Reactive Streams

Handling Data-Flows the Reactive Way

Dr. Roland Kuhn
Akka Tech Lead
@rolandkuhn

Introduction: Streams

What is a Stream?

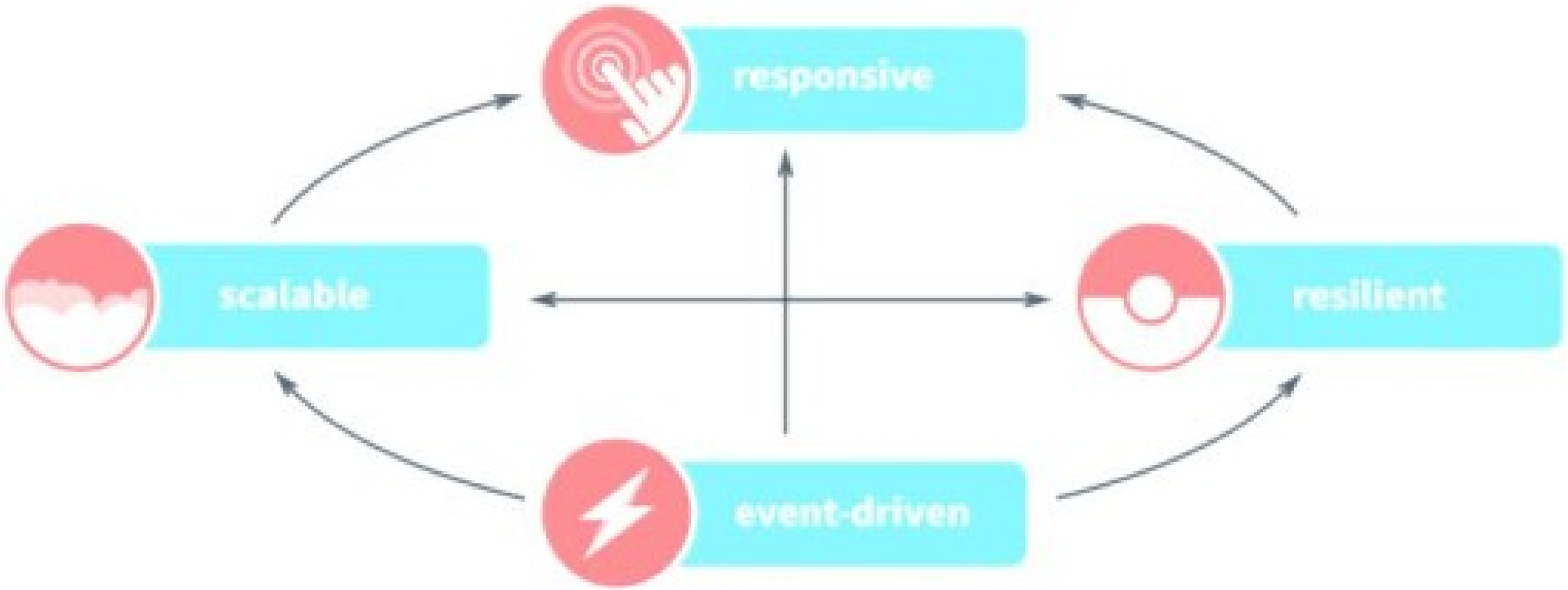
- ephemeral flow of data
- focused on describing transformation
- possibly unbounded in size

Common uses of Streams

- bulk data transfer
- real-time data sources
- batch processing of large data sets
- monitoring and analytics

What is special about Reactive Streams?

The Four Reactive Traits



http://reactivemanifesto.org/

Needed: Asynchrony

- Resilience demands it:
  - encapsulation
  - isolation
- Scalability demands it:
  - distribution across nodes
  - distribution across cores

Needed: Asynchrony

- Resilience demands it:
  - encapsulation
  - isolation
- Scalability demands it:
  - distribution across nodes
  - distribution across cores

Many Kinds of Async Boundaries

Many Kinds of Async Boundaries

- between different applications

Many Kinds of Async Boundaries

- between different applications
- between network nodes

Many Kinds of Async Boundaries

- between different applications
- between network nodes
- between CPUs

Many Kinds of Async Boundaries

- between different applications
- between network nodes
- between CPUs
- between threads

Many Kinds of Async Boundaries

- between different applications
- between network nodes
- between CPUs
- between threads
- between actors

The Problem:

Getting Data across an **Async Boundary**

Possible Solutions

Possible Solutions

- the Traditional way: blocking calls

Possible Solutions

Possible Solutions

- the Push way: buffering and/or dropping

Possible Solutions

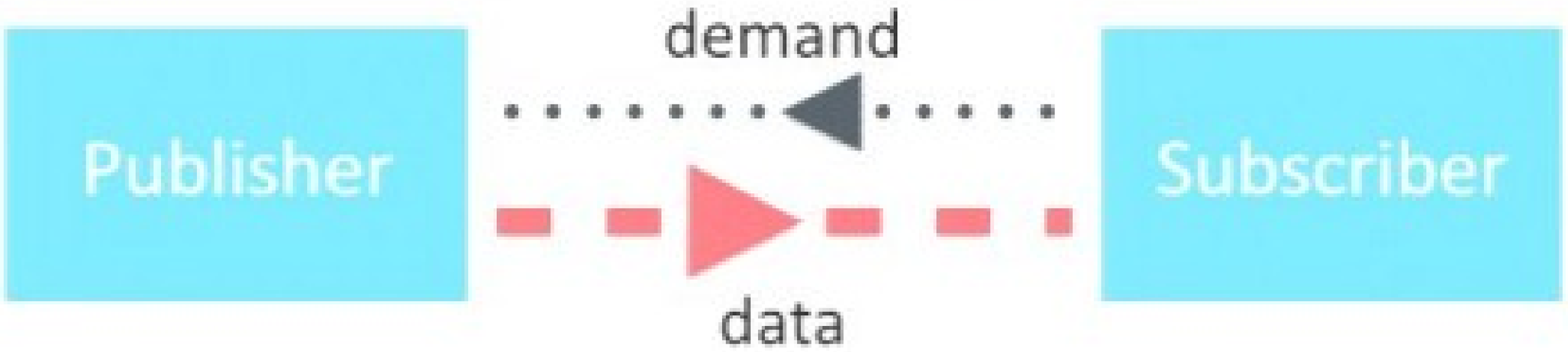
Possible Solutions

- the Reactive way:
  non-blocking & non-dropping & bounded

How do we achieve that?

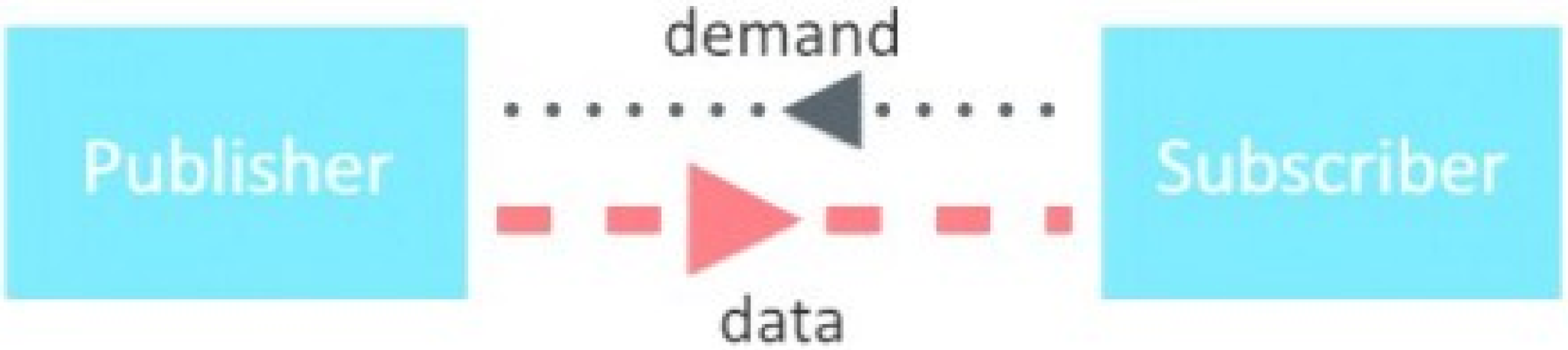
Supply and Demand

- data items flow downstream
- demand flows upstream
- data items flow only when there is demand
  - recipient is in control of incoming data rate
  - data in flight is bounded by signaled demand

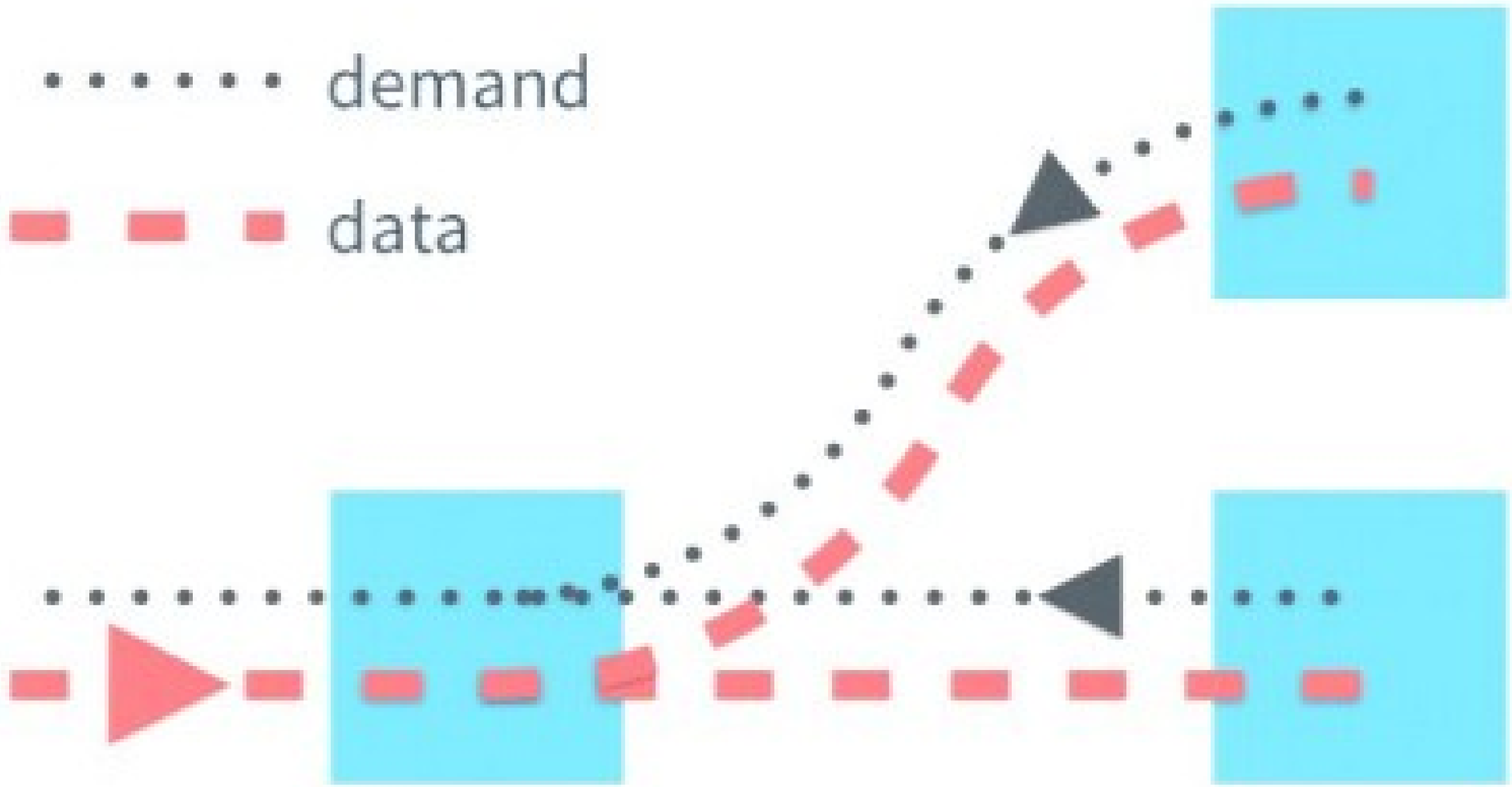


Dynamic Push–Pull

- “push” behavior when consumer is faster
- “pull” behavior when producer is faster
- switches automatically between these
- batching demand allows batching data

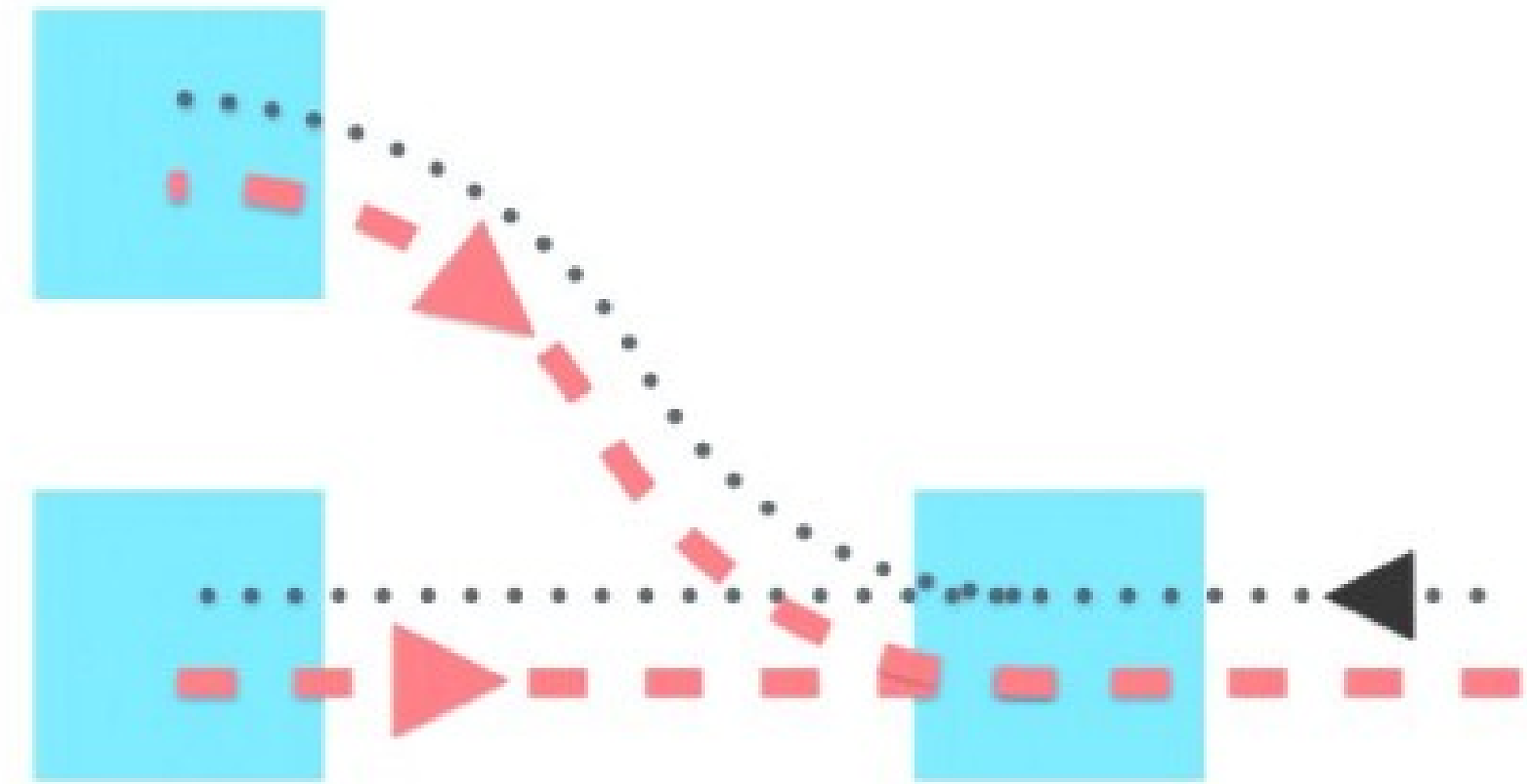


Explicit Demand: Tailored Flow Control



splitting the data means merging the demand

Explicit Demand: Tailored Flow Control



merging the data means splitting the demand

Reactive Streams

- asynchronous non-blocking data flow
- asynchronous non-blocking demand flow
- minimal coordination and contention
- message passing allows for distribution
  - across applications
  - across nodes
  - across CPUs
  - across threads
  - across actors

Are Streams Collections?

What is a Collection?

What is a Collection?

- Oxford Dictionary:
  - "a group of things or people"

What is a Collection?

- Oxford Dictionary:
  - "a group of things or people"
- wikipedia:
  - "a grouping of some variable number of data items"

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"

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)
- `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)

Streams have Unexpected Properties

- the observed sequence depends on
  - … when the observer subscribed to the stream
  - … whether the observer can process fast enough
  - … whether the streams flows fast enough

Streams are *not* Collections!

- java.util.stream:
  Stream is not derived from Collection
  "Streams differ from Coll's in several ways"
  - no storage
  - functional in nature
  - laziness seeking
  - possibly unbounded
  - consumable

Streams are *not* Collections!

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

So, Reactive Streams:
why not just java.util.stream.Stream?

Java 8 Stream

```
import java.util.stream.*;

// get some stream
final Stream<Integer> s = Stream.of(1, 2, 3);
// describe transformation
final Stream<String> s2 = s.map(i -> "a" + i);
// make a pull collection
s2.iterator();
// or alternatively push it somewhere
s2.forEach(i -> System.out.println(i));
// (need to pick one, Stream is consumable)
```

Java 8 Stream

- provides a DSL for describing transformation
- introduces staged computation
  (but does not allow reuse)
- prescribes an *eager* model of execution
- offers either push or pull, chosen statically

What about RxJava?

RxJava

```
import rx.Observable;
import rx.Observable.*;

// get some stream source
final Observable<Integer> obs = range(1, 3);
// describe transformation
final Observable<String> obs2 =
                              obs.map(i -> "b" + i);
// and use it twice
obs2.subscribe(i -> System.out.println(i));
obs2.filter(i -> i.equals("b2"))
    .subscribe(i -> System.out.println(i));
```

RxJava

- implements pure “push” model
- includes extensive DSL for transformations
- only allows blocking for back pressure
- currently uses unbounded buffering for crossing an async boundary
- work on distributed Observables sparked participation in Reactive Streams

The Reactive Streams Project

Participants

- Engineers from
  - Netflix
  - Oracle
  - Pivotal
  - Red Hat
  - Twitter
  - Typesafe
- Individuals like Doug Lea and Todd Montgomery

The Motivation

- all participants had 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)

see also Jon Brisbin's post on "Tribalism as a Force for Good"

Recipe for Success

- minimal interfaces
- rigorous specification of semantics
- full TCK for verification of implementation
- complete freedom for many idiomatic APIs

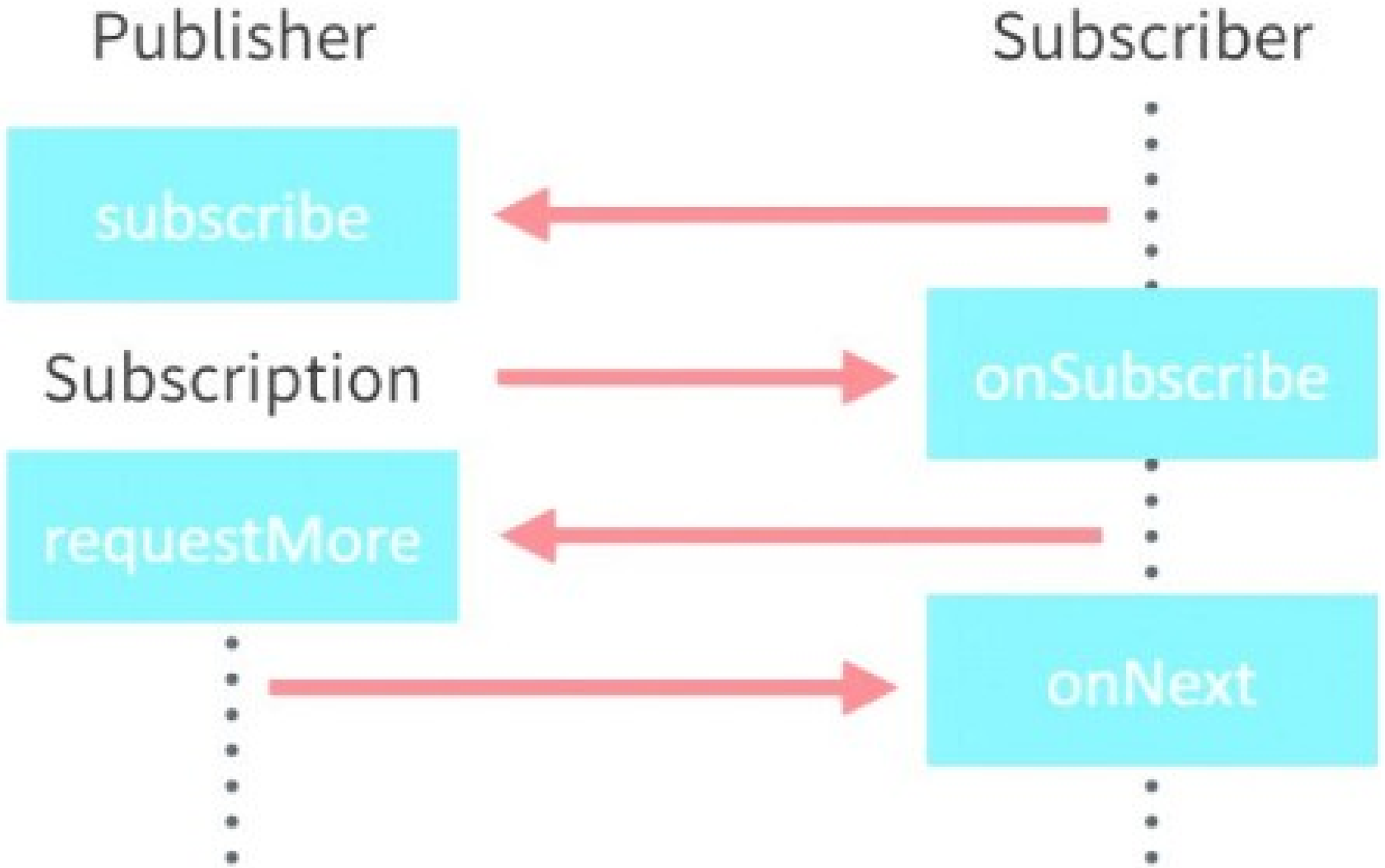
The Meat

```
trait Publisher[T] {
  def subscribe(sub: Subscriber[T]): Unit
}
trait Subscription {
  def requestMore(n: Int): Unit
  def cancel(): Unit
}
trait Subscriber[T] {
  def onSubscribe(s: Subscription): Unit
  def onNext(elem: T): Unit
  def onError(thr: Throwable): Unit
  def onComplete(): Unit
}
```

The Sauce

- all calls on Subscriber must dispatch async
- all calls on Subscription must not block
- Publisher is just there to create Subscriptions

How does it Connect?



Akka Streams

Akka Streams

- powered by Akka Actors
  - execution
  - distribution
  - resilience
- type-safe streaming through Actors with bounded buffering

Basic Akka Example

```
implicit val system = ActorSystem("Sys")
val mat = FlowMaterializer(...)

Flow(text.split("\\s").toVector).
  map(word => word.toUpperCase).
  foreach(tranformed => println(tranformed)).
onComplete(mat) {
  case Success(_) => system.shutdown()
  case Failure(e) =>
    println("Failure: " + e.getMessage)
    system.shutdown()
}
```

Basic Akka Example

```
implicit val system = ActorSystem("Sys")
val mat = FlowMaterializer(...)

Flow(text.split("\\s").toVector).
  map(word => word.toUpperCase).
  foreach(tranformed => println(tranformed)).
onComplete(mat) {
  case Success(_) => system.shutdown()
  case Failure(e) =>
    println("Failure: " + e.getMessage)
    system.shutdown()
}
```

Basic Akka Example

```
implicit val system = ActorSystem("Sys")
val mat = FlowMaterializer(...)

Flow(text.split("\\s").toVector).
  map(word => word.toUpperCase).
  foreach(tranformed => println(tranformed)).
  onComplete(mat) {
    case Success(_) => system.shutdown()
    case Failure(e) =>
      println("Failure: " + e.getMessage)
      system.shutdown()
  }
```

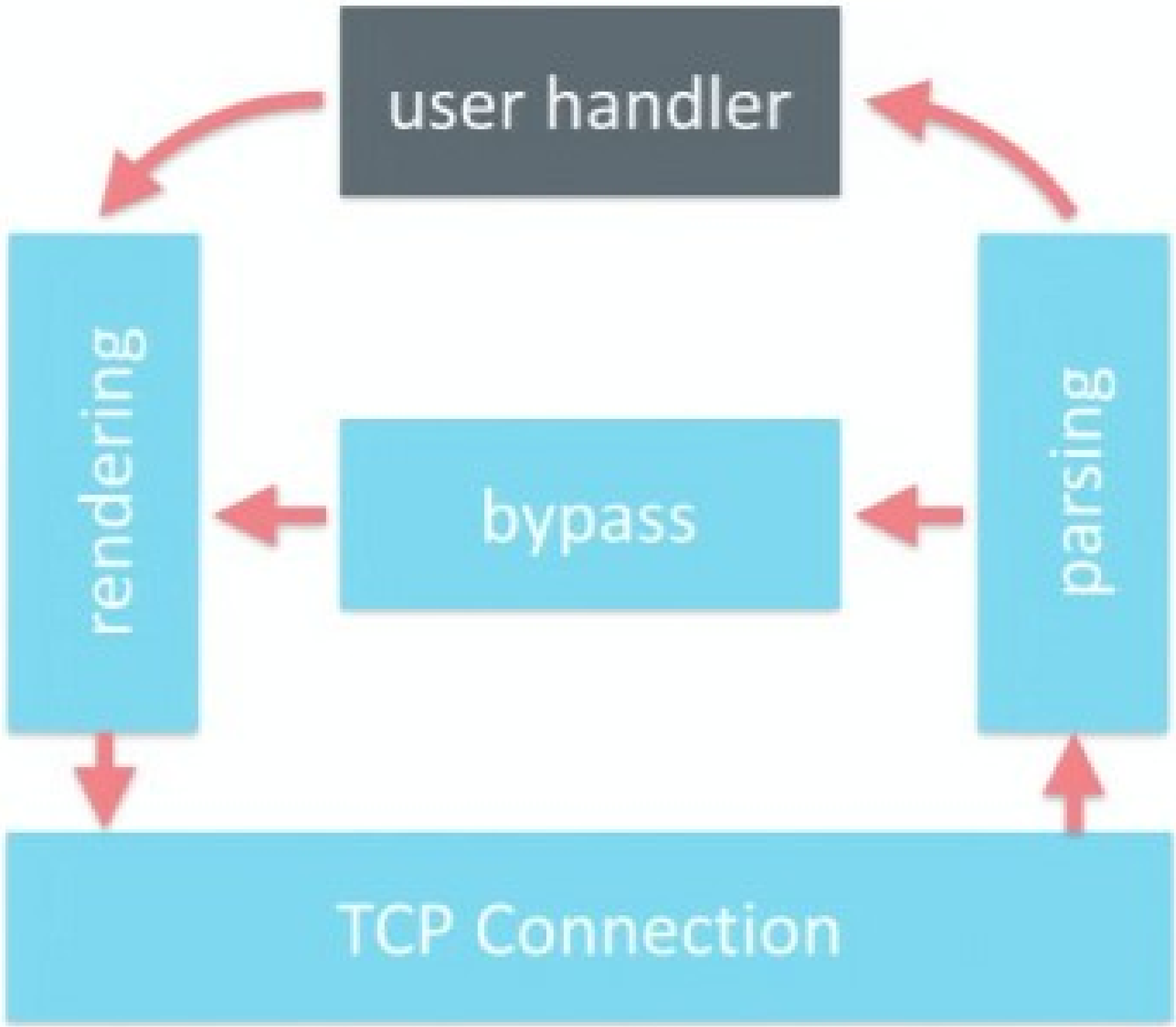
Java 8 Example

```
final ActorSystem system = ActorSystem.create("Sys");
final MaterializerSettings settings =
    MaterializerSettings.create();
final FlowMaterializer materializer =
    FlowMaterializer.create(settings, system);

final String[] lookup = { "a", "b", "c", "d", "e", "f" };
final Iterable<Integer> input = Arrays.asList(0, 1, 2, 3, 4, 5);

Flow.create(input).drop(2).take(3).  // leave 2, 3, 4
  map(elem -> lookup[elem]).         // translate to "c","d","e"
  filter(elem -> !elem.equals("c")). // filter out the "c"
  grouped(2).                        // make into a list
  mapConcat(list -> list).           // flatten the list
  fold("", (acc, elem) -> acc + elem). // accumulate into "de"
  foreach(elem -> System.out.println(elem)). // print it
  consume(materializer);
```

Akka HTTP Server Overview



Akka HTTP Server Part 1

```
val (bypassConsumer, bypassProducer) =
 Duct[(RequestOutput, Producer[RequestOutput])]
  .collect[MessageStart with RequestOutput]
    { case (x: MessageStart, _) => x }
  .build(materializer)
```

Akka HTTP Server Part 2

```
val requestProducer =
 Flow(tcpConn.inputStream)
  .transform(rootParser)
  .splitWhen(_.isInstanceOf[MessageStart])
  .headAndTail
  .tee(bypassConsumer)
  .collect {
    case (x: RequestStart, entityParts) =>
      HttpServerPipeline.constructRequest(x,
        entityParts) }
  .toProducer(materializer)
```

Akka HTTP Server Part 3

```
val responseConsumer =
 Duct[HttpResponse]
  .merge(bypassProducer)
  .transform(applyApplicationBypass)
  .transform(rendererFactory.newRenderer)
  .flatten(concat)
  .transform(logErrors)
  .toProducer(materializer)
  .produceTo(tcpConn.outputStream)
```

Akka HTTP server Part 4

```
val logErrors =
  new Transformer[ByteString, ByteString] {
    def onNext(element: ByteString) =
      element :: Nil
    override def onError(cause: Throwable) =
      log.error(cause, "Response stream error")
  }
```

Closing Remarks

Current State

- Early Preview is available:

```
"org.reactivestreams" % "reactive-streams-spi" % "0.2"
"com.typesafe.akka" %% "akka-stream-experimental" % "0.3"
```

- check out the Activator template "Akka Streams with Scala!" (https://github.com/typesafehub/activator-akka-stream-scala)

Next Steps

- we work towards inclusion in future JDK
- we aim at polyglot standard (JS, wire proto)
- try it out and give feedback!
- http://reactive-streams.org/
- https://github.com/reactive-streams