ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-THPHA176

STREAMING POOL - MANAGING LONG-LIVING REACTIVE STREAMS FOR JAVA

A. Calia, K. Fuchsberger, M. Gabriel, M.-A. Galilée, J.-C. Garnier, G.-H. Hemelsoet, M. Hostettler, M. Hruska, D. Jacquet, J. Makai, T. Martins Ribeiro, A. Stanisz, CERN, Geneva, Switzerland

Abstract

A common use case in accelerator control systems is subscribing to many properties and multiple devices and combine data from this. A new technology which got standardized during recent years in software industry are so-called reactive streams. Libraries implementing this standard provide a rich set of operators to manipulate, combine and subscribe to streams of data. However, the usual focus of such streaming libraries are applications in which those streams complete within a limited amount of time or collapse due to errors. On the other hand, in the case of a control systems we want to have those streams live for a very long time (ideally infinitely) and handle errors gracefully. In this paper we describe an approach which allows two reactive stream styles: ephemeral and long-living. This allows the developers to profit from both, the extensive features of reactive stream libraries and keeping the streams alive continuously. Further plans and ideas are also discussed.

INTRODUCTION

In practically any application within the operational environment of CERN accelerators, a common pattern is repeated:

- Subscribe to *N* properties of different devices,
- extract values and/or transform and/or combine them with values coming from other devices
- and buffer the incoming values to have a history of a certain length.

For the first part (subscription), a common mechanism exists through *JAPC* (Java API for Parameter Control). However, already this part is not ideally solved as it implies coupling down to the device layer of each application. For the two other steps, no common approach exists at all. The *Streaming Pool* project aims to close this gap by providing coherent means to implement processing- and abstraction layers. The code was kept general (not bound to any CERN specific library) and was open sourced under Apache License 2.0 [1].

Streaming Pool was designed with the following objectives in mind:

- Decoupling of Layers (e.g. When subscribing to the stream delivering the tune of a machine, the application does not have to know which device delivers this),
- rich set of operators for transformations,
- sensible defaults for error treatment
- and testability built in from the beginning.

Since this framework was never developed as a dedicated product, but always as a side product of operational applications under development, its feature set evolved according to the requirements from these applications. Therefore, some goals are only partly achieved in the current version (E.g. the decoupling of layers is possible, but in some applications not fully implemented). However, most of them (the last three in the above list) are fully available: Error treatment and testability are enabled by the internal design, while the rich set of operations are provided by the chosen technology (reactive streams), as described in the following sections.

REACTIVE STREAMS

A stream of data is a specialization of the Observer design pattern that is especially useful in contexts where the business logic of the application can be expressed as a series of transformations over a flow of data. In a stream there are typically three components:

- Publisher: is the source (or start) of the stream, it pushes the data through the stream.
- Processor: is an operation applied to a data item currently flowing through the stream.
- Subscriber: it consumes the data of the stream.

According to the items described above, a stream of data has a certain Publisher, zero or more Processors that act on the data (transforming them according to the business logic) and one or more Subscribers that consume the data.

Reactive streams are an initiative for creating truly asynchronous data streams that handle back pressure in a non-blocking way [2]. Being asynchronous, a reactive stream can switch context (thread of execution) at any time between Processors. In this scenario, boundaries between threads need special care. Avoiding the undesirable situation of unbounded buffers, reactive streams introduced the concept of back pressure. Data in a reactive stream flow on request, e.g. a Subscriber requests 10 items from the stream. In this way the Publisher does not flood the Subscriber with data that it is not able to process. A back pressure strategy is then needed in order to define the behavior when the aforementioned rule cannot be applied. For example, lets assume subscribing to a hardware device publication which delivers updates on a rate of 100 items per second. If the processing pipeline can only accept 90 per seconds there are typically three options (although more sophisticated techniques are possible):

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-THPHA176

- Drop latest: the item that comes from the device and cannot be processed by the pipeline is discarded.
- Keep latest: the latest item is kept and it will be processed first while the oldest are discarded. This strategy can be particularly useful when dealing with real-time data flows.
- Buffer: the new items that cannot be processed are stored in a buffer and are used according to the order of arrival. This strategy can be used when the data flow speed changes over time and the system can adjust its speed to the source (unbounded buffers are undesired).

LONG-LIVING STREAMS

The major reactive stream implementations for Java are Project Reactor [3] and RxJava [4]. Both implementations share the same concept regarding error handling: the subscription to the source collapses if an error occurs and the Subscriber has to re-subscribe (reconnect). This strategy is very useful e.g. in a web scenario: When an HTTP request generates an error, it is reported to the user and the client possibly reconnects and retries the request.

Streamingpool offers another approach to the problem by preventing an error from collapsing a stream. This strategy is useful in such scenarios where creating subscriptions is expensive, has side effects, or when the user wants to create a continuous analysis of real-time data while not being bothered with error-handling code and re-subscription to streams. For overcoming the problem of handling errors, each stream in the Streamingpool offers a parallel stream of `Throwable`, so that all the errors that would collapse the reactive stream are deflected to the errors stream. In this way, e.g. a timeout exception will not collapse the subscription and the user can handle the exception accordingly (reporting on a GUI for example). This approach opens the door to long-living reactive streams, where problems producing, processing or delivering one item do not hamper the ability of the stream to deliver future items.

ARCHITECTURE

The Streamingpool library has a simple yet powerful architecture. Its main goal is to provide an abstraction over the management of reactive streams in a software application.

It provides a simple API for discovering, providing and creating reactive streams. It focuses on sharing the streams and creating long-living data flows for online analysis or any business logic.

The key concept is the `StreamId`. A `StreamId` uniquely identifies a reactive stream in the Streamingpool in a type-safe way. The stream itself can be accessed (discovered) using the `DiscoveryService` interface. If a reactive stream is not present in the Streamingpool, it can be provided using the `ProvidingService` or lazily created using a `StreamFactory`, `TypedStreamFactory` or `StreamCreator`. Fig. 1 summarizes the afore-mentioned flow. Whenever the user discovers a `StreamId`, the Streamingpool checks if it already has the corresponding reactive stream in the pool of streams, reusing the existing ones. If this is not the case, the creation (materialization) is performed and a reactive stream is created from the information carried by the `StreamId`. The newly created stream is the saved into the pool so that it can be reused on subsequent calls.

In the next sections each component of the presented software architecture is explained in details.

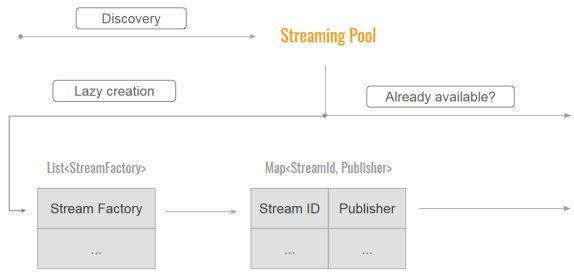

Figure 1: Streamingpool architecture flow.

StreamId

In order to identify a stream, Streamingpool uses the concept of StreamId.

An instance of a StreamId uniquely identifies a `Publisher <T>` so the developer must provide correct implementations of the `hashCode()` and `equals(Object o)` methods. In the case the before mentioned methods are not correctly implemented, the Streamingpool will not reuse streams and the behavior may be unpredictable (delays between reactive streams sharing subscriptions happen and this will cause mis-synchronization).

The fact that a StreamId identifies a specific stream, means it can carry information and it can be parametrized. For example, for accessing the hardware publication of a device, one could create a `DeviceStreamId<T>` and then parametrize it with the device identifier, Listing 1:

Listing 1: Usage of an hypothetical DeviceStreamId class.

```
DeviceStreamId.ofDevice("LHC.TUNE.BEAM1").
    subscribe(...);
```

DiscoveryService

The mechanism that allows a user to get streams from the Streamingpool is called `DiscoveryService`. When initializing the Streamingpool using Spring, a `DiscoveryService` bean becomes available for injection.

The API of the `DiscoveryService` consists of a single method: `discover(StreamId<T> id)` which returns a `Publisher<T>`.

Usually, in the Streamingpool streams are created lazily, at discovery time. Practically, it means that when the user discovers a `StreamId`, the Streamingpool triggers its creation if it is not present in the system. Therefore, a call to the method `DiscoveryService.discover(...)` is blocking.

Listing 2 shows an example of the discovery of a `StreamId` and a subscription to it using RxJava.

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-THPHA176

Listing 2: Discover and consume a StreamId.

```
DiscoveryService discoveryService = ...;
AnyStreamId streamId = new AnyStreamId();

Publisher<Any> stream =
  discoveryService.discover(streamId);

Flowable.fromPublisher(stream)
  .subscribe(System.out::println);
```

StreamFactory

A `StreamFactory` represents the mechanism for creating reactive streams (`Publisher<T>`) from a given `StreamId<T>`.

In the Streamingpool life-cycle, whenever the user tries to discover a `StreamId` for the first time it triggers the creation (materialization) of the corresponding `Publisher<T>`. Since the `StreamFactory` is the most generic way of creating reactive streams, its method signature is `create(StreamId<T> id, DiscoveryService discoveryService)` which must return an `ErrorStreamPair<T>`.

As shown in Listing 3, an implementation of a `StreamFactory` is not bound to a specific `StreamId` type, instead all registered factories are queried for every `StreamId` to be created. When a `StreamFactory` is not able to create the given `StreamId`, it can simply return `ErrorStreamPair.empty()`. In order to be able to create a hierarchy, along with the `StreamId` the `StreamFactory` receives a `DiscoveryService` that can be used to discover other dependent `StreamIds`.

Listing 3: StreamFactory example for creating an IntegerRangeId.

```
public class IntegerStreamFactory
    implements StreamFactory {
  @Override
  public <T> ErrorStreamPair<T> create(
      StreamId<T> id, DiscoveryService
      discoveryService) {
    if(!(id instanceof IntegerRangeId)) {
      return ErrorStreamPair.empty();
    }
    IntegerRangeId rangeId = (
        IntegerRangeId) id;
    int from = rangeId.getFrom();
    int to = rangeId.getTo();
    Flowable<Integer> rangeStream = range(
        from, to - from);
    return ErrorStreamPair.ofData((
        Publisher<T>) rangeStream);
  }
}
```

TypedStreamFactory

In Streamingpool, a `TypedStreamFactory` is a specialization of a `StreamFactory` that just creates one type of `StreamId`. For example, consider an use case that requires two different types of streams, identified e.g. by `DeviceTypeAStreamId` and `DeviceTypeBStreamId`. Either a single `StreamFactory` can be used for materializing either of them, or the responsibility can be split by using two different `TypedStreamFactory`, one that is responsible to create a `DeviceTypeAStreamId` and the other which creates `DeviceTypeBStreamId`.

StreamCreators

A `StreamCreator` is a special way (shortcut) of materializing a `StreamId` when the instance of the `StreamId` is known beforehand (e.g. a constant). Listing 4 shows how to create a `StreamCreator` that is used to materialize the `StreamId` `LIVE_DEVICE_ID` defined as constant. Notice that the API of a `StreamCreator` allows to specify it as a Java lambda and that it provides a `DiscoveryService` object for further discoveries.

Listing 4: StreamCreator example.

```
ImmutableIdentifiedStreamCreator.of(
  LIVE_DEVICE_ID,
  discoveryService ->
      DeviceApi.streamFrom("LIVE_DEVICE")
);
```

ProvidingService

Complementary to the `DiscoveryService`, the `ProvidingService` can be used to supply a `Publisher<T>` into the Streamingpool. The API is kept simple on purpose: `provide(StreamId<T> id, Publisher<T> stream)`. Practically, after providing a `Publisher` and the corresponding `StreamId`, the Streamingpool returns the same `Publisher` whenever the `StreamId` is discovered.

This method is mainly provided for unit tests. The preferred way of materializing a stream in the Streamingpool context is through the `StreamFactories` (or any of the variants described above).

ERROR HANDLING

As described before, Streamingpool allows holding long-lived reactive streams. Errors are handled gracefully (i.e. without collapsing any streams). This requires that the involved stream factories are implemented such that they deflect the errors onto a dedicated error stream:

Whenever a stream factory creates a `StreamId` it returns an `ErrorStreamPair` of a `Publisher<T>` for the data and the corresponding `Publisher<Throwable>` for any errors that may occur (see Listing 3).

Both the data and the error streams are then registered in the Streamingpool for future lookups. The error stream for any `StreamId` can be looked up by resolving the associated `ErrorStreamId` (Listing 5).

Listing 5: Usage of an ErrorStreamId for discovering error streams.

```
DiscoveryService discoveryService = ... ;
DeviceStreamId deviceId = DeviceStreamId.
    fromName("LHC.TUNE.BEAM1");
ErrorStreamId deviceErrorsId =
    ErrorStreamId.of(deviceId);

Publisher<DeviceData> dataStream =
    discoveryService.discover(deviceId);
```

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-THPHA176


```
Publisher<Throwable> errorStream =
    discoveryService.discover(
    deviceErrorsId);
```

It is also possible to subscribe for the error streams of all streams created by the pool. This is useful e.g. to create a dashboard showing all exceptions that have recently occurred and allows monitoring the health status of the application or system in question.

TESTING

Streamingpool is designed with unit testing in mind. The fact that the `DiscoveryService` does not materialize a stream if already present in the Streamingpool makes it easy to provide dedicated streams for testing. Through this mechanism, the logic under test can be isolated even in complex applications that use different layers of streams; a portion of a chain of processing can be isolated by providing the mocked input streams through the `ProvidingService`.

For example, consider the logic producing the stream for `StreamId` A should be tested. Further consider stream A depends on two input streams (`StreamIds` B and C). By providing mocked streams for B and C into the pool, when discovering A, the mocked streams will be used for B and C, allowing A to be tested independently.

Due to its effortless compartmentalization of components, the use of Streamingpool is a first step towards creating stable and robust software [5].

APPLICATIONS

The Streamingpool framework was developed mainly along a new application for LHC injection diagnostics [6]. In this application, the streaming pool is combined with the tensorics library [7, 8] to provide a reusable analysis framework [9]. While this is currently the most challenging use case, Streamingpool is also used for various simple control room applications, e.g. one that displays the remaining time for LHC injection kicker soft-start or the graphical user interfaces that control chromaticity [10] and coupling [11] of the LHC. Also in non-gui use cases, streaming pool started to be used. For example, CERNs TE-MPE-MS section provides (using Streamingpool) streams with the decoded beam permits that can be logged and consumed by other applications.

FUTURE DEVELOPMENTS

From the early days of Streamingpool it was clear that the logical next steps in the development would have to be transporting streams over the network. At that time, the reactive streams technology was still quite young, so it was decided to postpone the choice of technology for this and focus on the functionality described in the above sections. Meanwhile, the technology evolved and several options are available. For example, the Spring project included reactive controllers in their version 5.0. Using gRPC [12] as network layer is another option.

The second aspect, where future development will focus on, is including more diagnostics and debugging functionalities. Due to the standardized approach in Streamingpool generic Components (e.g. graphical user interfaces) can be built which e.g. can show the relations between the streams or the time structure of the related items. One example of such a generic GUI component which already exists, is a JavaFx panel that shows the exceptions of all error streams provided by a pool, which can be included in any application using Streamingpool as a backend (Fig. 2).

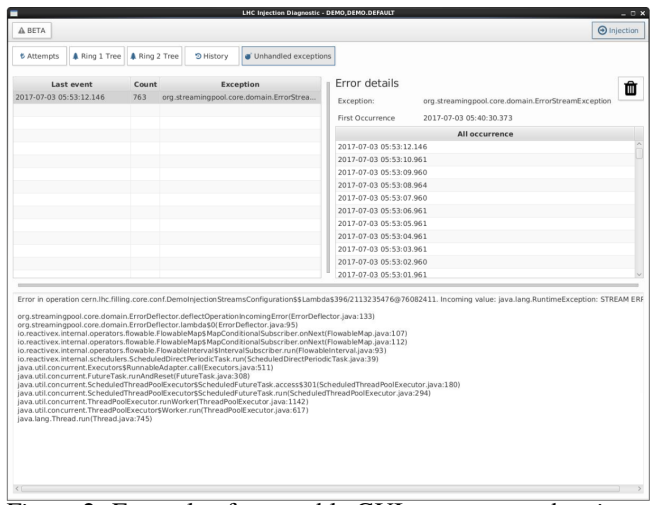

Figure 2: Example of a reusable GUI component, showing the deflected errors of an application.

REFERENCES

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

[2] http://www.reactive-streams.org/

[3] https://projectreactor.io/

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

[5] A. Calia, K. Fuchsberger, and M. Hostettler, "Testing the untestable: A realistic vision of fearless testing (almost) every single accelerator component without beam and continuous deployment thereof", in *Proc. IBIC'16*, Barcelona, Spain, Sep. 2016, pp. 399–402, DOI:10.18429/JACoW-IBIC2016-TUPG30

[6] A. Calia, K. Fuchsberger, G.H. Hemelsoet, and D. Jacquet, "Development of a new system for detailed LHC filling diagnostics and statistics", in *Proc. IPAC'17*, Copenhagen, Denmark, May 2017, pp. 1905–1907m doi:10.18429/JACoW-IPAC2017-TUPIK088

[7] K. Fuchsberger *et al.*, "Tensorics - a Java Library for Manipulating Multi-Dimensional Data With presented at ICALEPCS'17, Barcelona, Spain, Oct. 2017, paper THPHA177.

[8] https://github.com/tensorics

[9] K. Fuchsberger *et al.*, "A Framework for Online Analysis Based on Tensorics Expressions and Streaming Pool", presented at ICALEPCS'17, Barcelona, Spain, Oct. 2017, paper THPHA178.

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-THPHA176

[10] K. Fuchsberger and G. H. Hemelsoet, "LHC Online Chromaticity Measurement - Experience After One Year of Operation", in *Proc. IBIC'16*, Barcelona, Spain, Sep. 2016, pp. 20–23, doi:10.18429/JACoW-IBIC2016-MOBL04

[11] G.H. Hemelsoet *et al.*, "Online coupling measurement and correction throughout the LHC Cycle", presented at ICALEPCS'17, Barcelona, Spain, Oct. 2017, paper TUPHA119.

[12] https://grpc.io