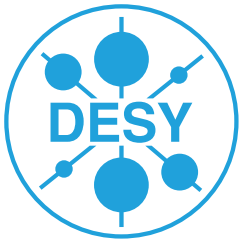


AIDA Beam Telescope Requirements for a Common DAQ

Hanno Perrey

From EUDET to AIDA Beam Telescope

EUDET style: "Event-based"

- One trigger per slowest DAQ system in the telescope
- No triggers from TLU while at least one system is *BUSY* (e.g. Mimosa26 double-frame readout: 115.2 µs)
- Data is written in single stream
- Low rates/data-taking efficiency for fast (LHC-type) DAQ systems

AIDA style: "Particle-based"

- TLU issues trigger for every particle (i.e. scintillator signal)
- Triggers stop only on *VETO* from any DAQ
- High-rate studies possible
- Much higher data-taking efficiency at high-rate beams
- Telescope DAQ needs to cope with high data rates (e.g. continuous readout of Mimosa28 quad-planes)

⇨ Data in asynchronous streams from various DAQs

Implications and To-do List for Telescope Framework

Hardware

- Integrate with AIDA TLU (common clock, timestamp triggers)
- Mimosa-based planes with fast device (FEI4/TimePix) for timestamping of hits in reconstruction
- Continuous readout of Mimosa planes (if triggers present)

DAQ Software (EUDAQ)

- Reduce disk IO/network bottlenecks: DAQs (can) store data locally
- Add all available timing (meta) information (from TLU/DAQ clocks) into data format
- Need hooks for online data verification and monitoring (next slides)

Analysis Software (EUTelescope)

- Merge data streams from different DAQs based on timing information and hits (already done by many groups)
- Output timetagged tracks

Reminder: EUDAQ Goals and Features

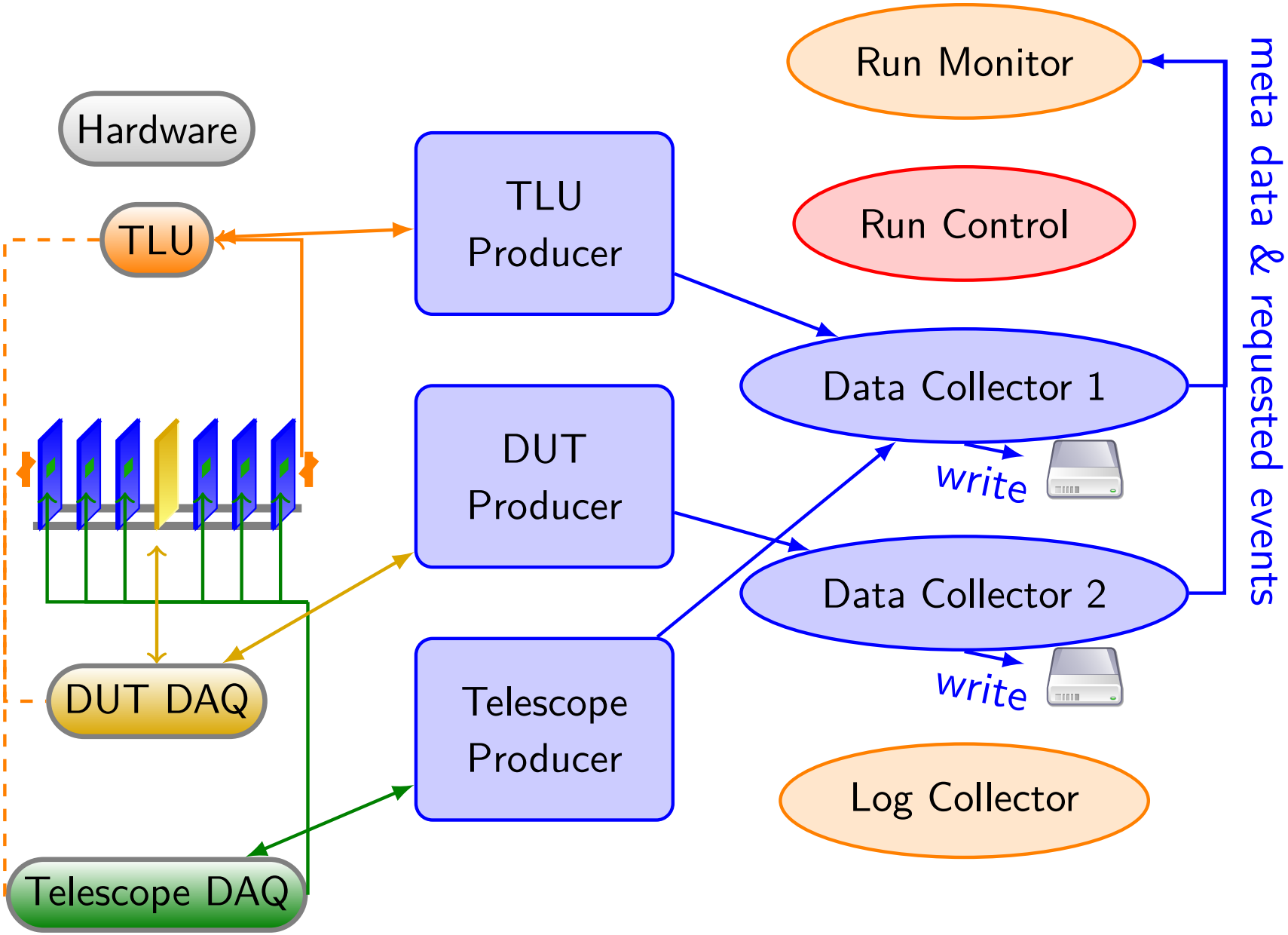
- Generic framework for data acquisition
- OS independent: Linux, Mac OSX, Windows
- Modular and flexible design
- Provides central DAQ control, data handling and storage, log collection, online monitoring
- Components communicate via TCP/IP and can run on different networked machines
- Focused on easy and flexible integration of the device under test including pre-existing DAQs
- All hardware communication done by "Producers" with equal rights

AIDA Beam Telescope DAQ: EUDAQ 2.0

- Accepts data packets from DAQs covering:
  - single trigger ("classic mode")
  - list of (timestamped) triggers
  - time range (shutter/data driven devices)
  - both time range and trigger list
- Writes data into multiple streams e.g. DAQ local storage
- Online verification in this scheme:
  - separate on-the-fly event building (full or partial for specific triggers)
  - cross-check timestamps e.g. against TLU information
  - merged data can be used for online monitoring/immediate offline analysis
- Will be backward-compatible to old integration efforts (no more than a recompilation should be needed)

⇨ Easier integration with a wider range of DAQ concepts

Schematic Layout of EUDAQ 2.0 (preliminary)



Integrating a Device into Telescope/EUDAQ 2.0

TLU

- Receive clock, trigger and/or shutter signals
- Timestamp events (trigger, shutter open/close, ...)

Producer to interact with RunControl and Hardware

- Implemented in C++, but Python interface exists (see backup)
- Receives commands from RunControl (Configure, Run Start/Stop)
- Talks to the DAQ hardware, receives data from there
- Sends its data to a DataCollector (optional)
- Gain flexibility with new event format and multiple data streams
  $\Rightarrow$ wider range of devices suitable for (full) integration

Optional: Data Converter Plugin

- Converters native device format into defined structure
- Used for conversion into LCIO and for online monitoring

Status of EUDAQ development

- A lot of maintenance has been done on EUDAQ over last months and is ongoing (git, CMake, platform-independence, Python interface, ...)
- RunControl can already assign a different DataCollector to each Producer and the resulting files can be merged offline
- Implementation of new event format started
- Best strategy for online (partial) event building being investigated
- A handful of developers committed to this task (both PhD and seniors)

Summary and Outlook

- EUDAQ is a very flexible DAQ framework and has been used successfully for a long time with EUDET-family of telescopes
- Support one-trigger-per-particle operation of AIDA beam telescope requires extension of EUDAQ and removal of IO/network bottlenecks
- Key changes: timestamped events, multiple data streams
- Makes integration with EUDAQ 2.0 easier and more flexible than before
- Development on EUDAQ is ongoing, contributions are welcome!

Download the newest version and follow the development at `http://eudaq.github.io`

Overview Backup Slides

Integrating a DUT into EUDAQ

The Anatomy of a *Producer*

- A Producer needs to implement command receiving methods of the `Producer` base class:
  - `OnConfigure`, `OnStartRun`, `OnStopRun`, `Terminate`
- It configures the hardware according to the config received from RunControl
- It (optionally) sends its data to the data collector:
  either in raw format or converted to a custom `StandardEvent` class
- It (optionally) logs status/error messages

Example code is provided!
$\rightarrow$ see also shortened examples on the following slides

Example C++ (Pseudo-) Code for a EUDAQ Producer

```
class ExampleProducer : public eudaq::Producer {
  public:
    ExampleProducer(){}
    virtual void OnConfigure(const eudaq::Configuration & config) {
      // .... configure your hardware
    }
    virtual void OnStartRun(unsigned param) {
      // .... prepare for and start run
    }
    virtual void OnStopRun() {
      // .... stop your DAQ
    }
    void ReadoutLoop() {
      while (true) {
        // while running:
        // ..... get raw data, put it into RawDataEvent and send
      }
    }
};
int main(int /*argc*/, const char ** argv) {
    ExampleProducer producer();  // Create a producer
    producer.ReadoutLoop();      // And set it running...
    return 0;
}
```

Example Python Code for a EUDAQ Producer

```
#!/usr/bin/env python2
execfile('PyEUDAQWrapper.py') # load the ctypes wrapper
from time import sleep
import numpy # for data handling
# create PyProducer instance
pp = PyProducer("testproducer","tcp://localhost:44000")

# wait for CONFIGURE cmd from RunControl
while not pp.Configuring:
    sleep(1)
# .... do your config stuff here ...
pp.Configuring = True
# check for RUNSTART cmd from RunControl
while not pp.StartingRun:
    sleep(1)
# ... prepare your system for the immanent run start
pp.StartingRun = True
# starting to run main DAQ loop
while not pp.Error and not pp.StoppingRun and not pp.Terminating:
    # prepare an numpy array for raw data storage
    data = numpy.ndarray(shape=[1,3], dtype=numpy.uint64)
    # .... get your data from your hardware ...
    pp.SendEvent(data)          # send event off
```

Running EUDAQ

- OS independent (Linux, Mac OSX, Windows)
- Components can be run on different networked machines
- Different interfaces available: Qt GUI, console, Python