WHITE PAPER

AN INTRODUCTION TO

Reactive Streams, Akka Streams and Akka HTTP

FOR ENTERPRISE ARCHITECTS

Lightbend

Table of Contents

Introduction to the world of fast, streaming "data in motion"

For many enterprises, the "big" in Big Data is less about the overall volume of data and much more about the need for speed in shuffling data around in real time. As vendors at every layer of the enterprise stack vie for their place in the evolving Big Data ecosystem, one of the busiest battlegrounds so far may be the back-end data movement and logic between systems.

Today we can see three major trends driving data volumes, velocity and variety at unprecedented and exploding levels that threaten traditional datacenters with a literal data deluge:

1. Internet of Things (IoT)
2. Real-time, data-driven customer interaction solutions
3. Real-time business insight platforms

By 2020, Gartner estimates that over 20 billion connected devices will be online, and that getting business value from this new big data opportunity **will be a $263 billion market**.

We use the term "Fast Data" to describe the trend that enabled this fundamental shift from "data at rest" to "data in motion" to happen. Our data used to be offline—now it's online. If we look at it, we can see three phases of data architectures:

- **Stage 1 —** In the early days, Big Data was still "data at rest", stored in HDFS or similar with offline batch processes crunching the data over night, often with hours of latency.

- **Stage 2 —** Later it became obvious that enterprises need to react in real-time to "data in motion"—to capture the live data, process it and feed back the result into the running system, with seconds and sometimes even sub-second response times. However this produced doubled the level of complexity by adding a "speed layer" on top of a "batch layer", which needed different models and data processing pipelines.

- **Stage 3 —** Nowadays, enterprises are fully embracing "data in motion" and, for most use cases and data sizes, moving away from the traditional batch-oriented architecture altogether towards pure-stream processing architecture using tools like **Spark**, **Kafka**, **Cassandra**, **Riak**, **Mesos** and others.

One of the main drivers of this movement is **Reactive Streams 1.0.0 for the JVM**, an effort to define a standard for passing streams of data between threads in an asynchronous and non-blocking fashion. This is a common need in Reactive systems, in which the volume of streaming "live" data to process is not predetermined.

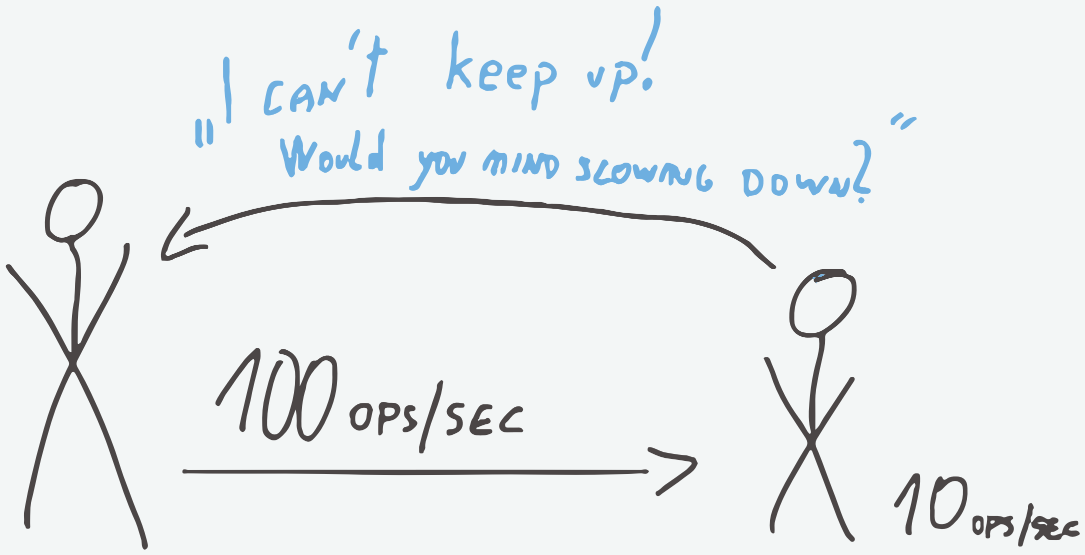
Engineers from Lightbend are among the founding members of the Reactive Streams initiative, with Akka Streams (which includes Akka HTTP) at the forefront of this paradigm shift to "data in motion".

A brief review of Reactive Streams

"I think we have all been interested in two key points: first, to ensure that stream implementations can 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
Senior Engineer at Netflix, Reactive Streams co-founder, creator of RxJava

As you can probably guess, handling streams of data—especially "live" data whose volume is not predetermined—requires special care in an asynchronous system. The most prominent issue is that resource consumption needs to be controlled in such a way that fast data sources don't overwhelm the stream destination. Asynchrony is needed in order to enable the parallel use of computing resources and collaborating network hosts or multiple CPU cores within a single machine. In other words, **back-pressure is an integral part of this model** in order to allow the queues which mediate between threads to be bounded.



Back-pressure communicates workload levels so that data passing never faces bottlenecks on either side

The main goals of Reactive Streams

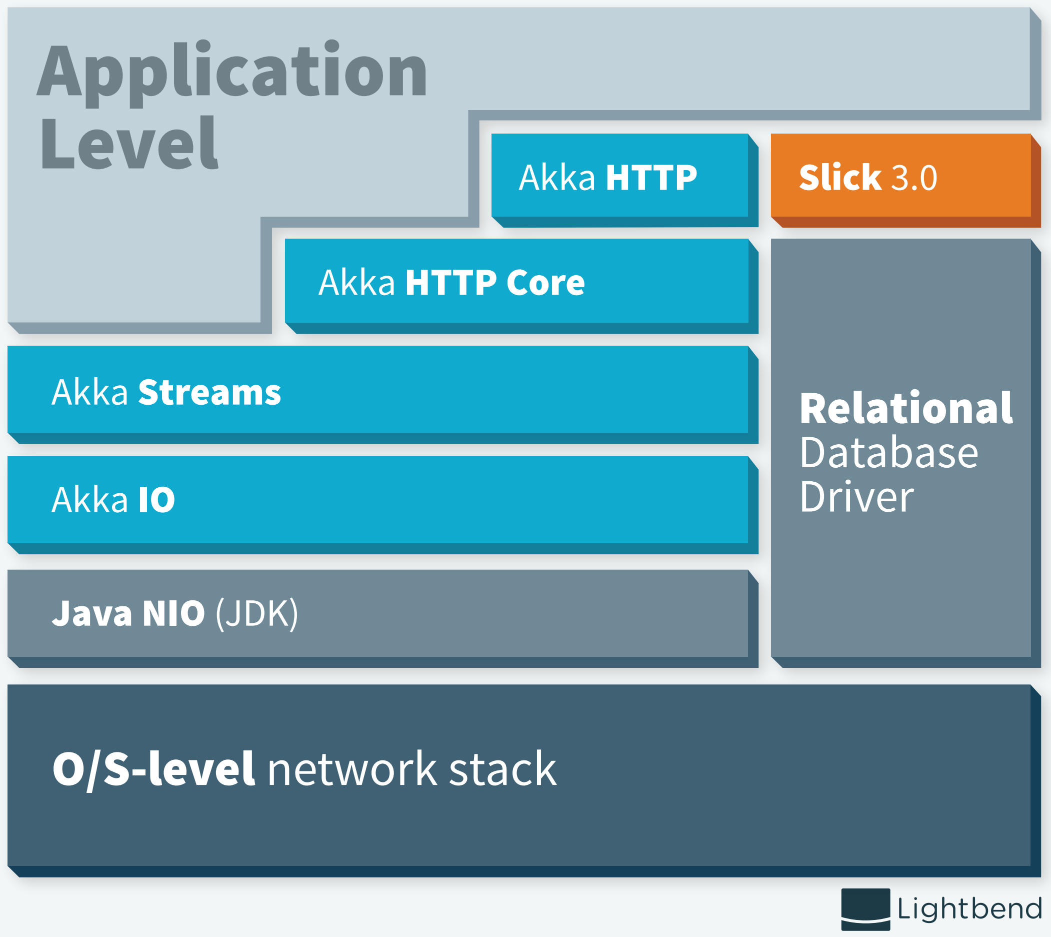
- Govern the exchange of stream data across an **asynchronous boundary**—think passing elements on to another thread or thread-pool—while ensuring that the receiving side is not forced to buffer arbitrary amounts of data.

- Highlight the benefits of asynchronous processing, which would be negated if the communication of back pressure were synchronous (see also the **Reactive Manifesto**), therefore care has to be taken to mandate fully non-blocking and asynchronous behavior of all aspects of a Reactive Streams implementation.

- To allow the creation of **many conforming implementations**, which by virtue of abiding by the rules will be able to interoperate smoothly, preserving the aforementioned benefits and characteristics across the whole processing graph of a stream application.

This work was done in collaboration between engineers from companies like Lightbend, Netflix, Red Hat, Pivotal, Oracle, Twitter and others. From the Lightbend side of things, our contributions to the initiative include **Slick 3.0**, a modern database query and access library for Scala, and Akka Streams (which includes Akka HTTP). In this infoguide, we'll look at these latter two technologies.

Where Akka Streams and Akka HTTP fit in your stack



This image above shows a general layout of an Akka-based system. The goal is to make processing is asynchronous and back-pressured, not only enabling a more elastic scalability but also saving you from overloading your systems. Akka Streams brings this to each layer, letting you safely and asynchronously utilize both low-level (e.g. TCP) and high level (e.g. REST API) protocols throughout the entire system.

Akka Streams can be thought of as the underlying infrastructure, to be used directly to write to TCP or more convenient APIs without sacrificing the performance gained by asynchronous processing. With APIs that are built with Akka Streams, you can apply back-pressure to a fast writer when needed while enjoying the safety of such a system based on the rigorously-tested Reactive Streams specification.

NOTE: Here is where combining Akka Streams with Slick, another technology in the Reactive Streams 1.0.0. specification results in an incredibly tight fit that will let your system go as fast as possible while retaining safety (and without users having to fine-tune anything).

The goals (and scope) of Akka Streams

"We think that the streaming advances in Akka—including the DSL, the higher level abstractions, and decoupling definition from execution—make Akka Streams a very compelling toolkit for developers facing these challenges."

Jonas Bonér
Creator of Akka and CTO of Lightbend

Akka Streams is a library for multi-core bounded memory data processing, designed from the ground up to take advantage of the available parallelism on current platforms. This is achieved by a fully-asynchronous, internal execution engine that avoids thread blocking; at the same time, it guarantees bounded memory usage by only keeping a subset of the processed data stream in-memory at any given time. This approach works even for real-time sources, like network connections, because back-pressure is automatically applied to inputs. As mentioned previously, the underlying model that makes this possible is the Reactive Streams specification.

This brings us to where Akka Streams fits into the daily toolbox of software developer. Unlike tools like Spark Streaming or Apache Storm, Akka Streams is a library that can be used for adding back-pressure to small utilities or even full-blown server applications without any investment in infrastructure. Akka Streams does not provide automatic scaling or distribution capabilities, but rather focuses on HTTP, TCP and support for other lower-level features that enable users to build more complex applications around it. This means you can use it as a very efficient tool for implementing integration across all the other more targeted frameworks.

Stream-like processing has been available on the JVM as the Java interfaces InputStream and OutputStream. Unfortunately, these models do not cope with the requirements of modern, multi-core architectures because they rely on blocking calls and do not have any built-in support for parallel processing. Retrofitting back-pressure into a system is not only extremely error-prone but also requires a lot of mostly manual work if it is at all possible without a complete rewrite.

Since Akka Streams is an implementation of the Reactive Streams specification, it fully integrates with other libraries implementing the standard and therefore enables access to a large ecosystem of other streaming libraries and connectors.

Below is a current view of the technologies found in Reactive Streams 1.0:

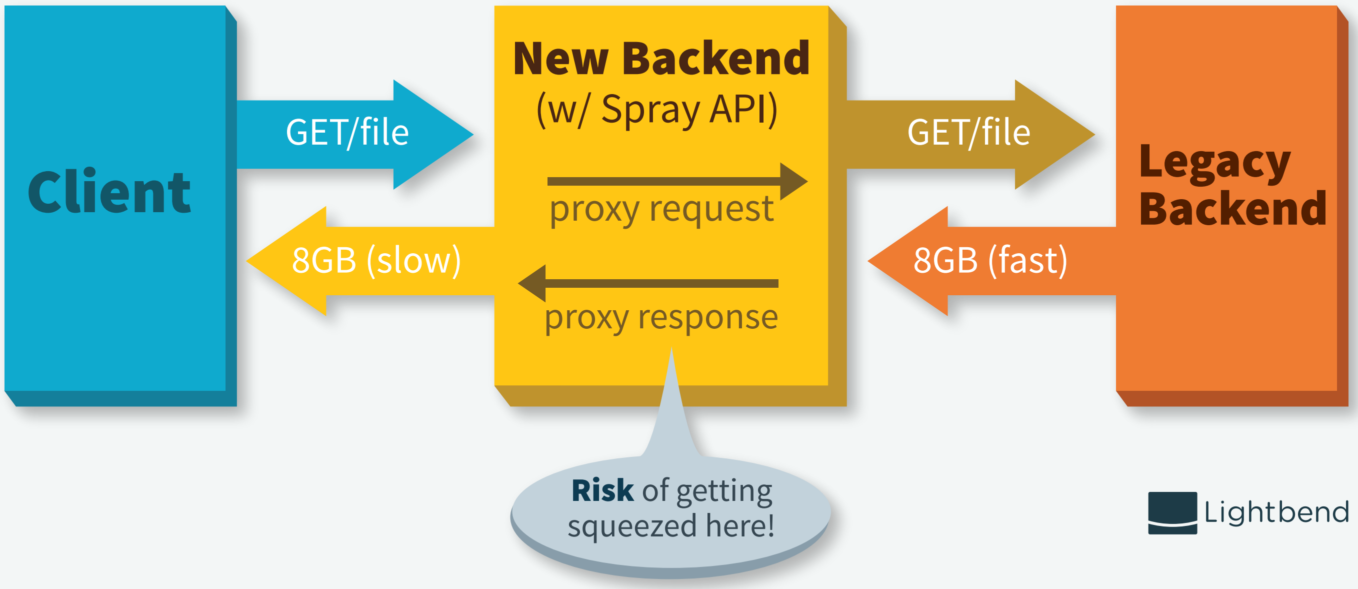
- **Akka** Streams (version 1.0)
    - See this **Activator template** and the **documentation**.
- **MongoDB** (version 1.0.0)
    - For the documentation see **here**.
- **Ratpack** (version 1.0.0)
    - See the "**Streams**" chapter of the manual.
- **Reactive Rabbit** (version 1.0.0)
    - Driver for RabbitMQ/AMQP
- **Reactor** (version 2.0.1.RELEASE)
    - For the documentation see **here**.
- **RxJava** (version 1.0.0)
    - See **github.com/ReactiveX/RxJavaReactiveStreams**.
- **Slick** (version 3.0.0)
    - See the "**Streaming**" section of the manual.
- **Vert.x 3.0** (version 3.0)

The goals (and scope) of Akka HTTP

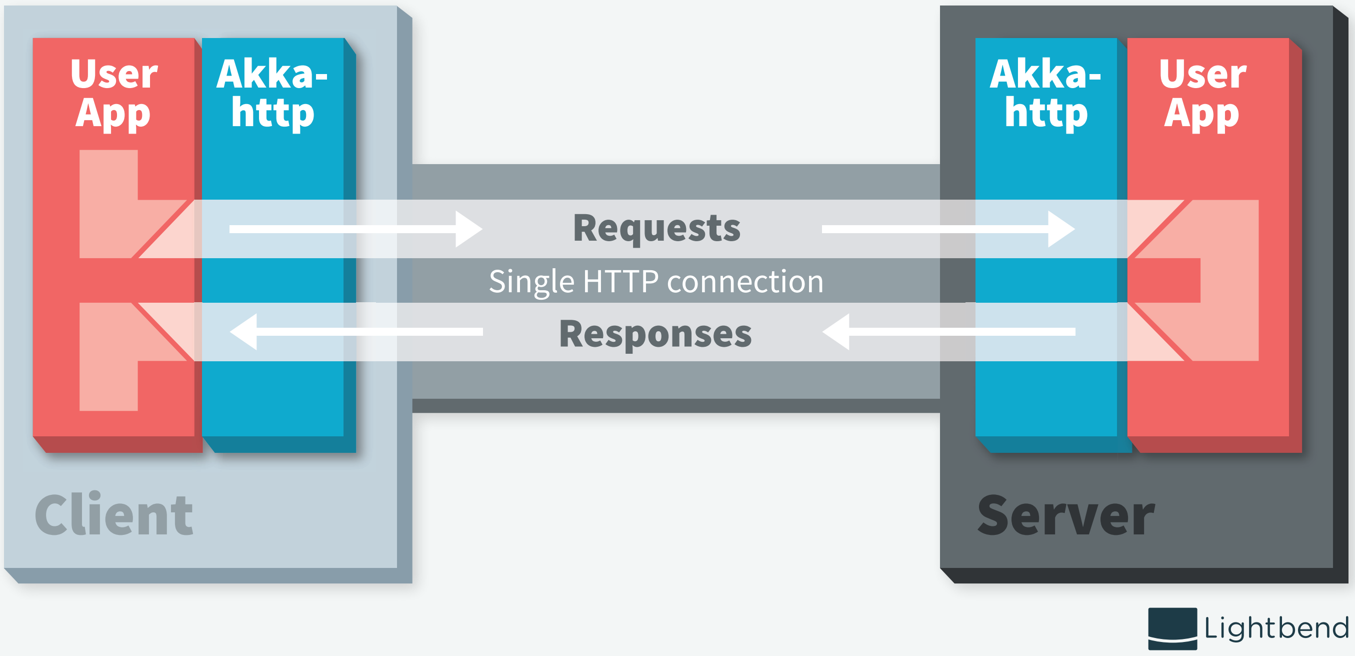
The successor to **Spray**, Akka HTTP is a fully HTTP 1.1 compliant server and client implementation that sits on top of Akka Streams. Unlike the previous actor-based interface of Spray, Akka HTTP provides an Akka Streams-based API that simplifies development of HTTP applications, especially when streaming or event-based functionality is necessary in cases like:

- WebSockets
- Server Sent Events (SSE)
- Large file uploads

Spray IO (before)



Akka HTTP (after)



The bounded memory guarantee of Akka Streams and its integration capability with other Reactive Streams implementations (i.e. Slick) opens new possibilities for serving HTTP results from large datasets without increased resource usage.

This was formerly a challenge with Spray, where handling of chunked requests is clunky and incomplete at times; however, now you can stream database queries directly as JSON-rendered results in bounded memory, which we demonstrated in **A Deeper Look at Reactive Streams with Akka Streams 1.0, Akka HTTP 1.0 and Slick 3.0**.

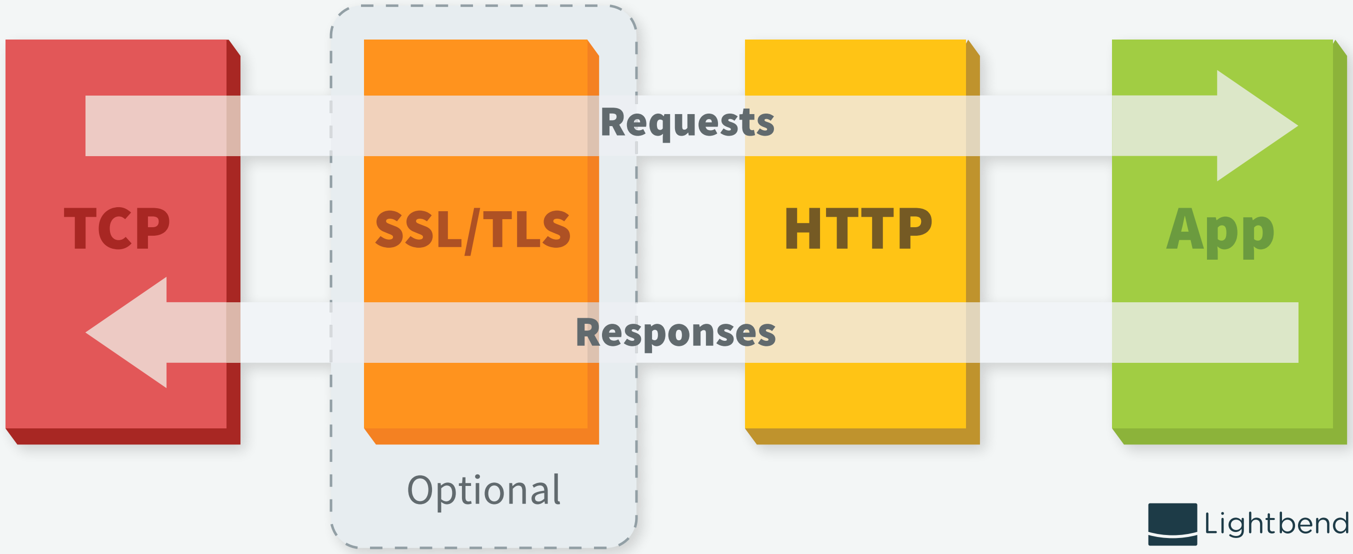
Akka HTTP also provides a rich Java API that is functionally equivalent to its Scala-targeted counterpart, but more tailored to Java usage patterns.

NOTE: At the time of this writing, Akka HTTP is version 1.0 and undergoing performance optimization and significant progress in this area will reduce the gap between other frameworks without sacrificing the convenience and load-resilience of the model itself.

How back-pressure works in Akka HTTP

A question many Enterprise Architects ask us: does back-pressure in Akka HTTP work with existing services, or do both sides of the connection need to be implemented using Akka for the back-pressure to work?

The good news is that TCP is a protocol that has back-pressure built into its very fabric. Since the same is true for Akka Streams itself, the only missing piece is faithful translation from the "language" of one to the other. The Akka IO and Akka HTTP modules take care of this in the background without any additional user logic. This also means that legacy clients will work the same as before.



In fact, any client that uses blocking APIs to write to a TCP socket, for example, will now properly respect the load of the server without any additional work.

This means that a slow or loaded processing step will now automatically regulate the transfer rate of the TCP connection through the Akka Streams fabric, which will automatically translate to longer blocking times at the client write site. Since HTTP is a protocol that is based on TCP, this applies to serving HTTP requests as well, file uploads for example will be properly throttled according to server load.

How does Akka Streams compare to other projects that have "Stream" in their names?

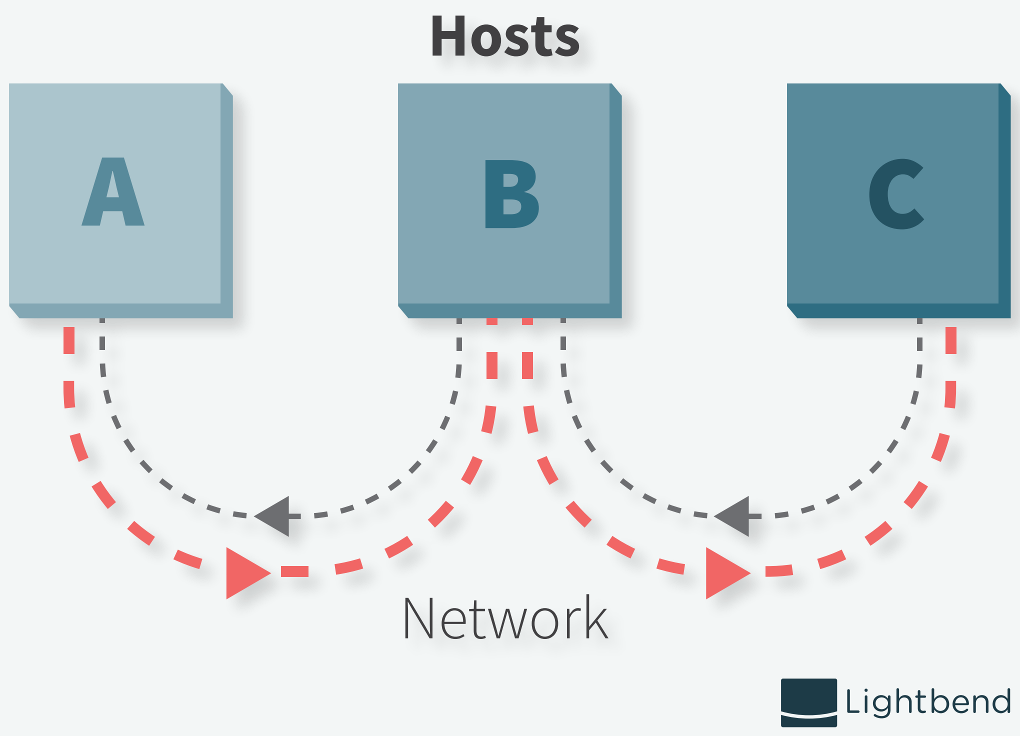
There are many existing projects that use "Streams" or "Streaming" in their name or description. Akka Streams (and the Reactive Streams specification) provides a set of already-in-the-JVM protocols carefully hidden from the user to efficiently transfer elements between concurrent processing entities, carefully regulating the flow of data elements to avoid overloading any of the processing steps.

So here is a quick breakdown about Akka Streams:

1. Akka Streams **does not** compete with Spark Streaming.
2. Akka Streams **is not** related to Java 8 Streams (i.e. collections for "bulk operations").
3. Akka Streams **is** strictly a library, not an application like Spark, Hadoop or Storm.

Let's compare it to Spark Streaming, which aims for streaming Big Data processing. Akka Streams does not compete with Spark, but rather works with it. In fact, recent collaboration between Databricks and Lightbend has led to **improving the back-pressure mechanisms used in Spark Streaming**, which was very much inspired by our lessons learnt from Reactive Streams. These technologies are better seen as complementary, where Akka Streams can provide significant integration and abstraction possibilities between these and other entities in a company infrastructure. In summary, Akka Streams can function as a "glue" to connect components together, while Spark Streaming is distributed computation engine for potentially-large data sets.

Akka Streams as a framework takes care of many more aspects than just composing streams of data together, and makes many assumptions about the nature of processing. We can look at TCP for example:



A TCP connection automatically propagates back-pressure over the network so if the server is still busy, clients are notified on the other end by the TCP protocol itself. This doesn't require anything special from the client, since this feature is provided by TCP itself and Akka Streams makes sure that server-side load is properly translated to TCP operations without breaking back-pressure.

The prime example of inter-operability here is, of course, with Akka HTTP, which directly works on top of Akka Streams. In addition, Akka 2.4 will ship with a new experimental module called **Akka Persistence Query**, which adds additional capabilities to **Akka Persistence** based applications.

Akka Streams lets you reuse data pipelines

Inside your Reactive applications, any kind of data transformation pipelines can be nicely expressed using Akka Streams. You'll get the benefit of a strongly typed pipeline which is asynchronous and can scale to multiple cores of your servers.

You also get the benefit of Akka Streams pipelines being reusable; for example, a member of the team can easily build a part of the pipeline, knowing what inputs and outputs it will have, and then you plug it together with the rest of the teams work on the rest of the pipeline. These parts can also be reused in other projects if they are generic (e.g. compression or analysis Flows, etc).

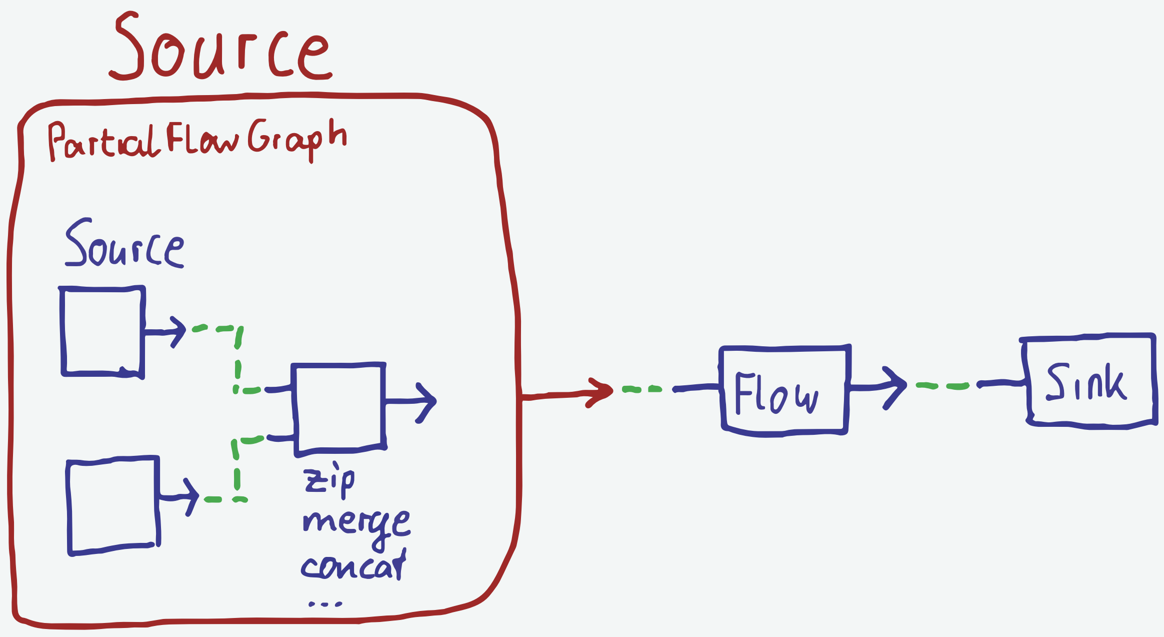
It also simplifies greatly the integration of asynchronous data producers and consumers as it makes sure that the rate of the consumption is properly taken into account and used to potentially throttle the producer side to avoid system overload and out-of-memory errors.

For Java enterprises, Akka Streams—like all modules of Akka (and the Lightbend stack in general)—comes with a Java API. This has been part of the project from its initial phases and is equivalent to the Scala API in terms of capabilities. The two DSLs are also fully interoperable with each other, reducing the potential frictions in a mixed language project.

Akka Streams for distributed applications

The main concern of the library is to provide a convenient and safe abstraction of in-JVM processing of sequences of events or data elements, similarly to Java's Input- and OutputStream, although with parallelism and concurrency in mind.

Communication channels across JVM instances are simply modeled as sources and sinks of elements (connectors/drivers for the appropriate channel) just like the Java library can model sockets as a pair of Input- and OutputStream instances.



Instead, we need a mechanism that uses bounded buffers, but prevents them from filling. In Akka Streams, *sources* are entities which can produce elements in a sequence, either on demand (e.g., a file source can serve the next binary chunk whenever requested) or delayed when necessary. Sources can be attached to various processing entities, then terminated by a sink that actually consumes the provided elements. The actual, regulated flow from the sources to the sinks is what is considered a *stream* in the Akka Streams mindset.

Akka Streams also provides various patterns for integration with Akka Actors, so users can mix-and-match the convenience of Streams with the flexibility of Actors. In fact, just like Futures in Scala, Akka Streams is a technology that helps you create highly efficient, concurrent systems. They both can help implementing Akka based applications but they are by no means a replacement for Actors.

In this view, Akka Streams is more of a programming abstraction rather than a distribution abstraction. This does not mean though that it cannot be used to implement such functionality, but instead of distribution baked into the core model, it can be provided as a set of libraries and frameworks on top of the basic abstractions.

Getting Started with Akka Streams & Akka HTTP

At the time of this writing, the current 1.0 release contains most of the necessary tools to start implementing load-resilient, highly concurrent server-client or data crunching applications. Nevertheless, there is plenty of work to be done.

One one front, we will keep adding more convenience utilities and tools expanding the flexibility of the library in response to the needs and experiences of our users using Akka Streams in practice, on the other front we will improve the stability and performance of the underlying machinery to reduce the cost of these abstractions to the possible minimum.

To get started you can read up all the details in the Akka Streams reference documentation for either **Scala** or **Java**. Or you can dive in head-first by checking out the sample Activator templates (available for both **Scala** and **Java 8**. If you're stuck or have any other questions you can reach out to the community via the official **akka-user mailing list** or the **akka/akka gitter channel**.

Final Thoughts: “Fast Data” in motion is gaining momentum

The fundamental shift in data today is towards low-latency, scalable, pure-stream processing architectures. A new approach is needed to understand how much data is ‘in motion’ and keeping the data flowing while limiting the resources that are consumed on the systems that the streams pass through.

Akka Streams, Akka HTTP and Slick are all founding technologies of the Reactive Streams initiative, supporting a standard for establishing back-pressure when passing streams of data between threads in an asynchronous and non-blocking fashion. This is needed in Reactive systems where the volume of streaming “live” data to process is not predetermined, but cannot be allowed to block and overload your systems.

Gartner predicts a quarter-of-a-trillion-dollar business opportunity for “Fast Data”, which is already alive in IoT, real-time data-driven customer interaction solutions, and real-time business insight platforms. These forces are driving a rapidly increasing volume, velocity and variety of data that threaten traditional data solutions with batch-oriented architectures.

It’s not “data at rest” anymore—it’s “data in motion”, and the journey begins with Akka Streams, Akka HTTP and other Reactive Streams technologies.

EXPERT TRAINING

Delivered On-site For Spark, Scala, Akka And Play

Help is just a click away. Get in touch with Lightbend about our training courses:

- Intro Workshop to Apache Spark
- Fast Track & Advanced Scala
- Fast Track to Akka with Java or Scala
- Fast Track to Play with Java or Scala
- Advanced Akka with Java or Scala

Ask us about local trainings available by 24 Lightbend partners in 14 countries around the world.

CONTACT SALES

Learn more about On-site training