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, Switzerland

Motivation

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.
While planning the renovation of the Beam Interlock System (BIS) software layer, reactive streams appeared as a promising solution:

- They provide an adequate model to the primary supervision software needs.
- 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 streams paradigm

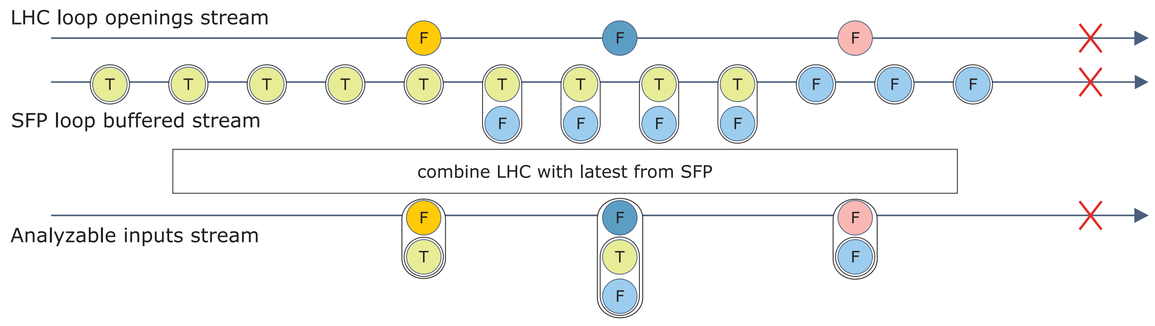

fig. 1 - a complex case, modeled visually

Streams are ideal to model flows of data, a recurring use-case in the machine controls environment: acquire, transform, deliver.
Basic data processing steps, such as conversion, filtering or buffering, are very simply expressed in marble diagrams [1]. Even more complex cases can likewise be visualized and conceived with clarity, such as this analysis helping validate a new BIS hardware loop (SFP), using the operational loop (LHC) as a reference.

Composable architecture

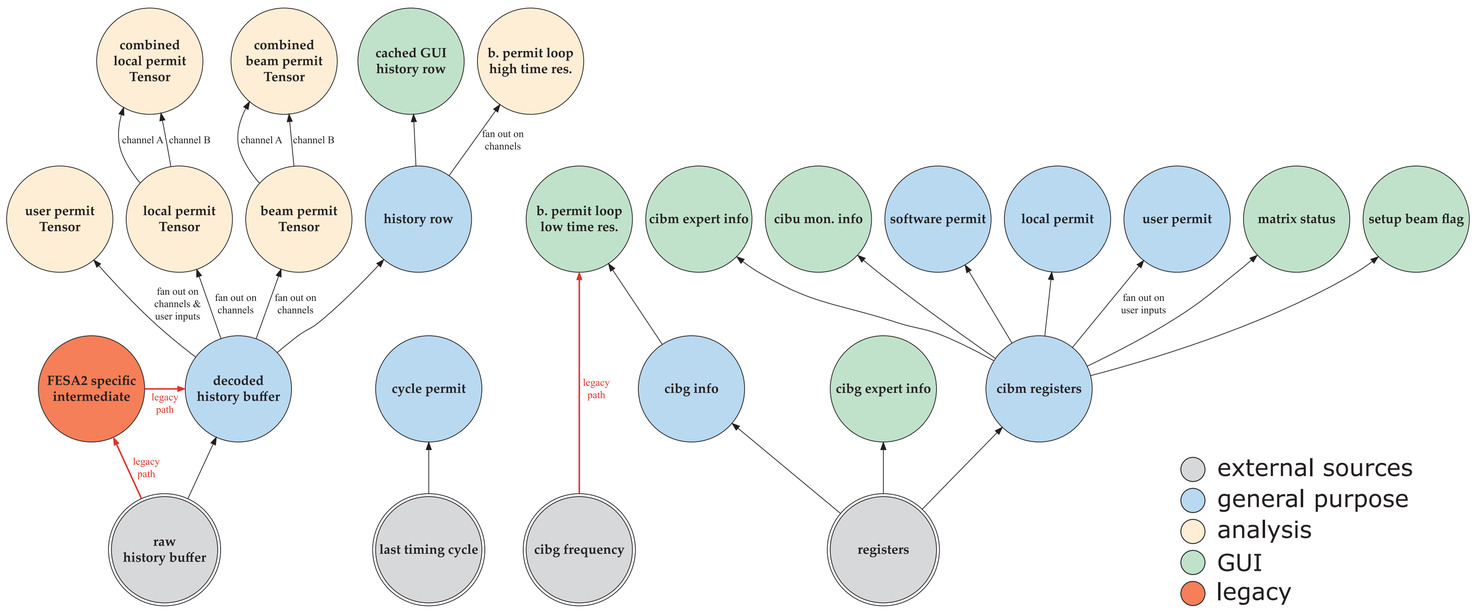

fig. 2 - the full tree of streams within the BIS software

A more functional approach to software development is required to properly work with streams. While there is a significant learning curve to adapt to this model, some benefits are gained from the start:

- Streams organize into a simple hierarchy, representative of the functionality of the system as a whole.
- Each stream is responsible for a single processing step, albeit of arbitrary complexity. This is better achieved with stateless processors, applying pure transformations to the data.
- Each stream acts as a clear separation between what is below it and what is above it; each layer does not need to concern itself with any other layer's business.

Re-usability

The Streamingpool framework [2, 3] is instrumental to having reusable streams. By fully decoupling the definition of a stream and its materialization, it allows any specific stream to be plugged into any arbitrary context, as long as the streams (usually just one) it directly depends on are provided somehow. Those can be mocks, simulated or alternative acquisition sources, or any implementation of the same contract.
In practice, as long as the data produced by a stream is not coupled to a context, this stream can be straightforwardly integrated into another application.

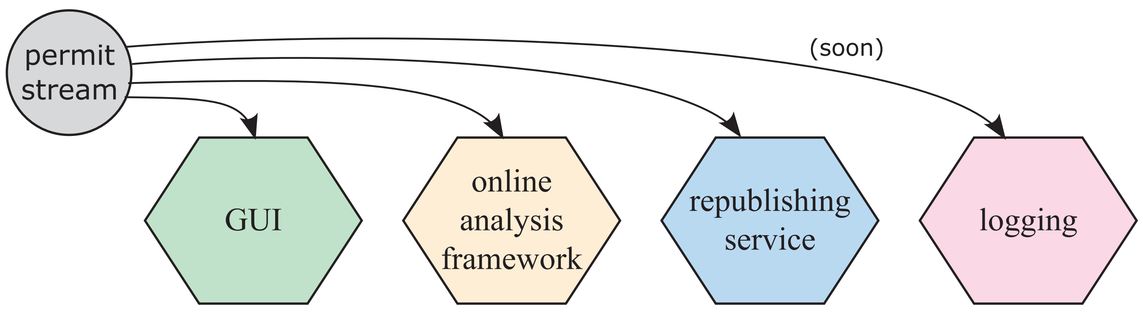

fig. 3 - BIS permit stream definition reuse across applications example

Dealing with legacy

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

  if (dev.isLegacy()) {    // apply a wart
    return new WartStreamId(dataStreamId);
  }                        // in a very well-contained manner

  return dataStreamId;
}
```

fig. 4 - wart containment procedure

Legitimate circumstances can also bend the architecture and code out of their original shape. While unavoidable, these warts should not leak throughout the code base, neither hinder further maintenance and evolution.
The streams hierarchy allows for proper containment of such flaws, both in the architecture and in the code. Besides, legacy paths can be very simply cut out when the circumstances change.

Headaches

Implementing a streams design uncovered some drawbacks. Most notably:

- RxJava [4] is the go-to streams library. Still, it has a steep learning curve and is not bug free.
- Streams do not prevent abstraction leaks. Moreover, parent streams behaviors impact the behavior of child streams, in sometimes complex ways.
- Full asynchronicity implies edge cases far too numerous to possibly cover with tests, when multiple streams are involved.

Prospects

Some topics need further attention to be evaluated and the streams ecosystem still grows:

- Fine tuning of the backpressure mechanisms. This is critical to performance optimization.
- Multiple efforts are bringing reactive streams over the network [5].
- Inter-operability with lower-level supervision software, whether in Java [6] or through native means [5].

References

[1] http://rxmarbles.com/

[2] A. Calia, K. Fuchsberger et al., "Streaming Pool - Managing Long-Living Reactive Streams for Java", ICALEPCS17, Barcelona, Spain (2017).

[3] https://github.com/streamingpool

[4] https://github.com/ReactiveX/RxJava

[5] https://reactivesocket.io

[6] C. Cardin, J.-C. Garnier et al., "Real-Time Java to Support the Device Property Model", ICALEPCS17, Barcelona, Spain (2017).