Akka Streams

Dr. Roland Kuhn
@rolandkuhn — Typesafe

Typesafe

Why Streams?

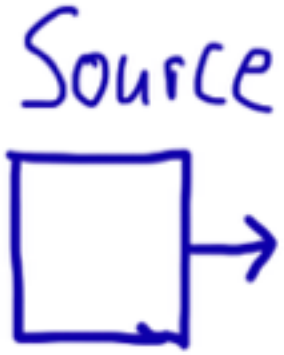
- processing big data with finite memory
- real-time data processing (CEP)
- serving numerous clients simultaneously with bounded resources (IoT, streaming HTTP APIs)

What is a Stream?

- ephemeral, time-dependent sequence of elements
- possibly unbounded in length
- therefore focusing on transformations

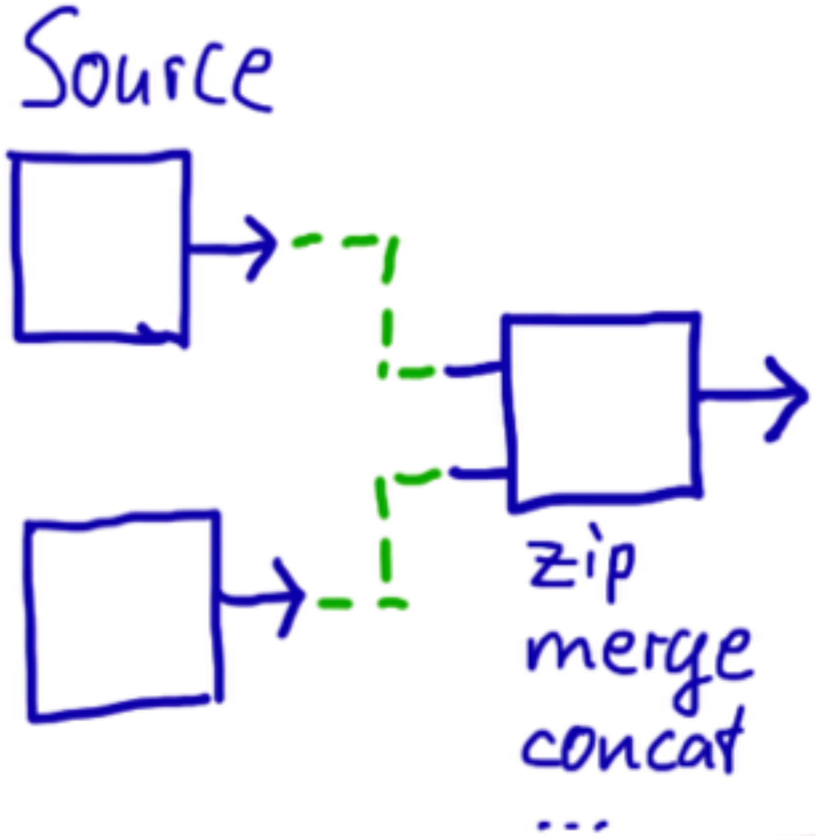
> *«You cannot step twice into the same stream. For as you are stepping in, other waters are ever flowing on to you.» — Heraclitus*

Declaring a Stream Topology

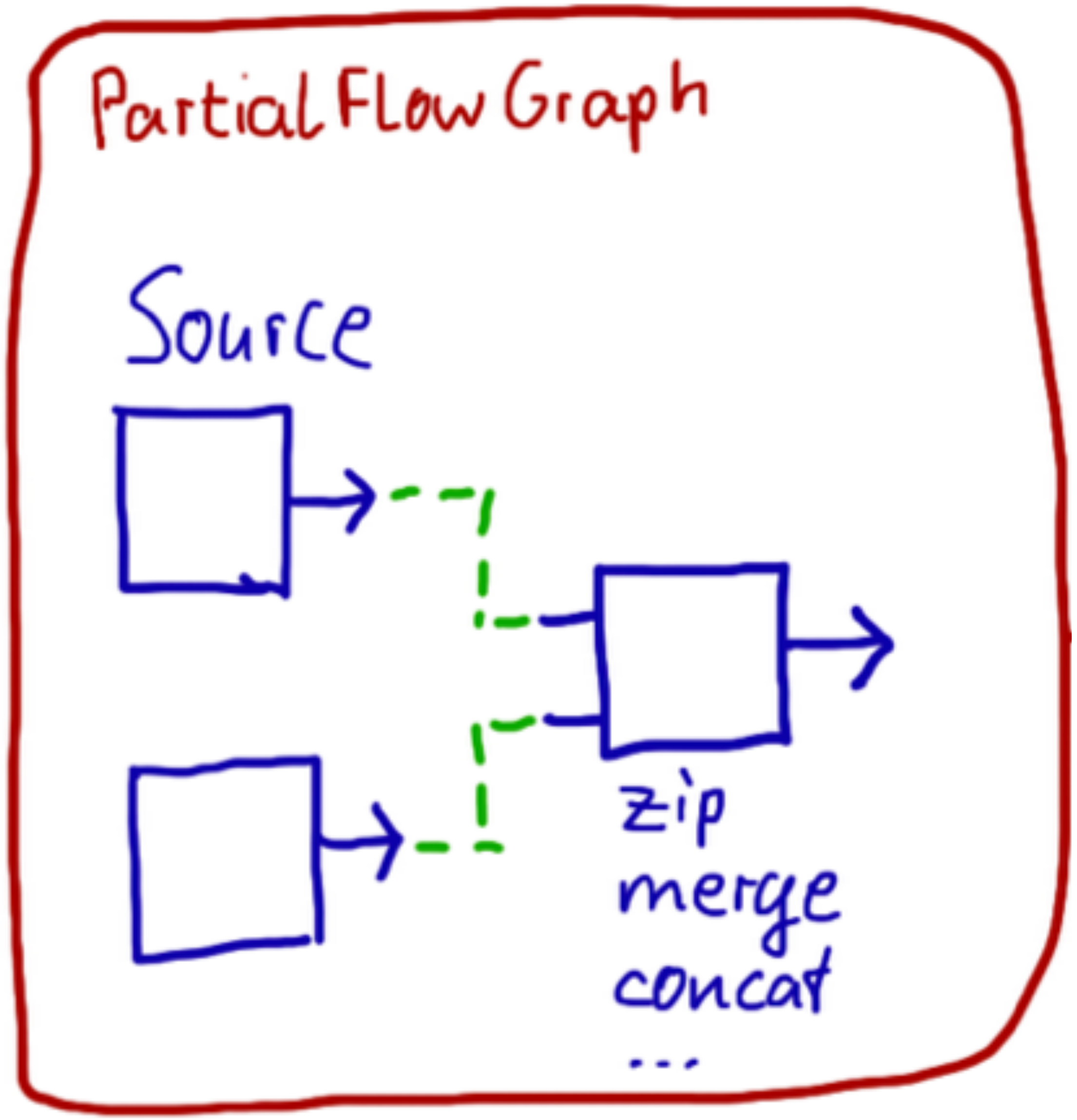


Declaring a Stream Topology

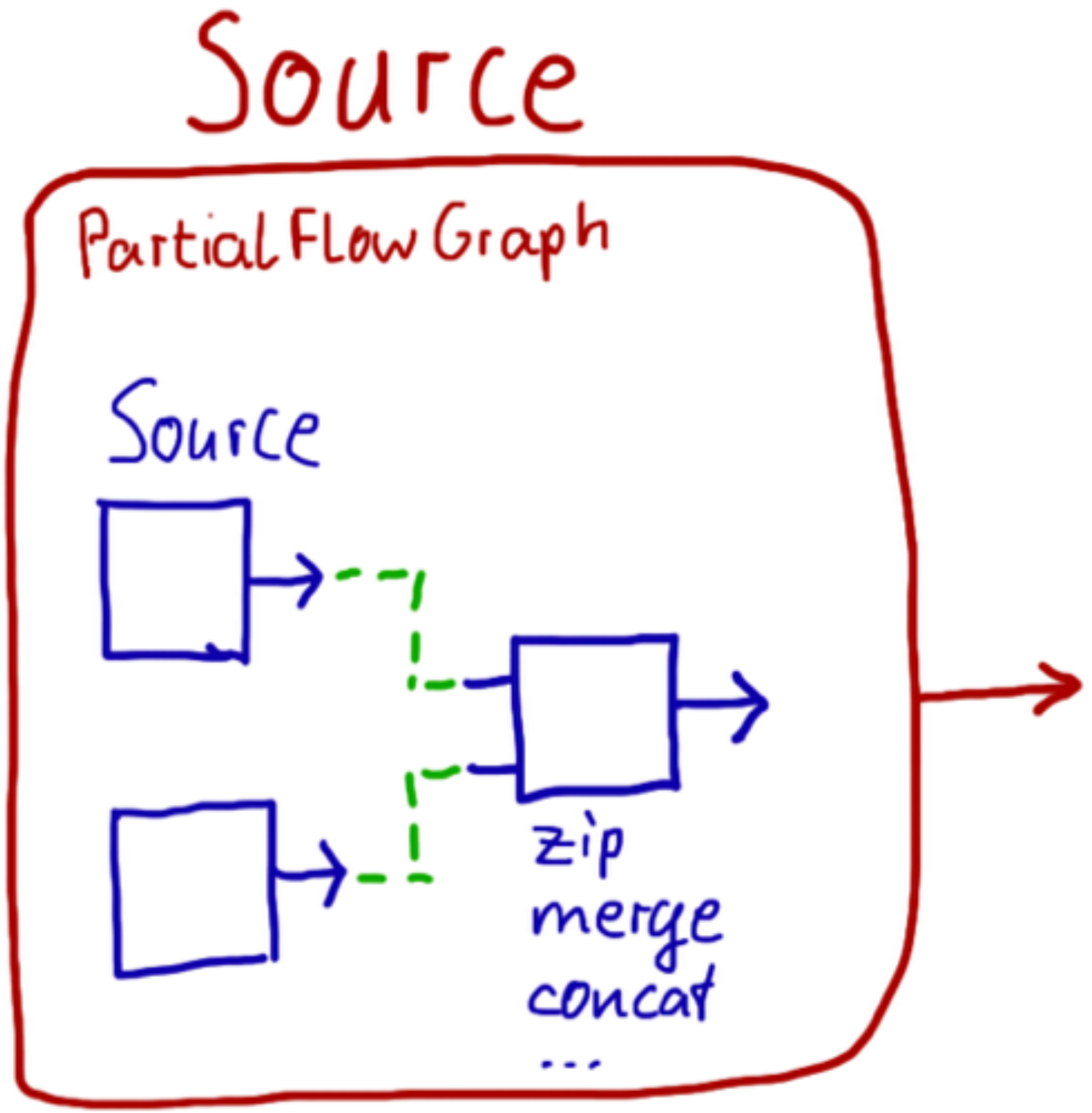
Declaring a Stream Topology



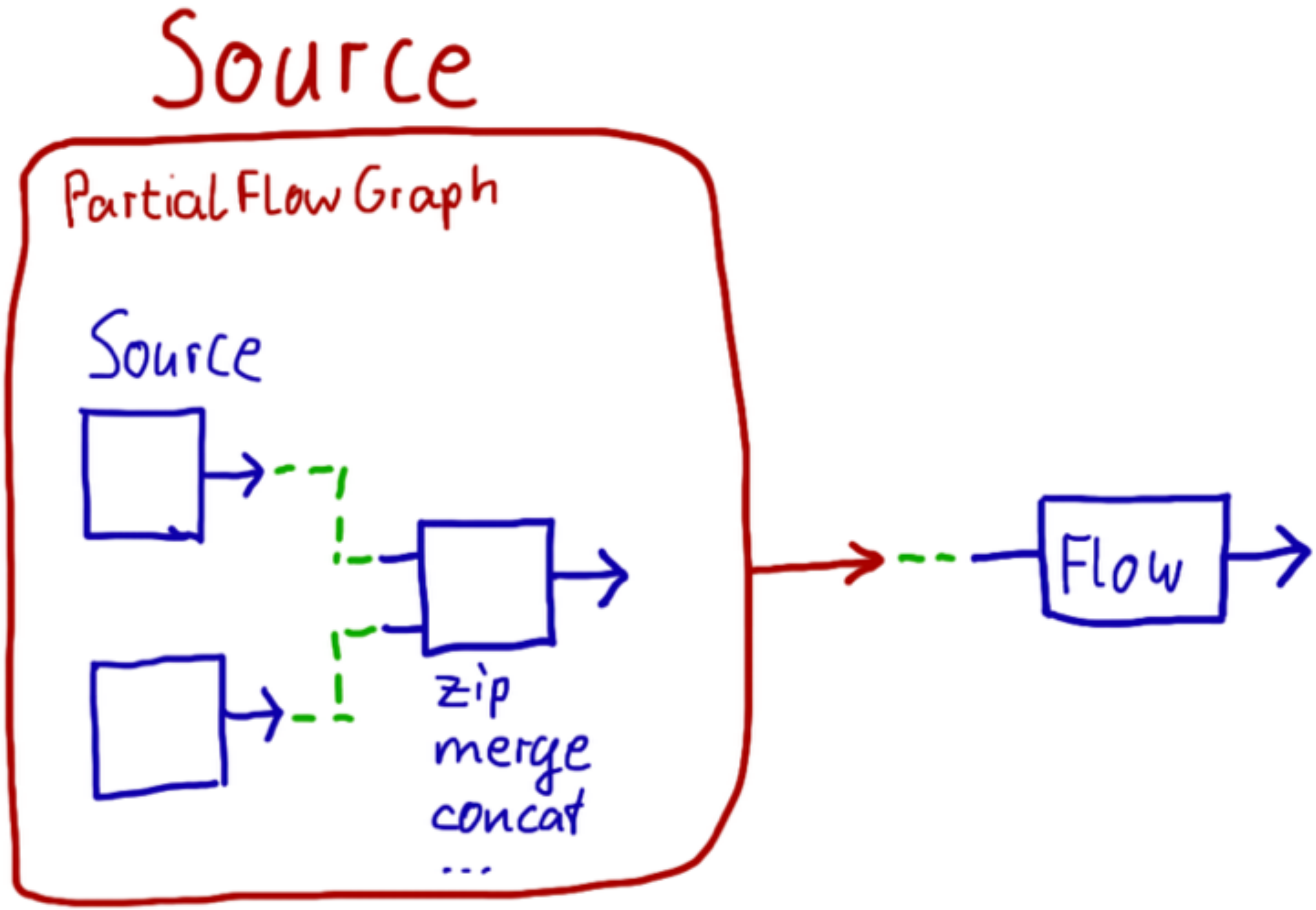
Declaring a Stream Topology



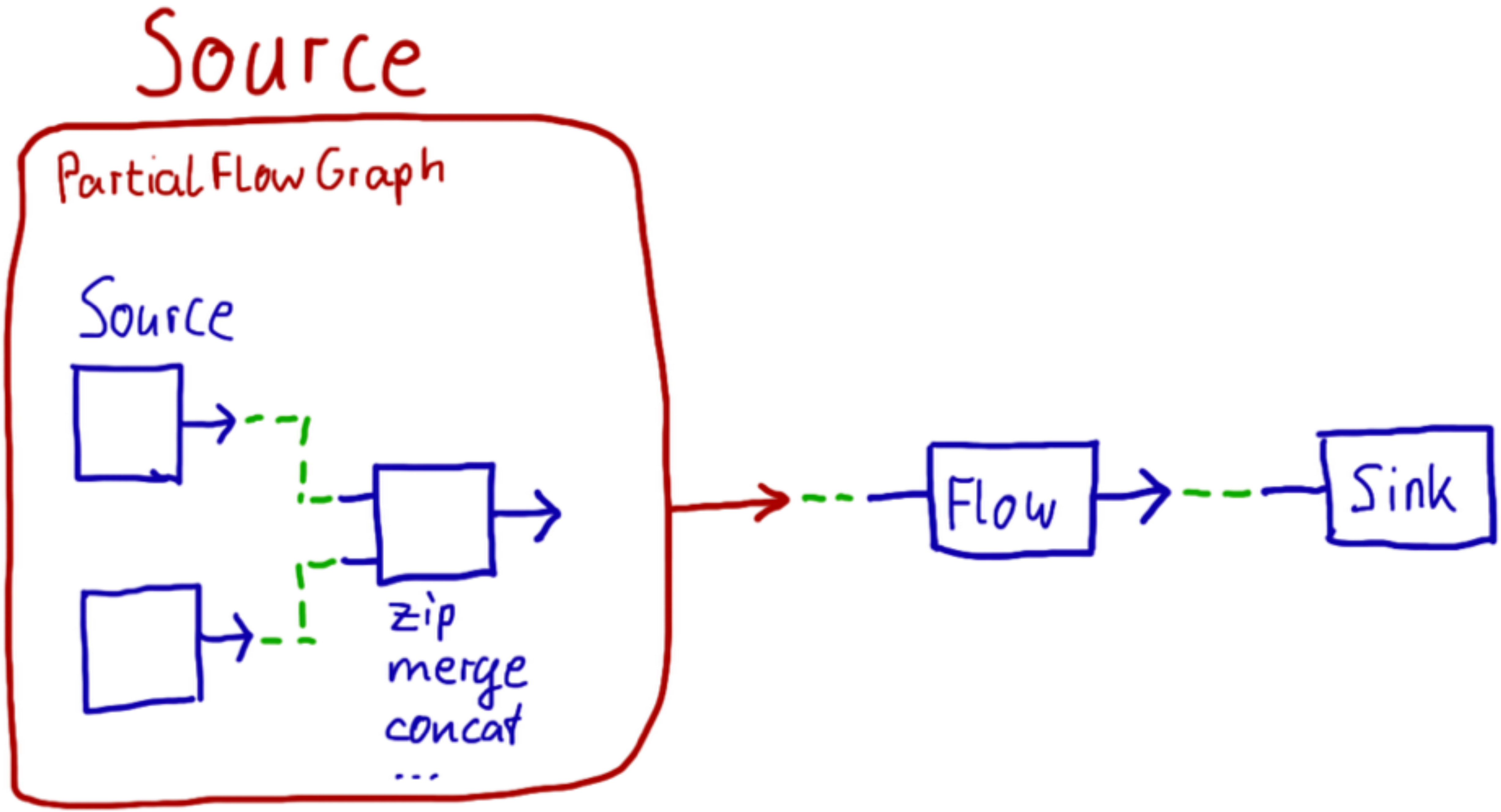
Declaring a Stream Topology



Declaring a Stream Topology



Declaring a Stream Topology



Declaring and Running a Stream

```
val upper = Source(Iterator from 0).take(10)
val lower = Source(1.second, 1.second, () => Tick)

val source = Source[(Int, Tick)]() { implicit b =>
  val zip = Zip[Int, Tick]
  val out = UndefinedSink[(Int, Tick)]

  upper ~> zip.left ~> out
  lower ~> zip.right
  out
}
val flow = Flow[(Int, Tick)].map{ case (x, _) => s"tick $x" }
val sink = Sink.foreach(println)

val future = source.connect(flow).runWith(sink)
```

Declaring and Running a Stream

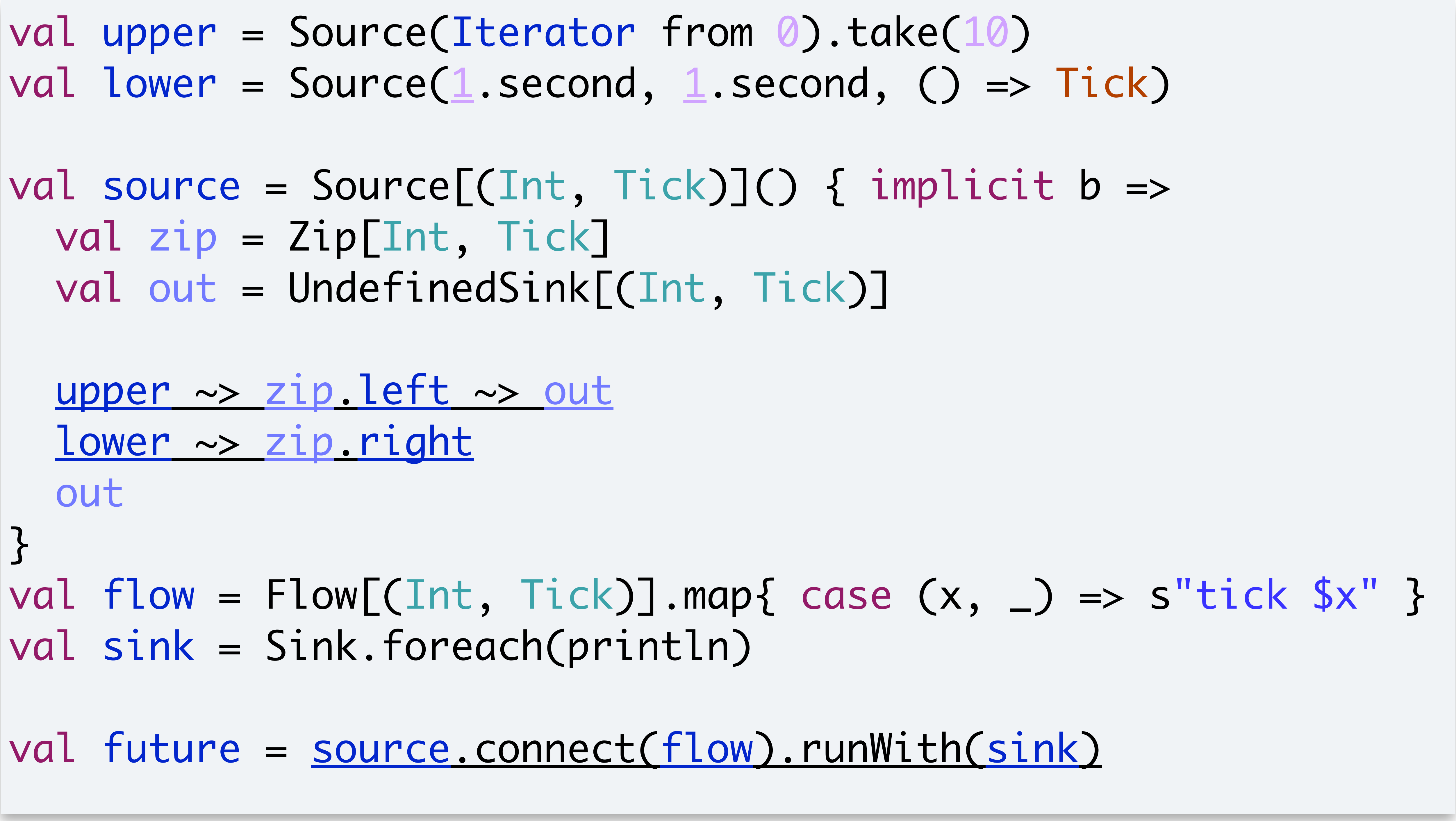
```scala
val upper = Source(Iterator from 0).take(10)
val lower = Source(1.second, 1.second, () => Tick)

val source = Source[(Int, Tick)]() { implicit b =>
  val zip = Zip[Int, Tick]
  val out = UndefinedSink[(Int, Tick)]

  upper ~> zip.left ~> out
  lower ~> zip.right
  out
}
val flow = Flow[(Int, Tick)].map{ case (x, _) => s"tick $x" }
val sink = Sink.foreach(println)

val future = source.connect(flow).runWith(sink)
```

Declaring and Running a Stream



```scala
val upper = Source(Iterator from 0).take(10)
val lower = Source(1.second, 1.second, () => Tick)

val source = Source[(Int, Tick)]() { implicit b =>
  val zip = Zip[Int, Tick]
  val out = UndefinedSink[(Int, Tick)]

  upper ~> zip.left ~> out
  lower ~> zip.right
  out
}
val flow = Flow[(Int, Tick)].map{ case (x, _) => s"tick $x" }
val sink = Sink.foreach(println)

val future = source.connect(flow).runWith(sink)
```

Materialization

- Akka Streams separate the ***what*** from the ***how***
  - declarative Source/Flow/Sink DSL to create blueprint
  - FlowMaterializer turns this into running Actors
- this allows alternative materialization strategies
  - optimization
  - verification / validation
  - cluster deployment
- only Akka Actors for now, but more to come!

Stream Sources

- `org.reactivestreams.Publisher[T]`
- `org.reactivestreams.Subscriber[T]`
- `Iterator[T] / Iterable[T]`
- Code block (function that produces `Option[T]`)
- `scala.concurrent.Future[T]`
- TickSource
- ActorPublisher
- singleton / empty / failed
- … plus write your own (fully extensible)

Stream Sinks

- `org.reactivestreams.Publisher[T]`
- `org.reactivestreams.Subscriber[T]`
- ActorSubscriber
- `scala.concurrent.Future[T]`
- blackhole / foreach / fold / onComplete
- … or create your own

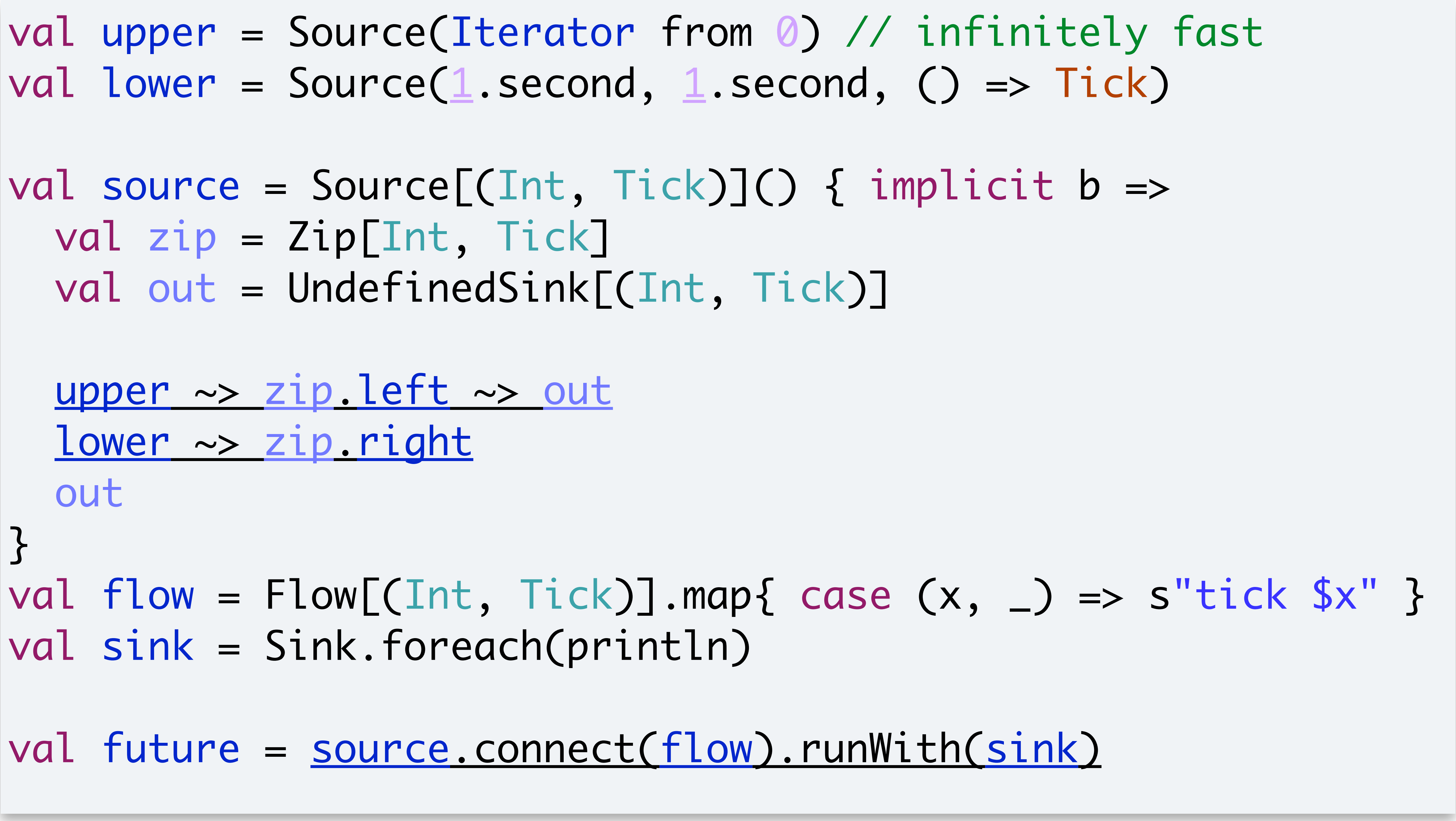
Linear Stream Transformations

- Deterministic (like for collections)
  - map, filter, collect, grouped, drop, take, groupBy, …
- Time-Based
  - takeWithin, dropWithin, groupedWithin, …
- Rate-Detached
  - expand, conflate, buffer, …
- asynchronous
  - mapAsync, mapAsyncUnordered, flatten, …

Nonlinear Stream Transformations

- Fan-In
  - merge, concat, zip, …
- Fan-Out
  - broadcast, route, balance, unzip, …

Why does this work?



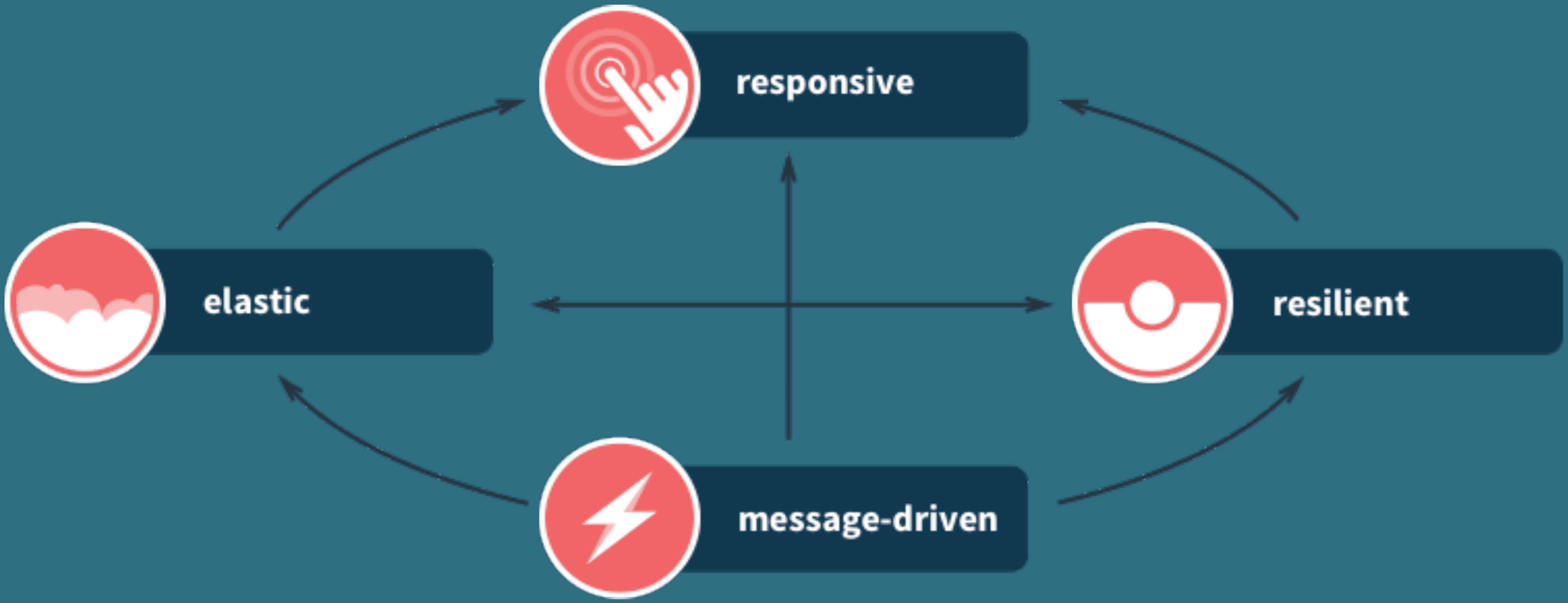
```scala
val upper = Source(Iterator from 0) // infinitely fast
val lower = Source(1.second, 1.second, () => Tick)

val source = Source[(Int, Tick)]() { implicit b =>
  val zip = Zip[Int, Tick]
  val out = UndefinedSink[(Int, Tick)]

  upper ~> zip.left ~> out
  lower ~> zip.right
  out
}
val flow = Flow[(Int, Tick)].map{ case (x, _) => s"tick $x" }
val sink = Sink.foreach(println)

val future = source.connect(flow).runWith(sink)
```

Reactive Traits



Back-Pressure:

the Reactive Streams Initiative

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

Recipe for Success

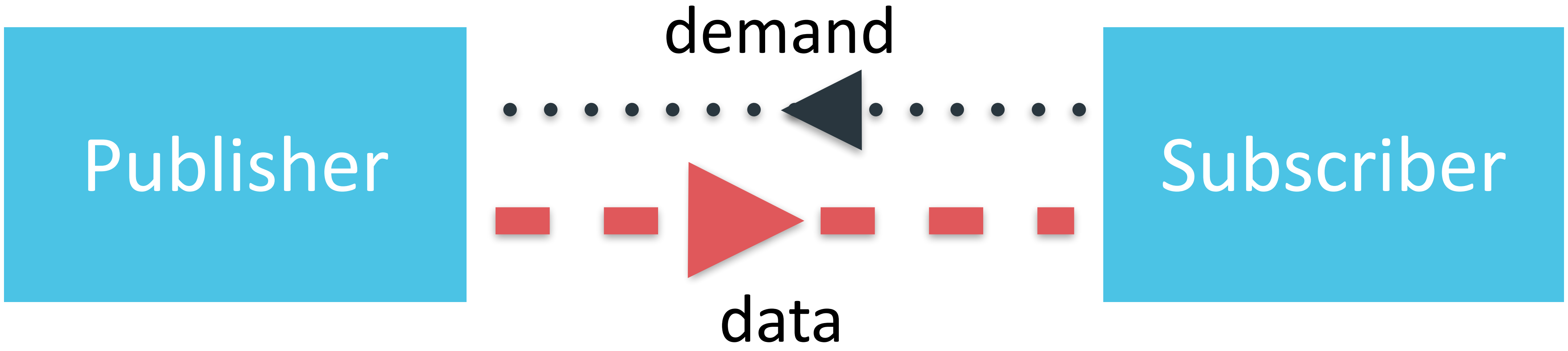
- minimal interfaces
- rigorous specification of semantics
- full TCK for verification of implementation
- complete freedom for many idiomatic APIs

The Meat

```scala
trait Publisher[T] {
  def subscribe(sub: Subscriber[T]): Unit
}
trait Subscription {
  def request(n: Long): 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
}
```

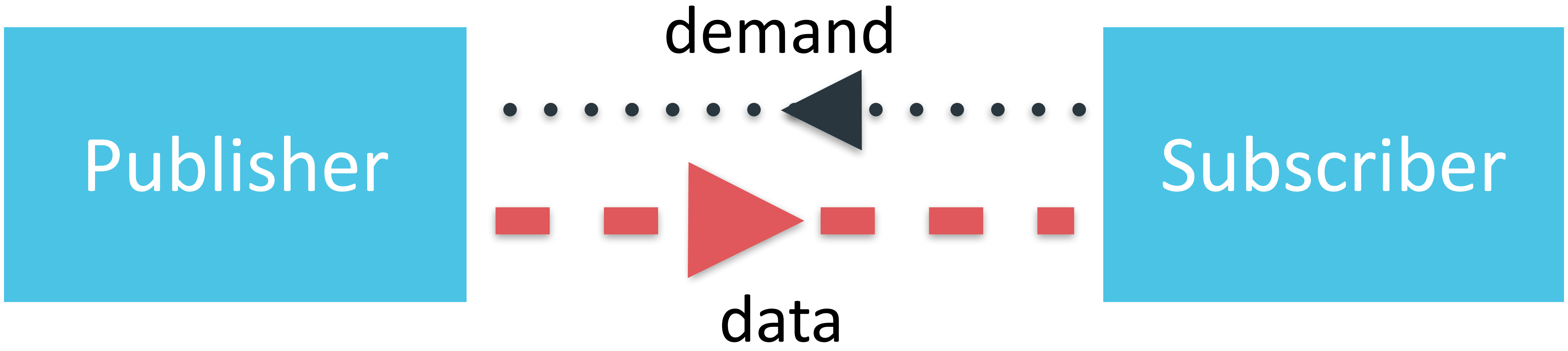
Supply and Demand

- 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

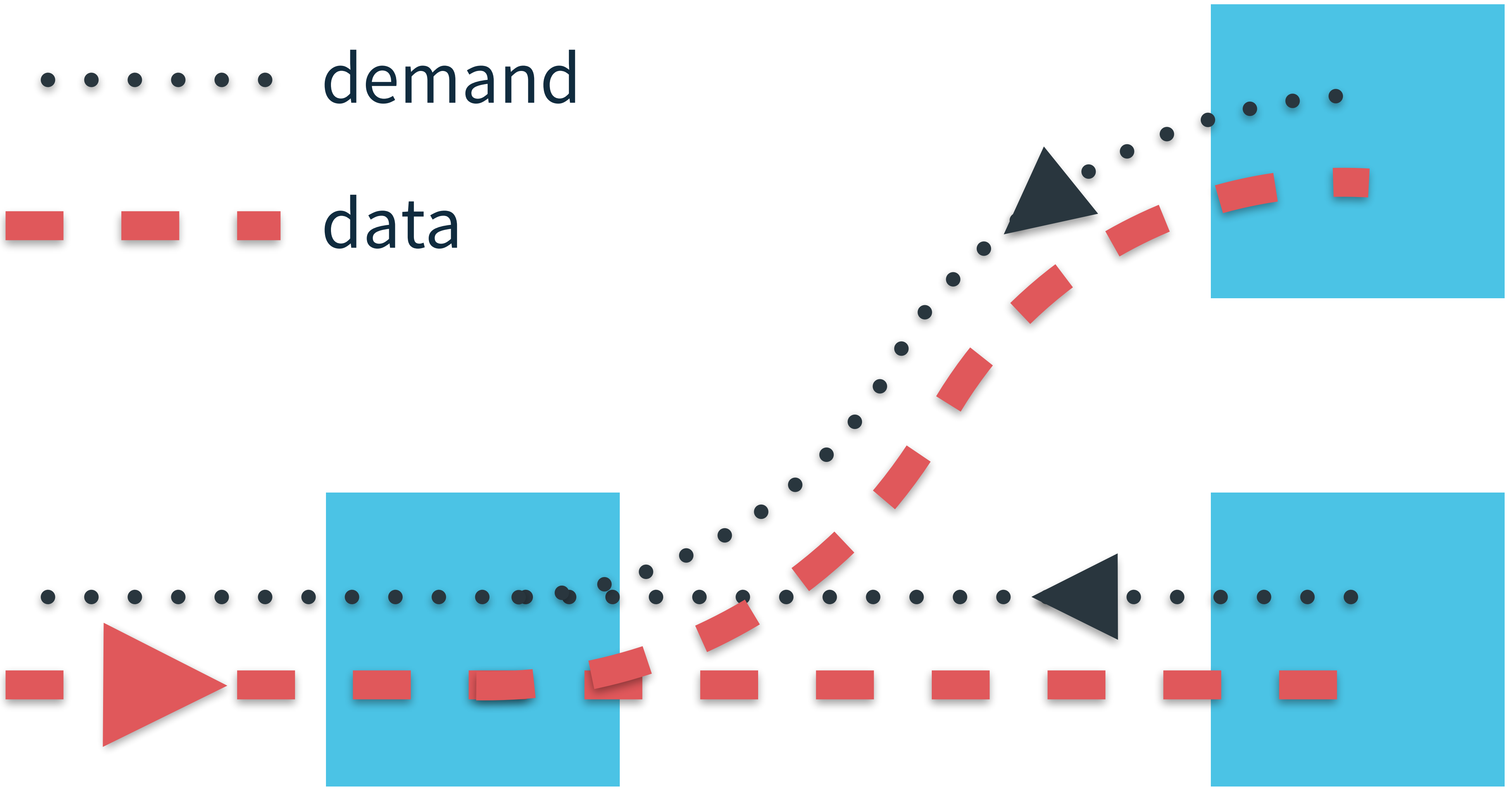


Dynamic Push–Pull

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

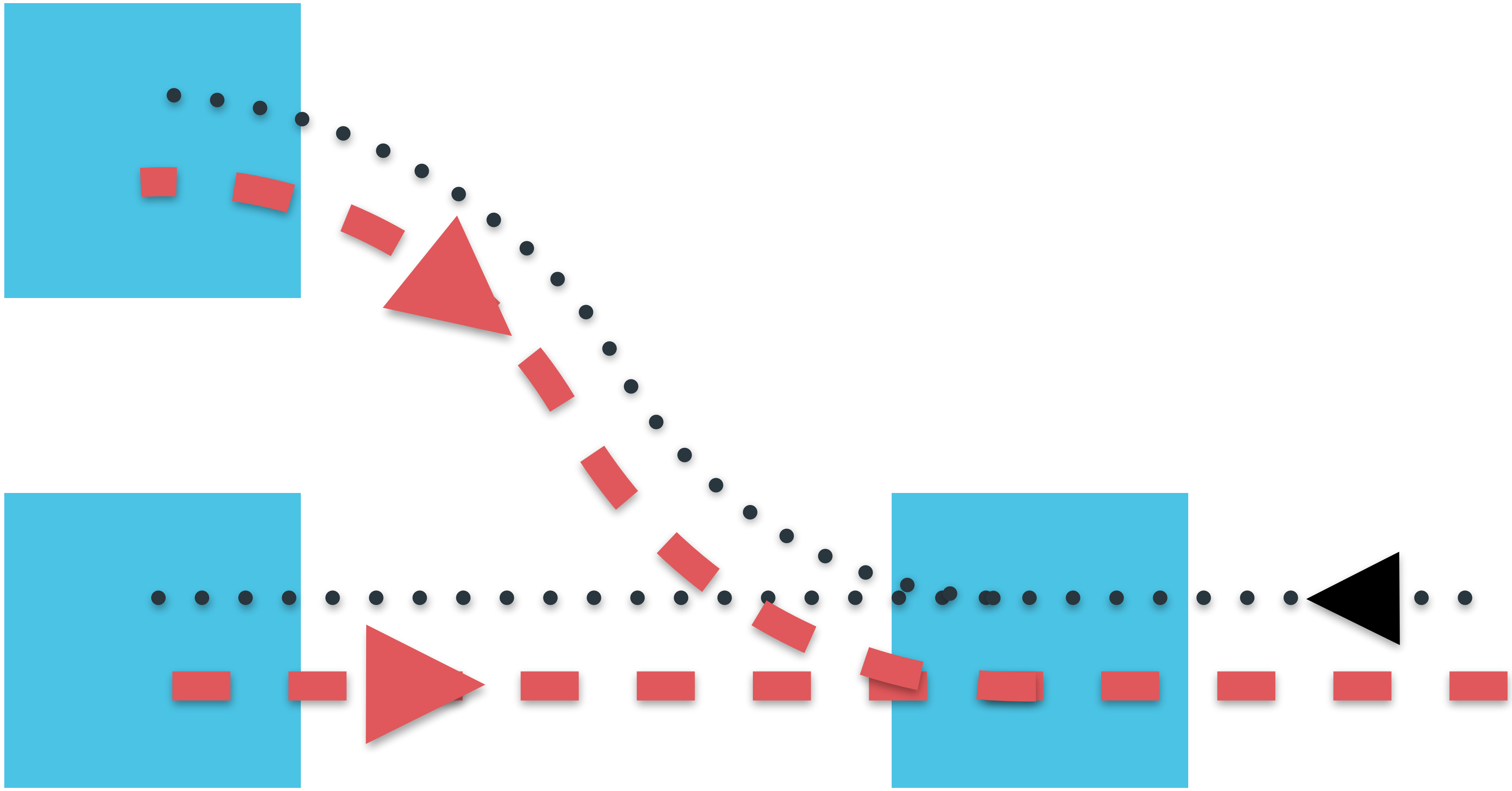


Explicit Demand: Tailored Flow Control



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

Interoperability is King

A fully working example

```
ActorSystem system = ActorSystem.create("InteropTest");
FlowMaterializer mat = FlowMaterializer.create(system);
RxRatpack.initialize();

EmbeddedApp.fromHandler(ctx -> {
    Integer[] ints = new Integer[10];
    for (int i = 0; i < ints.length; ++i) {
        ints[i] = i;
    }
    // RxJava Observable
    Observable<Integer> intObs = Observable.from(ints);
    // Reactive Streams Publisher
    Publisher<Integer> intPub = RxReactiveStreams.toPublisher(intObs);
    // Akka Streams Source
    Source<String> stringSource = Source.from(intPub).map(Object::toString);
    // Reactive Streams Publisher
    Publisher<String> stringPub = stringSource.runWith(Sink.<String>fanoutPublisher(1, 1), mat);
    // Reactor Stream
    Stream<String> linesStream = Streams.create(stringPub).map(i -> i + "\n");
    // and now render the HTTP response using Ratpack
    ctx.render(ResponseChunks.stringChunks(linesStream));
});
```

https://github.com/rkuhn/ReactiveStreamsInterop

When can we have it?

- Sample used pre-release versions:
    - reactive-streams 0.4.0
    - RxJava 1.0.0-rc.8 with rxjava-reactive-streams 0.3.0
    - reactor-core 2.0.0.M1
    - ratpack-core 0.9.10
    - akka-stream-experimental 0.10-M1
- stable versions expected within the next months
- Reactive Streams 1.0 some weeks away

Outlook

- Akka HTTP (successor of Spray.io)
  - fully stream-based
  - Java and Scala DSLs
  - client and server
- more stream-based APIs
  - file I/O (on JRE 7 and higher)
  - database drivers (community developed)
  - Akka Persistence with streams of events

Advertisement:

Berlin Scala User Group — Hack Sequel

Nov 14–16, 2014

There will be T-Shirts, catering and a prize!

Typesafe