12th NOBUGS Conference, Brookhaven National Laboratory, USA, 2018

Karabo: Lessons Learnt from a Control Framework Development

G. Flucke[a], M. Beg[a], M. Bergemann[a], V. Bondar[a], C. Danilevski[a], M. Duarte Trevisani[a], W. Ehsan[a], S. Esenov[a], R. Fabbri[a], H. Fangohr[a,c], D. Fullà Marsà[a], G. Giovanetti[a], D. Göries[a], S. Hauf[a], D. Hickin[a], E. Kamil[a], Y. Kirienko[a], A. Klimovskaia[a], T. Kluyver[a], A. Lemos[a], D. Mamchyk[a], T. Michelat[a], A. Parenti[a], H. Santos[a], A. Silenzi[a], C. Youngman[a], J. Zhu[a], S. Brockhauser[a,b]

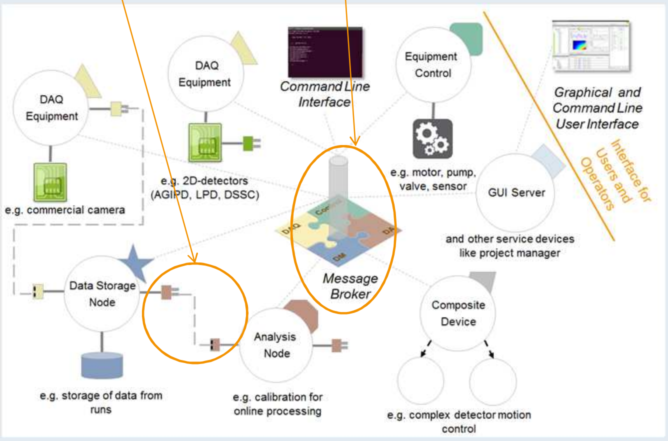
[a]European XFEL GmbH, Holzkoppel 4, 22869 Schenefeld, Germany, [b] Biological Research Center (BRC), Hungarian Academy of Sciences, Temesvri Krt. 62, Szeged, 6726, Hungary, [c]University of Southampton, Southampton, SO17 1BJ, United Kingdom

Abstract

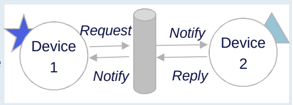
After surveying available solutions in 2011, the European XFEL decided to write a custom control and data processing system for its photon beam-lines and scientific instruments from scratch [1]. At the end of 2016, Karabo version 2 was released and has since then been used to commission the facility and execute the first user experiments.
A system complexity as required for these tasks could never be simulated beforehand which means that substantial improvements were needed to make Karabo as reliable and high-performance as needed to control the equipment. This poster guides through the main technical problems from which Karabo suffered (concentrating on the C++ implementation), and the means to overcome them to smoothly operate the European XFEL facility [3].

Karabo in a Nutshell

Karabo is designed to provide supervisory control and data acquisition for the European XFEL (EuXFEL). Hardware devices and system services are represented by Karabo devices of which many can run within the same server process, distributed among various control hosts. Devices communicate via a central message broker using language (C++ and Python) agnostic remote procedure calls (RPC). The Karabo design is event-driven, offering subscription to (remote) signals to avoid polling for parameter updates. Large data from detectors and for online monitoring the experiment is transported via flexible data pipelines using direct TCP connections. A specialty of the EuXFEL Karabo installation is that several hundred hardware devices can be controlled by a single programmable logic controller that communicates via a single TCP line with Karabo. Remote control is provided via a single C++ server with hundreds of devices, posing a particular challenge to concurrency.
The Karabo 2 release added features like alarm handling and a central configuration database, changed the pipeline protocol and refactored the C++ core to use a process wide common, multi-threaded event loop based on the **`boost::asio`** [2] library.


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
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

Synchronous Blocking Calls



- Remote calls trigger the execution of tasks
- May require a request to other devices or hardware
  - Can be done synchronously & asynchronously
- Synchronous: may block a programme thread
  - Can lead to thread starvation and big delays for further messages until reply received

→ **Avoid synchronous** and other **blocking constructs** where possible
  - E.g. when connecting to parameter updates of other devices
  - Where asynchronous refactoring is cumbersome, temporarily extend the thread pool
- Additions to Karabo to support these solutions:
  - Possibility to postpone the reply for a request to another thread (**`AsyncReply`**)
  - Failure handling for asynchronous requests
    - ► Backward compatibility forbade to just add a flag to asynchronous handlers – independent success and failure handlers complicate asynchronous call chains

Locking Mutexes to Avoid Parallel Execution

- Karabo used mutex locks to guarantee that an RPC method is not executed in parallel to itself in different threads
- For long (blocking) methods this can block many programme threads
  - E.g. when instantiating many devices by calling the instantiate method of a server
  - Can lead to thread starvation and big delays for further messages
- Order can get lost since undefined order of acquiring the lock

→ **Ensure sequential execution** by properly queuing **first-in first-out**
  - E.g. using a (proper – see on the right) **`strand`** object

Limitations of `boost::asio::strand`

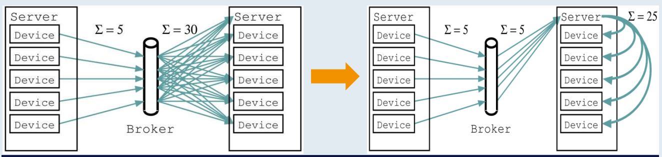
- Boost's **`strand`** provides ordered execution on a multi-threaded event loop
  - Handlers posted to one **`strand`** are executed one after another in order of posting
- Boost's **`strand`** does not guarantee that handlers posted to different strands can be executed in parallel
  - Several strands can effectively work like a single one, on a basically random basis
  - Problematic if a handler's execution takes very long and another handler reads TCP messages: reading can get extremely slow

→ **Abandon use of `boost::asio::strand`**
  - Ordering may already be guaranteed by programme logic (even using bare event loop)
  - Else replace by new Karabo strand that implements the first-in first-out queue logic

Congestion by Broadcasting Messages

- Karabo system topology is fully dynamic: no central instance knows which devices exist
- Therefore relies on broadcast messages received by all (e.g. "instance new" or "gone")
- That scales badly between two large C++ servers hosting hundreds of devices each
  - If one servers starts all its devices within 10 seconds, the broker will distribute N messages to each of the N devices (and the server instance) on the other server
  - Leads to N * (N+1) messages, e.g. about 25 kHz for 10 seconds for N = 500
- Karabo's broker client library and/or serialisation cannot cope with this load
  - Delay of other processing on the receiving server of easily a minute

→ **Send broadcast messages only once to each C++ server**
  - Server distributes locally to its devices: networking and serialisation reduced a lot



Data Copies in Serialisation of Large Data

- Karabo version 2 introduced a dedicated container for large data: **`NDArray`**
  - Can adopt or view raw memory
  - Behaviour similar to **`numpy.ndarray`** in Python
  - No data copies between Python and C++ for Python bindings
- Karabo's TCP interface required a single raw data buffer for handling **`NDArray`**'s data and meta data
  - Large data was copied unnecessarily
- Similarly, de-serialisation did extra copies

→ **Introduce a `BufferSet` to TCP and serialisation interfaces**
  - Internal data buffer of **`NDArray`** not copied anymore – only small meta data
  - Pipeline connections got six times faster, transferring reliably 1.35 GByte/s

Losing Message Order on Event Loop

```
set("motor/A",
    "targetPosition", 2);
execute("motor/A", "move");
```

- Order of messages sent from one device to another can matter a lot
- JMS broker used by Karabo ensures order
- Posting a handler to a multi-threaded event loop (from **`boost::asio`**) does not guarantee order (mis-order happens rarely, but especially on busy systems)

→ **Keep order by** placing relevant handlers in a **sequential queue**
  - E.g. using a (proper – see above) **`strand`** object

Conclusions

As Karabo installation sizes grew during commission the EuXFEL photon beam-lines and instruments and while carrying out the first experiments producing large data from fast detectors, several deficiencies were discovered in the C++ implementation. Unnecessary data copies and synchronous or blocking calls executed on the central multi-threaded event loop led to unacceptable delays on systems under load.
Fortunately, critical synchronous communication patterns could be switched to asynchronous behaviour, and data copies could be avoided for large pipeline data, both without the need to change any code of hardware control devices. Today, Karabo reliably controls experiments at EuXFEL instruments [3], providing the benefits of tight integration of instrument control and online data processing to the operators.
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 Karabo's middlelayer interface, written in single-threaded Python.

References

[1] B.C. Heisen *et al.*, "Karabo: an integrated software framework combining control, data management and scientific computing tasks",14th International Conference on Accelerator & Large Experimental Physics Control Systems, ICALEPCS 2013, San Francisco, U.S.A.
[2] B. Schäling, "The Boost C++ Libraries". XML Press, 2011.
[3] S. Hauf *et al.*, "The Karabo Distributed Control System", submitted to Journal of Synchrotron Radiation