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)

Key Concepts

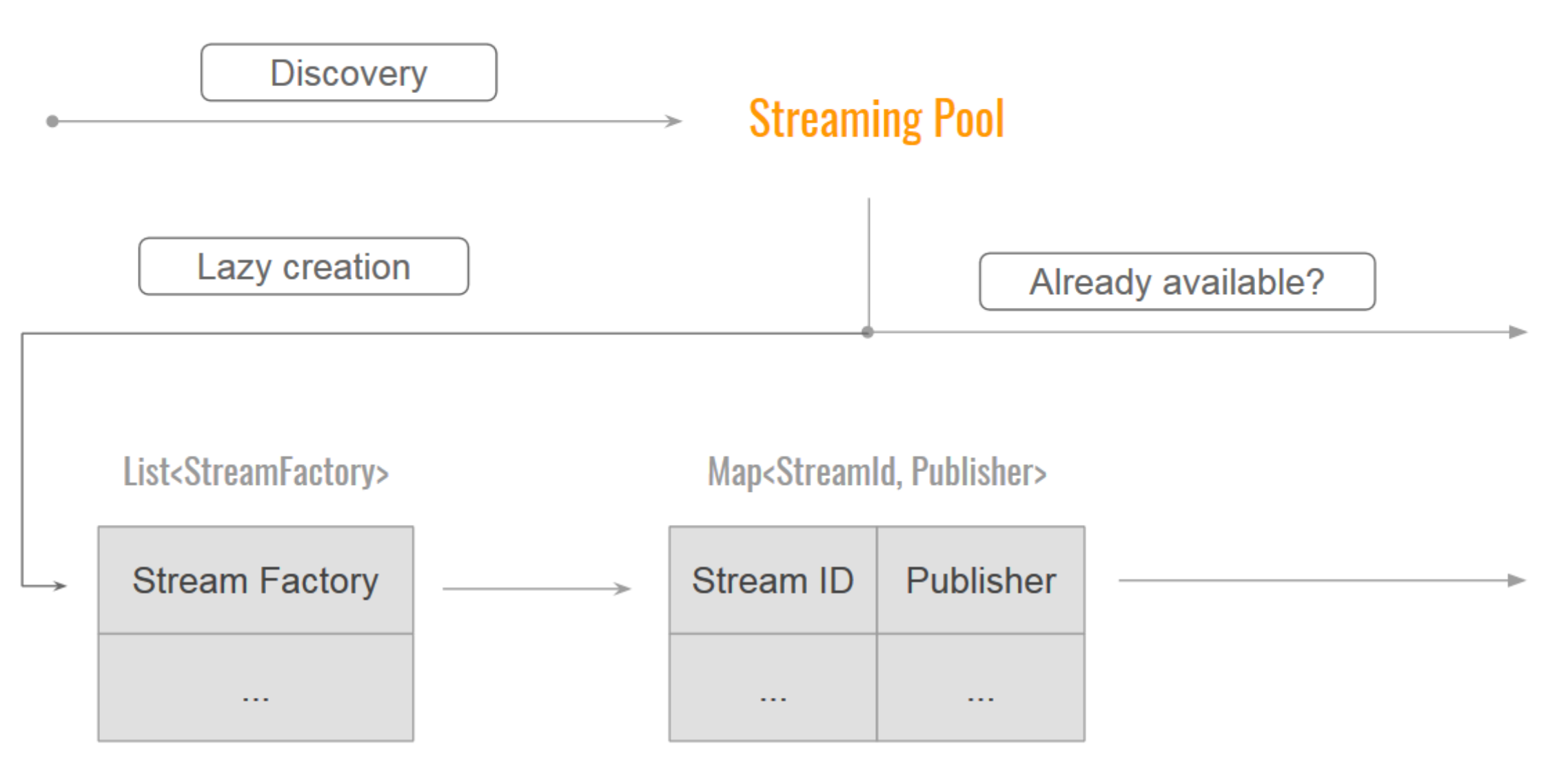
Typed Stream Discovery | Pool of Shared Streams | Lazy Stream Creation | Error Deflection

Architecture

The goal of the Streamingpool API is to provide an abstraction over the management of reactive streams in a software application. It provides mechanisms to discover, provide and create reactive streams. It focuses on sharing the streams and creating long-living data flows for online analysis or any business logic.
The key components in this mechanism are:

- `StreamId`: It uniquely identifies a reactive stream in the Streamingpool in a typesafe way.
- `DiscoveryService`: Through this interface a stream can be looked up ('discovered').
- `StreamingPool`: This is the central component, which manages the available streams. 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`.
- `StreamFactory`: Through this extension point, users of the library can describe how streams for certain stream ids will be created.

Architectural Overview:



Stream Ids

In order to identify a stream, Streamingpool uses the concept of StreamId.
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 3:

Listing 3: Usage of an hypothetical DeviceStreamId class.

```
DeviceStreamId streamId =
  DeviceStreamId.ofDevice("LHC.TUNE.BEAM1");
```

Stream Discovery

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.

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);
```

Error Handling

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.
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 1).

Listing 1: 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);
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.

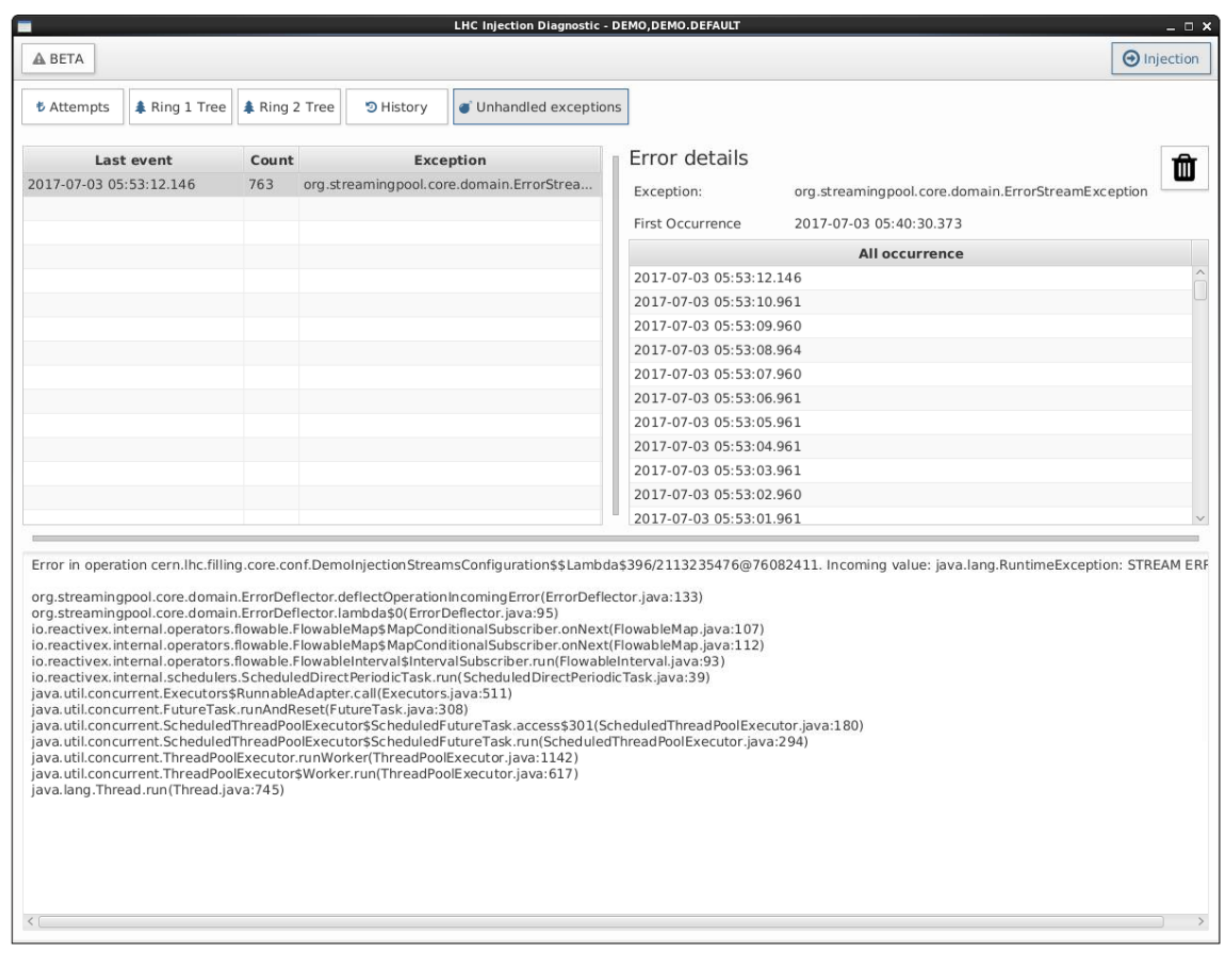
Applications

- Mainly developed along a new LHC injection diagnostics [6] application. Here it is used together with the tensorics library [7, 8] to provide a reusable analysis framework [9].
- Control room applications: e.g. 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.

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`.

Reusable GUI Components



Last event	Count	Exception
2017-07-03 05:53:12.146	763	org.streamingpool.core.domain.ErrorStrea...

Error details

Exception:	org.streamingpool.core.domain.ErrorStreamException
First Occurrence	2017-07-03 05:40:30.373

All occurrence
2017-07-03 05:53:12.146
2017-07-03 05:53:10.961
2017-07-03 05:53:09.960
2017-07-03 05:53:08.964
2017-07-03 05:53:07.960
2017-07-03 05:53:06.961
2017-07-03 05:53:05.961
2017-07-03 05:53:04.961
2017-07-03 05:53:03.961
2017-07-03 05:53:02.960
2017-07-03 05:53:01.961

```
Error in operation cern.lhc.filling.core.conf.DemoInjectionStreamsConfiguration$$Lambda$396/2113235476@76082411. Incoming value: java.lang.RuntimeException: STREAM ERF

org.streamingpool.core.domain.ErrorDeflector.deflectOperationIncomingError(ErrorDeflector.java:133)
org.streamingpool.core.domain.ErrorDeflector.lambda$0(ErrorDeflector.java:95)
io.reactivex.internal.operators.flowable.FlowableMap$MapConditionalSubscriber.onNext(FlowableMap.java:107)
io.reactivex.internal.operators.flowable.FlowableMap$MapConditionalSubscriber.onNext(FlowableMap.java:112)
io.reactivex.internal.operators.flowable.FlowableInterval$IntervalSubscriber.run(FlowableInterval.java:93)
io.reactivex.internal.schedulers.ScheduledDirectPeriodicTask.run(ScheduledDirectPeriodicTask.java:39)
java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:511)
java.util.concurrent.FutureTask.runAndReset(FutureTask.java:308)
java.util.concurrent.ScheduledThreadPoolExecutor$ScheduledFutureTask.access$301(ScheduledThreadPoolExecutor.java:180)
java.util.concurrent.ScheduledThreadPoolExecutor$ScheduledFutureTask.run(ScheduledThreadPoolExecutor.java:294)
java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1142)
java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:617)
java.lang.Thread.run(Thread.java:745)
```

Future Developments

It is clear that the current state of Streamingpool is only a first step and further effort has to go into several directions:

- **Network streams**: From the early days this step was foreseen. 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.
- **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.

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, M. Hostettler, "Testing the untestable: A realistic vision of fearless testing (almost) every single accelerator component without beam and continuous deployment thereof", IBIC16, Barcelona, Spain (2016).
[6] K. Fuchsberger et al., "Development of a new system for detailed LHC filling diagnostics and statistics", IPAC17, Copenhagen, Denmark (2017).
[7] K. Fuchsberger et al., "Tensorics - a Java Library for Manipulating Multi-Dimensional Data With Pleasure", ICALEPCS17, Barcelona, Spain (2017).
[8] `https://github.com/tensorics`
[9] K. Fuchsberger et al., "A Framework for Online Analysis Based on Tensorics Expressions and Streaming Pool", ICALEPCS17, Barcelona, Spain (2017).
[10] K. Fuchsberger, G. H. Hemelsoet, "LHC Online Chromaticity Measurement - Experience After One Year of Operation", IBIC2016, Barcelona, Spain (2016).
[11] G.H. Hemelsoet et al., "Online coupling measurement and correction throughout the LHC Cycle", ICALEPCS17, Barcelona, Spain (2017).
[12] `https://grpc.io`