electronic reprint

JOURNAL OF SYNCHROTRON RADIATION

ISSN: 1600-5775
journals.iucr.org/s

The Karabo distributed control system

Steffen Hauf, Burkhard Heisen, Steve Aplin, Marijan Beg, Martin Bergemann, Valerii Bondar, Djelloul Boukhelef, Cyril Danilevsky, Wajid Ehsan, Sergey Essenov, Riccardo Fabbri, Gero Flucke, Daniel Fulla Marsa, Dennis Göries, Gabriele Giovanetti, David Hickin, Tobiasz Jarosiewicz, Ebad Kamil, Dmitry Khakhulin, Anna Klimovskaia, Thomas Kluyver, Yury Kirienko, Manuela Kuhn, Luis Maia, Denys Mamchyk, Valerio Mariani, Leonce Mekinda, Thomas Michelat, Astrid Münnich, Anna Padee, Andrea Parenti, Hugo Santos, Alessandro Silenzi, Martin Teichmann, Kerstin Weger, John Wiggins, Krzysztof Wrona, Chen Xu, Christopher Youngman, Jun Zhu, Hans Fangohr and Sandor Brockhauser

J. Synchrotron Rad. (2019). **26**, 1448–1461

IUCr Journals
CRYSTALLOGRAPHY JOURNALS ONLINE

JOURNAL OF SYNCHROTRON RADIATION

ISSN 1600-5775

Received 24 October 2018
Accepted 9 May 2019

Edited by P. Fuoss, SLAC National Accelerator Laboratory, USA

[1] This article will form part of a virtual special issue on X-ray free-electron lasers.

‡ These authors contributed equally to this work.

Keywords: Karabo; control system; physics facility; experiment control; data analysis.

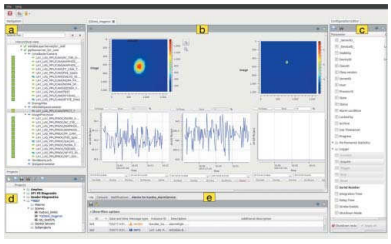
Supporting information: this article has supporting information at journals.iucr.org/s

The Karabo distributed control system[1]

Steffen Hauf,[a]‡ Burkhard Heisen,[a,b]‡ Steve Aplin,[c] Marijan Beg,[a] Martin Bergemann,[a] Valerii Bondar,[a] Djelloul Boukhelef,[a] Cyril Danilevsky,[a] Wajid Ehsan,[a] Sergey Essenov,[a] Riccardo Fabbri,[a] Gero Flucke,[a] Daniel Fulla Marsa,[a] Dennis Göries,[a] Gabriele Giovanetti,[a] David Hickin,[a] Tobiasz Jarosiewicz,[a] Ebad Kamil,[a] Dmitry Khakhulin,[a] Anna Klimovskaia,[a] Thomas Kluyver,[a] Yury Kirienko,[a] Manuela Kuhn,[c] Luis Maia,[a] Denys Mamchyk,[a] Valerio Mariani,[c] Leonce Mekinda,[a] Thomas Michelat,[a] Astrid Münnich,[a] Anna Padee,[a] Andrea Parenti,[a] Hugo Santos,[a] Alessandro Silenzi,[a] Martin Teichmann,[a] Kerstin Weger,[a] John Wiggins,[a] Krzysztof Wrona,[a] Chen Xu,[a] Christopher Youngman,[a] Jun Zhu,[a] Hans Fangohr[a,d] and Sandor Brockhauser[a,e]*

[a]European X-ray Free Electron Laser Facility GmbH, Holzkoppel 4, Schenefeld, Germany, [b]Cybus GmbH, Herlingsburg 16, 22529 Hamburg, Germany, [c]Centre for Free Electron Laser Science, DESY, Notkestrasse 85, 22607 Hamburg, Germany, [d]University of Southampton, Southampton SO17 1BJ, UK, and [e]Biological Research Centre (BRC), Hungarian Academy of Sciences, Temesvári krt. 62, Szeged 6726, Hungary.
*Correspondence e-mail: sandor.brockhauser@xfel.eu

The Karabo distributed control system has been developed to address the challenging requirements of the European X-ray Free Electron Laser facility, including complex and custom-made hardware, high data rates and volumes, and close integration of data analysis for distributed processing and rapid feedback. Karabo is a pluggable, distributed application management system forming a supervisory control and data acquisition environment as part of a distributed control system. Karabo provides integrated control of hardware, monitoring, data acquisition and data analysis on distributed hardware, allowing rapid control feedback based on complex algorithms. Services exist for access control, data logging, configuration management and situational awareness through alarm indicators. The flexible framework enables quick response to the changing requirements in control and analysis, and provides an efficient environment for development, and a single interface to make all changes immediately available to operators and experimentalists.



1. Introduction

1.1. EuXFEL characteristics

The European XFEL is a research facility with diverse and somewhat unusual requirements including a pulse structure (up to 27000 photon pulses per second) arranged into 10 Hz trains of pulses at 4.5 MHz (Altarelli *et al.*, 2006; Altarelli, 2011) and the use of state-of-the-art, high-repetition-rate, large-area 2D imaging detectors capable of detecting images of scattered photons produced by a single XFEL photon pulse. These result in very high data rates which must also be subsequently calibrated before further analysis (Kuster *et al.*, 2014). Custom detectors and instruments require dedicated interfaces between control software and complex analysis routines that provide calibrated detector data for online analysis and subsequent near-real-time feedback into the experiment control.

Given this set of requirements (Esenov *et al.*, 2009) and a review of relevant existing control and analysis systems

electronic reprint

available at the time [such as DOOCS (Grygiel *et al.*, 1996), EPICS (Dalesio *et al.*, 1994) and Tango (Götz *et al.*, 2003)], it was decided that a new distributed control system, Karabo, with integrated data acquisition and workflow capabilities should be designed and developed on top of a standardized electronics controller layer implemented by Beckhoff terminals and programmable logic controllers (PLCs).

This decision was supported by the DAQ-and-Controls section of the Detector Advisory Committee (DAC) which is the responsible international advisory body for the European XFEL.

The Karabo distributed control system has been and is still being developed since early 2012 (Heisen *et al.*, 2013), and has been in use since September 2017 to enable first scientific user experiments at the European XFEL. The intention is to release Karabo to the public using an open source software licence in the future.

1.2. Karabo

Karabo is a distributed control system that interfaces to hardware devices through software counterparts (called *Karabo driver devices*). While driver devices mirror the status and settings of the hardware equipment within the Karabo system, so-called middlelayer devices – and Karabo devices in general – can interact with other Karabo devices. Subsystems controlled with other control systems may be integrated into the Karabo system via gateway devices. Karabo enables centralized and peer-to-peer communication between devices. The use and control of the system is facilitated by a generic graphical user interface (GUI) and a command line interface (CLI): iKarabo. CLI commands can be combined to form reusable macros which are also accessible via the GUI. There is a range of basic services provided for managing configurations, raising alarms and the logging of system events.

Karabo driver devices represent hardware devices such as pumps, motors and cameras. Other Karabo devices may not be associated with hardware, but may carry out other roles, such as data analysis operations or coordination or composition of multiple other devices.

To represent a set of devices working together, for example as (part of) a beamline, a scientific instrument or an experiment, a *Karabo project* can be used.

Karabo projects allow groups of devices to be defined and project-specific configurations of the devices to be stored. Users can define multiple projects to be able to configure devices for respective use cases. Projects can be enriched through the creation of multiple graphical arrangements of display elements – so-called *Karabo scenes* – to provide customized views covering all the diagnostic or control elements from any device in compact and comprehensive views as is required for convenient operation. *Karabo macros* can also be stored as part of a project to support carrying out repetitive tasks associated with the project programmatically.

A Karabo project can contain other Karabo projects as subprojects, and thus it is possible to build hierarchical projects and organize complex operation configurations and views in a modular manner. Subprojects can thus be independently created and (re-)used in multiple projects, allowing compartmentalization of a complex system of devices, *e.g.* a detector component, for use in different experiments.

Beyond their use in static operation, Karabo projects also aid commissioning and such experiments where flexible changes of settings and devices need to be supported.

The interface used most commonly by operators is the GUI. For simple monitoring, Karabo is designed to support a cinema mode in which previously customized GUI scenes are quickly launched and displayed for immediate use.

To conduct parameter space exploration effectively, the *Karabo Scan Tool* can be used. It integrates into the GUI and the command line interface for custom configurations. The scan tool allows varying single or multiple independent parameters – such as motor positions or any other control parameter – automatically, while for each new parameter combination synchronously recording detector and sensor data, as well as other predefined observables that are derived from these data via analysis pipelines. The scan tool additionally provides a plotting widget for rapid assessment of scan progress and result quality.

When collecting data, specific Karabo devices gather and subsequently store entries into HDF5 files (Boukhelef *et al.*, 2013). Karabo also provides for the streaming of this high volume data through *Karabo pipelines* between devices possibly running on distributed hardware. Data acquisition onto disk and the parallel feeding of real-time data streams to be used by online analysis devices is tightly integrated into the Karabo system.

Control feedback loops can involve any number of Karabo devices. Due to the close integration of streamed data processing and control, this may include feedback from complex data analysis operations. In this way automation and stabilization of procedures and experimental protocols are supported. A specific Karabo device, the Karabo bridge, provides data streaming to applications outside of the Karabo ecosystem, thus integrating non-Karabo user tools into the control loop.

Karabo control system is installed at the European XFEL on hosts registered in a separate control network, as well as on desktop computers, in the control hutches of the instruments and the central accelerator control room. Users of the European XFEL are provided with access to an online cluster during their beam time which is connected to Karabo, so user-specific real-time analysis can be plugged into the system. A data analysis toolbox, *Karabo data* (European XFEL GmbH, 2018*a*), provides convenient access to the data collected.

Additionally, in this document, we provide details on Karabo's design (Section 2), Karabo's client environment (Section 3) and data analysis (Section 4) support. We describe usage examples of Karabo (Section 5), comment on our software engineering methods (Section 6), and close with a summary (Section 7) of the achievements and a discussion on future outlook.

electronic reprint

2. Karabo design

We distinguish between the *Karabo framework* and *Karabo devices*; where devices realize a particular functionality through use of the Karabo framework. The object oriented Karabo framework is implemented in C++ (Stroustrup, 1995) and Python 3 (Van Rossum & Drake, 2011). Similarly, devices may be implemented using application programming interfaces (APIs) in Python or C++.

2.1. Devices

Karabo devices are the smallest significant part in the Karabo system. A Karabo device may mirror a hardware device, and thus act as the interface from the control system to the hardware. Karabo devices can also be independent of hardware, and provide for example logic or arithmetic processing.

There is a set of API-specific base device classes from which all Karabo devices inherit. These Karabo-provided base classes implement the standardized communication with the distributed system. Other than the need for deriving from such a class, no further requirements on inheritance or composition are exerted by Karabo. Device objects, *i.e.* instances of the different types of device classes, are identified in the distributed system through unique string identifiers. At the European XFEL these identifiers follow a naming convention, encoding location, device type and instance. For example, `FXE_DET_LPD1M−1/FPGA/FEM_Q1M2` refers to the 1 Mpixel Large Pixel Detector (LPD, see Section 5.3), installed in the FXE hutch, and there to a sub-component which is an FPGA board acting as the control interface to the second module of the first detector quadrant.

2.2. Device properties

Devices have properties resembling their configuration and current status. These include hardware-specific read-only parameters such as a temperature measured by a hardware sensor or the current position of a motor. Other common parameters such as a device's state (see Section 2.4), its unique identifier or the device server hosting the device are also exposed to the distributed system as properties.

Device properties can have specifiable access levels, for example to hide expert options from facility users and lay operators.

Table 1 in Section S1 of the supporting information lists the currently supported property data types. Properties may be hierarchically organized into a node-leaf structure, which is reflected in Karabo's fundamental data type, the *Karabo hash* (Section 2.8).

In addition to the plain data types listed in the table, Karabo natively supports composite data types. A data type for images for example combines all the relevant image properties like the region of interest, binning information and encoding into a data container. The actual pixel data is represented therein using another composite type that maps to Python's `ndarray` class (Walt *et al.*, 2011) for multidimensional arrays.

2.3. Messaging in the distributed system

Intercomponent communication is a defining aspect of distributed control systems. In Karabo the distributed components are the devices, hosted on device servers (Section 2.9), macros and the CLI. Messages are routed via a central broker. Currently, Karabo uses the Java Messaging Service (JMS) broker (Hapner *et al.*, 2002), and the message layer is implemented using the Open Message Queue, OpenMQ(C) interface. Similarly to most communication brokers on the market, multiple JMS brokers can be clustered to share the communication load across different machines. Message routing is unaffected by configuration of the broker cluster. The Karabo design foresees that its servers are configured to reconnect to another broker of their cluster if the connection to their broker is lost.

Messages may either inform the distributed system about the change of a state or property (Section 2.2) on a device or request an action to be performed thereon. System messages additionally inform about new device (Section 2.1) and server instances (Section 2.9) in the system, track all running instances, and give notification of devices which have been shut down.

Communication between components is implemented in the fashion of signals and slots (Qt, 2018), which is a design construct introduced by the Qt framework (Dalheimer, 2002) to support the effective implementation of the observer pattern. This concept – of *signals* being sent from a device to other devices via the broker, and *slots* being called from other devices to trigger an activity – has been integrated into Karabo.

Broker-based messaging has been measured to cope with multiple kHz data-rates at EuXFEL production installations consisting of thousands of distributed devices (see Section 5). A single device can reliably consume 2 kHz of messages as measured in distributed tests. The message latencies seen at GUI server devices (Section 3.2) are monitored in production. The latency averages over 5 s periods are below 10 ms in normal operating conditions. To allow further scaling in the future, an effort is on the way to switch to a different broker architecture.

In addition to broker-based communication, Karabo supports peer-to-peer messaging between devices. Communication paths to the data-logging system, as well as within data-processing pipelines, are implemented in this way and allow higher data transfer rates than those possible over a broker.

The system has been shown to be capable of digesting for example the Gigabyte per second data-rates from the European XFEL's MHz-rate detectors over multiple infiniband and 10 G-ethernet channels as part of online detector calibration (Kuster *et al.*, 2014). Up to 256 Mpixel images per second have been processed, and experiments using the Adaptive Gain Imaging Pixel Detector (AGIPD; see

electronic reprint

Section 5.3) and LPD detectors routinely generate multiple tens of Terabytes of data during a five day user beam time.

Broker communication is generally used for scalar and vector data which are updated at slow rates (*e.g.* once per XFEL train). All image or multi-dimensional data as well as pulse-resolved data are transferred using peer-to-peer pipeline communication. While switching from one communication model to another is not supported on-the-fly, it requires only a small change in the code.

2.4. Karabo states

State is an important concept in most distributed control systems, as it represents the most essential information on a component's status to a supervising operator. Frequently, state is exposed to the user as a short but descriptive text.

For Karabo a fixed set of hierarchically organized states has been chosen to provide a condensed context-sensitive description of a hardware or software component via its Karabo device. More descriptive states derive from three basic states: `UNKNOWN`, `KNOWN` and `INIT`. Here, the `UNKNOWN` state is reserved for when the software cannot establish a connection to the hardware it is to monitor and control, or is otherwise functioning abnormally such that it cannot guarantee an accurate representation of the hardware's state. The `ERROR` state should be entered in case of a known hardware error. Most of the time the device will be in its `NORMAL` state, *i.e.* it will be correctly reading and reporting the hardware state. As is indicated in Fig. 1, the `NORMAL` state is the base state for many more specific states.

Depending on the device requirements, a finite state machine can be used to define possible transitions from one device state to another formally.

2.5. Alarms and notification system

Karabo is provided with integrated alarm notification, implemented as a service device. So-called alarm conditions are evaluated on a per-property and per-device level. Three alarm levels are supported: `WARN`, `ALARM` and `INTERLOCK`; each visually identified in a triply redundant fashion by shape,

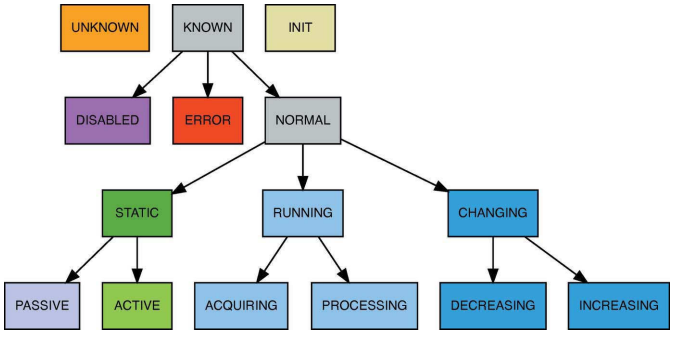


Figure 1
Overview of Karabo's basic unified states and their relation to one another. Not shown are more application-specific derived states. The colours shown in this diagram are the colour-codes Karabo uses for each state in its GUI.

Figure 2
Indicators used by Karabo for, from left to right, warning, alarm and interlock alarm types.

colour and lettering as indicated in Fig. 2. Property-related alarm thresholds T_i may be hard-coded or configured at initialization 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

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

If the quantity $v(t)$ goes beyond the low or high warning thresholds, the distributed control system notifies of the warning condition, indicating to the operator that an abnormal condition is imminent and alarm thresholds might soon be exceeded if no action is taken. Should this happen an alarm indication is sent.

The `INTERLOCK` alarm-type can only be triggered from interlocking hardware, as it is a policy for Karabo usage at European XFEL to not implement any software-based interlocks as part of machine-, equipment- or personal protection systems. The `INTERLOCK` alarm-type can be set only at device level.

In addition to property-related alarms, devices have a global alarm condition, which can be explicitly assigned through device logic. It will automatically evaluate to the highest alarm condition out of all property-related alarms and any explicit manual assignment.

Alarms can be defined to require acknowledgment, *i.e.* their notifications will not silently disappear if the condition triggering the alarm passes.

2.6. Karabo projects

Karabo projects allow associating sets of devices, scenes and macros while also supporting multiple device configurations for specififc use cases. Projects can be opened, saved, duplicated and marked as trashed through the Karabo GUI (Section 3.1) or CLI. Trashing a project will remove it from the standard overview of available projects; however, trashing is a revertible operation, therefore accidental deletion by the user is not possible. A Karabo device (and an associated hardware device) can be a part of multiple projects; this enables storing multiple configurations of a device depending on the required use case.

Each Karabo GUI client can open one project at a time. Projects can contain other projects (as a sub-project), to avoid duplication of effort and configurations.

Projects are stored in a central NoSQL (eXist-db) database but also simultaneously cached on the local hard drive for offline access. The persistance layer is well abstracted and its backend implementation can easily be changed.

electronic reprint

2.7. Karabo APIs

Karabo devices can be implemented using one of the three application programming interfaces (APIs) including a C++ and two Python implementations.

The *C++ API* allows implementation of devices in the C++ programming language [using the C++11 standard (ISO, 2011)]. The C++ API is the suggested API for low-level interaction with hardware or performance critical devices. Most of the bundled devices that implement tasks essential for the core system – so-called Karabo *service* devices – are implemented using this API. Interdevice communication on the same server using the C++ API supports a direct message passing shortcut instead of involving a message broker.

The bound Python API exposes the C++ API functionality via the Boost C++ libraries (Schäling, 2011) and its `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 API name reflects that each Python routine is bound to the corresponding C++ routine. Asynchronous execution is achieved by using an event loop, which works with multiple threads.

This *bound Python API* is suggested for implementation of devices interacting with hardware, as well as computational devices, implementing numerically demanding algorithms, since any bound C++ Karabo calls, *e.g.* input, output and serialization, can be done in parallel in a multi-threaded environment.

The middlelayer API is purely implemented in Python, with no dependencies on the other two APIs, and with the intention of being a pythonic interface, following Python conventions and standards [such as PEP8 (Van Rossum *et al.*, 2001)]. 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.

Examples of how a *Hello World* device can be implemented in these three APIs are given in Section S2 of the supporting information. In addition, a more complex macro for an absolute scan is provided in Section S4 of the supporting information showing the orchestration of a motor and a detector device and it briefly illustrates a few synchronization routines, *e.g.* 'waitUntil' or 'waitWhile'. These conditional functions are asynchronously evaluated every time event-driven changes are registered.

2.8. The 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 serialization of a Karabo hash is restricted to the types listed in Table 1 in Section S1 of the supporting information, as well as composite data types such as image and `ndarray` data. An extended toy example is shown in Fig. S4 in Section S3 of the supporting information.

The Karabo hash is used as the central data structure to communicate information between components of the system, both for communication via the broker and direct peer-to-peer messages. For example, a bunch of 64 photon pulses captured by a module of a fast detector, *e.g.* the LPD, with a resolution of 256 by 256, is sent as a Karabo hash that contains an `ndarray` with shape (64, 256, 256) and some additional metadata.

The Karabo hash is available in all three APIs. Serialization is supported to XML (Bray *et al.*, 1997), HDF5 (Folk *et al.*, 2011) and ZeroMQ (Fangohr *et al.*, 2018) as well as to a proprietary binary format to be used for communication within Karabo.

2.9. Device servers

Karabo devices are hosted by Karabo device servers specific to their API flavour. Due to the nature of the programming languages the implementation for each API is different.

The C++ device server runs a central event loop with many threads. It starts devices as part of its single process and the tasks of the devices are processed in parallel on the central event loop. This allows optimization of inter-device communication on the same server by bypassing the broker.

In Python bound the global interpreter lock (GIL) prevents true multi-threading. Hence, a separate process with a central event loop is started for each device, and short-cut communication is not possible. In the middlelayer API the device server is started as a single-threaded process with a central event loop. Each Karabo device can be subsequently started as a task on this event loop, similarly to C++.

2.10. Control feedback loop

A particular strength of the integration of control and scientific data into Karabo is that control feedback to the experiment based on data analysis outcomes is possible. Data from the detectors and sensors can be analysed within Karabo or with external tools (that can connect to the data stream via the Karabo bridge, Section 4.4) and the output of that real-time analysis can be used to instruct control elements such as motors and delay elements to optimize the experiment.

For example, such a feedback loop is used to improve the spatial stability of the X-ray beam: we utilize intensity position monitors (IPMs) which are analyzing the measured current from a quadrant detector. The X-ray beam is continuously steered to the centre point of the detector with piezo motors mounted on Kirkpatrick–Baez mirror systems to guarantee spatial stability.

While other control systems may require external tools for integrating more complex workflows (Brockhauser *et al.*,

electronic reprint

2012), Karabo has been designed to support them natively and to provide built-in GUI tools to facilitate creating them.

3. The Karabo client environment

We define a *Karabo ecosystem* as the set of Karabo software components interconnected via a central message broker, and communicating within a common Karabo message topic. The Karabo message topic represents a dedicated session on the broker limiting the message distribution to those software components belonging to this namespace.

A Karabo ecosystem consists of C++ and bound Python devices that interact with the hardware, the middlelayer devices and macros orchestrating them, the device servers hosting them, various service devices (logging service, alarm service, GUI servers, project data base service) and a client environment.

The user interface to the Karabo ecosystem is the *Karabo Client Environment*. It encompasses the *Karabo Graphical User Interface* (Karabo GUI, Section 3.1) and a command line interface (Karabo CLI, Section 3.3) named *iKarabo*. These tools allow users and experts to control and monitor the supervisory control and data acquisition (SCADA) infrastructure. Many control tasks can be equally performed from the GUI or the CLI, depending on requirements (*e.g.* scriptable access) or personal preference.

3.1. The Karabo GUI

Karabo's Graphical User Interface (Fig. 3) is a multipurpose application. The GUI is composed of detachable panels. For most operators it is the preferred entry point to the Karabo ecosystem, and commonly displayed on the control screens of the beamlines and instruments.

The Karabo GUI is an executable that needs to be pointed to the network host and port of a Karabo GUI server on start up. At the same time authentication allows different operator levels (administrator, expert, operator, user, observer) which differentiate in the amount of detail the Karabo GUI presents to the user. When this connection is established, a window similar to the one shown in Fig. 3 opens.

The *navigation panel* [Fig. 3(*a*)] offers an overview of the Karabo ecosystem topology as a tree view whose top-level nodes are the physical computer hosts on which the device servers are run. The latter are depicted as second-level nodes and can be expanded to reveal the device classes they are capable of instantiating. These device classes and the instantiated devices form the third and fourth levels in this hierarchical view of the SCADA topology.

As thousands of devices are needed to support beamlines and instruments at the European XFEL, the navigation panel allows textual filtering to allow operators to limit the displayed devices. More complex filters can be expressed as regular expressions.

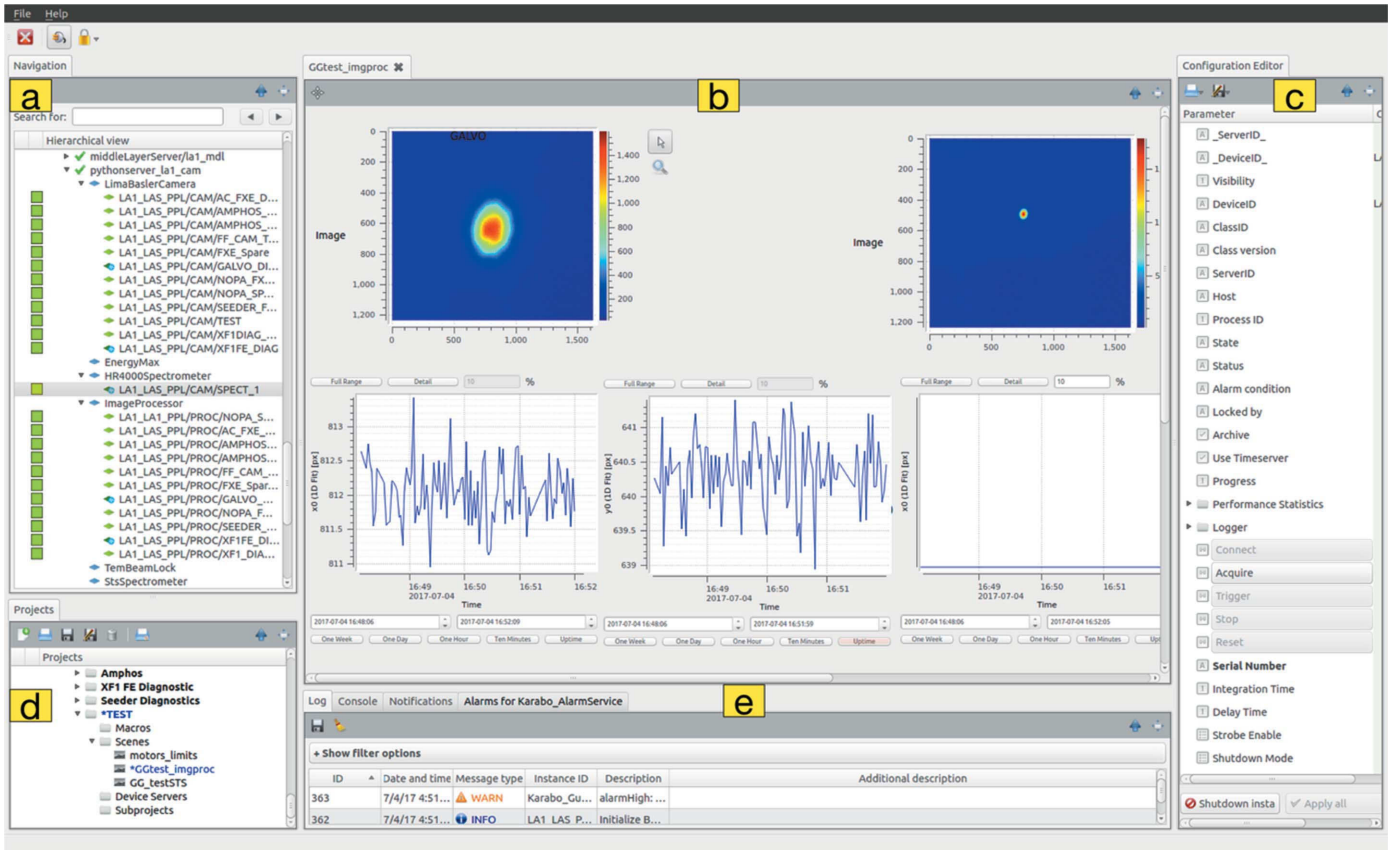


Figure 3
Karabo's graphical user interface.

electronic reprint

The *project panel* provides access to the database of available projects. 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. Every project can be opened stand-alone or is loaded as a subproject to another project.

The notification panel [Fig. 3(*e*)] is subdivided into a number of tabs:

(i) *The logging tab* shows information, warning and error messages issued by devices or servers. They can be filtered or sorted by date, type, device server or description. This list can be exported and cleared.

(ii) *The alarm tab* lists messages from the alarm system (Section 2.5), and allows eventual acknowledgement. On selection, the reporting device configuration can be displayed in the configuration panel [Fig. 3(*c*)] to help investigate the problem effectively.

(iii) *The console tab* provides access to a remote iKarabo session, allowing for client independent scripting access. Section 3.3 gives an overview of this Command Line Interface.

The configuration editor panel [Fig. 3(*c*)] displays and monitors the properties and commands of a selected device. A device can be selected either in the Navigation panel [Fig. 3(*a*)], in the project panel [Fig. 3(*d*)] or from any scene hosting a widget referring to a property of the device. A contextual help describes each of the device's properties, detailing their type, default value, timestamp of last update, as well as alarm and warning thresholds. Property displays are updated when properties change on the device. Reconfigurable properties can be set from the configuration editor panel. In a similar way, the initial configurations of devices, as stored in the project database, can be edited.

The central panel [Fig. 3(*b*)] is where scenes and macros can be displayed.

(i) 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.

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.

Any panel and tab in the GUI can be detached, displayed and moved as a stand-alone window on the desktop of the computer displaying the GUI. This allows arranging sets of scenes (or other panels) for viewing in the most beneficial way for the task at hand.

(ii) *The macro editor* is displayed in the central panel as well. Macros are meant for automation of recurring tasks. A field in the bottom part of the macro panel captures the standard output of the macro, which runs remotely on a dedicated macro server. Hence, macro execution follows the policy *edit local – run central* and each macro appears as a device in the system topology. A macro's properties and commands are rendered in the configuration editor and may be used in scenes.

3.2. Karabo GUI implementation details

The Karabo GUI is developed in PyQt. GUI clients do not communicate directly with devices via the central broker, but instead interface with a so-called GUI server. The GUI client/server protocol is a Karabo hash-based signal and slot exchange, conveyed over a single TCP connection. One benefit of this design choice is the portability of GUI clients. They run on Windows and Linux as well as OS X and binding to a Java Messaging Service (JMS) client library is not a requirement for implementation. The approach also provides convenient remote access to a Karabo ecosystem via SSH tunnelling. Additionally, the GUI server can filter out redundant requests to the same resource originating from multiple clients, thus decreasing broker traffic. Similarly, a GUI server can throttle the transmitted data rate (relevant especially for 2D image data) if a GUI client cannot sustain it.

3.3. The Karabo CLI

The *Karabo Command Line Interface* (Karabo CLI) is a tool for swift investigation and scripting. It is a light customization of the Interactive Python (IPython) shell (Pérez & Granger, 2007). As such it was named iKarabo, and benefits from the convenience that IPython provides, including autocompletion and contextual help. Operators can easily invoke the concise middlelayer and macro APIs on this interface. Domain-level operations such as stepwise or continuous sample scans can be carried out in a comprehensive way.

The iKarabo shell allows users to benefit from the general purpose language Python: arbitrarily complex control tasks can be expressed to provide automation and convenience in using the control system. Embedding Karabo – as a domain specific language – in an existing general purpose language Python is a better approach than defining a new domain specific language (Beg *et al.*, 2017).

3.4. Security

While Karabo is implemented with an open communication between the devices, basic security aspects are addressed during the installation at EuXFEL by separating the control network hosting the Karabo servers from the generic office network of the company (du Boulay *et al.*, 2008). Karabo services are made available to GUI clients via a single port in which the Karabo GUI server is listening. Command Line Interface use from external clients is only possible via the open port of the GUI server which enables macros sent to the Karabo macro server where they are interpreted and/or filtered and subsequently executed. Furthermore, the GUI implements an authentication system with different access levels which is used to hide different subsets of device para-

electronic reprint

meters. As the control software is a key component for the optimal use of the facility, its security is also considered as an important aspect. Hence, a more comprehensive solution for securing Karabo has been designed (Mekinda *et al.*, 2018).

4. Data analysis in Karabo

4.1. Introduction

Data analysis is important (i) during the experiment to ensure most effective use of the beam time, and (ii) subsequently to convert the investment of the experiment into the best research value possible.

The design of the data storage and analysis provision at the European XFEL aims to allow a comprehensive, state-of-the-art analysis of each experiment conducted, allowing for improvements in calibration routines and data analysis algorithms in the future. By recording all parameters and software versions of components in the process, we aim to provide full reproducibility of any data extraction and processing.

We distinguish between (i) rapid feedback data analysis, (ii) online data analysis and (iii) offline data analysis (Fangohr *et al.*, 2018): rapid feedback data analysis at European XFEL – with latencies of the order of seconds – is dominated by live processing of data streams on a dedicated compute resource (known locally as the online cluster). For online data analysis, *i.e.* data analysis carried out during the experiment but with higher latencies, and offline data analysis, *i.e.* data analysis carried out after the experiment, the Maxwell cluster (DESY, 2017) is available as a compute resource. This analysis is driven by the processing of data files (Section 4.5).

Fig. 4 shows a simplified view of the data flow in the Karabo system.

The facility aims to provide long-term storage (at least five years, aiming for ten years) for raw detector data (European XFEL GmbH, 2017) and reduced data sets. A major part of the stored data are uncalibrated images from the (2D) detectors.

4.2. Calibration

Before any analysis is carried out, detector-specific peculiarities and artefacts need to be removed from the data set, through application of appropriate corrections and calibration to the data. The data pipelines and calibration routines have been designed so that this calibration can take place on-the-fly, and can be regarded as a processing tool that is applied to the raw data. Facility users are not expected to access the raw data directly, and if data are retrieved from the EuXFEL's raw data archive then the calibration is applied to the raw data automatically before data are passed on to the user. This approach allows the use of a different calibration at a later point, which is needed if detector characteristics have changed during the experiment, or an improved calibration becomes available after the experiment (Kuster *et al.*, 2014). It is, of course, crucial to record which calibration routine has been applied for all later stages of the data analysis.

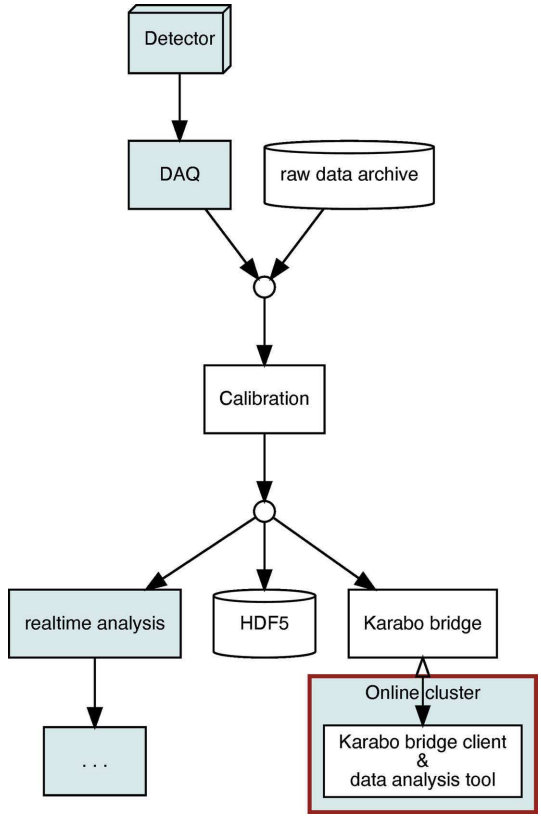


Figure 4
Simplified overview of data flow in Karabo: during the experiment, data from the detector goes via the data acquisition (DAQ), and through the appropriate detector calibration. It can then be used for real-time data display and analysis, for storage to HDF5 files, and be sent to further analysis tools through the Karabo bridge interface device. For offline analysis, data is read from EuXFEL's data archive and injected into the same data flow pipeline: first calibrated, and then offered to users for subsequent data analysis, either as files or through the Karabo bridge. The online cluster (bottom right, bordered by thick dark-red frame) is separated from the control network (see Section 4.4). Elements of the pipeline that are only applicable to real-time analysis during the experiment are rendered with a grey background.

4.3. Streaming of data through pipelines

During an experiment, Karabo's integrated data handling capabilities support streaming of data from detectors and sensors to HDF5 files and simultaneously to rapid-feedback calibration and online analysis devices. Karabo's peer-to-peer pipeline communication is essential in providing the necessary data throughput for these workflows. Data associated with pulses in an XFEL train are transferred through the pipeline, using the Karabo hash (Section 2.8) which provides a hierarchical structure and enables efficient serialization. The distributed nature of Karabo foresees data processing parallelization by spreading data analysis activities, *e.g.* detector image calibration, over many servers.

The (calibrated) streaming data can be used for:

(i) Near real-time analysis in Karabo devices, providing for example the latest set of detector image data and crucial control parameters as input to scenes in the Karabo graphical user interface. The same data can also be sent to the first stage of further data analysis tools that give rapid feedback on

electronic reprint

whether images contain X-ray signatures of sample hits, *i.e.* are interesting for scientific analysis.

(ii) Further data analysis using existing tools: we provide a network interface which allows sending data from any Karabo pipeline to external applications through a network connection (see Section 4.4, Karabo bridge).

4.4. Karabo bridge

The Karabo bridge (see Fig. 4) allows external data processing pipelines to connect to Karabo for real-time data processing without being implemented directly within Karabo.

The Karabo bridge translates the Karabo data stream into a well defined protocol, and makes it available outside the control network so that external applications can request and process the data on the online cluster. The division of networks increases the protection of the control network and isolates Karabo from being influenced by external programs, thereby enhancing the performance and security of Karabo. The bridge allows calibrated data to be processed a few seconds after collection without having to be read from the file system. Applications can then provide near real-time feedback to the experiment control room for quick decision-making.

The Karabo bridge currently sends data over a ZeroMQ (Hintjens, 2013) connection using MessagePack (Furuhashi, 2008–2013) serialization as the preferred protocol (European XFEL GmbH, 2018*b*).

Client programs that read data from the Karabo bridge can be relatively short and need only few (rather common) dependencies to connect to the Karabo pipeline and translate the Karabo bridge data stream into the appropriate form for each application. Example clients are currently available in Python and C++ (European XFEL GmbH, 2018*c*) and have enabled tools such as *CASS* (Foucar, 2016), *Dozor* (Zander *et al.*, 2015), *Hummingbird* (Daurer *et al.*, 2016), *OnDA* (Mariani *et al.*, 2016) and *pyFAI* (Ashiotis *et al.*, 2015) to connect to the Karabo pipeline. These tools are typically established applications that have already been used during experiments at other facilities and provide online data analysis routines such as for example azimuthal integration and crystallography hit-finding.

The Karabo bridge can also be used to feed information from external data analysis tools back into the control system (hollow arrowhead in Fig. 4) so that the output of near-real-time analysis from external application can be used as input for control feedback, for example to automate aspects of the experiment that otherwise would have to be adjusted manually by scientists (Fangohr *et al.*, 2018).

4.5. Karabo-data

For offline analysis, science users generally retrieve data from the file-based raw data archive. Where required the appropriate calibration and other processing for the detector and experiment is applied automatically before the processed files are made available to the user for a limited period of time.

The Karabo-data tool (European XFEL GmbH, 2018*a*) provides a (Python-based) library to extract data from these files more conveniently. The library can extract selected data sources of interest from the files associated with one run without the user needing to know which file contains which data source and which trains. Data can be converted into other formats such as, for example, pandas DataFrame objects (McKinney, 2011), comma-separated value files and others, so that the growing data science ecosystem with tools such as Jupyter (Kluyver *et al.*, 2016), matplotlib (Hunter, 2007), pandas, xarrays (Hoyer & Hamman, 2017), seaborn (Waskom, 2012–2018) *etc.* can be used to extract insight from the saved experimental data.

Karabo-data can also stream files so that the stream appears to come from the Karabo bridge interface, thus mimicking the data streaming that would take place during an experiment. This allows reuse of the same interface by other data analysis tools as well as those used during the experiment (see Fig. 4), and helps in developing and testing data analysis components before the experiment.

Fangohr *et al.* (2018) provide further details on data analysis with Karabo.

5. Achievements and examples

5.1. Karabo installation at the European XFEL

The European XFEL user operation has started with the experiments on the SASE1 (self-amplified spontaneous emission) beamline: the Femtosecond X-ray Experiment (FXE) (Bressler, 2011; Bressler *et al.*, 2012) and the Single Particles, Clusters and Biomolecules and Serial Femtosecond X-ray experiment (SPB/SFX) (Mancuso, 2011; Mancuso *et al.*, 2013).

The beamline and experiments entered their commissioning phase in the first half of 2017, with early user operation starting in September 2017. Throughout these phases, SASE1 has been controlled by Karabo.

As of September 2018, the Karabo control system installation at European XFEL is distributed over 250 dedicated control computers, hosting more than 7500 Karabo devices, with over a million control points (*i.e.* device properties).

5.2. Photon transport and vacuum systems in SASE1

Karabo has played an essential role in the commissioning and control of the SASE1 beamline, offering a system/subsystem view of the thousands of deployed devices needed to drive the vacuum and photon transport systems as well as beam diagnostics.

Fig. 5 shows an overview of the beam transport and vacuum systems alongside detailed panels for mirror and vacuum section control. The indicated GUI panels were created through the scene-builder functionality alone, requiring no custom GUI coding. They use icons from the standardized icon set. The indicator colours are those assigned to the unified states.

The electron accelerator and the undulator systems are controlled using the DOOCS control system on DESY side. A Karabo client providing a wrapper to the DOOCS system

electronic reprint

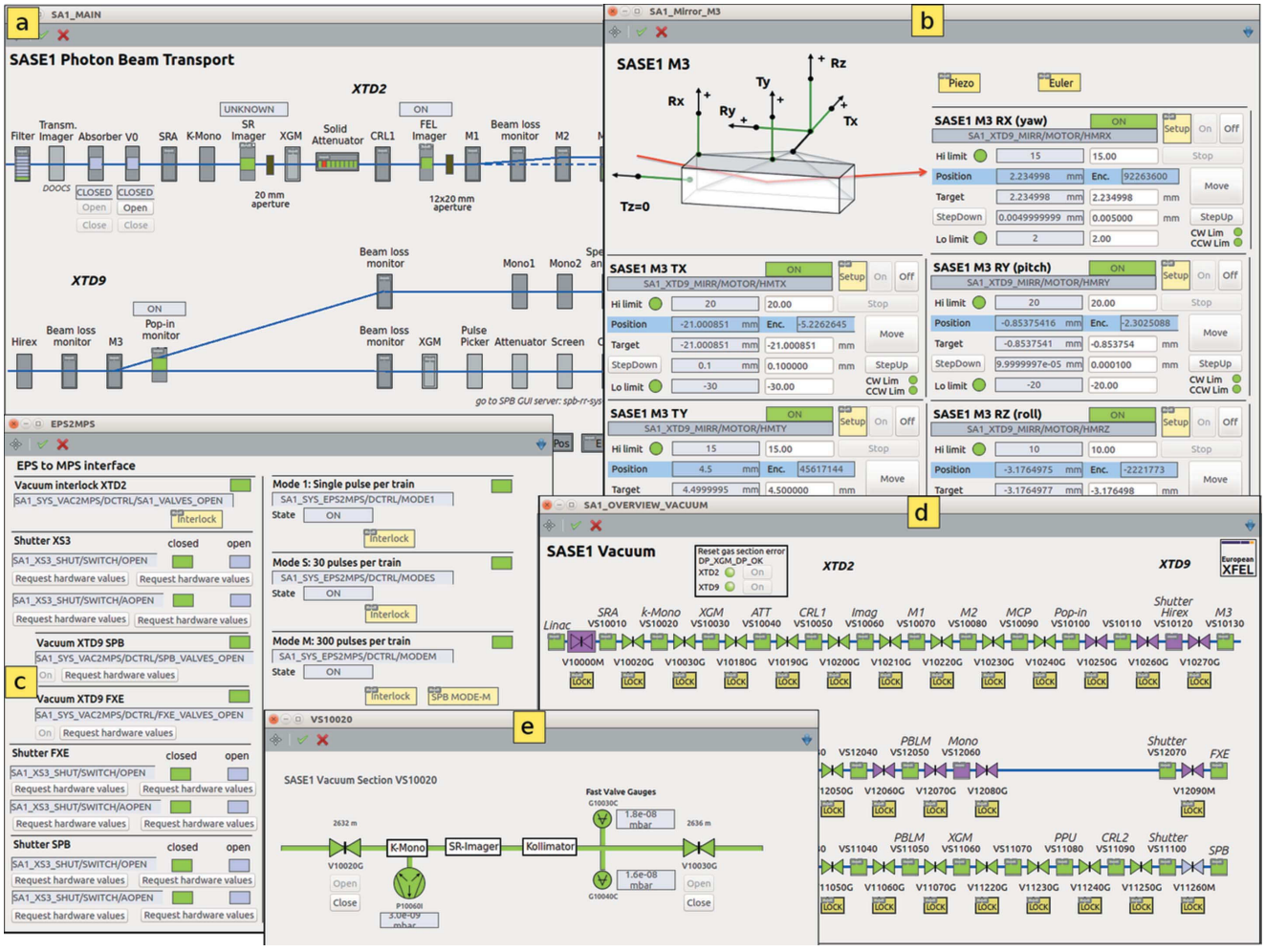


Figure 5
Karabo GUI panel examples: (*a*) beam transport overview, (*b*) mirror control, (*c*) status of equipment and machine protection system, (*d*) vacuum system overview, (*e*) vacuum section control. The panels use Karabo's standardized icon set, and the colour indicators reflect the unified state system as shown in Fig. 1. Each panel is a scene. The scenes are associated with Karabo projects.

allows integration with the Karabo ecosystem by sharing the configuration of the electron beam as well as the beam diagnostics elements. Similarly to what has been done for DOOCS, Karabo client wrapper libraries for other control systems such as EPICS, TANGO and TINE have been developed and are used for interacting with those control systems.

5.3. MHz-rate detector control and data acquisition

Karabo is used for control and data acquisition from European XFEL's MHz-rate 2D X-ray detectors: the Large Pixel Detector (LPD), the Adaptive Gain Imaging Pixel Detector (AGIPD) as well as the DepFET Sensor with Signal Compression (DSSC). LPD and AGIPD are in user operation at the FXE and SPB experiments, respectively, and have been designed to acquire images of 4.5 MHz bursts at 10 Hz, needed to match the XFEL pulse timing structure. The detectors currently produce up to 5080 Mpixel images per second, and DSSC will produce up to 8000, which requires the Karabo-based data acquisition (DAQ) system to digest rates between 10 and 15 Gigabytes s^{-1} (Kuster *et al.*, 2014). Subsequently, the detector raw data are calibrated using GPU- and CPU-based algorithms implemented using Karabo pipeline technology (Hauf, 2017).

Fig. 6 (background) shows a Karabo control and online preview panel for an LPD prototype consisting of two of the megapixel detector's 256 sensor tiles. The detector is imaging, for the first time, diffraction at a 4.5 MHz repetition rate from an XFEL beam at the FXE hutch (July 2017), resolving the 222 ns FEL pulse separation. The online preview displays raw and offset-corrected data. The intensity scale is inverted. Two months later the full megapixel detector had been commissioned and used for first user experiments. The foreground image in the figure shows a current (September 2018) screenshot of the online preview for corrected and geometry-assembled data from the LPD Megapixel system.

In online processing, data rates of up to 128 corrected megapixel images per second, with a latency to user processing of below 2 s, are routinely achieved, and 256 images s^{-1} (*i.e.* 1.792 Gigabytes s^{-1}) have been stably processed, as is shown in Fig. 7. This latency is measured from acquisition of a train on the detector to output to the Karabo bridge and

electronic reprint

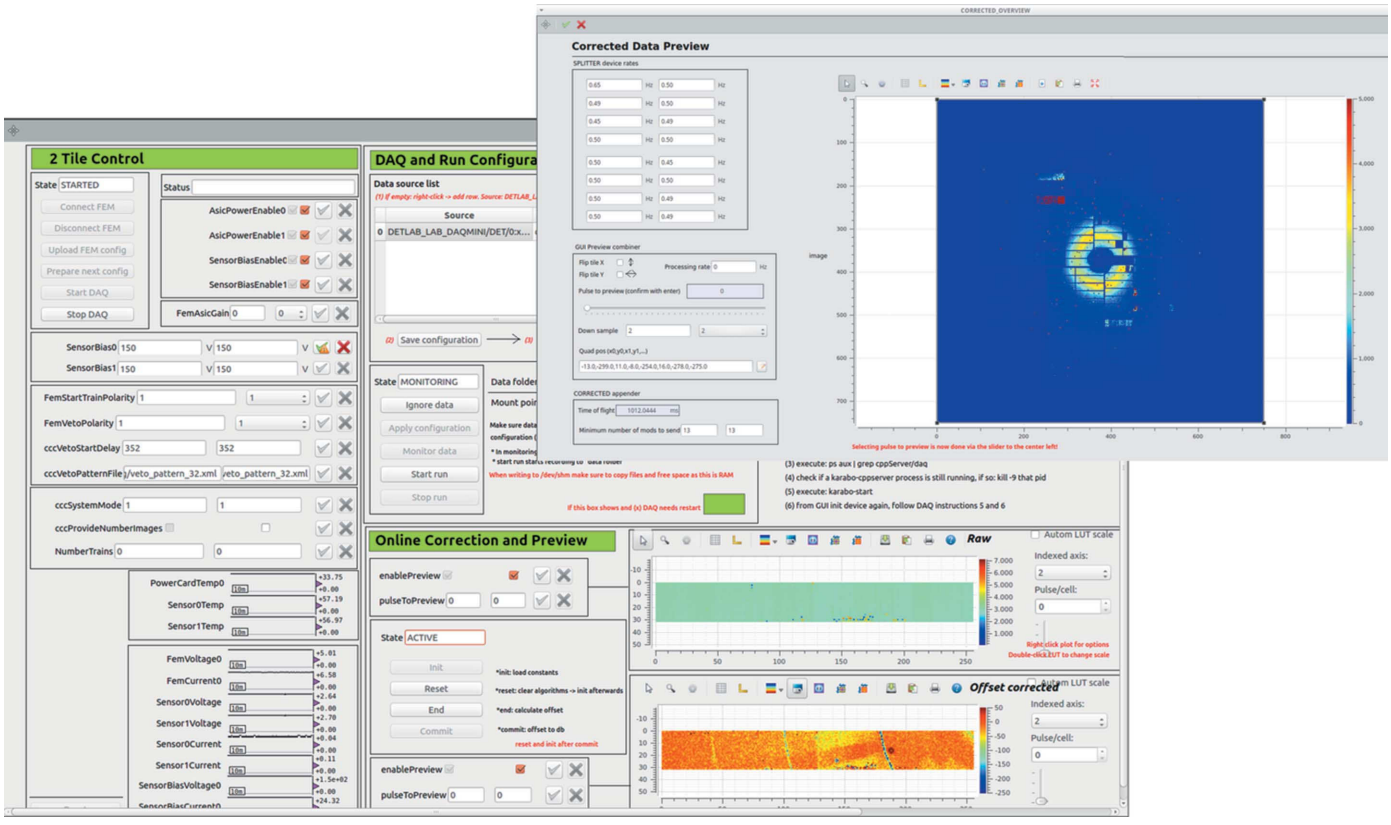


Figure 6
Karabo control and online preview scenes used during the first beam data acquisition with the LPD detector prototype at the FXE instrument. Diffraction images captured at 4.5 MHz are displayed. Panels on the background scene contain widgets for detector configuration, status and control, DAQ status and control, operating procedures, online previews of raw (top) and offset-corrected data (with visible diffraction rings), and calibration pipeline control and status. The foreground scene shows a corrected online preview of the LPD megapixel system.

includes offset and gain corrections, as well as train-matched combination of the 16 independent data streams for the detector modules into a single array.

5.4. Instrument and detector simulation

Karabo pipelines are also used for instrument and detector simulation activities at the European XFEL. Using pipelined inter-device communication, the SIMEX framework (Fortmann-Grote *et al.*, 2016, 2017) simulating SPB experiments and the X-ray Detector Simulation Pipelines (XDSPs) have been combined for start-to-end simulations of an XFEL beamline (Rüter *et al.*, 2015; Joy *et al.*, 2015).

5.5. Lessons learned

In early versions of Karabo development, serious delays of up to minutes were sometimes experienced when working with C++ servers hosting many devices. The most relevant development to overcome this was the replacement of function calls that block programme threads by, for example, asynchronous patterns with callback handlers for inter-device communication.

Furthermore, broadcasting system messages to inform, for example, about new devices caused delays on servers that run many devices within their processes. The problem was overcome by sending broadcasts only once to each server and then distributing them internally.

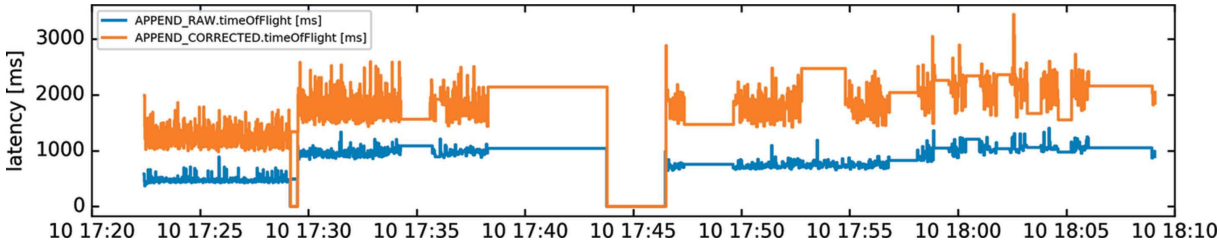


Figure 7
Latency for providing raw (blue) and corrected (orange) LPD megapixel images to user processing. Latency is measured from acquisition on the detector to output to the Karabo bridge. Steps in data are due to individual acquisition runs.

electronic reprint

Another important pitfall was the loss of the order of messages when posting them to the central event loop without further care.

6. Software engineering observations

The control and analysis software group at the European XFEL follows modern software engineering procedures: the group of 20+ software engineers is supported by an agile manager, coordinates work in daily stand-up meetings and uses a public backlog of activities. Requirements are captured and projects are carried out with iterative refinement and regular feedback from stakeholders.

Standard tools such as version control, unit, integration and system tests, regression tests, automatic test execution and continuous integration are used. Release cycles are planned and a dedicated test team ensures a high quality of new releases; installation of software and updates is automated by centralized deployment tools.

The demands of a research facility that provides services for users are diverse and often unpredictable, in particular during the start-up phase: operational requirements to support the first experiments compete with the build up and commissioning of new beamlines and instruments. The agile approach provides a flexible way of prioritizing requests and resolving them. However, prioritization of resource allocation to ensure operational service means that there are important parts of Karabo that are not completed yet or have accumulated technical debt and may need significant attention in the future.

7. Summary

Karabo is the main user interface for European XFEL staff supporting experiments and for visiting scientists carrying out their experiments. The system has been of central importance in commissioning and supporting early user experiments, resulting in the first publications of scientific results from the European XFEL (Grünbein *et al.*, 2018; Wiedorn *et al.*, 2018).

The decision to create a new distributed control system – Karabo – instead of re-using and attempting to modify an existing one to fulfil the facility's requirements has important implications: the flexible design of the framework, the unified treatment of control data and 'scientific' data within Karabo, and the ability to parallelize data analysis across distributed hardware opens opportunities to run facilities more effectively. On the other hand, developing such a software from scratch is a very significant task, and stabilizing any new software, in particular distributed software working with custom hardware, requires time.

Great progress has been made in the last two years, and Karabo is stable and used throughout the facility to enable beam transport, control, diagnostics, data acquisition, calibration and data analysis.

The focus now is on commissioning and supporting operation of a growing number of beamlines and instruments, supporting an increasing number of pulses per train, and, simultaneously, to continue improving the core of the Karabo software to best support users and the facility in the medium and long term.

Acknowledgements

The Control and Software Analysis (CAS) group and authors of this paper worked closely with other European XFEL scientific support groups and acknowledge their continuous efforts, input and cooperation. We thank the X-Ray Optics (XRO) led by Harald Sinn, the Vacuum (VAC) group led by Martin Dommach, the X-Ray Photon Diagnostics (XPD) group led by Jan Grünert, the Sample Environment (SE) group led by Joachim Schulz, the Undulator (UNSYS) group led by Joachim Pflüger, the Advanced Electronics (AE) group led by Patrick Gessler, the Information Technology and Data Management (ITDM) group led by Krzysztof Wrona, and the Detector Development (DET) group led by Markus Kuster, and the Laser (LAS) group led by Maximilian Lederer, as well as the scientific instruments FXE led by Christian Bressler, SPB led by Adrian Mancuso, SCS led by Andreas Scherz, SQS led by Michael Meyer, MID led by Anders Madsen, and HED led by Ulf Zastrau. We equally acknowledge the contributions from the Accelerator control group at DESY led by Tim Wilksen. All figures and pictures by the author(s) are published under a CC-BY 4.0 license (https://creativecommons.org/licenses/by/4.0/).

References

Altarelli, M. (2011). *Nucl. Instrum. Methods Phys. Res. B*, **269**, 2845–2849.

Altarelli, M., Brinkmann, R., Chergui, M., Decking, W., Dobson, B., Düsterer, S., Grübel, G., Graeff, W., Graafsma, H., Janos Hajdu, Jonathan Marangos, J. P., Redlin, H., Riley, D., Robinson, I., Rossbach, J., Schwarz, A., Tiedtke, K., Tschentscher, T., Vartaniants, I., Wabnitz, H., Weise, H., Wichmann, R., Karl Witte, A. W., Wulff, M. & Yurkov, M. (2006). *The European X-ray Free-Electron Laser.* Technical Design Report DESY 2006-097. DESY, Hamburg, Germany.

Ashiotis, G., Deschildre, A., Nawaz, Z., Wright, J. P., Karkoulis, D., Picca, F. E. & Kieffer, J. (2015). *J. Appl. Cryst.* **48**, 510–519.

Beg, M., Pepper, R. A. & Fangohr, H. (2017). *Am. Inst. Phys. Adv.* **7**, 056025.

Boukhelef, D., Szuba, J., Wrona, K. & Youngman, C. (2013). *Proceedings of the 14th International Conference on Accelerator and Large Experimental Physics Control Systems (ICALEPCS 2013)*, 6–11 October 2013, San Francisco, CA, USA, pp. 665–668. TUPPC045.

Boulay, D. du, Brockhauser, S., Chee, C., Chiu, K., Devadithya, T., Leow, R., McMullen, D. F., Quilici, R. & Turner, P. (2008). *Intl J. Online Eng.* **4**, 5–11.

Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E. & Yergeau, F. (1997). *World Wide Web J.* **2**(4), 27–66.

Bressler, C. (2011). *FXE Conceptual Design Report.* XFEL.EU TR-2011-005. European XFEL, Hamburg, Germany.

Bressler, C., Galler, A. & Gawelda, W. (2012). Technical Design Report XFEL EU TR-2012-008. European XFEl, Hamburg, Germany.

Brockhauser, S., Svensson, O., Bowler, M. W., Nanao, M., Gordon, E., Leal, R. M. F., Popov, A., Gerring, M., McCarthy, A. A. & Gotz, A. (2012). *Acta Cryst.* D**68**, 975–984.

electronic reprint

Dalesio, L. R., Hill, J. O., Kraimer, M., Lewis, S., Murray, D., Hunt, S., Watson, W., Clausen, M. & Dalesio, J. (1994). *Nucl. Instrum. Methods Phys. Res. A*, **352**, 179–184.

Dalheimer, M. (2002). *Programming with QT: Writing portable GUI applications on Unix and Win32.* Sebastopol: O'Reilly Media.

Daurer, B. J., Hantke, M. F., Nettelblad, C. & Maia, F. R. N. C. (2016). *J. Appl. Cryst.* **49**, 1042–1047.

DESY (2017). *Maxwell cluster*, https://confluence.desy.de/display/IS/Maxwell.

Esenov, S., Wrona, K. & Youngman, C. (2009). *European XFEL DAQ and DM Computing.* Technical Design Report – 2009 Public Version. European XFEL, Hamburg, Germany.

European XFEL(2017). *Scientific data policy*, http://www.xfel.eu/users/experiment_support/policies/scientific_data_policy/index_eng.html.

European XFEL (2018*a*). *Karabo-data: Python library and tools to process euxfel hdf5 files*, https://github.com/European-XFEL/karabo_data.

European XFEL (2018*b*). *Karabo Bridge protocol*, https://in.xfel.eu/readthedocs/docs/data-analysis-user-documentation/en/latest/karabo_bridge/protocol.html.

European XFEL (2018*c*). *Open source tools to support data analysis at European XFEL GmbH*, https://github.com/European-XFEL.

Fangohr, H., Beg, M., Bondar, V., Boukhelef, D., Brockhauser, S., Danilevski, C., Ehsan, W., Esenov, S. G., Flucke, G., Giovanetti, G., Goeries, D., Hauf, S., Heisen, B., Hickin, D. G., Khakhulin, D., Klimovskaia, A., Kuster, M., Lang, P. M., Maia, L., Mekinda, T., Michelat, A., Parenti, G., Previtali, H., Santos, A., Silenzi, J., Sztuk-Dambietz, J., Szuba, M., Teichmann, K., Weger, J., Wiggins, K., Wrona, L., Xu, C., Aplin, S., Barty, A., Kuhn, M., Mariani, V. & Kluyver, T. (2018). *Proceedings of the 16th International Conference on Accelerator and Large Experimental Control Systems (ICALEPCS2017)*, 8–13 October 2017, Barcelona, Spain, pp. 245–252. TUCPA01.

Folk, M., Heber, G., Koziol, Q., Pourmal, E. & Robinson, D. (2011). *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases (AD'11)*, 21–25 March 2011, Uppsala, Sweden, pp. 36–47.

Fortmann-Grote, C., Andreev, A., Appel, K., Branco, J., Briggs, R., Bussmann, M., Buzmakov, A., Garten, M., Grund, A., Huebl, A., Jurek, Z., Loh, N. D., Nakatsutsumi, M., Samoylova, L., Santra, R., Schneidmiller, E. A., Sharma, A., Steiniger, K., Yakubov, S., Yoon, C. H., Yurkov, M. V., Zastrau, U., Ziaja-Motyka, B. & Mancuso, A. P. (2017). *Proc. SPIE*, **10237**, 102370S.

Fortmann-Grote, C., Andreev, A. A., Briggs, R., Bussmann, M., Buzmakov, A., Garten, M., Grund, A., Hübl, A., Hauff, S., Joy, A., Jurek, Z., Loh, N. D., Rüter, T., Samoylova, L., Santra, R., Schneidmiller, E. A., Sharma, A., Wing, M., Yakubov, S., Yoon, C. H., Yurkov, M. V., Ziaja, B. & Mancuso, A. P. (2016). *arXiv:*1610.05980.

Foucar, L. (2016). *J. Appl. Cryst.* **49**, 1336–1346.

Furuhashi, S. (2008–2013). *Messagepack*, https://msgpack.org/.

Götz, A., Taurel, E., Pons, J., Verdier, P., Chaize, J., Meyer, J., Poncet, F., Heunen, G., Götz, E., Buteau, A., *et al.* (2003). *Proceedings of the 2003 International Conference on Accelerator and Large Experimental Physics Control Systems (ICALEPCS2003)*, Gyeongju, Korea, pp. 220–222. MP705.

Grünbein, M. L., Bielecki, J., Gorel, A., Stricker, M., Bean, R., Cammarata, M., Dörner, K., Fröhlich, L., Hartmann, E., Hauf, S., Hilpert, M., Kim, Y., Kloos, M., Letrun, R., Messerschmidt, M., Mills, G., Kovacs, G. N., Ramilli, M., Roome, C. M., Sato, T., Scholz, M., Sliwa, M., Sztuk-Dambietz, J., Weik, M., Weinhausen, B., Al-Qudami, N., Boukhelef, D., Brockhauser, S., Ehsan, W., Emons, M., Esenov, S., Fangohr, H., Kaukher, A., Kluyver, T., Lederer, M., Maia, L., Manetti, M., Michelat, T., Münnich, A., Pallas, F., Palmer, G., Previtali, G., Raab, N., Silenzi, A., Szuba, J., Venkatesan, S., Wrona, K., Zhu, J., Doak, R. B., Shoeman, R. L., Foucar, L., Colletier, J.-P., Mancuso, A. P., Barends, T. R. M., Stan, C. A. & Schlichting, I. (2018). *Nat. Commun.* **9**, 3487.

Grygiel, G., Hensler, O. & Rehlich, K. (1996). *Proceedings of the 1st International Workshop on Emerging Technologies and Scientific Facilities Controls (PCaPAC96)*, DESY, Hamburg, Germany.

Hapner, M., Burridge, R., Sharma, R., Fialli, J. & Stout, K. (2002). *Java Message Service*, p. 9. Sun Microsystems Inc., Santa Clara, CA, USA.

Hauf, S. (2017). *XFEL detector tools documentation*, https://in.xfel.eu/readthedocs/docs/pydetlib/en/latest/.

Heisen, B., Boukhelef, D., Esenov, S., Hauf, S., Kozlova, I., Maia, L., Parenti, A., Szuba, J., Weger, K., Wrona, K. & Youngman, C. (2013). *Proceedings of the 14th International Conference on Accelerator and Large Experimental Physics Control Systems (ICALEPCS2013)*, 6–11 October 2013, San Francisco, CA, USA, pp. 1465–1468. FRCOAAB02.

Hintjens, P. (2013). *ZeroMQ: Messaging for Many Applications.* Sebastopol: O'Reilley Media.

Hoyer, S. & Hamman, J. (2017). *J. Open Res. Softw.* **5**, 10.

Hunter, J. D. (2007). *Comput. Sci. Eng.* **9**, 90–95.

ISO (2011). Technical Committee: ISO/IEC JTC 1/SC 22 Programming languages, t. e. & system software interfaces, (2011). Iso/iec 14882:2011 information technology – programming languages – c++, https://www.iso.org/standard/50372.html.

Joy, A., Wing, M., Hauf, S., Kuster, M. & Rüter, T. (2015). *J. Instrum.* **10**, C04022.

Kluyver, T., Ragan-Kelley, B., Pérez, F., Granger, B. E., Bussonnier, M., Frederic, J., Kelley, K., Hamrick, J. B., Grout, J., Corlay, S., Ivanov, P., Avila, D., Abdalla, S., Willing, C. & Jupyter Development Team (2016). *Proceedings of the 20th International Conference on Electronic Publishing (ELPUB)*, pp. 87–90. Göttingen, Germany.

Kuster, M., Boukhelef, D., Donato, M., Dambietz, J.-S., Hauf, S., Maia, L., Raab, N., Szuba, J., Turcato, M., Wrona, K. & Youngman, C. (2014). *Synchrotron Radiat. News*, **27**(4), 35–38.

Mancuso, A. (2011). *Conceptual Design Report: Scientific Instrument Single Particles, Clusters, and Biomolecules (SPB).* Report TR-2011-007. European XFEL, Hamburg, Germany.

Mancuso, A. P., Reimers, N., Borchers, G., Aquila, A. & Giewekemeyer, K. (2013). *Technical Design Report: Scientific Instrument Single Particles, Clusters and Biomolecules (SPB).* Technical Report. European XFEL GmbH, Hamburg, Germany.

Mariani, V., Morgan, A., Yoon, C. H., Lane, T. J., White, T. A., O'Grady, C., Kuhn, M., Aplin, S., Koglin, J., Barty, A. & Chapman, H. N. (2016). *J. Appl. Cryst.* **49**, 1073–1080.

McKinney, W. (2011). *Python for High Performance and Scientic Computing* pp. 1–9. Sebastopol: O'Reilly Media.

Mekinda, L., Bondar, V., Brockhauser, S., Danilevski, C., Ehsan, W., Esenov, S., Fangohr, H., Flucke, G., Giovanetti, G., Hauf, S., Hickin, D., Klimovskaia, A., Maia, L., Michelat, T., Muennich, A., Parenti, A., Santos, H., Weger, K. & Xu, C. (2018). *Proceedings of the 16th International Conference on Accelerator and Large Experimental Control Systems (ICALEPCS2017)*, 8–13 October 2017, Barcelona, Spain, pp. 1142–1148. THBPA02.

Pérez, F. & Granger, B. E. (2007). *Comput. Sci. Eng.* **9**, 21–29.

Qt (2018). *Qt*, http://Doc.qt.io/qt-5/signalsandslots.html. [Accessed 12/09/2018.]

Rüter, T., Hauf, S., Kuster, M., Joy, A., Ayers, R., Wing, M., Yoon, C. H. & Mancuso, A. P. (2015). *2015 IEEE Nuclear Science Symposium and Medical Imaging Conference (NSS/MIC)*, pp. 1–4. IEEE.

Schäling, B. (2011). *The Boost C++ Libraries.* XML Press.

Stroustrup, B. (1995). *The C++ programming language.* Pearson Education India.

Van Rossum, G. & Drake, F. L. (2011). *The Python language reference manual.* Godalming: Network Theory Ltd.

electronic reprint

Van Rossum, G., Warsaw, B. & Coghlan, N. (2001). *Python*, http://www.python.org.

Walt, S. van der, Colbert, S. C. & Varoquaux, G. (2011). *Comput. Sci. Eng.* **13**, 22–30.

Waskom, M. (2012–2018). *Seaborn plotting library in Python*, https://seaborn.pydata.org/.

Wiedorn, M. O., Oberthür, D., Bean, R., Schubert, R., Werner, N., Abbey, B., Aepfelbacher, M., Adriano, L., Allahgholi, A., Al-Qudami, N., Andreasson, J., Aplin, S., Awel, S., Ayyer, K., Bajt, S., Barák, I., Bari, S., Bielecki, J., Botha, S., Boukhelef, D., Brehm, W., Brockhauser, S., Cheviakov, I., Coleman, M. A., Cruz-Mazo, F., Danilevski, C., Darmanin, C., Doak, R. B., Domaracky, M., Dörner, K., Du, Y., Fangohr, H., Fleckenstein, H., Frank, M., Fromme, P., Gañán-Calvo, A. M., Gevorkov, Y., Giewekemeyer, K., Ginn, H. M., Graafsma, H., Graceffa, R., Greiffenberg, D., Gumprecht, L., Güttlicher, P., Hajdu, J., Hauf, S., Heymann, M., Holmes, S., Horke, D. A., Hunter, M. S., Imlau, S., Kaukher, A., Kim, Y., Klyuev, A., Knoška, J., Kobe, B., Kuhn, M., Kupitz, C., Küpper, J., Lahey-Rudolph, J. M., Laurus, T., Le Cong, K., Letrun, R., Xavier, P. L., Maia, L., Maia, F. R. N. C., Mariani, V., Messerschmidt, M., Metz, M., Mezza, D., Michelat, T., Mills, G., Monteiro, D. C. F., Morgan, A., Mühlig, K., Munke, A., Munnich, A., Nette, J., Nugent, K. A., Nuguid, T., Orville, A. M., Pandey, S., Pena, G., Villanueva-Perez, P., Poehlsen, J., Previtali, G., Redecke, L., Riekehr, W. M., Rohde, H., Round, A., Safenreiter, T., Sarrou, I., Sato, T., Schmidt, M., Schmitt, B., Schönherr, R., Schulz, J., Sellberg, J. A., Seibert, M. M., Seuring, C., Shelby, M. L., Shoeman, R. L., Sikorski, M., Silenzi, A., Stan, C. A., Shi, X. T., Stern, S., Sztuk-Dambietz, J., Szuba, J., Tolstikova, A., Trebbin, M., Trunk, U., Vagovic, P., Ve, T., Weinhausen, B., White, T. A., Wrona, K., Xu, C., Yefanov, O., Zatsepin, N., Zhang, J. G., Perbandt, M., Mancuso, A. P., Betzel, C., Chapman, H. & Barty, A. (2018). *Nat. Commun.* **9**, 4025.

Zander, U., Bourenkov, G., Popov, A. N., de Sanctis, D., Svensson, O., McCarthy, A. A., Round, E., Gordeliy, V., Mueller-Dieckmann, C. & Leonard, G. A. (2015). *Acta Cryst.* D**71**, 2328–2343.

electronic reprint