Asynchronous stream processing

with

Akka streams

Johan Andrén

JFokus, Stockholm, 2017-02-08

Lightbend

Johan Andrén

Akka Team

Stockholm Scala User Group

Akka

Make building powerful concurrent & distributed applications **simple**.

Akka is a toolkit and runtime
for building highly concurrent,
distributed, and resilient
message-driven applications
on the JVM

What's in the toolkit?

Actors – simple & high performance concurrency

Cluster / Remoting – location transparency, resilience

Cluster tools – and more prepackaged patterns

Streams – back-pressured stream processing

Persistence – Event Sourcing

HTTP – complete, fully async and reactive HTTP Server

Official **Kafka, Cassandra, DynamoDB integrations**, tons more in the community

Complete **Java & Scala APIs** for all features

Reactive Streams

Reactive Streams timeline

Oct 2013
RxJava, Akka and Twitter-people meeting

Apr 2015
Reactive Streams Spec 1.0
TCK
5+ impls

Akka Streams, Rx
Vert.x, MongoDB, ...

"Soon thereafter" 2013
Reactive Streams
Expert group formed

??? 2015
JEP-266
inclusion in JDK9

Reactive Streams

> Reactive Streams is an initiative to provide a standard for asynchronous stream processing with non-blocking back pressure. This encompasses efforts aimed at runtime environments (JVM and JavaScript) as well as network protocols

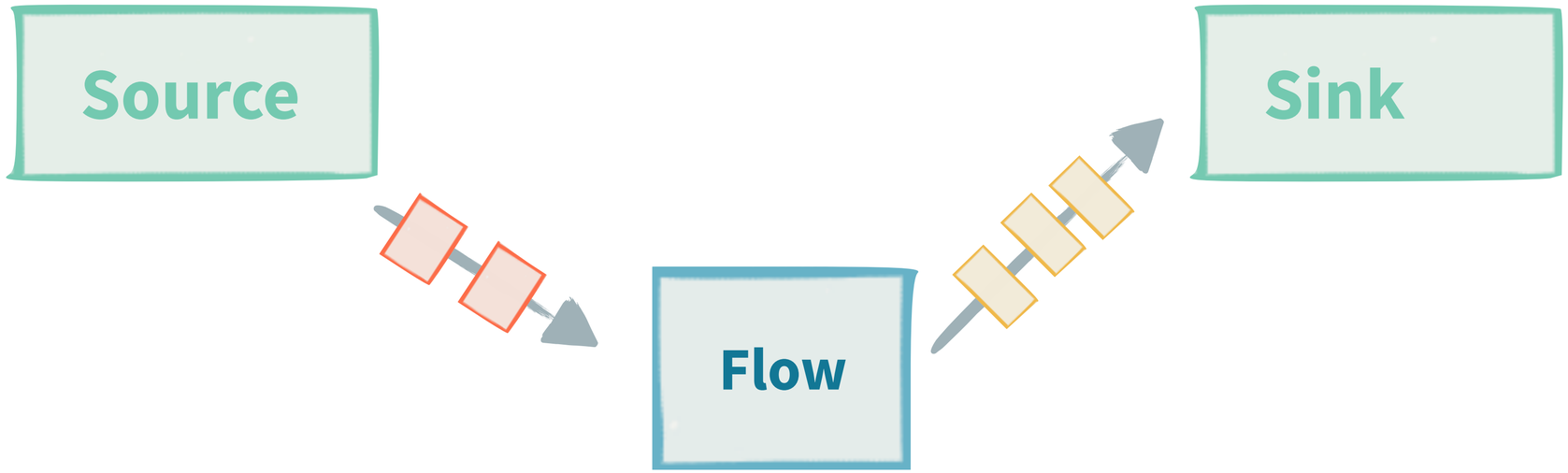
http://www.reactive-streams.org

Reactive Streams

> Reactive Streams is an initiative to provide a standard for asynchronous stream processing with non-blocking back pressure. This encompasses efforts aimed at runtime environments (JVM and JavaScript) as well as network protocols

http://www.reactive-streams.org

Stream processing

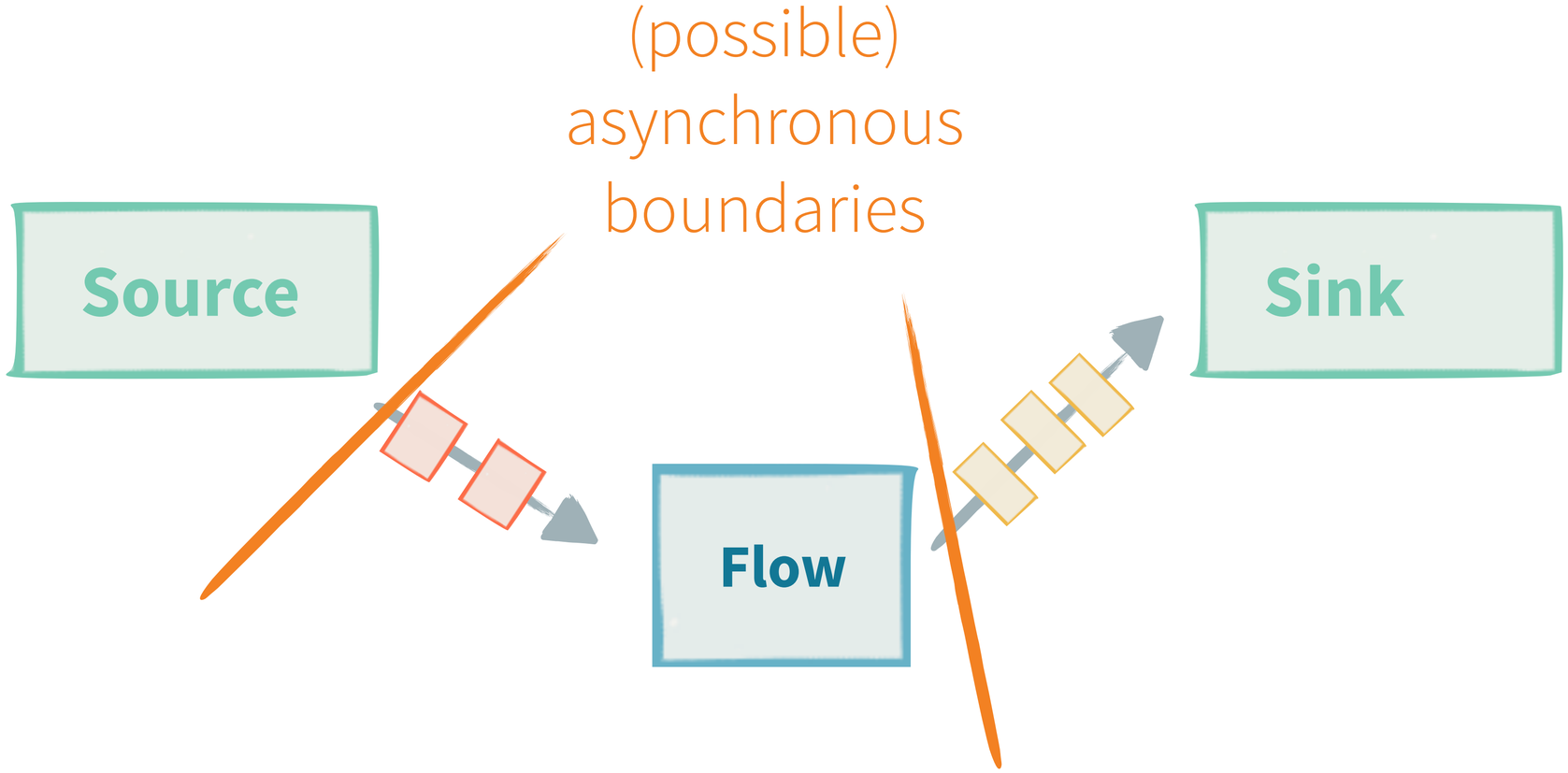


Reactive Streams

> Reactive Streams is an initiative to provide a standard for asynchronous stream processing with non-blocking back pressure. This encompasses efforts aimed at runtime environments (JVM and JavaScript) as well as network protocols

http://www.reactive-streams.org

Asynchronous stream processing

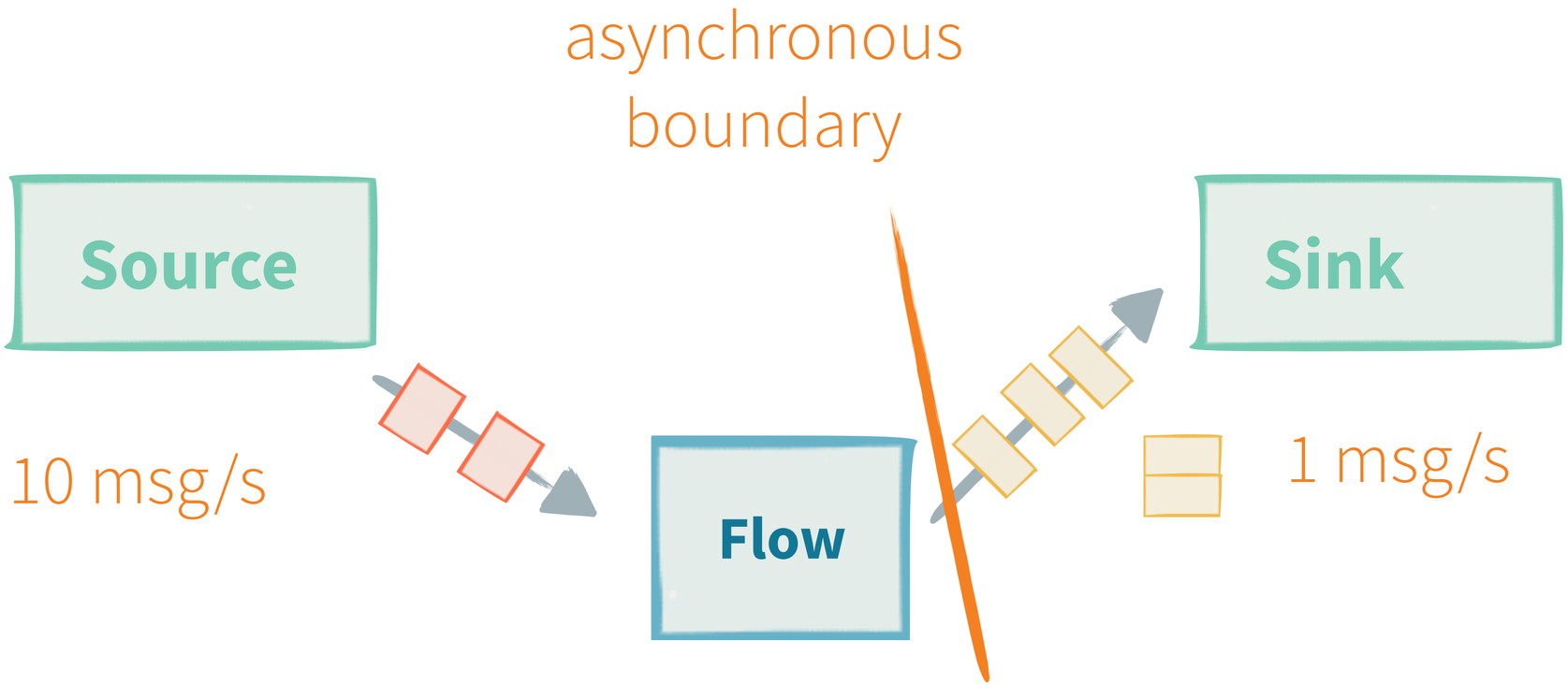


Reactive Streams

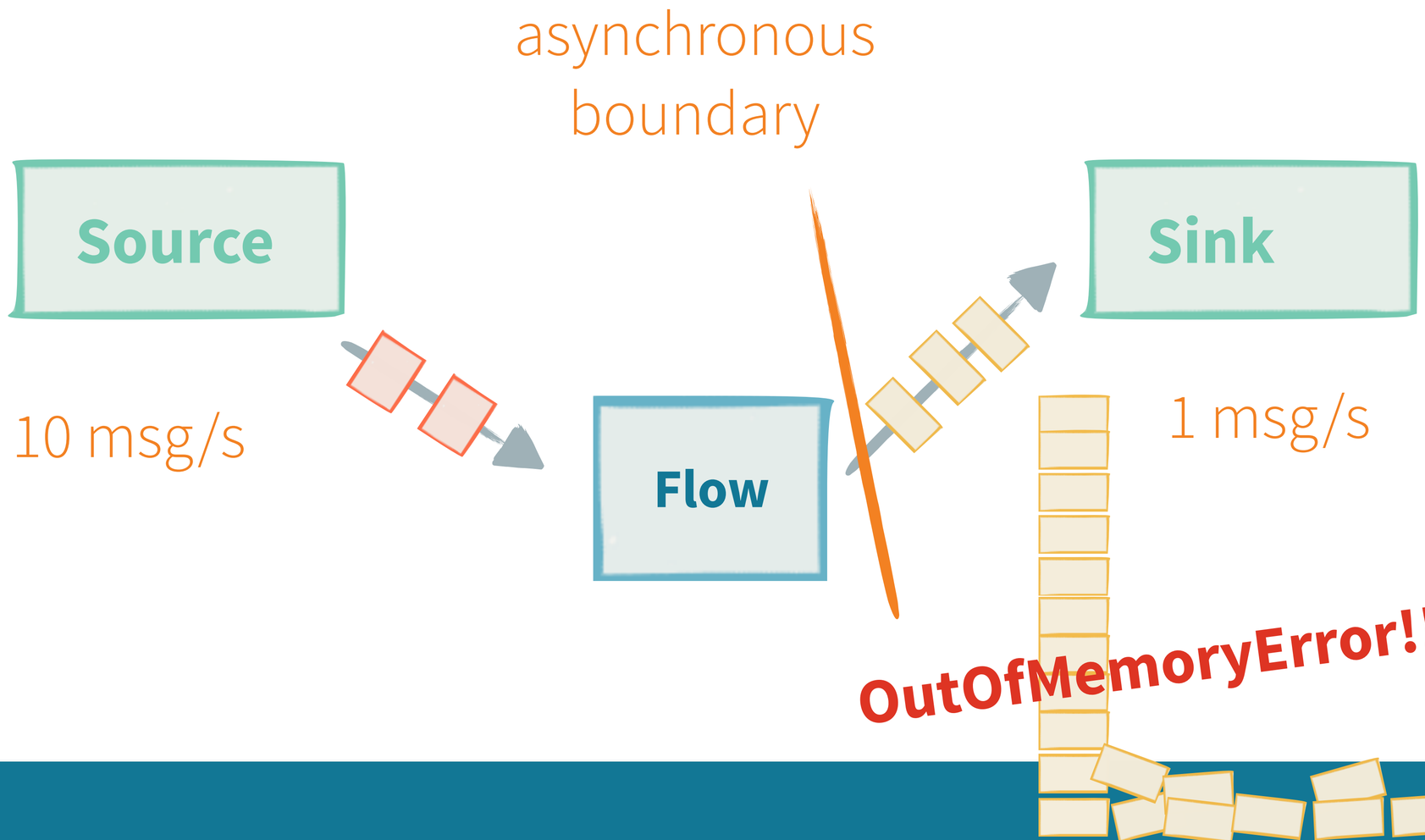
> Reactive Streams is an initiative to provide a standard for asynchronous stream processing with non-blocking back pressure. This encompasses efforts aimed at runtime environments (JVM and JavaScript) as well as network protocols

http://www.reactive-streams.org

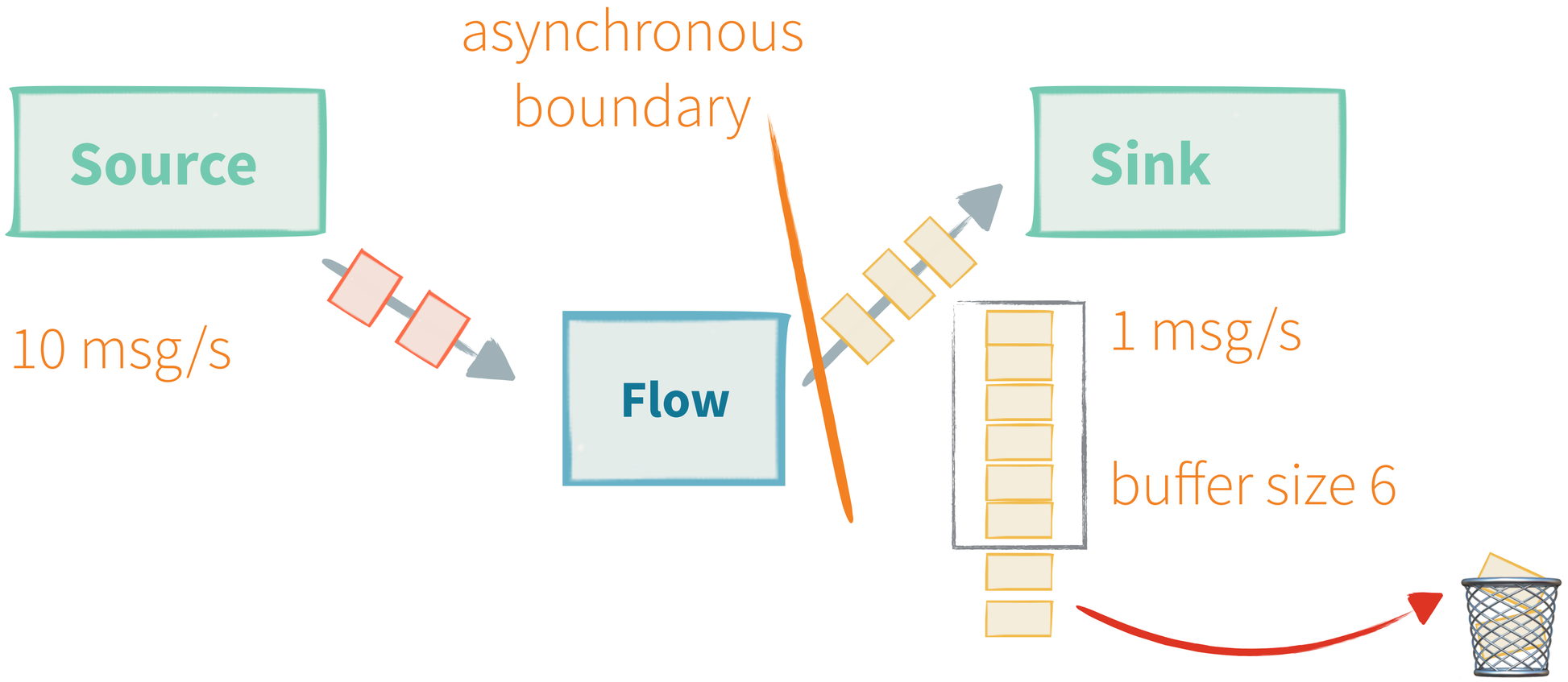
No back pressure



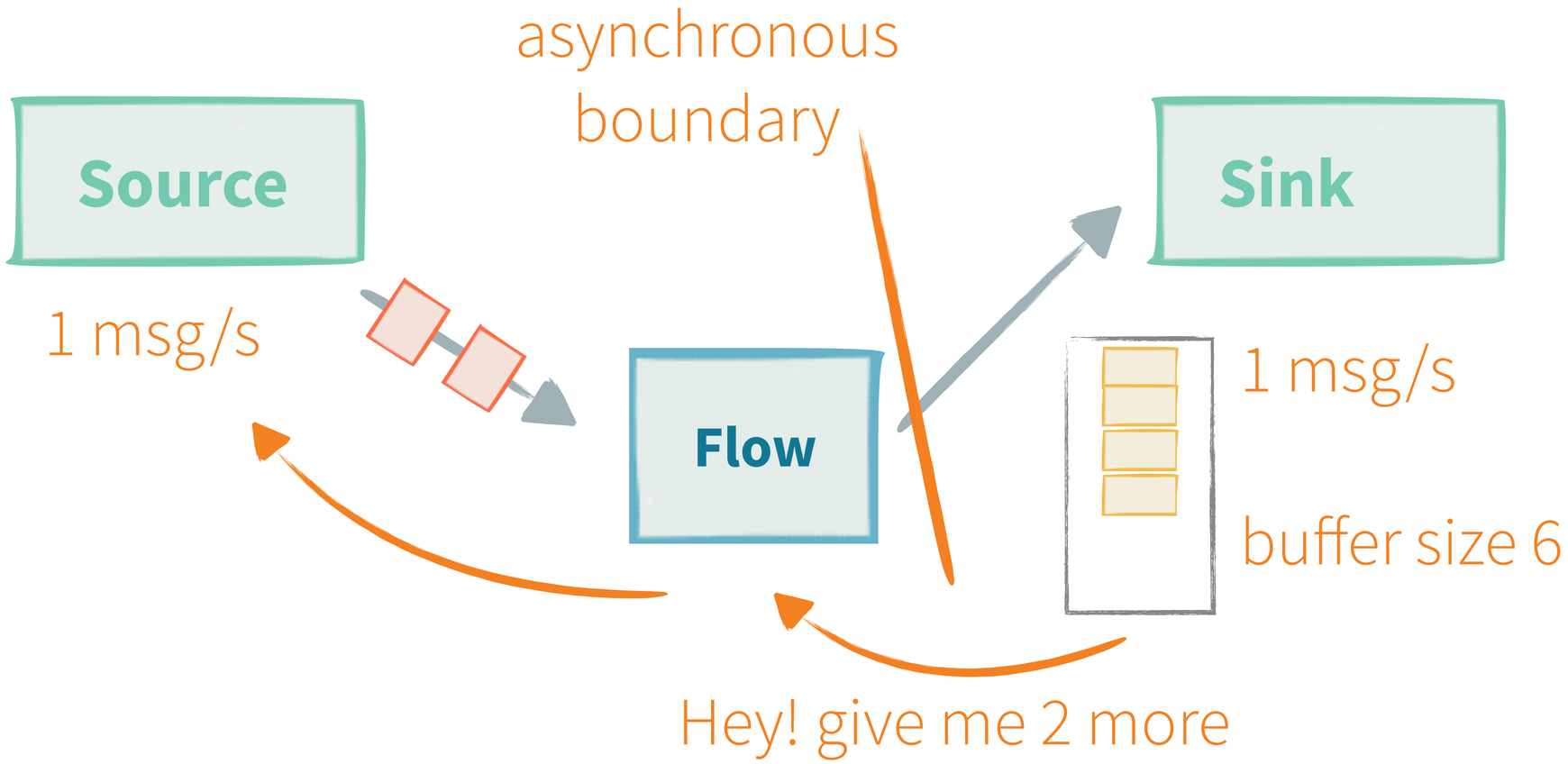
No back pressure



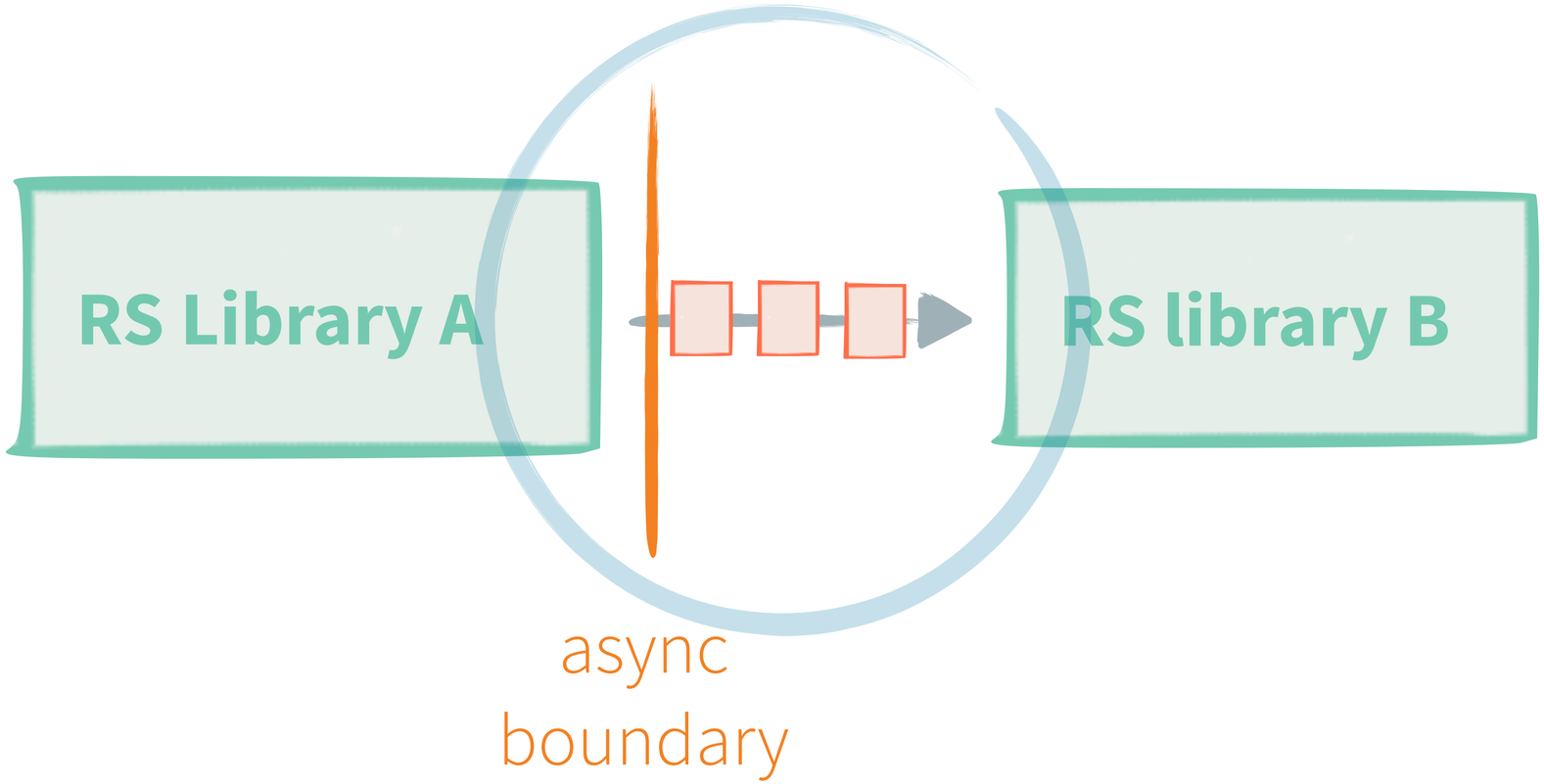
No back pressure - bounded buffer



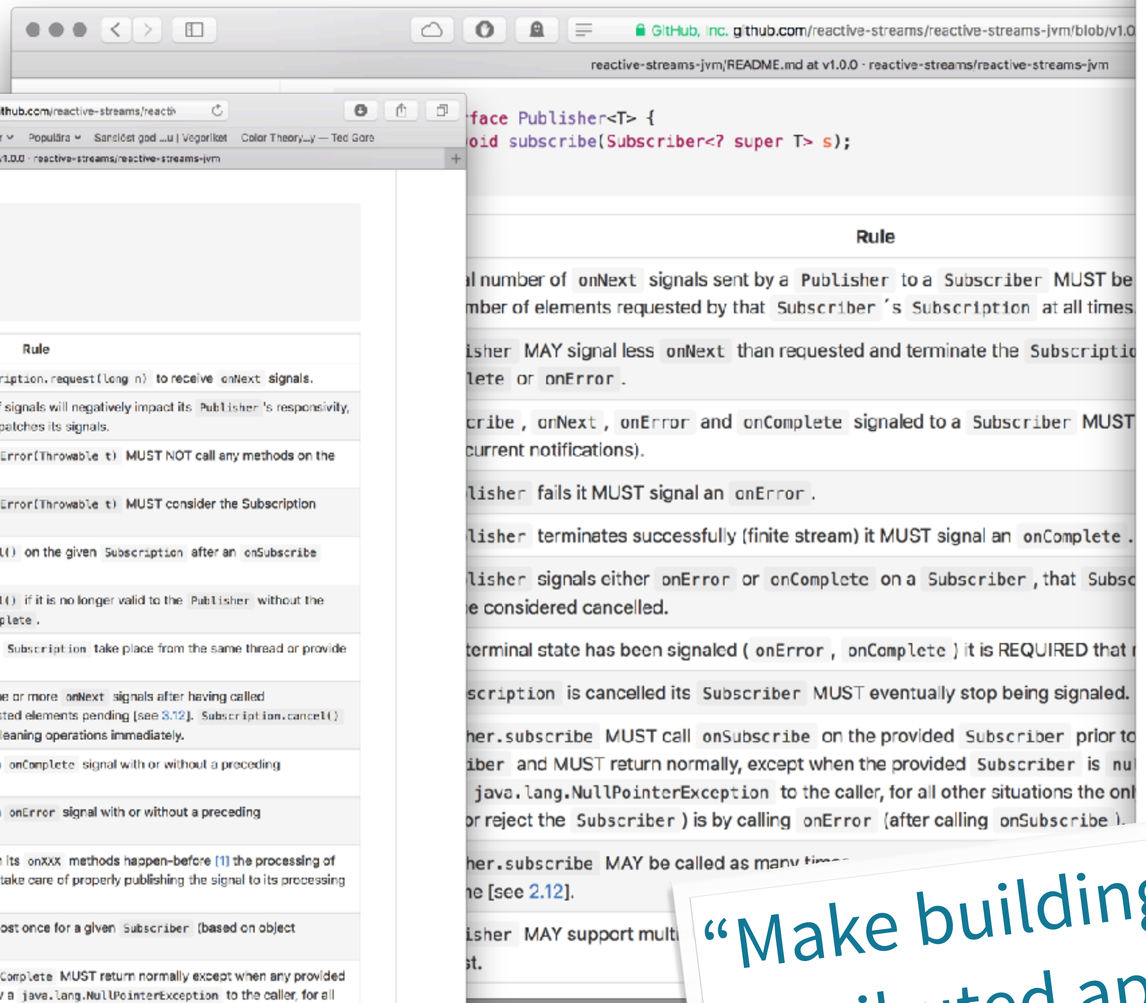
Async non blocking back pressure

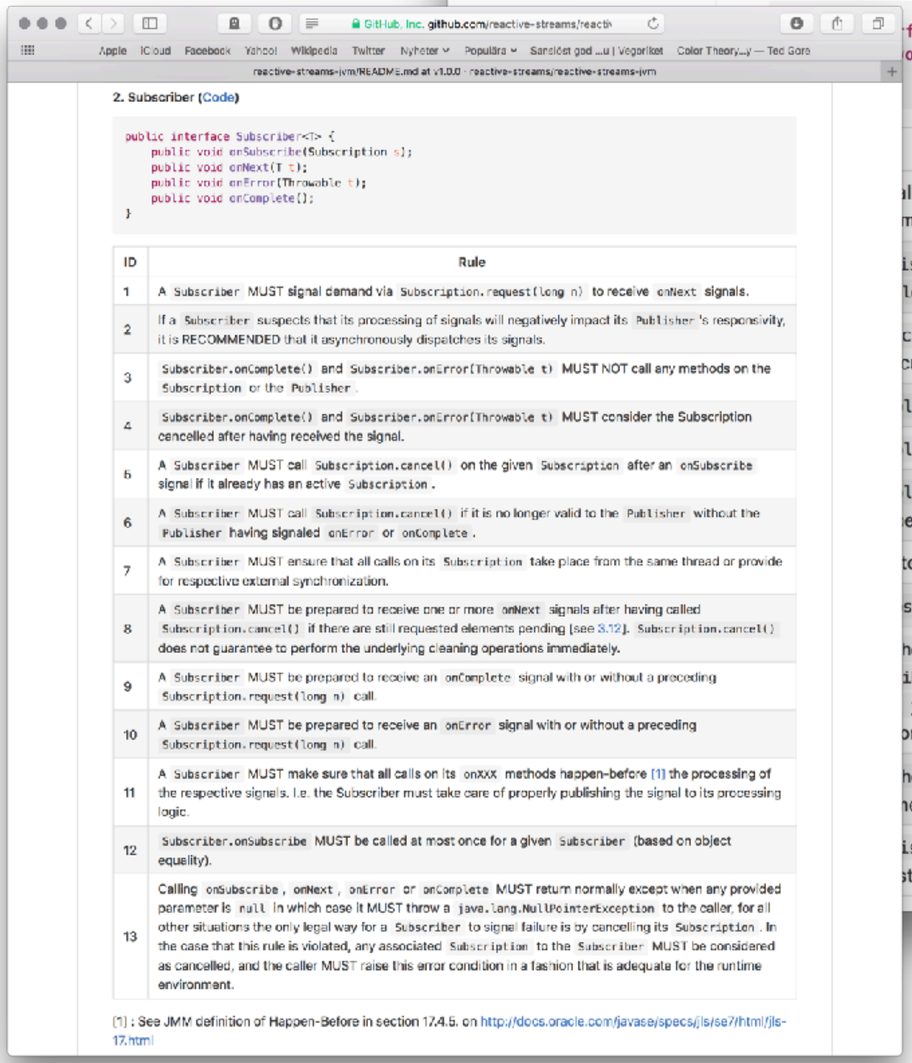


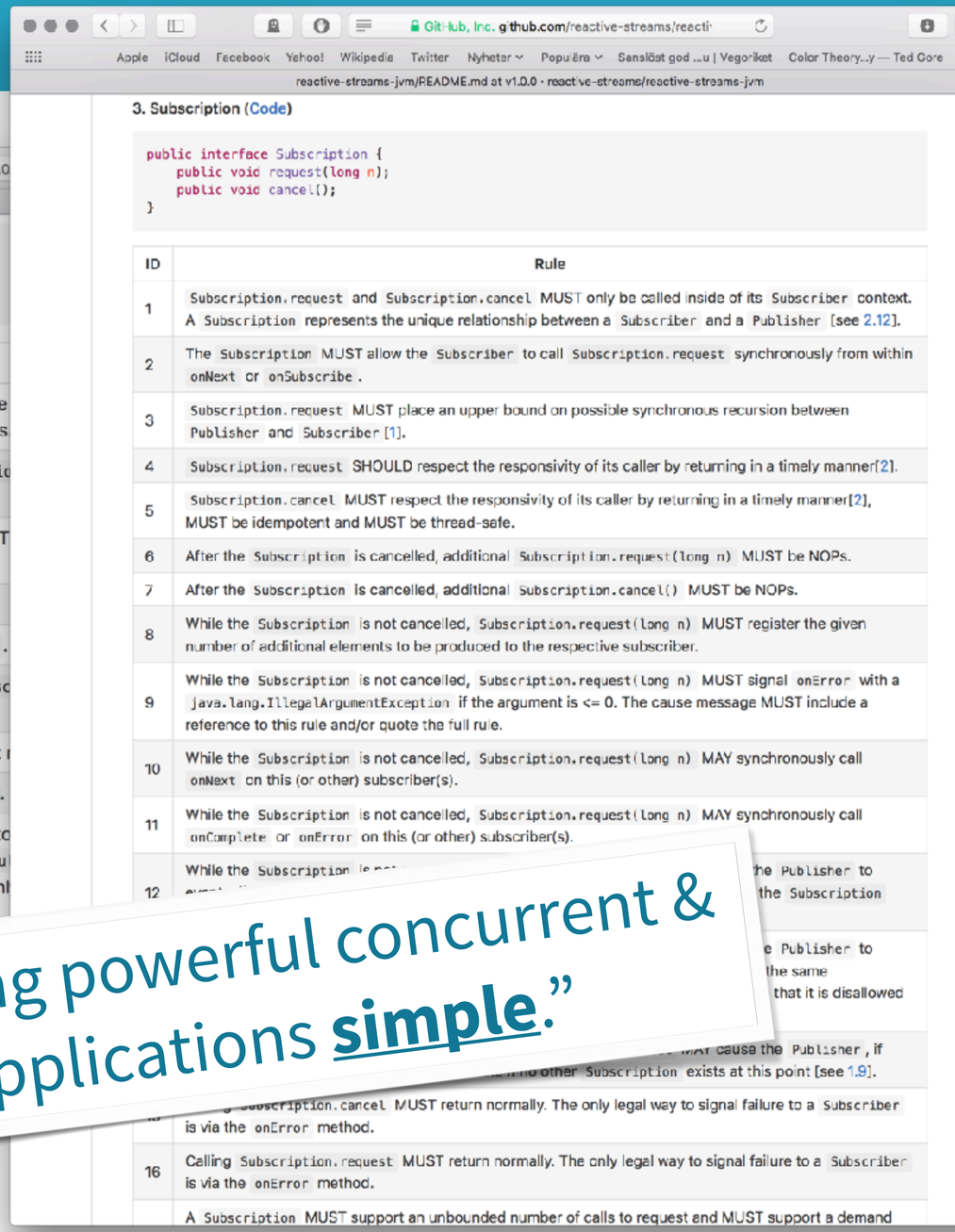
Reactive Streams



Reactive Streams







"Make building powerful concurrent & distributed applications **simple**."

Akka Streams

Complete and awesome
Java and Scala APIs
(Just like everything in Akka)

Akka Streams in ~20 seconds:

```
final ActorSystem system = ActorSystem.create();
final Materializer materializer = ActorMaterializer.create(system);

final Source<Integer, NotUsed> source =
        Source.range(0, 20000000);

final Flow<Integer, String, NotUsed> flow =
        Flow.fromFunction((Integer n) -> n.toString());

final Sink<String, CompletionStage<Done>> sink =
        Sink.foreach(str -> System.out.println(str));

final RunnableGraph<NotUsed> runnable = source.via(flow).to(sink);

runnable.run(materializer);
```


complete sources on github

Akka Streams in ~20 seconds:

```
implicit val system = ActorSystem()
implicit val mat = ActorMaterializer()

val source = Source(0 to 20000000)

val flow = Flow[Int].map(_.toString())

val sink = Sink.foreach[String](println(_))

val runnable = source.via(flow).to(sink)

runnable.run()
```

Akka Streams in ~20 seconds:

```
Source.range(0, 20000000)
        .map(Object::toString)
        .runForeach(str -> System.out.println(str), materializer);
```


complete sources on github

Akka Streams in ~20 seconds:

```
Source(0 to 20000000)
  .map(_.toString)
  .runForeach(println)
```


complete sources on github

Numbers as a service

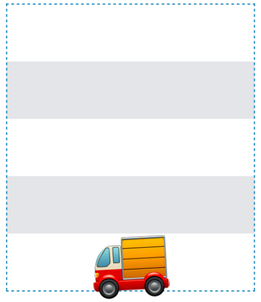
```
final Source<ByteString, NotUsed> numbers = Source.unfold(0L, n -> {
    long next = n + 1;
    return Optional.of(Pair.create(next, next));
}).map(n -> ByteString.fromString(n.toString() + "\n"));


final Route route =
        path("numbers", () ->
                get(() ->
                        complete(HttpResponse.create()
                                .withStatus(StatusCodes.OK)
                                .withEntity(HttpEntities.create(
                                        ContentTypes.TEXT_PLAIN_UTF8,
                                        numbers
                                )))
                )
        );

final CompletionStage<ServerBinding> bindingCompletionStage =
        http.bindAndHandle(route.flow(system, materializer), host, materializer);
```

Numbers as a service

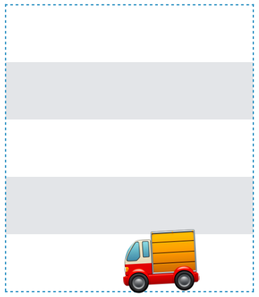
```
val numbers =
  Source.unfold(0L) { (n) =>
    val next = n + 1
    Some((next, next))
  }.map(n => ByteString(n + "\n"))

val route =
  path("numbers") {
    get {
      complete(
        HttpResponse(entity = HttpEntity(`text/plain(UTF-8)`, numbers))
      )
    }
  }
val futureBinding = Http().bindAndHandle(route, "127.0.0.1", 8080)
```

Back pressure over TCP

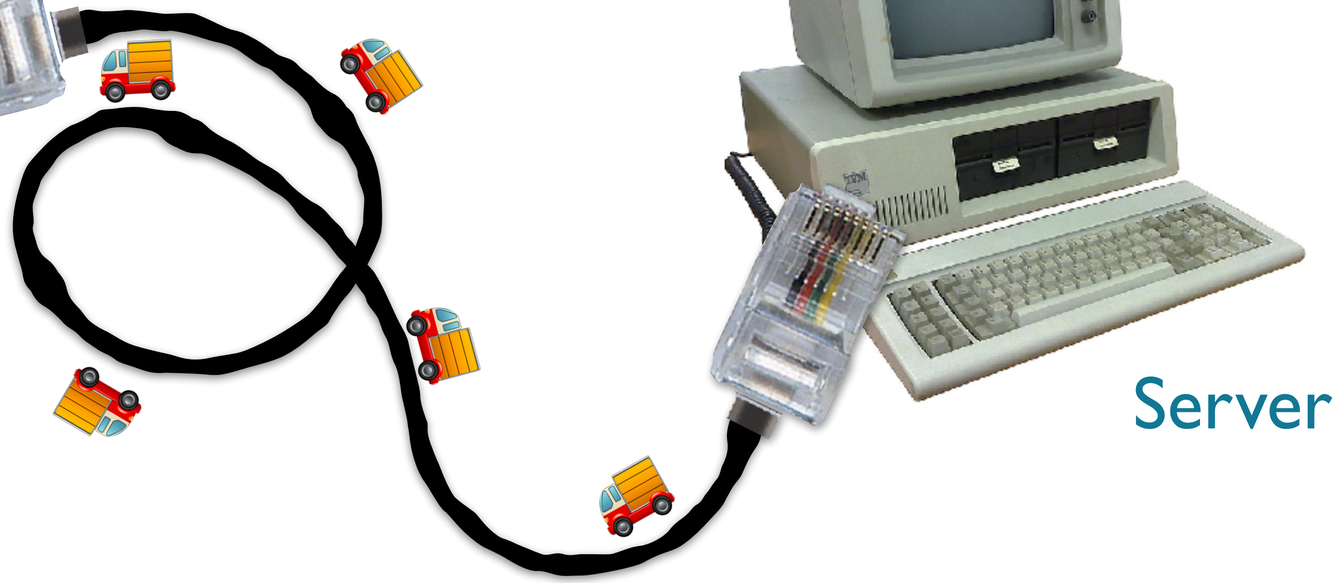
Client



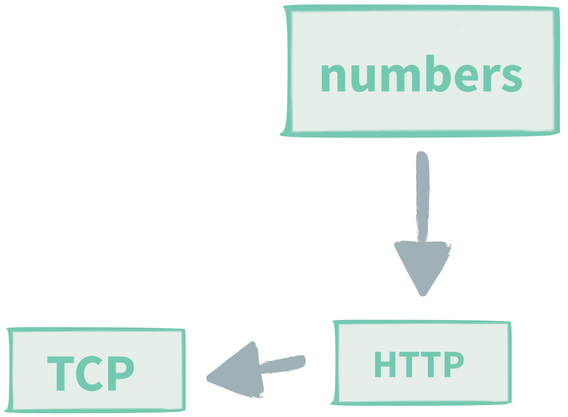
send buffer



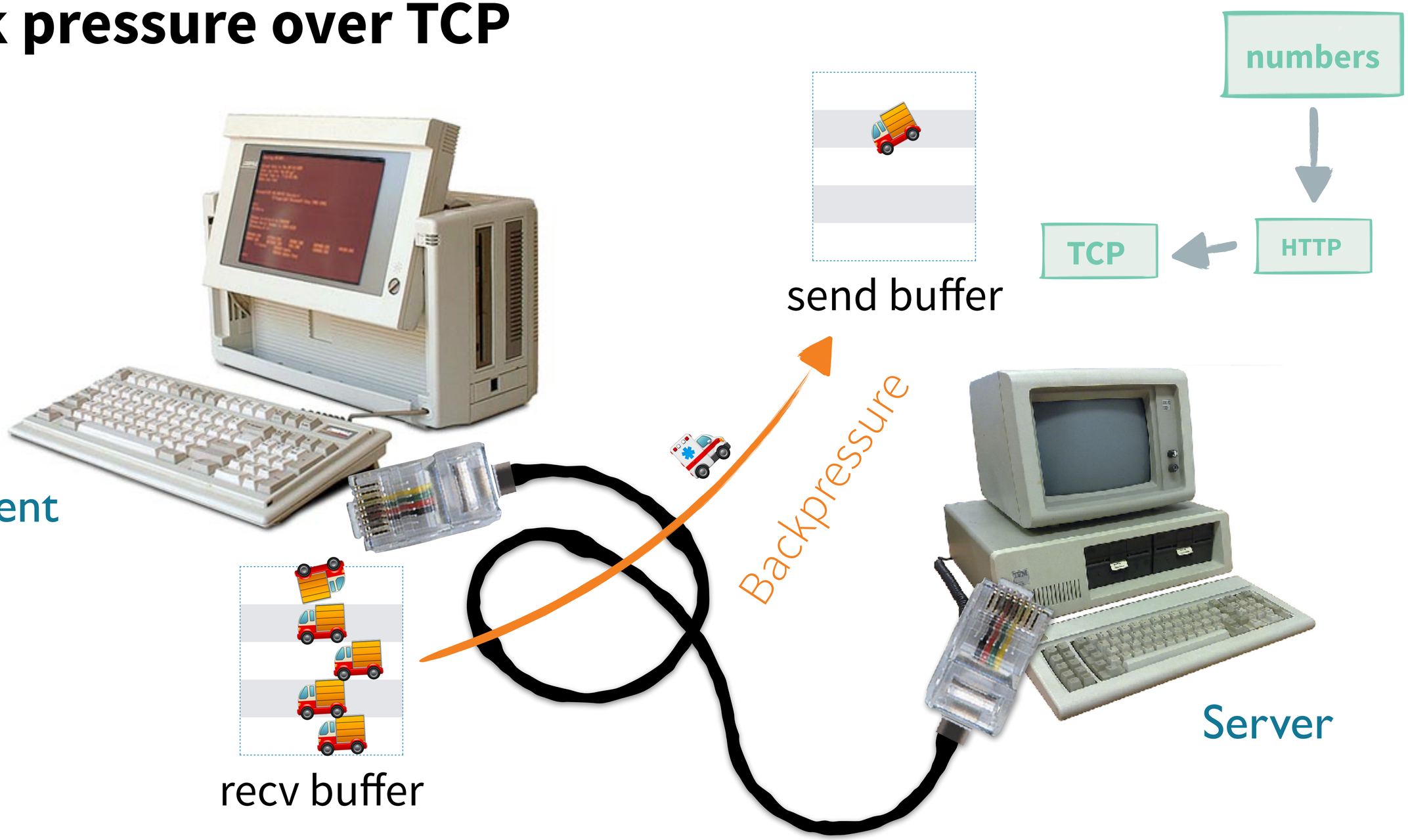
recv buffer



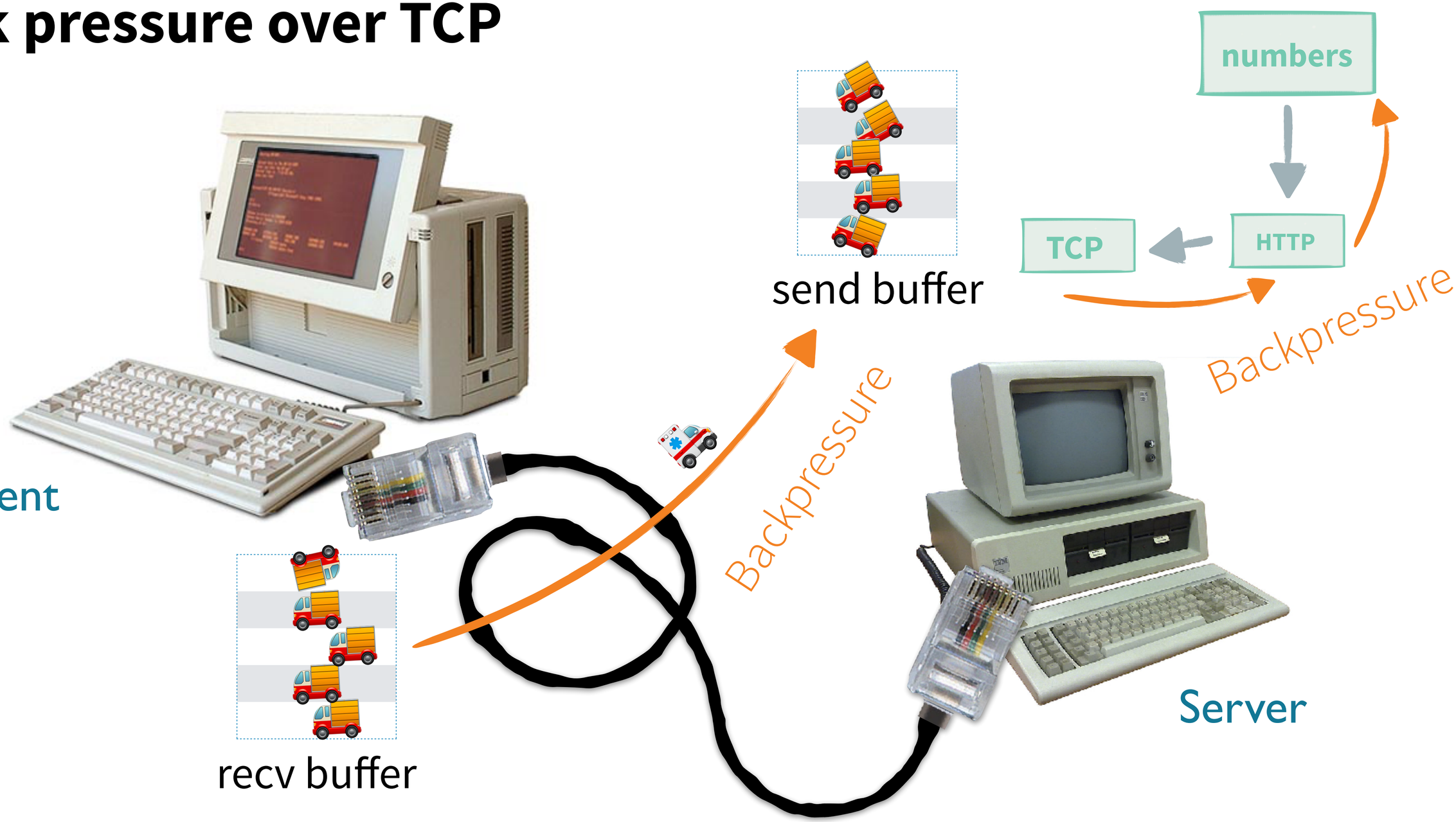
Server



Back pressure over TCP



Back pressure over TCP



A more useful example

Credit to: Colin Breck

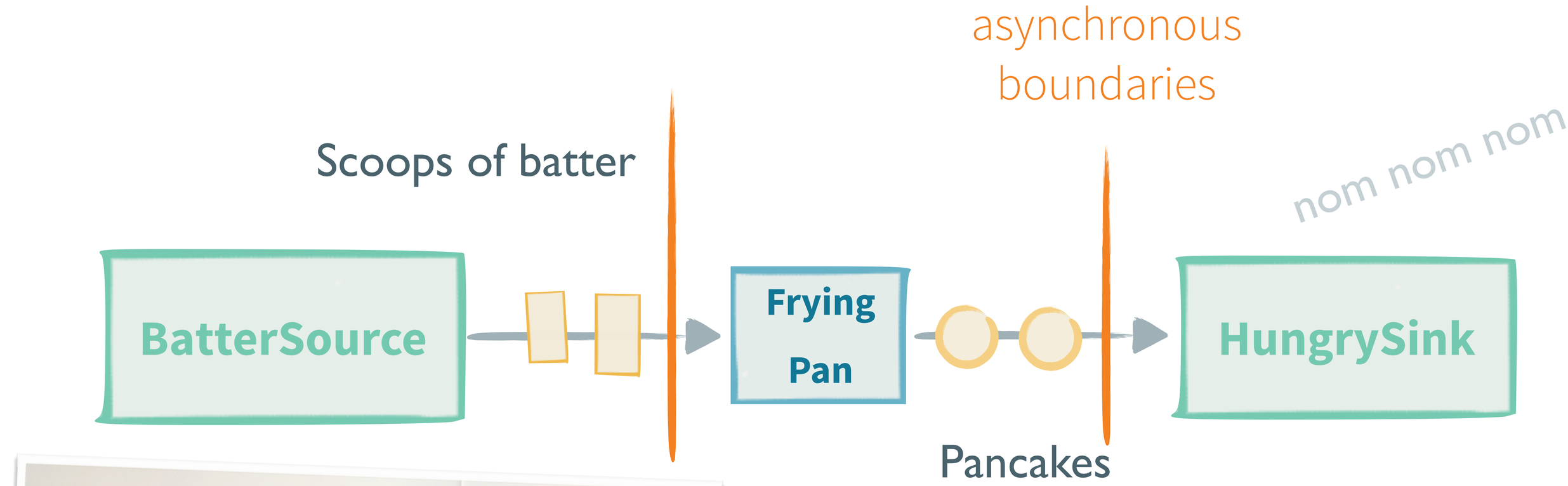
```
final Flow<Message, Message, NotUsed> measurementsFlow =
        Flow.of(Message.class)
                .flatMapConcat((Message message) ->
                        message.asTextMessage()
                                .getStreamedText()
                                .fold("", (acc, elem) -> acc + elem)
                )
                .groupedWithin(1000, FiniteDuration.create(1, SECONDS))
                .mapAsync(5, database::asyncBulkInsert)
                .map(written ->
                        TextMessage.create("wrote up to: " + written.get(written.size() - 1))
                );

final Route route = path("measurements", () ->
        get(() ->
                handleWebSocketMessages(measurementsFlow)
        )
);

final CompletionStage<ServerBinding> bindingCompletionStage =
        http.bindAndHandle(route.flow(system, materializer), host, materializer);
```


complete sources on github

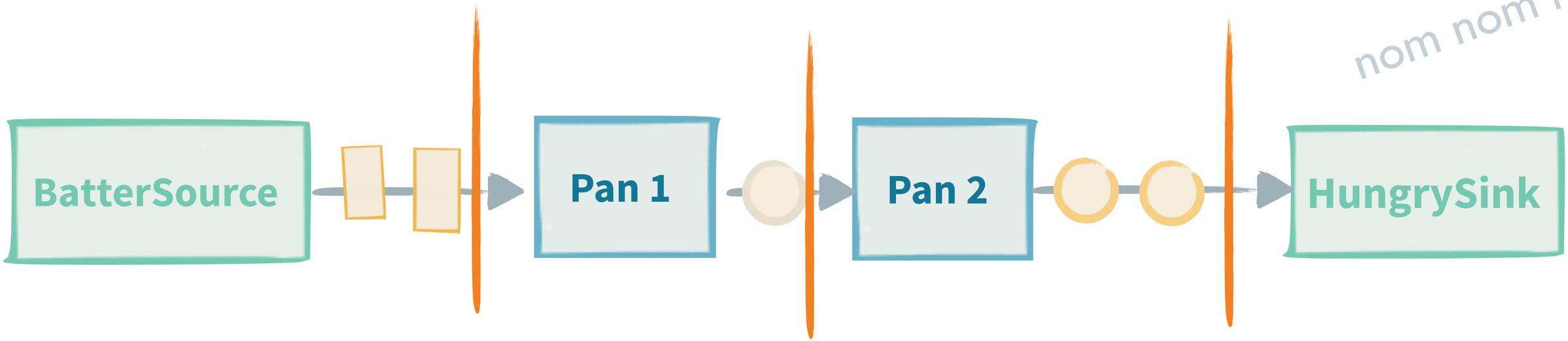
A more useful example

```
val measurementsFlow =
  Flow[Message].flatMapConcat(message =>
      message.asTextMessage.getStreamedText.fold("")(_ + _)
    )
    .groupedWithin(1000, 1.second)
    .mapAsync(5)(Database.asyncBulkInsert)
    .map(written => TextMessage("wrote up to: " + written.last))

val route =
  path("measurements") {
    get {
      handleWebSocketMessages(measurementsFlow)
    }
  }

val futureBinding = Http().bindAndHandle(route, "127.0.0.1", 8080)
```

The tale of the two pancake chefs



Rolands pipelined pancakes



Rolands pipelined pancakes

```
Flow<ScoopOfBatter, HalfCookedPancake, NotUsed> fryingPan1 =
  Flow.of(ScoopOfBatter.class).map(batter -> new HalfCookedPancake());

Flow<HalfCookedPancake, Pancake, NotUsed> fryingPan2 =
  Flow.of(HalfCookedPancake.class).map(halfCooked -> new Pancake());

Flow<ScoopOfBatter, Pancake, NotUsed> pancakeChef =
  fryingPan1.async().via(fryingPan2.async());
```

Rolands pipelined pancakes

```
// Takes a scoop of batter and creates a pancake with one side cooked
val fryingPan1: Flow[ScoopOfBatter, HalfCookedPancake, NotUsed] =
  Flow[ScoopOfBatter].map { batter => HalfCookedPancake() }

// Finishes a half-cooked pancake
val fryingPan2: Flow[HalfCookedPancake, Pancake, NotUsed] =
  Flow[HalfCookedPancake].map { halfCooked => Pancake() }

// With the two frying pans we can fully cook pancakes
val pancakeChef: Flow[ScoopOfBatter, Pancake, NotUsed] =
  Flow[ScoopOfBatter].via(fryingPan1.async).via(fryingPan2.async)
```

Patriks parallel pancakes

Patriks parallel pancakes

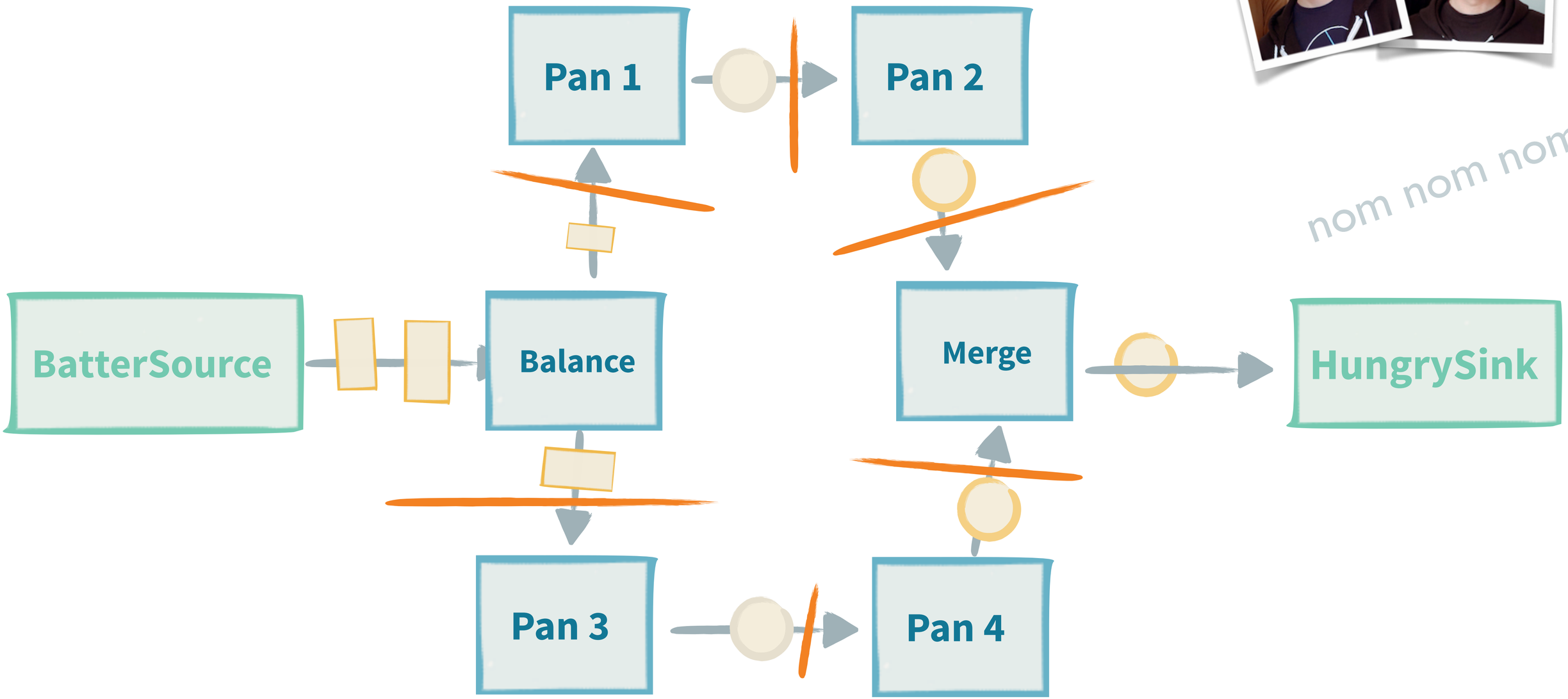
```
Flow<ScoopOfBatter, Pancake, NotUsed> fryingPan =
  Flow.of(ScoopOfBatter.class).map(batter -> new Pancake());

Flow<ScoopOfBatter, Pancake, NotUsed> pancakeChef =
  Flow.fromGraph(GraphDSL.create(builder -> {
    final UniformFanInShape<Pancake, Pancake> mergePancakes =
      builder.add(Merge.create(2));
    final UniformFanOutShape<ScoopOfBatter, ScoopOfBatter> dispatchBatter =
      builder.add(Balance.create(2));

    builder.from(dispatchBatter.out(0))
      .via(builder.add(fryingPan.async()))
      .toInlet(mergePancakes.in(0));

    builder.from(dispatchBatter.out(1))
     .via(builder.add(fryingPan.async()))
     .toInlet(mergePancakes.in(1));

    return FlowShape.of(dispatchBatter.in(), mergePancakes.out());
  }));
```


section in docs

Patriks parallel pancakes

```
val pancakeChef: Flow[ScoopOfBatter, Pancake, NotUsed] =
  Flow.fromGraph(GraphDSL.create() { implicit builder =>
    import GraphDSL.Implicits._

    val dispatchBatter = builder.add(Balance[ScoopOfBatter](2))
    val mergePancakes = builder.add(Merge[Pancake](2))

    // Using two pipelines, having two frying pans each, in total using
    // four frying pans
    dispatchBatter.out(0) ~> fryingPan1.async ~> fryingPan2.async ~> mergePancakes.in(0)
    dispatchBatter.out(1) ~> fryingPan1.async ~> fryingPan2.async ~> mergePancakes.in(1)

    FlowShape(dispatchBatter.in, mergePancakes.out)
  })
```

Making pancakes together



Built in stages

Source stages

fromIterator, apply, single, repeat, cycle, tick, fromFuture, fromCompletionStage, unfold, unfoldAsync, empty, maybe, failed, lazily, actorPublisher, actorRef, combine, unfoldResource, unfoldResourceAsync, queue, asSubscriber, fromPublisher, zipN, zipWithN

Sink stages

head, headOption, last, lastOption, ignore, cancelled, seq, foreach, foreachParallel, onComplete, lazyInit, queue, fold, reduce, combine, actorRef, actorRefWithAck, actorSubscriber, asPublisher, fromSubscriber

Flow stages

map/fromFunction, mapConcat, statefulMapConcat, filter, filterNot, collect, grouped, sliding, scan, scanAsync, fold, foldAsync, reduce, drop, take, takeWhile, dropWhile, recover, recoverWith, recoverWithRetries, mapError, detach, throttle, intersperse, limit, limitWeighted, log, recoverWithRetries, mapAsync, mapAsyncUnordered, takeWithin, dropWithin, groupedWithin, initialDelay, delay, conflate, conflateWithSeed, batch, batchWeighted, expand, buffer, prefixAndTail, groupBy, splitWhen, splitAfter, flatMapConcat, flatMapMerge, initialTimeout, completionTimeout, idleTimeout, backpressureTimeout, keepAlive, initialDelay, merge, mergeSorted, mergePreferred, zip, zipWith, zipWithIndex, concat, prepend, orElse, interleave, unzip, unzipWith, broadcast, balance, partition, watchTermination, monitor

File IO Sinks and Sources

fromPath, toPath

Additional Sink and Source converters

fromOutputStream, asInputStream, fromInputStream, asOutputStream, asJavaStream, fromJavaStream, javaCollector, javaCollectorParallelUnordered

But I want to connect other things!

@doggosdoingthings

Alpakka

A community for Akka Streams connectors

http://github.com/akka/alpakka

Alpakka – a community for Stream connectors

In Akka

- TCP
- Actors
- Reactive Streams
- Java Streams
- Basic File IO

Existing Alpakka

Cassandra	Kafka	AMQP/ RabbitMQ
AWS DynamoDB	AWS SQS	AWS S3
MQTT	JMS	Azure IoT Hub
Files	FTP	SSE

Alpakka PRs

AWS Lambda	IronMQ
MongoDB*	
druid.io	
Caffeine	
HBase	

But my usecase is a unique snowflake!

GraphStage API

```
public class Map<A, B> extends GraphStage<FlowShape<A, B>> {
  private final Function<A, B> f;
  public final Inlet<A> in = Inlet.create("Map.in");
  public final Outlet<B> out = Outlet.create("Map.out");
  private final FlowShape<A, B> shape = FlowShape.of(in, out);
  public Map(Function<A, B> f) {
    this.f = f;
  }
  public FlowShape<A,B> shape() {
    return shape;
  }
  public GraphStageLogic createLogic(Attributes inheritedAttributes) {
    return new GraphStageLogic(shape) {
      {
        setHandler(in, new AbstractInHandler() {
          @Override
          public void onPush() throws Exception {
            push(out, f.apply(grab(in)));
          }
        });
        setHandler(out, new AbstractOutHandler() {
          @Override
          public void onPull() throws Exception {
            pull(in);
          }
        });
      }
    };
  }
}
```

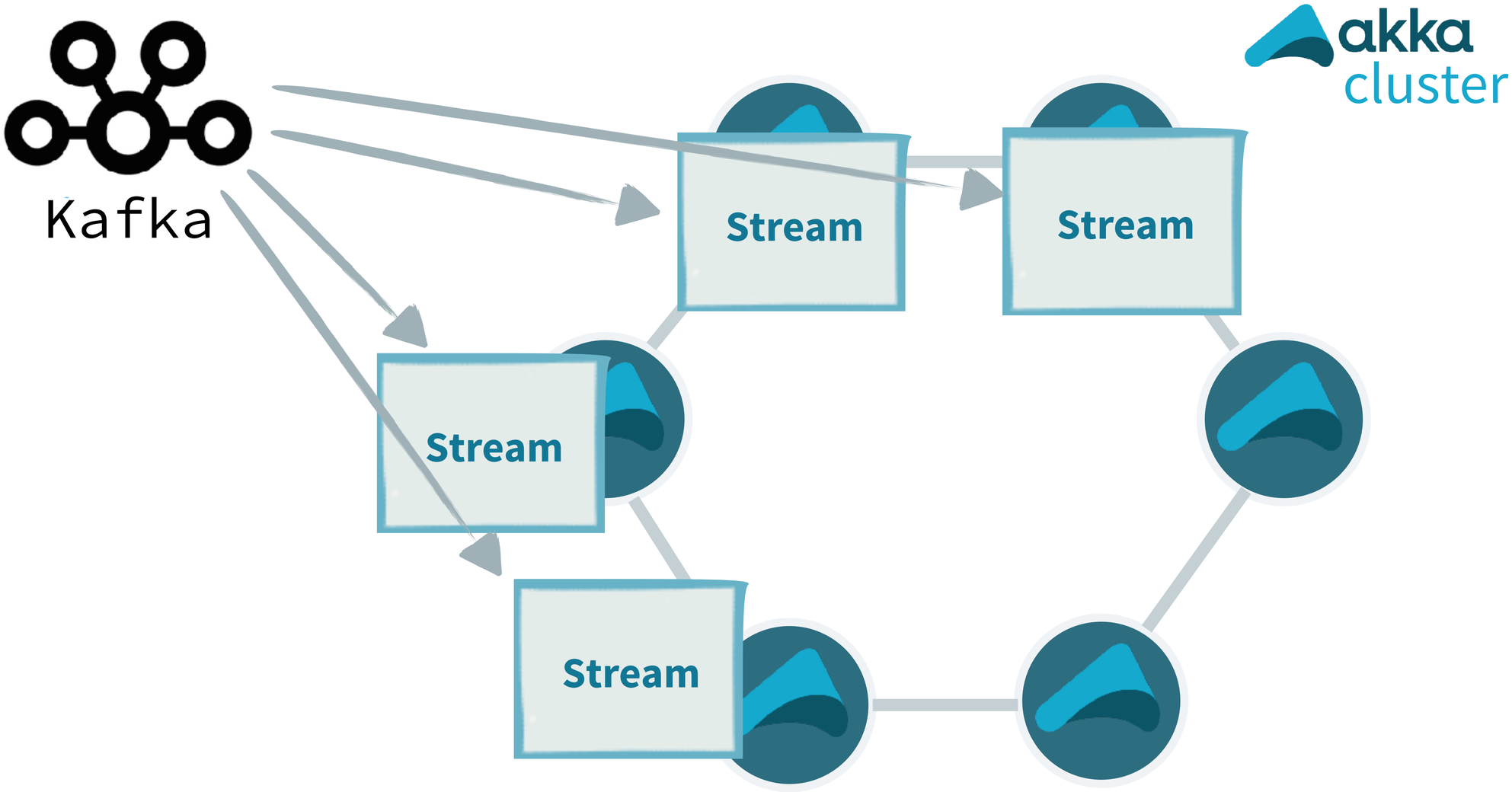

complete sources on github

GraphStage API

```
class Map[A, B](f: A => B) extends GraphStage[FlowShape[A, B]] {

  val in = Inlet[A]("Map.in")
  val out = Outlet[B]("Map.out")
  override val shape = FlowShape.of(in, out)

  override def createLogic(attr: Attributes): GraphStageLogic =
    new GraphStageLogic(shape) {
      setHandler(in, new InHandler {
        override def onPush(): Unit = {
          push(out, f(grab(in)))
        }
      })
      setHandler(out, new OutHandler {
        override def onPull(): Unit = {
          pull(in)
        }
      })
    }
}
```


complete sources on github

What about distributed/reactive systems?



The community

~200 active contributors!

Mailing list:

https://groups.google.com/group/akka-user

Public chat rooms:

http://gitter.im/akka/dev developing Akka

http://gitter.im/akka/akka using Akka

Easy to contribute tickets:

https://github.com/akka/akka/issues?q=is%3Aissue+is%3Aopen+label%3Aeasy-to-contribute

https://github.com/akka/akka/issues?q=is%3Aissue+is%3Aopen+label%3A%22nice-to-have+%28low-prio%29%22

Akka

http://akka.io

Thanks for listening!

Runnable sample sources (Java & Scala)

https://github.com/johanandren/akka-stream-samples/tree/jfokus-2017

@apnylle

johan.andren@lightbend.com

Lightbend