Karabo Overview

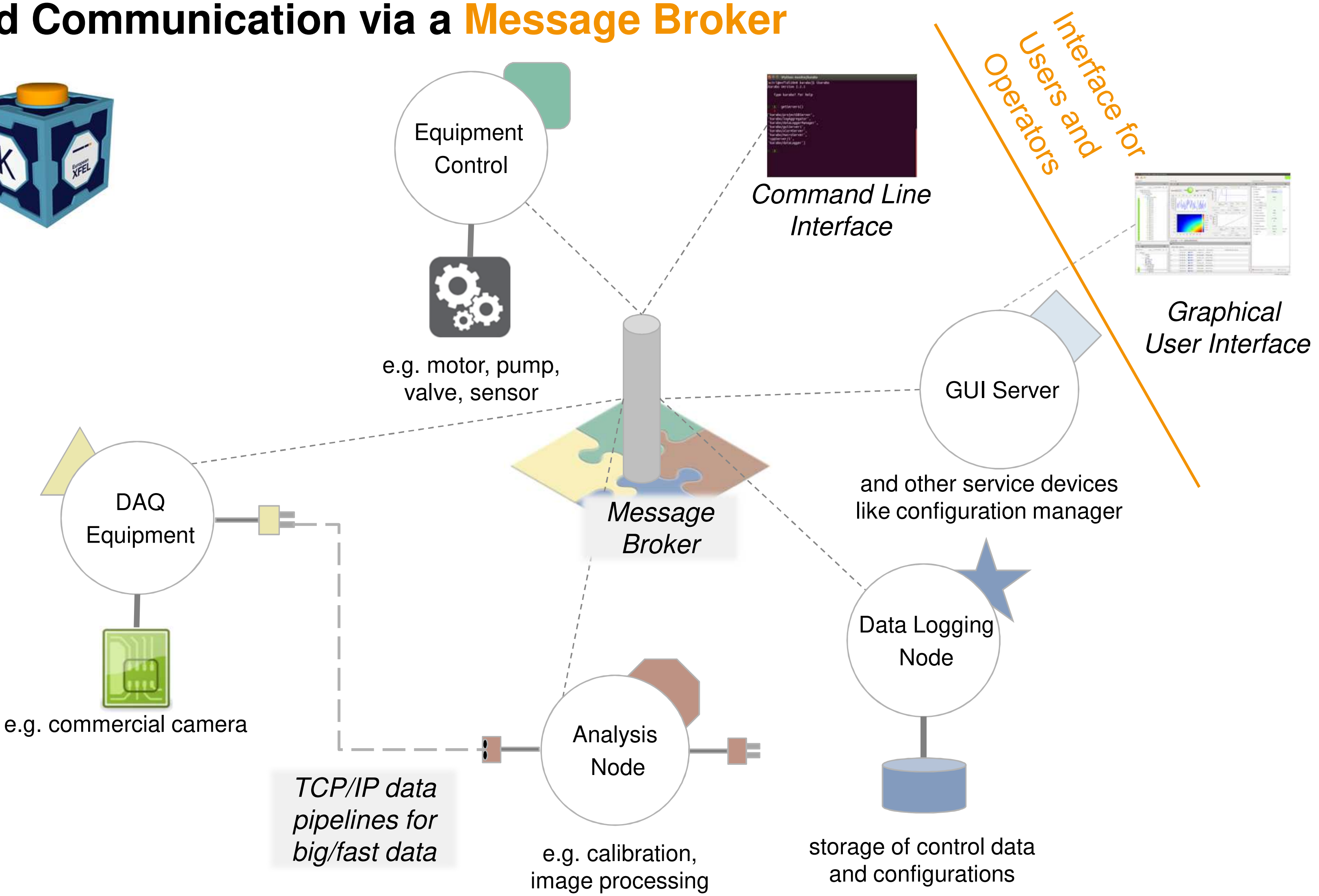
Dr. Gero Flucke
for the Controls group @ European XFEL GmbH

Satellite Workshop:
An introduction to developing in the Karabo SCADA Framework

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)

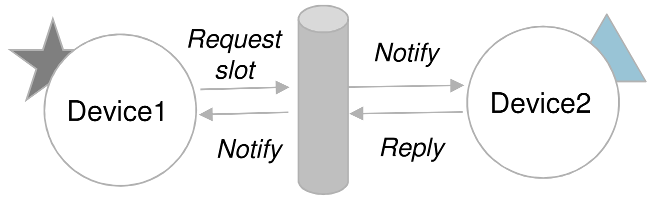


Core Components of Karabo

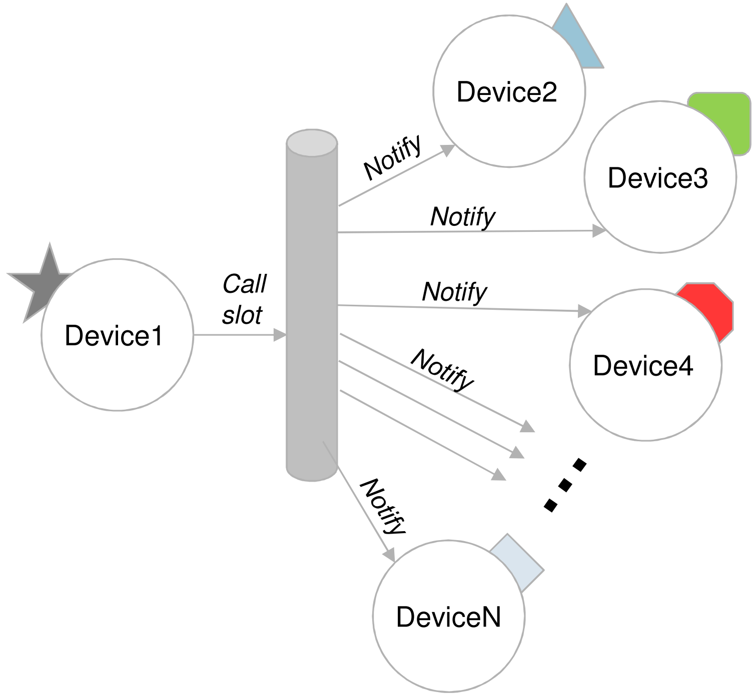
- Device
  - Core controllable object, providing e.g.
    - ► Equipment control: interface to motor, pump, valve, camera, etc.
    - ► Data provider: camera, spectrometer, customized 2D detectors
    - ► Data analysis: calibration, beam position extraction, etc.
    - ► Coordination of other devices (“middlelayer”)
    - ► System service: data logging, GUI server, project (configuration, etc.) database,…
  - Self-description (schema):
    - ► Properties (read-only, init-only, reconfigurable), commands – device state aware
- Device server: Program “hosting” devices (detail: in bound Python API launches them)
- Broker: Core (3rd party) component distributing control messages
- Command line client (both Python APIs): ikarabo, karabo-cli
- Generic, but customizable GUI

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)

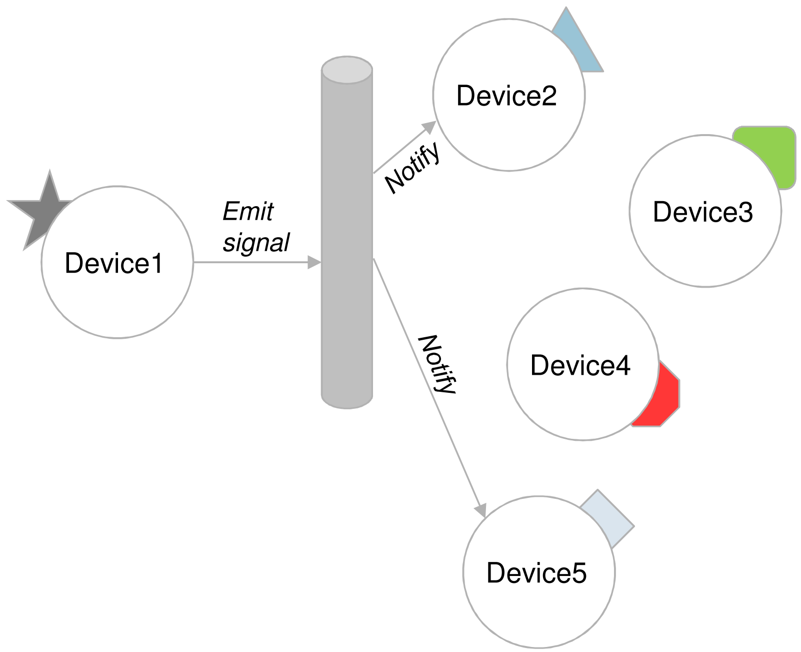


- 1-to-all: Broadcast
  - Always fire-and-forget
  - Still costly, so used rarely:
    - ► System topology: instance new and gone
    - ► Problematic device states (UNKNOWN, ERROR)



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**.



Hash: Karabo's Flexible Data Container

- A nested key-value container with attributes:
  - key: string
    - ▶ direct nested access: separate key levels by dot: `h.get("key1.key2.key3")`,
  - value: any type,
  - attributes per value: another key-value container.

- Hash available in all three Karabo APIs:
  - C++
  - Python
    - ▶ "Bound" (C++ bindings)
    - ▶ "Middlelayer" (pythonic)

- Serialisation to XML and binary format.
  - Supported data types:
    - ▶ Scalars, complex, strings, Hash and vectors thereof,
    - ▶ "NDArray" for pipelines,
    - ▶ "ImageData": NDArray and meta data

```
In [1]: from karabo.bound import Hash

In [2]: h = Hash('a', 'square')

In [3]: h['b.c'] = 42

In [4]: h
Out[4]:
'a' => square STRING
'b' +
  'c' => 42 INT32

In [5]: h.setAttribute('a', 'colour','red')

In [6]: h
Out[6]:
'a' colour="red" => square STRING
'b' +
  'c' => 42 INT32
```

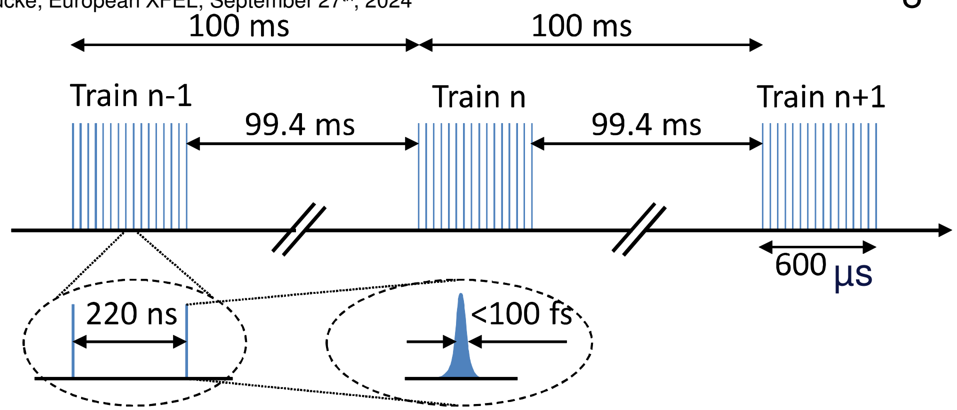
Remotely Callable Methods: Slots and Commands

- Slots can have up to four arguments and return values
  - Scalars, bool, string, Hash (and vectors of any of these)

- This flexibility should be restricted to framework functionality and some specific protocols
  - These protocols nowadays usually just use “Hash-in, Hash-out”

- Slots exposed in device schema (i.e. exposed to GUI): “commands”
  - No arguments
    - ► E.g. for motor: 1) set “targetPosition”, 2) execute “move” command
  - Return value mostly irrelevant/ignored (in doubt return device state?)
  - Can be restricted to specific device states
  - Should quickly return (<< 5 seconds)
  - Longer actions like slow movements are just “triggered”:
    - ► “move” command sets state to MOVING, starts movement and returns
    - ► when target reached, go back to state ON
    - ► “targetPosition” can be reconfigured again, but not while in state MOVING

Timestamps in Karabo



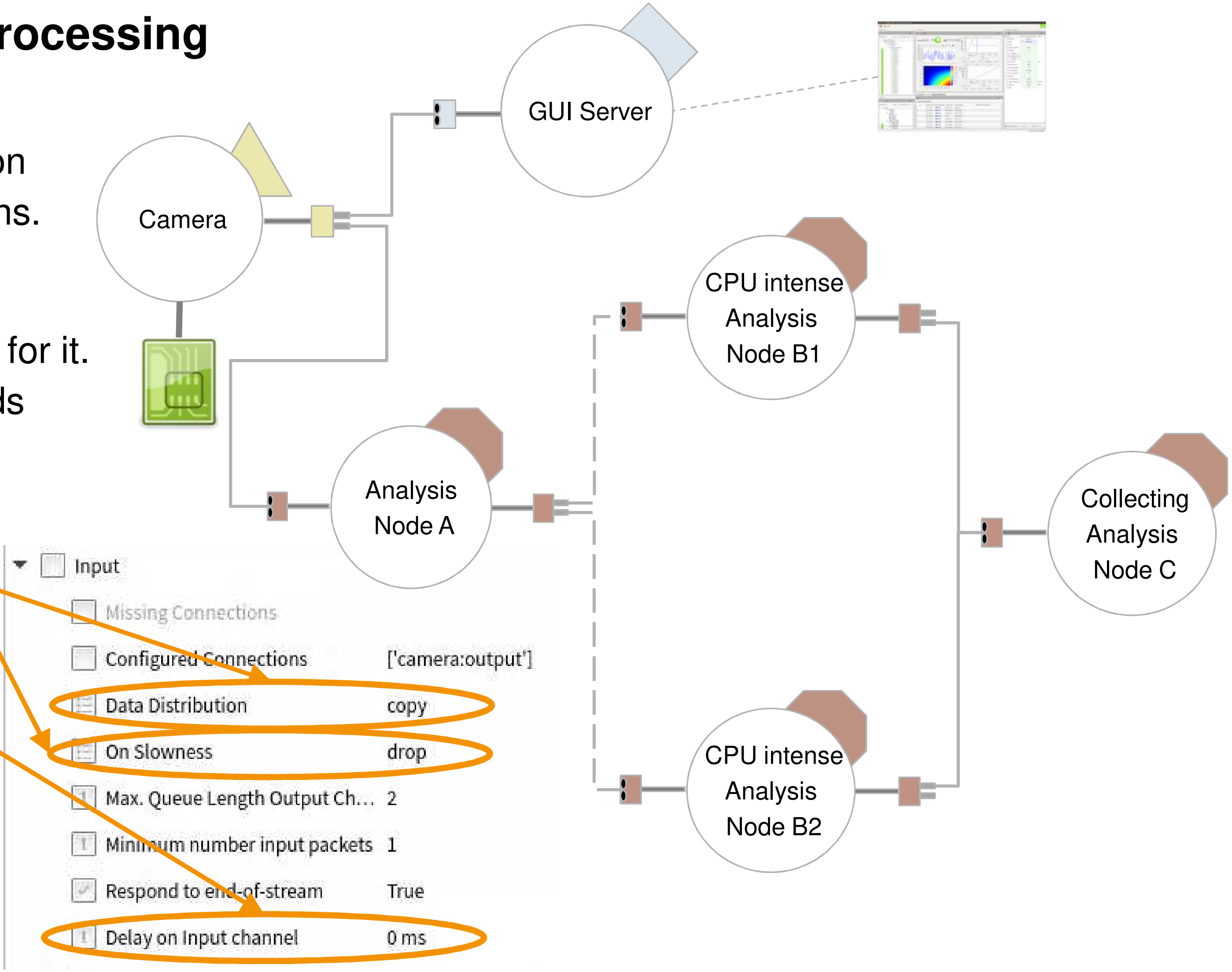
- Timestamps consist of three uint64 numbers
  - full seconds of unix epoch (since Jan 1st, 1970): "sec"
  - attoseconds (10^{-18}): "frac"
  - train id ("tid") – uniquely identifies each of the 10 Hz trains of up to 2700 photon pulses
  - Stored as Hash attributes of device properties

```
In [1]: from karabo.bound import Hash, Timestamp
^[[A
In [2]: h = Hash("a", 1)

In [3]: now = Timestamp()

In [4]: now.toHashAttributes(h.getAttributes("a"))

In [5]: h
Out[5]: 'a' sec="1644422351" frac="621966905000000000" tid="0"
=> 1 INT32
```

- Timing sources:
  - Ideally hardware source synchronized with XFEL accelerator timing system
  - Property update without specified stamp:
    - ► "sec" and "frac" from local system clock (usually synchronized within few ms via NTP)
    - ► "tid" extrapolated from "signalTimeTick" that provides time, train id and train repetition frequency (distributed at 1 Hz by TimeServer)

Karabo Pipelines for Data Processing

- Complement broker communication
  - Using direct TCP/IP connections.
- Designed for (large) multi-D data.
  - Sent only when receiver ready for it.
  - Serialisation of `NDArray` avoids any copies.
- Offer flexible configuration
  - Get *copy* of all data or *share* with others
  - *Drop* or *queue* if receiver slow
  - GUI server throttles to 2 Hz
  - …



Karabo: Three APIs

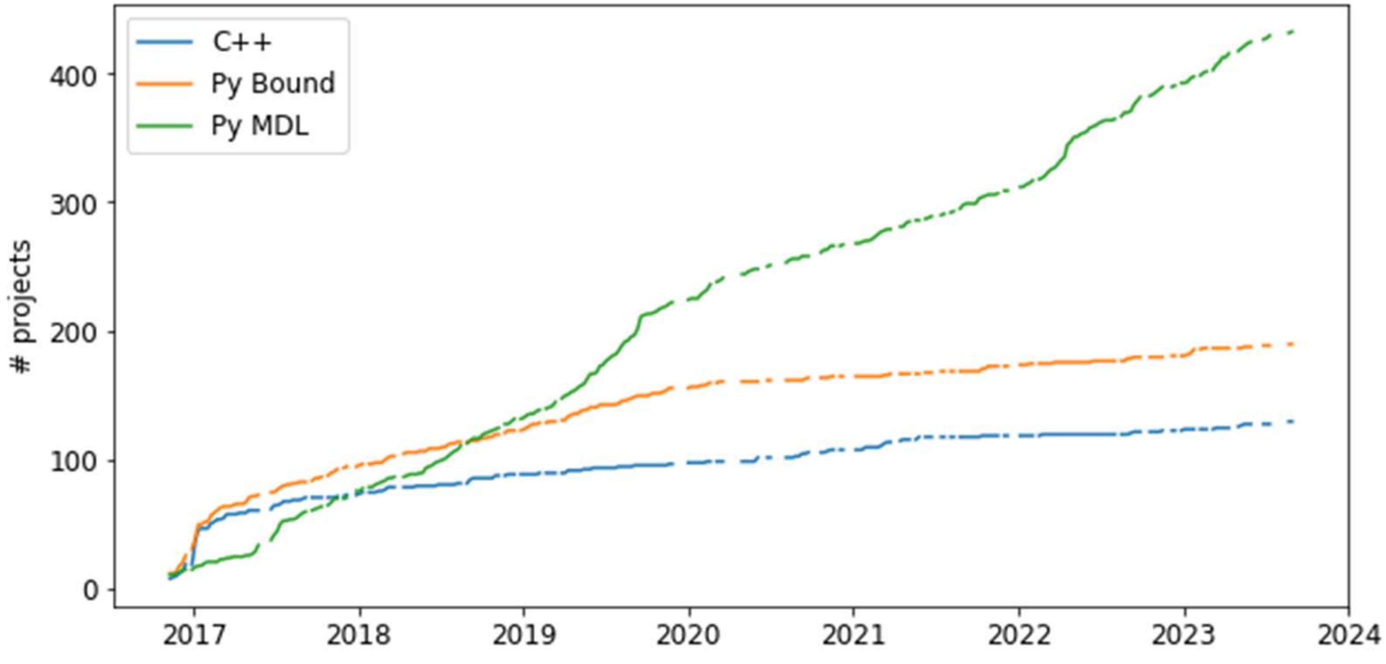
- **C++** (C++17 standard)
  - The start of Karabo, based on a lot of the `boost` libraries
  - (Still) most service devices (data logging, GUI server)
  - Devices that require high performance (digitizers)

- **Bound** Python
  - Python bindings on top of C++ (now using `pybind11`, few things pure Python)
  - Partially not „pythonic“, but more following underlying C++ patterns
  - Pipelining more performant than the one of Middlelayer

- **Middlelayer** Python
  - Complete re-write, based on `asyncio`
  - Especially designed to interact with other devices (therefore “middlelayer”)
    ► Nowadays most popular API, not only for middlelayer devices
  - Used as macro language (without need for asyncio‘s `await`)

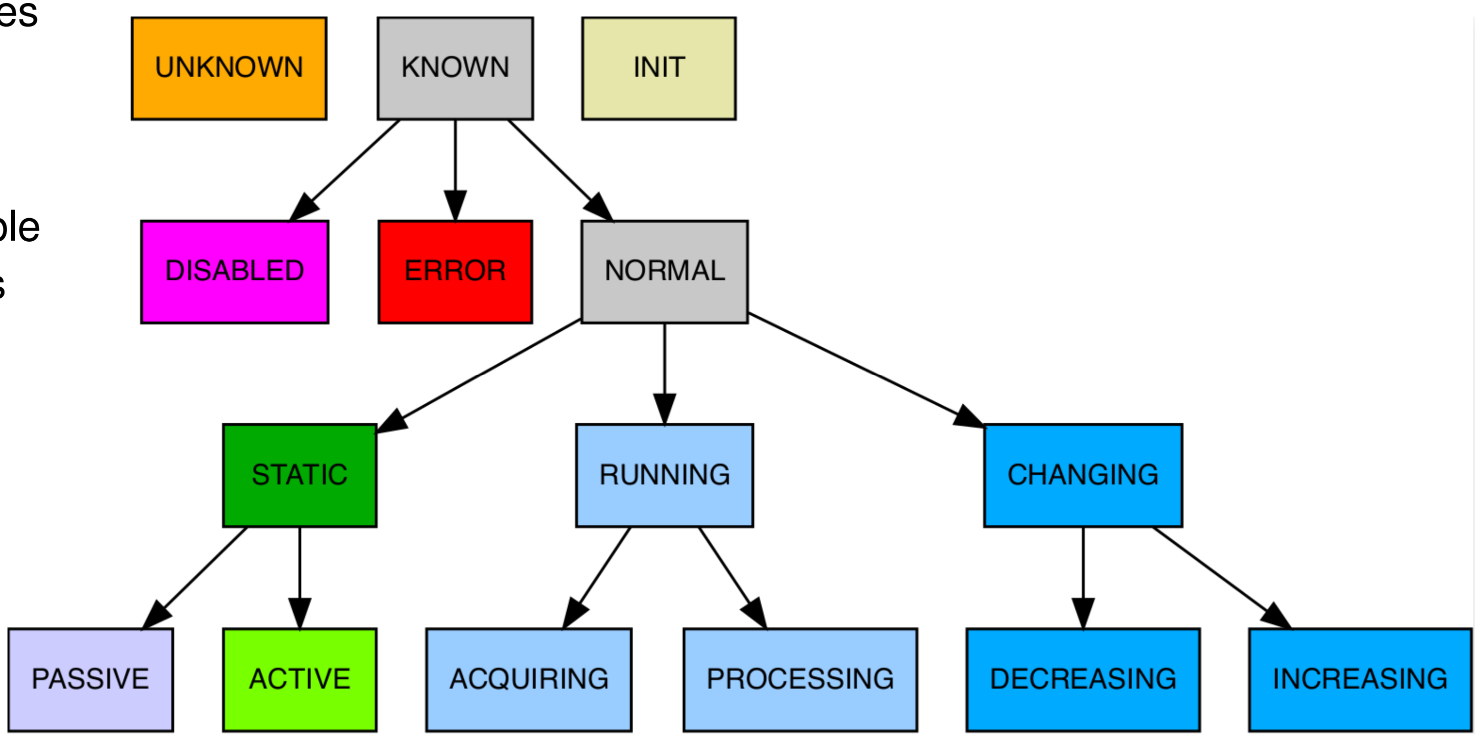
Usage of the Three APIs

- Middlelayer API
  - often has the most expressive syntax
  - shortest "time-to-market".
- C++ and Python Bound
  - actively maintained
  - new devices are still being implemented
    - ► especially in high-performance fields.

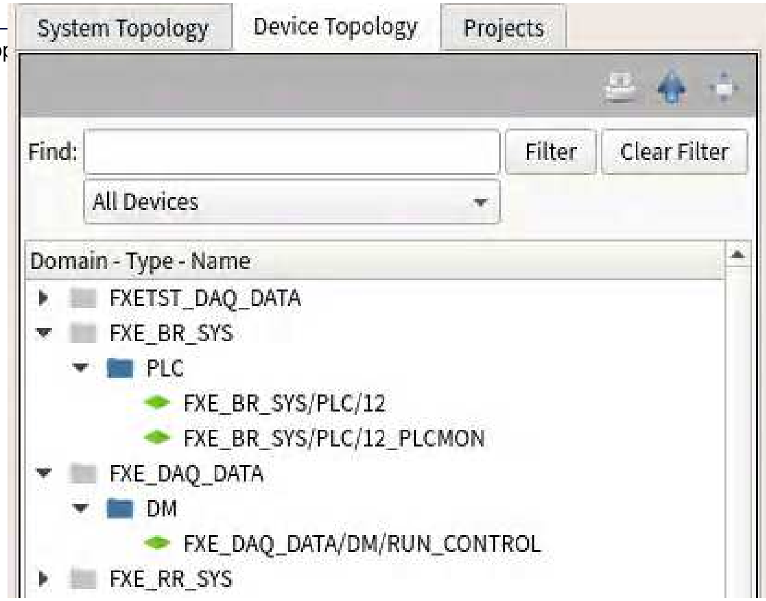


Unified Device States

- Predefined list of device states
- Device schema
  - can restrict access to its commands / reconfigurable properties to some states
- Inheritance system
  - E.g. ERROR is more concrete than KNOWN
- State significance order for state aggregation
- Unified colour representation in the GUI



About 60 more states inheriting (e.g. GUI colour) from those of the last row.

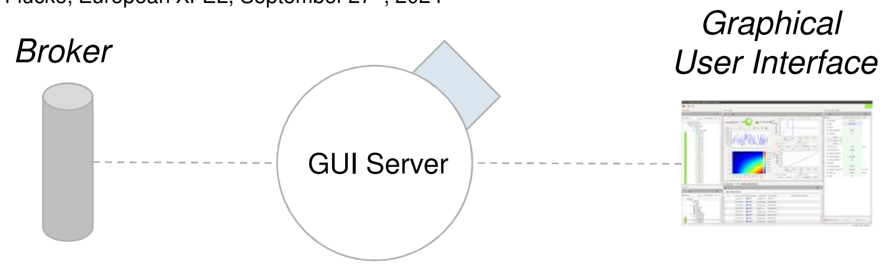


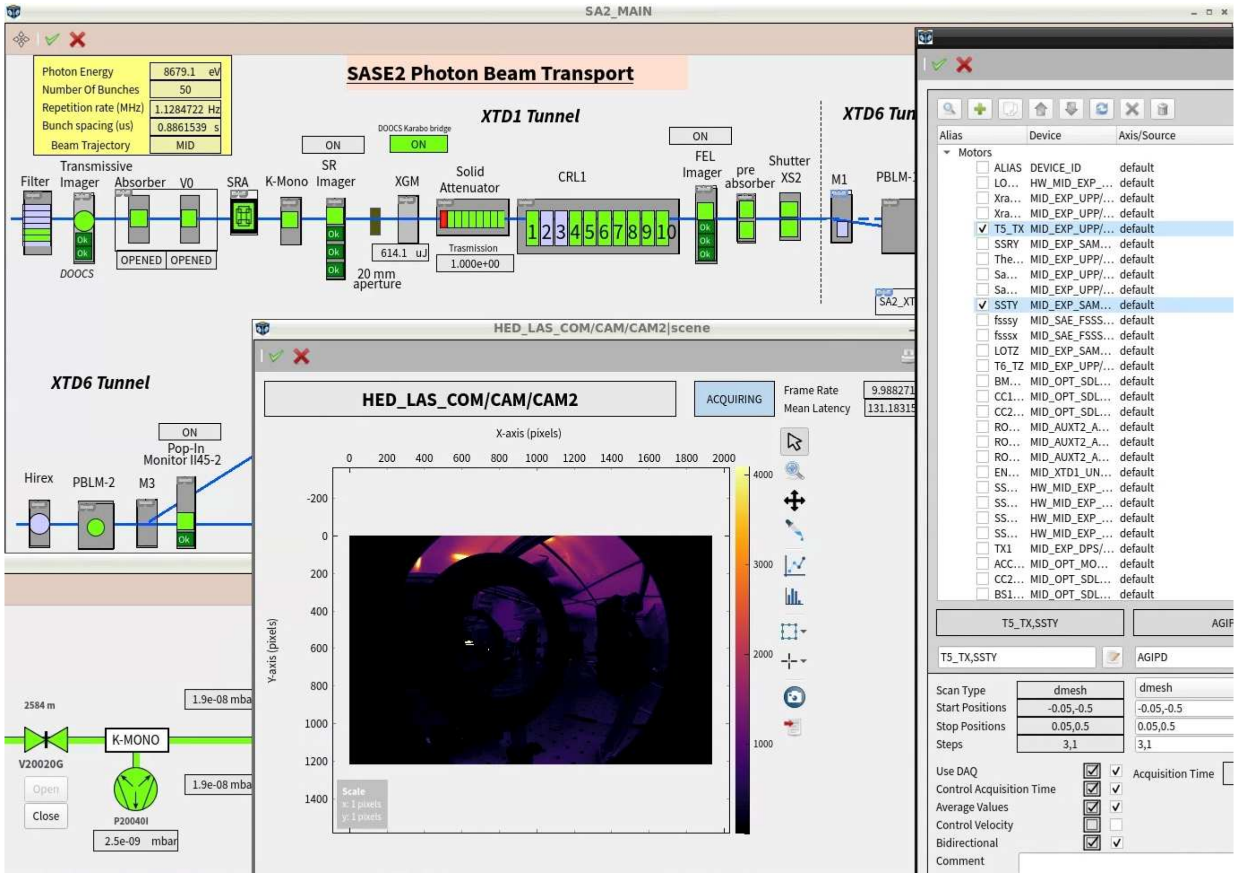
More Device Concepts

- European XFEL naming convention, e.g.
  - Not enforced by Karabo (but GUI's "Device Topology" ignores devices that don't have 2 slashes)
- Device locking
  - A device can lock other devices to reject commands and reconfigurations from others
  - Soft lock on purpose: to avoid operational deadlock, `slotClearLock` can be called by everybody
- Capabilities exposed via "instanceInfo"
  - provides_scenes, provides_macros, provides_interfaces
- Interfaces
  - motor, multi axis motor, trigger, camera, processor, device instantiator
  - An interface promises some commands and proeprties
  - Where is documented what exactly which interface requires?

Generic, Extendible Karabo GUI:

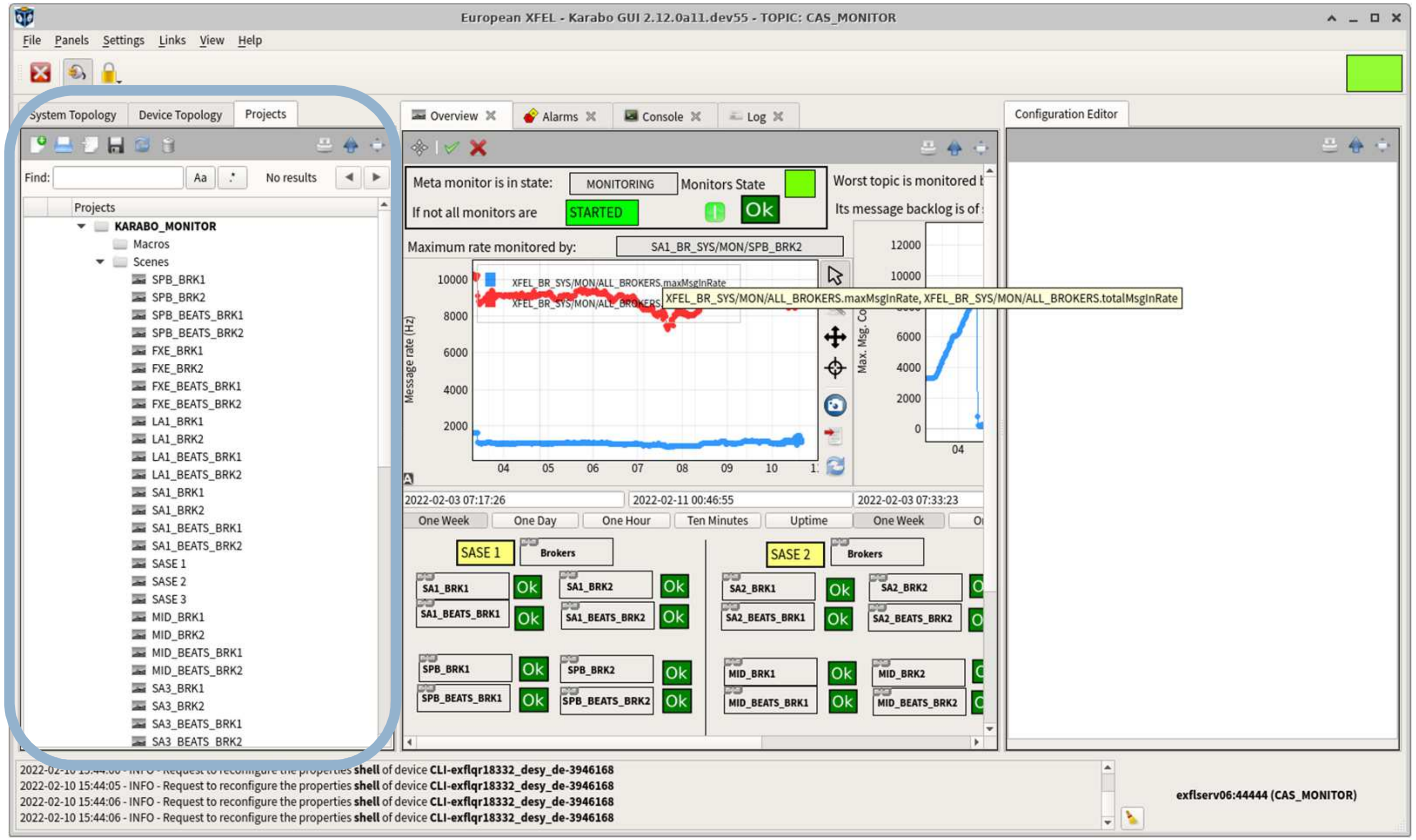
- Separate Python Package
  - Shares Hash with MDL
  - Well matched to the framework
  - PyQt5-based
- Connects to Karabo via the GUI-server (tcp, point-to-point)
- Extendible via "gui-extensions"
- Distinguishing features:
  - GUI scene builder (drag'n'drop)
  - Projects to logically group devices, scenes and macros





GUI as Project Interface

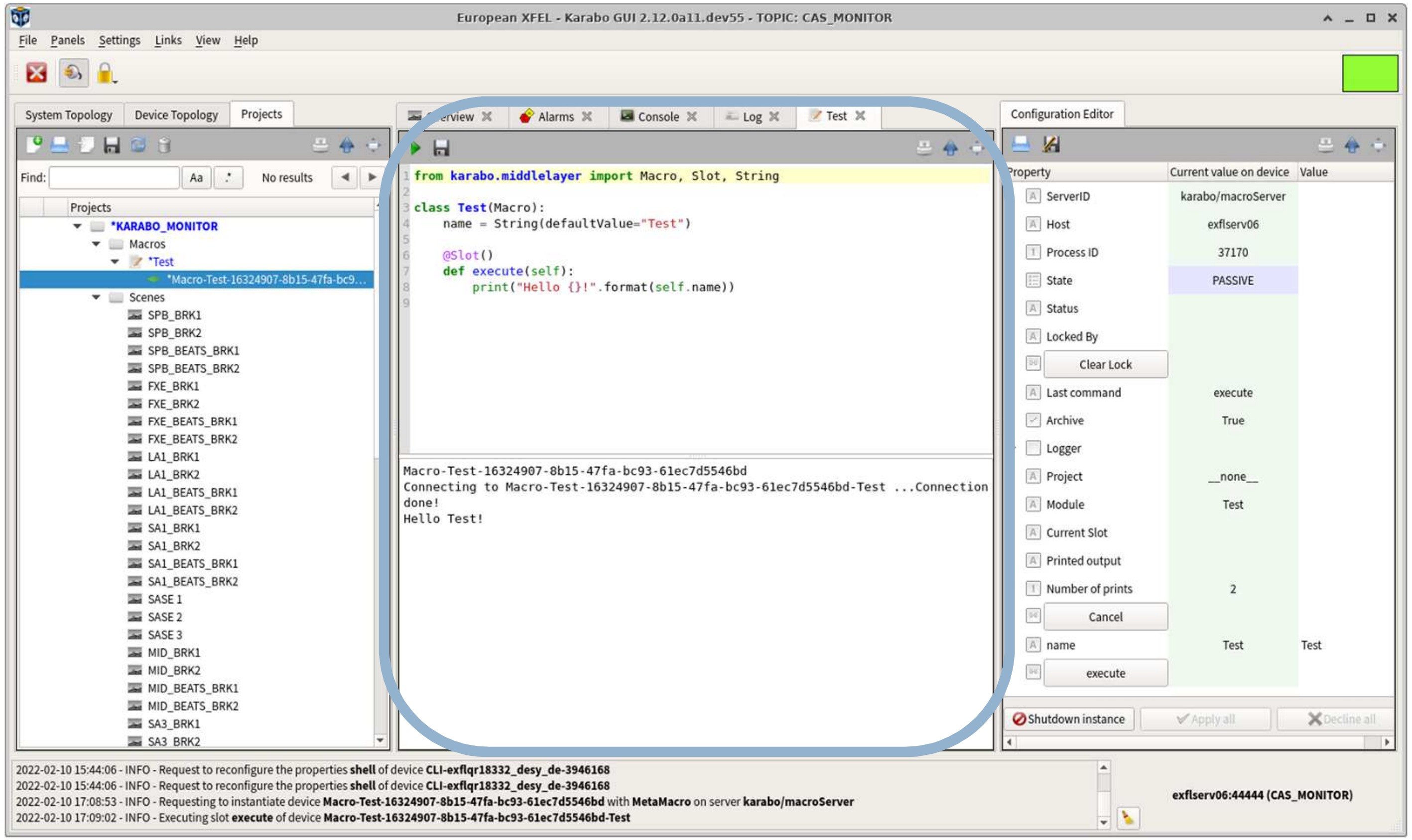
- Project data:
  - device configurations,
  - scenes,
  - macros,
  - sub-projects.
- Projects stored in central data base as XML files.
  - local storage option



GUI as Macro Interface

Macros

- Aim for (simple) procedures
  - By scientists
  - Middlelayer syntax
- Run on special macro server
  - Gives control if macro runs havoc
- Code injected via GUI
  - Stored in projects
  - Output in GUI
- More complex and matured macros often converted to Middlelayer devices



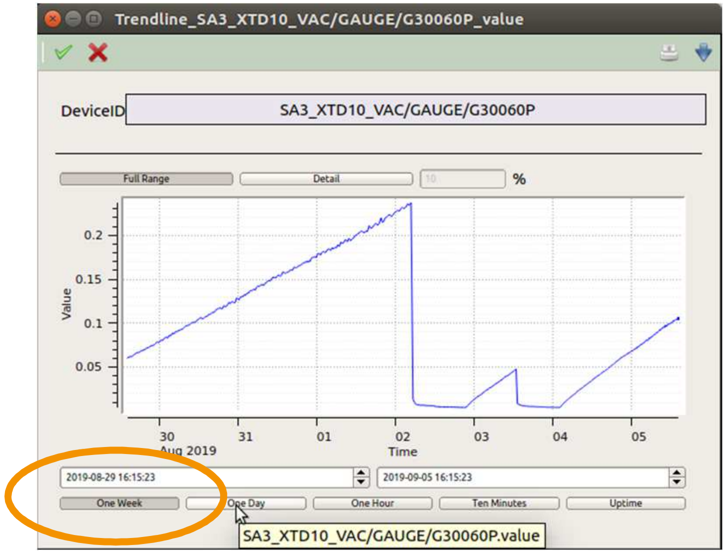
System Service: Service Manager

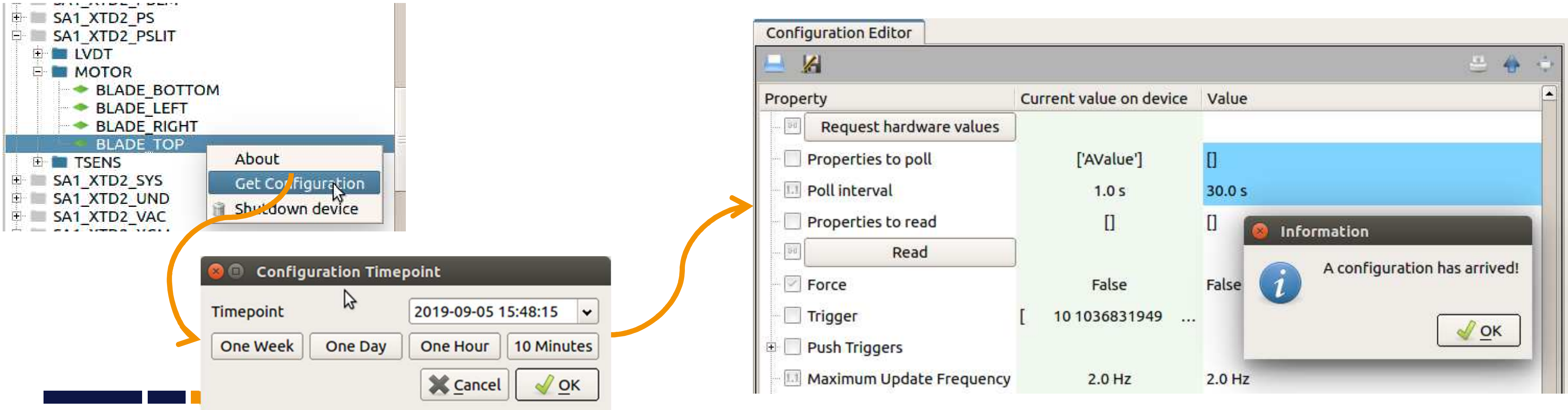
- Special device exposes running services (servers) of an installation
  - Start, restart, stop (kills if needed)
- Needs special service running on each host with a Karabo installation
  - Communicates via web protocol with service manager

Framework Service: Data Logging

- In-built data logging and retrieval mechanism.
  - Control data only, no pipelines.
  - Implemented via data logger service devices (text file or InfluxDB time series database).

- Main control use cases:
  - ▶ Past data for trendlines: single scalar property vs time.
  - ▶ Past configurations: all device properties at point in time.





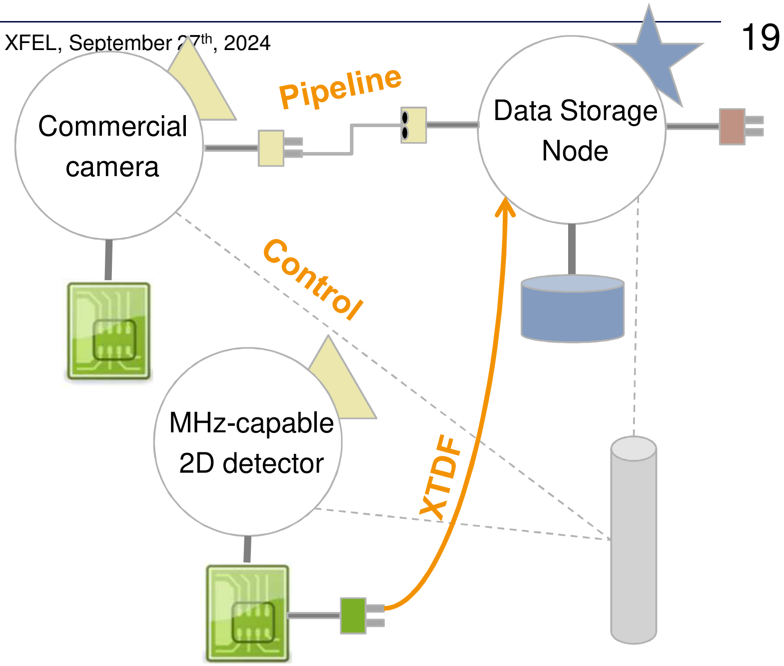
Karabo Data Acquisition (DAQ) Integration

- Focus on scientific instrument data with long term storage: EuXFEL data policy

- Support for different types of data sources:
  - **Control** data with train resolution: e.g. sensors, motors → **slow data**
  - 2D or pulse resolved data: e.g. **pipeline** from cameras, digitizers → **fast and/or medium sized data**
  - MHz-capable 2D detectors (XFEL train data format - **XTDF**) → **big & fast data**

- Data stored in HDF5 files, indexed per train
  - 9 PB raw data (Oct. '19) stored since experiments started
  - 12 GB/s achieved (600 images per train)

- Provide data stream for online display and analysis:
  - Calibration of big 2D detectors (1.8 GB/s, 2s latency),
  - External tool via Karabo-to-ZeroMQ bridge



Raw Data Generated at EuXFEL