KARABO: AN INTEGRATED SOFTWARE FRAMEWORK COMBINING CONTROL, DATA MANAGEMENT, AND SCIENTIFIC COMPUTING TASKS

B. C. Heisen*, D. Boukhelef, S. Esenov, S. Hauf, I. Kozlova, L. Maia, A. Parenti, J. Szuba, K. Weger, K. Wrona, C. Youngman, European XFEL GmbH, 22761 Hamburg, Germany

Abstract

The expected very high data rates and volumes at the European XFEL [1] demand an efficient concurrent approach of performing experiments. Data analysis must already start whilst data is still being acquired and initial analysis results must immediately be usable to re-adjust the current experiment setup.

We have developed a software framework, called Karabo, which allows such a tight integration of these tasks (see Fig. 1). Karabo is in essence a pluggable, distributed application management system. All Karabo applications (called devices) have a standardized interface for self-description/configuration, program-flow organization (state machine), logging and communication. Central services exist for user management, access control, data logging, configuration management etc. The design provides a very scalable but still maintainable system that at the same time can act as a fully-fledged control or a highly parallel distributed scientific workflow system. It allows simple integration and adaption to changing control requirements and the addition of new scientific analysis algorithms, making them automatically and immediately available to experimentalists.

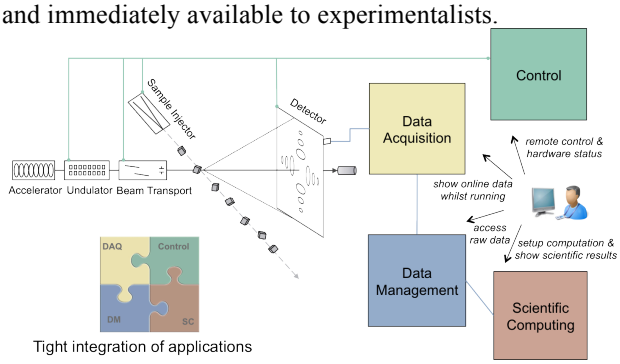

Figure 1: A homogenous software framework.

TECHNOLOGY

Karabo is written in C++ making extensive use of template programming mechanisms and the boost [2] libraries. Karabo and all its components are fully available to the Python programming language. This is achieved using a mixture of language binding mechanisms (boost-python) and complete re-writes of selected components in pure Python. Karabo runs on current Linux operating systems (e.g. Ubuntu, Scientific Linux, SUSE, Fedora) and under Mac OS X and is distributed as a software bundle (binary or sources) including all necessary dependencies.

*burkhard.heisen@xfel.eu

ESSENTIAL COMPONENTS

Devices and Device-Servers

Similar to existing successful control systems such as Tango [3] or DOOCS [4], the basic building blocks of Karabo are controllable objects (devices) managed by a device-server. A device-server is a generic application, which is capable of loading device-class libraries at runtime (plugin-functionality). Device-servers and devices are identified by unique name strings and communicate via a central message broker.

Upon an initialization request, devices are constructed and initially configured using factory mechanisms. After instantiation the device-server keeps devices running in an event driven way. Events pertaining to a device are given by its properties and commands. Properties have "get/set" and commands have "execute" functionality. Events can optionally be integrated into a finite-state-machine, which predefines possible callback-sequences through a device internal state transition table. The base-device interface provides functionality ensuring a standardized self-description of all available properties and commands and their associated state-machine logic.

A large set of meta-information (called attributes) is available to detail the description of properties and commands. Typical attributes for example are: default value, physical unit, displayed label, value-bounds, alarm and warn thresholds, access mode (initially configurable, reconfigurable, read-only), etc.

Device-Client

Device-client objects can remotely control devices and device-servers. Available functionality is "set/get/monitor" on device properties and "execute" on device commands. Devices can remotely be instantiated on any device-server equipped with the respective device-class plugin. Device and device-server instances may also be remotely terminated. All functionality comes in a "wait" (synchronous, return value provided) and in a "no-wait" (asynchronous) version allowing for sequential or parallel control, respectively. As Karabo's communication is inherently event-driven, any synchronous interface makes use of local caching approaches and does not poll the remote component.

The device-client also provides functionality for exploring the distributed system topology (i.e. which device-servers run on what hosts with what device-class plugins available, which devices are running on what

ISBN 978-3-95450-139-7

device-server etc.). The self-description of each device is available to the device-client and utilized for inquiring about all available properties and commands.

Device-client instances run as engines behind all user interfaces of Karabo, they power the command line (CLI) as well as the graphical (GUI) interface. Moreover, each device optionally can run an internal instance of a device-client, enabling devices to sub-control other devices. In this way it is possible to introduce logical devices that can abstract several lower-level devices into one reduced interface (device-composition).

User Interfaces (CLI and GUI)

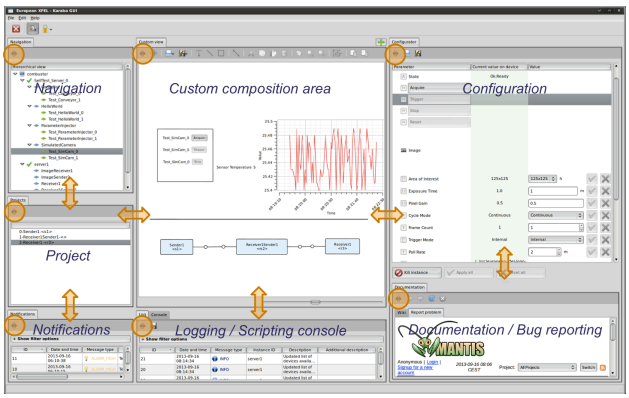

Figure 2: Multi-purpose GUI. Each panel may be undocked and made full screen on different monitors (orange circles).

Being a distributed system, the user can interact with Karabo from any connected computer. On the non-graphical side, an IPython [5] based command-line interpreter running the device-client is available. Any control commands can be typed and are directly executed. The interactive session is aided by sophisticated auto-completion on currently available device-servers, devices, their properties and commands, taking into account the restrictions of the individual device's state-machines. Any interactively given command can as well be written into a regular IPython script and be executed as a macro. This enables the user to write complex command sequence scripts, as for example those needed for slowly ramping up voltages or scanning physical properties.

Like the CLI the GUI is a multi-purpose tool providing all functionality needed to set-up and interact with the Karabo system. Seven dock-able main panels form the basic layout (see Fig. 2):

1. *Navigation* – Real-time (online) view of the system (device-servers, plugins, devices, etc.)
2. *Project* – Offline configuration of projects (e.g. compute workflows, initial configurations, etc.)
3. *Notification* – Informs about warnings, alarms, end-of-runs, etc.
4. *Custom* – PowerPoint like page allowing to build expert panels from a mixture of online (e.g. device properties/commands) and layout widgets (textboxes, lines, shapes, etc.)
5. *Logging* – Accumulated logging information of all currently running devices
6. *Configuration* – Auto-generated panel (utilizing the device's self-description) giving access to properties/commands of the currently clicked device
7. *Documentation* – Integrated internet-browser allowing wiki-page editing for device documentation and also bug-reporting

The GUI is implemented in Python using the PyQt [6] framework. GUIs do not directly communicate with the broker, but connect via TCP to a GUI-Server device, which forwards the communication to the broker. This architecture facilitates a server side pre-processing of data reducing each client's CPU time (as done once for all clients). Whilst running, the GUI always tracks which properties are currently visible and reports this information to the GUI-Server, which then only sends data that is currently needed (visible), largely reducing the network load.

Message Broker

The message broker is a central service to which all Karabo components (device-servers, devices, clients) must connect for message exchange (Fig. 3). The message broker must be started as the first component in a Karabo installation and forms a single-point of failure. Conceptually, the broker must at least support publish/subscribe and message selection functionality. Karabo currently uses OpenMQ [7], which is an implementation of the JMS specification and also runs as integral part of the Glassfish [8] application server.

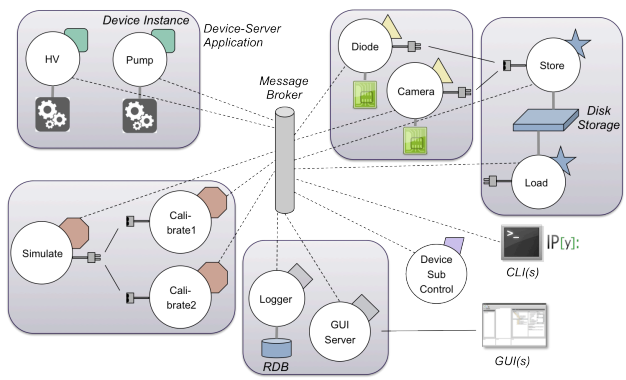

Figure 3: Karabo's basic architecture and components. Gray rectangles indicate device-servers, circles show devices of different category (green: control, yellow: DAQ, blue: DM, red: SC). Dashed lines indicate message-based control whilst solid ones indicate direct point-to-point communication. For simplicity device-server – broker communication lines are not shown.

SELECTED FEATURES

The Hash Object

The hash is the core data structure in the Karabo software framework. It is as fundamental to Karabo as the

ISBN 978-3-95450-139-7

inbuilt dict type is for the Python language. Like the Python dict the hash is a fully recursive key-value associative container (keys are strings and unique, values can be of any type). In addition, it preserves insertion order, whilst still being optimized for random key-based lookup. Different iterators are available for each use case. Individual keys within the hash can be optionally annotated with a set of attributes. The hash can seamlessly be serialized to/from e.g. XML, Binary, HDF5, JMS-Message formats and is used in the implementation of various Karabo backend functionality.

Signals and Slots

Signals and slots is a language construct originally introduced in Qt [9] for communication between objects, which makes it easy to implement the Observer pattern [10] while avoiding boilerplate code. We have re-implemented this communication pattern in Karabo with some substantial enhancements.

a) Object communication cannot only happen within one application (as in Qt) but is possible across applications and across connected hosts.
b) No auxiliary tool (like Qt's Meta-Object-Compiler) is needed, the implementation uses only regular language constructs
c) Signals and slots can be defined anywhere in the code (e.g. in conditionals) and are immediately active in runtime.
d) Being part of Karabo, signals and slots are available under C++ and Python and can be called across languages

As in Qt, slots are regular functions using the language's native data types as arguments. Signal/slot communication is type safe, i.e. a signal must match the signature of the receiving slot (in fact a slot may have a shorter signature than the signal it receives because it can ignore extra arguments).

Karabo's signal-slot mechanism can be used on top of any message-oriented middleware that at least supports publish/subscribe and message selection functionality (like e.g. JMS).

Most of Karabo's high-level functionality is based on the signals and slots mechanism making it a mostly event driven system. Besides signals and slots, two other broker-based communication patterns (direct call and request/response) exist and are used wherever appropriate.

Scientific Workflow System

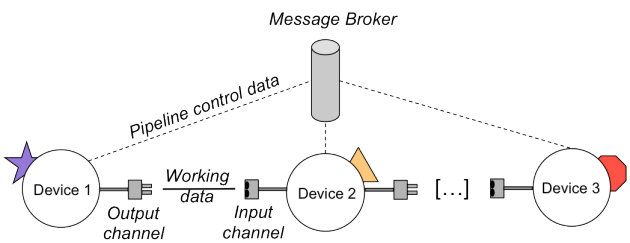

Figure 4: Rough sketch of the workflow communication.

Besides functioning as a regular control system, Karabo at the same time can be used to execute highly configurable scientific workflows similar to e.g. Triana [11]. Workflow nodes are regular devices with the addition of so-called input and output (IO) channels (see Fig. 4). Any workflow node may have zero or more IO channels and each IO channel is capable of reading/writing data of a pre-defined type. Currently, built-in data types are the Karabo hash as well as an image and a raw byte container (e.g. allowing to send files of any format). If needed, users can register custom data types to the Karabo serialization factory, which are immediately usable without re-compilation (plugin technology). Input and output channels can be connected in an N-to-N fashion and use the device's unique names in combination of the IO channel's name for addressing. Data is directly exchanged between workflow nodes (point-to-point) - the central broker is not involved. Nodes running in the same device-server exchange data transparently via memory-pointers in contrast to using TCP if running in different device-servers. Input channels can be configured to either receive a copy of any data sent by the connected output or to share the data amongst all other connected input (fan-out scenario for parallel computing on data chunks). Furthermore, a minimum number of data tokens to be accumulated before the internal compute method is triggered can be configured. This allows the integration of algorithms that cannot work in a streaming fashion but rather need to see all or larger parts of data for processing. Technically, input devices report their availability to the connected output devices in an event-driven way, allowing for features such as automatic data-flow adjustment and load balancing. A special "end-of-stream" token allows the design of circular workflow topologies and thus the implementation of global iteration cycles. IO operations are always performed asynchronously and transparently. In other words: whilst workflow nodes are computing on data, the next data is already sent in the background. In case data could be sent faster than being received, the sending channel can be configured to either wait for the input channel, queue or drop the data or just throw an exception. By default, output channels are waiting for input channel availability ensuring safe flow of data without risking any memory overruns. In this mode the whole workflow will automatically adapt its speed to the maximum possible data throughput.

As workflow nodes are devices, they can be written in C++ or Python and used interchangeably within the same workflow. From a device-developers perspective, coding a new workflow node firstly needs the definition of input and output channel plus any other auxiliary parameters (e.g. thresholds, algorithm types, etc.) using the device's self-description interface. Secondly, the processing algorithm must be implemented by overriding the compute function that will be called by Karabo. When this function is called, it ensures that all IO channels are prepared for data reading and writing.

ISBN 978-3-95450-139-7

Existing modules can be assembled into workflows either graphically (using Karabo GUI's central *custom* widget to drag and drop modules and connections) or via an XML file.

User Centricity and Access Control

Prior to using any control interface (be it CLI or GUI) users must log in with username, password and authentication provider (e.g. Kerberos [12]). This information gets automatically extended with data describing the broker (host, port, topic) and the local machine before sending it as a SSL encoded SOAP message to a web-based authentication service. The authentication service is connected to a central database, which computes the current access levels for the requesting user from the context of the provided information. Following properties are taken into account: i) users role ii) the Karabo sub-system used (broker information) iii) the location (IP address) of the user and iv) the current time. The web-service subsequently sends back a global access level and a list of device instance specific exceptions. Currently, five different access levels are used: *observer*, *user*, *operator*, *expert,* and *admin*. Properties and commands of devices on the other hand can be restricted to certain access levels using the "requiredAccessLevel" attribute. As all property and command descriptions are available to the user interfaces, client-side validation and adapted visualization (restricted auto-completion in CLI, invisible widgets in GUI) taking into account the current user's access levels is performed. In future, concepts like locking (single user device access) can be implemented by sending the user id and access level as part of any message to the devices (see Fig. 5 for an overview).

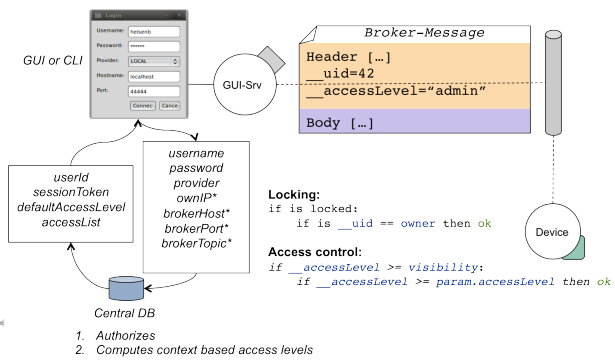

Figure 5: Illustration of the access control system.

CURRENT STATUS AND OUTLOOK

The first version of Karabo (1.0) was released for a restricted community of users (mainly European XFEL associated) in August 2013. The release included the Karabo backbone and some example devices. Currently, many other devices are being implemented for the new Karabo framework. The development focuses on control devices (e.g. motors, cameras, pump, valves, etc.), compute devices (e.g. calibration pipeline, CrystFEL [13] pipeline, parallel EMC [14], etc.) and also on integration devices to other control system such as DOOCS. Many more features are conceptually clear and are scheduled for future releases such as: Inter-device alarm handling, improved plotting of archived data, integration of routine image processing workflow modules, device-client availability in JAVA, etc.

CONCLUSIONS

Karabo is a software framework written from scratch and intended to fit the increased requirements regarding data volumes and rates of modern photon science facilities such as free electron X-ray lasers. By design it integrates control, data acquisition, data management and scientific computing tasks into one homogeneous software framework. It relies on modern programming language technologies and high quality third-party software packages and aims to provide users as well as developers a clean interface and intuitive access to data, hardware and computing resources.

REFERENCES

[1] M.Altarelli et al., XFEL: The European X-ray Free-Electron Laser technical design report, DESY XFEL Project Group (2006)
[2] http://www.boost.org. Boost C++ Libraries website
[3] http://www.tango-controls.org. The TANGO official website.
[4] http://doocs.desy.de. DOOCS website
[5] http://ipython.org. IPython website
[6] http://www.riverbankcomputing.com/software/pyqt PyQt website
[7] https://mq.java.net. OpenMQ website
[8] https://glassfish.java.net GlassFish website
[9] http://qt-project.org The Qt website
[10] Gamma, Helm, Johnson, Vlissides, "Design Patterns", Addison-Wesley, 1995.
[11] http://www.trianacode.org. Triana Website
[12] http://web.mit.edu/Kerberos Kerberos Website
[13] T. A. White, R. A. Kirian, A. V. Martin, A. Aquila, K. Nass, A. Barty and H. N. Chapman. "CrystFEL: a software suite for snapshot serial crystallography". J. Appl. Cryst. 45 (2012), p335–341.
[14] N.-T. D. Loh and V. Elser, "Reconstruction algorithm for single-particle diffraction imaging experiments", Phys. Rev. E 80, 026705 (2009).

ISBN 978-3-95450-139-7