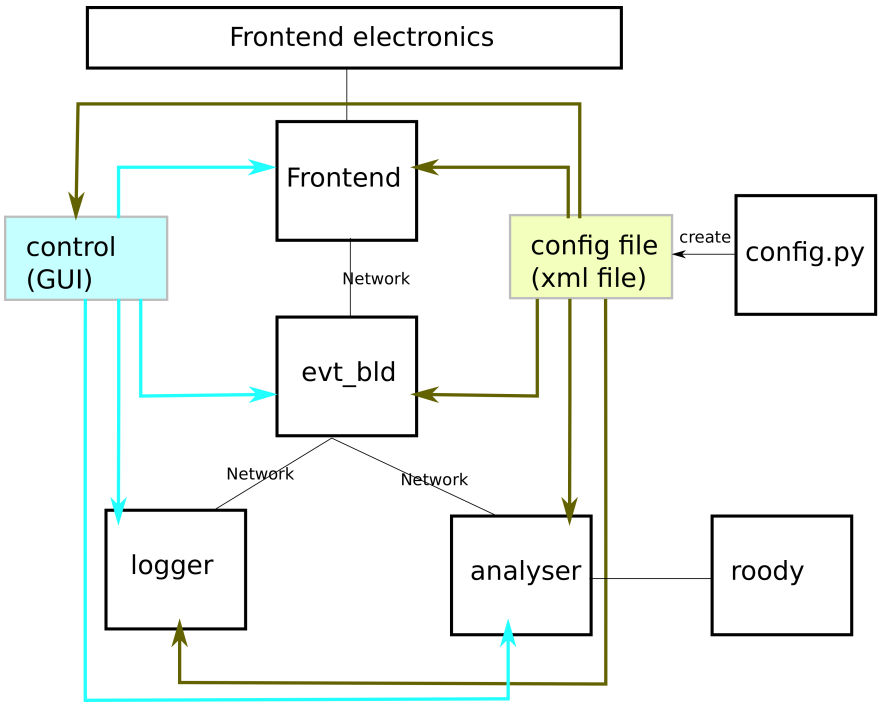

Figure 1: Components of the DAQ

1 Purpose of this DAQ ?

I wrote this DAQ in the hope for it to become the (prototype of) standard DAQ used by IMP. As far as I learned, by the time this DAQ was written, no 'standard' DAQ existed in IMP and nobody was working on this, even (probably) nobody was thinking about such a thing. This is, in my humble opinion, one major thing that we were behind others (e.g. GSI, NSCL/FRIB). However, to really develop a universal DAQ applicable to many experiments, we may need a whole group to do that. I cannot do that on my own. So in this version of DAQ, I kept many things as simple as possible and many things remained unoptimized. The bottom line was to make sure that it works and can be easily used by others. How easy could it be ? Well, in most cases, one should be able to set it up by just clicking mouse and fill in some parameters (like module base addresses) without any coding (except for the online analysis part). To that end, one has to use only the modules predefined in this DAQ, unsupported modules won't work properly (in fact, they won't work at all). What if one needs to use an unsupported module? There are three ways to work it around: i) find an alternative module; ii) contact me to include your module (gaobsh@impcas.ac.cn); iii) do it yourself, you should be able to include your module easily since special care was taken to accomplish that.

Part of the DAQ designment is inspired by the NSCL DAQ.

2 Overview of the DAQ

This DAQ consists of the following parts, see Fig. 1:

- config. This is a GUI program (python script) to help user create the configuration file used by the DAQ. The configuration file (xml file) contains all the information needed to setup the DAQ (used modules, base addresses, register settings of modules, etc.).
- frontend. This is the part that communicates with the electronics (e.g. VME modules).

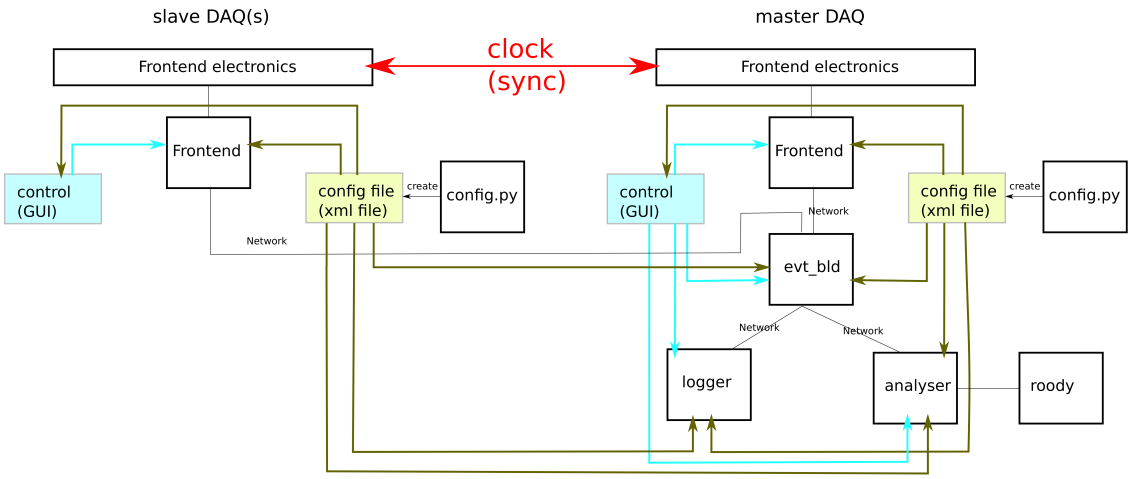

Figure 2: Configuration of the DAQ with two DAQs

- evt_bld. This is the event builder. It receives data from the frontend and build a complete event based on timestamps of the fragments readout by frontend.
- logger. This program takes data from evt_bld and records it in hard drives.
- analyser. This program also takes data from evt_bld, it analysis the events and makes histograms instead of recording them.
- roody. This program displays the histograms made by the analyser. Strictly speaking, roody is not really part of the DAQ. It should be seen as a 'hist displayer'.
- control. This is a GUI program which controls the running of the DAQ (e.g. start, stop, quit...) and shows statistics (e.g. kb/s, loads of ring buffers...). It also receives messages from other components and save it into files (and/or show it on the screen). It can also sample data from ring buffers of other components and print it out (for debugging).

The communications between different programs is done via sockets. This makes it easier to combine two DAQs into one as shown in Fig. 2. Combining multipole DAQs involves the following modifications:

- The slave DAQ(s) don't have the event builder and the subsequent components, the data readout by its frontend are send to the event builder of the master DAQ.
- To enable the event builder and subsequent components of the master DAQ work properly, they need read the configuration file of the slave DAQ(s) too.
- Because the event builder build events based on time stamps, the clocks of the DAQs should be synchronized.

The basic requirement is that all the supported modules (trigger type) should have time stamps for each event, so that the event builder can build events based on their time stamps. However, there should be some mechanism that allows user to use modules that don't have time stamps (may at the cost of significantly downgraded performance though).

3 Frontend

The first question about frontend is how to (possibly) write a frontend which can be applicable in various experiments using various modules ? The basic idea is to include all supported modules in the code, then select those used in real experiments by using if statements. To be more specific, the frontend can be thought of as doing the following things: i) initialize modules, ii) wait for trigger, iii) readout data and goto last step again. So for example what we are going to do in step iii is to read out data from all the modules that are actually used in the experiment. How do we know which modules are used ? We can find it out from the configuration file.

About the trigger: It is very difficult to decide the trigger mechanism when we don't know the electronics setup at all. Since here we are trying to develop a 'universal' DAQ, a 'universal' trigger mechanism has to be figured out. The best one that I can think of is to use the I/O register module V977 as the 'trigger module'. So in order to use this DAQ, it is preferable to have a V977 module (as the trigger module). However, any module *can* be used as a trigger module. The function 'module::if_trig()' should be overwritten if the module is going to be the trigger module.

Since frontend will communicate with the electronics, connections of the hardware is needed to run the program. However, for testing purposes, it should provide a mechanism to run the program without really connecting the hardware electronics (just to see if the DAQ works at all).

The main task of frontend is to read data from modules and send the data out to event builder with minimum pre-processing. The modules can be categorized into two types: 'trigger type' and 'scaler type'. The 'trigger type' modules are read whenever there is a trigger, the 'scaler type' modules are read regularly, independent of the trigger. Each scaler-type module has a reading period. If the period is 0, then the module is never going to be readout. Meanwhile the frontend has to communicate with the controller. So four threads should be used:

- Thread 1, It waits for trigger and read data from modules. Then it packs and copies the data into a ring buffer.
- Thread 2. It reads data regularly from scaler type modules and packs and copies the data into the ring buffer as in thread 1.
- Thread 3. It reads data packets from the ring buffer and sends them out to the event builder (Tcp server).
- Thread 4. It communicates with the controller (Tcp client).

To make life easier, it is better to organize these things in classes. To do this, (at least) five classes are needed:

- One class for each of the threads.
- Another class called 'initializer' which initialize everything (including the modules, ring buffer ...). This class can also be used in other parts (e.g. event builder) of the DAQ to initialize.

4 event builder

The main task of event builder is to build event from the data received from frontend based on their time stamps. Four threads should be used:

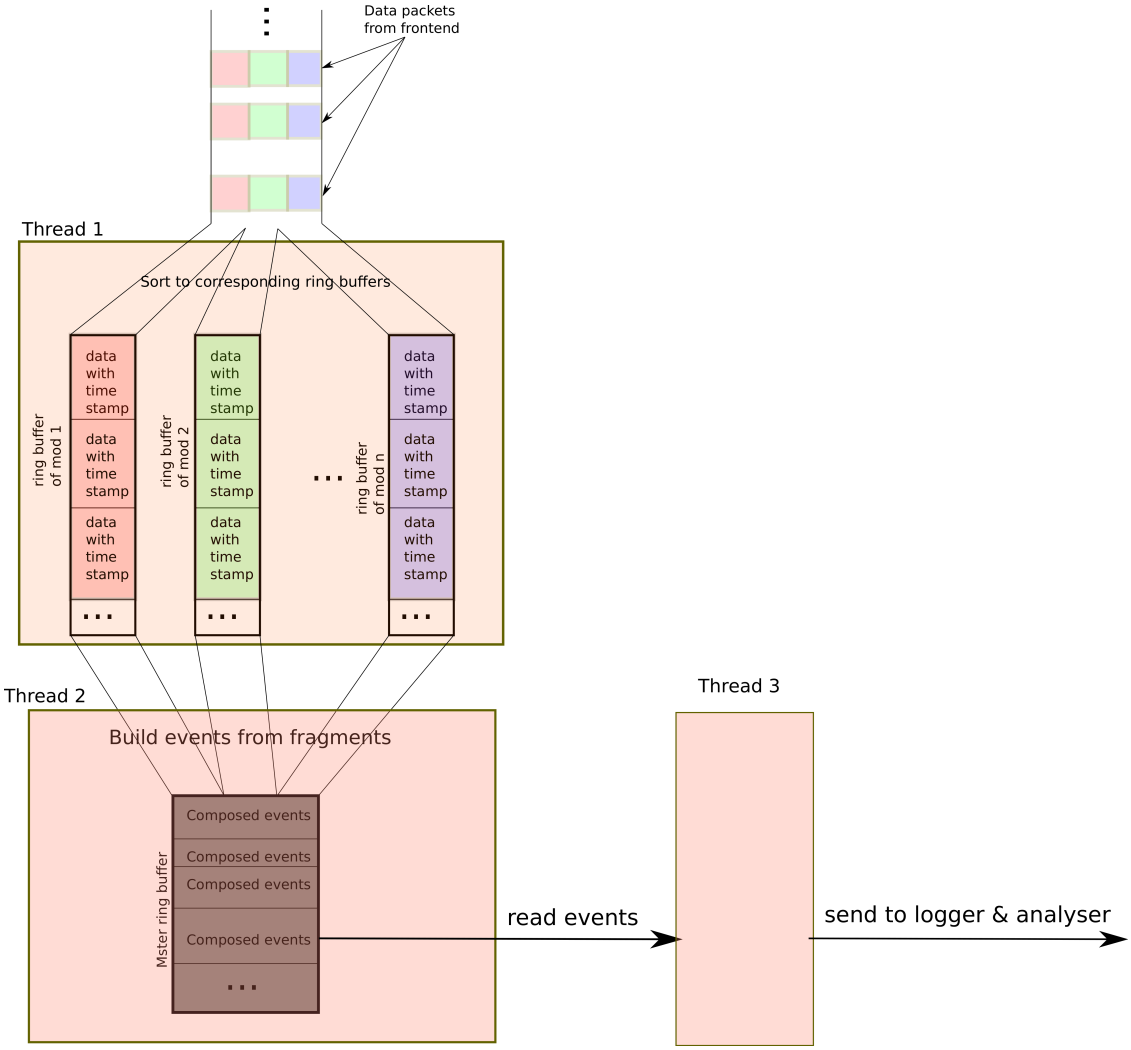

Figure 3: Data flow in the event builder

- Thread 1. It receives data from frontend (Tcp client) and saves the data into a ring buffer.
- Thread 2. It sorts the data into different ring buffers. Here each single module (adc, tdc ...) has a dedicated ring buffer. In each ring buffer, the events are sorted by their time stamps.
- Thread 3 (main thread). It reads data from the ring buffers in thread 1 and build events out of the fragments contained in the ring buffers. It then copies the composed event into the 'master ring buffer'.
- Thread 4. It read the composed events from the 'master ring buffer' and sends them out to logger & analyser (Tcp server).
- Thread 5. It communicates with the controller (Tcp client).

The processing of the data flow is demonstrated in Fig. 3.

5 Analyzer

This is the online analysis of the experimental data. It receives (composed) events from the event builder and do meaningful analysis and fill user defined histograms. However, the displaying of histograms are done by another program 'roody' (see next section). Four threads should be used in analyser:

- Thread 1. It receives events from event builder and save them in a ring buffer.
- Thread 2 (main thread). It analysis the events stored in the ring buffer and fill the user defined histograms.
- Thread 3. It communicates with the histogram displayer (Tcp server). For example it sends data to and receives requests from the displayer (roody).
- Thread 4. It communicates with the controller (Tcp client).

Analyser is the only program that needs user coding, so the source files should be organized such that the user coding file is clearly separate from the other back end files. For this purpose, a file named user_code.cpp should be used for the user code. In this file two functions exists for the user to modify: book_hists() and do_analysis(). The book_hists() defines histograms that the user wants to see from the online analysis. The do_analysis() receives a pointer to an event and do whatever analysis the user wants, and then fill the histograms. Note: when filling the histograms, synchronization should be considered because thread 3 may also be reading the histograms.

6 roody

Roody is developed by the TRIUMF (https://www.triumf.info/wiki/DAQwiki/index.php/ROODY). It is better to get a stable working version (based on the same version of root as Analyser, preferably root version of 5.32) and not to change that.

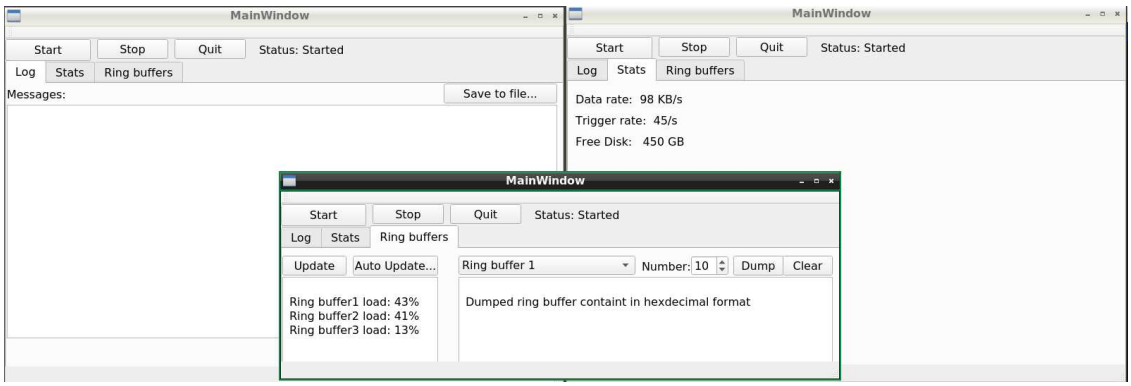

Figure 4: The program 'control'

7 control

The program control (a python script) is the controller of the whole DAQ. Since it is just a controller of the DAQ, it should be designed such that the DAQ is OK if the control program exited (unexpectedly). For example, we can launch the DAQ without launching the controller at all (however we have no control over the DAQ though, e.g. to start a run). We can also start a run using the control then exit the controller, then we can launch the controller again and everything should be as if we haven't exited the controller. The controller performs the following tasks:

- Controls the status of the DAQ (start, stop, quit).
- Receives messages from other components of the DAQ and save it to files.
- Shows the statistics of the current run (e.g. kb/s, ring buffer loads, free disk space...)
- Shows the samples of the data in the ring buffers (for debugging).

To accomplish the above tasks, it should be like 4. The widgets are:

- Start (button). When pressed, a dialog should be popped up asking for the run number and the title, then start a new run. When the DAQ status is already 'Started', this button should be disabled to prevent double start.
- Stop (button). When pressed, a dialog should be popped up asking like 'are you sure to stop ?', when confirmed, stop the current run. When the DAQ is already 'Stopped', this button should be disabled to prevent double stop.
- Quit (button). When pressed, a dialog should be pooped up asking like 'are you sure to quit ?', when confirmed, quit the whole DAQ. The button should be disabled if the DAQ is currently started, i.e. the user can quit the DAQ only if it is stopped.
- Status (label or text box). This shows the current status of the DAQ: either started or stopped. For simplicity, we don't provide 'paused' status. To obtain the status, the control program should query the frontend (i.e. the control itself should not keep track of the status of the DAQ, the frontend should do that).
- The log page (page). It contains two widgets:
  - Messages (read only text box). It shows the messages received from other components of the DAQ.

  - Save to file... (button). When clicked, a dialog popped up asking for the file name to save the contents in the Messages, then save the messages.

- Stats (page). It contains a read only text box showing the following information:
  - Data rate.
  - Trigger rate.
  - Free disk space.
  - Something else... (To be added on demand).

- Ring buffers (page). It contains the following widgets:
  - Update (button). When clicked, update the ring buffer loads shown in the text box below the button. Because this ring buffer information is mainly used for debugging purposes, it is not supposed to be updated very frequently.
  - Auto Update...(button). When clicked, a dialog pops up asking for the update frequency of the ring buffer loads, then update the loads automatically according to the frequency.
  - Ring buffer loads (read only text box). It shows the load of each ring buffer (occupancy percentage of the buffer).
  - Ring buffer name (ComboBox). It selects the ring buffer whose contents are to be dumped.
  - Number (SpinBox). It sets the number of items in the ring buffer to dump.
  - Dump (button). When clicked, the contents of the specified ring buffer is dumped in the text box below the button (in hexadecimal format)
  - Clear (button). When clicked, a dialog pops up asking for confirmation, then clears the contents of the text box below the button.
  - Dumped contents (read only text box). It shows the dumped contents of a ring buffer in hexadecimal format.

8 config

This program create a configuration file (or modifies an existing one) used by all other components of the DAQ. All information needed to run the DAQ is contained in the configuration file which is a xml file.

The *config* program should look like Fig. 5. It includes several pages of which the *Frontend* page is most important and provides the most crucial information. It includes the following widgets:

- Open... (button). When clicked, it pops up a window allowing the user to select the configuration file to be opened. In this case, we work on an existing file.

- Save (button). When clicked, the modifications are saved to the existing file. In case of new file, it pops up a window asking for the file name of the new (created) file and then save it.

- Supported Modules (list widget). In this list, all the supported modules are shown.

- Add==> (button). Add the selected module in the left list into the right list (Selected modules). All the settings of the added module is set to its default values.

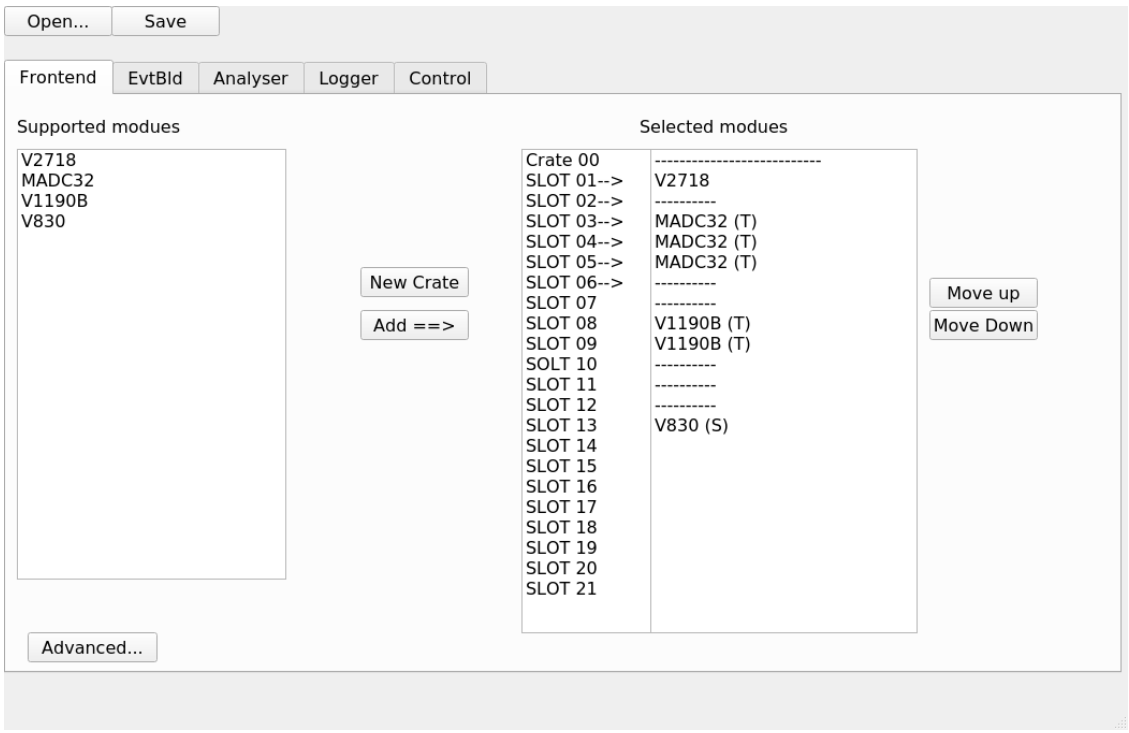

Figure 5: The *config* program.

- Selected Modules (list widget). In this list, all the selected modules are shown and these modules are to be used in the DAQ. The letters (T) and (S) indicates their type (T for trigger type and S for scaler type). When double click on one item, a window pops up for the user to modify the settings of the item (module). Right click the item allows the user to remove, duplicate (and maybe some other operations) the module.
- New Crate (button). When clicked, a new crate is created allowing user to add modules to a new crate.
- Advanced... (button). When clicked, it pops up a window for more advanced settings of the frontend. These settings are usually not supposed to be modified by most users.
- Move up/ Move Down (button). It allows the user to change the slots of the selected item in the 'Selected Modules' list.