Online and Offline Data Analysis at European XFEL

Thomas Michelat
Control and Analysis Software Group

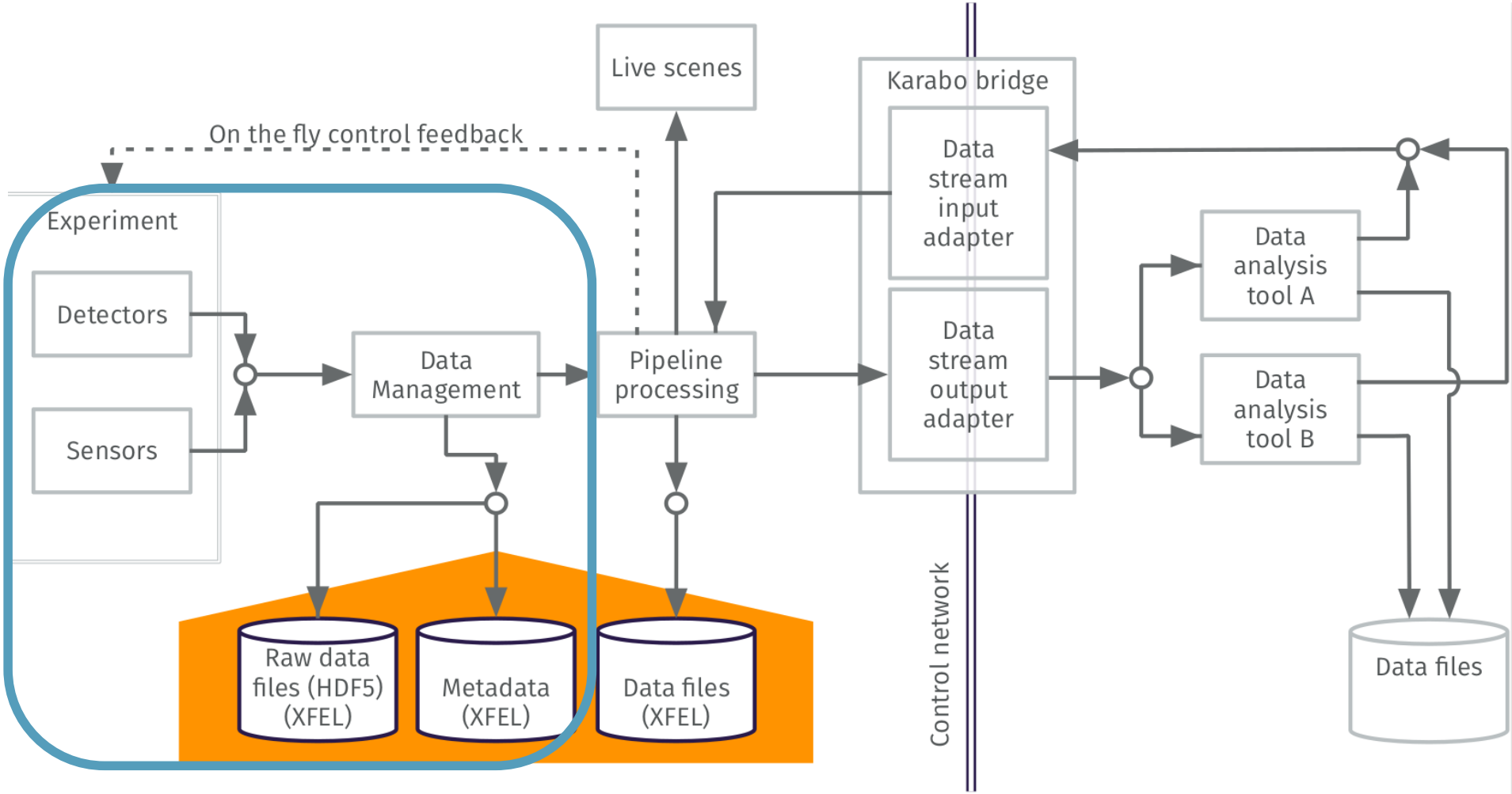
Hamburg, 23 January 2018

Outline

- Online data analysis
- Offline data analysis
- Outlook

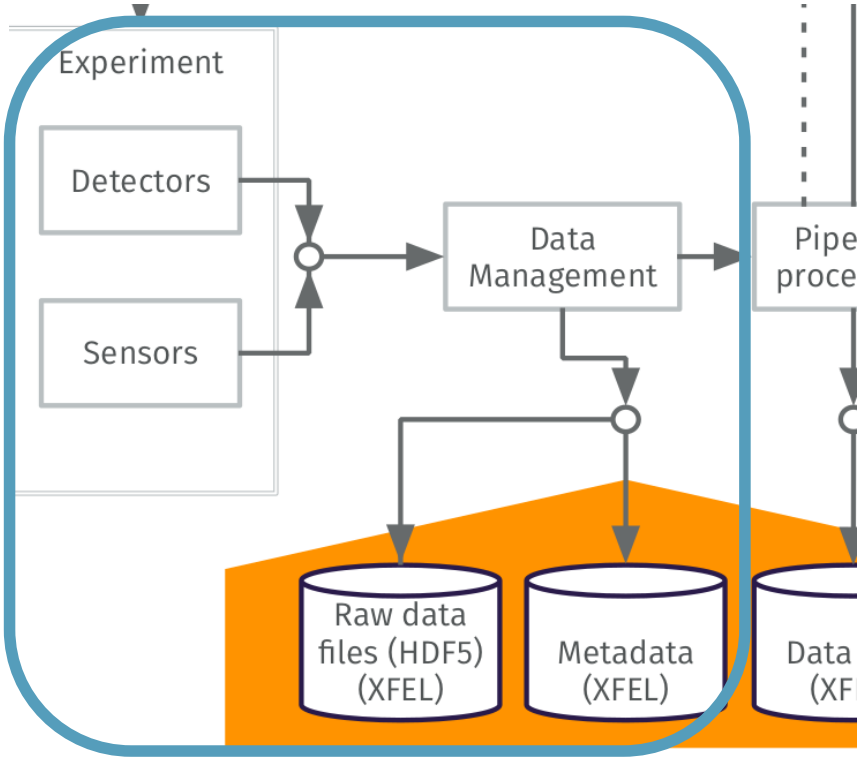
Online Analysis

Data Acquisition



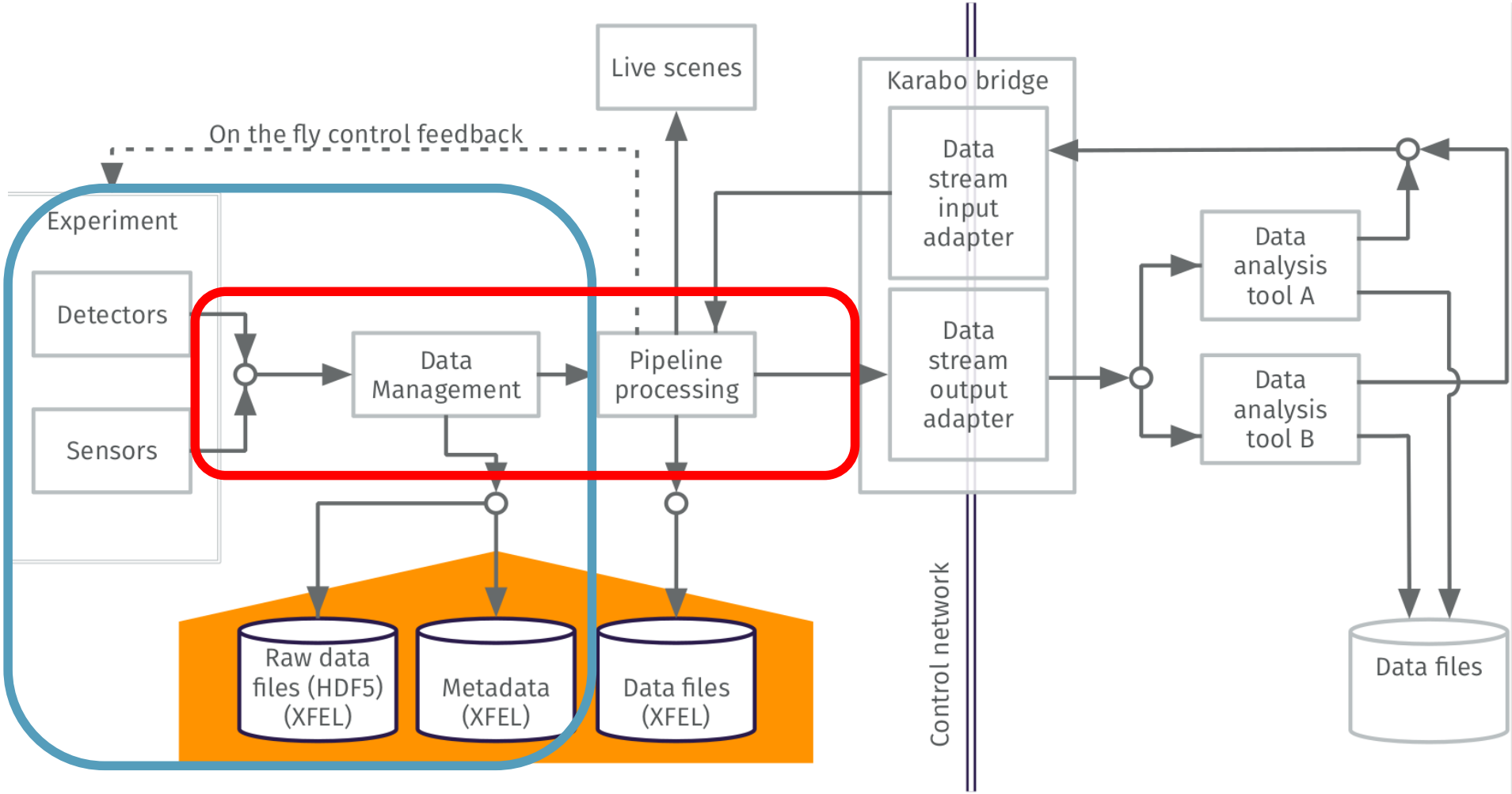
Data Acquisition

- Various data sources
  - Detectors
  - Cameras
  - Sensors
  - Actuators
  - Computing
  - …



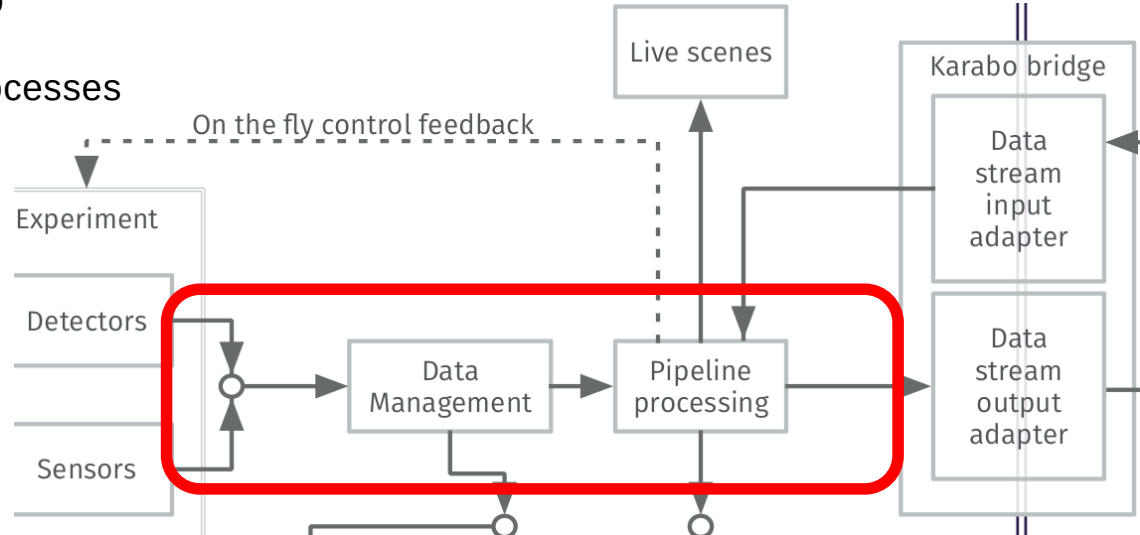
- Interesting sources are gathered in the DAQ system
  - Synchronized by train ID
  - Stored to file (HDF5)
  - Streamed over TCP

Karabo Data Pipeline

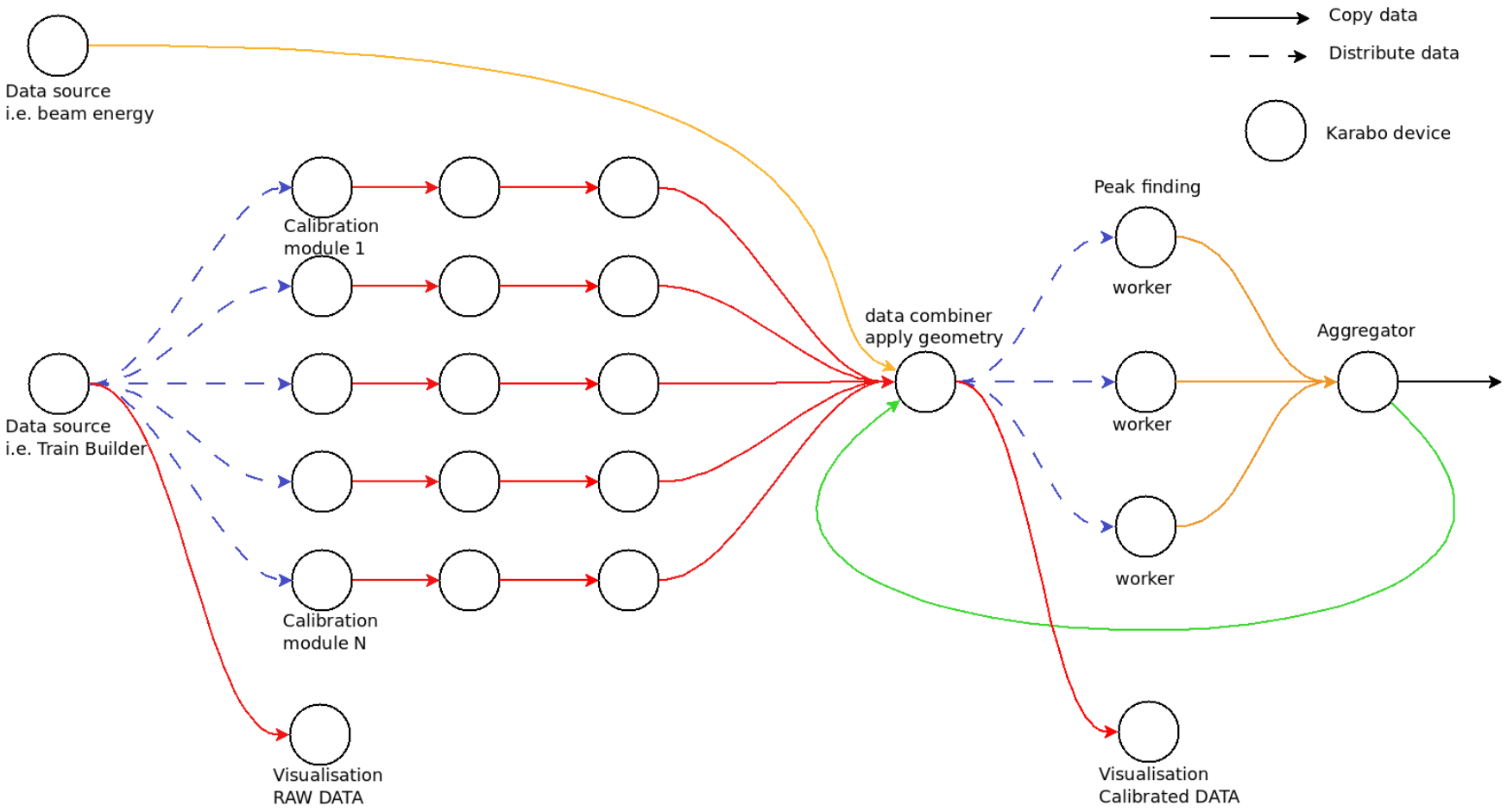


Karabo Data Pipeline

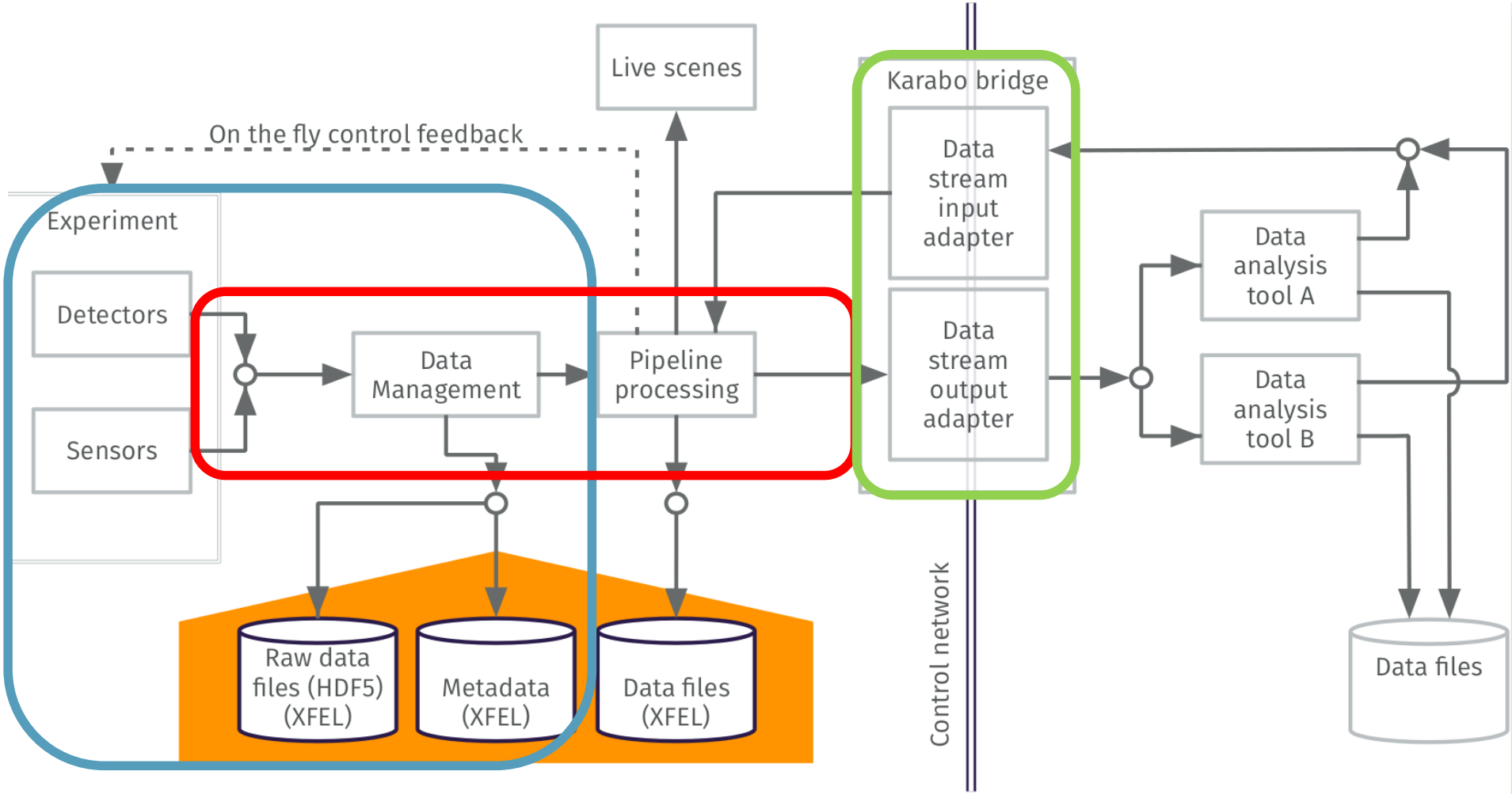
- Peer-to-peer model with **TCP** protocol
- Direct **data channels** between Karabo devices (applets)
- Feature complete
  - Dispatches data 1-to-n / n-to-1
  - Copy data to n clients
  - Policy on busy client: wait, queue, drop, exception
- Standardized format and **data container** (Karabo Hash)
- Provides data source and timestamp
- Standardized **interface** between processes
  - Focus on the device logic
- Easily scalable



Karabo Data Pipeline - example

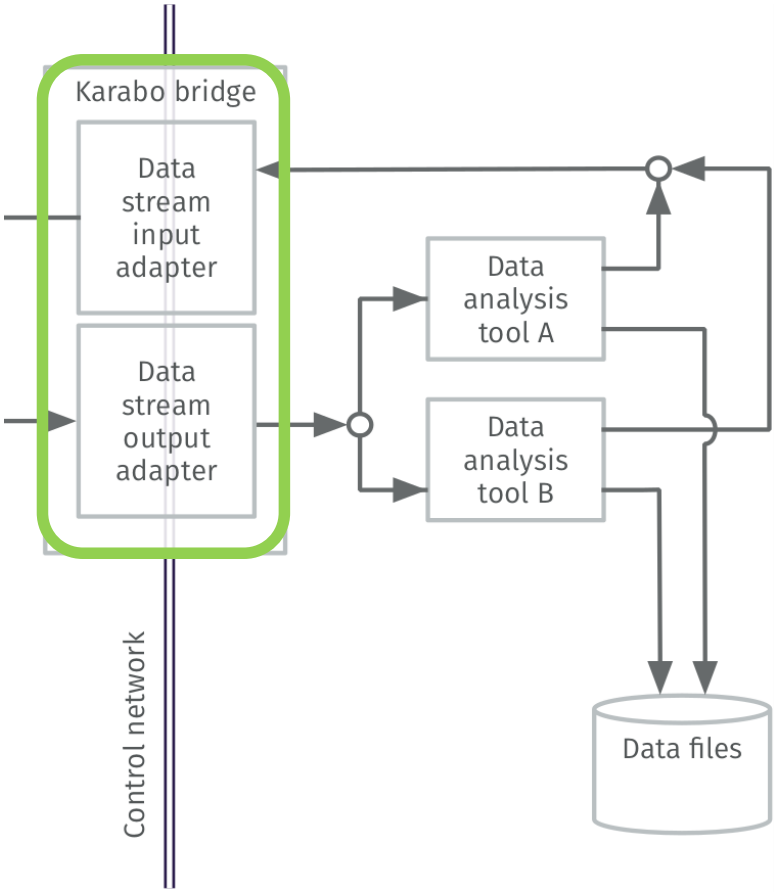


Karabo Bridge



Export Data Pipeline – Karabo Bridge

- We provide an interface to listen to Karabo pipelines
  - **Integrate** existing (complex) user provided tools
  - Quick (dirty) **specific** scripts to use during an experiment
- Karabo Bridge requirements
  - Loosely coupled **Interface** between Karabo and external programs
  - Export data in a **generic** container
  - Using **straightforward** network interface
  - Low latency
- Development in collaboration with CFEL Chapman Group (S. Aplin, A. Barty, M. Kuhn, V. Mariani)



DEMO

Karabo Bridge Client

- Install the client

```
pip install -e git+https://github.com/European-XFEL/karabo-bridge-py.git#egg=karabo-bridge-py
```

- How to use it

Import Karabo bridge client

```
In [1]: from karabo_bridge import KaraboBridge
```

How to use it?

```
In [2]: help(KaraboBridge)
```

```
Help on class KaraboBridge in module karabo_bridge.KaraboBridge:

class KaraboBridge(builtins.object)
 |  Karabo bridge client for Karabo pipeline data.
 |
 |  This class can request data to a Karabo bridge server.
 |  Create the client with::
 |
 |      krb_client = KaraboBridge("tcp://153.0.55.21:12345")
 |
 |  then call ``data = krb_client.next()`` to request next available data
 |  container.
 |
 |  Parameters
 |  ----------
 |  endpoint : str
 |      server socket you want to connect to (only support TCP socket).
 |  sock : str, optional
 |      socket type - supported: REQ.
 |  ser : str, optional
 |      Serialization protocol to use to decode the incoming message (default
 |      is msgpack) - supported: msgpack,pickle.
 |
```

Karabo Bridge Client

- Connection to a server
- Request data
- Data is contained in a dictionary
- One entry per data source in the train

At object instantiation, the client connects to the karabo bridge server.

```
In [3]: kb = KaraboBridge('tcp://max-exfl093:45632')
```

request the next data available on this server.

```
In [4]: train = kb.next()
```

The data container is a dictionary.

```
In [5]: type(train)
Out[5]: dict
```

It contains all data sources in this data pipeline for an XRAY train

```
In [6]: train.keys()
Out[6]: dict_keys(['detector', 'DETLAB_LAB_LPD-1/FPGA/FEM_Q2M0', 'DETLAB_LAB_LP
D-1/FPGA/FEM_Q1M0', 'DETLAB_LAB_LPD-1/FPGA/FEM_Q0M0', 'DETLAB_LAB_LPD-
1/FPGA/FEM_Q3M0'])
```

Karabo Bridge Client

- Each data source is a dictionary
  - It contains device parameters
  - And source metadata
- All data are python built-in types
- Big array are Numpy array
- Requesting data will return the latest available train in the pipeline

find the parameters keys for a specific data source.

```
In [7]: train['detector'].keys()
```

```
Out[7]: dict_keys(['image.data', 'image.trainId', 'ignored_keys', 'metadata',
'trainId', 'image.pulseId', 'image.cellId'])
```

All sources are associated with medata, containing: source name, train ID and UNIX epoch.

```
In [8]: train['detector']['metadata']
```

```
Out[8]: {'source': 'DETLAB_LAB_DAQ-0/DET/0:xtdf',
 'timestamp': {'frac': 0, 'sec': 1516198931, 'tid': 1516199957}}
```

Example, getting detector image (numpy array) and display array informations.

```
In [9]: im = train['detector']['image.data']
print('shape:', im.shape)
print('dtype:', im.dtype)
print(im[0, 0, 0:2, 0:2])
```

```
shape: (5, 4, 256, 256)
dtype: float64
[[3910.05812037 4041.62319897]
 [4113.51273402 4056.11034393]]
```

While data is flowing through the karabo pipeline, you can request data.

```
In [18]: for i in range(5):
    data = kb.next()
    print(data['detector']['trainId'])
```

```
1516200478
1516200481
1516200483
1516200483
1516200484
```

Karabo Bridge Client

- You can instantiate many clients
  - Data can be dispatched among them

You can create as many clients as you need (data will be distributed over the different clients).

In [52]:
```
client_2 = KaraboBridge('tcp://max-exfl093:45632')
data = client_2.next()
print(data['detector']['trainId'])
```

```
1516380752
```

In [53]:
```
client_3 = KaraboBridge('tcp://max-exfl093:45632')
data = client_3.next()
print(data['detector']['trainId'])
```

```
1516380753
```

  - Or copy to all
    - ► PUB-SUB sockets

Karabo Bridge Client – Try this at home!

- Karabo Bridge server simulation
  - Does not require Karabo
  - Helps you test integration of the client to your tool

```
Terminal
tmichela@exflpcx17673 ~/projects/karabo-bridge-py/examples
 % ./demo.sh
demo.sh: starting (simulated) server
demo.sh: starting client
Client : received train ID 15163874924
Client : - detector image shape is (32, 16, 512, 128), 64.0 Mbytes
Client : received train ID 15163874931
Client : - detector image shape is (32, 16, 512, 128), 64.0 Mbytes
Server : buffered train: 15163875269
Client : received train ID 15163874939
Client : - detector image shape is (32, 16, 512, 128), 64.0 Mbytes
Client : received train ID 15163874945
Client : - detector image shape is (32, 16, 512, 128), 64.0 Mbytes
Server : buffered train: 15163875274
Client : received train ID 15163874950
Client : - detector image shape is (32, 16, 512, 128), 64.0 Mbytes
Client : received train ID 15163874955
Client : - detector image shape is (32, 16, 512, 128), 64.0 Mbytes
Server : buffered train: 15163875280
Client : received train ID 15163874960
Client : - detector image shape is (32, 16, 512, 128), 64.0 Mbytes
Client : received train ID 15163874965
Client : - detector image shape is (32, 16, 512, 128), 64.0 Mbytes
Client : received train ID 15163874970
```

```python
# server.py

from karabo_bridge import server_sim

# start a simulated karabo bridge server
# and bind a socket on port 4545 of this machine (localhost).
server_sim(4545)
```

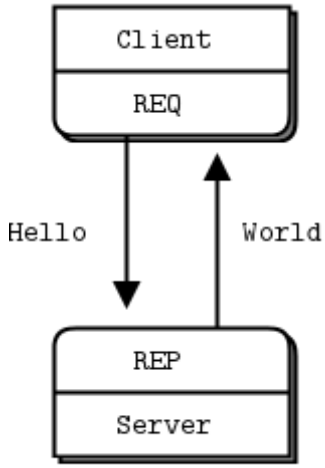
```python
# client.py

from karabo_bridge import KaraboBridge

# connect the client to localhost if running on the same machine as the server.
client = KaraboBridge('tcp://localhost:4545')

while True:
    data = client.next()
    det_data = data['SPB_DET_AGIPD1M-1/DET/detector']
    print("Client : received train ID", str(det_data['header.trainId']))
    print("Client : - detector image shape is {}, {} Mbytes".format(
          det_data['image.data'].shape, det_data['image.data'].nbytes/1024**2))
```

Karabo Bridge – technical details
Networking library



- ZeroMQ
  - Intelligent **socket library** for messaging
  - Many kind of connection **patterns**
  - Multiplatform, **multi-language** (30+)
  - **Fast** (8M msg/sec, 30 usec latency)
  - Small (20k lines of C++ code)
  - Open source LGPL with a **large community**

  - Message blobs of 0 to N bytes
  - One socket to many socket connection
  - Queuing at sender and receiver
  - Automatic TCP (re)connect
  - Zero-copy for large message

  - **Easy** to use

ØMQ Hello World

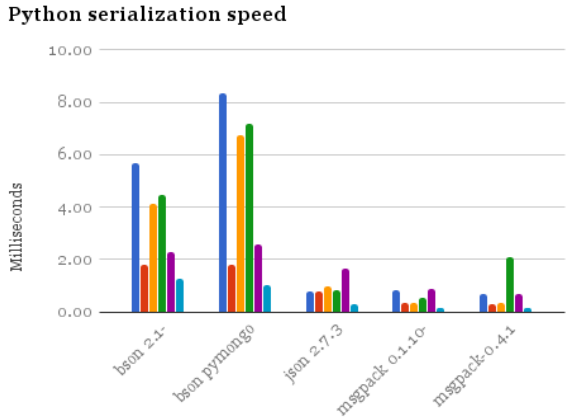
```
import org.zeromq.ZMQ;
public class hwclient {
    public static void main (String[] args){
        ZMQ.Context context = ZMQ.context (1);
        ZMQ.Socket socket = context.socket (ZMQ.REQ);
        socket.connect ("tcp://localhost:5555");
        socket.send ("Hello", 0);
        System.out.println (socket.recv(0));
    }
}
```

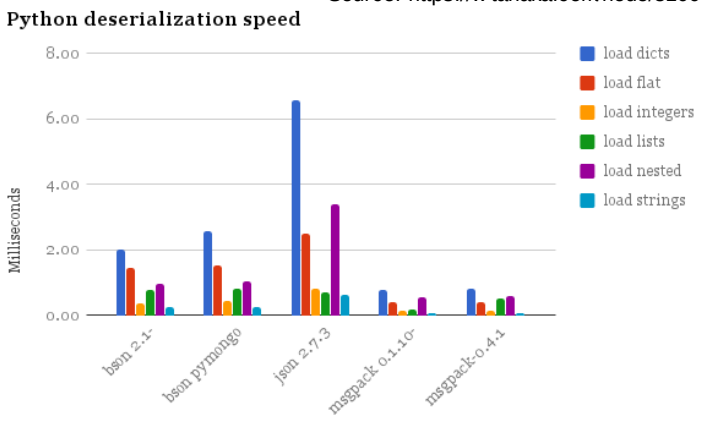
```
import org.zeromq.ZMQ;
public class hwserver {
    public static void main (String[] args) {
        ZMQ.Context context = ZMQ.context(1);
        ZMQ.Socket socket =
   context.socket(ZMQ.REP);
        socket.bind ("tcp://*:5555");
        while (true) {
            byte [] request = socket.recv (0);
            socket.send("World", 0);
        }
    }
}
```

Karabo Bridge – technical details
Message serialization

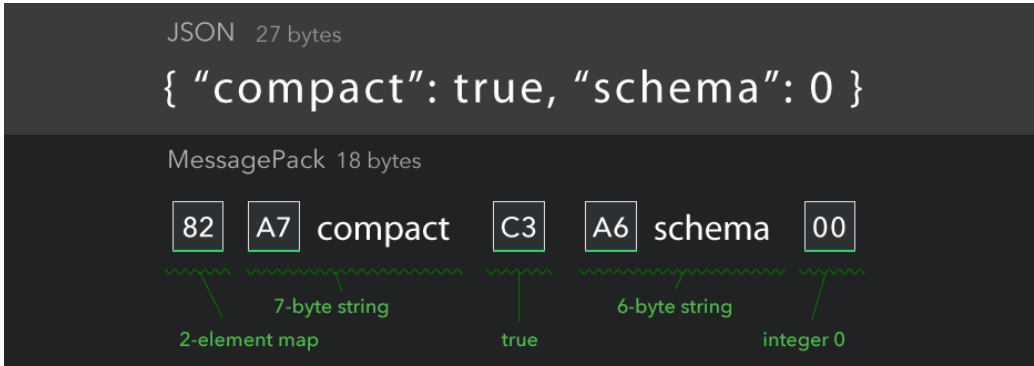
- Serialization
  - Pickle
  - boost::serialization
  - **MessagePack**
  - Protobuf
  - …

Source: https://w tanaka.com/node/8100

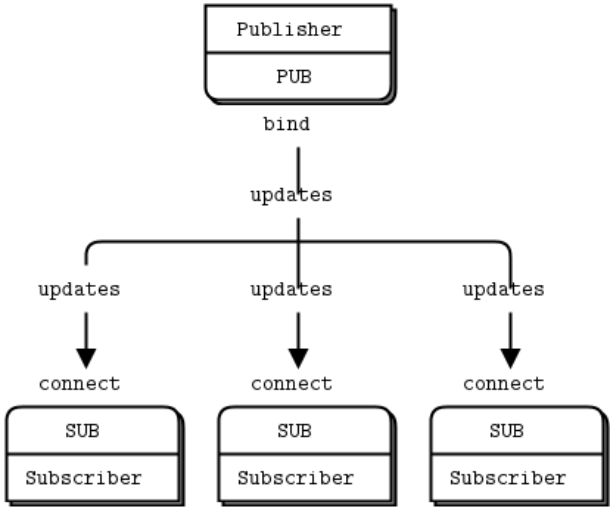




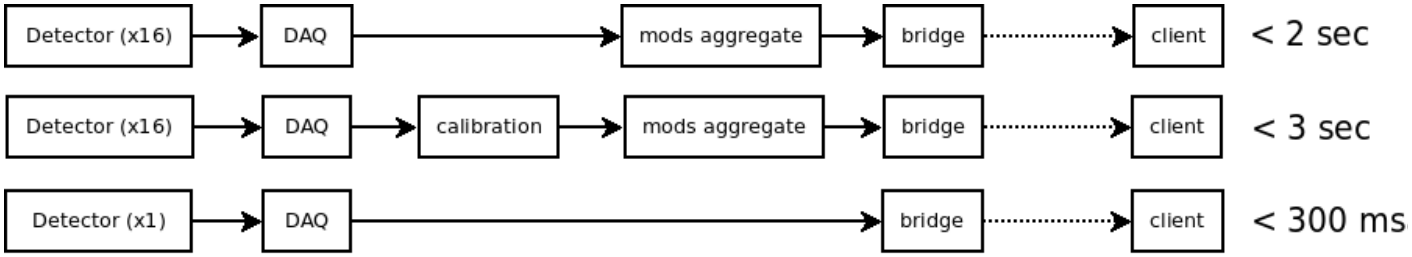
- MessagePack
  - Simple and open source design: https://github.com/msgpack/msgpack/
  - **JSON-like** binary format
  - But **faster** and **smaller**
  - **Multi-language** (80+ implementation available)
  - Easy implementation if need to support new language



Karabo Bridge - Client side



- Connect to Karabo bridge server(s)
  - Broadcaster – Subscriber (1-to-n):
    - ► Data is **copied** to many clients
  - Server – client (n-to-n): data is sent on request
    - ► Requests are distributed among servers
    - ► Data is **dispatched** between clients
- Data is provided in a generic container (msgpack)
  - Python client: dictionary
  - C++ client: std::map
- Performances
  - SPB instrument – AGIPD detector



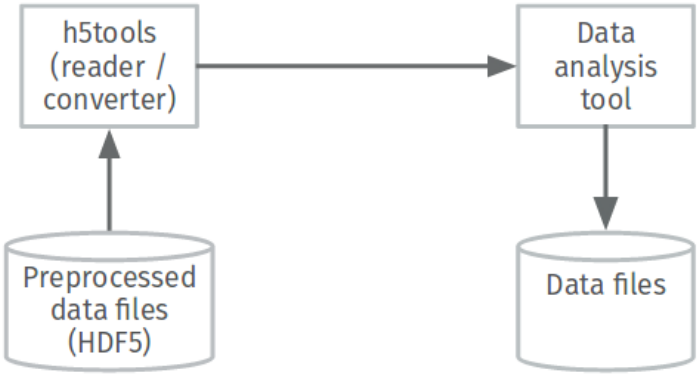
Summary Karabo Bridge

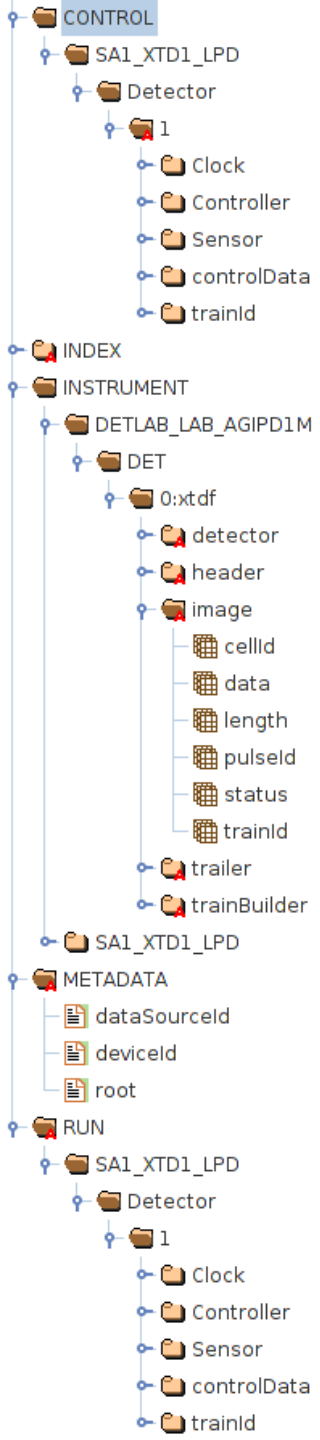
- Network interface to access scientific data during experiment in near real time
- Support all existing data type in Karabo
- Keep the same data structure and names
- Easy set-up to export any data pipeline from Karabo
- Client implementation
  - Python: https://github.com/European-XFEL/karabo-bridge-py
  - C++: implementation existing (under preparation)
  - ...
- Successful use during first experiments
  - OnDA
  - Hummingbird
  - CASS
  - …
- Any request?

Offline Analysis

Data Files

- For each experiment **proposal**
  - Data are stored in **run** folders
  - **Runs** contain a collection of **HDF5** files
  - HDF5 files are structured in EuXFEL specific format
- Euxfel-h5tools
  - Command line mode: quick overview
  - Library to read run data more conveniently





DEMO

European XFEL h5tools

- How to use

This example present basic functionalities of the python package euxfel_h5tools provided by the European XFEL. We will parse a run directory, extract related informations, read train related data, combine data from different data sources.

In [1]:
```
# The European XFEL specific HDF5 tools
from euxfel_h5tools import RunHandler, stack_detector_data
```

In [2]:
```
# Path to the data run we want to analyse
run_dir = '/gpfs/exfel/data/exp/XMPL/r0803/'
```

The run directory contains many HDF5 files. In this case each of them contains a single data source (AGIPD detector modules), but it can contains many sources and thus be difficult to know where to find a particular parameter.

In [3]:
```
!ls $run_dir | grep .h5
```

```
CORR-R0803-AGIPD00-S00000.h5
CORR-R0803-AGIPD01-S00000.h5
CORR-R0803-AGIPD02-S00000.h5
CORR-R0803-AGIPD03-S00000.h5
CORR-R0803-AGIPD04-S00000.h5
CORR-R0803-AGIPD05-S00000.h5
CORR-R0803-AGIPD06-S00000.h5
CORR-R0803-AGIPD07-S00000.h5
```

By instantiating a RunHandler class, the run directory is parsed and contained data is sorted per train.

In [5]:
```
# Instanciate the run handler with the path to the run folder.
run1 = RunHandler(run_dir)
```

European XFEL h5tools

- Prompt run information

You can find basic information about the run with the method `infos()`. `instrument` devices are devices which are pulses related (have more that one parameter value per train), `control` devices are train related or slower.

In [6]:
```
# Display general information about this run.
run1.infos()
```

```
Run information
        Duration:       0:02:48.400000
        First train ID: 1541484692
        Last train ID:  1541486376
        # of trains:    251

Devices
        Instruments
        - SPB_DET_AGIPD1M-1/DET/0CH0:xtdf
        - SPB_DET_AGIPD1M-1/DET/10CH0:xtdf
        - SPB_DET_AGIPD1M-1/DET/11CH0:xtdf
        - SPB_DET_AGIPD1M-1/DET/12CH0:xtdf
        - SPB_DET_AGIPD1M-1/DET/13CH0:xtdf
        - SPB_DET_AGIPD1M-1/DET/14CH0:xtdf
        - SPB_DET_AGIPD1M-1/DET/15CH0:xtdf
        - SPB_DET_AGIPD1M-1/DET/1CH0:xtdf
        - SPB_DET_AGIPD1M-1/DET/2CH0:xtdf
        - SPB_DET_AGIPD1M-1/DET/3CH0:xtdf
        - SPB_DET_AGIPD1M-1/DET/4CH0:xtdf
        - SPB_DET_AGIPD1M-1/DET/5CH0:xtdf
        - SPB_DET_AGIPD1M-1/DET/6CH0:xtdf
        - SPB_DET_AGIPD1M-1/DET/7CH0:xtdf
        - SPB_DET_AGIPD1M-1/DET/8CH0:xtdf
        - SPB_DET_AGIPD1M-1/DET/9CH0:xtdf
        Controls
        -
```

European XFEL h5tools

- Start train iterator
- Get data per train
- Find the data sources
- Extract interesting parameter

The RunHandler class contains a generator method that can Iterate over trains. The returned object is a tuple with 2 values: (1) the train ID of the returned train and (2) the data it contains.

In [8]:
```
trains = run1.trains()

# get the first train in the run by calling next().
first_train = next(trains)
print('* The train generator returns a ', type(first_train))

train_id, data = first_train

# train_id is an int (unique identifier for each XRAY train)
print('* The returned train is:', train_id)
# data is a dictionary, each item is a data source.
print('* data sources in the first train:\n', data.keys())
```

```
* The train generator returns a  <class 'tuple'>
* The returned train is: 1541484692
* data sources in the first train:
 dict_keys(['SPB_DET_AGIPD1M-1/DET/13CH0:xtdf', 'SPB_DET_AGIPD1M-1/DET/1CH0
:xtdf', 'SPB_DET_AGIPD1M-1/DET/2CH0:xtdf', 'SPB_DET_AGIPD1M-1/DET/5CH0:xtdf
', 'SPB_DET_AGIPD1M-1/DET/12CH0:xtdf', 'SPB_DET_AGIPD1M-1/DET/9CH0:xtdf', '
SPB_DET_AGIPD1M-1/DET/4CH0:xtdf', 'SPB_DET_AGIPD1M-1/DET/15CH0:xtdf', 'SPB_
DET_AGIPD1M-1/DET/0CH0:xtdf', 'SPB_DET_AGIPD1M-1/DET/6CH0:xtdf', 'SPB_DET_A
GIPD1M-1/DET/8CH0:xtdf', 'SPB_DET_AGIPD1M-1/DET/7CH0:xtdf', 'SPB_DET_AGIPD1
M-1/DET/11CH0:xtdf', 'SPB_DET_AGIPD1M-1/DET/10CH0:xtdf', 'SPB_DET_AGIPD1M-1
/DET/3CH0:xtdf', 'SPB_DET_AGIPD1M-1/DET/14CH0:xtdf'])
```

We want to get the parameter 'image.data' from the source 'SPB_DET_AGIPD1M-1/DET/0CH0:xtdf'. The data source represente the output data of a device in a karabo data pipeline. in this case the source is the output of the module 0 of the AGIPD detector from the instrument SPB. This parameter contains the pixels values for each pulses in the train.

In [9]:
```
image_mod0 = data['SPB_DET_AGIPD1M-1/DET/0CH0:xtdf']['image.data']

# image.data is a numpy.array
# 1st dimention: pulse index
# 2nd and 3rd: x, y
print('data shape:', image_mod0.shape)
```

```
data shape: (64, 512, 128)
```

European XFEL h5tools

- Combine parameter from different sources
- Iterate over run
- Find a specific train

The detector contains 16, each are independant data sources. We can combine all of them in a single array.

In [10]:
```
# Combine all modules into a single array
full_detector_image = stack_detector_data(data, 'image.data')

# shape: (pulses, modules, x, y)
full_detector_image.shape
```

Out[10]:
```
(64, 16, 512, 128)
```

We can iterate easily over all train using our train generator. Here we iterate over the next 10 trains and combine all the detector modules in a single array.

In [11]:
```
i = 0
for tid, data in trains:
    full_detector_image = stack_detector_data(data, 'image.data')
    print('train:', tid, 'det:', full_detector_image.shape)
    i+=1
    if i == 5:
        break
```

```
train: 1541484693 det: (64, 16, 512, 128)
train: 1541486128 det: (64, 16, 512, 128)
train: 1541486129 det: (64, 16, 512, 128)
train: 1541486130 det: (64, 16, 512, 128)
train: 1541486131 det: (64, 16, 512, 128)
```

We can also retrieve a specific train contained in the run.

In [12]:
```
# Retrieve a specific train by his train ID
tid, data = run1.train_from_id(1541486130)
print('retrieved train 1541486130:', tid)
# Or by index
tid, data = run1.train_from_index(100)
print('retrieved 101th train:', tid)
```

```
retrieved train 1541486130: 1541486130
retrieved 101th train: 1541486226
```

European XFEL h5tools

- Find trains to exclude
- Filter

Here we check if a a detector module data is missing in any train.

In [13]:
```
for i in range(len(run1.ordered_trains)):
    nb_sources = len(run1.ordered_trains[i][1])
    if nb_sources < 16:
        print('train {}: only {} modules found'.format(i, nb_sources))
```

```
train 2: only 8 modules found
train 250: only 8 modules found
```

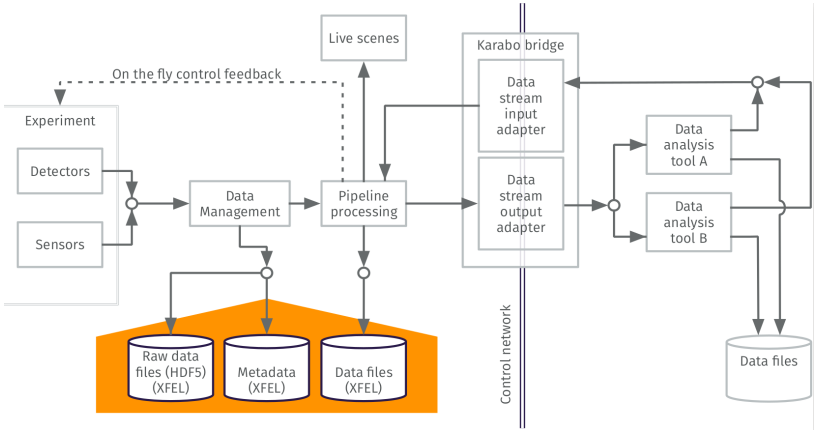
While retrieving train data, it is possible to filter only interesting data sources, and parameters.

In [20]:
```
# dict holding only what we are interested in
devs = {'SPB_DET_AGIPD1M-1/DET/5CH0:xtdf': {'image.data', 'image.gain'}}

for i in range(12, 15):
    tid, data = run1.train_from_index(i, devices=devs)
    print('train:', tid)
    print('sources', data.keys())
    print('parameters', data['SPB_DET_AGIPD1M-1/DET/5CH0:xtdf'].keys())
    print('***')
```

```
train: 1541486138
sources dict_keys(['SPB_DET_AGIPD1M-1/DET/5CH0:xtdf'])
parameters dict_keys(['image.gain', 'metadata', 'image.data'])
***
train: 1541486139
sources dict_keys(['SPB_DET_AGIPD1M-1/DET/5CH0:xtdf'])
parameters dict_keys(['image.gain', 'metadata', 'image.data'])
***
train: 1541486140
sources dict_keys(['SPB_DET_AGIPD1M-1/DET/5CH0:xtdf'])
parameters dict_keys(['image.gain', 'metadata', 'image.data'])
***
```

Roadmap



- Improve Karabo bridge
  - Data feedback to Karabo
  - Support more language on client side
  - Improve documentation and use cases examples
- Improve offline h5 tools
  - Add command line mode
  - Provide pipelined based functionalities:

  Run → data reduction → calibration → analysis → filter → conversion → output
- Add functionalities:
  - Data conversion
    - ► Numpy, plain text, Matlab, png, CXI, JSON, python, csv, …
  - Analysis functions
    - ► Photon count, detector geometry, integration, …
  - Filter
    - ► Use analysis snippets as input for data reduction
- Performance improvements

Summary

- Outlined basics of online and offline data analysis
- Support resources (https://github.com/European-XFEL)
  - Karabo bridge client
  - Euxfel-h5tools
  - Data analysis recipes
  - Example data on Maxwell cluster: /gpfs/exfel/exp/XMPL/
- Keen to work with users
  - Helps prioritize features
  - Avoid duplication of effort and code
  - --> Get in touch
- Contact
  - Thomas.Michelat@xfel.eu, Hans.Fangohr@xfel.eu, Sandor.Brockhauser@xfel.eu
- Literature
  - H. Fangohr et al, Data Analysis support in Karabo at European XFEL, ICALEPSC 2017, online: http://icalepcs2017.vrws.de/papers/tucpa01.pdf