

Implementation of a Finite State Machine for the EUDAQ framework

Beryl Bell, Hampshire College, United States

September 8, 2015

Abstract

EUDAQ is a C++ based generic data acquisition framework(DAQ), usable across various platforms. Runtime errors have been occurring during the use of the software. These issues are rooted in a failure of the EUDAQ software to adhere to a finite state machine. Thus the software can exist in states that are undefined by the assumed finite state machine. For instance the program could be run before all of its producers were configured. The object that tracks the state of the producers in the original EUDAQ framework was located, determined to be insufficient. An adjustment of this state tracking is made by creating two new classes that were added and implemented for the finite state machine branch of EUDAQ.

Contents

1	Introduction	3
2	EUDAQ Framework	4
2.1	Class Hierarchies	4
2.2	Transmission Control Protocol	7
2.3	Status Class	8
2.4	Finite State Machine For EUDAQ	10
2.5	Status Class as State Tracker	11
3	MachineState and ConnectionState Class	12
3.1	Implementation ConnectionState Class	12
3.2	Implementation of MachineState	13
3.3	Modification of GUI	15
4	Backward Compatibility	15
5	Moving Forward	16
5.1	Transition States	16
5.2	Error Handling	17
6	Summary	18
	Appendices	19

1 Introduction

EUDAQ is a data acquisition framework. Conceivably this framework could be used in any number of data acquisition projects. The framework is written for and tested on the EUDET Pixel Telescope, but can be used for many different devices. EUDAQ is written in C++ using the Qt libraries for its graphic interface and is structured in a such a way that the framework can be used on Windows, Mac OS X, and Linux.

The framework is built in such a way that the writing of additional software for new devices that one wishes to add to their system is straightforward. One simply needs to extend the preexisting producer class and override all required methods. EUDAQ takes care of the remaining setup.

EUDAQ goes through several steps to begin its data acquisition. Ideally these steps are in the order: Connection of Devices, Configuration of Connected Devices, Run Data Acquisition, Stop, Adjust Configuration, repeat. However, the framework should still function even when the steps are not always precisely in this order. Thorough use of the framework has revealed a number of situations in which the software fails because the steps are not performed in the presumed order.

One of the steps in which the original EUDAQ was failing was during the run step of the of the devices. The original EUDAQ software allowed a user to start a run without configuring all their connections. Different devices have different configuration routines. Configuration is one of the processes that must be defined during the extension of the producer class. Some producers require variables to be set. Others need to perform specific configuring commands on the connected devices. Users were experiencing a failure of the framework when they were able to perform a run before the devices were properly configured. Undefined states are a symptom of the EUDAQ software not adhering to a Finite State Machine(FSM). These undefined states allowed for the user to perform actions before the program was ready to perform them.

In the original EUDAQ software states such as OK and OK:CONFIGURED are displayed in the main Run Control GUI and updated automatically. By starting at the top panel of the run control this state was tracked to the class that was keeping track of certain levels. The investigation of this error lead to the need for several modifications of the way that the state was being tracked in the EUDAQ framework and specifically in the producers. This modification prompted us to break the backwards compatibility of the EUDAQ software so as to require the future producers to properly implement FSMs.

2 EUDAQ Framework

EUDAQ has tiers of background processes that facilitate the functionality of state tracking. These background classes will contain the bulk of the modifications that constitute the difference between the original EUDAQ software and fsm branch of EUDAQ. The various class hierarchies are addressed, as well as the specific transport protocol used for data collection and logging. The class that should define the state of the machine, called Status, is explored and traced throughout the EUDAQ framework.

2.1 Class Hierarchies

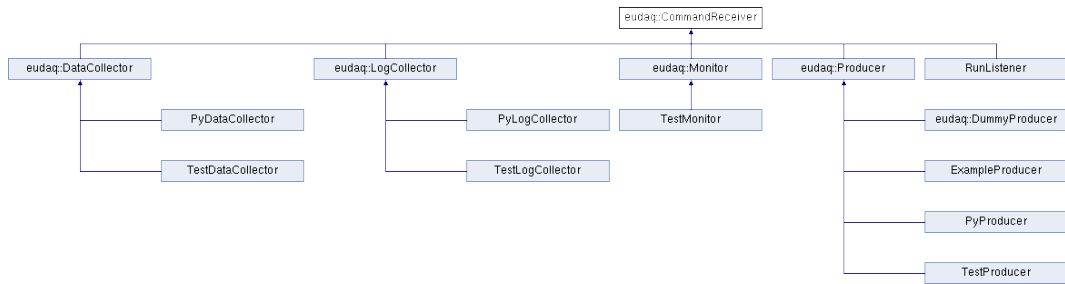


Figure 1: CommandReceiver Hierarchy

CommandReceiver The `CommandReceiver` class is the template for all classes that are required to either send or receive information at any point during the run time. This means that not only all the `LogCollectors`, `DataCollectors`, and `Monitors` inherit from this class, but so does any producer. In order to acquire data from a new producer a class specifically for that producer must be extended from the `Producer` class. As we can see in Figure 1, the producer class inherits from the `CommandReceiver` class, as does the `DataCollector`, `LogCollector`, `Monitor`, and `RunListener`. Modification of the base class `CommandReceiver` will prove an essential tool for propagating changes over all executables connected to EUDAQ.

Additionally, the `CommandReceiver` keeps an instance of `Status`. In the original program the `Status` class was responsible for keeping a redundant copy of log messages that were being handled by the EUDAQLOG system, which is the logging system that takes messages from the entire system and displays them in the logger. This redundancy will be addressed when the EUDAQ structure is modified for more appropriate state tracking.

RunControl The `RunControl` hierarchy represents the interface through which the user interacts and controls the connections to the EUDAQ framework. Every class that inher-

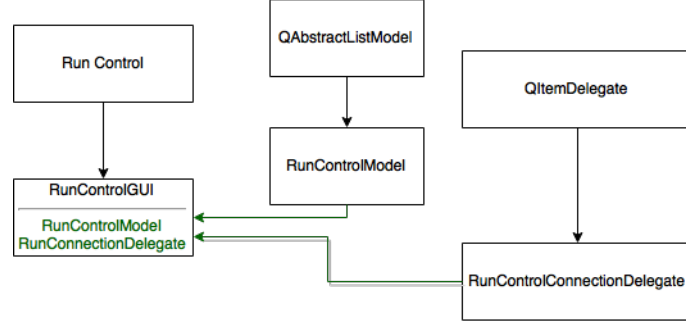


Figure 2: RunControl Hierarchy

its from CommandReceiver communicates directly with the RunControl. As is shown in Figure 2 the RunControl class is at the top of this hierarchy. This implements functions to send and receive commands as well as receive packets from the various CommandReceivers.

The RunControlConnectionDelegate handles the display of the connections and their states on the front panel of the RunControlGUI. In order to do this the RunControlModel keeps a vector of objects called RunControlConnections. These RunControlConnections have an instance of the Status class and an instance of the ConnectionInfo class. These two variables are kept updated by the main RunControl and so they always accurately represent what status has been logged by the status class and what the state of the connection is.

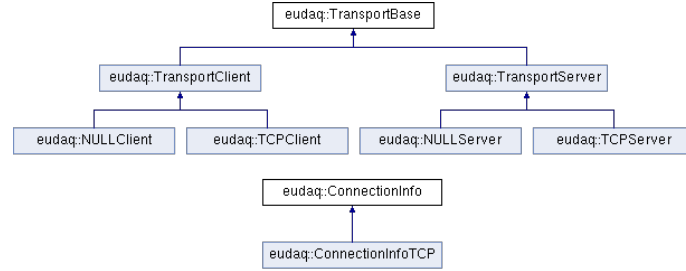


Figure 3: TransportBase Hierarchy

TransportBase The TransportBase hierarchy is essential to understanding how data and commands are sent from the producers connected to the EUDAQ framework to the main RunControl. The TransportBase hierarchy branches between server and client classes as is showed in Figure 3. Both the Transport Client class and the Transport Server class inherit directly from the TransportBase class. These classes are then fur-

ther extended to use the Transmission Control Protocol(TCP) server. All commands and data within the EUDAQ framework are sent using the TCP, for reasons that will be explained in Section 2.2.

There are also two `ConnectionInfo` classes that parallel the server structure. One of them is declared within the scope of `TransportBase` and the other is declared specifically within the `TransportTCP` class header. A `ConnectionInfo` object saves the information about the connection of whichever source it was declared in. The main difference between the `ConnectionInfo` and the `ConnectionInfoTCP` objects is that the TCP version specifically tracks the connection address of its origin, where as the base version does not.

There are two essential objects that provide additional functionality for the transport of data and commands within the EUDAQ framework that are not part of the `TransportBase` hierarchy. First is the `TransportFactory` class.

This class serves the purpose of creating and managing the server and client class for an individual connection. For example one can find in the `RunControl` class a call to the `TransportFactory` to create the command server for the `RunControl`.

The second essential class is the `TransportEvent` class. This class represents the object that is passed via the transport servers. The class itself is simple. It defines an enum with which the `EventType` can be defined. There are three types of `EventTypes`: `CONNECT`, `DISCONNECT`, and `RECEIVE`. The definition of these types allow producers and the `CommandReceiver` to properly handle events. The `TransportEvent` also keeps track of its sender via an instance of `ConnectionInfo`. Finally there is a string in the case of a `RECEIVE` event. This string stores the serialized version of the `Status`.

Serializable The transfer of the `Status` object is central to the tracking of state in EUDAQ. However, when using the transport server and specifically when using TCP we can not simply send the whole `Status` object via our server. It must first be serialized. Serialization is a method by which one can condense complex tree-like structures such as classes into a string which can be sent via our normal transport servers.

Every object that needs to be sent via the transport server inherits from this class. There are 10 classes that inherit from `Serializable`, as we can see in Figure 4. The children of this class simply need to implement the two key methods `Serialize` and `Deserialize`. The `Serializer` hierarchy then provides the rest of the functionality for storing the variables passed in a buffer and preparing them to be passed via the server. In the case of the `Status` class there are three variables that are serialized and deserialized using this method: the level, the message, and the tags.

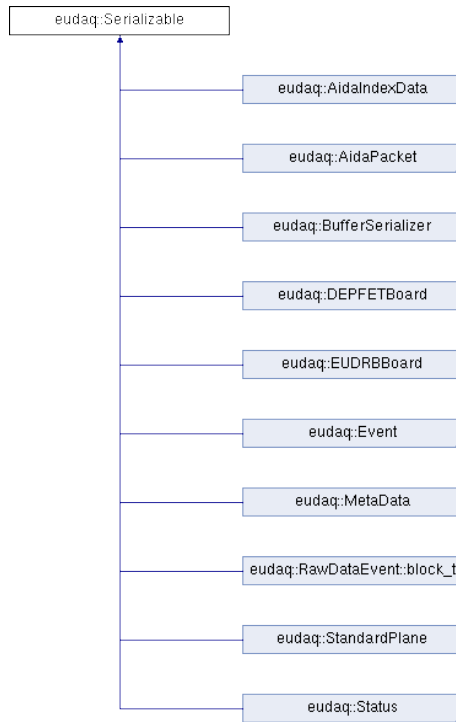


Figure 4: Serializable Hierarchy

2.2 Transmission Control Protocol

There is a large variety of protocols that one can choose to work with when transmitting information over a network. The Transmission Control Protocol has several features that make it quite useful for the purposes of data acquisition. The TCP protocol is specifically designed so as to ensure that all packets that are sent are received. This is a valuable property in data acquisition. The downside to this guarantee is that passing information via a TCP server is notably slower.

In order to connect the TCP must perform a three way handshake. This essentially consists of the client sending a SYN to the server, the server responding with a SYN-ACK and finally the client returning the ACK. This process ensures that both the client and the server are aware of the connection.

After this initial handshake is performed the server can begin sending data in packets to the client. This transmission of the data requires a similar handshake process to the initial connection. The server sends a packet with the data and then waits for the response from the client. If the ACK is received within the timeout period then the server sends the next packet. Otherwise the server resends the same packet. This is the reason that connections implementing TCP can significantly increase the wait time in between processes. Though it ensures the completion of the sent packet, the transmis-

sion of a single data packet can take a significant period of time.

2.3 Status Class

The Status class is the critical object that EUDAQ had been using to track the state of the connections and of the whole machine. Statuses are the objects that are serialized and passed between the producers and the Run Control. They are also responsible for the display on the front panel of the Run Control which displays important messages such as if the connection is ok and if the configuration has happened successfully. The main focus of the Status class for state tracking is the enum type and how it is used by the producers and the Run Control. As the class and its function is essential to this project, most of the text of the original Status header has been included. This will also clarify what is being referenced when specific functions and types are discussed.

```
15  class DLLEXPORT Status : public Serializable {
16  public:
17      enum Level {
18          LVL_DEBUG,
19          LVL_OK,
20          LVL_THROW,
21          LVL_EXTRA,
22          LVL_INFO,
23          LVL_WARN,
24          LVL_ERROR,
25          LVL_USER,
26          LVL_BUSY,
27          LVL_NONE // The last value, any additions should go before this
28      };
29      Status(int level = LVL_OK, const std::string &msg = "")
30          : m_level(level), m_msg(msg) {}
31      Status(Deserializer &);
32      virtual void Serialize(Serializer &) const;
33
34      Status &SetTag(const std::string &name, const std::string &val);
35      std::string GetTag(const std::string &name,
36                        const std::string &def = "") const;
37      static std::string Level2String(int level);
38      static int String2Level(const std::string &);
39      virtual ~Status() {}
40      virtual void print(std::ostream &) const;
41      int GetLevel() const { return m_level; }
```



```

42
43     protected:
44         typedef std::map<std::string , std::string> map_t;
45         int m_level;
46         std::string m_msg;
47         map_t m_tags; ///Metadata tags in (name=value) pairs of strings
48     };
49
50     inline std::ostream &operator<<(std::ostream &os, const Status &s) {
51         s.print(os);
52         return os;
53     }
54 }
55
56 #endif // EUDAQ_INCLUDED_Status

```

As has been noted before, the Status class extends the Serializer class. This allows for the structure of the Status class to be compressed into a string and passed via the TCP servers.

In lines 45-47 we see some of the variables associated with the Status class. These variables track a level, a message, and a series of tags. The constructor at line 29 takes both a level and a message. One should note that if there is no status or message given then the default state is OK with no message.

From line 17-28 the enum type for the levels is defined. This would ideally be where the vital states of the connection's state would be defined. However, the levels defined here are a mixture of log messages and levels to indicate whether the CommandReceiver is still connected to the RunControl. There is also no indication of the message that one can see on the front panel indicating the configuration of a connection. This is a obstacle for the use of the Status class to maintain the machine state. Additionally, this raises the question of where the configuration status is tracked , if not in the Status class.

On further inspection two important things are found. First, the EUDAQLOG system relies explicitly on the Status class to transmit its logs. It does this via an independent logging system. However its LogMessages inherit directly from the Status class. Thus, one can not remove the levels from the status class without modifying the entire structure of the EUDAQLOG.

The second discovery is that the CommandReceivers do not track their own configuration states. There is no record of this essential information. The "Configured" message that appears on the front panel is a string, written by the programmer of the producer, sent with the status on configure. However, this message is not kept anywhere and when the status changes the message disappears. There is also no standardization

of this message. The person writing the code could conceivably write anything in this message slot, or nothing.

2.4 Finite State Machine For EUDAQ

In order to forestall runtime errors the EUDAQ software must be required to adhere to a finite state machine defined by the architecture of the program. This is an attempt to prevent the software from entering deadlocks, or providing pathways that can crash the producers. Due to the fact that this finite state machine defines what we are attempting to do with the EUDAQ software it is appropriate to give a precise definition of what a FSM is and what the FSM we want for the EUDAQ software looks like.

A FSM is formally defined by two core principals:

1. The Machine can be in one of a finite number of states.
2. The Machine can transition between these states via well defined triggers.

These requirements address the first of the issues that was present in the EUDAQ software. If we have well defined states and transitions then we can disallow situations that will cause EUDAQ to crash, such as starting a run while producers are still unconfigured. By requiring these well defined states we can draw a chart of what the EUDAQ FSM should look like.

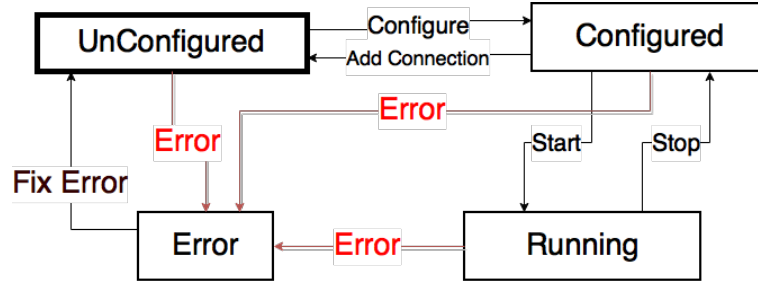


Figure 5: EUDAQ Finite State Machine

We can then define these states precisely for both an individual connection and for the entire EUDAQ program.

Connection States:

1. **ERROR:** An error has occurred with some part of this connection, whether in the server or in the producer. The connection can not be used until the error is resolved.
2. **UNCONF:** Any of the configuration variables for this connection have not been set, and the connection is not ready to be run.
3. **CONF:** All of the configuration variables have been set and all additional required configuration processes have been run. This connection is ready to be run.
4. **RUNNING:** The connection is currently running. The run state is defined differently for producers, data collectors, monitors, and loggers. For example in the case of a running producer this means that the connection is producing data.

Machine State

1. **ERROR:** In the case that any one of the connections is in an error state, then the whole program is in an error state and a run can not be started.
2. **UNCONF:** In the case that any one of the connections is not configured but all of them are not in an error state then the state is unconfigured and a run can not be started.
3. **CONF:** In the case that all of the connections are in the configured state or are running then the state of the whole program is configured and the run may be started.
4. **RUNNING:** In the case that all of the connections are running then the program is considering running. The only actions that are available at this point are stopping the run, terminating the program, or logging a message.

At this point we should compare our desired FSM to the original EUDAQ framework and examine the options that it presents.

2.5 Status Class as State Tracker

In the original EUDAQ framework the object responsible for keeping track of the state of individual producers was the Status class and its instances for the various producers, datacollectors, loggers, and other connections. The functionality of the Status class needs to be matched to the desired functionality of a state tracker in order to evaluate

the practicality of using it to define a FSM.

The major point of comparison is how well the enum states of the Status class correspond to our desired states for the FSM. There are 10 defined levels. As we have noted before in Section 2.3 most of these levels are for logging messages. Further examination of the uses of Status throughout the EUDAQ framework shows us that they are almost exclusively used for logging. However, their presence is redundant as the same task is preformed simultaneously by the EUDAQLOG system. This examination has made it clear that the producers do not keep track of their configuration or error status except through this class. This information which is vital to ensuring a smooth run of the EUDAQ software is not stored anywhere.

3 MachineState and ConnectionState Class

There is a clear need for a modification of the EUDAQ software to ensure proper tracking of state throughout the run of the program. These changes must occur on two levels, one on the level of the individual connection to track its own status and ensure that there is some record of its configuration status. Additionally we want the whole system to be able to emit its own state so that it can be determined if a run can be started, or if errors need to be resolved. The Status class is a good candidate for the tracking of state on the level of an individual connection, but due to the reliance of the EUDAQLOG system we do not want to confuse the purpose of the Status class by adding unrelated functionality. In order to use the structure that EUDAQ has for transferring Statuses we shall attempt to derive a class specifically for the state of the connection from the structure of the Status class. For the larger scale there is no preexisting structure that meets the requirements for tracking the state of the whole machine.

3.1 Implementation ConnectionState Class

Our new class should require the connection state to be in one of the states determined by the FSM we defined. In the same way that the levels are defined for the Status class we define the states of the ConnectionState class.

```
30     enum State {
31         STATE_UNCONF,
32         STATE_CONF,
33         STATE_RUNNING,
34         STATE_ERROR
35     };
```

This enum explicitly corresponds to the states of our FSM. As with the level an int is declared to keep track of the current state of the connection. This variable is set on

construction. As our end goal is to use this class to replace Status in the main body of the EUDAQ software, there is some additional functionality that must be added. One of the essential levels that was used when the Status was received by the RunControl is the BUSY level. This was used to determine the state in the ConnectionInfo. The current replacement in the ConnectionState class is the boolean isBusy. This is a variable that is set on construction, but is by default false. Any producer can set this to true when it sets its state in order to indicate that it is busy.

As with the Status class the relevant variables that define the ConnectionState must be serialized and deserialized to be passed via the TCP connection. By extending the Seralizer class and implementing the methods Serialize and Deserialize the Connection-State and its variable may be passed via the same structure that the Statuses were passed.

After this class was created there was the process of modifying the relevant classes to use the ConnectionState class rather than Status. There were approximately 20 classes that used or referenced the Status class. The classes that required modification spanned from the producers that referenced the SetState() method defined in CommandReceiver to the front end of the GUI which used the Statuses to create the state log on the front panel of the RunControl.

3.2 Implementation of MachineState

An activity updated record of the state of the whole program is required for the final functionality that is demanded of the EUDAQ software. However, with the exception of the state variable in the RunControlGUI, no part of EUDAQ had previously been keeping track of the state of the program. In order to remedy this the helper class MachineState was created.

```

13  class MachineState
14  {
15      public:
16
17          MachineState();
18          int GetState();
19          //Returns the state of the whole machine
20          GetState(ConnectionInfo id);
21          // Returns the state of a single connection
22
23          void SetState(ConnectionInfo id, Status* state);
24          // Sets the connection associated with id to state.

```

```

25      //if the connection does not exist, add it to the array
26
27      bool HasRunning();
28      //Returns true if there are running connections
29      void RemoveState(ConnectionInfo id);
30      //Removes a connection when it is disconnected....
31
32      void Print();
33      private:
34
35      std::map<ConnectionInfo, Status> connection_status_info;
36  };

```

The MachineState class maintains a list of connections and their states using a map of ConnectionInfo and Status objects. This allows for easy access of states for individual connections. As one can see in line 20, there exists a simple getter for the state of a connection. The functionality of the mapping type means that the state of any connection is easily accessed by sending the ConnectionInfo object associated with that connection to the MachineState class.

On line 18 a getter by the same name that takes no arguments will return the state of the whole program, based on the connections that MachineState currently has stored. This value is decided each time the function GetState is called. This is where the finite state machine that we defined for the EUDAQ software is formally used.

In particular one should note that our FSM does not allow for a Disconnected state. Thus, when a connection is no longer connected its ConnectionInfo and Status are simply removed from our map. If the connection is reestablished later it can be remapped, but we do not want to consider disconnected devices in our state determination. The name of the device and the status DEAD are still displayed on the front panel of the run control in the case the connection has been dropped. This is for user convenience only and not part of the state machine.

An instance of MachineState is kept and maintained by the RunControl base class. This way it can be updated every time the RunControl receives an update from one of the connections. This new system also allows us to shift the responsibility of state recording from the GUI class to the base of RunControl. Previously the RunControlGUI kept an enum responsible for determining which state the program was in. The enum still exists in the GUI class, but it now corresponds to the FSM defined for EUDAQ and is updated by the MachineState.

3.3 Modification of GUI

In addition to the changes and clarification of state that occurred within the EUDAQ framework, it is also important to keep the user of the software informed about the current state. In the current version of EUDAQ the MachineState does not only control which transitions the user can perform, but also what state is displayed on the front panel.

As we can see in Figure 6, there is one very notable change from the old to new GUI,

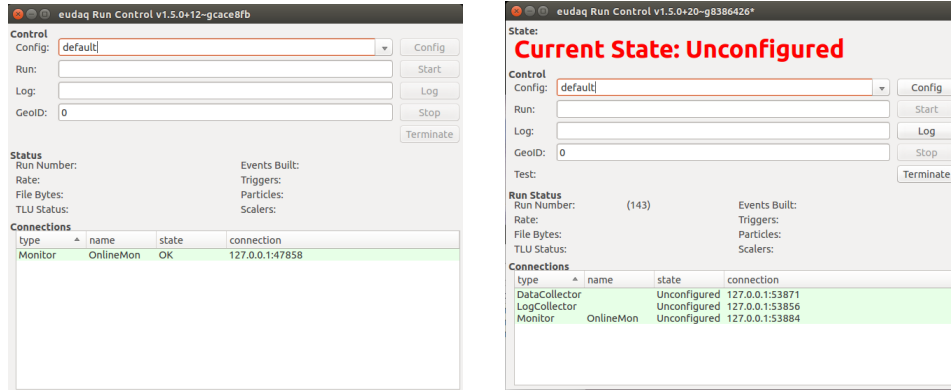


Figure 6: The Before and After GUI

and some more subtle ones. The first change is the presence of the current state label at the top of the RunControl panel. As stated, this label changes dynamically with the state of the program, with different colors corresponding to different states. At all times the user will be aware of what state the program is in and thus what actions are allowed.

The second, smaller change is in the Connections list. Instead of displaying the levels from the Status class the state is now listed next to the connection. This way the user knows precisely which connections are in which state. This information eases the process of troubleshooting your run.

4 Backward Compatibility

Currently the fsm version of the EUDAQ framework is in conflict with the implementation of most of the device specific producer code written for the original EUDAQ framework. As the state is no longer updated with the Status class, producers that use Statuses and SetState() find that their producer code no longer compiles with the fsm version of the EUDAQ software. In this way the FSM modification is not backwards compatible.

However, this breaking of backwards compatibility is a deliberate choice made to encourage the modification of producer code so that the state of all producers is always clearly defined. The alternative situation in which we allow producers to choose whether to define their own states or not creates more problems in the long run for the software. However, in order to ease this transition the new methods for defining the state are almost identical to the methods used for defining levels.

```

1 //Before:
2     SetState(LVL_OK,    Some Message );
3
4 //Now:
5     SetConnectionState(STATE_CONF,    Some Message );

```

In most cases the users of individual producers will simply modify one line that previously existed in their code to something referencing to a state rather than a level. Often the producers update their status at the stages which are essential to updating their state anyway, such as on configuring, on starting, and on stopping. In the end the producers will gain more control in the context of the larger framework. This modification also allows individual producers to dictate more explicitly what actions can be performed on them at what time. For instance, if one had a producer that needed to be reconfigured after each run it would be simple to alter the code to set the state to UNCONF after each run, thus requiring the user to reconfigure the producer when it needs to be reconfigured.

Although this change will force modification of every producers code, the end result is a system that functions more smoothly and gives the individual producers more control and allows them to customize their states to their own needs.

5 Moving Forward

The addition of the FSM modification to the main branch allows us to consider what further changes would improve the functionality of the EUDAQ framework and the FSM classes themselves.

5.1 Transition States

The current implementation of the MachineState is based on the four states that we require the EUDAQ software to adhere to during its run. However due to the reality of dealing with multiple producers that preform at different speeds for things such as configuring and starting runs, we often find that our program can spend significant periods of time in states where the connections are not all in the same state. This occurs especially while the run is starting or stopping and some of the connections are in the

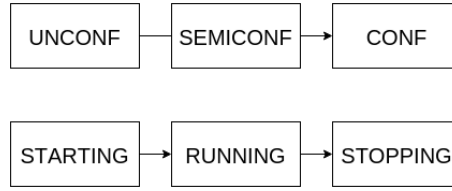


Figure 7: Transition States

CONF state while others are still in the RUNNING state.

We have defined our state machine in such a way that we always have a precise definition for what these mixed connection states mean for the program state as a whole. In the case described above, we know that the state of the whole machine is only RUNNING if all of the connections are in state RUNNING. Thus, case where some of our connections are CONF and some are RUNNING we know that the state of the whole machine is CONF. Our machine state remains well defined.

However, in reality we might want to know more details about what the program is attempting to do. Often it is useful to know if the program is starting a run or stopping a run. A solution to this might be to introduce some addition transition states for the FSM. Some examples of these transition states are shown in Figure 7.

The challenge with integrating these additional states with the main program is that the change would require much more work for each of the producers to ensure that they emit the proper states during starts and stops.

5.2 Error Handling

In implementing the FSM modifications in EUDAQ a certain quirk of the system became apparent. Although every producer has the option to be in a state of error and the program as a whole can emit error states, there is no well defined way to address these errors. For the most part one either has to reconfigure with different settings or adjust the physical device that is in the error state. However, in our FSM for EUDAQ (Figure 5) we have defined that the idea transition out of ERROR state would be to an UNCONF state via some error fixing method. This transition presents some difficulty because there are several ways that one can attempt to fix an error within the EUDAQ framework and none of them are defined explicitly by EUDAQ.

There is the additional difficulty that none of the connections currently have a way to unconfigure themselves, making it nearly impossible to bring the machine back to the original state of UNCONF. Unlike the SetState to SetConnectionState modification which only required the producers to change part of one line of their code, the change

to implement an unconfigure method would be involved for each of the producers.

This presents a conflict between the desire for a clean FSM and the reality of how errors are handled for producers. Although a thorough mechanism to handle errors from EUDAQ might be impossible to realize, it would be useful and possible to implement more guidance and some method that takes the connections from ERROR to UNCONF.

6 Summary

The contributions of this work have been twofold. In the process of assessing how well the EUDAQ framework adhered to a FSM material for further documentation of the EUDAQ framework was gathered. This new documentation can inform future programmers on the mechanisms of EUDAQ, to expedite the introduction to EUDAQ. The next modifications of EDUDAQ should be easier due to this.

The additional documentation of the configuration files and how they are handled in the EUDAQ framework can be found in the appendix of this report. The documentation follows the configuration files and their use in the framework and could illuminate the methods by which one would modify the configuration methods so as to provide greater flexibility. The OnConfigure() methods of individual producers are not included due to their length and density.

The changes in the actual EUDAQ software constitute the main bulk of the contribution. The history of these changes are summarized on the git page created for the fsm-branch of EUDAQ. In essence the changes consisted of the addition of the two state classes: ConnectionState and MachineState, and then the propagation of their use throughout the rest of the framework. The CommandReceiver class was modified to use ConnectionState instead of Status. The functions that send and set these states throughout EUDAQ were modified in turn to handle ConnectionState objects. The front end of the RunControl now uses the ConnectionState objects to display states associated with connections, as opposed to the Status levels.

RunControl has an instance of MachineState which handles the determination of the current state for the RunControl. This controls what user input is valid at certain times and is the final step that prevents the user from having access to actions such as starting a run when the MachineState is not fully configured.

Finally several additional steps in the fine tuning of the FSM have been identified as possibilities for future work. This has been done in an attempt give direction to pre-empting issues that may arise in the use of the fsm version of the EUDAQ software.

References

- [1] EUDAQ Software User Manual, *EUDAQ Development Team*
URL: <http://eudaq.github.io/manual/EUDAQUserManual.pdf>

Appendices

~~~~~  
~~~~~Configuration Tracking~~~~~  
~~~~~

-----  
Configuration Class Structure:  
-----

-----\_Public\_-----  
-----

Constructors:  
-----

Configuration has three constructors:

- (1) Takes two strings  
-> config: Which is the name of the configuration  
-> section: Which is the section of the configuration desired
- (2) Takes two istreams  
-> config: Which is the name of the configuration  
-> section: Which is the section of the configuration desired
- (3) Takes another instance of a Configure class

-----  
Functions:  
-----

File access and modification:

- (1) Save  
-> Takes ostream  
-> Writes file to ostream  
-> Void function
- (2) Load  
-> Takes istream  
-> Loads file from istream  
-> Void function
- (3) Print const

- > Takes ostream
- > Prints the config file to given stream
- > Void function

(4) Print const

- > Calls Print(std::cout)
- > Void function

Two Section Selection functions:

(1) SetSelection

- > Takes a string of the section name
- > Sets current section name to given string
- > Always returns true

(2) SetSelection const

- > Takes a string of the section name
- > Tests if that section exists
- > if the section exists then sets the current section equal to that section
- > Returns true if the section exists, false if it doesn't

Two operator modifications:

(1) operator[]

- > Takes string representing a key as argument.
- > Returns string associated with that key

(2) operator=

- > Takes another instance of Configuration called other
- > Makes the m\_config and m\_section of current Configuration equal to those other.

Ten Get Functions:

(1) string Get const

- > Takes a string representing a key and a string representing a default
- > Tries to get string associated with key
- > If string is found, returns string. Otherwise returns def(ault)

(2) double Get const

- > Takes a string representing a key and a double representing a default
- > Tries to get double associated with key
- > If double is found ,returns double. Otherwise returns def(ault)

(3) int64\_t Get const

- > Takes a string representing a key and a 64 bit int representing a default
- > Tries to find 64 bit int associated with key
- > If 64 bit int is found, returns int. Otherwise returns def(ault)

(4) uint64\_t Get const

- > Takes a string representing a key and a 64 bit int representing a default
- > Tries to find 64 bit int associated with key
- > If 64 bit int is found, returns int. Otherwise returns def(ault)

(5) template Get const

- > Takes a string representing a key and a template representing a default
- > Tries to find template associated with key
- > If template is found, returns int. Otherwise returns def(ault)

(6) int Get const

- > Takes a string representing a key and an int representing a default
- > Tries to find int associated with key
- > If int is found, returns int. Otherwise returns def(ault)

(7) template Get const

- > Takes a string representing a key, a string representing a fallback key, and a template for the default
- > Tries to find template associated with key
- > If key is not found, tries to find fallback
- > If template is found, returns template. If template is not found, but the fallback is, returns template associated with fallback. Otherwise returns def(ault)

(8) string Get const

- > Takes a string representing a key and a char representing default.
- > Tries to find char associated with key
- > If char was found, return char. Otherwise return def(ault)

(9) string Get const

- > Takes a string representing a key, a string representing a fallback key, and a string for the default
- > Tries to find string associated with key
- > If key is not found, tries to find fallback
- > If string is found, returns template. If string is not found, but the fallback is, returns string associated with fallback. Otherwise returns def(ault)

(10) string Name const

- > Returns the name of the configuration, if it exists

Set Function:

(1) template Set

- > Takes a string representing a key and a template of a value

```

-----Private-----

-----
Types:
-----

(1) section_t
-> map of a string to a string

(2) map_t
-> map of a string to a section_t

-----
Variables:
-----

(1) mutable string m_section
-> String representing section
(2) mutable section_t m_cur
-> Map of a string to a string representing current key(?)

-----
Functions:
-----

(1) string GetString const
-> Takes a string representing a key
-> Tries to find key
-> If key is found returns the key
-> If key is not found throws exception "Configuration: key not found"

(2) SetString
-> Takes a string representing a key and a string representing a value
-> Finds the key string in the current section map and assigns the value string to
   it
-> Void function

~~~~~

Files that use Configuration:

Configuration.cc
CommandReceiver.cc
PluginManager.cc
RunControl.cc
EventSynchronisationBase.cc
DataCollector.cc
PALPIDFSConverterPlugin.cc
FileReader.cc
PALPIDESSConverterPlugin.cc

```

```

Configuration.cc

```

Base Configure File. Handles the reading and storing of settings from the .conf file.

```

CommandReceiver.cc

```

Prints the Configure file via std

```
void CommandReceiver::OnConfigure(const Configuration ¶m) {
 std::cout << "Config:\n" << param << std::endl;
}
```

```

PluginManager.cc

```

Handles the different plugins for various Producers. For conf, handles the need for the GeoID in the case of the EUTel, when the runheader needs this information.

```

EventSynchronisationBase.cc

```

From the .conf file, accesses the Event Structure settings. Then it accesses the settings for what a long busy time is (longTimeDiff\_) and the number of events (NumberOfEventsToSync\_).

```
eudaq::Configuration conf(BOREvent.GetTag("CONFIG"));
conf.SetSection("EventStruct");

longTimeDiff_ = conf.Get("LongBusyTime", longTimeDiff_); // from config file
longTimeDiff_ =
 BOREvent.GetTag("longTimeDelay", longTimeDiff_); // from command line

NumberOfEventsToSync_ =
 conf.Get("NumberOfEvents", NumberOfEventsToSync_); // from config file
NumberOfEventsToSync_ = BOREvent.GetTag(
 "NumberOfEvents", NumberOfEventsToSync_); // from command line
```

```

DataCollector.cc

```

Upon receiving the Configure command the DataCollector function uses Configure object

to obtain configuration setting from the .conf file. The important settings that it acquires are the file Type and Pattern that the DataCollector will write. These settings are stored in the variable m\_writer which is an instance of the std File Writer Class.

```
void DataCollector::OnConfigure(const Configuration ¶m) {
 m_config = param;
 m_writer = std::shared_ptr<eudaq::FileWriter>(
 FileWriterFactory::Create(m_config.Get("FileType", "")));
 m_writer->SetFilePattern(m_config.Get("FilePattern", ""));
}
```

Additionally, if the event is BORE(Begining of Run Event) the class sets its tag to the configure settings

```
if (ev.IsBORE()) {
 ev.SetTag("STARTTIME", m_runstart.Formatted());
 ev.SetTag("CONFIG", to_string(m_config));
 found_bore = true;
}
```

-----  
FileReader.cc  
-----

The FileReader Class is specifically designed to accept settings from the .conf files and to read in the data as collected from the DataCollector. The FileReader class has two defining constants, the file and the file pattern.

```
eudaq::Configuration conf(GetDetectorEvent().GetTag("CONFIG"));
conf.SetSection("EventStruct");
```

~~~~~  
~~~~~Command Reciever Notes~~~~~  
~~~~~

-----Command Handler-----

The CommandHandler handles the events from the transport server. It takes a Transport Event as a parameter.

In the case that the event is a recieve type:

The function stores the configure parameter and the command in the strings called param and cmd.

The function then does a case by case consideration of what the cmd is, and takes the respective action.

In the case that the cmd is not recongized, the function calls



OnUnrecognised(cmd,param)

#### -----Command Reciever-----

This CommandReciever constructor handles the retrieval of information via the TCP server. It takes three strings: type, name, and runcontrol as well as one bool startthread as parameters.

It sets the defaults for the:

- m\_cmdclient from string runcontrol,
- m\_done(false),
- m\_type from string type,
- m\_name from string name,
- and m\_threadcreated(false).

First the reciever ensures that the transport server is valid.

- In the case that the retrieval of the packet is impossible, the program throws an error

- In the case that the end of the string is found early at any point, the program throws an error

- In the case that the first part of the message is not OK, the program throws an error

- In the case that the second part of the message is not EUDAQ, the program throws an error

- In the case that the third part of the message is not CMD, the program throws an error

- In the case that the fourth part of the message is not RunControl, the program throws an error

If an error has not been thrown at this point the program sends the command "OK CMD RunControl" to the transport server along with the type and name given to the command reciever.

- If the RunControl does not send a packet, the program throws an error.

- The response from the RunControl is recieved in the string "packet"

- If the first part of the packet is not OK, the program throws an error.

This program is then set as the callback for the transport server and the CommandHandler is set as the handler. If the thread has not been started, the program starts the thread.