ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-WECPR03

STATUS OF THE KARABO CONTROL AND DATA PROCESSING FRAMEWORK

G. Flucke*, N. Al-Qudami, M. Beg, M. Bergemann, V. Bondar, D. Boukhelef, S. Brockhauser[1], C. Carinan, R. Costa, F. Dall'Antonia, C. Danilevski, W. Ehsan, S. G. Esenov, R. Fabbri, H. Fangohr[2], D. Fulla Marsa, G. Giovanetti, D. Goeries, S. Hauf, D. G. Hickin, E. Kamil, Y. Kirienko, A. Klimovskaia, T. A. Kluyver, D. Mamchyk, T. Michelat, I. Mohacsi, A. Muennich, A. Parenti, R. Rosca, D. B. Rück, H. Santos, R. Schaffer, A. Silenzi, K. Wrona, C. Youngman, J. Zhu
European XFEL GmbH, Schenefeld, Germany
[1]also at Biological Research Center of the Hungarian Academy of Sciences, Szeged, Hungary and at University of Szeged, Szeged, Hungary
[2] also at University of Southampton, Southampton, United Kingdom

Abstract

To achieve a tight integration of instrument control and (online) data analysis, the European XFEL decided in 2011 to develop Karabo, a custom control and data processing system. Karabo provides control via event-driven communication. Signal/slot and request/reply patterns are implemented via a central message broker. Data pipelines for e.g. scientific workflows or detector calibration are implemented as direct TCP/IP connections. The core elements of Karabo are self-describing devices written in C++ or Python. They represent hardware, orchestrate other devices, or provide system services like data logging and configuration storage. To operate Karabo, a Python command line interface and a generic GUI written in PyQt are provided. Control and data widgets compose Karabo scenes that are provided by devices or are manually customized and stored together with device configurations in a central database. Since 2016, Karabo is used to commission and operate the currently three photon beam lines and six scientific instruments at the European XFEL. This contribution summarizes the status of Karabo, highlights achievements and lessons learned, and gives an outlook for future directions.

INTRODUCTION

The European X-ray Free Electron Laser (EuXFEL) facility provides hard and soft X-ray beams via three photon beamlines to six instruments. Up to 27,000 photon pulses per second are arranged into 10 Hz trains of pulses at 4.5 MHz. High-repetition-rate, large-area 2D imaging detectors capable of detecting images of scattered photons produced by a single XFEL photon pulse create very high data rates. Detector data needs to be calibrated on-the-fly with low latency to provide feedback to the experiment control.

In view of these requirements it was decided that a new distributed control system, Karabo, with integrated data acquisition and workflow capabilities should be designed and developed. Hence, Karabo has been developed since early 2012 [1] and has been in use since September 2017 to enable scientific user experiments at the EuXFEL [2].

* gero.flucke@xfel.eu

KARABO IN A NUTSHELL

Karabo is designed to provide supervisory control and data acquisition for the European XFEL. Hardware devices, system services, and control procedures are represented by Karabo software devices of which many can run within the same server process, distributed among various control hosts. Devices are self-describing their properties, commands, configuration possibilities, and their availability depending on the *state* of the device. This description can be expanded at run-time, e.g. according to discovered hardware details. Devices can expose that they have specific *capabilities* like providing scenes or macros (see section on user interfaces) or *interfaces*. An interface, e.g. as a motor or a camera, defines a set of commands and properties.

Control communication is routed via a central broker. Currently, Karabo uses the the Java Messaging Service (JMS) broker [3] that can be clustered. Large data from detectors is transported via data pipelines implemented as direct TCP connections. A graphical and a command line interface provide flexible ways to interact with a Karabo system that is defined by a specific communication topic on the broker. Temporary procedures can be implemented as macros that run centrally on dedicated macro servers. Karabo's graphical user interface (GUI) connects into the system via a TCP connection to a gui server device.

An illustration of a Karabo system is shown in Fig. 1.

KARABO COMMUNICATION

A unique id identifies each object in a Karabo installation. Uniqueness is ensured when registering to the installation. Any message is routed via the broker according to this id. The header of a message contains the name of a *slot*, i.e. a method of the object that has been registered to be callable remotely.

Two broker communication patterns are implemented: In the *request/reply* case a (remote) slot is called and its success or failure of execution received. Up to four slot arguments

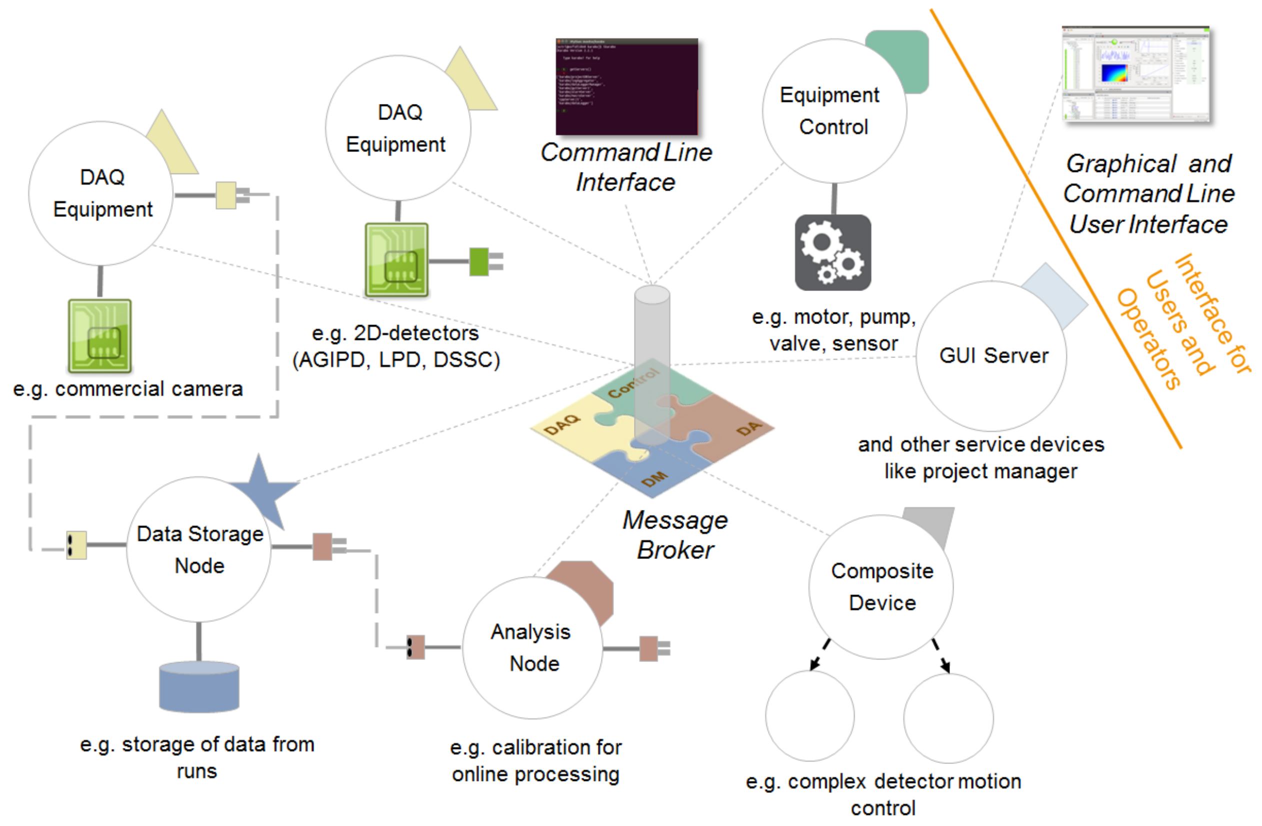

DAQ Equipment
e.g. commercial camera
DAQ Equipment
e.g. 2D-detectors (AGIPD, LPD, DSSC)
Command Line Interface
Equipment Control
e.g. motor, pump, valve, sensor
Graphical and Command Line User Interface
Interface for Users and Operators
GUI Server
and other service devices like project manager
Control
DAQ
DA
DM
Message Broker
Data Storage Node
e.g. storage of data from runs
Analysis Node
e.g. calibration for online processing
Composite Device
e.g. complex detector motion control

Figure 1: A Karabo installation showing Karabo devices with various tasks. Broker and pipeline communication lines are indicated.

and return values are supported. These can be any serialisable data type (see Table 1) or the Karabo hash (see below). In the *signal/slot* case a slot is subscribed to the *signal* of any other object. If the signal is emitted, all subscribed slots are called with the up to four arguments of the signal. Given this *publish/subscribe* mechanism, Karabo is designed to be event-driven and regular polling of properties is not needed.

By policy, slots with arguments are reserved for system level communication. Commands that a device receives are slots without argument or explicit return value. So the Graphical User Interface (GUI) can represent any command by a simple button. Commands implicitly return the state that the device is in after slot execution.

The described broker communication is complemented by Karabo's pipeline system to form data workflows. Pipelines directly connect devices via TCP to transport large data items like camera images. Data that a device writes to its pipeline *output channel* is sent to other devices that have their *input channels* configured to receive that data. Although data is sent only when an input channel reports readiness to receive more data, buffering ensures that data item $N + 1$ can be transmitted while item N is being processed.

Flexible configuration possibilities support workflows with different purposes like low latency online data calibration, reduced rate previews, parallelisation of computing intense steps, and complete offline analysis:

- input channels can receive a *copy* of all data items or *share* the items with other channels to distribute the load,
- an input channel can receive data of several outputs, e.g. to collect from shared processing,
- there are several ways an output channel should react in case that it is faster than the receiving input channels: *drop* items, *queue* them, or *wait* until the input channel is ready,
- for a reduced rate preview, readiness to receive the next data item can be artificially delayed.

KARABO IMPLEMENTATION

"We distinguish between the *Karabo framework* and *Karabo devices*; where devices realise a particular functionality through use of the Karabo framework. Besides test devices, the framework contains a few devices to provide system services. The object oriented Karabo framework is implemented in C++11 and Python 3"[2]. Devices can be implemented in three application programming interfaces (APIs).

The *C++ API* is the suggested API for low-level interaction with hardware or performance critical devices. Most of the system service devices are implemented using this API. The C++ server uses a multi-threaded event loop. Since it starts devices as part of its single process, inter-device communication on the same server can bypass the broker. Communication patterns are generally provided in a synchronous and an asynchronous way, but for performance reasons the use of the synchronous interface is generally discouraged.

The *bound Python API* exposes the C++ API functionality via the Boost C++ libraries [4] and their `boost::python` bindings to the Python programming language. Its feature set and function signatures mirror those of the C++ API, allowing programmers to easily transition between APIs.

The *middlelayer API* is natively implemented in Python, except that it relies on the openMQ(C) library for broker communication. It has "no dependencies on the other two APIs, and with the intention of being a pythonic interface, following Python conventions and standards. This API offers device proxies to comfortably control other software components and is the recommended API for implementing composition and aggregation of multiple devices. Cooperative multi-tasking is implemented using Python's `asyncio` library providing a central event loop ensuring in-order execution of tasks. Karabo's macro scripting has been developed on top of this API."[2]

The Karabo GUI is developed in PyQt and re-uses a fraction of the middlelayer implementation. This pure Python approach simplifies portability and the Karabo GUI is supported both on Linux and Windows.

Karabo Hash

"Karabo's basic data structure is the so-called Karabo *hash*. It is a hierarchical key/value container supporting element-specific attribute assignment (also as key/value pairs) and preserving insertion order. Keys are unique strings that may contain a separator character, indicating nodes in the hierarchy. The default separator is the dot (.), and thus a key 'this.is.karabo' would refer to a leaf 'karabo' located under the subnode 'is' of the top-level node 'this'. The values can take any type, but serialisation of a Karabo hash is restricted to the types listed in"[2] Table 1 as well as composite data types such as image and multi-dimensional array (`ndarray`) data. "Serialization is supported to XML, HDF5 and ZeroMQ as well as to a proprietary binary format to be used for communication within Karabo."[2] Serialisation to and deserialisation from this binary format is tuned to avoid any copy of image and array data.

Table 1: Plain C++ Data Types Used by Karabo Serialisation. Vectors thereof are also supported [2].

Boolean	`bool`
Integer	`char (raw data),`
	`(un)signed char (int8),`
	`(unsigned) short (int16)`
	`(unsigned) int (int32),`
	`(unsigned) long long (int64)`
Float	`float (float32), double (float64)`
Complex	`complex<float>, complex<double>`
String	`string`

Unified States

To ease implementation, Karabo does not require that devices implement a strict finite state machine. Nevertheless, all devices must define a list of states as a subset of Karabo's unified list of states. Figure 2 shows Karabo's base states with their inheritance tree and the colours used to represent the states in the GUI. Many more states inherit from these base states. Inheritance means further specification and inheritance of the representing colour if not already specified. Commands are intended to trigger the transition from one state to another. The self-description of the device defines which commands the device accepts and which properties can be changed, depending on the state of the device. All APIs provide mechanisms to aggregate the states of several devices, i.e. to define the most significant of them.

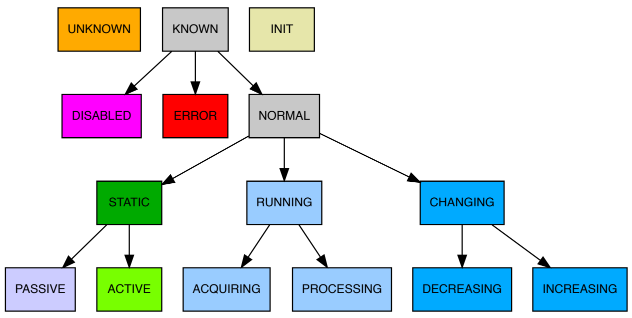


Figure 2: Overview of Karabo's basic unified states and their inheritance relation to one another [2].

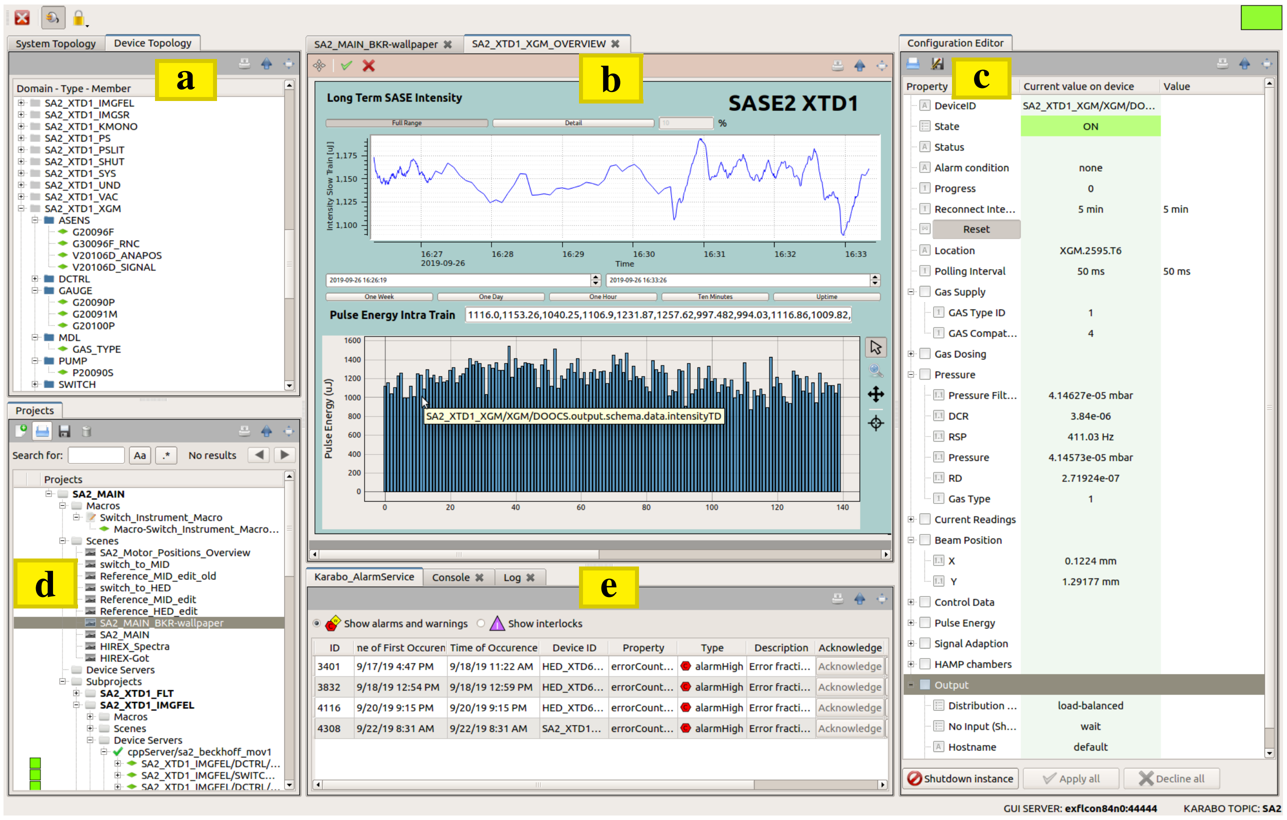
USER INTERFACES AND MACROS

The use and control of a Karabo system is facilitated by a generic GUI and a command line interface (CLI). The CLI is named iKarabo as it is a light customisation of the IPython shell and directly interacts with the broker. The interaction of the GUI with the system is mediated by a TCP connection to a dedicated GUI server device and thus allows remote access via SSH tunnelling. The GUI server device also takes care to shield the clients from data rates higher than what they can process or what the human eye can follow. By default, updates are sent to clients with a maximum rate of 2 Hz. For the usually big pipeline data, the client has in addition to confirm that it processed it before the next update will be sent.

The GUI is a multi-purpose application with detachable panels (see Fig. 3) to

- acquire an overview of all servers, devices and alarm notifications of a Karabo system,
- configure, instantiate and operate devices,
- edit and run macros,
- edit and view *scenes* as customiseable collections of graphical elements to intuitively display and if desired also modify properties, including data from Karabo pipelines,

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-WECPR03



Property	Current value on device	Value
DeviceID	SA2_XTD1_XGM/XGM/DO...	
State	ON	
Status		
Alarm condition	none	
Progress	0	
Reconnect Inte...	5 min	5 min
Reset		
Location	XGM.2595.T6	
Polling Interval	50 ms	50 ms
Gas Supply		
GAS Type ID	1	
GAS Compat...	4	
Gas Dosing		
Pressure		
Pressure Filt...	4.14627e-05 mbar	
DCR	3.84e-06	
RSP	411.03 Hz	
Pressure	4.14573e-05 mbar	
RD	2.71924e-07	
Gas Type	1	
Current Readings		
Beam Position		
X	0.1224 mm	
Y	1.29177 mm	
Control Data		
Pulse Energy		
Signal Adaption		
HAMP chambers		
Output		
Distribution ...	load-balanced	
No Input (Sh...	wait	
Hostname	default	

ID	ne of First Occuren	Time of Occurence	Device ID	Property	Type	Description	Acknowledge
3401	9/17/19 4:47 PM	9/18/19 11:22 AM	HED_XTD6...	errorCount...	alarmHigh	Error fracti...	Acknowledge
3832	9/18/19 12:54 PM	9/18/19 12:59 PM	HED_XTD6...	errorCount...	alarmHigh	Error fracti...	Acknowledge
4116	9/20/19 9:15 PM	9/20/19 9:15 PM	HED_XTD6...	errorCount...	alarmHigh	Error fracti...	Acknowledge
4308	9/22/19 8:31 AM	9/22/19 8:31 AM	SA2_XTD1...	errorCount...	alarmHigh	Error fracti...	Acknowledge

Figure 3: Karabo's graphical user interface. The tool tip shows the device property displayed by the widget.

- store device configurations, macros and scenes in the project data base,
- and run a CLI.

The *navigation panel* [Fig. 3(a)] offers two views of the Karabo topology. The four levels of the tree of the searchable and filterable *system topology* are the computer hosts, the Karabo servers they run, the devices classes that these servers are aware of and the running devices of these classes. The *device topology* shown in the figure is adapted to the EuXFEL device naming convention that requires device ids to be composed of three parts separated by slashes '/'. A tree view of all devices is displayed, with three levels representing the three parts of the ids.

The *configuration editor panel* [Fig. 3(c)] shows all properties and commands of a selected device and gives access to their descriptions, types, default values, alarm limits, value ranges, and time stamps. For online devices, commands and property reconfigurations can be applied if the current device state does not prohibit that.

"The *project panel* provides access to the database of available projects" and to edit them. "Once a project has been selected and loaded, the panel [Fig. 3(d)] shows the components of a project: subprojects (in bold face), macros, scenes, device servers, and one level down devices and device configurations." [2]

The *notification panel* [Fig. 3(e)] is subdivided into a number of tabs to display log messages, to give an overview of currently active alarms (see below), and to provide access to a CLI running on the macro server.

The central panel [Fig. 3b)] is dedicated to *scenes* and *macros*.

"A Karabo scene is a collection of graphical elements to intuitively display and if desired also modify properties. A rich set of widgets are provided by Karabo, including state-aware coloured icons, trend lines, spark lines, bit fields, XY-plots, analogue gauges, knobs, sliders and image displays" [2] as well as command buttons and links to other scenes. Since the development cycle of widgets dedicated to special procedures may be shorter than that of the Karabo GUI, it is possible to extend the list of widgets by loading so-called GUI-extensions at run-time. "A scene can be created by dragging-and-dropping properties and commands from the configuration panel into the desired locations. This is called the design mode. When the design of a scene is completed, a scene can be locked so that type, position and geometry of widgets cannot be modified any further. This is referred to as the control mode and is the default mode for all SCADA operations." [2] Besides being edited in the GUI, scenes can also be provided by devices that have the according capability.

Macros are meant for automation of recurring tasks. They are executed remotely on the dedicated macro server. Their standard output is captured and displayed in the bottom part of the macro panel.

Besides the generic GUI, Karabo provides executables to open selected panels or scenes only.

SYSTEM SERVICES

The backbone of a Karabo installation consists of several system services that are implemented as devices or are dedicated servers. The GUI server device and the macro server have been mentioned above. A project database manager device encapsulates the communication to the database used to store projects. Further services are detailed out below.

Data Logging

To understand any incident in a controlled system, it is of utmost importance to record the properties of all devices and to make their history easily available. Karabo's data logging is implemented by a manager device that distributes the tasks to specific logger and reader devices that run on a configurable number of servers. The servers can be run on individual hosts to share the load.

Both GUI and CLI provide access to historic data in two ways: all value updates of a single device property in a given range in time or all properties of a device at a given point in time. Currently, logged data is written to text files per device. To speed-up reading, custom index files are created per property. A prototype exists that replaces the file based backend by a time series data base [5].

Data Acquisition

Selected pipeline data and the properties of selected devices are stored for dedicated runs by the data acquisition (DAQ) of the EuXFEL [6]. The data is stored in HDF5 files and indexed according to the 64-bit id of the train it belongs to. The indexing eases the correlation of data from different sources.

The data of the big custom made detectors is sent to the DAQ by 16 UDP based data streams, following the XFEL Train Data Format protocol. The DSSC detector will provide up to 800 images per train and thus data rates of 16 GB/s. So far 600 images per train have been reliably stored. Online monitoring of this data is made possible since the DAQ provides data of a subset of the trains as Karabo pipeline output.

The DAQ is implemented by Karabo devices that are not part of the Karabo framework. This allows an independent development cycle.

Alarm System

Karabo provides an integrated alarm notification system. "Property-related alarm thresholds can be hard-coded or configured at device initialisation time for scalar values. These are evaluated at each property update on the device, resulting in a new value $v(t)$, such that for normal operations

$$T_{\mathrm{alarm}_{low}} \leq T_{\mathrm{warn}_{low}} \leq v(t) \leq T_{\mathrm{warn}_{high}} \leq T_{\mathrm{alarm}_{high}}.$$

If the quantity $v(t)$ goes beyond the low or high warning thresholds, the distributed control system notifies of the warning condition.... Alarms can be defined to require acknowledgement, i.e. their notifications will not silently disappear if the condition triggering the alarm passes."[2]

Devices also have a global alarm condition that can be set explicitly through device logic. It automatically evaluates to the highest alarm condition of all property related alarms and any explicit assignment. Besides the `WARN` and `ALARM` levels, it can also take the `INTERLOCK` value that reflects such a condition of an interlocking hardware.

An alarm service device keeps track of all active alarm conditions and the dates of their first and most recent occurrences.

Scanning

Since still in active development, to quickly follow the needs of the EuXFEL instruments, Karabo's scanning functionality is implemented as a device outside of the framework. Through device provided scenes it is well integrated with the GUI, but it can also be steered via the CLI. The device relies heavily on device interfaces: The scanned axes are the positions of one or several *motors*. *Trigger sources* have to comply with the camera interface to be started and stopped. Furthermore, up to six properties of any kind of devices can be specified as *data sources*, e.g. processor devices that calculate numbers from the data produced by the trigger sources. The capabilities to preview data sources with respect to the scanned motor positions are currently extended to support not only scalar values, but also vectors. Similar to SPEC [7], absolute (`ascan, a1scan`, etc.) and relative scans (`dscan`, etc.) as well as `mesh` scans are supported. Continuous scans are in an experimental state. The scan device is also well integrated with the DAQ which it configures, starts and stops.

Due to the interaction with Karabo devices complying with interfaces, the chosen approach is very flexible and complex scan patterns can be easily achieved by preparing virtual motor or specialised processor devices.

ACHIEVEMENTS

The Karabo control and data processing framework is used to control the photon beam lines and scientific instruments of the European XFEL since early commissioning end of 2016. Meanwhile all six instruments have started user operation. As of September 2019, almost 14,000 Karabo devices are running at the EuXFEL and expose more than 1.6 million control points, i.e. properties and commands. The devices are distributed among 12 broker topics that usually resemble an instrument or a beamline. The few cases where cross talk between topics is needed, special devices duplicate information from one topic to another using a connection to a GUI server of the other topic.

A significant improvement with regards to X-ray beam drift has been achieved by implementing a feedback mechanism based on a closed loop PID control algorithm in a Karabo device [8]. Depending on the signal of a chosen diagnostic detector, X-ray optical elements are aligned.

The DAQ stored almost 9 PB of data of which about 6 PB belong to experiments of external users. Online preview of a fraction of the data produced by the big customised X-ray detectors (AGIPD, LPD, DSSC) can be processed with a rate of 1.792 GB/s. Special *Karabo bridge* devices complete the involved calibration pipelines to send the data out of Karabo via ZeroMQ such that user programmes can analyse it [9]. The latency from acquisition to ZeroMQ output is below 2 s.

Deployment of new framework releases in the control network is usually scheduled during shutdowns of the accelerator, i.e. eight times in 2018 and 2019. Release integrity is ensured by the high code coverage of the unit and integration tests run during the development cycle, i.e. 67% and 74% of the C++ and Python code base, respectively. In addition, a dedicated release test cycle runs when finalising a release and covers device code outside the framework as well.

LESSONS LEARNED

In early versions of Karabo development, serious delays arose especially for big C++ servers hosting many hundreds of devices or for pipelines with a high data throughput as used for the online calibration.

The key ingredient to speed-up the big servers was to avoid any blocking function calls on the common event loop to prevent thread starvation. That means that in general the asynchronous API for the request/reply pattern in inter-device communication have to be used. Also, a mechanism needed to be developed that enables a slot to delay its reply to potentially run in another thread. This avoids blocking when the reply requires a further request, e.g. to hardware.

If the Karabo C++ framework were to be rewritten from scratch, one could provide simpler support for asynchronous programming, e.g. by the use of coroutines as are fundamental to the success of the middlelayer interface, written in single-threaded Python.

Although Karabo provides a C++ data container (“`NDArray`”) for big array data that can adopt or view raw memory and avoids data copies when binding to the `numpy.ndarray` in Python, originally the data was copied when serialising and de-serialising: A common buffer for the `NDArray` and its meta data was used. The delaying data copies are now avoided by adapting the interface of the serialisation routines.

A problem that is noticed only under system stress is that by simply posting messages on a multi-threaded event loop, their execution order can get lost. Instead, first-in first-out queuing is needed.

Since the Karabo system topology is fully dynamic, i.e. no object knows which devices etc. exist, Karabo relies on broadcast messages received by all objects, in particular “instance new” or “gone” messages. But broadcast messages scale badly between two servers with many hundreds of devices each: If one server starts N devices and another server has M devices up running, the “new” messages of the N devices will be send to each of the M devices and their server, resulting in $N \cdot (M + 1)$ messages. For $M = N = 500$ this are 250,500 messages that pose a big load on the broker and on the deserialisation on the receiver side that can lead to delays of a minute. To overcome this, broadcast messages are now only sent to the server that internally distributes them to its M devices without any deserialisation overhead.

FUTURE DIRECTIONS

The primary goal of the Karabo framework is to serve a smooth operation of the user experiments at the six EuXFEL instruments. To further strengthen this support, four areas of improvements for Karabo are currently discussed:

The text file based data logging backend shall be replaced by using a time series database, namely `influxDB` [5]. This will ease data exploration, e.g. using external tools.

Storing device configurations in projects is a very flexible approach. While being beneficial in small installations, a centrally managed configuration storage has been identified as better applicable for fully comissioned components.

While Karabo’s design foresees an authentication and authorisation mechanism to provide access restrictions, this is not implemented yet. Now that more and more parts of the system are stable, this becomes a limitation.

While the operation of the JMS broker turned out to be very smooth, the openMQ(C) client library lacks an asynchronous interface and is not well maintained. Therefore, brokers supporting the AMQP and MQTT protocol have been investigated. Especially MQTT provides features that could be very beneficial for Karabo like retained messages and the last will and testament. For the Karabo C++ API, a prototype of the core communication class using the MQTT protocol has already been implemented.

In parallel to the outlined improvements, the clean-up of the dependencies to solve issues with conflicting open source licences (namely GNUv2 and Apache 2.0) shall be finished. That will allow to release Karabo to the public with an open source licence as planned since the beginning. As a result of the so far achieved refactoring, the Karabo GUI can already be installed into a Conda [10] environment.

SUMMARY

Due to the specific needs to integrate experiment control with workflow capabilities and high data rates, the European XFEL decided to develop a new control system, Karabo. Its core entity is the self-describing device that represents hardware, is a workflow node, implements a system service or orchestrates other devices. Control communication is routed via a central broker. For workflows, data is sent through flexible pipelines that use direct TCP connections. All this is available through the generic GUI or the CLI.

Since 2016 Karabo is used to commission the facility and to successfully conduct user experiments. The development had to overcome few shortcomings to avoid severe delays: Karabo nowadays communicates mainly using asynchronous interfaces and avoids any unnecessary data copies. Further improvements are planned in the areas of data logging, storage of device configurations, and authentication

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-WECPR03

and authorisation. Finally, Karabo is planned to be released to the public using an open source software licence.

REFERENCES

[1] B. Heisen *et al.*, "Karabo: An integrated software framework combining control, data management, and scientific computing tasks," in *Proc. ICALEPCS'13*, San Francisco, CA, USA, 2013, pp. 1465–1468.

[2] S. Hauf, B. Heisen, *et al.*, "The Karabo distributed control system," *J. Synchrotron Rad.*, vol. 26, pp. 1448–1461, 2019. doi:10.1107/S1600577519006696

[3] M. Hapner, R. Burridge, R. Sharma, J. Fialli, and K. Stout, "Java message service," *Sun Microsystems Inc.*, p. 9, 2002.

[4] B. Schäling, *The Boost C++ Libraries*. XML Press, 2011.

[5] InfluxData Inc., *InfluxDB*, https://docs.influxdata.com/influxdb.

[6] D. Boukhelef, J. Szuba, K. Wrona, and C. Youngman, "Software development for high speed data recording and processing," in *Proc. ICALEPCS'13*, San Francisco, CA, USA, 2013, pp. 665–668.

[7] Certified Scientific Software, *Spec*. https://certif.com/content/spec

[8] V. Bondar *et al.*, "Beam position feedback system supported by Karabo at European XFEL," presented at ICALEPCS'19, New York, NY, USA, 2019, paper MOPHA040, this conference.

[9] H. Fangohr *et al.*, "Data analysis support in Karabo at European XFEL," in *Proc. ICALEPCS'17*, Barcelona, Spain, 2017, pp. 245–252. doi:10.18429/JACoW-ICALEPCS2017-TUCPA01

[10] Continuum Analytics, *Conda*. https://docs.conda.io/projects/conda/en