ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-THPDP017

A DATA ACQUISITION MIDDLE LAYER SERVER WITH PYTHON SUPPORT FOR LINAC OPERATION AND EXPERIMENTS MONITORING AND CONTROL

V. Rybnikov*, A. Sulc, DESY, Hamburg, Germany

Abstract

This paper presents online anomaly detection on low-level radio frequency (LLRF) cavities running on the FLASH/XFEL DAQ system. The code is run by a DAQ Middle Layer (ML) server, which has online access to all collected data. The ML server executes a Python 3 script that runs a pre-trained machine-learning model on every shot in the FLASH/XFEL machine. We discuss the challenges associated with real-time anomaly detection due to high data rates generated by RF cavities and introduce a DAQ system pipeline and algorithms used for online detection on arbitrary channels in our control system. The system's performance is evaluated using real data from operational RF cavities. We also focus on the DAQ monitor server's features and its implementation.

DAQ DATA ON-LINE ACCESS

The DOOCS [1] based FLASH DAQ [2] system (Fig. 1) has been in use since 2004. A similar XFEL DAQ [2] system has been running since 2019. The collected data is provided not only for the purpose of the LINACs operation but also for the experiments running on XFEL [3]/FLASH [4] photon beamlines. To be able to implement online (nearly real-time) control of the LINACs the data access is provided for Middle Layer (ML) servers running on the DAQ computers. The data access is done via Buffer Manager (BM) [5]. The BM provides ML servers with the synchronized data for every short in the LINAC.

A DAQ monitor server is a generic ML server that can be configured according to the user's requirements. To configure the server one needs two pieces of information:

- a set of DAQ channels which data is to access,
- a path to a Python script for the data processing.

All this information is provided via dedicated server's DOOCS properties accessible via the network.

DAQ MONITOR SERVER IMPLEMENTATION

The DAQ monitor server is implemented as a DOOCS server. It runs on a DAQ computer and has access (via BM) to all collected DAQ data. The server can execute a Python [6] script feeding it with the required DAQ channel data.

Server Design

The server is written in C++ [7]. It makes use of the Python/C API [8] (independent on the Python 3 version) to provide the data exchange between C++ and Python worlds.

Server Configuration

The server configuration with respect to channels to be read from the DAQ BM is implemented via setting the server's property 'DAQ.REQ'. It accepts XML [9] strings. The XML string consists of a number of sections corresponding to the number of DAQ channels to read. Every section contains at least the following information:

- the DAQ channel name,
- the sub-channel number,
- channel sender source (server block name)
- and an event type mask.

All this information is used to perform the correct channel subscription at the BM.

Any DOOCS server has a configuration file for storing its property values. In addition, every property of XML type creates a corresponding text file containing the XML string content. The DAQ monitor server uses this file to set the value of the 'DAQ.REQ' property during its start-up. It

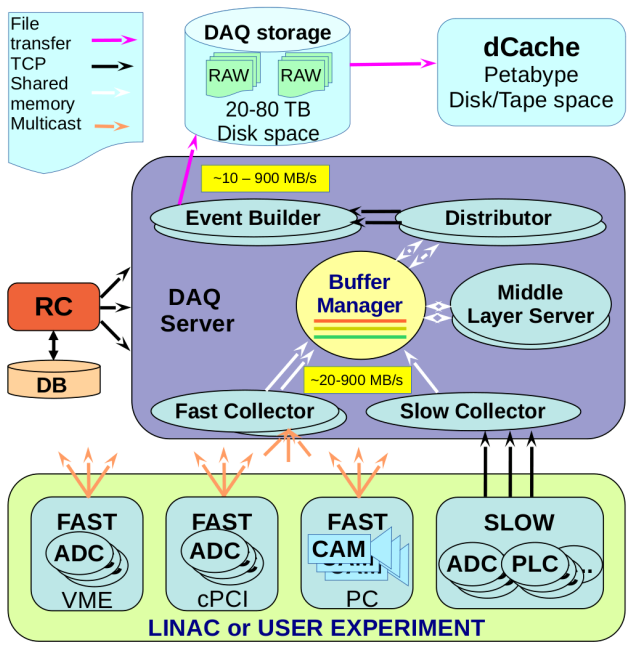

Figure 1: FLASH and XFEL DAQ architecture.

* vladimir.rybnikov@desy.de

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-THPDP017

Table 1: DOOCS Properties Controlling Python Script Execution

Name	Purpose
PYTHON.USE	If not 0, then 2 next properties are considered.
PYTHON.SCRIPT	The full path to a python.
2*PYTHON.FUNCTION	A function name in the Python script defined by PYTHON.SCRIPT to be called.

Table 2: Statistics DOOCS Properties for Python Script Execution

Name	Purpose
BM.EVN_TOTAL	Total number of data shorts seen by the server.
EVENTS.PROC	The number of data shorts processed by the scripts.
PYTHON.TIME	Python script execution time history.

allows it to be configured with the same set of DAQ channels after a new (re)start-up.

Server Initialization

On getting the XML configuration string the DAQ monitor server performs the following steps:

- parses the XML string and prepares a list of required channels
- creates new DOOCS properties dedicated to the required input channels. The types of properties can be of all DOOCS types (scalars, arrays, images)
- subscribes to the BM to get the channels
- goes to RUN state for receiving the DAQs data

IN OPERATION

The DAQ monitor server gets the DAQ channel data from the BM on every shot in the machine (currently 10Hz). The DOOCS properties dedicated to the input data channels are updated with the new values. The way the data will be handled further depends on the Python script. Its execution is controlled by the 3 server's properties shown in Table 1.

The function defined by PYTHON.FUNCTION is to be called by the DAQ monitor server if the Python script is defined by PYTHON.SCRIPT can be executed and PYTHON.USE is not zero.

The function is called with the parameters prepared by the server as a list of dictionaries containing the DAQ channel data.

The execution of the Python function is to be canceled for the next machine shot if the function call for the previous data is still not complete. It means that the time execution exceeds the time between two consecutive shots in the machine (in our case 100 ms). The statistics on processed and lost events as well as time processing are given by the DAQ monitor properties in Table 2.

The results from the Python script are returned as a list of dictionaries. The dictionaries can be divided into 2 groups: results and parameters.

Results from the Python Script

The results have to have at least the following keys: `{'OUT':'SOMENAME output description', 'TYPE':'type', 'DATA': data}`. The keyword `OUT` is a command to the DAQ monitor server to create a DOOCS property named `SOMENAME` with the type of `type` and with the value of `data`. There can be some additional keys for some complex data types (e.g. images). All supported data types are listed in the Table 3. Once a new output property is created it will update with every output value from the Python script if its name is in the output list of dictionaries.

For simple data types, the properties containing their histories are also created. It allows us to see an output value change with the time.

Table 3: Optional DOOCS Properties Attributes

Type	Data Type
Single	BOOL, SHORT, INTEGER, FLOAT, LONG, DOUBLE and TEXT
Array	SHORT, INTEGER, FLOAT, LONG, DOUBLE
Complex	SPECTRUM, IMAGE

Parameters from and to the Python Script

The parameters have the following format: `{'PARAM': 'SOMENAME parameter description', 'TYPE':'type', 'DATA': data }`. The keyword `PARAM` is a command to the DAQ monitor server to create a DOOCS property named `SOMENAME` with the type of `type` and the value `data`. Every "parameter" DOOCS property can be changed by operators. They are passed to the Python script along with input DAQ channel data. In this way, one can control and tune the Python script algorithm.

For both outputs and parameters, returning an existing output or parameter without `DATA` field leads to the removal of the corresponding DOOCS property.

The type of outputs and parameters can be changed on the fly just by using a different type in the 'TYPE' field.

Monitoring and Controlling the Server

The DAQ monitor server can be controlled and monitored by any DOOCS client. A user interface based on Java Doocs Data Display JDDD [10] (see. Fig. 2) is used as the main tool. First of all the panel allows the user to monitor the signals on inputs, outputs, and set parameters (in the tab 'Channels', not shown in the Figure).

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-THPDP017

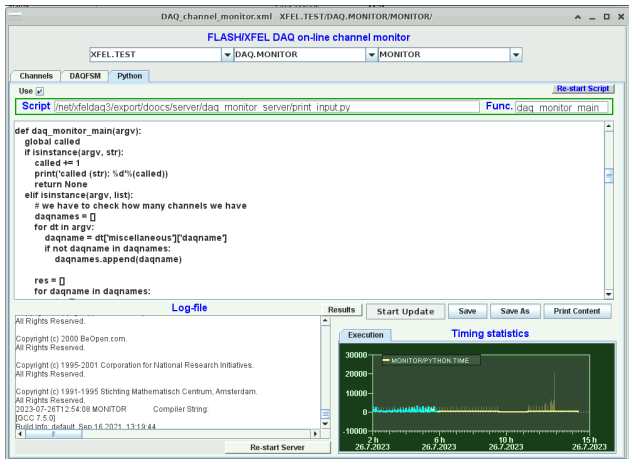

Figure 2: JDDD DAQ monitor panel.

The tab DAQFSM in Fig. 2 contains the information on the server status and statistics regarding the DAQ (e.g. data processed, data skipped, history of processing time, etc.).

The tab shown in Fig. 2 provides the main parameters regarding the Python script to run. The panel allows editing the text on the fly and executing it after the text has been changed. It reduces the time of debugging and tuning the script algorithms.

USE CASE: ON-LINE MONITORING OF LLRF CAVITIES AT EUROPEAN XFEL

In Ref. [11] we propose a neural network approach for detecting anomalies and predicting failures in superconducting radio frequency (SRF) cavities at the XFEL.

One of the most common issues in SRF cavities is quench which entails abrupt transitions from the superconducting state to the normal resistive state. These transitions typically occur due to localized disturbances that elevate the surface temperature beyond the critical threshold and lead to an increase in resistance and loss of superconductivity of a cavity.

Detecting such disturbances like quenches has an essential importance for the operability of European XFEL. Currently, the European XFEL employs an online quench detection system, relying on the measurement of the loaded quality factor (Q) during the RF pulse's decay phase [12]. Nevertheless, substantial efforts have been undertaken to enhance this approach, aiming to reduce the incidence of false positives in the detection process.

Model

This method employs recurrent neural networks to analyze cavity operational time series data. The raw input comprises a sequence of 1820 waveform values (in frequency-amplitude pairs over microseconds) with 6 channels - probe, forward, and reflected (each in frequency-amplitude pair). Thus, before any pre-processing, each raw pulse consists of an array of 10920 float32 values received from DAQ.

As a preprocessing step, the channels are subsampled to retain every 10th value, reducing the number of elements in each pulse per channel to 182. The frequency-amplitude pairs are then transformed into in-phase and quadrature components to handle abrupt frequency jumps arising from the angular coordinate representation of frequency.

The resulting 6 linearized 182-value channel vectors (totaling 1092 values) are fed into a single-layer LSTM recurrent network to capture temporal dynamics. This architecture is slightly simplified from Ref. [11] to minimize computational overhead. The LSTM layer outputs a 64-dimensional hidden representation per pulse. A final linear layer then projects this to a 64-dimensional anomaly score vector used to assess the pulse's fault status by calculating its L2 distance from a hypersphere center

To handle the significant class imbalance between limited faulty data and abundant healthy data, same as in Ref. [11] we employ a semi-supervised deep anomaly detection loss function [13].

The new model is re-trained due to a slightly simplified architecture to improve run-time and to adapt to domain drift on the accelerator over time.

Training Data

The training data utilized in this work consisted of known fault datasets captured by operators, often 5-second snapshots taken just before system shutdown, as well as short 5-second healthy snapshots captured during June, July, and August 2023. This recent data was used to adapt the model to current operating conditions. The healthy snapshots were captured daily every 4 hours.

Since not every faulty snapshot taken by the operators necessarily contained useful faulty signals, and up-to-date annotated labels were not available unlike in Ref. [11], we first utilized the approach from Ref. [12] to train a model to detect anomalous cases and filter out probable uninformative faults. This allowed us to isolate the informative (like quenches) faulty examples.

With the filtered subset of informative faults and known healthy examples, we then retrained the model using the semi-supervised loss function from Ref. [13], treating the identified faults and healthy cases as labeled examples. This follows a similar semi-supervised approach as shown in Ref. [11]. By using the anomaly detection model to filter the operator-captured faults, we obtained a cleaner set of useful faulty data despite lacking complete up-to-date labels. The combination of recent healthy snapshots and informative faults enabled the adaptation of the model to current operating conditions.

Implementation Details

To minimize computational time, we took advantage of a useful feature of PyTorch where the hidden states of the LSTM layer can be returned and passed to future events. This allows processing each pulse individually as they are received, rather than passing the entire sequence at once,

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-THPDP017

while still obtaining the same output as if the full sequence was passed.

Specifically, we maintained a dictionary containing the name of each cavity and its last hidden state. When a new event arrived, we passed the corresponding cavity's previous hidden state to the LSTM layer as an argument. The LSTM then outputs a new hidden state, which we use to update the value for that cavity in the dictionary. By storing the hidden state in this manner and passing it sequentially, we could process each pulse event individually in a streamlined fashion, while retaining the full context and getting the same result as batch processing the entire sequence. This approach minimized computational time by avoiding reprocessing previous pulses.

Furthermore, if the algorithm is run in parallel with the DAQ, there is a buffer that records the incoming if the algorithm pipeline is slower than the rate of the data flow.

Before the raw DAQ data are passed to the model, data must be transformed into a format suitable format. The preparation consists of data subsampling of the input waveforms, transformation into IQ coordinates, and normalization of each waveform into range $(-1, 1)$.

Computational Setup

We conducted the experiment on one of our DAQ machines regularly used during the operation. The machine is equipped with 32 CPUs Intel(R) Xeon(R) E5-2650 with a maximum frequency of 3.6GHz. The machine has 256 GB RAM.

Time Performance

The run time consists of two components. One is data preparation, where input data are transformed into a suitable format for the algorithm. The other part is the scoring algorithm as it is described in Ref. [11].

When 32 cavities at A25 are evaluated online, the data preparation takes 1141 µs and 6113 µs for anomaly detection. Note that the run-time of the anomaly detection can be significantly improved on GPU.

Example Result: Quench on 2023-08-05 at 22:56

In the A5 L2 European XFEL LLRF section, a quench took place. In Fig. 3 we show the last 200 waveforms before the cavity was switched off. This incident was promptly logged by our operators, and an offline snapshot was captured for subsequent post-mortem analysis, following the same format as our online algorithms.

Figure 4 shows a notable rise in the anomaly score for cavity `C5.M1.A5.L2` at 22:55:39. Approximately 75 pulses before the station is shut down compared to baseline (score averaged over last 30 minutes) grows approximately 1/5 which indicates a likely anomaly.

Figure 3 shows full-resolution amplitude waveforms for the 200 pulses preceding the shutdown. These waveforms appear visually free of anomalies.

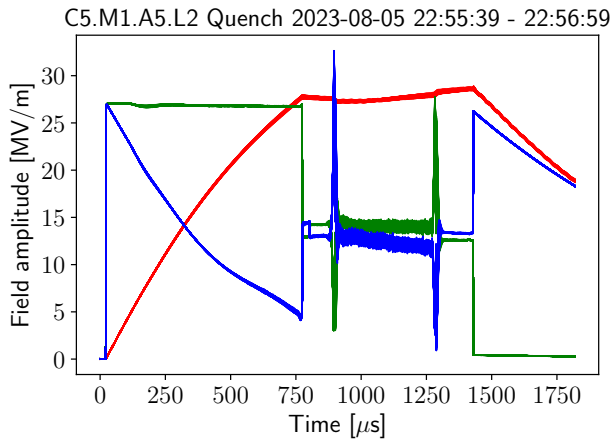

Figure 3: It shows 200 amplitude pulses (time range 22:56:39-22:56:59) recorded from station `C5.M1.A5.L2` before the station stops recording any signals. The red curve represents the probe, and the green and blue curves represent the forward and reflected amplitudes, respectively.

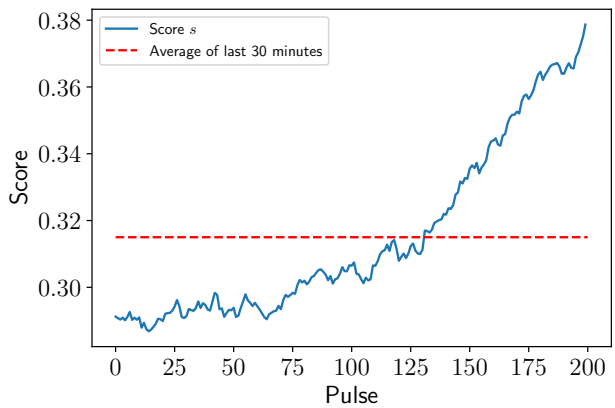

Figure 4: Shows the anomaly scores generated by the anomaly detection model [11] on `C5.M1.A5.L2` for the 200 pulses prior to the modules being deactivated.

On-line Scoring

The previously discussed offline example does not reflect the online evaluation of the entire pipeline.

Apart from the aforementioned capability of making offline snapshots, the most important feature is to be capable of running an arbitrary algorithm in an on-line regime with minimum delay and recording the results back in DAQ. In order to prove the capacity of the approach, we also ran the algorithm on 32 cavities and recorded their runtime to showcase that any algorithm programmed in Python can be deployed with a minimum number of modifications.

CONCLUSION

In this work, we show a DAQ monitor server framework for FLASH/XFEL DAQ systems that has proved to be a convenient tool for building up ML servers for controlling, monitoring, and feedback purposes.

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-THPDP017

To illustrate our capabilities, we utilized one of our experimental machine learning models [11] as a demonstration of our ability to conduct computationally demanding calculations effectively, particularly in anomaly detection.

We presented an instance where a station experienced a quench and, using an offline snapshot designed to mirror offline operations. These experiments serve as a showcase of our proficiency in deploying Python scripts and running them in an online environment.

ACKNOWLEDGEMENT

We acknowledge DESY (Hamburg, Germany), a member of the Helmholtz Association HGF, for its support in providing resources and infrastructure. Furthermore, we would like to thank all colleagues of the MCS group for their contributions to this work and help in preparing this paper.

REFERENCES

[1] K. Rehlich, "Status Of The Ttf VUV-FEL Control System", in *PCaPAC'05*, Hayama, Japan, Mar. 2005, paper TUA1.

[2] A. Agababyan *et al.*, "Multi-processor based fast data acquisition for a free electron laser and experiments", in *15th IEEE-NPSS Real-Time Conference*, Batavia, IL, USA, 2007, pp. 1–5. doi:10.1109/RTC.2007.4382853

[3] *European XFEL - The European X-Ray Free-Electron Laser Facility*. DESY - Deutsches Elektronen-Synchrotron, 2023. http://xfel.desy.de

[4] "FLASH - Free-Electron Laser in Hamburg", *DESY - Deutsches Elektronen-Synchrotron*, 2023, http://flash.desy.de.

[5] V. Rybnikov *et al.*, "Buffer Manager Implementation for the FLASH Data Aquisition System", in *Proc. PCaPAC'08*, Ljbuljana, Slovenia, Oct. 2008, paper TUP010, pp. 102–104.

[6] M. Lutz, *Learning Python 5th Edition*, O'Reilly Media, Inc., Sebastopol, CA, USA, 2013.

[7] B. Stroustrup, *The C++ Programming Language*, 4th ed., Addison-Wesley Professional, USA, 2013.

[8] *Python Software Foundation*, "Python C-API Documentation", 2023, https://docs.python.org/3.9/c-api/intro.html

[9] E. T. Ray, *Learning XML*, O'Reilly and Associates, Cambridge, MA, USA 2003.

[10] E. Sombrowski, A. Petrosyan, K. Rehlich, and W. Schutte, "jddd: A Tool for Operators and Experts to Design Control System Panels", in *Proc. ICALEPCS'13*, San Francisco, CA, USA, Oct. 2013, paper TUMIB09, pp. 544–546.

[11] A. Sulc, A. Eichler, and T. Wilsken, "A data-driven anomaly detection on SRF cavities at the European XFEL", *J. Phys.: Conf. Ser.*, vol. 2420, no. 1, pp. 012070, 2023.

[12] J. Branlard, V. Ayvazyan, O. Hensler, H. Schlarb, Ch. Schmidt, and W. Cichalewski, "Superconducting Cavity Quench Detection and Prevention for the European XFEL", in *Proc. ICALEPCS'13*, San Francisco, CA, USA, Oct. 2013, paper THPPC072, pp. 1239–1241.

[13] L. Ruff *et al.*, "Deep semi-supervised anomaly detection", *arXiv*, 2019. doi:10.48550/arXiv.1906.02694