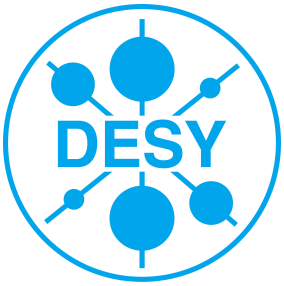

Software Development for a common DAQ at test beams

Darya Shirokova, Novosibirsk State University, Russia

Supervisor: Jan Dreyling-Eschweiler, Deutsches Elektronen-Synchrotron

September 2, 2016

Abstract

EUDAQ is a modular data acquisition framework written in C++. It implements a finite state machine so that the software can only be in one of a finite number of states. Previously the initial state of the machine was the unconfigured state. However in some cases it is feasible to separate hardware configuration from an initial hardware setup which is more time demanding. The first part of this work describes the extension of the EUDAQ FSM. The second part is concentrated on the integration of a new type of producers — a slow producer. It allows one to use devices that are not synchronized with the others and produce data at their own rate.

Contents

1 Introduction

EUDAQ is a modular, operating system independent data acquisition framework written in C++ and focused on an easy and flexible integration of the user's detector DAQs (EUDAQ Producers). It was designed for the EUDET Pixel Telescopes (see [1]), but it can be also used in other systems [3]. It provides logging, data storing, online monitoring functions and also GUI and console interfaces for interaction between a user and the software.

1.1 EUDAQ architecture

The EUDAQ software consists of different components communicating via TCP/IP protocol that allows different components to run on different machines.
The EUDAQ architecture is shown in figure 1. The central part of the software is the `RunControl` and all the other components (`LogCollector`, `DataCollector`, `OnlineMonitor` and `Producers`) are connected to it. It has the `RunControlGUI` interface through which a user can interact with the software. The `RunControl` sends commands to the components which extend from the `CommandReceiver` class (see 2.3). It allows one to perform configuration, start a run, etc. Besides, it distributes IP addresses and ports for the connection establishment between other components (e.g. `LogCollector` and `Producer`, `DataCollector` and `Producer`).

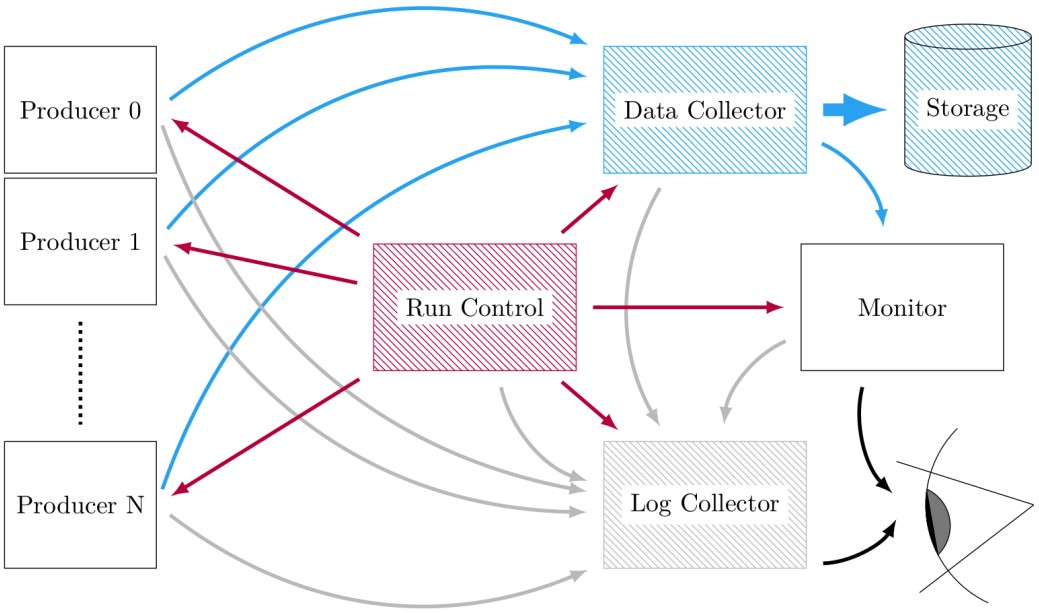

Figure 1: EUDAQ architecture [2].

The `LogCollector` collects debugging, error or information messages from other components and shows them to a user.

The DataCollector combines data streams from all producers into a one stream and writes the data to a binary file. The working principle of the DataCollector is described in more details in the section 3.1.
The OnlineMonotor is used to generate plots for online-monitoring (hitmaps, correlations plots, etc). ROOT is used for generating histograms.
The Producer is a device that produces any data, e.g. telescope sensors, the TLU, the DUT (device under test), etc. Every user device must extend from the Producer class or the SlowProducer class, which was newly developed within this work (see 3.3). User can send commands to a producer (OnInit, OnConfigure, OnStartRun, etc) using the RunControl interface. Data from different producers are collected by the DataCollector.

2 EUDAQ FSM

2.1 Description

A finite state machine (FSM) is a computational model when a machine can only be in one of a finite number of states. The two basic principles are that it can only be in a one state at a certain time and changing the current state is implemented via predefined transitions.
Since all the states and transitions between them are well-defined, the using of the FSM prevents the occurrence of unpredictable situations such as deadlocks, running the system before configuration, etc.
Each component connected to the RunControl can always be characterized by the current state. The previous version of the EUDAQ FSM [4] implemented the following connection states:

- UNCONFIGURED: the initial state of every connection. Configuration parameters have not been set yet. Available transitions are OnConfigure and OnTerminate.
- CONFIGURED: configuration parameters have already been set. Available transitions are OnStartRun, OnConfigure and OnTerminate.
- RUNNING: the state of the operating connections. For producers it means producing data, the OnlineMonitor are providing plots, the LogCollector is continuing to collect log messages and the DataCollector combines data streams from producers. The only available transition is OnStopRun.
- ERROR: this state can be used by users in case of errors during configuration, running process, etc. The only available transition is OnTerminate.

The state of the machine is determined by the lowest state of the connected components (LogCollector, DataCollector, OnlineMonotor, Producers) in the following priority: ERROR, UNCONFIGURED, CONFIGURED, RUNNING. It means, for example, that even if only one connection is in the ERROR state, the whole machine will also be in

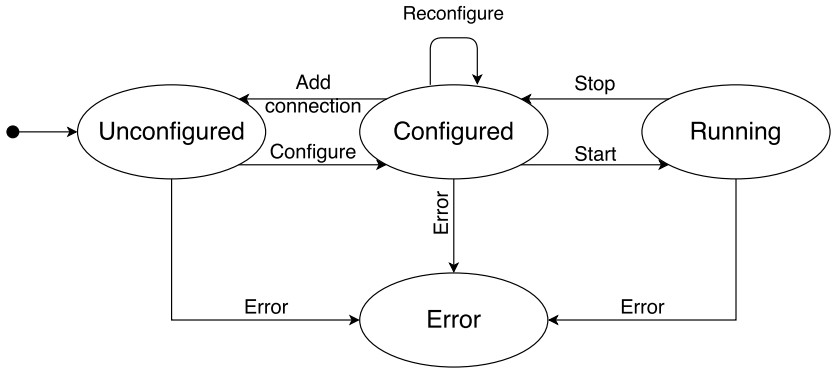

Figure 2: The previous version of the EUDAQ FSM.

that state. This prevents such mistakes as running the system before every component has finished the configuration. The states of the previous version of the EUDAQ FSM and transitions between them are shown in figure 2.
One should take into account that ERROR state of the FSM is not properly defined. There are no well-determined rules how the machine can go into this state and what should be done to fix the error since it highly depends on the reason of the error. Because of that the only action that is allowed in the ERROR state is the termination of the program.

2.2 FSM modifications

Some detectors require not only setting up different parameters, but also the first initialisation of the hardware. Previously both these steps were carried out during the configuration. However, setting up the hardware is only necessary during the start and can be time demanding, so reinitialisation of the hardware can take additional time when reconfiguration is needed.
For these reasons it turns out to be feasible to separate initialisation and configuration processes. In order to do that one has to introduce a new UNINITIALISED state and an initialisation transition. The UNINITIALISE state is defined as an initial state of every connection and the transit to the UNCONFIGURED state can be done by initialisation. In comparison with the previous implementation of the FSM described in section 2.1 it introduces the following changes:

- UNINITIALISED: the initial state of every connection. Initialisation has not not been conducted yet. Available transitions are `OnInitialise` and `OnTerminate`.
- UNCONFIGURED: initialisation has already been conducted, but configuration parameters have not been set yet. Available transitions are `OnConfigure` and `OnTerminate`.

The new version of the FSM is schematically shown in figure 3.

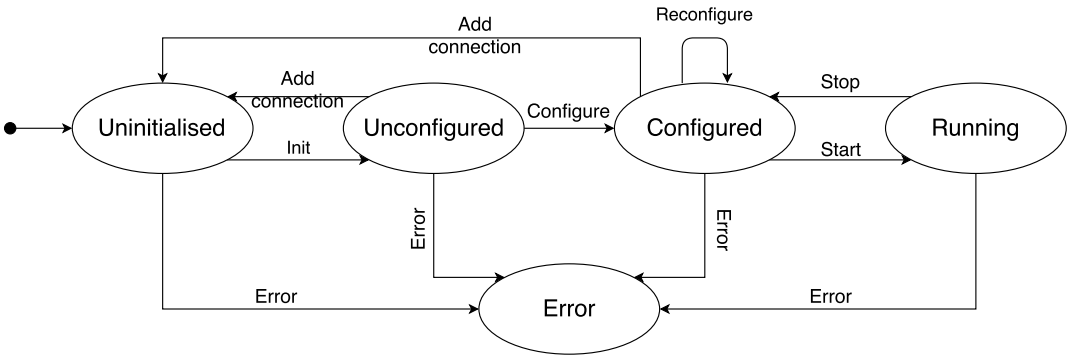

Figure 3: The new version of the EUDAQ FSM.

2.3 Implementation

The more detailed description of the FSM implementation can be found in [4]. The state of the connections are defined in the `ConnectionState` class that extends from the `Serializable` interface to convey the information via TCP/IP. States of the FSM are listed in the enumerator:

```
enum State {
      STATE_UNINIT,
      STATE_UNCONF,
      STATE_CONF,
      STATE_RUNNING,
      STATE_ERROR
    };
```

The `MachineState` class is responsible for keeping track of the state of the whole system. It has a map of all connections and their states and using this information it defines the current state of the whole machine:

```
std::map<ConnectionInfo, ConnectionState> connection_status_info;
```

In order to change the state, the `RunControl` sends a command to the `CommandReceiver`, which is the base class for those who needs to receive commands. The common set of the `CommandReceiver` functions (transitions) that can be used to change the current state of the connection is the following:

```
    virtual void OnInitialise(const Configuration &param) {
       SetConnectionState(eudaq::ConnectionState::STATE_UNCONF);}
    virtual void OnConfigure(const Configuration &param);
    virtual void OnStartRun(unsigned /*runnumber*/) {
       SetConnectionState(eudaq::ConnectionState::STATE_CONF);}
    virtual void OnStopRun() {}
```

One should note that `OnInitialise` function that is used to set the connection into the CONFIGURED state has default implementation in the base `CommandReceiver` class, so users who do not need an initialisation step can simply skip it. In addition, this ensures backward compatibility for existing producers.

The interaction between a user and the machine is implemented via `RunControlGUI`. An example of the interface is shown in figure 4. In the bottom of the window connected components and their current state are displayed. A current state of the whole machine is depicted on the top. A user can initiate an allowed transition by pressing one of the buttons of the interface: `Init`, `Config`, `Start`, `Stop`, `Terminate`.

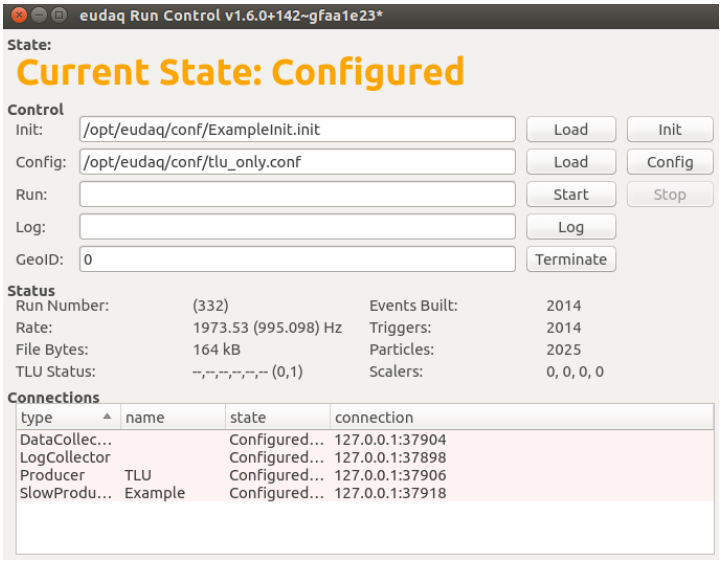

Figure 4: RunConrolGUI.

One of the issues that should be taken into account is that currently all the restrictions on the FSM are imposed by GUI via disabling transition buttons. It means that in principle one can set any new state independently of the previous state, the machine does not check if it is allowed or not.
The new version of the FSM was successfully tested and merged with the latest version of the software (v1.7) [5].

2.4 Use of the FSM

In order to add a new device to the system one has to extend either from the `Producer` class or from the `SlowProducer` class (see 3.3) and implement the transition functions. A good simple example can be found in `ExampleProducer.cxx` file. To change the state of the connection the following function should be used:

```
SetConnectionState(eudaq::ConnectionState::NEW_STATE, "Message");
```

whereas NEW_STATE is a connection state from the enumerator, see above.

3 Slow producer

3.1 DataCollector working principle

Every producer has to be connected to the `DataCollector` in order to send data. The `DataCollector` combines data streams from all producers into one stream and writes the data to a binary file. Before writing data to a file the `DataCollector` waits for events from all connected producers.
If there is any busy hardware that can not send data at the moment, the trigger is not sent. Thus, the event rate is limited to the slowest device.
The working principle of the `DataCollector` is schematically shown in figure 5. At the beginning of the run every producer has to send the Begin of Run Event (BORE). The `DataCollector` stores an event in the `OnCompleteEvent` function only when it has received an event from all connected producers. As the event is completed a new cycle of data receiving is started. At the end of the run the End of Run Event (EORE) must be sent.

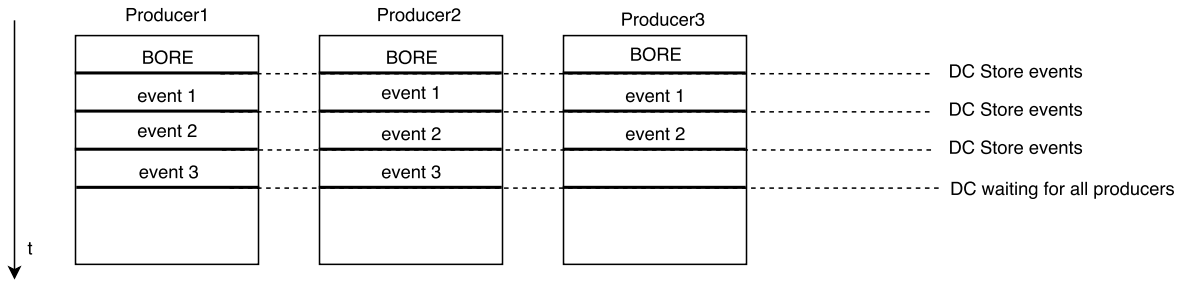

Figure 5: DataCollector working principle.

This way of data collecting ensures the data synchronization, but it turned out that in some cases it is not flexible enough.

3.2 Slow control devices

Sometimes it could be helpful to use so called slow control devices that, for example, perform the monitoring of temperature, pressure, etc. But this kind of devices has a lower event rate in comparison with the telescope rate about 2 kHz. For example, monitoring data could be provided one time per second.
However, due to the fact that trigger is only sent if every hardware is not busy, there is an obvious problem with integration of the slow control devices: it can slow down the whole system.
One of the possible solutions is proposed in the following section.

3.3 Slow producer

A slow producer is a special type of producers which interacts with the `DataCollector` in a different way using the hardware conception of the triggerless data taking. Instead of sending busy signals it can simply provide data at its own rate. The `DataCollector` distinguishes the slow producer from the usual one, it waits only for events from simple producers and ignores the absence of those from slow producers. Schematically this new working principle is shown in figure 6.

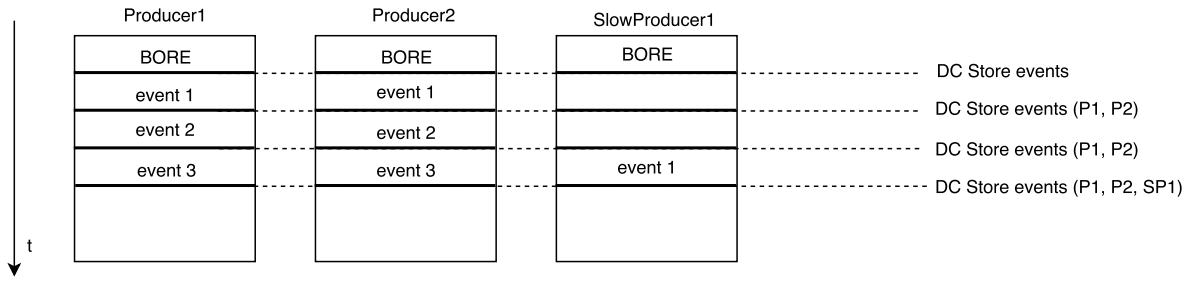

Figure 6: DataCollector interaction with slow producers.

Programmatically the concept of the slow producer is implemented as a new `SlowProducer` class. So in order to integrate a slow device one has to simply extend it from the `SlowProducer` class instead of the `Producer` class, but having then the same functionality.
The `DataCollector` now stores a map where the key is the producer number in the buffer array and the second argument is a producer type (either slow or not):

```
std::map<size_t, std::string> m_ireceived;
```

At any time the `DataCollector` knows how many slow producers are connected to it:

```
void DataCollector::OnConnect(const ConnectionInfo &id) {
  ...
  if (id.GetType() == "SlowProducer") {
    m_slow++;
  }
}

void DataCollector::OnDisconnect(const ConnectionInfo &id) {
  ...
  if (id.GetType() == "SlowProducer") {
    m_slow--;
  }
}
```

In the `OnReceive` function the `DataCollector` only checks if all not slow producers have sent an event and then goes to the `OnComplete` function

```
void DataCollector::OnReceive(const ConnectionInfo &id,
                              std::shared_ptr<Event> ev) {
```

```
    ...
    bool tmp = false;
    // Add the producer that has sent data.
    m_ireceived[GetInfo(id)] = id.GetType();
    // Compute the number of "fast producers" that have sent data.
    int fastwaiting = 0;
    for (std::map<size_t, std::string>::iterator it =
         m_ireceived.begin(); it != m_ireceived.end(); ++it) {
      if (it->second != "SlowProducer")
        fastwaiting++;
    }
    // If all fast producers are ready, complete the event.
    if (fastwaiting == m_buffer.size() - m_slow)
      tmp = true;
    if (tmp)
      OnCompleteEvent();
  }
```

Since all the producers that have sent an event including the slow one are stored in the map, in the `OnComplete` function the `DataCollector` reads out and stores only non empty data buffers, or in other words it ignores slow producers that have not sent any data.
This approach has been tested with the TLU and the `ExampleSlowProducer` extended from the `SlowProducer` class. The `ExampleSlowProducer` sends one event per second while the frequency of the TLU is 1000 Hz. The rate of the system has not been limited by the slow producer and all events from both producers have been written in the raw data file (the events from the `ExampleSlowProducer` occur with a period of one thousand TLU events).

3.4 Integration of user DAQs

A brief summary of two possible types of user devices and difference between them is presented in table 1.

	TLU common	No TLU common
Data	event based data	monitoring data
Hardware	receives trigger, sends busy	triggerless device
EUDAQ	Producer	Slow Control

Table 1: User DAQ types.

4 Summary

This work has introduced the following contributions:

- The modification of the EUDAQ FSM: a new state was integrated to the existing system.

- Testing and merging the EUDAQ FSM with latest version of the software (v1.7).
- Development of the new approach of data collecting which considers the existence of the slow control devices.

References

[1] https://telescopes.desy.de/Main_Page

[2] Test Beam Measurements for the Upgrade of the CMS Pixel Detector and Measurement of the Top Quark Mass from Differential Cross Sections *Simon Spannagel*

[3] EUDAQ Software User Manual *EUDAQ Development Team*

[4] Establishment of a Finite State Machine for the EUDAQ framework *Beryl Bell*

[5] https://github.com/eudaq/eudaq/tree/v1.7-dev