Karabo goes AMQP:
Replacement of the Core Communication Broker

Dr. Gero Flucke
for the Controls group @ European XFEL GmbH

September 25th, 2024

Outline

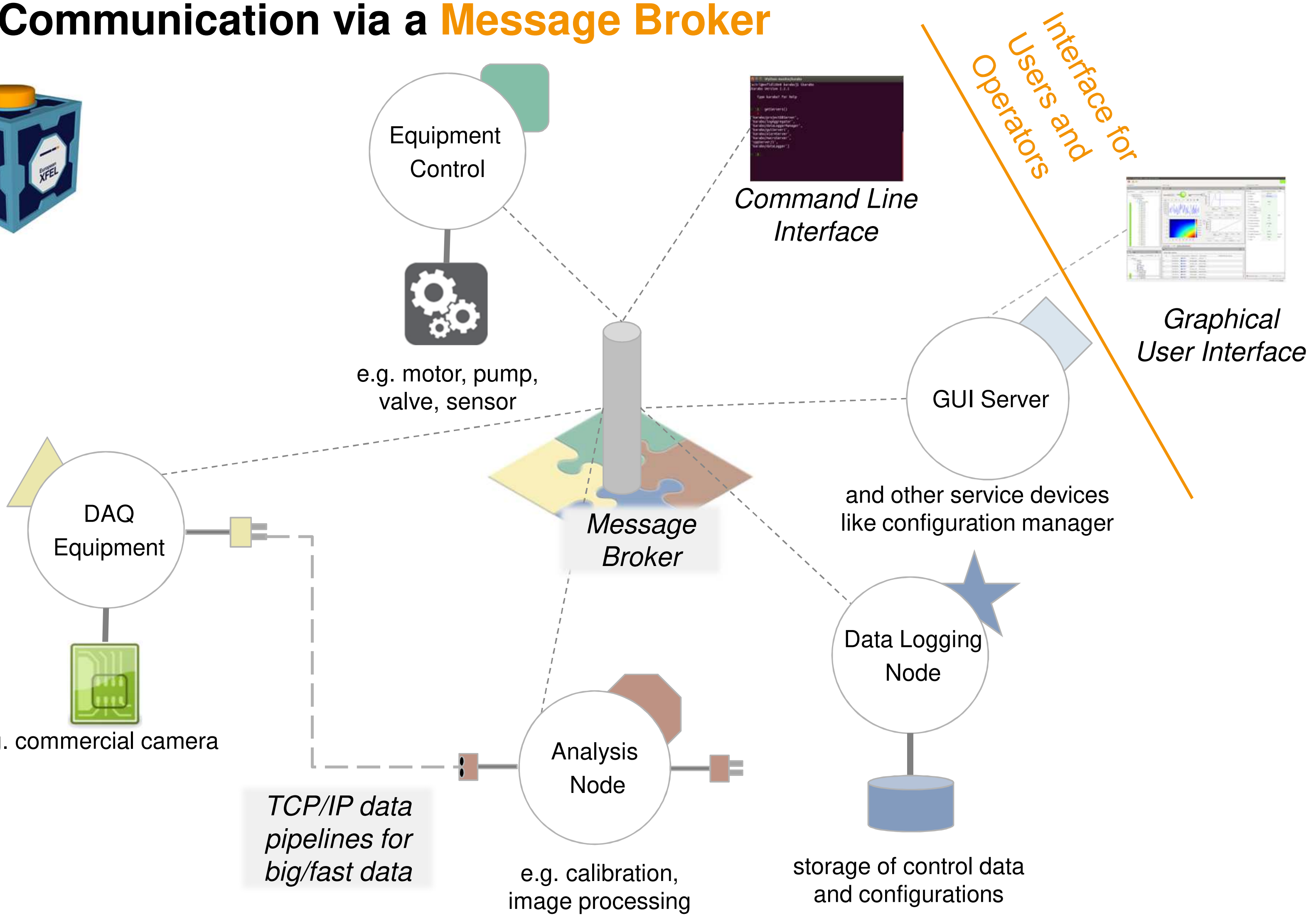
- Karabo Communication Basics
- Struggles with the original JMS broker implementation
- Refactoring Strategy
- Timeline
  - Refactoring
  - Deployment
  - Issues on the way
- Summary

Karabo:
Supervisory **C**ontrol and **D**ata **A**cquisition
at the beamlines and instruments of the
European XFEL (Hamburg Metropolitan Area)

Karabo: Device Based Communication via a Message Broker

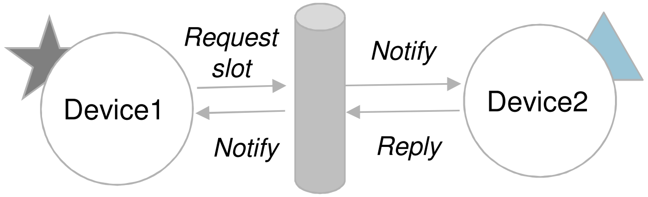
Self-describing
Karabo Devices

- Equipment control, e.g. motors, valves,…
- Detectors
  - e.g. cameras
- Online data analysis
- Data Logging
- Other system services
  - GUI entry point
  - DAQ for big/scientific data (not shown)

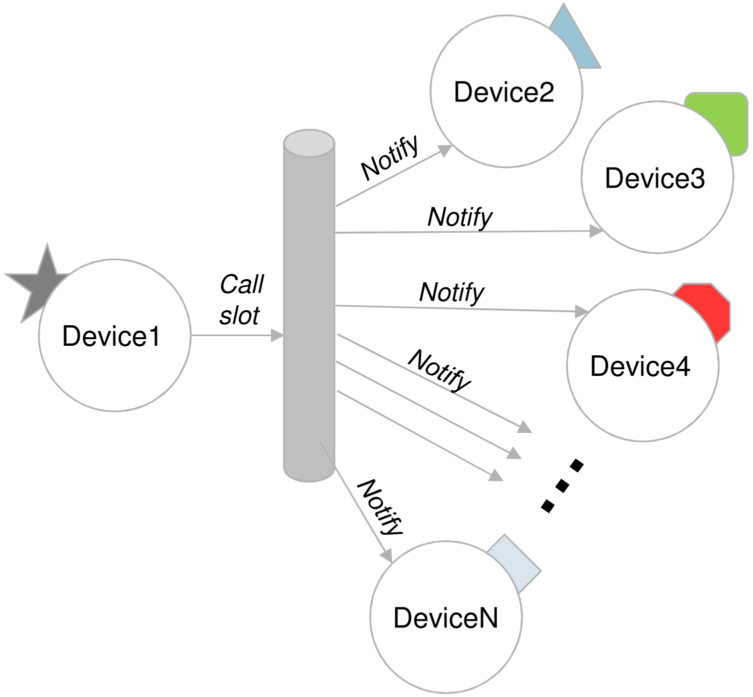

Interface for Users and Operators
Equipment Control
e.g. motor, pump, valve, sensor
Command Line Interface
Graphical User Interface
GUI Server
and other service devices like configuration manager
Message Broker
DAQ Equipment
e.g. commercial camera
TCP/IP data pipelines for big/fast data
Analysis Node
e.g. calibration, image processing
Data Logging Node
storage of control data and configurations

Karabo Communication Patterns

- 1-to-1: Request and reply
  - Device registers methods as “slots”.
  - Request from remote with up to four arguments
    - ► Reply if done with up to four values.
    - ► Requester can suppress reply (fire-and-forget)

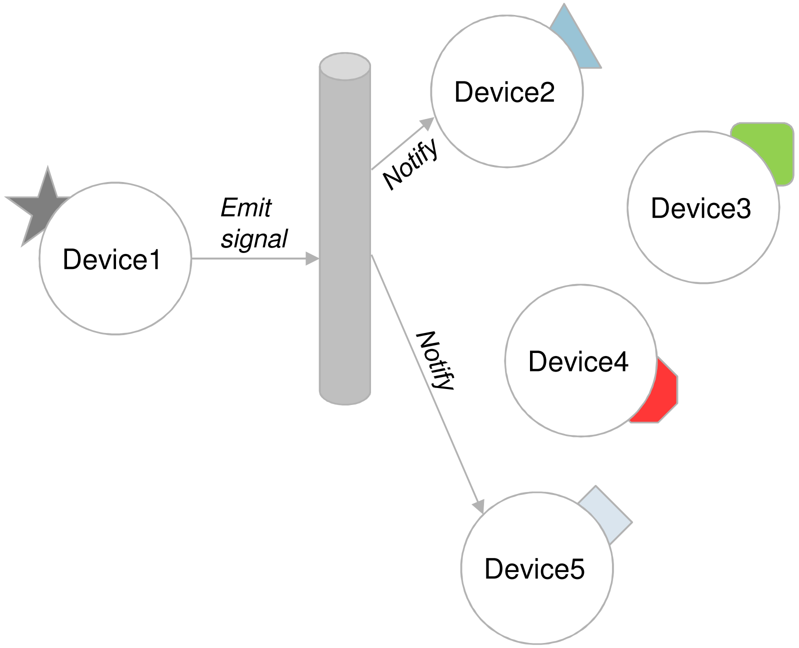


- 1-to-all: Broadcast (for system purpose only)
  - Always fire-and-forget
  - Still costly, so used rarely:
    - ► System topology: instance new and gone
    - ► Problematic device states (UNKNOWN, ERROR)



Karabo Communication Patterns (ctd.)

- Publish/subscribe
  - Devices (2 & 5) subscribe slots to a remote “signal”.
  - When signal is “emitted”,
    all *subscribed* slots are called.
    - ▶ No publishing overhead for “popular” devices
    - ▶ **Karabo framework is completely event-driven**:
      regular **polling obsolete**.



Karabo APIs

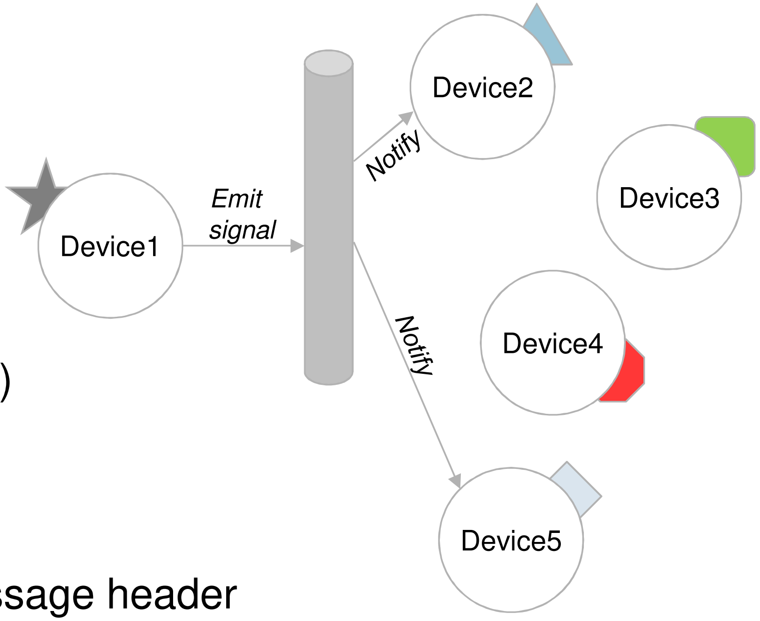
- **C++**:
  - "First language" of Karabo
  - High performance framework devices (GUI server, data logging) and digitizers, some cameras
- Python "**Bound**":
  - Python bindings on top of C++ (now using `pybind11`)
    ► Communication completely covered by C++
  - Many similarities with C++ ➔ not very Pythonic
- Python "**Middlelayer" (MDL)**:
  - Pure Python (early use of `asyncio` library, single thread)
  - Nowadays most popular API, not only for middlelayer devices

Karabo installation size e.g. about:

- 2700 devices
- 400 k properties
- 1.2 kHz message rate *to* broker
- 4.3 kHz message rate *from* broker

(13 such installations at EuXFEL)

Original Karabo Broker Implementation



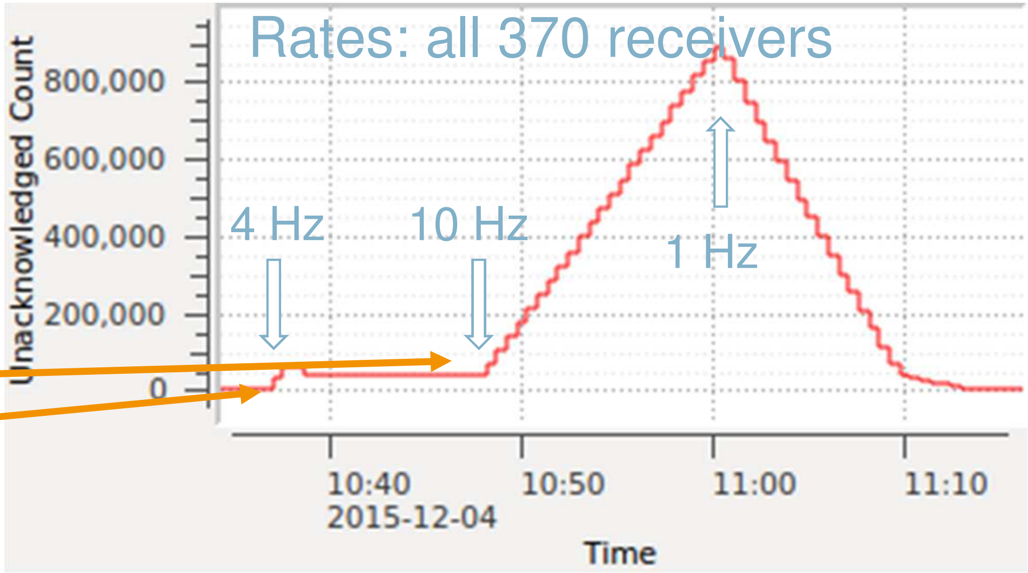
- Broker: Java Messaging System (JMS) – OpenMQ
- Client library: OpenMQc
  - C/C++ library also directly called from pure Python API (MDL)
- Each device subscribes once to a Karabo “topic” (e.g. for one of the EuXFEL instruments)
  - Broker filters messages according to target device ids in message header
    - ► message header carries target information
    - ► signal subscription not on the broker, but on Karabo device emitting the signal

Problems with JMS Broker and OpenMQc Library

- Global message backlog
  - Messages not consumed kept on broker
    - ► If max. reached, broker refuses messages: **No communication anymore!**
  - Causes:
    - ► Badly behaving processes
    - ► Many receivers in one process

- Message drop
  - Hacked into the code as last rescue
  - Triggered memory leak in OpenMQc

- OpenMQc library not maintained
  - Memory leak problem fixed only four years after we reported in 2015



Problems worked around over years:
- Except backlog from badly behaving processes
  - ► Closely monitoring backlog
  - ► On-call staff hunting process to kill
  - ► Failed very few times per year (potential loss of few hours of beamtime)

Refactoring Strategy I

- Broker communication is critical:
  - ➔ Need to be able to switch back to old broker at any time
    - ➔ Allow switching back and forth by just changing the environment variable **`$KARABO_BROKER`**

- Introduce abstract base class for broker communication (both APIs: C++ and MDL)
  - Concrete class for each broker protocol supported
  - Dynamically choose concrete class according to protocol part of broker address
    - ► tcp://somehost:7777 - tcp ➔ `JmsBroker`
    - ► amqp://anotherhost:5672 – amqp ➔ `AmqpBroker`
    - ► …

- Long timeline of project:
  - No long lived feature branch in git (fear of divergence)
  - But merge smaller changes
    - ► always keeping JMS communication intact

Refactoring Strategy II

- Tests, tests, tests!
  - Continuous integration
    - Unit test of communication class `SignalSlotable` runs for all supported brokers
    - Integration tests are repeated for JMS and most promising new broker
  - Big test suit of our TestPortal for each release candidate and intermediate alpha releases
    - Includes tests beyond Karabo framework (device packages)
    - Integrated new stress tests
      - high data rate
      - message order

Timeline of Refactoring

- Autumn 2015: Worrying rate tests with message loss (data logging)
- Nov. 2016: Karabo 2 released
  - To commission the facility
- Sept. 2017: First EuXFEL instruments go into production
- 2018: First serious studies of alternative brokers
  - Mainly studies concerning MQTT protocol
- April 2020: Official start of internal project to investigate broker protocols
  - MQTT: Liked for IoT applications
  - AMQP: Wall Street proven
- Nov. 2020: `Broker` base class released in all APIs

Timeline of Refactoring (ctd.)

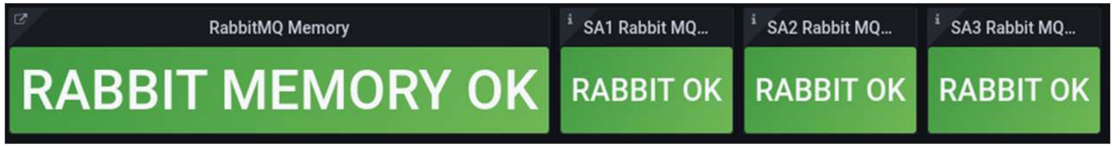
- May 2021: Experimental MQTT broker support
  - Caveat: MQTT does **not** keep messages in order if sent from A to B via different routes
  - Karabo operation requires order, but signals and direct slots cannot use the same route
    ► complicated and fragile custom code needed to keep order

- Oct. 2021: Experimental AMQP and Redis broker support
  - Message order integration test on CI

- 2022: Work on
  - Robustness
  - “Fail-over” for cases of lost broker connection

Timeline of Deployment

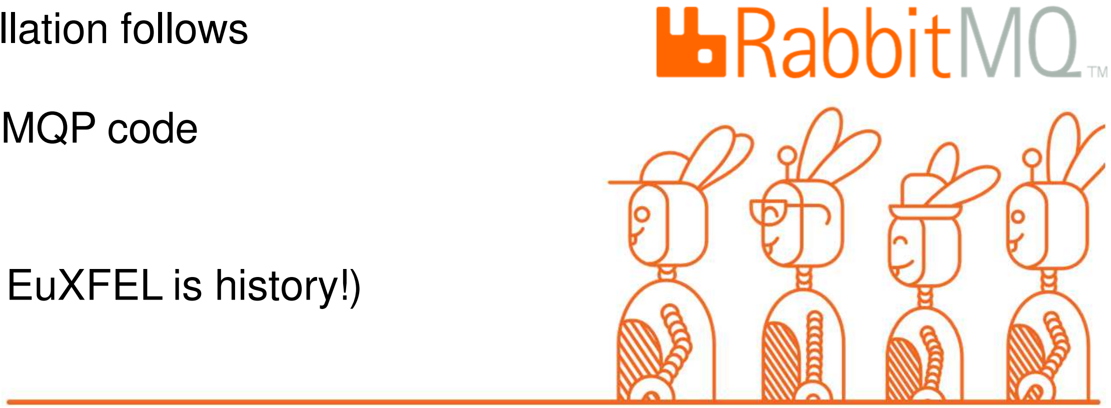
- March 2023: Deploy use of AMQP at first instrument SXP
  - RabbitMQ broker provides ready-to-use monitoring
  - Smooth start:
    - ► Just like a normal Karabo deployment!


RabbitMQ Memory | SA1 Rabbit MQ... | SA2 Rabbit MQ... | SA3 Rabbit MQ...

RABBIT MEMORY OK | RABBIT OK | RABBIT OK | RABBIT OK

SXP, CTRL and DAQ staff
March 2023

- July 2023: Instrument with biggest control installation follows
- Oct. 2023: Fix of a race condition in Karabo's AMQP code
  - Lead occasionally to crashes
- Dec. 2023: Deploy AMQP facility wide (JMS @ EuXFEL is history!)



Issues and Actions after Full AMQP Deployment

- Jan. and April 2024: Crash of one RabbitMQ broker process (not Karabo)
  - Fixed with hardware replacement and RabbitMQ upgrade
  - “Fail-over” did not work

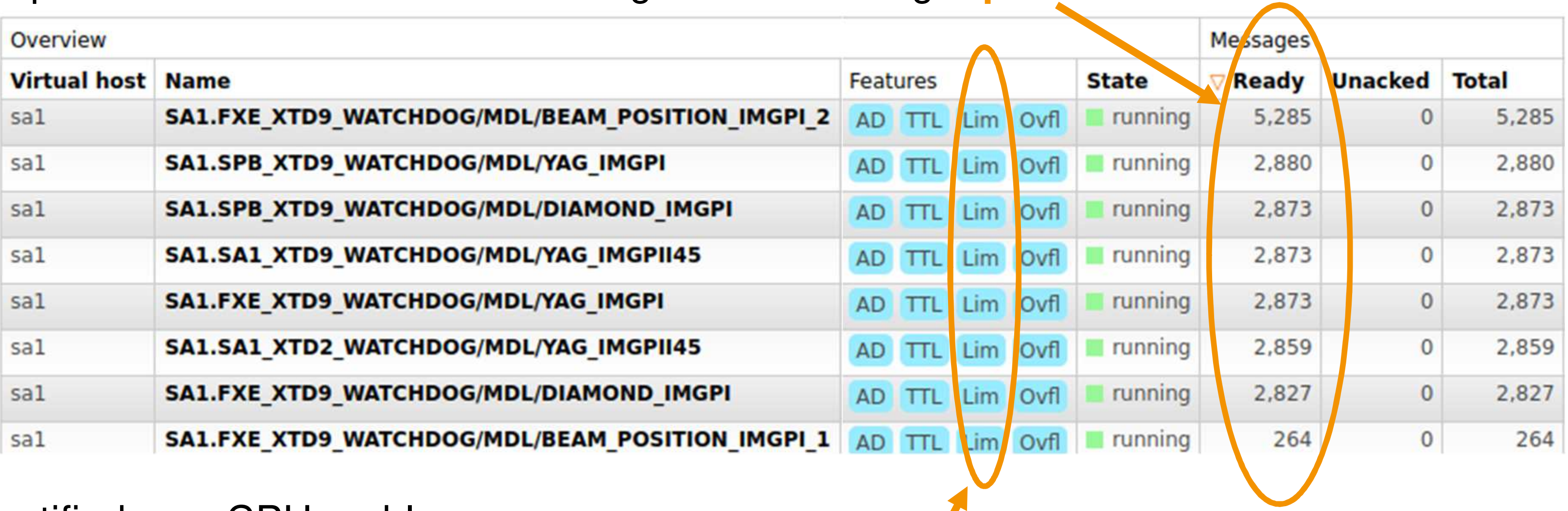
- Spurious missing broker subscriptions
  - If many subscriptions are run concurrently (e.g. if a data logger starts)

- Repeated crashes when many devices in a single process start concurrently

- ➔ Spring 2024: Decision to rewrite the Karabo C++ interface to AMQP from scratch
  - Incl. unit tests with concurrency situations
  - Released June 2024
    - ► Deployed for Karabo “backbone” and DAQ in July 2024
    - ► “Fail-over” postponed to November release

Issues and Actions after Full AMQP Deployment (ctd.)

- Apr./Sept. 2024: AMQP Broker monitoring shows message **queues** for devices

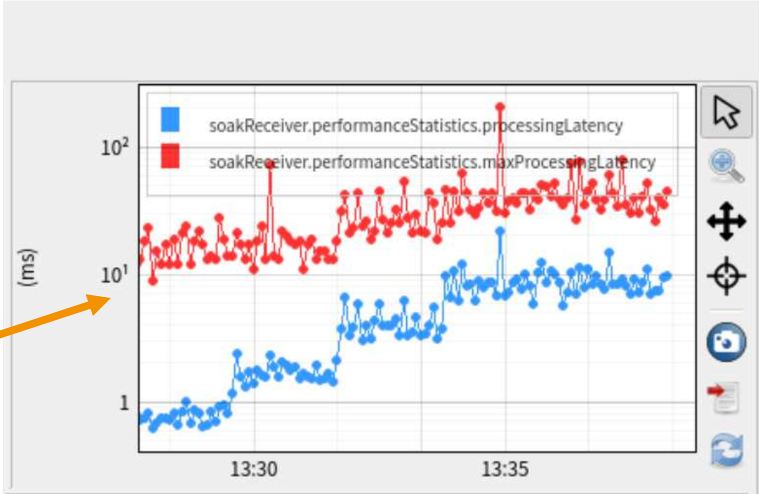


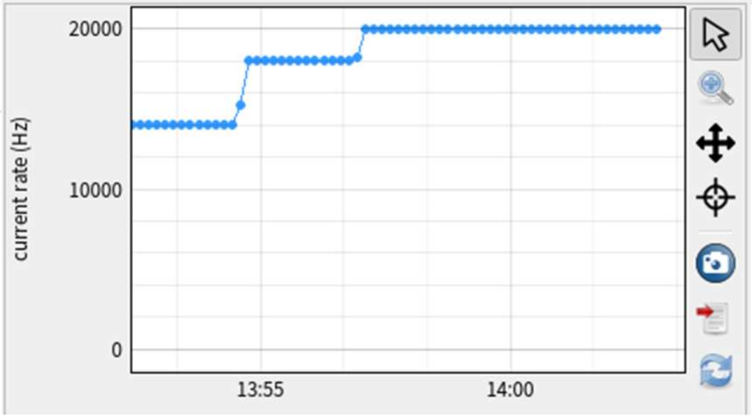
Overview				Messages		
Virtual host	Name	Features	State	▽ Ready	Unacked	Total
sa1	SA1.FXE_XTD9_WATCHDOG/MDL/BEAM_POSITION_IMGPI_2	AD TTL Lim Ovfl	🟩 running	5,285	0	5,285
sa1	SA1.SPB_XTD9_WATCHDOG/MDL/YAG_IMGPI	AD TTL Lim Ovfl	🟩 running	2,880	0	2,880
sa1	SA1.SPB_XTD9_WATCHDOG/MDL/DIAMOND_IMGPI	AD TTL Lim Ovfl	🟩 running	2,873	0	2,873
sa1	SA1.SA1_XTD9_WATCHDOG/MDL/YAG_IMGPII45	AD TTL Lim Ovfl	🟩 running	2,873	0	2,873
sa1	SA1.FXE_XTD9_WATCHDOG/MDL/YAG_IMGPI	AD TTL Lim Ovfl	🟩 running	2,873	0	2,873
sa1	SA1.SA1_XTD2_WATCHDOG/MDL/YAG_IMGPII45	AD TTL Lim Ovfl	🟩 running	2,859	0	2,859
sa1	SA1.FXE_XTD9_WATCHDOG/MDL/DIAMOND_IMGPI	AD TTL Lim Ovfl	🟩 running	2,827	0	2,827
sa1	SA1.FXE_XTD9_WATCHDOG/MDL/BEAM_POSITION_IMGPI_1	AD TTL Lim Ovfl	🟩 running	264	0	264

- Identified as a CPU problem
  - ► Overloaded host or Python (MDL) process
- No danger for communication in full installation (as the backlog of the JMS broker was)
  - ► Each message queue on the broker has a **limit**
  - ► Problem stays local

Performance Reached

- C++
  - Single device on a server
    - ▶ Withstands receiving small messages: 20 kHz
    - ▶ Can send small messages at 17 kHz
  - Server withstands simultaneous high sending and receiving rates
    - ▶ Single receiver device 9.0-9.5 kHz
    - ▶ 20 kHz sending rate (latencies < 100 ms)
      - 200 senders at 100 Hz each
      - 1000 senders at 20 Hz
  - Note: Much contingency
    - ▶ Even GUI servers and data loggers receive less than 1 kHz
- MDL (i.e. single threaded Python)
  - Can send and receive in total ≥ 2 kHz
  - Side effect: now pure Python broker client library
    - ➔ simplifies distribution


soakReceiver.performanceStatistics.processingLatency
soakReceiver.performanceStatistics.maxProcessingLatency
(ms)
10^2
10^1
1
13:30
13:35



Summary

- Karabo control system communicates via a message broker
  - Some deficiencies of the originally used JMS broker and OpenMQc client library spotted early

- Replacing core technology in a used system is delicate
  - Always need to be able to switch back
  - Requires confidence in test coverage
  - About five years from first serious studies to full AMQP deployment
    - ► There are always short term goals that prevent more rapid progress
    - ► Some things need to be iterated

- The effort was worth it:
  - No global danger for communication in a Karabo installation (less emergency call)
  - Higher performance
  - RabbitMQ broker ready for encryption