ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-TUSH102

PShell: FROM SLS BEAMLINES TO THE SwissFEL CONTROL ROOM

A. Gobbo†, S. Ebner
Paul Scherrer Institut, 5232 Villigen PSI, Switzerland.

Abstract

PShell is a DAQ scripting tool developed at PSI, in use since beginning of 2015. Initially a beamline data acquisition system at SLS, PShell is being used by various groups for creating tools for the commissioning and operation of the SwissFEL machine.

New features were added to meet SwissFEL requirements, such as supporting beam synchronous data and streamed cameras. In addition to providing a workbench for developing data acquisition logic, PShell also offers a convenient way to create user interfaces ("panels") to trigger the execution of logic. In order to improve user experience and to simplify operation tools these panels can also be launched and used as stand-alone applications.

INTRODUCTION

The PShell project started in 2014. Data acquisition software is not standardized at PSI and the Controls Group aimed selecting a preferred solution, to which it could provide long term support. This tool was meant to be offered as an alternative to new systems, and also for replacing existing aged or ad-hoc solutions. It should be a natural successor of FDA, another DAQ software developed in-house, which had a graphical and declarative programming style, but showing limitations, as many use cases fit better the use of scripting.

The make or buy decision was not trivial. The group spent time assessing existing alternatives such as GDA [1], and Sardana [2], but a different concept was aimed. It was intended a lighter and more flexible solution, based in modern tools, aligned with the standard protocols at PSI: REST [3] for service configuration, ZMQ [4] for data streaming, EPICS for hardware access. Furthermore, a tool not attached to a GUI toolkit, not using any heavy framework. The GUI aspects should be entirely detached and the final goal is the use of web and mobile interfaces. Even if the main GUI environment (called "workbench") is currently Swing-based, it is a stepping stone towards a full functional web based front end.

Another non trivial decision was the technology to use – Java or Python. Java offers a more stable development platform, allowing more control of the project in the long time, providing greater reliability due to the nearly inexistence of native code, and also providing frictionless updates – what was particularly interesting for a server software intended to have high availability. Java offers great advantages regarding deployment as well, comparing to Python. Furthermore, it would simplify rich client development and enable scripting on any dynamic language supported by the Java scripting API, such as Python (with Jython 2.7 [4]) or JavaScript. The Python platform have, in other hand, NumPy and the scientific software stack based on it, which would enable data analysis embedded in the data acquisition scripts, using libraries known to the user. NumPy cannot be loaded directly in Jython because it contains native libraries.

The reasoning for the choice of Java was that, even in a pure-Python solution we cannot avoid interfacing to external data analysis software: for performance reasons, and also for interacting with code written in MATLAB or a different version of Python. Another push for the Java solution is the existence of ways to run CPython code in the same process with little overhead, such as JEP [5], which is a solution for non-demanding cases.

ARCHITECTURE

The project is based on Java 8. In the beginning of the project it was clear the great variety of uses of this software, as different beamlines had different experiences, tools and expectations. The focus was standardizing the logic and data layer, but leaving the users free on the choice of GUI. In this way even if beamlines have different preferences for interfaces, the DAQ code is homogeneous.

To reach this goal PShell provides architectural freedom. Logic is executed by a core engine, and interfaces can be CLI (using the command line interface), GUI (the embedded workbench, running on the same process as the core), remote (a custom developed application), web (using the built-in web client or a custom web application) or mixed. The core engine can also be embedded in other Java applications.

The workbench can be the user front end, or else just used as a development environment. PShell can be executed then in pure server mode, having custom user graphical interfaces.

Remote access benefit from a built-in web server and a REST interface. Writing client code is simple. Remote calls typically trigger and monitor script execution, evaluate interpreter statements, monitor devices and access data.

Regardless the architectural choice, PShell runs in a single process, and is deployed as a single jar file. Figure 1 shows a diagram of the relations between the system components.

Core Engine

The core engine consists of:

- Script interpreter based on the Java Scripting API. Python is the primary language (Jython 2.7) but JavaScript and Groovy are also supported. A set of built-in functions is available for user scripts, simplifying scanning, plotting, data persistence and data

† alexandre.gobbo@psi.ch

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-TUSH102

analysis. Some function names are inspired by SPEC [6].

- A data persistence manager, having HDF5 [7] as the default file format, but also supporting text files. The data model in PShell assumes a hierarchical storage, and extension libraries can add support to other hierarchical file formats. Scan data can be automatically saved using data layout schemas. There are predefined layouts, but they can also be customized.
- A plugin manager for loading extensions and plugins, in the form of .jar files, or else .java files dynamically compiled.
- A global device pool. Scripts can reference devices declared in the device pool and can create new ones. EPICS, Modbus, serial, sockets, PSI detectors and stream cameras are supported out of the box but new device families may be added in extensions.
- Automatic versioning system based on GIT. A local repository includes the scripts, configuration files, and the user plugins. Files are automatically committed every time scripts are executed. The local repository can be automatically pushed to a remote GIT repository.
- A command line console interface.

Server

PShell has multiple server options. It contains an embedded web server and a REST interface which provides access to all core functionalities. The standard web client allows controlling script execution and features an interactive console, plotting and data access components. A data server, based on ZMQ, streams out scan data and also provides access to data files. Finally, a raw socket connection gives access to the console.

The web client technology is currently based on HTML5, jQuery [8], Flot [9] and Bootstrap [10].

Workbench

The GUI workbench constitutes a development environment for scripts and also an alternative for end user interface, providing:

- Script editor with code auto-completion.
- Advanced console with auto-completion and command history.
- Script execution control and debugging.
- Plotting environment.
- Data browser.
- Image renderer.
- System configuration panels.
- Device configuration and supervising panels.

Add-ons

Other than scripts, the system can be customized with extensions and plugins.

Extensions are merely .jar files, placed in a specific folder and added to the class path. They can depend on PShell .jar file or not. Two different types of extensions can be created:

- Simple libraries, making classes available to the core and scripts at run time. For example libraries can add new family of devices, data file formats or mathematical algorithms.
- Workbench components, adding or replacing functionality using Java's Service Provider Interfaces [11], a standard way to create extensible applications in Java. By implementing certain interfaces defined in PShell, and declaring them in a configuration file within the extension .jar file, the feature can be identified and used. Examples are registering a new plot types or a new logic executing components in the workbench.

User plugins must contain a class implementing the *Plugin* interface. Through this interface the plugin receive events from the core engine, have access to the current context, control the execution of scripts, evaluate commands in the interpreter and access the device pool. Plugins should be placed in a specific folder. They can be in two forms:

- Static: a project compiled into a .jar file.
- Dynamic: a single .java file, implementing the *Plugin* interface, compiled on-the-fly, as needed (recompiled if the source file changes).

Plugins can be configured to be loaded on start-up, and can be dynamically reloaded in the workbench.

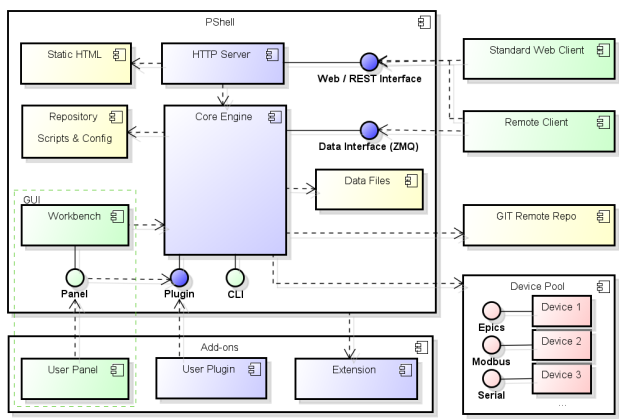

Figure 1: Component diagram

Panels

Advanced users can work directly on the workbench, but custom graphical user interfaces are useful for guiding and simplifying the user experience, for inputting and checking script parameters, for better presenting the results, and also for hiding scripts from end users. A simple method for creating custom interfaces was aimed.

Users can quickly learn coding DAQ scripts, and ideally they should be able to create custom GUIs to trigger those scripts just as easily, in the same way they can create synoptic caQtDM [12] panels with no programming background.

PShell proposes a straightforward solution to create end user graphical interfaces to DAQ scripts. It is called "panel", a special type of graphical plugin that can be loaded to the workbench, and also can be executed detached, as a

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-TUSH102

stand-alone application. When detached, the plugin runs in its own process and holds an instance of the core engine, as if the core was running embedded in an external application. Detached mode is selected by a command line option, with the name of the panel to be loaded.

A panel consist of a single .java file, dynamically compiled as needed. Panels can be created and loaded within the workbench, where they can also be edited. However the suggest way to edit panels is by using Netbeans Swing GUI editor [13], which is a very user-friendly environment for GUI development in Java. Swing components are drag and drop to the user panel. PShell internal widgets are also available for composing these interfaces - for plotting, image rendering, and device control. The panel can access the core engine and easily control the execution of scripts, in an asynchronous fashion.

It was desirable that the panel inherits *JPanel* (so it can be placed in any Swing container), but also that it inherits a *Plugin* class, so that it contains all methods needed to interface the core. The use of auxiliary classes in order to access plugin functionality would make the panel programming more difficult for users with no experience in Java: plugin methods should be directly available in code completion. The intended design was possible by making use of "default methods", introduced in Java8. All plugin functionality is coded in default methods of the *Plugin* interface, implemented by *Panel* class. Java8 does not support multiple inheritance of state, just of behavior. But assuming that there is only one running instance of a plugin, it is possible to build a global dictionary of plugin states, in fact realizing a multiple inheritance of state.

Data Analysis

Online data analysis capabilities were the greatest challenge for proposing a Java-based DAQ solution. Different solutions were implemented to deal with the problem.

The first need was a set of basic mathematical functions, accessible to scripts, to deal with the common calculations following a scan, in particular typical beamline alignment problems: fitting (polynomial, harmonic, Gaussian), peak search, least square optimization, derivative, interpolation, FFT, and statistical functions. These were implemented with the use of Apache Commons Math [14]. Façade methods were written to provide simple access to this functionality on user scripts.

The second need was imaging processing algorithms. PShell includes ImageJ [15] binaries, and provide many helper functions for scripts using ImageJ, such as image conversions, basic operators, convolution, Euclidean distance map, thresholding, binary operations, FFT and particle analysis.

Finally, it remains the problem of interacting with CPython code using NumPy/SciPy. The issue is not running externalized data analysis (in different processes), which can be done for example over ZMQ communication, but finding ways of calling CPython code directly from scripts. PShell includes functions to start CPython as an external process, executing a given function with parameters and getting results back. This method is simple to use but not performant, only fitting cases when the overhead of instantiating Python is negligible. A better solution is obtained by using JEP. A CPython interpreter is created in the same process, sharing the same memory space. With simple wrapper calls, CPython code can be called directly from DAQ scripts. The overhead is just a memory copy each time Java arrays are transformed to/from a NumPy array. JEP supports both CPython 2 and 3, no matter which DAQ scripting language is used.

FIRST DEPLOYMENTS AT SLS

The first deployment of PShell was on beamline X03DA (Pearl) at SLS, in production since February 2015, with an increasing scope over time. All user interfaces for this beamline data acquisition are done within the workbench. There are 4 typical DAQ experiments at X03DA. For each one a specific panel has been created, so most user operation is straightforward. Non-standard experiments are scripted. Figure 2 shows a screenshot of user interface at X03DA.

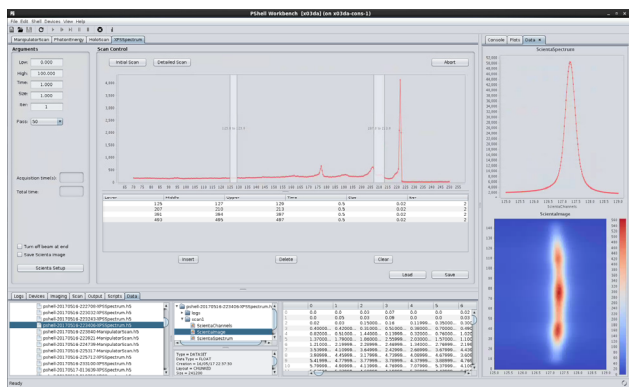

Figure 2: PShell workbench at X03DA beamline.

Next systems were installed in X07MA(X-Treme), and X11MA (SIM) beamlines. In both cases it was implemented scripting and user interfaces for DAQ.

The scope of use in some beamlines is different: on X06DA (PX3) it is used mainly for beamline optics alignment, having its functions accessible to the beamline control software. In this case PShell can be used in server mode. On X12SA (CSAXS) it has been used for device test and commissioning.

Currently X07MB (Phoenix) and X10DA (SuperXAS) are in the process of migrating from FDA to PShell. In order to facilitate this transition a plugin was included to load and execute FDA configuration files, and also to convert those files into PShell scripts.

PShell AT SwissFEL

In the SwissFEL control room most synoptic display panels are implemented using caQtDM. Although caQtDM enables many advanced operations - for example by running shell scripts - sometimes it is not enough to build more advanced utilities, for commissioning, machine development or routine operation. In those cases a full-featured programming language is more appropriate. PShell is used as an alternative, proposing a structured

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-TUSH102

way to produce scripted logic, and also user interfaces. It has been used by various groups (Operations, RF, Diagnostics and Photonics), both to execute one-time scans, for testing and commissioning, and also to create permanent operator utilities.

PShell is used in SwissFEL control room since May 2016. New requirements appeared with the use at SwissFEL: support to new families of devices and adaptation to the use in a control room environment.

Beam-Synchronous Devices and Data Streaming

The first adaptation for SwissFEL was integrating beam-synchronous data, streamed over ZMQ at up to 100Hz [16], using the in-house developed BSREAD. This library provides data serialization and stream control over ZMQ. The PShell device model was adapted to support beam-synchronous data, and so were the scanning functions – which can now mix traditional and streamed devices. New functions have been added to deal specifically with beam-synchronous data.

SwissFEL cameras are accessed through servers, that stream out images and image processing data also using BSREAD. The purpose of these servers is to provide standard measures (such as characterization of the beam) at 100Hz.

PShell features an imaging framework that allows including new imaging sources. The support to the camera servers was added, including access to the image processing data. EPICS, beam-synchronous and imaging data can be used together in the same DAQ process. Figure 3 shows the edition of a DAQ script using mixed data sources.

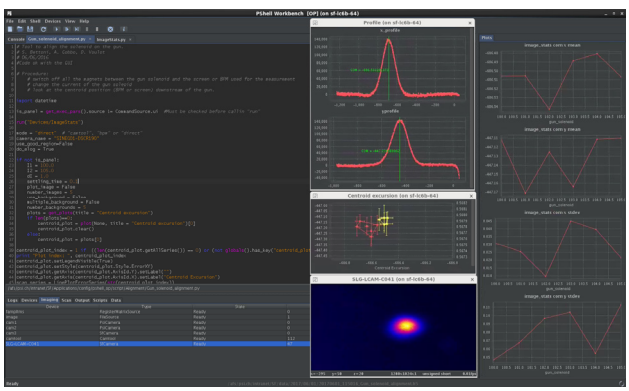

Figure 3: script mixing device and imaging sources.

User Interfaces

In the SwissFEL control room, the operation tools are organized in a start menu called “launcher”, which is present in all operation consoles. Each menu item starts a specialized window – such as a synoptic panel. This does not fit well with the use of PShell workbench – which is suited to centralize all user operation. In order to adapt to this organization of tools, it was implemented the use of panel plugins in “detached mode”. For each script implementing a user tool, a matching panel is created. The “launcher” menu starts the panels detached, which are perceived as a stand-alone window. The workbench is therefore used as a development environment but it is not seen by the end user (operator). Figure 4 shows a detached panel controlling a DAQ script.

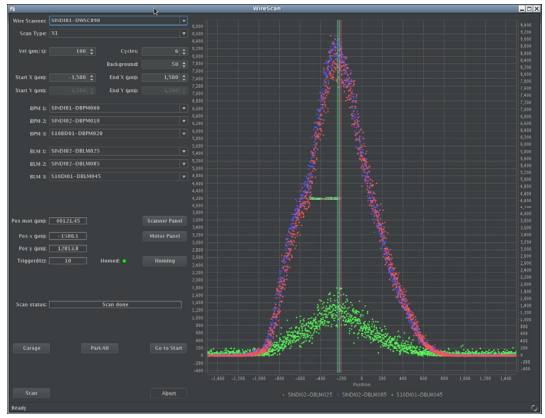

Figure 4: Detached panel plugin.

Multiple Instances

A big difference comparing to beamline operation is that commonly in beamlines there is a single PShell process, while in the control room there are many.

Each PShell deployment is defined by a configuration structure in a repository, pushed to a remote GIT server. We called it a “context”. PShell execution is typically “exclusive” - only one instance of PShell runs per context. In this way we ensure no conflicting concurrent access is done, by running only one high-level DAQ script at a time. Parallelism, if needed, is handled by a specific API, allowing scripts forking concurrent threads. We can have multiple visualization clients (local, remote, web), but a single instance of the core. This behaviour fits well beamline operation.

In in a multi-console control room environment users are expected to work on multiple panels simultaneously, having access to different aspects of the machine. Creating one context for each operator panel would be excessive. For this kind of use, PShell can be started also in “local mode”, where the context is shared between multiple running instances. Operator panels are launched in local mode, sharing therefore the same system and device configuration.

The control room has also a server instance of PShell, running in exclusive mode, enabling remote access. The server is needed because some DAQ scripts are triggered by client software, such as caQtDM panels and external Python scripts. The server instance can be inspected through its web interface.

By targeting the server REST interface, clients are able to start script execution and display ongoing data and results. In the case of caQtDM, it is easy to make use of the REST interface to trigger actions, but caQtDM cannot use it to read back scan data and plot it. The interface is, in this case, done through EPICS channels. These channels can be created by PShell itself with a Channel Access Server [17]. The creation of EPICS channels to publish results is straightforward, and so it streaming out results over ZMQ streams.

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-TUSH102

OTHER USES

Proscan

An environment for automated execution of test scripts for PROSCAN has been implemented as a PShell plugin. A test database containing devices and test scripts is loaded and displayed in a table. A test plan defines a sequence of tests (or groups of parallel execution). Test plans can be saved and reloaded. They are intended to run automatically, validating different aspects of the facility in the course of maintenance days.

SwissFEL ESB-MX Sample Changer

PShell is the development platform for the control software of the new MX sample changer system, in development at PSI to equip the ESB-MX instrument for SwissFEL. This project benefits from the existing imaging functionalities, inherited from ImageJ, the supported devices, and from the extensible device model, which allows to quickly integrate new hardware. Furthermore, the use of workbench as a the prototyping tool enables a smooth transition from prototyping to the final product, allowing people involved in the project to better understand the system as each process is implemented as a single Python script.

CONCLUSION

During the past years PShell has been used in production environment and has grown in scope. Today it reaches a stable state, where it is able to cope with a wide range of needs. The project development costs were low thanks to the number of very high quality libraries available for the Java platform, and the power of Java IDEs.

Efforts will be made in the future for enhancing the web interface, in particularly improving the plotting capabilities to match the workbench's, aligning it to more modern technologies used within the group such as Polymer [18] and Plotly [19].

The desktop interfaces remain necessary though. The creation of end user interfaces was much simplified with the use of panel plugins. This type GUI development can be carried on even by users with no background in software development.

At SLS an increase in the number of deployments is expected, as FDA is planned to be decommissioned from other beamlines and new beamline DAQ systems are being discussed.

At SwissFEL more emphasis will be put on enabling the users creating their own graphical interfaces, in simplifying device configuration (for example autocompleting EPICS channel names), and also in providing better integration with CPython.

A seamless use of CPython remains a major objective. The use of JEP enables directly calling CPython code with little overhead and boilerplate code, but writing wrapper functions remains a difficulty to users, in particularly understanding the conversion between NumPy and Java arrays. Two possibilities remain to be explored. The first is JyNI [20], a project that proposes directly importing native Python modules into Jython. This project is still in beta state, but should be assessed when a first version is released.

Finally, an alternative is PShell not targeting the Java scripting framework for running scripts, but calling CPython in interactive mode – in other words functioning as an IDE for Python development. PShell built-in functions should be adapted. The scan library has already been ported to pure Python. The plotting environment can be accessed remotely (over ZMQ) and the scanning and plotting functions can be adapted to use this server interface. The Java device classes wouldn't be available, but pure Python counterparts could be created in order the scripts to be compatible.

ACKNOWLEDGEMENT

The authors would like to thank all the users that have helped this project to reach a mature state, for their patience in the earlier stages. In particularly, for the especial attentiveness and constructive criticism: Matthias Kurt Muntwiler, Didier Voulot, Jan Gui-Hyon Dreiser, Cinthia Piamonteze, Roger Kalt, Luka Debenjak and Marco Boccioli.

REFERENCES

[1] GDA, http://www.opengda.org
[2] Sardana, http://www.sardana-controls.org
[3] ZMQ, http://zeromq.org
[4] Jython: Python for the Java Platform, http://www.jython.org
[5] JEP, https://github.com/ninia/jep
[6] SPEC, http://certif.com
[7] The HDF Group. Hierarchical Data Format, version 5, 1997-2017, https://www.hdfgroup.org/hdf5
[8] jQuery, https://jquery.com
[9] Flot, http://www.flotcharts.org
[10] Bootstrap, http://getbootstrap.com
[11] The Java Tutorials, The Extension Mechanism, http://docs.oracle.com/javase/tutorial/ext
[12] caQtDM, epics.web.psi.ch/software/caqtdm
[13] Swing GUI Builder (formerly Project Matisse), http://netbeans.org/features/java/swing.html
[14] Apache Commons Math, http://commons.apache.org/math
[15] C. Schneider *et al.*, "NIH Image to ImageJ: 25 years of image analysis", *Nature Methods*, vol. 9, no. 7, 2012, pp. 671-675.
[16] C. Milne *et al.*, "SwissFEL: The Swiss X-ray Free Electron Laser", *Applied Sciences (Switzerland)*, vol. 7, article no. 720, 2017.
[17] CAS: Channel Access Server Library, http://www.aps.anl.gov/epics/extensions/cas
[18] Polymer, https://www.polymer-project.org
[19] Plotly, https://plot.ly
[20] Jython Native Interface, http://jyni.org