Extending the FairRoot framework to allow for simulation and reconstruction of free streaming data

M. Al-Turany, D. Klein, A. Manafov, A. Rybalchenko and F. Uhlig
Planckstrasse1, 64291 Darmstadt, Germany
E-mail: m.al-turany@gsi.de

Abstract. The FairRoot framework is the standard framework for simulation, reconstruction and data analysis for the FAIR experiments. The framework is designed to optimise the accessibility for beginners and developers, to be flexible and to cope with future developments. FairRoot enhances the synergy between the different physics experiments. As a first step toward simulation of free streaming data, the time based simulation was introduced to the framework. The next step is the event source simulation. This is achieved via a client server system. After digitization the so called "samplers" can be started, where sampler can read the data of the corresponding detector from the simulation files and make it available for the reconstruction clients. The system makes it possible to develop and validate the online reconstruction algorithms. In this work, the design and implementation of the new architecture and the communication layer will be described.

1. Introduction
Using FairRoot one can create simulated data and/or perform analysis within the same framework. The framework delivers base classes which enable the users to construct their detectors and/or analysis tasks in a simple way [1, 2]. The user analysis is organized in tasks (FairTask). Tasks are executed sequentially in the order they are added to the run manager (Figure 1).

![Figure 1. Task hierarchy runs sequentially in one process](image)

One way to improve the performance of FairRoot on multi-core processors is to run the
independent tasks in parallel. Which can be done by running each task in a separate thread or as separate process (Figure 2).

However, by multi-threading an error in one thread can bring down all the threads in the process where in multi-process the different processes are insulated from each other by the OS or even the network, i.e: An error in one process cannot bring down directly another one. On the other hand, the inter-thread communication is faster than interprocess one. Trying to find the correct balance between reliability and performance we decided to use the multi-process concept with message queues for data exchange, i.e: Each "Task" is a separate process, which can be also multithreaded, and the data exchange between the different tasks is done via messages. This concept allow us to create different topologies of tasks that can be adapted to the problem itself, and the hardware capabilities. Some of the advantages of such a system are:

- Flexibility: Different data paths can be modeled by simply changing the topology of Tasks.
- Scalability: Spread the work over several processes and machines on the fly.
- Adaptive: Sub-systems are continuously under development and improvement and can be exchanged individually.

2. ZeroMQ: Base for data transport layer in FairRoot
Extending FairRoot to support multi-processes needs a solid and stable communication layer that should handle the whole communication in a transparent and stable way. Fortunately, such a layer exists already and is called ZeroMQ [3]. ZeroMQ is a very lightweight messaging system, specially designed for high throughput and low latency scenarios. It is open source, embeddable socket library that redefines the term socket as a general transport endpoint for atomic messages. ZeroMQ sockets provide efficient transport options for inter-thread, inter-process and inter-node communication (see Figure 3). Moreover it provides a Pragmatic General Multicast pattern (PGM), which is a reliable multicast protocol [3].

The zero in ZeroMQ refers to a culture of minimalism that permeates the project. They want to add power by removing complexity rather than exposing new functionality [4]. The library provides a built-in routing strategies for one-to-many or many-to-one communication scenarios. Each socket potentially comes with a sending and a receiving queue of configurable sizes.

2.1. Messaging Patterns
ZeroMQ has four built-in patterns for messaging:
Figure 3. Message transport options in ZeroMQ. **Named Pipe**: Piece of random access memory (RAM) managed by the operating system and exposed to programs through a file descriptor and a named mount point in the file system. It behaves as a first in first out (FIFO) buffer.

- Request-reply, which connects a set of clients to a set of services (remote procedure call and task distribution).
- Pub-sub, which connects a set of publishers to a set of subscribers (data distribution).
- Pipeline, which connects nodes in a fan-out/fan-in pattern that can have multiple steps and loops. (parallel task distribution and collection).
- Exclusive pair, which connects two sockets exclusively. (connecting two threads in a process).

3. **Data transport layer**

The new data transport layer in FairRoot is called FairMQ - the package provides a number of components which can be connected to each other in order to construct a processing topology. They all share a common base which is called FairDevice. All devices are grouped by three categories:

- **Source**: FairDevices without inputs are categorized as sources. Currently there is only one type of source device implemented: Sampler. A sampler is used to feed the pipeline (Task topology) with data from files.
- **Message-based Processor**: Message based processors are devices that operate on messages without interpreting their content. Five message-based processors have been implemented:
  - Sink: Save Messages (e.g: files, /dev/null)
  - Splitter: One input to many outputs, split via round-robin algorithm
  - Merger: Merge two or more input messages to one output, merge via fair-queued algorithm
  - Proxy: Has N-inputs and M-outputs, devices can connect to it on the fly (it binds on both sides)
  - Buffer: Pass messages through
- **Content-based Processor**: This is the place where the message content is accessed and the user algorithms process the data.

3.1. **Payload Protocol**

The framework does not dictate any application protocols. Potentially any content-based processor or any source can change the application protocol. Therefore, only a generic message
class is provided (FairMessage). It works with any arbitrary and continuous chunk of memory. One has to pass a pointer to the memory buffer, the size in bytes, and optionally a function pointer to the destructor, which will be called once the message object is discarded.

4. Integration to the existing software

Before moving the whole FairRoot framework and the user code to this new concept we have to proof that we can deliver the same results as before and try to minimize the changes of the user code as far as possible. To do that we introduce some wrapper classes (FairMQProcessorTask). This wrapper is a ZeroMQ class which encapsulates the FairTask (Where the user code is located) and run it in a ZeroMQ environment. We also modify our CMake building scripts in order to create a shared library and an executable for each Task. With this we are able to run the same algorithms in ROOT\[5\] event loop or in a distributed environment (see Figure 4)

![Figure 4. Integrating the existing software](image)

5. Results

To benchmark and test the implemented classes we took the third example of the FairRoot distribution \[6\](Figure 5).

![Figure 5. Example 3 geometry: 4 tracking stations with a dipole field](image)

We simulated 20 thousand events of 300 protons per event and run the digitization in the traditional way. However the reconstruction is ported to the new scheme and we compared following scenarios:
5.1. Traditional ROOT, single process, single thread
This case is shown in Figure 6. The data from files is stored in "TClonesArrays" containers of ROOT and after processing them the results are again send to TClonesArrays that are written to ROOT files. The wall time for this was 162 seconds and the memory usages is 241 Mbyte. This case is actually our reference case and the following results will be compared to these numbers.

Figure 6. From digits to hits with ROOT

5.2. ZeroMQ based reconstruction
This case is shown in Figure 7. We used the "FairMQProcessorTask" described above to run the same task in a distributed environment. However, even though we make the IO asynchronous with the processing we have an overhead of copying the payloads from TClonesArray and back, this overhead is in fact the price we pay for being compatible with the two scenarios.

Figure 7. From digits to hits with ZeroMQ

Figure 8. Results: From digits to hits with ZeroMQ using push-pull pattern
The results of this case are summarized in Figures 8 and 9. From the upper part of Figure 8 one can see that most of the time is spend in processing (142 s), so if we distribute the processing to two (lower part of the same Figure) we are now limited by the speed of the sampler which reads the data from disk. In this case we are also using the push-pull pattern, i.e: when a queue gets full the threads (process) are simply waiting until it becomes free again, thus no data is lost. In the Figure 9 we used the pub-sub pattern. In this mode when the queue gets full (subscribers are slow) messages are dropped\(^1\). Here one can see that if the processor is not fast enough we simply lose data (32 %).

The last case we investigated here is what do we gain if we remove the copy overhead which is shown in Figure 7. i.e: Remove the TClonesArray and implement the algorithms directly in the FairMQProcessorTask without using the ROOT containers (see Figure 10).

As can be seen from Figure 11 we could gain a factor 2 in performance with a minimum overhead of memory usage (4%) by using the concurrency concept introduced to FairRoot. This example was simply a proof of concept and the results we get encourage us to go further in this direction. We plan to use Multiprocessing for the task topology and multithreading inside the tasks. Using the pub-sub pattern together with the time based simulation in FairRoot we are able to simulate free data streams (as they will be in the experiment) so we can offer a realistic environment for developing the reconstruction code needed to handle this scenario, which will be present in the future Fair experiments.

\(^1\) The queue size is defined by setting a high water mark (HWM) for the sockets, when a socket reaches its HWM, it will either block or drop data depending on the socket type.
6. Outlook and future plans
We need to design and develop a dynamic deployment system (DDS) that can utilize any resource management system (e.g: Slurm, GridEngine, ...etc.). A monitoring and logging system is also under development. Moreover, on the long term we will drop the backward compatibility to the single process system.

References
[1] M. Al-Turany et al. The FairRoot framework J. Phys.: Conf. Ser. 396 022001, 2012.
[2] M. Al-Turany FairRoot: http://fairroot.gsi.de.
[3] http://www.zeromq.org/
[4] P. Hintjens, http://zguide.zeromq.org/page:all.
[5] R. Brun and F. Rademakers. Root - an object oriented data analysis framework Nuclear Instruments and Methods in Physics Research A, 389, 81-86, 1997
[6] https://subversion.gsi.de/trac/fairroot/browser/fairbase/tags/v-13.10/example/Tutorial3
[7] M. Al-Turany and T. Stockmanns, FairRoot: Time-based simulation and reconstruction, PHN-IS-IT-11, GSI SCIENTIFIC REPORT 2011