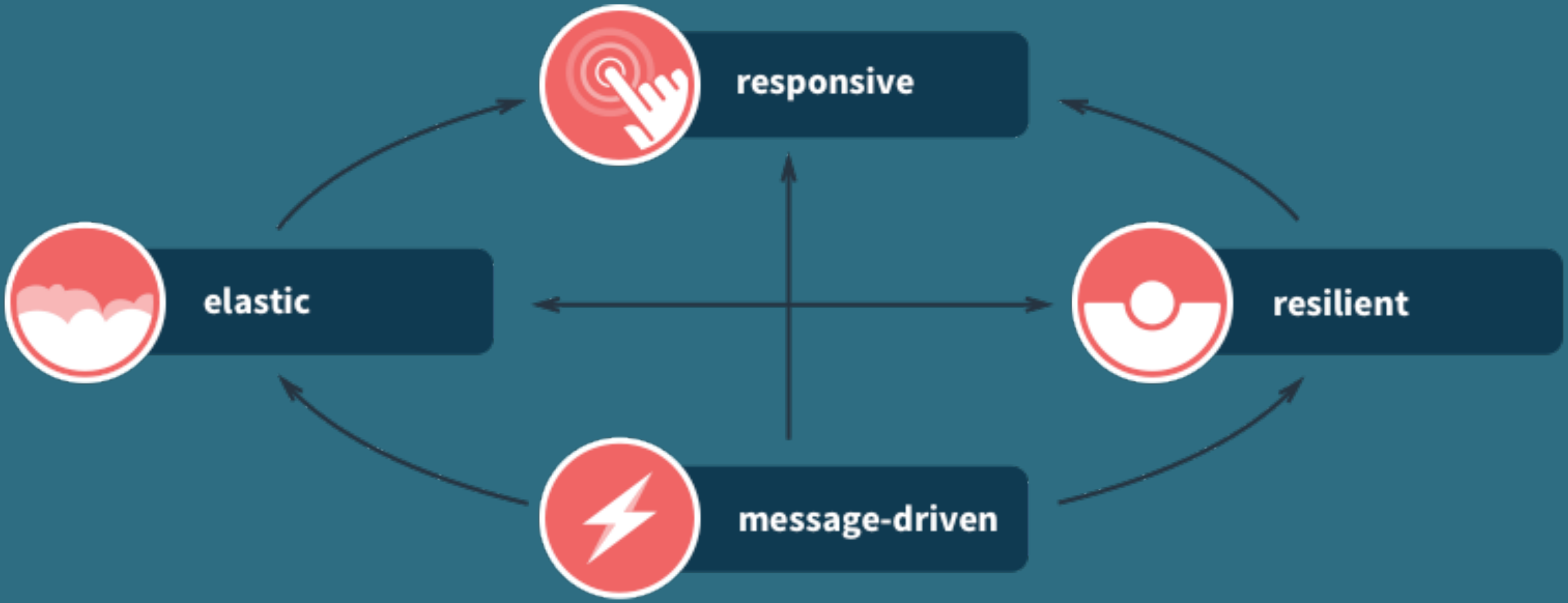
Stream merging



merging the data means splitting the demand

THE Reactive Streams Initiative

The traits of Reactive



responsive

elastic

resilient

message-driven

- **define** minimal interfaces—essentials only
- **outline** rigorous specification of semantics
- **create** a TCK for verification of implementation
- **ensure** complete freedom for many idiomatic APIs
- **verify** that the specification is efficiently implementable

> *«**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.

Exciting

Opportunities

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 elision

- Compile-time, using Scala Macros
  - fold ++ take(n where n > 0) == fold
  - drop(0) == identity
  - <any> ++ identity == <any>
- Run-time, using intra-stage simplification
  - map ++ dropUntil(cond) ++ take(N)
  - map ++ identity ++ take(N)
  - map ++ take(N)

Opportunity: Operation fusion

- Compile-time, using Scala Macros
  - filter ++ map == collect
- Run-time, using intra-stage simplification
  - Rule: <any> ++ identity == <any>
    Rule: identity ++ <any> == <any>
  - filter ++ dropUntil(cond) ++ map
  - filter ++ identity ++ map == collect

Opportunity: Execution optimization

- synchronous intra-stage execution N steps then trampoline and/or give control to other Thread / Flow

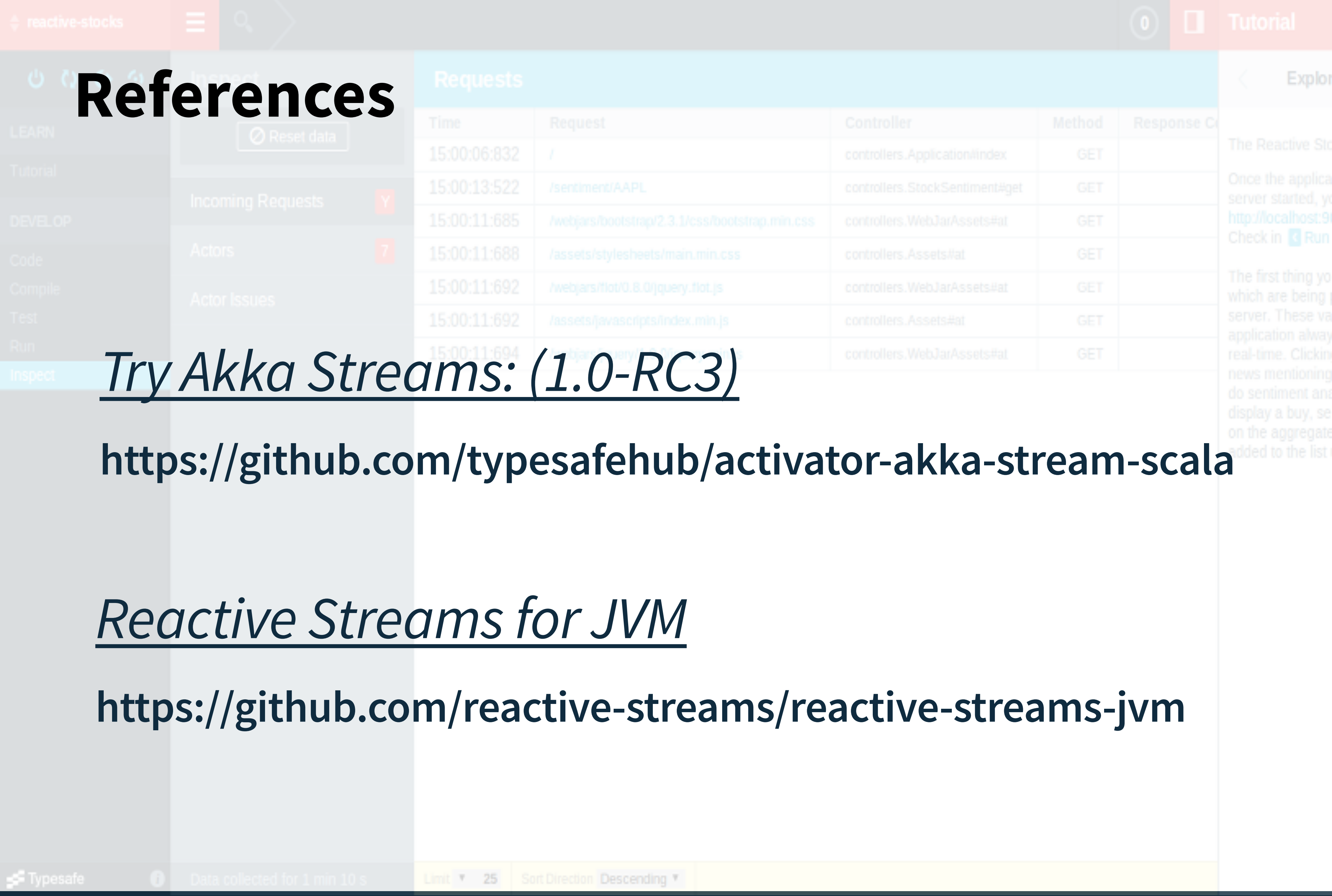
References

Try Akka Streams: (1.0-RC3)

https://github.com/typesafehub/activator-akka-stream-scala

Reactive Streams for JVM

https://github.com/reactive-streams/reactive-streams-jvm



Time	Request	Controller	Method	Response C
15:00:06:832	/	controllers.Application#index	GET	
15:00:13:522	/sentiment/AAPL	controllers.StockSentiment#get	GET	
15:00:11:685	/webjars/bootstrap/2.3.1/css/bootstrap.min.css	controllers.WebJarAssets#at	GET	
15:00:11:688	/assets/stylesheets/main.min.css	controllers.Assets#at	GET	
15:00:11:692	/webjars/flot/0.8.0/jquery.flot.js	controllers.WebJarAssets#at	GET	
15:00:11:692	/assets/javascripts/index.min.js	controllers.Assets#at	GET	
15:00:11:694	[illegible]	controllers.WebJarAssets#at	GET	

The Reactive Sto

Once the applica
server started, yo
http://localhost:9
Check in Run

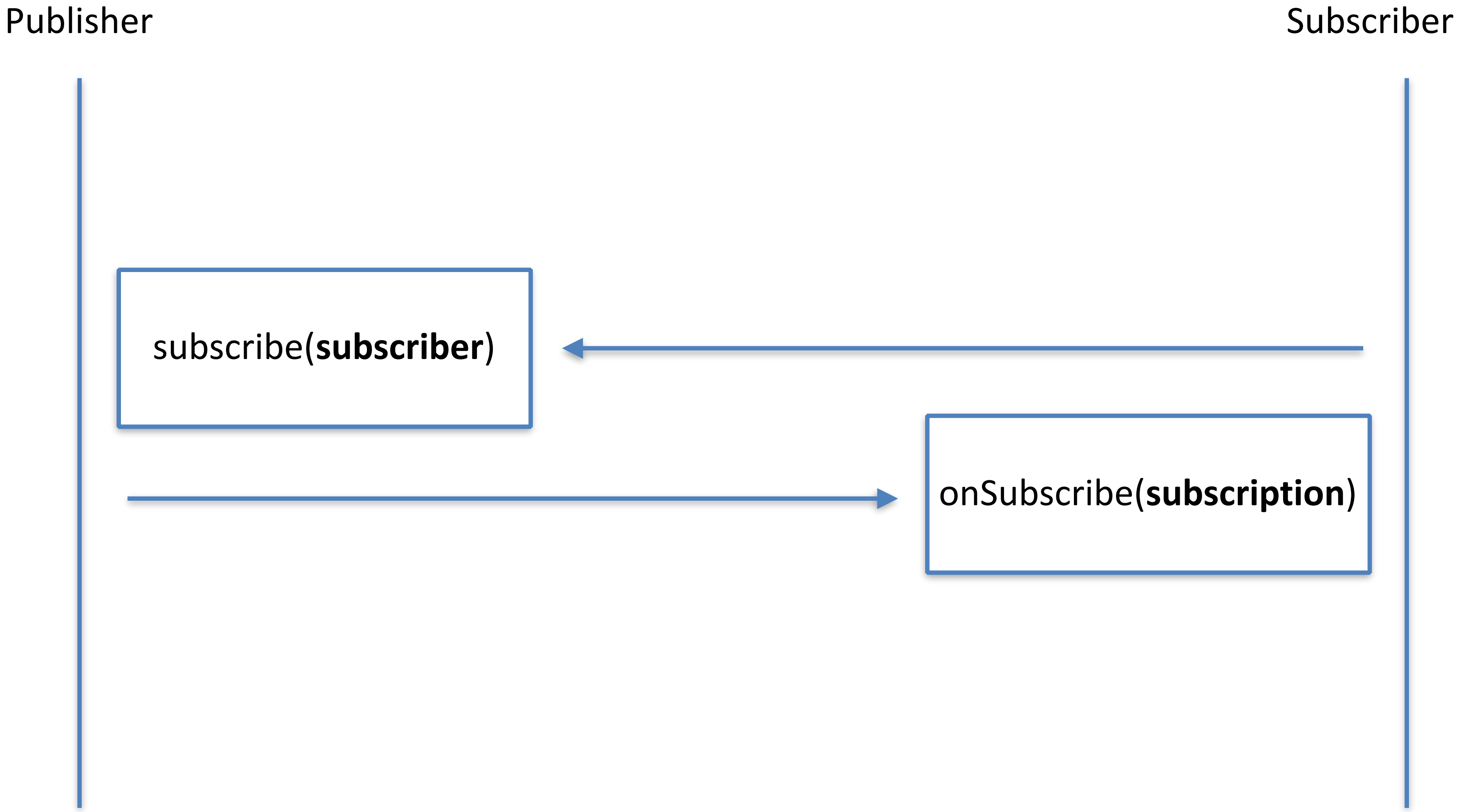
The first thing yo
which are being
server. These va
application alway
real-time. Clickin
news mentioning
do sentiment ana
display a buy, se
on the aggregate
added to the list

Typesafe

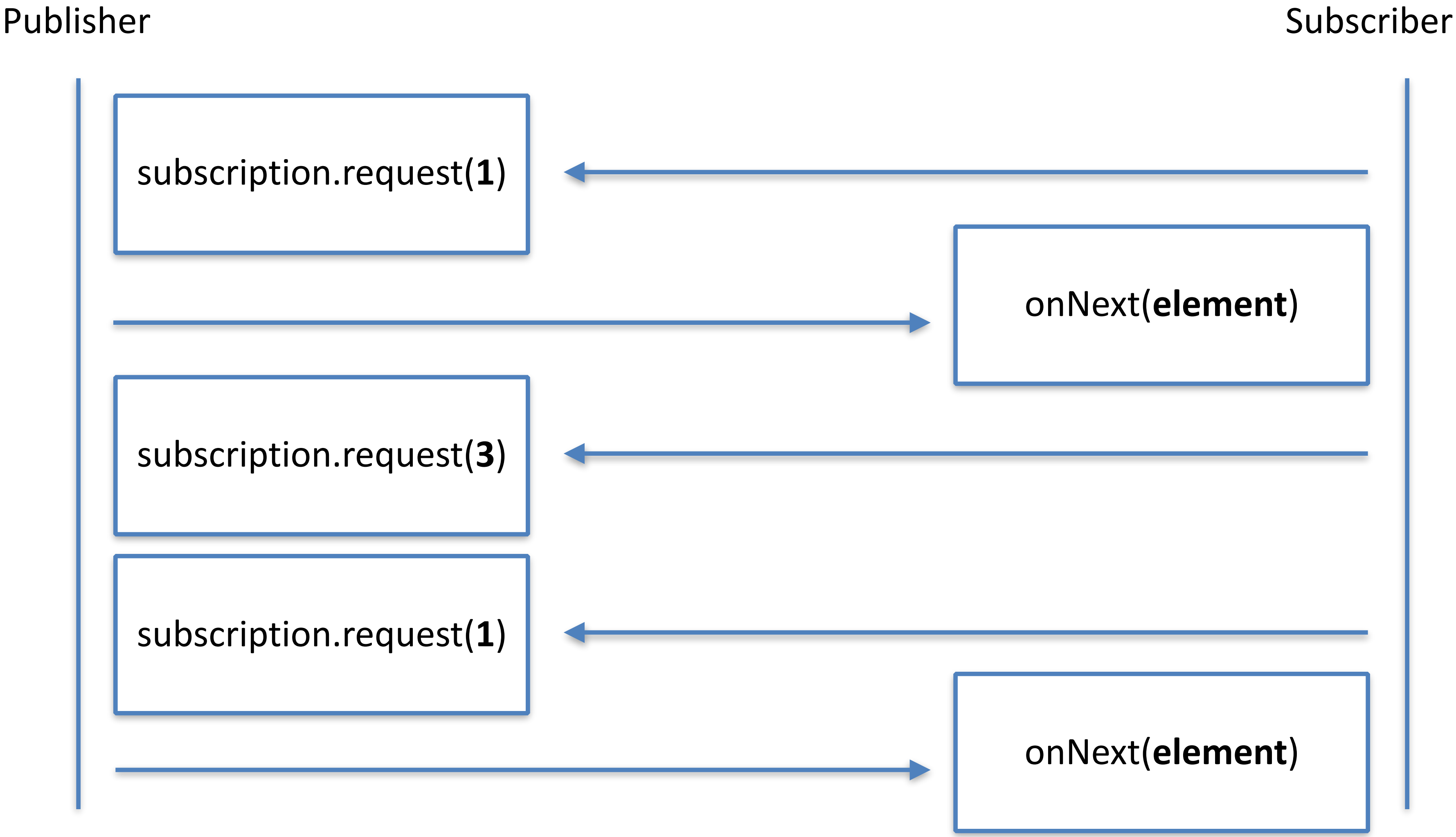
Reactive Streams protocol

```
public interface Publisher<T> {
  public void subscribe(Subscriber<T> s);
}
public void Subscription {
  public void request(long 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();
}
public interface Processor<T, R>
  extends Subscriber<T>, Publisher<R> { }
```

How does it connect?



How does data flow?



Publisher

Subscriber

subscription.request(**1**)

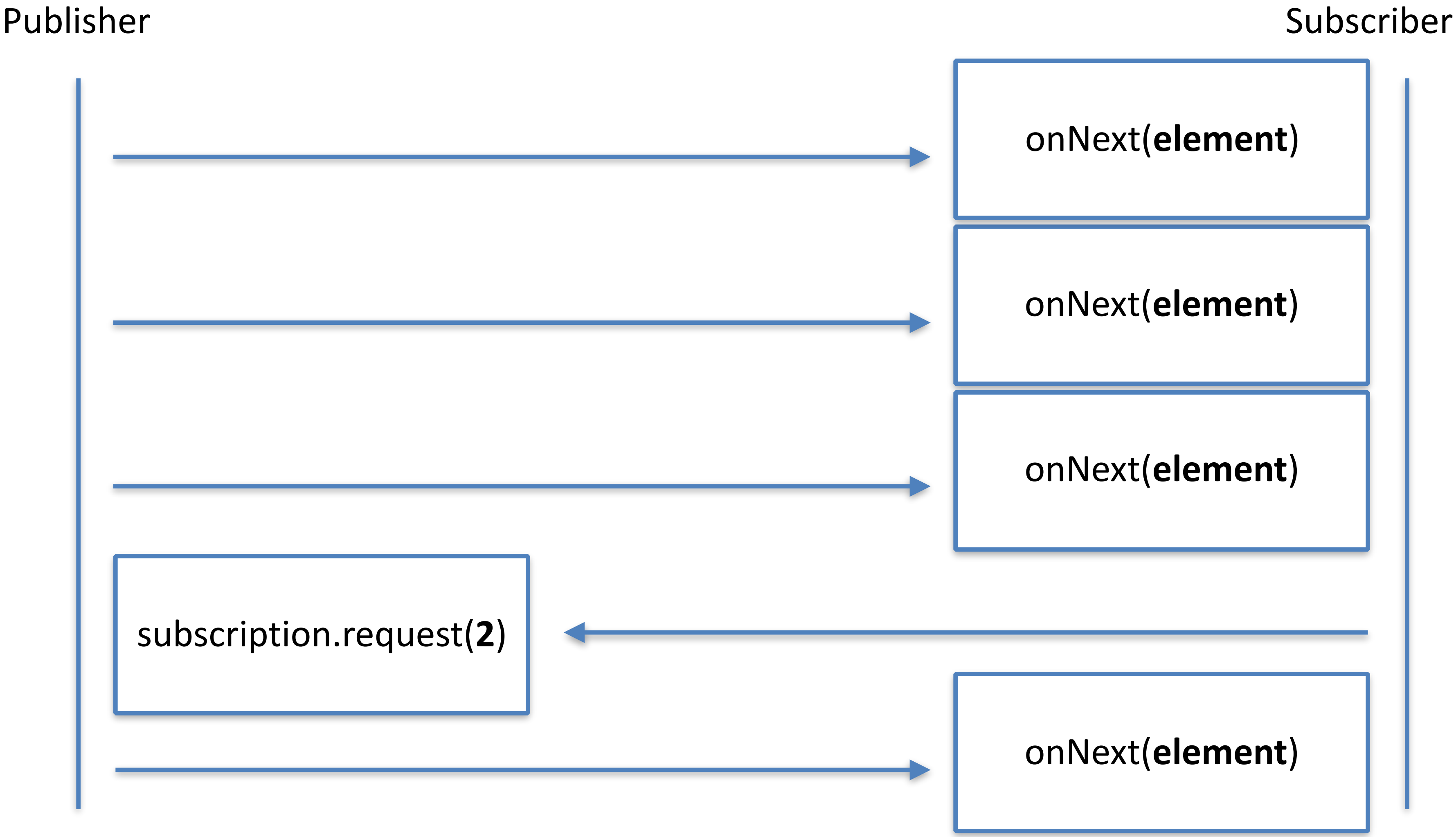
onNext(**element**)

subscription.request(**3**)

subscription.request(**1**)

onNext(**element**)

How does data flow?



Publisher

Subscriber

onNext(**element**)

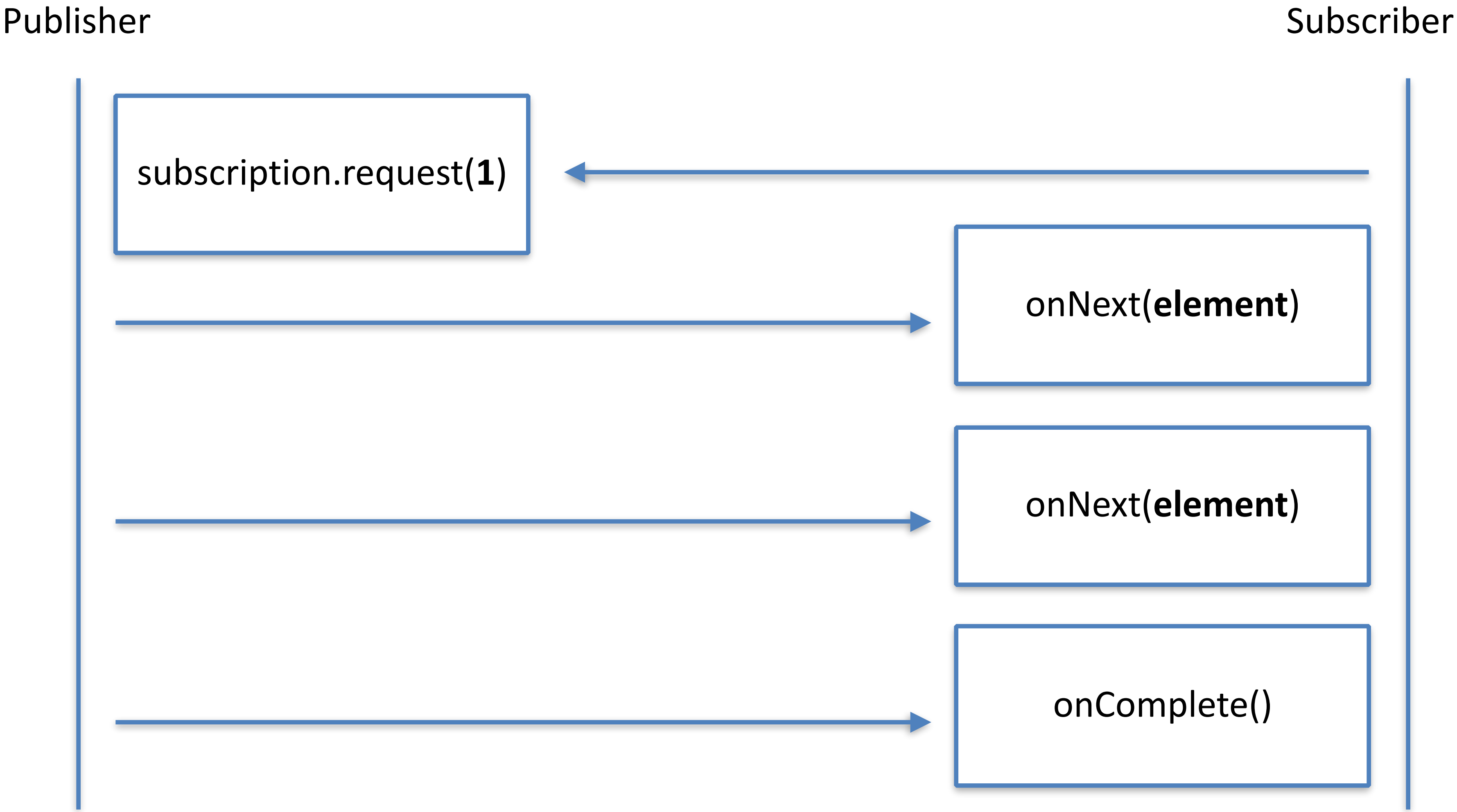
onNext(**element**)

onNext(**element**)

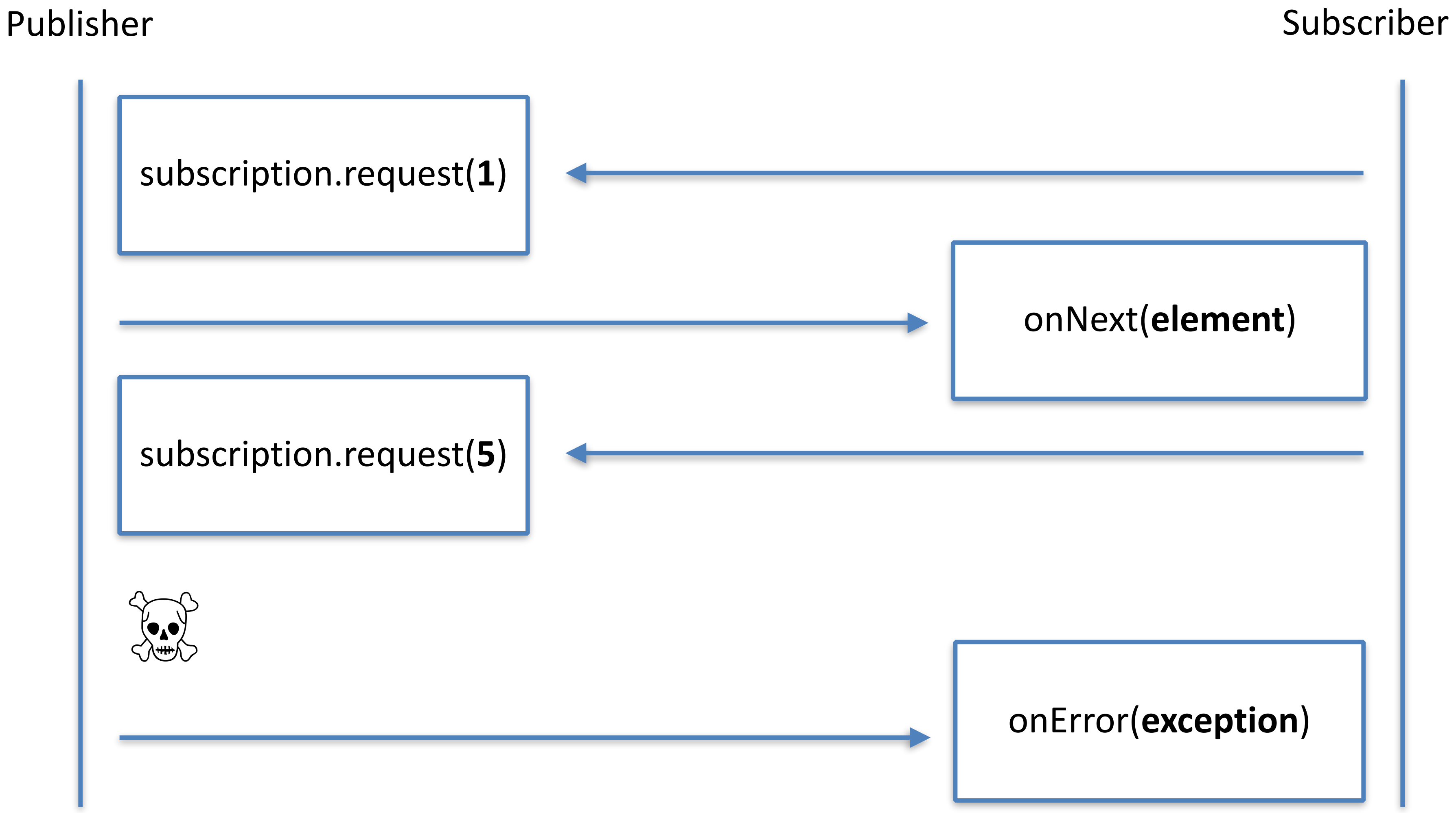
subscription.request(**2**)

onNext(**element**)

How does it complete?



What if it fails?



Publisher

Subscriber

subscription.request(**1**)

onNext(**element**)

subscription.request(**5**)

☠

onError(**exception**)

live

demo

time