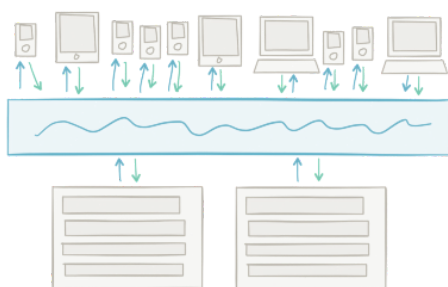


A Journey into Reactive Streams



Data in a modern system is in constant motion.

Rather than acting on data *at rest*, modern software increasingly operates on data in *near real-time*. Big data is less important than fast data, and fast data is crucial to fast knowledge. Stream processing is one way to help turn data of all sizes—big, medium, or small—into knowledge as quickly as

possible. As systems embrace *data in motion*, traditional batch architectures are being re-imagined as pure stream-based architectures. In these systems live data is captured, processed, and used to modify behaviour with response times of seconds or less. There's major business value in sub-second response times to changing information, rather than the hours, days, or even weeks a system with a traditional batch-based architecture may take to respond. Specifications such as Reactive Streams, and stream processing libraries such as Akka Streams, provide the standards and plumbing necessary to implement such systems effectively.

Systems that fit well within the abstraction of stream processing include ETL (extract, transform, load) systems, CEP (complex event processing) systems, and other various reporting/analytics systems. There's even a growing NoETL movement, similar to the NoSQL movement, that promises to upend the way organizations think of the way data flows through their systems. The possibilities are endless, limited only by the imagination of the software developers who work in data-intensive domains.

. . .

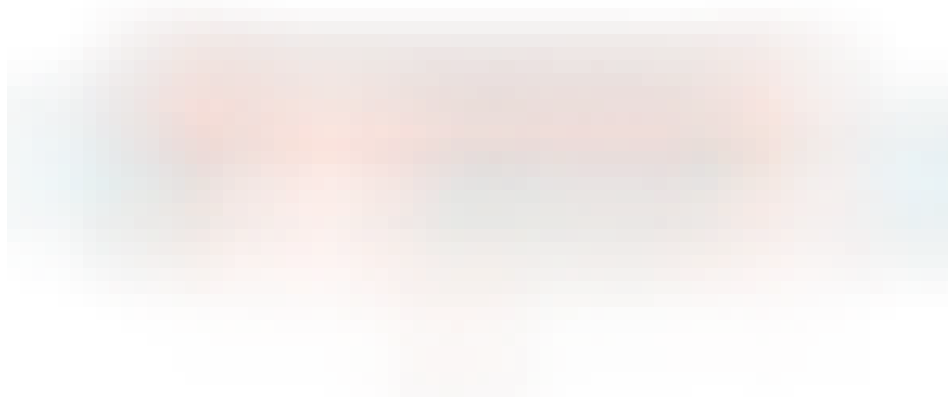


An Introduction to Streams

Streams are a series of elements emitted over *time*. The series potentially has no beginning and no end. Contrast that to arrays, which are a series of elements arranged in *memory*. Arrays have a beginning and an end.

Here's a scenario: you'd like to average a series of integer values. Consider the implications of working with an unknown number of elements; if you have no idea of the number of elements in a stream—and there's potentially no end to the stream—how and when do you emit the average? Do you emit a moving average? If so, how often to you recalculate the average? These are some subtle (and not so subtle) differences between working with streams and collections.

Let's visualize the series of elements above as a stream instead of an array. A publisher emitted 6 elements. We've attached to the stream in time to process 4 of them, but we detach before the 6th element is emitted.



This highlights an important distinction between streams and arrays—in a stream processing system it's not only possible, but rather expected, that not every single element of a stream will be visited. Not only will elements be emitted at different speeds—sometimes in bursts, other times in a trickle—but in many cases we're not guaranteed to process emitted elements at all. With an array it's possible to return a *final* average; an array is completely in memory. The elements of a stream may not even exist yet. When there's no concept of finality we need to think in terms of movement, ever-changing sets of data, and values that are always in motion. It's up to us to define how that motion looks and behaves.

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

Exploring Reactive Streams

Reactive Streams is a specification for library developers. Library developers code directly against the Reactive Streams API, while developers building business systems will work with an *implementation* of the Reactive Streams specification.

Reactive Streams started as an initiative in late 2013 between engineers at Netflix, Pivotal, and Typesafe. Some of the earliest discussions began in 2013 between the Play and Akka teams at Typesafe. At the time streaming was difficult to implement—Play had “iteratees”, but this was for Scala only and fairly difficult to grasp for many developers; Akka had Akka IO, but this was also too complex for business-level programming. The discussions continued, eventually leading to a chat between Roland Kuhn and Erik Meijer. This conversation was the seed of the Rx-based approach for passing data across an asynchronous boundary. Viktor Klang reached out to other members of the development community and a greater initiative began. The specification became a GitHub project shortly thereafter and reached 1.0 as of May 2015. If you’re interested in learning more about the history of Reactive Streams and how it came together, here’s an interview by Viktor Klang with Ben Christensen of Netflix, Stephane Maldini of Pivotal, and Dr. Roland Kuhn of Typesafe about their work on the specification.

The goal of Reactive Streams is to bring the level of abstraction up a notch—rather than business developers having to worry about the low-level plumbing of handling streams, the spec itself defines those problems. It’s then up to library developers to handle most of those problems for us.

We set out to define a local protocol that avoids to overwhelm data recipients without having to block any Threads, but in the end the scope in which we apply it is much larger: when streaming an HTTP payload to or from a server the natural back-pressure propagation of TCP should seamlessly integrate with the stream transformations performed on the data before or after sending it over the network.—Dr. Roland Kuhn, Typesafe

Even if you’re not a library developer working on a Reactive Streams compliant library, it’s very useful to understand the Reactive Streams

initiative. It aims to define and solve some of the most challenging problems in the space of stream processing.

Adding to the above, I think we have all been interested in two key points. First to ensure that stream implementations could exist without any unbounded buffering and switch between reactive and interactive models automatically based on consumption rate. Second, we wanted a solution that could allow interop between libraries, systems, networks and processes.—Ben Christensen, Netflix

Let's begin to explore some of the key concepts and challenges that Reactive Streams aims to address.

. . .

Consider two processes where one asks the other for something and synchronously expects the answer back. If the second process does not provide an answer, then the first process is broken.—Clarification of “Asynchronous Boundary”, Roland Kuhn

The concept of an *asynchronous boundary* is at the core of the spec. In essence, an asynchronous boundary decouples components of your system.

If you're unfamiliar with terms such as asynchronous, synchronous, blocking, or non-blocking, I recommend reading a blog post by Jamie Allen. He uses the example of washing dishes to explain each of the terms above.



Synchronous, sequential function calls.

First, let's explore *synchrony*. Assume we have three functions, **f1**, **f2**, and **f3**. Each is processed in order, one after the other. Not only are we tied to a particular order of execution, but the success of each is at the mercy of the other. For instance, let's say **f1** performs some kind of blocking operation such as persisting to a database. This blocks **f2** and **f3** from making

progress until **f1** completes. Therefore, the total execution time of this entire flow is the sum of the response time of each function (plus context switching).

Not only does synchrony and coupling create a maintenance burden, but it reduces the ability to craft fully responsive systems. They tend to embrace “responsiveness anti-patterns” that impacts scalability and resilience, such as running expensive computation on the same thread as the caller, relying on a successful response from a recipient, etc.

For instance, a failure in **f1** will prevent **f2** and **f3** from executing at all. Because **f1** performs a risky operation (persisting to a database) it shouldn't be *expected* to complete—the system should be designed in such a way that it can operate normally, albeit with limited functionality, when **f1** inevitably fails. Synchrony also prevents **f2** and **f3** from executing in parallel. Assuming that **f2** and **f3** are pure functions with no side effects (such as blocking IO) and don't require the output from **f1**, there's no reason why they can't execute in parallel while **f1** is blocked.

Take another example of synchrony, the single-threaded event loop. This is typical of event-based frameworks such as Node.js—events are emitted, queued, and processed by an event handler in a synchronous loop, and the recipients of those events are anonymous function callbacks. Bottlenecks occur when blocking occurs on the same thread as the event loop, such as one of the callbacks initiating a trip to the database.

In order to fully exploit the resources at our disposal, namely multi-core CPU architectures and the network, we need a completely different model in order to decouple the components of our system.

Asynchrony is needed in order to enable the parallel use of computing resources, on collaborating network hosts or multiple CPU cores within a single machine.
—<http://www.reactive-streams.org>



Composing functions rather than focusing on the sequence of operations.

When we think in terms of how data structures are *composed* rather than the *sequence in which they execute*, we clearly realize performance gains. For instance, if **f2** and **f3** execute and complete in parallel to **f1**, by the time **f1** is finished executing the entire result is ready. Rather than the execution time being the sum of all response times, it's the response time of the longest running operation.

Blocking calls aren't the true enemy, *synchronous* blocking calls are.

. . .

Back pressure is an integral part of this model in order to allow the queues which mediate between threads to be bounded. — <http://www.reactive-streams.org>

Another main objective of the Reactive Streams specification is to define a model for *back pressure*, a way to ensure a fast publisher doesn't overwhelm a slow subscriber. Back pressure provides *resilience* by ensuring that all participants in a stream-based system participate in *flow control* to ensure steady state of operation and graceful degradation. This implies interoperability between libraries.

The rest of this section expects some familiarity with the *publish-subscribe* pattern.



The ideal paradigm for a stream processing system is to *push* data from the publisher to a subscriber. This allows the publisher to operate as fast as possible, but it's easy to imagine a scenario in which part of our system can become flooded as

data flows full force. Push works well when subscribers are faster than the publisher. However, it's fairly difficult to predict from one moment to the next whether a subscriber will be faster than a publisher over a period of time. Unfortunately, most stream processing systems force us to choose between push or pull at development time.

We have a few options for dealing with this problem.

The first solution is to *pull*. This protects us from overwhelming slower subscribers. It also wastes system resources when subscribers are faster than the producer because it prevents a fast publisher from running at full speed when subscribers have enough capacity to keep up.



Another option is to increase the buffer size of subscribers. This is possible, but not always realistic. Having unlimited memory would be nice, but we'd have to increase the buffer size of queues throughout our entire system. Even one slow subscriber creates the risk of a cascading failure

through the entire system if that subscriber fails due to being flooded, even *after* increasing buffer size. Strategically increasing buffer sizes—either through memory increases or by using a distributed queue such as Kafka—does make sense, but applying that strategy to every step in our entire flow doesn't.



Another option is to simply drop elements. Working in stream processing gives us some flexibility for dropping what can't be handled, but it's not always appropriate. Imagine a system with wildly varying differences between the volume of elements

emitted from a fast publisher, e.g, a publisher emitting 100 elements per second, versus a slow subscriber, e.g, a subscriber with a capacity of handling 1 element per second. That's a huge amount of dropped elements! Now imagine we scale-out dynamically by provisioning resources in the cloud, and the subscriber is suddenly capable of handling 200 elements per second. We went from dropping elements to processing all of them. The

consistency of our results will vary wildly depending on conditions at run-time, which in some domains is acceptable, but in others is completely unacceptable.

Back-pressure may cascade all the way up to the user, at which point responsiveness may degrade, but this mechanism will ensure that the system is resilient under load, and will provide information that may allow the system itself to apply other resources to help distribute the load.—The Reactive Manifesto



Demand flows upstream. Elements flow downstream.

What we need is a bi-directional flow of data—elements emitted downstream from publisher to subscriber, and a signal for demand emitted upstream from subscriber to publisher. If the subscriber is placed in charge of signalling demand, the publisher is free to safely push up to the number of elements demanded. This also prevents wasting resources; because demand is signalled asynchronously, a subscriber can send many requests for more work before actually receiving any.

The demand signal doesn't terminate at the next upstream step, but rather flows all the way back through the flow to the original source. Propagating back pressure all the way through the entire flow gives the system a chance to respond to excessive load; perhaps a specific Reactive Streams library is designed to spin up new subscribers on different threads to handle the volume and load balance elements between those subscribers. A different library may even balance the load across a networked cluster of subscribers. This type of elasticity would be up to the library you choose; Reactive Streams does not provide these strategies, the implementations do. Reactive

Streams provides the hooks necessary for library developers to implement them.

Reactive Streams is *effectively push-based*. It has a single mode of operation, which works like a pull-based system when the subscriber is slower and a push-based system when the subscriber is faster. This is dynamic and changes in near real-time. To understand this behaviour let's explore the Reactive Streams API.

The full API is shown below:



```
1 public interface Processor<T, R> extends Subscriber<T>, Pub
2
3 public interface Publisher<T> {
4     public void subscribe(Subscriber<? super T> s);
5 }
6
7 public interface Subscriber<T> {
8     public void onSubscribe(Subscription s);
9     public void onNext(T t);
10    public void onError(Throwable t);
11    public void onComplete();
12 }
13
14 public interface Subscription {
15     public void request(long n);
16     public void cancel();
17 }
```

ReactiveStreamsAPI.java hosted with by GitHub

[view raw](#)

Of particular interest is **request(long n)** in the **Subscription** interface. This is how a subscriber signals demand for work. When a subscriber signals demand by invoking **request(1)**, that's effectively a *pull*. When a subscriber signals demand by requesting more elements than the publisher is ready to emit, e.g, **request(100)**, that changes the flow to *push*.

Ultimately, the primary value of back pressure is to ensure resilience across all compliant libraries. Interoperability between libraries, all sharing a common philosophy and vision, helps avoid serious problems that are almost unavoidable without a common strategy.



*A lack of back pressure will eventually lead to an Out of Memory Exception (OOM), which is the worst possible outcome. Then you lose not just the work that overloaded the system, but **everything**, even the stuff that you were safely working on.—Jim Powers, Typesafe*

. . .

Let's "go with the flow" and dive in a little deeper. A Reactive Streams compliant library can be used to process streams linearly or in a graph.



A simple graph, demonstrating a split and join.

In this example, function **f1** is applied to incoming data from a *source*. A source can be anything, such as:

- SQL queries
- Event Log systems
- HTTP-based services
- TCP sockets
- Staging directories in file systems (e.g., FTP servers)
- Messages queues like Kafka
- Etc

The output of **f1** is *broadcast* (duplicated) to two sub-streams and the elements processed by **f2** and **f4**. This is called a “split” or a “fan out”. The output from both functions are then *merged* before being processed by **f3**. This is called a “join” or a “fan in”. The output of **f3** is then sent to a *sink*.

If you cannot solve a problem without programming, you cannot solve a problem with programming. —Klang's Conjecture

The diagram above points out an interesting side-effect of working with streams; data, graphs, and flows are at the heart of designing a stream processing system, which is a natural fit for many of the problems we solve with software. The above diagram looks like many “back of the napkin” sketches I’ve drawn over the years.

The following is an implementation of the blueprint in Scala using Akka Streams. This article is not an in-depth tutorial on Akka Streams, so be patient if the following syntax looks unfamiliar. If you’re interested in executing this code, feel free to download the Typesafe Activator which includes everything you need to get started with Akka Streams.

```
1 object GraphExample {
2   import akka.actor.ActorSystem
3   import akka.stream.ActorMaterializer
4   import akka.stream.scaladsl._
5   import FlowGraph.Implicits._
6   import scala.util.{ Failure, Success }
7   import scala.concurrent.ExecutionContext.Implicits._
8
9   implicit val system = ActorSystem("Sys")
10  implicit val materializer = ActorMaterializer()
11
12  def main(args: Array[String]): Unit = {
13
14    val out = Sink.foreach(println)
15
16    val g = FlowGraph.closed(out) { implicit builder =>
17      sink =>
18        val in = Source(1 to 10)
19
20        val bcast = builder.add(Broadcast[Int](2))
21        val merge = builder.add(Merge[Int](2))
22
23        val f1, f2, f3, f4 = Flow[Int].map(_ + 10)
24
25        in ~> f1 ~> bcast ~> f2 ~> merge ~> f3 ~> sink.inlet
26            bcast ~> f4 ~> merge
27      }.run()
28
29    // ensure the output file is closed and the system shut
30    g.onComplete {
31      case Success(_) =>
32        system.shutdown()
33      case Failure(e) =>
34        println(s"Failure: ${e.getMessage}")
35        system.shutdown()
36    }
37  }
38 }
```

StreamExample.scala hosted with [by GitHub](#)

[view raw](#)

For those who aren't familiar with Scala, ignore much of the code except the two lines below.

1	<code>in ~> f1 ~> bcast ~> f2 ~> merge ~> f3 ~> out</code>
2	<code> bcast ~> f4 ~> merge</code>
<code>StreamCombinators.scala</code> hosted with by GitHub view raw	

Note how similar those lines of code look to the blueprint. This gives developers a powerful abstraction for thinking about the problem rather than the plumbing. For instance, ETL systems are already designed in a very similar manner—they ingest data from a variety of sources, operate on the values, and output some type of result. The biggest difference is that the code that defines those systems is riddled with plumbing concerns, making the solution itself hard to see. They also typically ingest data at rest, while stream-based systems ingest data in near real-time.

. . .

Reactive Streams Implementations

When you're ready to build your own stream processing system you'll need to choose a library. Implementations of the Reactive Streams specification are compatible with each other, which is an obvious benefit of choosing a Reactive Streams compliant library. You can even integrate different stream processing systems together that use different libraries as long as they comply with the specification.



Reactive Streams is the glue. Compliant libraries communicate via the Reactive Streams protocol.

A few Reactive Streams implementations are listed below:

- RxJavaReactiveStreams with RxJava 1.x or RxJava 2.x
- Project Reactor
- Vert.x
- Akka Streams
- Slick

. . .

With this conceptual overview of streams in general and the Reactive Streams spec you're one step closer to building the next generation of systems. Choosing a new set of tools is only one small part of a much bigger shift in how we craft software. Innovating requires bold and decisive action. It requires learning. It requires imagination. It requires change.

While the asynchronous boundary is about decoupling in time, what Reactive Streams does not yet give us is decoupling in space—*distribution*. This would allow us to distribute load across nodes and clusters, ideally with *location transparency*. If these terms are unfamiliar I invite you to read an article about reactive programming that I wrote last year.

The future becomes very exciting indeed when libraries such as Spark Streaming, Cassandra, and Kafka begin to implement the spec. We're not there yet, but the move towards streaming is inevitable. I wouldn't be surprised to see these and more libraries become Reactive Streams compliant in the future.

I invite you to check out the next part in this series, Diving into Akka Streams, where we explore Akka Streams in more depth.

If you're eager to start coding right away I highly recommend downloading Typesafe Activator. It contains everything you need to get started with Akka, Play, and Scala, including helpful tutorials.

Happy hacking! (And hAkking!)

About The Author

Kevin Webber is Enterprise Advocate at Lightbend. He's passionate about helping large organizations transition from heritage architectures to real-time distributed systems that embrace the principles of reactive

programming. In his spare time he organizes ReactiveTO and Programming Book Club Toronto. He rarely writes about himself in the third-person, but this is one of those moments.