Karabo-GUI: THE MULTI-PURPOSE GRAPHICAL FRONT-END FOR THE Karabo FRAMEWORK

Burkhard Heisen, Martin Teichmann, Kerstin Weger, John Wiggins,
European XFEL, Hamburg, Germany

Abstract

The Karabo GUI is a generic graphical user interface (GUI) which is currently developed at the European XFEL GmbH. It allows the complete management of the Karabo distributed control and data acquisition system. Remote applications (devices) can be instantiated, operated and terminated. Devices are listed in a live navigation view and from the self-description inherent to every device a default configuration panel is generated. The user may combine interrelated components into one project. Such a project includes persisted device configurations, custom control panels and macros. Expert panels can be built by intermixing static graphical elements with dynamic widgets connected to parameters of the distributed system. The same panel can also be used to graphically configure and execute data analysis workflows. Other features include an embedded IPython scripting console, logging, notification and alarm handling. The GUI is user-centric and will restrict display or editing capability according to the user's role and the current device state. The GUI is based on PyQt technology and acts as a thin network client to a central Karabo GUI-Server.

THE KARABO DISTRIBUTED CONTROL AND DATA ACQUISITION SYSTEM

Karabo is the control and data acquisition system which will be used on all beamlines of the European XFEL to control the equipment, acquire data and process it [1]. It is a centralized system where all communication is done via a central broker, except for high-bandwidth data streams which are transported via dedicated point-to-point connections.

A Karabo system is a collection of *devices*. Those devices can serve many purposes, they might be a driver for a particular hardware, a composite device that controls several other devices if some coordination between different hardware is necessary, a data processing device that may be just one of hundreds of equal ones in a server farm processing data, or a device saving raw or processed data to disk, not to mention many service devices which keep Karabo running.

There are many different approaches of a GUI for a control system. Some use an already existing development enviroment and extend them to the needs of developing GUIs (e.g. GDA [2], CSS [3]). Others developed stand-alone graphical editors (e.g. JDDD [4], Taurus [5]). Karabo has one fully integrated GUI, which is not only a graphical editor but can be used for all control and data acquisition tasks.

PROJECTS

A typical user of Karabo is not interested in the entirety of the system, rather a specific task to work on. Those tasks are often overlapping, as an example a vacuum technician may want to interact with the same components of an apparatus a scientist works on.

All information necessary for a task can be bundled into a *project*. These are all the devices needed and the configuration with which they should be started, or into which they should be reconfigured to perform the desired tasks. Graphical visualizations of tasks can be added as *scenes* to the project. They are used to show and edit the configuration of devices and the connections between them. Repetitive tasks reoccuring for users of a project may be programmed as macros. If taking some data is the purpose of a project, the data to be taken can be stored in the project as *monitors*. The project itself, however, does not contain the data, which is written to disk independently.

The projects are generally stored on a central server. This way everything needed for a task is persisted, and enables users to use their projects on different computers, and makes the administration of Karabo installations easier, as projects can be archived much simpler.

The project is the core concept of the Karabo GUI, and most of the rest of this paper is a description of its components. Figure 1 shows a screenshot of a running Karabo GUI with all components.

LIVE NAVIGATION AND CONFIGURATION

A Karabo device is run by a Karabo *server*, which is software running on a computer. The device code to be run is installed as plugins into those servers. The installation and running of those servers is beyond the scope of Karabo, but it is typically done with common server administration software.

In the GUI, such an installation is shown as a tree, for each computer the available servers are shown, and the available plugins for each server. A user can thus see the entire Karbo installation. From the GUI, the devices can be instantiated from their *device class*, the code to be run for a device.

Before a device is even instantiated, its parameters are already known to the device server and communicated to the GUI. Thus a user can configure a device interactively by filling out a form which is automatically generated from the description of the device class. Those configurations can be stored in the project. As the full live navigation tree of a Karabo installation can grow large and confusing, those stored configuration give a good overview of the devices needed in a particular project.

ISBN 978-3-95450-148-9

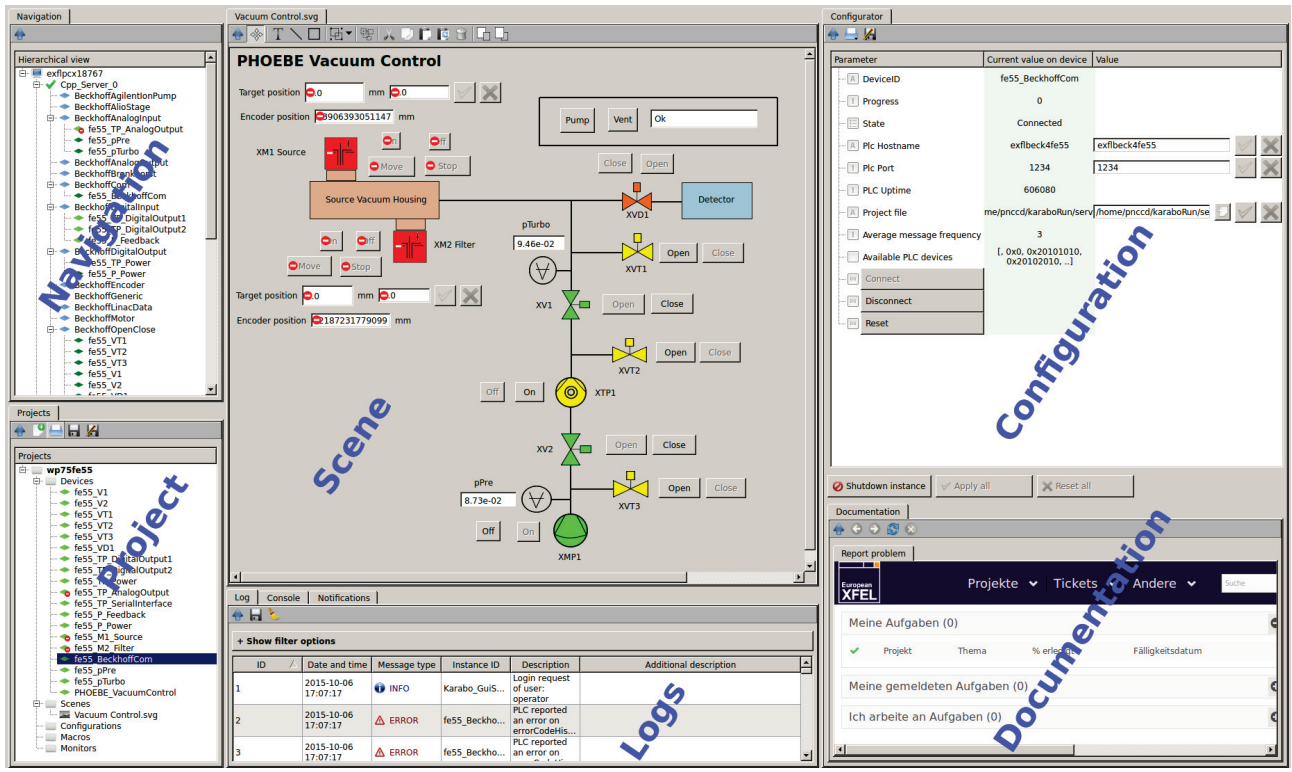

Figure 1: Screenshot of the GUI. (a) The live *Navigation*, with computers, servers, device classes and devices. (b) the *Configuration* of one device, automatically generated from its self-description, (c) a *Scene* with several device properties shown, (d) the *Project*, (e) the message *Logs* and (f) *Documentation*.

GRAPHICAL VISUALIZATION

The properties and commands of devices can be visualized in a graphical *scene*. This is done by simply dragging the property from the device's configuration view into the central panel of the GUI, where those scenes are shown. This is possible for both already running devices and device classes which are known to the system. The latter can be pre-configured and stored in the project, and their properties can be part of a scene.

Within the scene, the user can choose the way the data is shown. Besides variations of ways to show numbers on dials and other numerial indicators, there are also advanced widgets. As an example, numerical values can be shown in a trendline widget, showing the time evolution of a property. By simply moving the viewed time axis to the past, one can even retrieve historical data from Karabo's data logging service.

Commands to a device are represented as push buttons. They may show icons depending on the state of the device.

Once a scene has been designed, it is typically switched from design to production mode, such that the widgets actually react to user input. This can always be switched back, so that the design of the scene can be edited again. Scenes may also be detached from the rest of the GUI. This way a scene effectively looks like an independent application.

Scenes can also be used to design data processing workflows. Devices which produce or consume high-bandwidth data can be dragged into the scene and are shown as boxes with plugs representing the input or output data streams. Those plugs are then connected by "wires", as seen in Fig. 2. For high-throughput data applications, when the devices should run on many computers in parallel, one box in the scene may also represent a large number of computers each running the same device. Samples of the data can also be shown in the GUI for inspection.

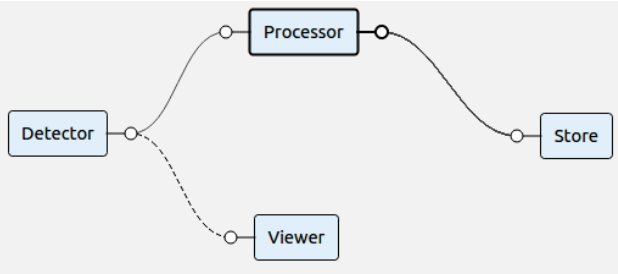

Figure 2: A simple workflow. In this workflow, a *Detector* device creates a data stream which is processed by the *Processor* device group, whose results are sent to the *Store* device. Samples of the data are copied to a *Viewer* device.

Technically, the file format of the scenes follows the SVG standard with some extensions. This allows the user to copy and paste graphical elements from an SVG editor into the scene. While the graphical editing capabilities of the Karabo

ISBN 978-3-95450-148-9

```
from karabo import *

class Scan(Macro):
    camera = RemoteDevice("camera1")
    start = Float(description="Begin")
    stop = Float(description="End")
    steps = Int()
    average_intensity = Float()

    @Slot()
    def execute(self):
        """Perform a camera scan"""
        with getDevice("motor1") as m:
            m.targetPosition = self.start
            m.move()
            waitUntil(lambda:
                m.status == "stopped")
            self.camera.takeImage()
```

Figure 3: A macro code example. An excerpt of a scan macro showing the syntax of the macro language. Macros are classes with executable slots that perform one task. They can have arbitrary parameters, which serve as input or output. Macros may continuously control devices with `RemoteDevice`, or temporarily with `getDevice`.

GUI are limited, this feature allows for the creation of visually appealing scenes using external SVG editors.

MACROS

While devices in Karabo are also used to automate repetitive tasks, they are often too cumbersome to be used, as they need to be installed on servers by administrators. This is where macros come into play: they are basically Karabo devices which can be entered directly into the GUI and started with a mouse click (they can also be started from the command line, but that is beyond the scope of this article).

The code is not executed in the GUI, as those are often running on user machines with an unreliable internet connection. Instead, they are sent to a central macro server and executed there.

Macros are much more than just a list of commands to be executed sequentially. They are written as a Python class, where the methods of the class can be executed by the user. They essentially behave like devices, so they can also be configured, and their properties can be made editable in a scene. A short example is shown in Fig. 3.

Finally, the only difference with a normal device is that macros should be used for specific tasks which are relevant in a particular context only. Everything generalizable should be made into an actual device and maintained by computer administrators.

LOGGING AND A CONSOLE

Karabo devices can broadcast messages that users may be interested in within the distributed system. Those are shown in a logging panel within the GUI, where those messages can be sorted, searched and filtered.

There are many tasks which can be more easily done on a command line than graphically. Therefore the GUI has a console panel in which an IPython session can be started to control Karabo. The same programmer's interface as for the macros is used, so that commands for a macro may be tested on the command line.

TECHNICAL DETAILS

The GUI is written in Python 3.4, using PyQt4 for the graphical output. Many of the widgets in the scene use PyQwt5 and guiqwt. The same code can be run under Windows, MacOS and Linux operating systems. Connection to a Karabo installation is established via TCP with a dedicated protocol. This way the GUI can also be used outside of the protected network of a Karabo installation.

Like the rest of Karabo, the GUI is completely event driven. The user cannot initiate any blocking operation. When the GUI sends a user's request to the network, instead of waiting for a response the GUI continues to work normally and will show results of the request when it arrives.

Projects are ZIP files, which contain the the data as XML files and in case of the macros, as Python source files. This allows users to inspect and change projects using other tools.

CONCLUSION

The Karabo distributed control system contains a single graphical user interface which integrates everything to control and use an installation. The user interacts directly with the live system. Via the self-description of the running device a user gets an immediate idea about its capabilities, allowing for an intuitive use of the system.

REFERENCES

[1] B. C. Heisen, D. Boukhelef, S. Esenov, S. Hauf, I. Kozlova, L. Maia, A. Parenti, J. Szuba, K. Weger, K. Wrona, C. Youngman, "Karabo: an Integrated Software Framework Combining Control, Data Management, and Scientific Computing Tasks", ICALEPCS2013, San Francisco, CA, USA 2013

[2] http://www.opengda.org

[3] Jan Hatje, M. Clausen, Ch. Gerke, M. Moeller, H. Rickens, "Control System Studio (CSS)", ICALEPCS07, Knoxville, TN, USA, 2007

[4] E. Sombrowski, A. Petrosyan, K. Rehlich, P. Tege, "JDDD: a Java DOOCS data display for the XFEL", ICALEPCS07, Knoxville, TN, USA 2007

[5] http://www.taurus-scada.org

ISBN 978-3-95450-148-9