ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-TH1BCO06

THE KARABO CONTROL SYSTEM

S. Hauf*, N. Anakkappalla, J. T. Bin Taufik, V. Bondar, R. Costa, W. Ehsan, S. Esenov, G. Flucke, A. Garcia-Tabares, G. Giovanetti, D. Goeries, D. Hickin, I. Karpics, A. Klimovskaia, A. Parenti, A. Samadli, H. Santos, A. Silenzi, M. A. Smith, F. Sohn, M. Staffehl, C. Youngman, European XFEL, Schenefeld, Germany

Abstract

The Karabo distributed control system has been developed to address the challenging requirements of the European X-ray Free Electron Laser facility, which include custom-made hardware, and high data rates and volumes. Karabo implements a broker-based SCADA environment. Extensions to the core framework, called devices, provide control of hardware, monitoring, data acquisition and online processing on distributed hardware. Services for data logging and for configuration management exist. The framework exposes Python and C++ APIs, which enable developers to quickly respond to requirements within an efficient development environment. An AI driven device code generator facilitates prototyping. Karabo's GUI features an intuitive, coding-free control panel builder. This allows non-software engineers to create synoptic control views. This contribution introduces the Karabo Control System out of the view of application users and software developers. Emphasis is given to Karabo's asynchronous Python environment. We share experience of running the European XFEL using a clean-sheet developed control system, and discuss the availability of the system as free and open source software.

INTRODUCTION

Karabo is a supervisory control and data acquisition (SCADA) system, developed to meet control and data acquisition requirements of the European X-ray Free Electron Laser (European XFEL) [1]. Development of Karabo was started in 2010, after after surveying other well-known systems such as Tango [2], EPICS [3], and DOOCS [4] as possible control solutions for the planned facility in 2009. At the time of this survey, the anticipated complexity of the European XFEL, and the data volumes generated at the facility were found challenging to address with existing SCADA systems. Consequently, the development of Karabo was started. Karabo is successfully being used to control the photon systems and instrumental end stations of the European XFEL since 2017. In June 2023 it was made available to the public as free and open source software [5]. In this contribution we give a general overview of Karabo's architecture, discuss the operational benefits and drawbacks of this architecture, and conclude with an outlook on how AI-driven agents can facilitate development in the Karabo Ecosystem.

KARABO: CONCEPTS AND ARCHITECTURE

There are two features of the European XFEL that distinguish it from earlier light sources and have direct impact on the requirements for a control and data acquisition system at the facility: the unique time structure of the accelerator that enables MHz bunch repetition rates, and bespoke 2d imaging detectors [6] capable of resolving this time structure, resulting in data rates between 10-20 GB/s. Additionally, the facility initially rapidly changed during construction and commissioning, and nowadays instrumental setups are substantially modified depending on experimental needs, which can change with each user group on a weekly basis. For the control system the above translates to the following boundary conditions:

- the control system needs to scale and can grow alongside the facility,
- has time correlation woven into its fundamental data model,
- process data rates of tens of GByte/s at latencies of a few 100 ms,
- and cater to dynamic experimental setups that change on different time scales,

while being highly reliable and resilient to failure events. Karabo was written from scratch with these requirements in mind.

Karabo's Asynchronous and Event Driven Design

Karabo communicates by asynchronously exchanging messages via a central message broker. For every Karabo installation, a broker name space, referred to as a *Karabo topic* exists, and distributed components are uniquely identifiable within this topic. *Devices* add functionality to the base system in form or pluggable software libraries, and the combination of all online and offline instances of devices in a topic constitutes the control system for this topic.

A completely event-driven publish and subscribe signal-slot messaging pattern is implemented on top of the distributed broker messaging. By subscribing to signals of another instance updates are propagated through the system without a need for polling. Any configuration update will include timing information comprised of timestamp and a unique timing identifier which facilitates correlation on a global facility level. Any number of distributed components can subscribe to a given signal which issued only once, regardless of the number of subscribers, thereby minimizing network traffic. Instance methods are registered as *slots* to make them available throughout the distributed system.

* steffen.hauf@xfel.eu

Content from this work may be used under the terms of the CC BY 4.0 licence (© 2023). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-TH1BCO06

The payload of any distributed message is a *Karabo Hash*. This hierarchical key/value container additionally supports per-element attribute assignment. The keys are strings and as value the following data types are supported: integers, floating points, strings, the Hash itself, vectors of all of these, the Schema and a special container for multi-dimensional arrays.

The European XFFEL is currently transitioning from Java Messaging Service (JMS) broker technology [7] using an Open Message Queue implementation, OpenMQ(C), to RabbitMQ brokers [8], implementing an AMQP protocol [9]. This transition is expected to be completed by the end of 2024 and is necessary because the OpenMQ(C) library is not actively supported anymore. AMQP in combination with RabbitMQ will additionally add better fail-over support in case of short broker or network disruptions.

A separate asynchronous communication channel exists for ”large” data such as images or digitizer traces. These pipeline connections are peer-2-peer TCP/IP connections between two devices, specifically an output and an input channel, that circumvent the broker. Pipeline connections can sustain data rates of a few Gigabytes per second, and can be configured to implement scatter/gather topologies, with in-built buffer queues [10].

Karabo's Application Programming Interfaces

Three application programming interfaces (API) facilitate developers to implement the distributed components of Karabo, the devices. Each API has distinctive features from which specific applications can benefit.

- The C++ API is beneficial for high-performance applications, or for integrating third-party C or C++ libraries.
- The Python Bound API provides high-performance pipeline processing when an application additionally needs to leverage Python packages like NumPy or SciPy. Its name indicates that it mainly consists of Python-binding on-top of the C++ API.
- The Middlelayer API (MDL) is a native-Python, low-boilerplate integration option that excels through rapid development and iteration cycles. We will discuss the expressive Python code that can be written in this API further in Sections and .

Table 1 lists a selection of tools and services critical in operation of the facility and operational support. The choice of API for each service is indicative of each API's aforementioned strengths. A more detailed discussion of the examples listed in the table can be found in [11].

Figure 1 shows the number of devices implemented in each of the three APIs over time. It is evident from the figure, that from 2019 onward, most new devices have been implemented in the Middlelayer API. Feedback from developers suggests that this is mainly due to its expressive syntax and the rapid development cycles Python facilitates. However, as is shown in Table 1, devices critical for operation of the European XFEL leverage the specific strengths of the C++ and Bound APIs, and are actively being maintained and extended. New devices requiring C++ or Python Bound capabilities are also common: recent developments in the field of online data reduction for instance are being implemented on top of these.

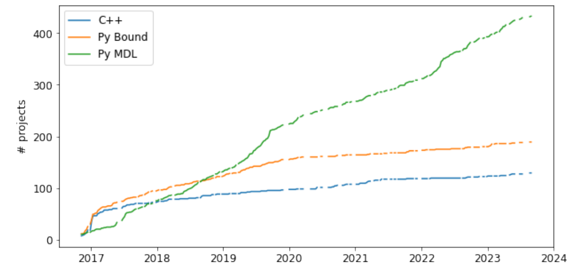

Figure 1: Number of devices implemented in Karabo's three Application Programming Interfaces over time.

The Karabo GUI: Karabo's Graphical User Interface

The Karabo GUI is the main operator interface for the SCADA system for control tasks. It is a single, stand-alone application, implemented in Python using the Qt [12] layout manager. The GUI connects to GuiServer devices over a TCP/IP connection, rather than the message broker, and authenticates users upon login. The message format nevertheless is based on the Karabo Hash, and a Python native implementation there-of. This design makes the GUI client application highly portable: it is available on various Linux flavors, as well as MacOs X and Microsoft Windows. TCP connections additionally facilitate an increasingly important feature: remote access to the system, using e.g., SSH tunneling.

Configuration updates to the GUI are usually throttled to

Table 1: Examples of Important Karabo Services at the European XFEL by API with Annotations

Service	API	Comments
Data Acquisition	C++	Ingests data rates of up to 20 GB/s.
Data Logging	C++	Event-driven logging into InfluxDB of 3 million control parameters.
Beckhoff Integration	C++	Communication Interface to PLCs comprising $\approx$ 90 % of control solutions at the facility.
Online Calibration	Bound	Online processing of up to 4000 images/s via pyCuda.
Karabacon	MDL	Scantool with GUI, processing plugins and scan history.
Recovery Portal	MDL	Time machine-like tool capable of configuring entire beam lines to a prior point in time.

Content from this work may be used under the terms of the CC BY 4.0 licence (© 2023). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-TH1BCO06

Content from this work may be used under the terms of the CC BY 4.0 licence (© 2023). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI

2 Hz for broker data, and dynamically for image or other pipeline data, depending on client load. This design ensures a responsive interface, even if many cameras are viewed. Generally, connections to distributed components are established via the GUI servers on a per-need basis - only if a device is in view, will data from it be passed to the GUI.

The GUI application is organized into multiple panels, each optimized for different tasks. Distinct features such as projects and scene panels coexist with control system essentials like a configuration panel to modify the configuration of online and offline devices, and a system topology view. Projects store the configurations of a selection of device, alongside macros (small scripts), and scenes, in a non-relational database [13].

To create synoptic views, any device property can be dragged from the configuration panel and dropped onto a scene. Scenes consist of controllers (widgets) appropriate to a property's data type. A scene-model [14] defines the layout of scenes. Generally, operators interact with the control system through these scenes or the configuration panel (as shown in Fig. 2).

Using projects and scenes, operators can build flexible user interfaces without coding. Alternatively, a programmatic composition of scene models is possible, and can be embedded into device code. These device-provided scenes can be accessed by double-clicking on an instance in a topology panel of the GUI.

IMPLICATIONS OF ARCHITECTURAL CHOICES

In the following we report operational and development implications of the architectural choices that were made for Karabo, as observed in more than 5 years of facility operation. These observations are structured around the major architectural choices identified in Section .

Event-driven and Asynchronous Design

From an operational point of view, the most important observation is that operators can struggle with an event-driven system if they are accustomed to systems that mainly poll data. Especially during commissioning and the early operation phase, a recurrent request was to implement polling loops, as a reassurance that data was indeed updating. Such requests rarely occur nowadays, and if so, are motivated by the need to facilitate a particular type of experiment or data analysis, rather than a lack of trust in the system in general.

From a software-engineering perspective the event driven and asynchronous design has been a significant enabler in two major aspects. Device composition and orchestration is greatly simplified, and higher level functionality can frequently be implemented without the need to know, or even alter messaging specifics in more basic devices. Especially in the middelayer API, it is straight-forward for a developer to express even complex device coordination tasks, involving multiple subordinate devices, in an expressive fashion with Python's asyncio facilties. The following listing exemplifies this on an excerpt for the Karabacon Scantool source code.

```
async def move(self):
    fut = [dev.move() for dev in self.devices]
    await gather(*fut, return_exceptions=True)
```

Here, *self.devices* is a list of *Device-Proxies*, essentially wrappers around a remote device in the distributed systems that expose the device as a local Python object. The devices here all adhere to a motor-interface, which guarantees that there is an asynchronous *move* slot, which can be *awaited* until the movement has completed. However, movements should happen concurrently on many motors, so in the first line we do not wait, but only trace *future* events. The *gather* method will then asynchronously wait until all future events have occurred. Practically, this means, that the listed *move* method will execute motions on multiple motors concurrently, and only return when all motions have completed.

A second major advantage of event-driven updates manifested itself in the seamless integration of the InfluxDB time series database [15] as the main data logging backend of Karabo [16]. InfluxDB's ingestion APIs are structured around the concept of non-regular time series data, and the underlying database model will efficiently store such data. InfluxDB, in combination with the Grafana front-end is the enabling technology for the Data Operation Center, where shift staff support all data-related services during operation. Additionally, InfluxDB drives the Recovery Portal, a tool which provides a time-machine like interface, through which entire beam lines can be configured to a previous point in time.

Language-Native APIs

Software, including SCADA systems, are initially often developed with certain target programming languages in mind. Over time, a motivation or requirement to support additional languages may arise. Frequently, binding techniques are then used. For the nowadays common case to expose C/C++ code in Python, these are, e.g., Cython [17] (Karabo Middelayer DOOCS binding), Python Ctypes (pyEpics [18], pydoocs [19], Boost::Python [20] (pyTango [21], Karabo) or PyBind11 [22] (future Karabo releases).

However, such bindings will often necessarily, and intentionally, closely reflect the specifics of the language that is being bound. This can lead to syntactic constructs one would otherwise not usually find in the target language. The following listings exemplify this by comparing a Karabo interface definition between C++ and Python Bound APIs of Karabo. For the Bound API the intention was to have syntactic similarity to C++ to facilitate developers developing in both APIs.

```
static void expectedParameters(
    karabo::util::Schema& expected) {

    STRING_ELEMENT(expected)
            .key("_serverId_")
            .displayedName("_ServerID_")
            .adminAccess()
            .assignmentInternal()
```

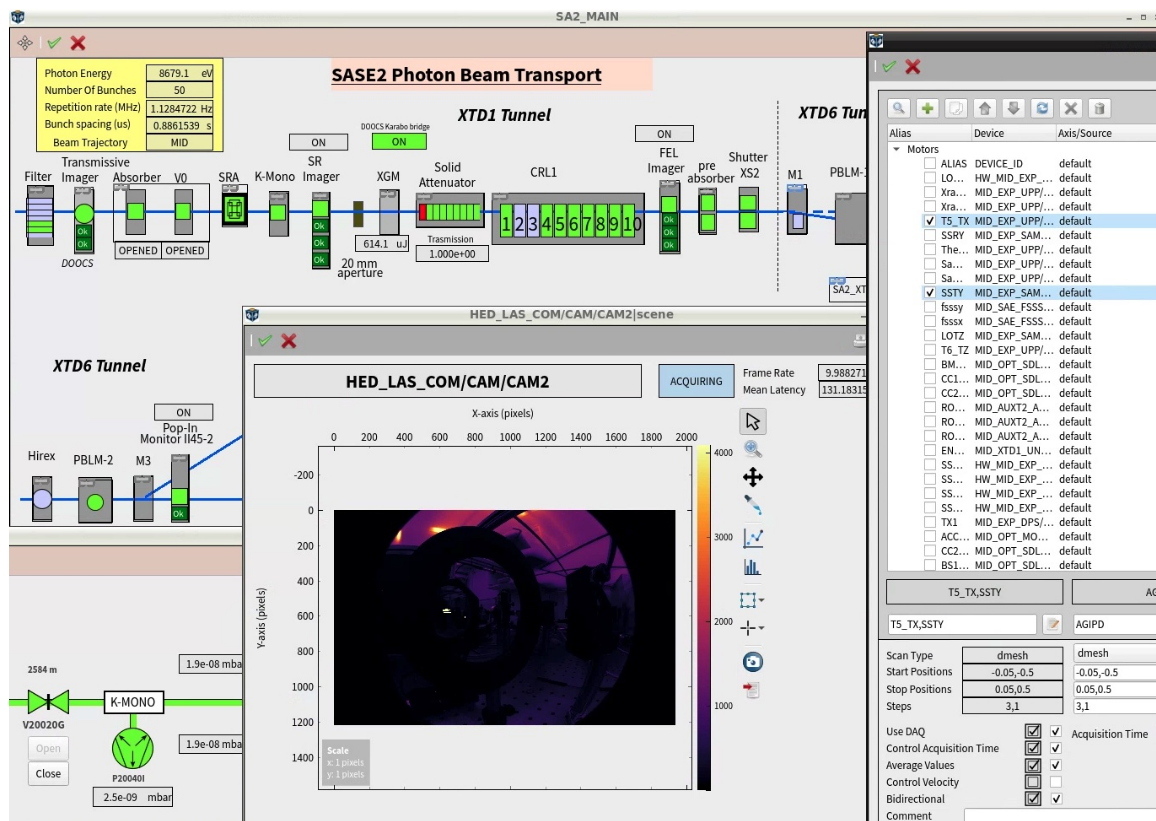

Figure 2: Karabo scenes showing an overview of the European XFEL SASE2 beam line, a vacuum section there-in, a camera, and the scantool source selection dialog.

```
            .noDefaultValue()
            .init()
            .commit();
    // ...

}

@staticmethod
def expectedParameters(expected):
    (
        STRING_ELEMENT(expected)
        .key("_serverId_")
        .displayedName("_ServerID_")
        .expertAccess()
        .assignmentInternal()
        .noDefaultValue()
        .init()
        .commit(),
    # ...
    )
```

The Karabo Middlelayer API uses an alternative approach. It is written solely in Python (when using the AMQP broker), and makes full use of Python's advanced asynchronous features (see Section , This facilitates a concise syntax that looks like Python, rather then a domain-specific language on top of Python, such that above code snippet in the Middlelayer API looks like this

```
_serverId_ = String(
    displayedName="_ServerID_",
    requiredAccessLevel=AccessLevel.EXPERT,
    assignment=Assignment.INTERNAL,
    accessMode=AccessMode.INITONLY,
    defaultValue="__none__",
    daqPolicy=DaqPolicy.OMIT)
```

.

The benefit is that a Python developer not familiar with Karabo can be expected to understand these syntactic constructs without much additional training, and as is detailed in Section , so will a Large Language Model trained on Python sources.

Coding-free Synoptic Views - The Karabo GUI

A distinguishing feature of the Karabo GUI is the ease with which scenes can be created to for synoptic views of what is controlled. No coding is required, and the basic steps are usually obvious to anyone who has worked with a presentation software like Microsoft Powerpoint: select the property to be added, drag it onto the scene, right-click select the appropriate widget, and then position it. By repeating these steps, synoptic views involving many devices, consisting of widgets ranging from lamp indicators to camera views can be created, and further styled.

At the European XFEL many operators make use Karabo's scene creation tools. This has the benefit, that operators can style views to there needs without expert support, even on an ad-hoc basis. At the same time programmatic scene models allow devices to ship with curated scenes, which are consistent over all instances of the same device class.

Content from this work may be used under the terms of the CC BY 4.0 licence (© 2023). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-TH1BCO06

Content from this work may be used under the terms of the CC BY 4.0 licence (© 2023). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI

The downside of such simplicity is that scenes are often created for "one-time" use, and not curated or maintained afterwards. Currently, the GUI does not track scene author or ownership, and accordingly verification of the continuing utility of a given scene is not always straight forward. We foresee to resolve this issue through authentication facilities that are currently being added to the GUI.

Scalability

Karabo's dynamic topology removes the need of a central database that defines the system. In operation, but even more for developers, this has the advantage that Karabo can run self-contained on a single host, and is quickly set up to do so. Accordingly, in the development workflow at European XFEL, devices usually see initial development on a standalone installation, are then added to continuous integration environments, and finally deployed into the production environment using an Ansible tool-chain [23]. Development on a production host, or even hot-patching a bug on such a host, is a discouraged exception.

Especially the Karabo middlelayer API also has low hardware requirements. In practice this has been utilized to install Karabo on SoCs like Raspberry PIs, in order to e.g. interface USB hardware. The Karabo Hash has been successfully decoded in Micropython [24] on ESP-32 microcontroller [25] based IoT prototypes.

LLM AI Agents and Karabo

The release of *ChatGPT* (Chat Generative Pre-trained Transformer) [26] in November 2022 resulted in a significant shift in the general public's perception of what artificial intelligence in form of Large Language Models (LLM) is capable of. The free-of-charge web interface lead to significant experimentation with the system, also by researchers. The GPT model series, currently available as version GPT4 are *foundational models*. Their training data set is so large, that they can be directed to fulfil many tasks including writing, and documenting code. Since Karabo has only been released to the public domain recently, it is fair to assume that before June 2023, the Karabo code base was not part of the training data sets of ChatGPT, or early GPT4 versions. Nevertheless, by giving sufficient context in the input prompt alone, the models have been shown to be well suited to document code implemented on-top of the Karabo framework, regardless of the Karabo API.

Specifically, we pass the source code to the model, and request it to output documentation in a diff format that is fashion such that line numbers are not required in the output. This is necessary, as currently LLMs models do not reliably count. The following listing is the documentation of a function used in the European XFEL data acquisition software. Here GPT4 was used to batch-document approx 20,000 lines of code, adding or updating documentation as required.

```
/**
 * @brief Method to get the train envelope for
 * a given train ID.
 *
 * This method retrieves the train envelope for
 * a given train ID from a container. If the
 * train ID is not found in the container,
 * a new train envelope is created and added
 * to the container.
 *
 * @param container: a map containing train
 * envelopes, indexed by train ID.
 * @param train_id: the ID of the train for
 * which to retrieve the envelope.
 *
 * @return A reference to the train envelope
 * for the given train ID.
 */
karabo::util::Hash& getTrainEnvelop(
EnvelopContainer& container,
const train_t& train_id);
```

Expanding on the documentation generation capabilities of GPT4, in a next step we attempted to have the model code a Karabo middlelayer device. These tests were done before the Karabo open source release in June 2023, so a reasonable assumption is, that the GPT4 model has not been trained with Karabo source code. Instead, all information was provided as a system prompt, consisting of approximately two screen pages of condensed documentation for the Middlelayer API, alongside meta-instructions to the model, such as to take user input by the letter, and to output a single self-consistent code block for all requests. Using a Jupyter notebook interface to the OpenAI API (which GPT4 had largely coded using the same strategy as well), an iterative approach for the following tasks was implemented:

- Given a user prompt, produce a Karabo Middlelayer device with the desired features
- In iterations, try instantiating this device, feeding exceptions back into the model, which was tasked to create updated code based on these exceptions.
- Write a unit test for this device, and in iterations debug the unit test.
- Given an additional prompt, create a device provided scene for the device, debug this in iterations, and display the result in the notebook. Here the LLM was given a distinct system prompt, which describes the Karabo scene model, and also defines that the underlying layout model is similar to SVG.

A short demo of this process can be found at [27]. As an example, Fig. 3 shows an AI-coded Karabo scene, running on an AI-coded device. The code and scene prompts describing this AI-generated implementation of a Kashiyama NeoDry pump in Karabo are given in the Appendix.

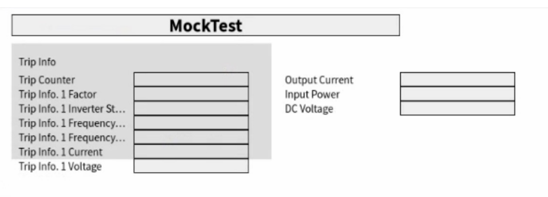

Figure 3: A Karabo Scene produced by the GPT4 Large Language Model.

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-TH1BCO06

While the produced code is certainly not production ready, we intend to investigate use cases in advanced code template generation. Our initial tests indicate that the Karabo Middlelayer API is especially well suited for AI code generation, as it is designed to avoid domain specific language constructs, and prefer a *Pythonic* coding style.

CONCLUSION

Karabo is a mature supervisory control and data acquisition system used for the photon systems and instruments at the European XFEL. The Karabo GUI is the main user interface when carrying out scientific experiments at the facility.

Karabo has been designed to meet the data acquisition requirements of large-scale research facilities: data can be correlated through a unique timing identifier, and high-performance data logging and acquisition systems exist. The event-driven nature of the system ensures that data traffic is minimized while significant changes are reliably propagated. We have discussed how these design choices benefit the facility, and developers implementing software in Karabo.

The system is freely available under a mixed MPL 2.0 and GPLv3 license at GitHub.com [5].

APPENDIX

Prompts used for the LLM-coded Karabo Middlelayer Device integrating a Kashyama Neodry pump.

Code: *Write a Karabo Middlelayer Device that monitors the readback parameters of a Kashiyama NeoDry vacuum pump. The middlelayer device connects over ethernet to the RS485 serial port on the pump. The communication protocol to read the pump parameters is Modbus-RTU. The middlelayer device should poll periodically the following read-only parameters of the pump: trip counter, Trip info. 1 Factor, Trip info. 1 Inverter status, Trip info. 1 Frequency (High), Trip info. 1 Frequency (Low), Trip info. 1 Current, Trip info. 1 Voltage, output current, input power, DC voltage. You provide code for all these parameters, no just stubs or a limited example set. These should be exposed as read-only Karabo properties. Remember that any Karabo property is camelCased The only user provided configuration inputs are the tcp address and port of the serial converter, and a polling interval in seconds. These too are Karabo properties. Please use pymodbus python module to implement rtu-over-tcp modbus.*

Scene: *Please group all trip info related parameters in a box that is labeled "Trip Info"*

REFERENCES

[1] M. Altarelli, "The European X-ray free-electron laser facility in Hamburg", *Nucl. Instrum. Methods Phys. Res. B*, vol. 269, no. 24, pp. 2845–2849, 2011. doi:10.1016/j.nimb.2011.04.034

[2] Tango, https://www.tango-controls.org/

[3] EPICS, https://www.epics-controls.org/

[4] DESY Object Oriented Control System (DOOCS), https://doocs.desy.de

[5] The Karabo SCADA Framework, https://github.com/European-XFEL/Karabo

[6] M. Kuster *et al.*, Detectors and calibration concept for the European XFEL, *Synchrotron Radiat. News*, vol. 27, no. 4, pp. 34–38, 2014. doi:10.1080/08940886.2014.930809

[7] M. Hapner, *Java Message Service API tutorial and reference: messaging for the J2EE platform*. Addison-Wesley Professional, 2002.

[8] RabbitMQ, https://www.rabbitmq.com/amqp-0-9-1-reference.html

[9] S. Vinoski, "Advanced message queuing protocol", *IEEE Internet Comput.*, vol. 10, no. 6, pp. 87–89, 2006. doi:10.1109/MIC.2006.116

[10] S. Hauf *et al.*, "The Karabo distributed control system", *J. Synchrotron Radiat.*, vol. 26, no. 5, pp. 1448–1461, 2019. doi:10.1107/S1600577519006696

[11] D. Goeries *et al.*, "The Karabo SCADA System at the European XFEL", *Synchrotron Radiat. News*, to be published.

[12] The Qt Company, https://www.qt.io

[13] W. Meier *et al.*, "eXist: An open source native XML database", in *NODe 2002: Web, Web-Services, and Database Systems*, 2003, 169-183. doi:10.1007/3-540-36560-5_13

[14] Pydantic, https://pydantic.dev

[15] InfluxDB, InfluxData, https://www.influxdata.com

[16] G. Flucke *et al.*, "Karabo Data Logging: InfluxDB Backend and Grafana UI", in *Proc. ICALEPCS'21*, Shanghai, China, Oct. 2021, pp. 56–61. doi:10.18429/JACoW-ICALEPCS2021-MOBL04

[17] S. Behnel *et al.*, "Cython: The best of both worlds", *Comput. Sci. Eng.*, vol. 13, no. 2, pp. 31–39, 2011. doi:10.1109/MCSE.2010.118

[18] M. Newville, "PyEpics: Python Epics Channel Access", Consortium for Advanced Radiation Sciences, University of Chicago, 2016.

[19] PyDOOCS, https://confluence.desy.de/display/DOOCS/Python+Client+Interface#PythonClientInterface-AboutPydoocs

[20] S. Koranne, "Boost c++ libraries", in *Handbook of Open Source Tools*, Boston, MA: Springer, 2011, pp. 127–143. doi:10.1007/978-1-4419-7719-9_6

[21] S. Rubio-Manrique *et al.*, "Dynamic Attributes and Other Functional Flexibilities of PyTango", in *Proc. ICALEPCS'09*, Kobe, Japan, Oct. 2009, paper THP079, pp. 824–826.

[22] Seamless operability between C++ 11 and Python, https://github.com/pybind/pybind11.

[23] Lorin Hochstein and Rene Moser, *Ansible: Up and Running: Automating configuration management and deployment the easy way*. O'Reilly Media, Inc., 2017.

[24] Charles Bell, *MicroPython for the Internet of Things*. Springer, 2017. doi:10.1007/978-1-4842-3123-4

[25] A. Maier *et al.*, "Comparative analysis and practical implementation of the ESP32 microcontroller module for the internet of things", in *2017 Internet Technol. Appl. (ITA)*, Wrexham, UK, 2017, pp. 143–148. doi:10.1109/ITECHA.2017.8101926

[26] ChatGPT release notes, https://help.openai.com/en/articles/6825453-chatgpt-release-notes

[27] Karabo AI Coding, https://syncandshare.xfel.eu/index.php/s/kt6NbSjJfMg7Pf5

Content from this work may be used under the terms of the CC BY 4.0 licence (© 2023). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI