ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-THPHA152

RENOVATION AND EXTENSION OF SUPERVISION SOFTWARE LEVERAGING REACTIVE STREAMS

M.-A. Galilée, J.-C. Garnier, K. Krol, T. Ribeiro, M. Osinski, A. Stanisz, M. Pocwierz, J. Do, A. Calia, K. Fuchsberger, M. Zerlauth, CERN, Geneva, Switzerland

Abstract

Inspired by the recent developments of reactive programming and the ubiquity of the concept of streams in modern software industry, we assess the relevance of a reactive streams solution in the context of accelerator controls. The promise of reactive streams, to govern the exchange of data across asynchronous boundaries at a rate sustainable for both the sender and the receiver, is alluring to most data-centric processes of CERN's accelerators. Taking advantage of the renovation of one key software piece of our supervision layer, the Beam Interlock System GUI, we look at the architecture, design and implementation of a reactive streams based solution. Additionally, we see how this model allows us to re-use components and contributes naturally to the extension of our tool set. Lastly, we detail what hindered our progression and how our solution can be taken further.

INTRODUCTION

A common issue with software systems for which maintenance and evolution stretched over years and multiple core developers is the emergence of a cluttered architecture. It occurs even in seemingly simple cases such as supervision software, where the base use-case is the acquisition, processing and exposition of data from operational devices.

The Beam Interlock System supervision GUI is one such example and as part of its renovation, a particular attention was paid to render its architecture more robust both in the short and longer terms. Reactive streams appeared as a promising solution to this endeavor:

- They provide an adequate model to the primary supervision software need, the transformation and consolidation of acquired data, which is then delivered to tailored user interfaces
- They can allow for flexible designs, which nevertheless promote coherent maintenance actions in the longer term.
- They can be combined into re-usable blocks, which can easily be built upon, even outside of the initial supervision scope.

THE REACTIVE STREAMS PARADIGM

"Reactive Streams is an initiative to provide a standard for asynchronous stream processing with non-blocking back pressure." [1]

Asynchronous streams

The stream concept matches well our base use-case: unbounded flows of data, acquired asynchronously, which must then be processed and delivered to client layers. This abstraction is powerful, as it makes very little assumption: any kind and amount of data can be transmitted, at any rate, any process can be applied to it, it can come from any source, etc. It only matters that we can consider data as originating from a source and emitted in sequence. Streams can then be used to model everything in between the acquisition layer and the delivery and fit accordingly.

Further, streams are defined with simple yet expressive semantics. This basic language allows to accurately describe what processing is applied to the data, and how the different streams relate to one another. A very useful tool to visualize streams and their semantics is the marble diagrams [2]. Figure 1 shows an advanced example of marble diagrams, modeling the combination of two streams in order to analyze together the data from both sources, on specific events.

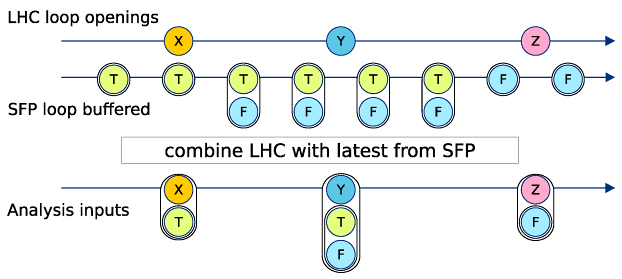

Figure 1: Complex case, modeled visually.

When implementing complex use cases, involving timing considerations or multiple streams, this kind of representation became essential. It served first as a design tool to frame the problems, then as a visual documentation.

Staying Reactive

Backpressure is what happens when the consumer of a stream is slower than the data sources. Unhandled backpressure can have various and likely harmful consequences, depending on the actual data sources and implementations involved. These range from permanent data loss, to software failure or even to frozen obstructed real-time controls devices.

Reactive streams as a concept only make the backpressure explicit to handle and do not magically solve it, but prevent ignoring. Thankfully, reactive streams libraries are provided with built-in strategies to deal with backpressure, and high-level ways to selectively apply these strategies.

This flexibility is a key-feature when different consumers are involved, each with different requirements, and yet we cannot have the system integrity impacted by the supervision software's backpressure.

ARCHITECTURE & DESIGN

While the reactive streams paradigm is flexible in what it can express, it forces to describe our data processing in specific and finite ways. With this constraint, the processing emerges as a pipelining structure, representative of the complexity and nature of the operations applied to the data. Nevertheless, each step of the processing can be arbitrarily simple (or complex...). Being a composition of single steps, the overall pipelining architecture can be elaborate, still flexible and transparent to scrutiny.

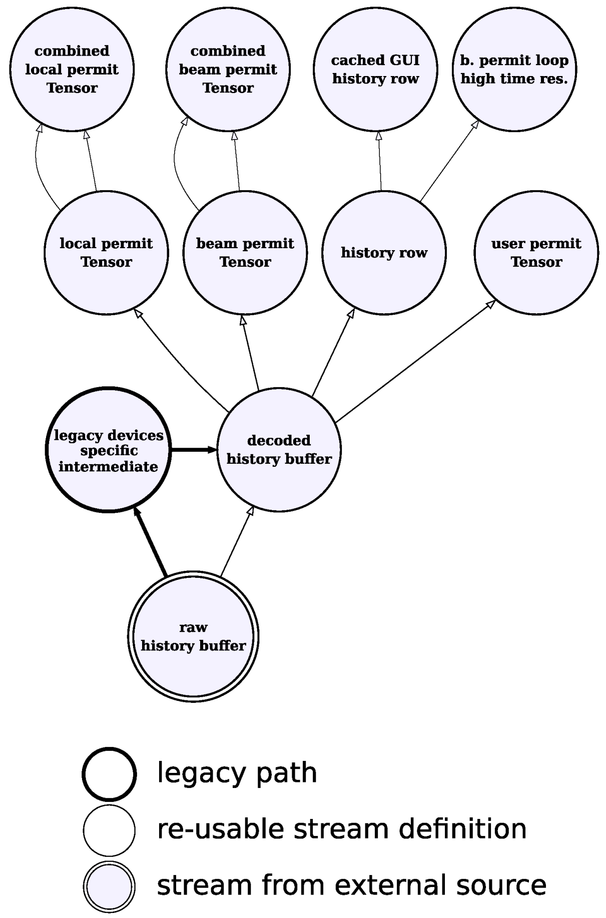
combined local permit Tensor
combined beam permit Tensor
cached GUI history row
b. permit loop high time res.
local permit Tensor
beam permit Tensor
history row
user permit Tensor
legacy devices specific intermediate
decoded history buffer
raw history buffer
legacy path
re-usable stream definition
stream from external source

Figure 2: A sample of the streams hierarchy within the renovated BIS software.

Instrumental to this type of architecture is the Streamingpool framework [3,4]. By fully decoupling the streams definition and materialization stages, one can reuse and compose streams definitions freely and materialize high-level streams without having to burden with the lower-level streams at all.

Some notable properties of the resulting architecture:

Clarity

The streams hierarchy reflects what processing the application does, and how. If all the processing is done through streams, then the hierarchy becomes an accurate map of the processing layer, making clear what components are relevant to the publication of each processed data type.

Modularity

The modularity property arises at two levels.

Within the processing layer, it becomes more natural to have fine granularity processing steps, instead of massive steps doing all the work in one-go. On top the aforementioned clarity, this brings the ability to fork off an existing processing chain, and avoid rebuilding the same processing steps from scratch.

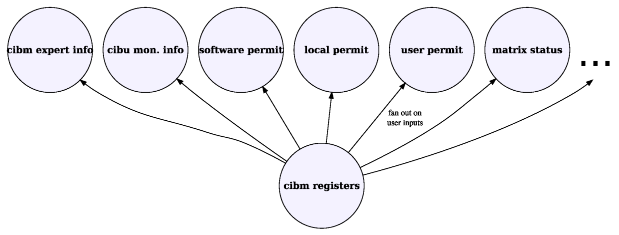

Figure 3: A single stream definition reused to build numerous higher level streams.

Then at the application scale, by having a clean separation between the acquisition, processing and user interface layers, these layers can be reused in different contexts. This is especially true of the processing layer, designed as a library from the beginning, and effectively reusable across multiple software pieces.

Flexibility

Our renovation effort is an iterative process, and this implies preserving backward compatibility to some degree, at least for some time. Such circumstances lead to having the architecture and code bent out of their original shape to accommodate for legacy, namely code warts. While unavoidable, these warts should not leak throughout the code base, neither inadvertently grow and hinder further maintenance and evolution.

As can be seen in Figures 2 and 3, our streams architecture is flexible enough to permit these legacy warts, yet keeps them contained. The "legacy devices specific intermediate" is used in practice in the same way as the "WartStreamId" showcased in Figure 4. By leaving warts clearly visible but unbound the rest of the code, we have an easy way to revisit the streams hierarchy and eventually cut out the legacy paths.

TAKEAWAYS

Cognitive Focus Transfer

Over the course of the renovation, our cognitive focus shifted significantly.

```
public StreamId<MyData> decodedDataStream(Device
     dev) {
  StreamId<MyData> dataStreamId =
    DerivedStreamId.derive(
      RawStreamHelper.rawDataStream(dev),
      DataConversionFactory.getConversionMethod(dev
           )
    );

  // code wart
  if (dev.isLegacy()) {
    return new WartStreamId(dataStreamId);
  }
  // => tightly contained

  return dataStreamId;
}
```

Figure 4: wart containment procedure.

At first, the reactive streams paradigm presents a steep learning curve. Coming from a foremost object-oriented background, thinking in terms of streams and taking a slightly more functional approach to programming required some time of adaptation and our focus was first drawn to the conceptual aspects of reactive streams. Luckily, helpful material on the subject is readily available [5, 6].

Quickly we switched from the concepts to the semantics of streams. We chose RxJava [7] out of the many existing streaming libraries ([8–10] to name a few others), for being the reference implementation and its continued development. Nevertheless, the same semantics are shared across the different libraries, so this particular effort can be expected to be roughly similar for any of them. As stated previously, marble diagrams [2] turned out to be an essential tool to effectively think and design with streams. Still, it took numerous tiny demos and proofs of concept to become comfortable with reactive streams in practice.

Past the conceptual and technical learning stages, we could focus more and more on the problems we meant to solve with streams. In practice, the question became "what" should our processing pipeline do, in terms of operations, rather than "how" it should be implemented. And from then on, for any further piece of processing, the "how" stayed solved: with streams, following the same composition patterns.

Complexity Revealed

When introducing a new paradigm and technology into a project, a crucial point is the assessment of the overhead it brings along. As mentioned, once the conceptual and technical aspects were addressed, streams actually reduced the overall complexity of the processing layer. More interestingly, modeling our problems within this paradigm brought up real issues we could have overlooked otherwise.

Working in an truly asynchronous context, we were forced to consider not only of the nominal cases, but edge cases as well. As these cases grow quickly in number, especially when data coming in from multiple streams must be put together, it is critical to assess both their impact and prevalence. Typically, acceptance tests should provide a reasonable coverage for these cases but cannot cover their full extent.

Failure also becomes an explicit aspect of programming. Multiple strategies to adequately handle errors are supplied by the different reactive streams libraries. Yet they must be carefully evaluated and chosen to match the context of the application: should the error be logged? propagated forward? can we let the stream collapse? if not, should it replay already published values or try to pick up from where it failed?

While dealing with this complexity upfront has a cost, it brings valuable insights on what our software can do and its limits. And when handled well, it also makes the software more robust and reliable. This contrasts with less strict programming approaches, where edge and failure cases risk being discovered much later in the software development cycle, typically during validation, or worse, during operation. The cost of dealing with them is then expectedly much higher.

Software Outreach

As we mentioned earlier, modularity is a key property of the structure of the renovated BIS GUI. In practice, it means we can effectively reuse the stream definitions from one application to another. The same code runs unchanged, plugged into different consumers. In this way we avoid the pervasive issue of having the same logic implemented over and over again, as less modular architectures tend to cause.

Furthermore, it brings homogeneity to the software stack —all software pieces share the same capabilities— and to the data layer —if high-level data is delivered to one client, it can be delivered as well to another client. Figure 5 shows how the processing layer we defined as part of the BIS GUI in the first place is reused across various other applications.

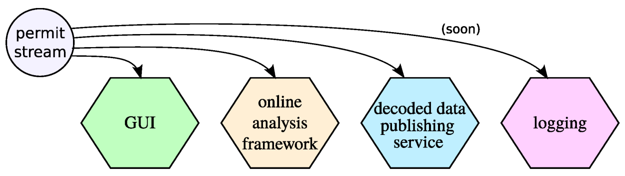

Figure 5: BIS permit stream definition reuse across applications.

Impediments

Implementing a streams design uncovered some drawbacks. Most notably:

- As we mentioned previously, paradigm shift has an upfront cost. While we expect this cost to be balanced with simpler maintenance in the longer term, reaching decent proficiency in the reactive streams paradigm required more time and focus than initially estimated.

- RxJava [7] is the reference streams library and seemed the most mature at the time we selected it. Still, it has not as polished as we expected. The implementation of less common cases —long-living streams being one— revealed few gaps in the documentation and sometimes bugs and partially implemented features. As well, other frameworks grew along and retrospectively could have been fitter choices. Project Reactor [9] integration into Spring 5 and Spring 5 serving of reactive streams over the network are recent advances very relevant to our developments.

- Streams promote modularity and flexible design. However, they do not prevent the leaking of implementation details to other layerse. It remains easy to entangle processing and acquisition, or client and processing, if no adequate data abstraction is established in advance.
- Behavior leaks from parent streams to their children streams and so on. In particular, it is important to understand, along the hierarchy of streams, how backpressure is handled. Different strategies produce different behaviors (memory growth, discarding of data items, slowing down of parent stream, etc.) and should be applied with care to ensure re-usability across different contexts and applications. If possible, allowing the end consumers to define the strategy themselves leads to greater flexibility.

PERSPECTIVES

The wider adoption of reactive streams in the Java ecosystem culminated with the recent integration of Flow [11] into Java 9. This considerably strengthens the position of reactive streams as a standard software solution and dependable technology.

Our own renovation effort continues, in order to extend our solution both in terms of functional and non-functional features.

Performance & Backpressure Optimization

A more delicate part of the processing layer is the tuning of the backpressure mechanisms, in order to balance the flexibility and re-usability of the solution with its performance. A particular case that remains to be solved is the design of a stream accommodating concurrently slow and fast consumers, while minimizing the adverse impact of these consumers on one another. This would greatly alleviate the behavior leak issue mentioned earlier.

Crossing the Copper Border

Having reactive streams functioning over the network will be the next step toward further software re-usability and homogenization. Streams would then be available as a service, potentially discharging the client layers not just from the complexity but from the processing load. Multiple efforts are already bringing reactive streams over the network: for instance, Spring 5 with Project Reactor [9] in the Java ecosystem, or the Reactive Sockets [12] community effort, at the protocol level.

Reaching the Hardware Front-end

The Java layer is one part of our software stack. Interoperability with lower-level supervision software, deployed to the hardware front ends, would be a natural extension of this effort. This could be brought in different manners, whether by re-using the same Java technologies directly on the devices [13] or through compatible, native technologies [12]. Still, such enterprise has a much wider scope than supervision software renovation and the availability of reactive streams technology fit for front end computers is only one concern among many.

REFERENCES

[1] `http://www.reactive-streams.org/`

[2] `http://rxmarbles.com/`

[3] A. Calia, K. Fuchsberger, *et al.*, “Streaming Pool - Managing Long-Living Reactive Streams for Java”, presented at ICALEPCS’17, Barcelona, Spain, Oct 2017, paper THPHA176, this conference."

[4] `https://github.com/streamingpool`

[5] `https://gist.github.com/staltz/868e7e9bc2a7b8c1f754`

[6] `http://www.lihaoyi.com/post/WhatsFunctionalProgrammingAllAbout.html`

[7] `https://github.com/ReactiveX/RxJava`

[8] `https://doc.akka.io/docs/akka/current/scala/stream/index.html`

[9] `http://projectreactor.io/`

[10] `http://vertx.io/`

[11] `http://download.java.net/java/jdk9/docs/api/java/util/concurrent/Flow.html`

[12] `https://reactivesocket.io`

[13] C. Cardin, J.-C. Garnier, *et al.*, “Real-Time Java to Support the Device Property Model”, presented at ICALEPCS’17, Barcelona, Spain, Oct 2017, paper THPHA153, this conference.