Python based integration of GEM detector electronics with JET data acquisition system

Wojciech M. Zabołotny[a], Adrian Byszuk[a], Maryna Chernyshova[b], Radosław Cieszewski[a], Tomasz Czarski[b], Simon Dalley[e], Colin Hogben[e], Katarzyna L. Jakubowska[b], Grzegorz Kasprowicz[a], Krzysztof Poźniak[a], Jacek Rzadkiewicz[b,c], Marek Scholz[d], Amy Shumack[e]

[a]Institute of Electronic Systems, Warsaw University of Technology, ul. Nowowiejska 15/19, 00-665 Warszawa, Poland;
[b]Institute of Plasma Physics and Laser Microfusion, ul. Hery 23, 01-497 Warszawa, Poland;
[c] National Centre for Nuclear Research, ul. Andrzeja Sołtana 7, 05-400 Otwock, Świerk, Poland;
[d] Institute of Nuclear Physics, Polish Academy of Sciences, ul. Radzikowskiego 152, 31-342 Kraków, Poland;
[e] Euratom/CCFE Fusion Association, Culham Science Centre, OX14 3DB Abingdon, United Kingdom;

ABSTRACT

This paper presents the system integrating the dedicated measurement and control electronic systems for Gas Electron Multiplier[1] (GEM) detectors with the Control and Data Acquisition system (CODAS) in the JET facility in Culham, England. The presented system performs the high level procedures necessary to calibrate the GEM detector and to protect it against possible malfunctions or dangerous changes in operating conditions. The system also allows control of the GEM detectors from CODAS, setting of their parameters, checking their state, starting the plasma measurement and to reading the results. The system has been implemented using the Python language, using the advanced libraries for implementation of network communication protocols, for object based hardware management and for data processing.

Keywords: Data acquisitions systems, Experiment control systems, Python, GEM detectors, KX1 diagnostic, JET, CODAS

1. INTRODUCTION

The KX1 soft X-ray diagnostic on the JET tokamak in Culham[2] is focused chiefly on measurement of plasma impurities. The upgrade of this diagnostic is based on GEM detectors,[3,4] connected to dedicated readout electronics.[5,6] Control and data processing functions are implemented in a FPGA firmware,[7,8] and in an embedded PC. The structure of the system is shown in Figure 1. An embedded device server[9] provides software controlling the operation of the readout system and surrounding systems like low voltage (LV) and high voltage (HV) power supply units for each GEM detector and analog front-end.

The embedded device server allows detector to be used autonomously, e.g. when controlled by an expert from the Matlab console. However to integrate the detector with the JET data acquisition system (CODAS - the COntrol and Data Acquisition System),[10–13] an additional layer - the so called device server is necessary.

The device server should provide an interface between CODAS and the embedded server, and should preserve information about the state of the KX1 diagnostic if the embedded server must be restarted. Additionally the device server allows separation of the private TCP/IP network, used to connect components of the KX1 diagnostic, from the internal CODAS network.

Further author information: (Send correspondence to W.M.Z.)
W.M.Z.: E-mail: wzab@ise.pw.edu.pl, Telephone: +48 22 234 7717

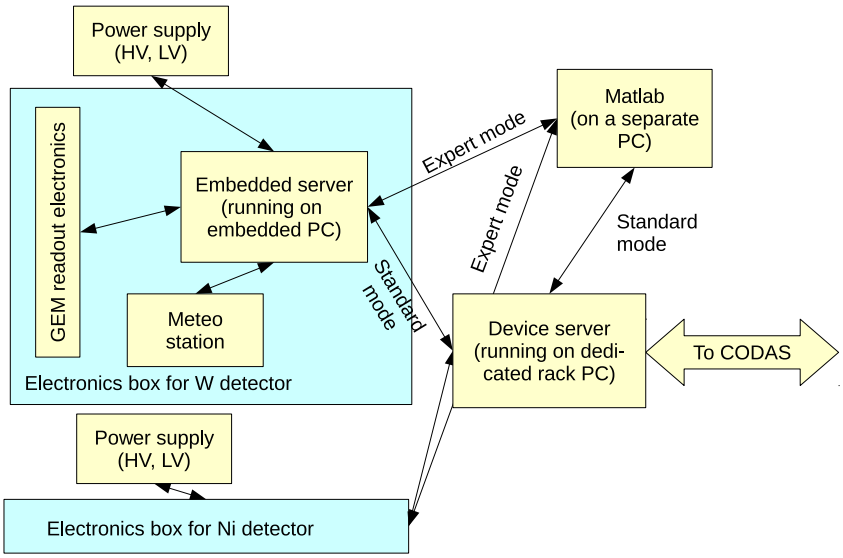

Figure 1. Embedded and device servers in the KX1 diagnostic.

2. REQUIREMENTS FOR THE DEVICE SERVER

The device server must perform multiple operations in parallel:

- Periodic control of the detector parameters (monitoring thread)
- Implementation of an event based HTTP protocol for JET devices.[14]
- Implementation of data acquisition from the GEM readout electronics
- Continuous control of the safe operation of the detector

The above tasks must be performed in parallel, however as some of them use the same hardware resources, reliable synchronization mechanisms must be provided to avoid misconfiguration of hardware, which can result in corruption of data or even in damage of hardware.

The device server handles two detectors (“Ni” – nickel monitoring detector, and “W” – tungsten monitoring detector), therefore it should be running on hardware which is independent of both those detectors. Additionally it must be able to process huge data sets in the RAM memory, and to store significant amount of data on the local hard disk (e.g. the acquired data from the GEM detectors). Therefore a separate rack mounted PC computer, running under control of the Debian Linux OS was chosen as a hardware platform.

It is important, that data acquisition at JET is not performed continuously. The plasma pulses are separated by a significant amount of time, which may be used for reading and archiving of data. This relaxes requirements related to real-time data processing, and allows use of solutions based on programming languages which are slower than C or C++, but easier in development and maintenance.

This was especially important, as the system was developed as a unique device. Different requirements and constraints were discovered during the process of development and testing, and therefore a flexible language allowing for fast prototyping and testing was necessary.

The data acquisition and processing algorithms were developed in Matlab, but as Matlab is not supported in JET it was not possible to use it in the data acquisition and processing chain (even though it could be used as a system configuration and calibration tool). Therefore another language, comparable with Matlab was necessary, which could be used both in the embedded PC and in the rack PC.

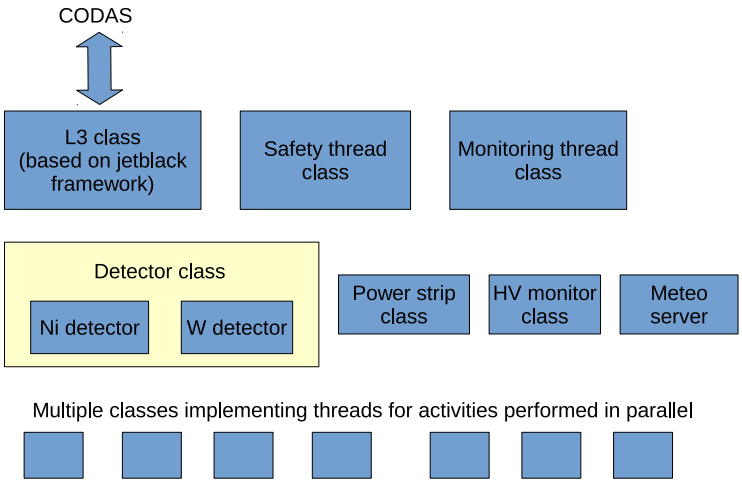

Figure 2. Classes used in the device server.

Python has been chosen as a language fulfilling the above requirements and offering following features:

- allows implementation of multithreaded programs, using the *threading*[15] module. Multiple threads are tightly coupled, as they can share variables, and it is easy to ensure appropriate synchronization between threads.
- offers reasonable performance, as the modules (libraries) are compiled to the bytecode at first use.
- offers multiple extension libraries. These allow implementation of e.g. a TCP/IP server in just a few lines of source code. Python also offers multiple modules for scientific computing like optimization, linear algebra, integration, interpolation, special functions, FFT, signal and image processing, etc.
- is an object oriented language, and offers easy handling of complex data structures
- offers fully automated memory management
- offers advanced, object based exception handling
- offers the possibility to implement time critical functions in C or C++

A more detailed description of how the above features have been used in implementation of the device server will be presented in the following sections. The general structure of the device server implemented as a set of Python classes is shown in Figure 2.

3. MAIN BLOCKS AND THREADS IN THE DEVICE SERVER

As described before, the device server performs multiple tasks in parallel. Some of those activities were separated into dedicated threads, and some of them have been implemented in an event driven software framework. The Python language allows efficient use of both of those approaches.

3.1 The monitoring thread

The parameters of the GEM detector vary in time, due to their sensitivity to such parameters as temperature, pressure and humidity. Therefore the detector should be continuously calibrated. This calibration is performed by the monitoring thread, which periodically measures the energy spectra of X-ray radiation emitted by a ^{55}Fe iron source installed in the diagnostic. The acquired calibration data are further processed by the software calculating the current gain of the detector (written in Python, as described in Section 4.1, or in Matlab), which is included in the configuration data of the system, and used for data analysis. The monitoring thread uses the same hardware as measurement of plasma pulse radiation. Therefore it had to be implemented as a thread with lower priority, which is described in more detail in Section 5.1.

3.2 Detector Protection System

To decrease risk of detector damage due to operation in incorrect operating conditions, the system has been equipped with a Detector Protection System (DPS), shown in Figure 3. The task of the DPS is to shut down the KX1 electronics in a potentially dangerous condition:

- Temperature outside the defined safe operation range
- Humidity outside the defined safe operation range. Especially humidity which is too high causes risk of condensation in the electronics system, which may lead to damage.
- A high voltage value outside the defined safe operation range (a value which is too high may cause the detector breakdown, while a value which is too low may be a symptom of power supply overload)
- Lack of a gas flow in the detector gas system

The DPS runs as a separate thread, periodically reading sensors measuring the above operating conditions. For each checked parameter it is possible to define a safe operating range, a warning range, and an alarm range. When the parameter enters the warning range, only a warning message is issued to the operator and to logs. When the parameter enters the alarm range, an error message is issued to the operator and to logs, and then the affected detector is shut down.

An essential hardware component of the Detector Protection System is the "Power Strip".[16] It is a device allowing control of up to six mains power sockets via a TCP/IP network, e.g. using a WWW interface or via a SNMP protocol.

The shutdown is performed on two levels. First the embedded systems are requested to switch off all subsystems in a controlled manner, and after that the "Power Strip" is requested to switch off the mains power for the particular detector electronics box and its HV/LV power supply.

It is possible that some sensors may not produce valid information e.g. due to malfunction. In such a situation the authorized operator should be able to decide, that the detector may be used without information about the particular parameter. To ensure that, the detector protection system allows the definition of a list of masked sensors.

Implementation of the DPS must take into consideration, that just after switching on, the electronics boxes perform lengthy initialization (booting of the system, loading of FPGA firmware etc.), and do not provide access to sensors. However if such situation (no access to sensors) occurs during normal operation, it may be a symptom of loss of control and as a potentially dangerous event it should lead to shutdown. To allow proper initialization of the system after power-up, and to correctly handle a situation, where connectivity is lost during the operation, the DPS implements a five minute long "grace period" after the start of the device server.

A special situation occurs, when the system is malfunctioning, but must be switched on under special supervision to investigate the cause of the problems. This may be achieved by temporarily masking of the sensors causing system shutdown, or by total disabling of the DPS by not starting the device server. In the latter case the embedded servers may still be accessed in the expert mode from Matlab.

The DPS relies on correct operation of the device server running on the rack PC. However it is also important to protect the detector against the situation when either the rack PC is malfunctioning or the network connection between the electronic boxes and the rack PC is broken. The described malfunction is handled by an IP watchdog function in the "Power Strip". It periodically sends the ICMP ping packets to the rack PC, and if it does not receive any response for three minutes,

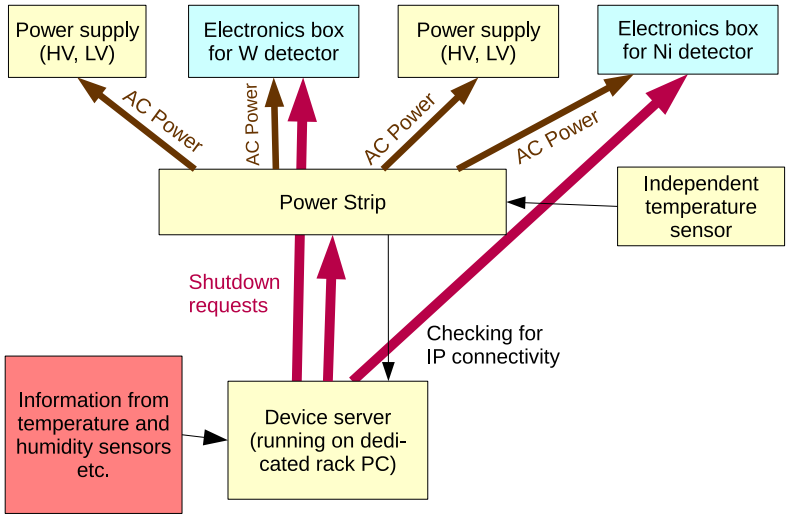

Figure 3. Structure of the detector protection system.

it switches off the power to the appropriate mains socket. (In fact, as this function of the "Power Strip" was designed to restart the server, it attempts to power-cycle the connected device three times, before switching it off permanently.)

Some external monitoring devices required special handling. E.g. one third party weather station sometimes refused TCP/IP connection, and it was necessary to query it for current values of environmental parameters a few times, before considering it to be unavailable. Handling of such "difficult hardware" was significantly simplified with the use of Python's object features. For those sensors special server classes were created, which separated updating of the current values from reading of the last received values. The thread which must rely on current values (like DPS) could use the "update" function which attempts to obtain the current values, which may take some significant time. This function also stores the last received values. If it is not possible to receive the current values within a predefined time, the last received values are marked as unavailable. Another thread which should not execute such time consuming actions (e.g. the thread responsible for data acquisition after the plasma pulse), may quickly read the last received parameters using the "read" function.

3.3 CODAS communication framework

Communication with the CODAS system is based on the HTTP protocol for interfacing to JET plant equipment.[14] One of the implementations of this protocol is the "jetblack" framework, written by Collin Hogben. This part of the system has been implemented with an event driven programming paradigm.

The implementation in the device server is based on the "jetpc" sources provided by the creators of the "jetblack" framework. The L3 class implements methods used to handle requests received from CODAS via HTTP protocol.

Dedicated methods of this class implement setting of *setup parameters*, used to control the KX1 diagnostic, and reading of *state variables*, describing the current state of the system. Setting of parameters is usually done by the operator, via a dedicated "CODAS Level 1 interface". The set parameters' values are also stored in the file on the local disk, allowing settings to be recovered after restart of the device server.

Other requests sent by CODAS change the state of the device server, e.g. requesting it to prepare to measure the plasma pulse, or requesting it to read the measured data from the embedded servers and to store them to local files.

Some requests sent by CODAS require more time and must be handled in a procedural way. In this case a dedicated thread is started to perform such an action, and CODAS later queries the device server for information on whether such a thread has completed its activity, and what the status of the operation is.

It is important, that certain actions requested by CODAS have high priority, e.g. preparation of the system for measurement of the plasma pulse must be performed immediately, or the data may be lost. If hardware resources needed for measurement are used by another thread, e.g. the one performing the monitoring measurement, its activity must be aborted to free those resources. This requirement creates some problems, which are described in more detail way in Section 5.1.

The CODAS communication framework also allows predefined actions to be triggered in the device server:

- Switching on and off of the "electronics boxes" containing the embedded server, and the FPGA based systems
- Switching on and off of the low voltage power supply for the analog front-end
- Switching on and off of the high voltage for the GEM detector (the HV values should be set in advance, using the setup parameters)

During the development of the device server, it was often necessary to redefine the format of binary data. This problem could be easily solved by defining dedicated "DataWriter" classes, which converted the data from the internal representation to the required external representation and wrote them to an output node.

3.4 Data Acquisition thread

The measurement and data acquisition is a relatively long process, as it may involve setting of a new configuration of the data acquisition electronics, waiting for the trigger (in the case of a plasma pulse measurement), and reading of the acquired data from the FPGA. Therefore the data acquisition has been implemented in a separate thread, which can be started either by the monitoring thread (and is then executed with low priority to acquire the calibration data from the ^{55}Fe source), or by the CODAS communication framework (and is then executed with high priority to measure the radiation originating from the plasma).

The data acquisition is performed in tight cooperation with the embedded server. Commands, responses to the commands, and data are transmitted between the device server and the embedded server using a specially developed protocol based on the msgpack[17] library.

3.5 Configuration server

An important functionality of the device server is handling of configuration data. These data describe configuration of the embedded server, of the FPGA based electronics and of the analog front end, and they shape the metrological characteristics of the whole system. The system configuration is stored in a textual, human readable format, compatible with the Python "ConfigParser"[18] module.

The configuration data are prepared in Matlab, and three different presets may be defined, which can later be selected using the "setup parameters".

Certain parts of the configuration data may be redefined dynamically during the operation of the system, due to the operation of the "automatic gain control" (AGC) process (described in the section 4.1).

The system may receive the new configuration data at any time, but it should not disturb an ongoing measurement. Therefore the configuration server keeps the previous configuration until the new one is fully received and verified. When a measurement is started (either by a monitoring thread or by a CODAS request), a snapshot of the current configuration is taken and is used until the measurement is completed.

Due to the fact that hardware reconfiguration is time consuming, the configuration server detects whether the new configuration differs from the previously used one, and reconfigures the hardware only if necessary.

4. ADDITIONAL PROCESSES RUNNING OUTSIDE OF THE DEVICE SERVER

The multithreading approach implemented in the *threading*[15] module works correctly for threads which spend a significant amount of time waiting for completion of an I/O operation. However, in case of tasks involving complex data processing, this approach is not efficient, due to the use of the so called Python "Global Interpreter Lock".[19]

Therefore such tasks have been implemented as separate processes. An additional advantage of such an approach is that those computational tasks do not affect the stability of the device server (e.g. in the case that erroneous data cause the crash of the processing task, and the task must be restarted).

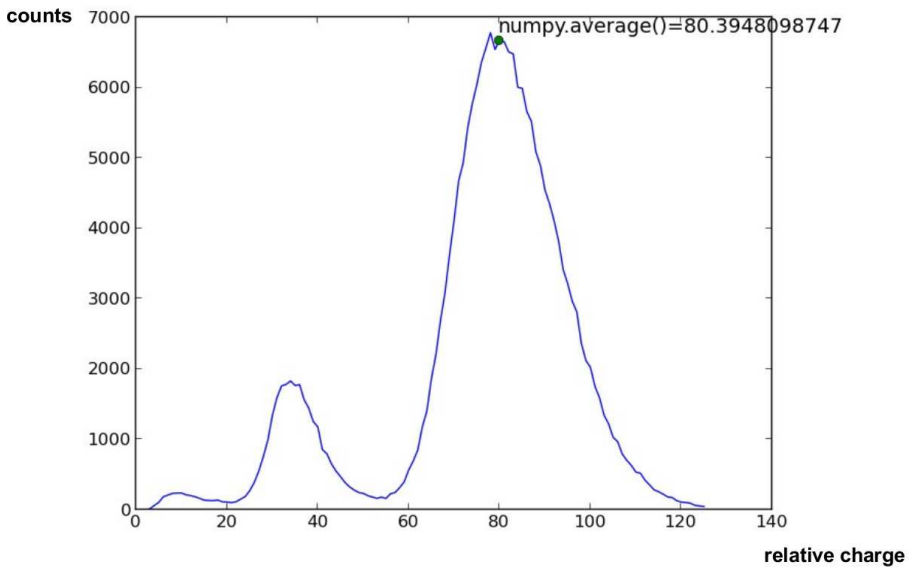

Figure 4. Energy spectrum of the ^{55}Fe source radiation (plotted here as charge collected by the detector, which is proportional to X-ray energy). Known position of the highest peak (at 5.9 keV) may be used to calculate current gain of the detector.

4.1 Automatic gain control process

The first of those external computational tasks is the Automatic Gain Control (AGC) process, which uses the calibration data obtained from the measurement of the radiation of the ^{55}Fe source. Thanks to the known energy spectrum of this radiation (see Figure 4), it is possible to calculate the current gain of the detector, and to include the calculated value in the system configuration data, so that variations of the detector's gain may be compensated when processing the data acquired from the plasma pulse.

The AGC process uses the Python NumPy module,[20] to calculate the weighted average of charge corresponding to the energy of the peak.

4.2 Process for separation of orders of reflection

In the Bragg spectrometer the relationship between the energy and angle of reflection is not univocal. The reflection angle Θ fulfills Bragg's law: $n\lambda = 2dsin\Theta$, where λ is the wavelength, d is the distance between the atomic layers in the crystal and the integer n is the order of reflection.

So the same angle of reflection may correspond to different wavelengths, and hence different energies, depending on the "order of reflection". An important characteristics of the upgraded KX1 diagnostic is the use of the GEM detector in proportional mode, which allows measurement of a photon's energy based not only on the angle of reflection but also on the charge produced by that photon in the GEM detector. However, due to the fact that the gain of the detector may vary even during a plasma pulse, it is difficult to set fixed borders between different orders of reflection. Therefore the order separation thread analyses the histograms recorded for different strips (i.e. for different reflection angles) and attempts to find the local minima, which are supposed to best separate hits corresponding to different orders of reflection (see Figure 5).

This code for order separation uses the algorithm for finding of local minimum, implemented in the Python NumPy module.

5. ADVANTAGES AND DISADVANTAGES OF PYTHON BASED IMPLEMENTATION

Implementation of the system using the Python language allowed use of multiple mature and efficient solutions for network communication, especially for implementation of the TCP/IP servers and their clients, for handling of configuration data and for measurement data processing.

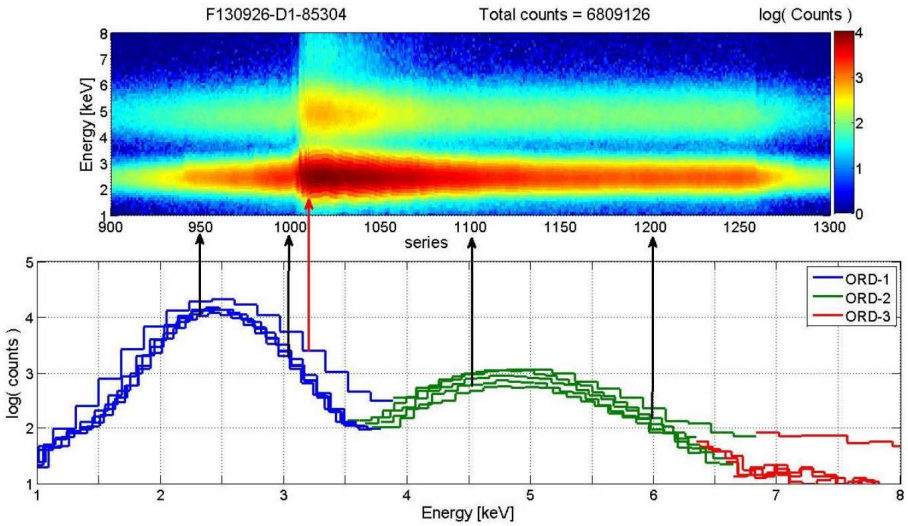

Figure 5. Estimation of borders separating different orders of reflection on the energy (charge) axis. It can be seen that the large increase in rate of hits, changes the detector's gain and causes a shift of the separation borders.

The object features of Python allowed creation of an object based layer for hardware management, which simplified development of the device server, which handles two similar, but not identical detectors in parallel. This approach was also very fruitful in implementation of the detector protection system, where different sensors could be handled in a similar way due to the object based abstraction layer.

The implementation of the device server also intensively uses the Python advanced exception handling. Exceptions are used both to signal problems and cleanly abort failed operations. They were also used to implement aborting of lower priority threads, which is described in the next subsection.

The device server uses multiple threads based on multithreading support provided by Python. As most threads spend significant time waiting for a command received via the network, or for completion of an I/O operation, the model of parallel execution provided by *threading*[15] module is suitable, however it lacks one significant functionality, described in the next subsection.

5.1 Implementation of prioritized threads in Python

As described in previous sections, operation of the device server sometimes requires that lower priority activity should be aborted, to allow execution of other, higher priority task. Different activities are performed by different threads, so the natural solution would be to kill the lower priority thread in such a situation, or better, to raise an exception in this thread (which could allow aborting its activity cleanly by the exception handler). Unfortunately the *threading*[15] module used in the device server does not allow one thread to raise an exception in another thread. An unofficial extension[21] exists that implements this feature, but it is not supported and does not work reliably in all versions of Python. Another possibility could be to use the *multiprocessing*[22] module, which implements parallel processing based on multiple processes. This implementation allows the "terminate" function to be called on a child thread, allowing it to be aborted, but it is impossible to implement the handler, allowing the aborted thread to terminate cleanly. The *multiprocessing* module makes communication between threads significantly more difficult, as processes do not share variables.

Therefore in the final version of the device server a model of cooperative thread termination has been implemented. In the object describing the state of the particular detector there is a special variable *AbortLevel* containing the priority of the lowest priority thread which is allowed to run. Each thread, when performing any time consuming operation, is obliged to periodically call the *CheckAbortRequest* function. This function checks if the priority of the current thread is lower than *AbortLevel*, and in such case raises the corresponding exception.

With such an approach it is possible to implement clean termination of lower priority threads, however to achieve low latency of termination, it is necessary to carefully implement threads so that the time period between consecutive calls to *CheckAbortRequest* is not too long.

5.2 Problems related to the interpreted character of Python

Python is an interpreted language, and this feature was very convenient during development and debugging, as it allowed us to work interactively with the developed system. However, the result is that some errors (e.g. mistyped names of variables or functions) are detected only at runtime, when the affected line of code is executed. That means, that rarely executed blocks of code may contain errors which will not be detected during testing. This may be especially dangerous when those rarely executed procedures handle emergency situations (which may also be difficult to testing without creating a risk to the system). To avoid this problem it is recommended to use tools, which assess the Python code quality, particularly trying to detect undefined functions and variables (e.g. the *pylint*[23]).

5.3 Problems related to Python syntax

Another problem faced during development was related to the fact, that Python uses only indentation to specify program blocks (instead of "begin-end" or curly braces). When the Python source code was edited in an editor with tabs set to be equivalent to 4 spaces, and which tried to automatically optimize the indentation by replacing spaces with tabs, it resulted in serious corruption of the source. To avoid this problem it is definitely preferable to switch off use of tabs in the editor. Starting of the Python interpreter with the "-tt" option may also help detect a problem with mixed spaces and tabs in the source code.

6. CONCLUSIONS

The presented system solves the problem of integration of the electronics systems supporting the GEM detectors on the KX1 X-ray diagnostic with the Control and Data Acquisition System (CODAS) at the JET facility.

The implementation is done almost solely in Python language, and may be an example of successful application of Python for control and data acquisition systems. Availability of a broad set of libraries (modules) allowed efficiently handling of such tasks as communication via HTTP protocol, implementation of TCP/IP servers and clients and communication with hardware with handling of exceptions.

Use of Python allowed for fast prototyping and convenient interactive testing in the development stage.

Some disadvantages of Python have been also exposed. Its threading model does not support native implementation of threads with different priorities, and it was necessary to implement a special "cooperative" model of a thread's preemption. The runtime only detection of undeclared variables and functions creates a risk that some errors (mistyped variable or function) in rarely executed procedures may not be detected in development. Therefore additional code analysis tools should be used especially in mission critical applications.

ACKNOWLEDGMENTS

This work, partly supported by the European Communities under the contract of Association between EURATOM and IPPLM (number FU07-CT-2007–00061), was carried out within the framework of the European Fusion Development Agreement. This scientific work was also partly supported by the Polish Ministry of Science and Higher Education within the framework of the scientific financial resources in the year 2013 allocated for the realization of the international co-financed project.

REFERENCES

[1] Sauli, F., "GEM: A new concept for electron amplification in gas detectors," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* **386**(2-3), 531 – 534 (1997).

[2] Bartiromo, R., Bombarda, F., Giannella, R., Mantovani, S., Panaccione, L., and Pizzicaroli, G., "JET high resolution bent crystal spectrometer," *Review of Scientific Instruments* **60**(2), 237–243 (1989).

[3] Chernyshova, M., Czarski, T., Dominik, W., Jakubowska, K., Rzadkiewicz, J., Scholz, M., Pozniak, K., Kasprowicz, G., and Zabolotny, W., "Development of GEM gas detectors for X-ray crystal spectrometry," *Journal of Instrumentation* **9**(3), C03003 (2014).

[4] Rzadkiewicz, J., Dominik, W., Scholz, M., Chernyshova, M., Czarski, T., Czyrkowski, H., Dabrowski, R., Jakubowska, K., Karpinski, L., Kasprowicz, G., Kierzkowski, K., Pozniak, K., Salapa, Z., Zabolotny, W., Blanchard, P., Tyrrell, S., Zastrow, K.-D., and JET EFDA Contributors, "Design of T-GEM detectors for X-ray diagnostics on JET," *Nuclear Instruments and Methods in Physics Research, Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* **720**, 36–38 (2013).

[5] Kasprowicz, G., Czarski, T., Chernyshova, M., Czyrkowski, H., Dabrowski, R., Dominik, W., Jakubowska, K., Karpinski, L., Kierzkowski, K., Kudla, I. M., Pozniak, K., Rzadkiewicz, J., Salapa, Z., Scholz, M., and Zabolotny, W., "Readout electronics for the GEM detector," *Proc. SPIE* **8008**, 80080J–80080J–9 (2011).

[6] Kasprowicz, G., Czarski, T., Chernyshova, M., Dominik, W., Jakubowska, K., Karpinski, L., Kierzkowski, K., Pozniak, K., Rzadkiewicz, J., Scholz, M., and Zabolotny, W., "Fast ADC based multichannel acquisition system for the GEM detector," *Proc. SPIE* **8454**, 84540M–84540M–8 (2012).

[7] Zabołotny, W. M., Czarski, T., Chernyshova, M., Czyrkowski, H., Dąbrowski, R., Dominik, W., Jakubowska, K., Karpiński, L., Kasprowicz, G., Kierzkowski, K., Kudła, I. M., Poźniak, K., Rzadkiewicz, J., Sałapa, Z., and Scholz, M., "Optimization of FPGA processing of GEM detector signal," *Proc. SPIE* **8008**, 80080F–80080F–9 (2011).

[8] Poźniak, K. T., Byszuk, A., Chernyshova, M., Cieszewski, R., Czarski, T., Dominik, W., Jakubowska, K., Kasprowicz, G., Rzadkiewicz, J., Scholz, M., and Zabolotny, W., "FPGA based charge fast histogramming for GEM detector," *Proc. SPIE* **8903**, 89032F–89032F–6 (2013).

[9] Zabołotny, W. M., Byszuk, A., Chernyshova, M., Cieszewski, R., Czarski, T., Dominik, W., Jakubowska, K. L., Kasprowicz, G., Poźniak, K., Rzadkiewicz, J., and Scholz, M., "Embedded controller for GEM detector readout system," *Proc. SPIE* **8903**, 89032N–89032N–12 (2013).

[10] Bombi, F., Ciscato, D., Congiu, S., Noll, P., and Zimmermann, D., "CODAS the JET control and data acquisition system," *Nuclear Science, IEEE Transactions on* **25**, 243–247 (Feb 1978).

[11] Jones, E., "Status of the JET control and data acquisition system CODAS," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* **247**(1), 58 – 67 (1986).

[12] van der Beken, H., Best, C., Fullard, K., Herzog, R., Jones, E., and Steed, C., "Coda: The jet control and data acquisition system.," *Fusion Technology* **11**(1), 120–137 (1987).

[13] Krom, J., "The evolution of control and data acquisition at JET," *Fusion Engineering and Design* **43**(3-4), 265–273 (1999).

[14] Hogben, C. and Griph, S., "Interfacing to JET Plant Equipment Using the HTTP Protocol." `http://www.iop.org/Jet/fulltext/EFDR02004.PDF` (April 2002). [Online; accessed 9-June-2013].

[15] "threading — Higher-level threading interface." `https://docs.python.org/2/library/threading.html`. [Online; accessed 21-June-2014].

[16] "Power Strip IP." `http://www.creotech.pl/en/offer/measuring-systems/power-strip-IP`. [Online; accessed 21-June-2014].

[17] "MessagePack - It's like JSON but fast and small." `http://msgpack.org/`. [Online; accessed 21-June-2014].

[18] "ConfigParser — Configuration file parser." `https://docs.python.org/2/library/configparser.html`. [Online; accessed 21-June-2014].

[19] "Global Interpreter Lock." `https://wiki.python.org/moin/GlobalInterpreterLock`. [Online; accessed 21-June-2014].

[20] "NumPy." `http://www.numpy.org/`. [Online; accessed 21-June-2014].

[21] "Killable Threads." `http://tomerfiliba.com/recipes/Thread2/`. [Online; accessed 21-June-2014].

[22] "multiprocessing — Process-based "threading" interface." `https://docs.python.org/2/library/multiprocessing.html`. [Online; accessed 21-June-2014].

[23] "Pylint - code analysis for Python." `http://www.pylint.org/`. [Online; accessed 21-June-2014].