Flexible online monitoring for high-energy physics with Pyrame

To cite this article: Miguel Rubio-Roy et al 2017 J. Phys.: Conf. Ser. 898 032009

View the article online for updates and enhancements.

Related content
- Pyrame, a rapid-prototyping framework for online systems
  Frédéric Magniette, Miguel Rubio-Roy and Floris Thiant
- High-Energy Physics: High hopes for B-factory
- High-Energy Physics: Studies in Natural Sciences Vol 20
  L. H. Ryder

Recent citations
- Pyrame 3, an online framework for Calice SiW-Ecal
  F. Magniette and A. Irles
- Latest R&D news and beam test performance of the highly granular SiW-ECAL technological prototype for the ILC
  A. Irles
Flexible online monitoring for high-energy physics with Pyrame

Miguel Rubio-Roy, Floris Thiant and Frédéric Magniette
LLR, Ecole Polytechnique, CNRS, F-91128 Palaiseau, France
E-mail:rubio-roy@llr.in2p3.fr, thiant@llr.in2p3.fr, magniette@llr.in2p3.fr

Abstract. The present work reports on the new online monitoring capabilities of the software suite Pyrame, an open-source framework designed for high energy physics (HEP) applications. Pyrame provides an easy-to-deploy solution for command, control and data-acquisition of particle detectors and related test-benches. Pyrame’s new online monitoring architecture is based on the distribution of the data treatment operations among multiple modules in the system, with multiple input and output streams. Uncontrolled data loss is prevented by providing data at the speed of consumers. In addition to the distributed data treatment capabilities, Pyrame includes a performance-oriented module dedicated to real-time data acquisition, capable of handling and storing data at 4 Gbit/s for further treatment.

1. Introduction
Data quality monitoring (DQM) in high-energy physics (HEP) experiments is essential and widely implemented in most large experiments. It provides important real-time information during the commissioning and production phases that allows the early identification of potential issues and eases their resolution. Also, in the era of experiments that produce huge amounts of data[1][2], DQM, and more generally online monitoring, helps by early preventing the acquisition of faulty or low quality data, thereby reducing the cost of dead time. Many other fields have increasingly similar needs, as the volume and complexity of output data grows [3].

Existing and performant solutions for online monitoring exist for large experiments such as CMS [4], ALICE [5] and others. The new Pyrame 3.0 aims to provide a flexible and generic solution for online monitoring, scalable from small test benches to medium-sized experiments.

Pyrame’s new online monitoring architecture is based on the distribution of the data treatment operations among any module of the system, with multiple input and output streams. A common problem in such systems is that different treatment speeds can lead to uncontrolled data loss. To avoid such situation we implement a mechanism that breaks the real-time constraint, at any treatment level, by buffering data and distributing it at the speed of the consumers (potentially leading to subsampling in the worst case). In addition to the distributed data treatment capabilities, Pyrame includes a performance-oriented module, the acquisition chain, dedicated to real-time data acquisition, capable of handling and storing data at 4 Gbit/s for further treatment.

This approach allows us to use the same code for online and offline data treatment. In the latter case, subsampling is forbidden and a throttleable input is used, so that we can control the
Figure 1. Architecture of Pyrame’s data chain. The real time domain contains the acquisition chain and the converters’ input. The first dispatcher breaks the real-time constrain and we enter a domain in which subsampling is allowed.

input speed. Legacy ROOT, R, or Python/Panda offline data treatment codes can be adapted to be fed by a file in offline mode or by one of the available online data sources.

The distribution of the data treatment chain over any module allows us to use a modular design. Data filtering operations can be processed first, providing cut or tagged data with a first level of grouping. The resulting data can then be used in concentrating modules, such as multiplexers (e.g.: from multiple detection layers). Finally, the output can be used in higher level treatment processes like online event-builders/reconstruction modules. Pyrame includes in its stock tools a generic data plotter, an event display and an event builder.

All Pyrame modules run in a virtual machine [6]. In the new 3.0 version, a number of new features have been added to the application programming interface (API) provided to modules in order to manage and distribute online and offline data. Some new external functions common to all modules have also been added to interface data producers with data consumers through the Pyrame protocol.

Furthermore, Pyrame 3.0 brings important gains in performance across the board. Python modules perform Pyrame transactions twice as fast as before and the rest of the supported languages obtain a 25% gain. Now, all languages have approximately the same overhead at $30 \pm 5 \mu s$ per Pyrame transaction\(^1\).

2. Architecture

Figure 1 shows the essential building blocks of the data acquisition and treatment chain in Pyrame. The acquisition server and the decoders present in the converters lie on the real-time domain of the chain. These components are optimized for the maximum performance with big preallocated buffers, minimum number of system calls and the capability to freeze certain data processing activities in order to absorb big data bursts.

After data enters the acquisition chain, user-defined decapsulation plugins ("uncap") minimally analyze the data to allow the identification as data or control packets. Control

\(^1\) Performance measured on an HP EliteBook 8570w laptop with an quad-core Intel Core i7-3740QM mobile CPU at 2.7 GHz and 16 GB of DDR3L RAM at 1.6 GHz.
packets are left available for other modules to gather, while data packets continue the treatment chain with raw file storage, shared memory buffers and data output through network sockets. Data converters can be plugged to any of those three data treatment steps.

Data decoded by the user-defined input plugins at the converters, or by any other module, can be fed to a dispatcher. At this point the real-time constraint is abandoned and sub-sampling becomes possible. The data sources directory allows consumers to discover data producers and communicate with them. On the consumer side, a mechanism named "event-loops" allows the module to receive data from any data source.

3. Generic data format

We introduce an event-based data structure that provides a generic, standard, data transmission format between modules. The structure defines four types of entities: fields, domains, events and blocks.

- Fields constitute the basic data unit. Any piece of information, encoded as a string, can be a field (e.g.: integer, float, string, image).
- Domains contain an arbitrary number of fields grouped by a common semantic.
- Events contain three domains related to time, space and data.
- Blocks serve as containers of events. Blocks contain a properties domain with data common to all events.

| Domain: | field 1 | field 2 | ... |
|---------|---------|---------|-----|
| Event:  | time domain | space domain | data domain |
| Block:  | properties domain | event 1 | event 2 | ... |

3.1. Simple ASCII format

As an example of data format that can be used with Pyrame, we have defined a simple ASCII format (SAF). Data structure (domain, event, block) and data format (SAF) have well-encapsulated implementations, so that SAF can be easily replaced with other data encoding formats.

In SAF: fields are separated between them by commas and represented by strings; domains are separated between them by pipes (|); events start with an exclamation mark (!); blocks start with their properties domain and a concatenation of events. Multiple blocks are separated by newlines. The general form is:

\[
\text{prop1, prop2, \ldots | time1, time2, \ldots | space1, space2, \ldots | data1, data2, \ldots | time1, \ldots}
\]

3.2. Schema

In order to describe the data format to consumers and allow them to parse data properly, dispatchers produce a schema with the names of the fields on each domain. Dispatchers provide it to consumers on request. This allows the programmer to write format-independent code by using names to get event content.
Figure 2. Diagram of data generation, dispatcher, transmission to client and processing by event-loop

4. Dispatcher
The dispatcher is the subsystem that allows Pyrame modules to buffer data to be output at the consumer speed. The dispatcher stores data in a preallocated circular buffer of defined size. When in online mode, it provides the subsampling capacity required by slow consumers, by allowing the destruction of old data in the buffer when it is full (round-robin algorithm). In offline mode consumers register upon the dispatcher and the latter keeps track of the retrieved data. When all registered consumers have accessed a particular block, it is destroyed and its position in the circular buffer can be reused.

A data-ready multi-client TCP listening socket is opened by the dispatcher when initialized. Every time new data is available consumers are informed. This way, consumer polling is reduced to the minimum.

5. Data sources
Pyrame provides a central module to index the dispatchers (i.e.: data sources) present on the system. The module stores the name and connectivity characteristics of the data source so that any client can search them and connect to them.

The data transmission channels of a module are its Pyrame-protocol socket and its data-ready socket. The two TCP ports, the common IP address and the schema that identify and describe these channels are stored in the data source directory.

6. Event-loop
Pyrame modules can instantiate an event-loop thread by providing a data-source name, callbacks for several situations (new block, new event, end of current block and reinitialization) and block-retrieval policy. The policy determines which block is obtained next. If policy is “first”, the event-loop will ask the dispatcher for the next available block after the last already treated. This approach is exhaustive only if the consumer is faster then the producer. If policy is “last”, the event-loop will get the available block list from the dispatcher and get the most recent block. Real-time monitors will usually prefer “last” policy to show up-to-date information, while offline data processing will prefer “first” policy. However, other uses can be imagined depending on the processing speed of producer and consumer, as well as the characteristics of the data production (e.g.: continuous data or presence of bursts).

The event-loop can be started, stopped and rewound through its API. The module using it can also make it advance one block or event at a time. Application examples of these commands are data plotters, in which play/stop/rewind commands can be easily made available.
The data-ready socket mechanism allows Pyrame to put event-loops in sleep mode while waiting for data. In the case of slow producers, this mechanism avoids excessive polling.

7. Stock tools

7.1. Data plotter

The generic ROOT-based online data plotter can take multiple data sources as inputs and present their data in histograms, graphs or 3D spaces. The data viewer uses event-loops and exposes their commands (one step, run, stop, etc.) with external Pyrame functions. In run mode data is plotted as fast as possible from the source output.

To define the plots, the user can choose the data fields that will be used for each axis, along with the plot type and the graphical options (color, lines, etc.). Depending on the plot type, one to three data fields will be needed: one for histograms, two for graphs, three for 3D plots. Multiple plots of any of the three types can be presented on the canvas. Insertion of new plots and removal of existing ones is possible at any moment.

Filters can be added to each plot. When present, every new event is checked against the declared filter. A filter is defined with a field name, a logical operator and a value (e.g.: \texttt{time}<1000 \text{ or } \texttt{voltage}!=0). Fields used on filters can be different from those used for the axis.

7.2. Event builder

A generic event builder is provided in Pyrame. It allows to group events from multiple data sources in a parametrizable way. The grouping is done at two levels. First, the event builder talks to every data source to get their block number lists. As soon as every source has produced data blocks with the same id, the event builder starts gathering them. Then, from every loaded event, a particular field is extracted. This field is a parameter of the event builder and is called the \texttt{choice field}. Another parameter is the \texttt{tolerance}. Two or more events are grouped together when they have the same value for \texttt{choice field}, within \texttt{tolerance}. Finally, the event builder produces, on its own dispatcher, blocks containing these grouped events.

A typical application of the event builder is as a time multiplexer for a particle detector, for example on CALICE’s SiW-ECAL. In this case, every event is tagged by a timestamp, composed by a spill number and a bunch-crossing id (BCID). The blocks are dispatched with the spill number as block number. Thus, the event builder will group events with the same timestamp within the tolerance, allowing getting traces from the detector. The tolerance allows to handle small shifts between the clocks of the different detector parts.

As event building can be very time consuming, the Pyrame event builder has been designed to work in a distributed way. Several networked instances can share the workload, for example on the nodes of a cluster. They can communicate together to decide which instance takes every block number. This way, they can collaborate to aggregate the full stream in real time.

7.3. Event display

The Event Display polls data from the event-builder through an event-loop. It gets the 3D tracks and it represents them on a 3D plot. The tracks are plotted one at a time. Two functioning modes are available: in monitor mode, the event-loop is configured with “last” policy and data is plotted as soon as it arrives, without pause between tracks. In examination mode, the event-loop has “first” policy and the display is paused after each track. The user can then resume by pressing a key. This mode is useful to perform a careful examination of the acquired data.

8. Performance

Pyrame’s performant acquisition server allows to absorb large data sets even in bursts. The architecture described in this paper implements subsampling after the acquisition server
processing as a way of allowing online data treatment without worrying about interfering with proper raw data acquisition. In order to estimate the amount of data processing available to online modules, we have performed measurements\textsuperscript{2} of the speed at which the dispatcher and the event-loop generate and consume data blocks, respectively.

Figure 3 shows the data rate provided by the dispatcher and the event-loop as a function of the block size. The modulation of the latter is obtained by increasing the number of events in each block. For this test, we have used a dispatcher with a buffer depth of 20000 blocks and an event size of 40 B. First, the dispatcher fills its buffer with 20000 blocks of the same size. The time elapsed for this generation provides the speed of the dispatcher. Then, the event-loop consumes all the blocks and the speed is measured in the same way. This process is repeated for every block size.

The dispatcher shows little overhead per block at 8 \(\mu\)s for blocks of one event. This is the cost of allocating and building the necessary memory structures, adding them to the circular buffer and serializing the data. The speed of the dispatcher stabilizes rapidly at about 240 Mbit/s once the increase of time per block and the increase of size per block compensate.

Regarding the event-loop, the overhead is about 190 \(\mu\)s for blocks of one event. This is the cost of searching in the dispatcher buffer for the next block, transmitting the information and parsing it. The event-loop speed stabilizes at 170 Mbit/s for blocks larger than 40 kB, limited by the parsing engines.

9. Application example
Pyrame was originally designed to be the control and command software of the CALICE’s SiW-ECAL. As previously described\textsuperscript{[8][9]}, the calorimeter is composed of wafers of silicon, layered

\textsuperscript{2} Performance measured on an HP EliteBook 8570w laptop with a quad-core Intel Core i7-3740QM mobile CPU at 2.7 GHz and 16 GB of DDR3L RAM at 1.6 GHz. Shown data rate values exclude TCP, IP or Ethernet overheads.
**Figure 4.** Setup used for CALICE’s SiW-ECAL. The acquisition chain is fed with input from both ECAL and HCAL. Converters decode the data online only for ECAL. Once out of the real time domain, dispatchers and event-loops in the different modules exchange data. An event builder takes data from multiple converters to build high level events. Finally, the angle fitter reconstructs the track. Several data plotters (DP) show data along the path. The hit camera and the event display show the pixels that have been hit.

with tungsten plates. The timing requirements are adapted to future ILC specifications: a data acquisition window is opened at 5 Hz. During the acquisition time, the readout chips are self-triggered and the corresponding energies are stored. When the window is closed, the readout begins and raw data are sent to the acquisition server through an electronic chain.

Integrated in Pyrame’s acquisition chain, the data are stored into files but also distributed to online decoders through shared memories. These decoders (one per detection layer) extract physical events from raw data and publish them through a dispatcher. They also apply cuts, reducing the impact of some of the chips imperfections. After this step, the real-time constraint is broken and application programs can access data at their own pace, leading to subsampling in worst case.

Two monitoring applications are used directly on these dispatchers: the hitcam and the energy histograms. They allow to check the layers’ output, one at a time.

The hitcam is an application displaying a map of the chip self-triggering events on a 2D matrix that represents a detection layer. Color expresses the frequency of hits, allowing to easily spot noisy channels, badly calibrated or turned off chips. The hitcam is based on an event-loop whose new_event callback updates the colored matrix. The other first-level monitoring program is a histogram of energy. It is based on Pyrame’s generic data plotter. It allows to check the energy distribution to detect bad calibrations or overestimated cuts.

At this level, data is aggregated by layer and dispersed along the timeline. Pyrame’s event-builder recombines them by time, applies a new cut to reduce coherent noise pollution and dispatches the aggregates.

An Event Display polls the aggregated points and draws them on a 3D space. There, the user can take time to observe the tracks. To observe all the events, the event-loop can be configured in “first” mode and a key-press is required to go on. On the other hand, the display can be used as a monitor of the system’s status. In this case, the event-loop will be configured in “last” mode to show only up-to-date events and no pause will be applied.

In a silicon calorimeter, the deposited energy is very dependent on the angle of incidence of the particle. Thus it is of major concern to determine these angles with precision in order to correct the measured energy. Taking events directly from the event-builder dispatcher, an
angle fitter interpolates the tracks’ parameters. This fitter is a ROOT program based on an event-loop and on the TMinuit ROOT package (with the Migrad algorithm). Calculated angles are published again in a dispatcher and one can use Pyrame’s data plotter to build histograms of them. The histogram view allows the user to see the distribution of angles and check the systematics of the beam.

This acquisition chain and monitoring tools have been successfully used during numerous test-beams at CERN and DESY, providing efficient data checking and easing the fine-tuning of parameters. During the last one, at CERN in June 2016, one additional functionality was added. Because the test-beam was in common with CALICE’s SDHCAL, a data exchange was set to mix data from both detectors. Instead of transmitting raw data, which can be hard to decode, we chose to exchange only data from converters (after cut and geometrical reconstruction). In addition to the benefit of only transmitting clean data, the end of the real-time constraint at the converters guaranteed us that data sharing could not saturate the acquisition chain and cause raw data loss.

10. Conclusions
Pyrame 3.0 includes a number of new features for the support of online monitoring and data processing. The modular and distributed approach with sub-sampling capabilities at each step, provides a flexible and non-blocking path for data treatment, aggregation and visualization. These characteristics and the performance of the current implementation render Pyrame’s online monitoring subsystem a flexible solution fit for test benches and medium-sized experiments.

Acknowledgments
This project has received funding from the European Union’s Horizon 2020 Research and Innovation programme under Grant Agreement no. 654168.

References
[1] La Rocca P, Riggi F 2014 J. Phys.: Conf. Series 515 012012
[2] Ananya A et al 2014 J. Phys.: Conf. Series 513 012037
[3] Stephens Z D, Lee S Y, Faghri F, Campbell R H, Zhai C, Efron M J, Iyer R, Schatz M C, Sinha S, Robinson G E 2015 PLOS Biology 13 7 e1002195
[4] Bainbridge R, Baulieu G, Bel S, Cole J, Cripps N, Delaere C, Jesus A C A, Drouhin F, Fulcher J and Giassi A 2008 J. Phys.: Conf. Series 119 022008
[5] Von Haller B et al 2009 16th IEEE-NPSS Real Time Conf. (Beijing) 354-360
[6] Magniette F, Rubio-Roy M and Thiant F 2015 J. Phys.: Conf. Series 664 082028
[7] Pollacco E et al 2012 Physics Procedia 37 1799-1804
[8] Cornat R, Gastaldi F, Magniette F 2014 J. Instr. 9 C01030
[9] Gastaldi F, Cornat R, Magniette F, Boudry V 2015 Proc. Sci. 193