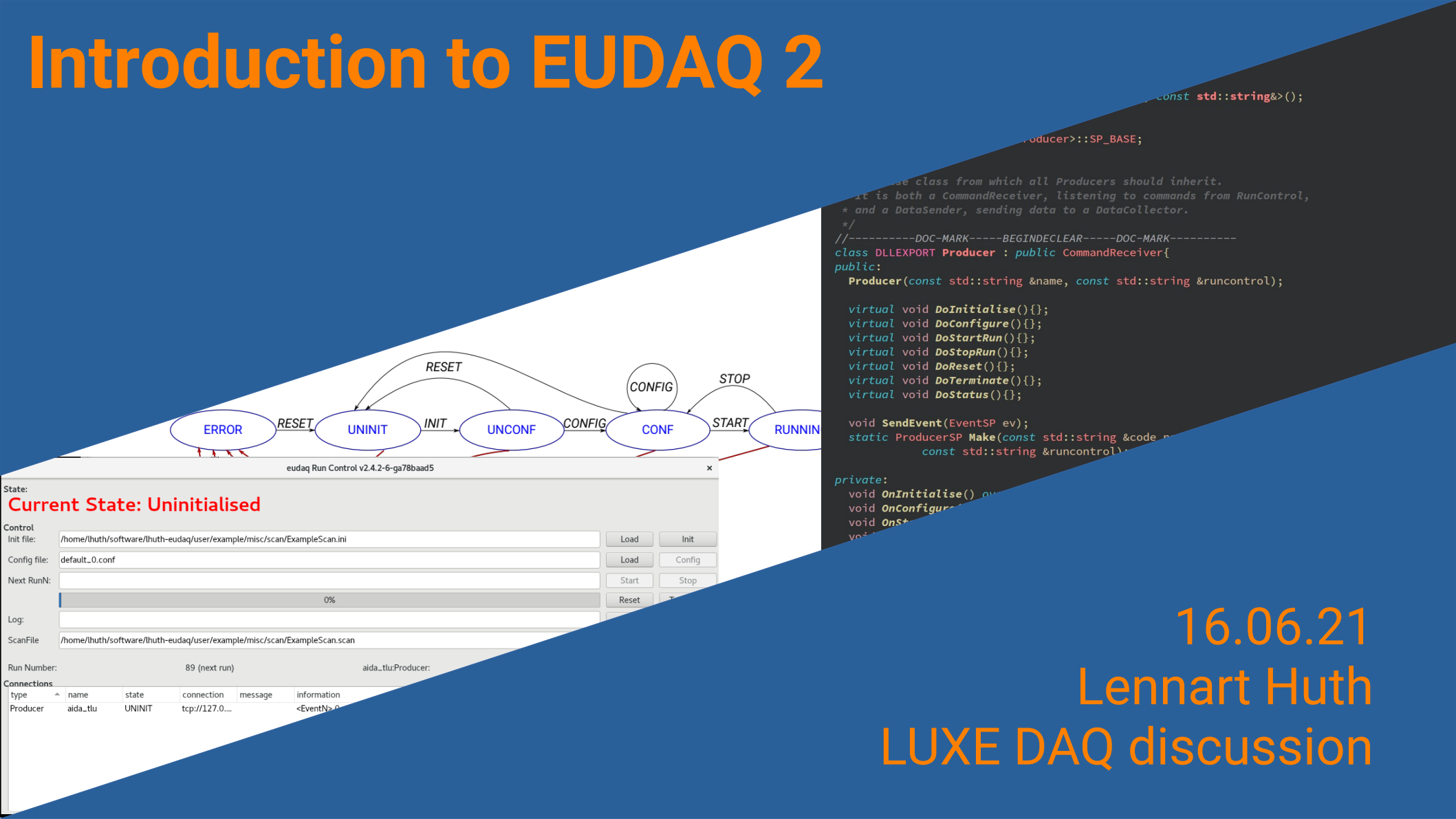
Introduction to EUDAQ 2



16.06.21

Lennart Huth

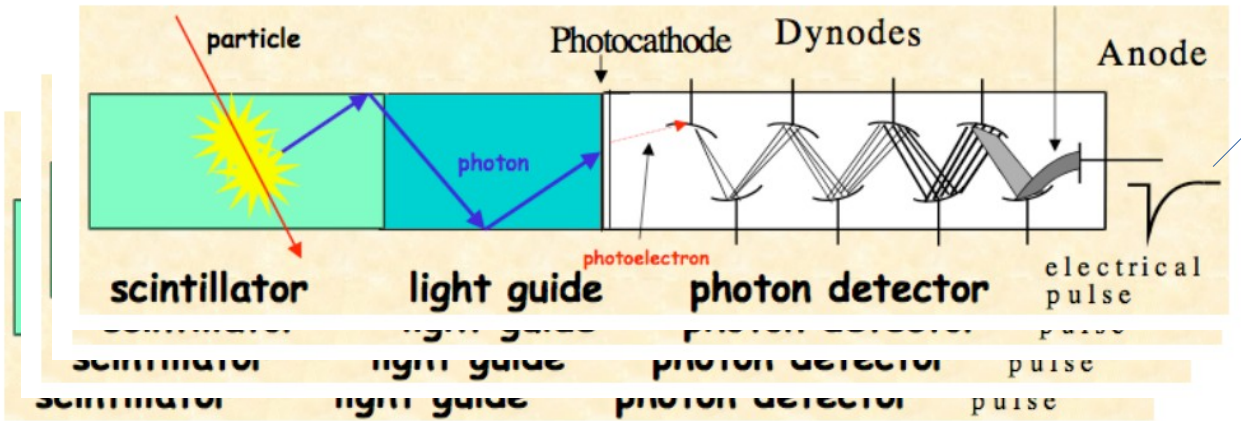
LUXE DAQ discussion

Outline

- Basics of data acquisition system
- Multi-platform network DAQ: EUDAQ2
- Implementing a device in EUDAQ2
- Data quality monitoring

I interpret this meeting as a discussion – so feel free to interrupt me during the presentation or ask any question at the end!

From particle interaction to data



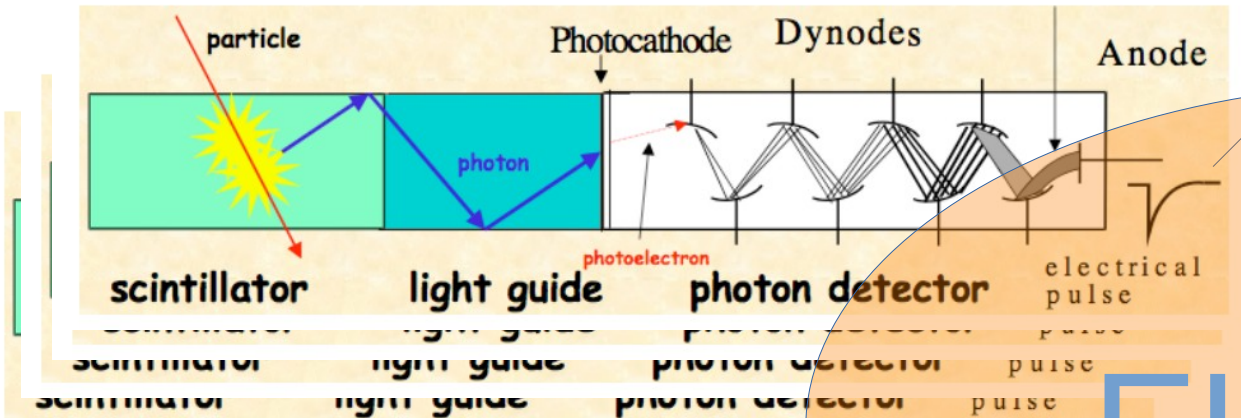
Digitization

Processing

Data quality monitoring

Storage

From particle interaction to data



Digitization

Processing

EUDAQ2

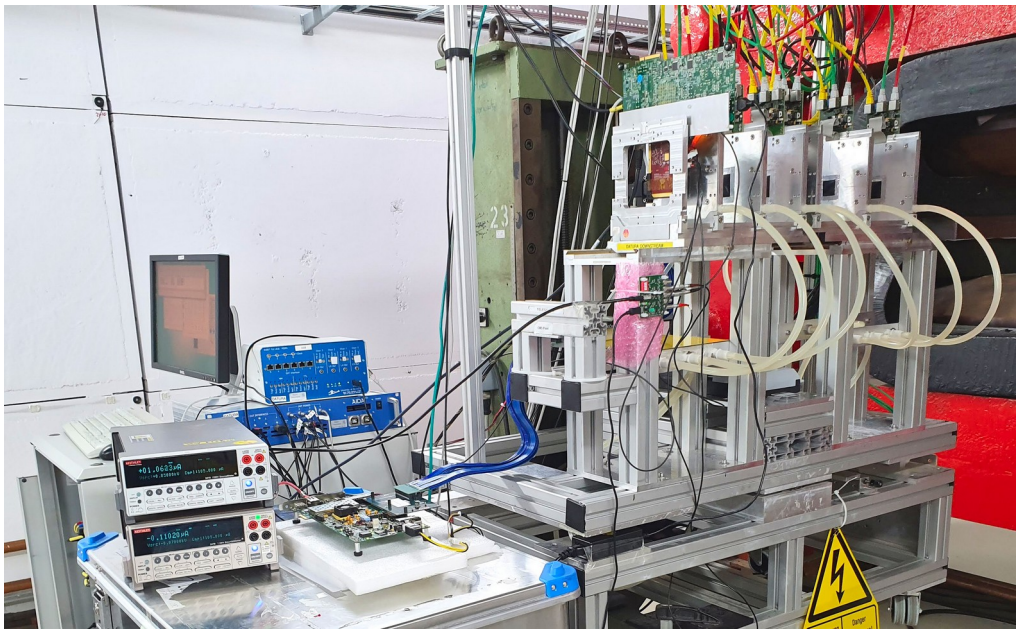
Storage

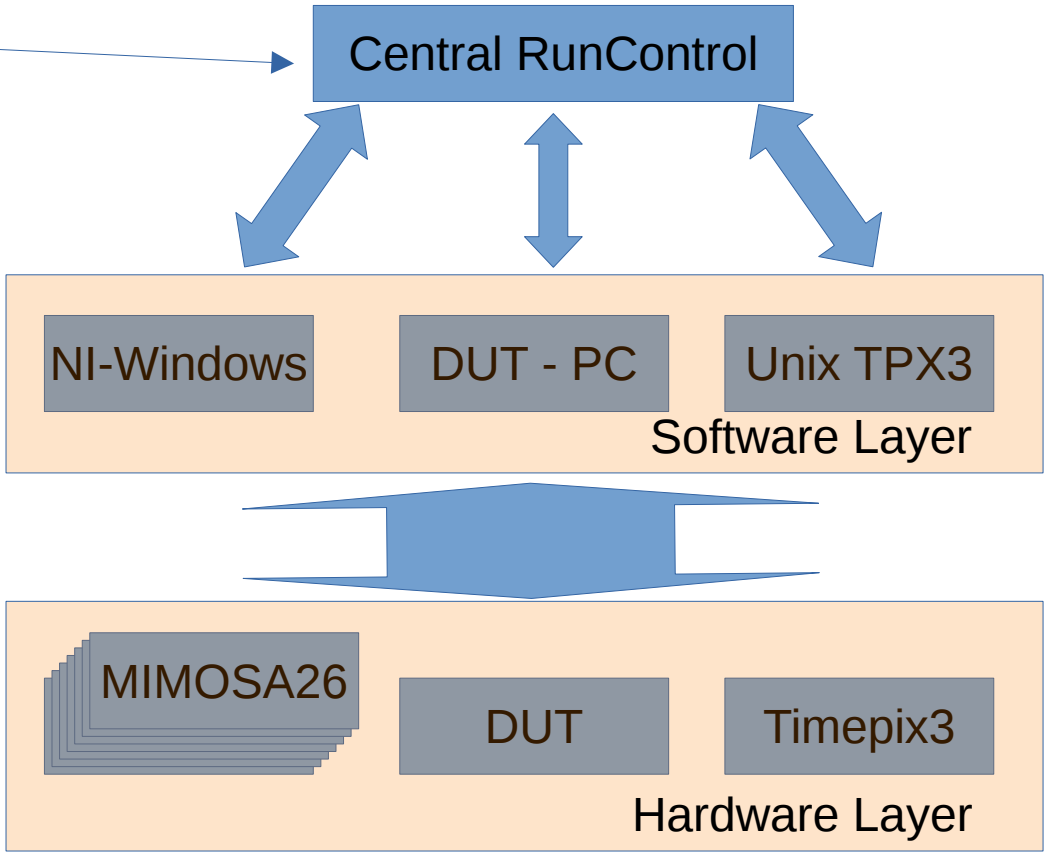
Data quality monitoring

A bit more complex example

Operating a MIMOSA Telescope at the test beam with an additional timing layer and a DUT

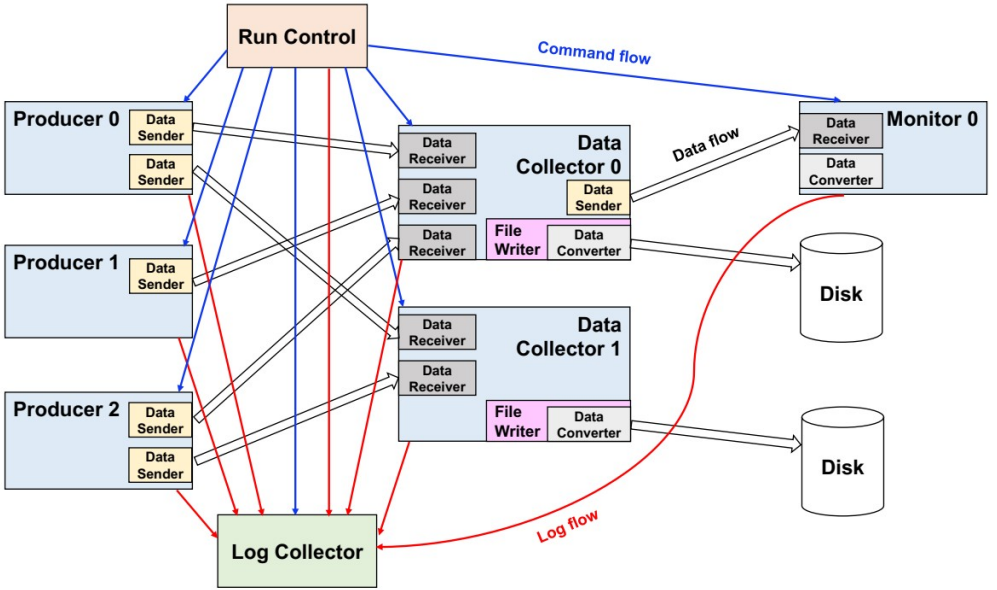
In charge of steering and monitoring all DAQ sub systems

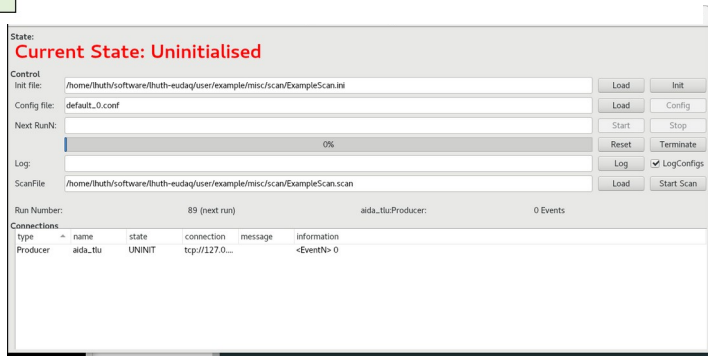




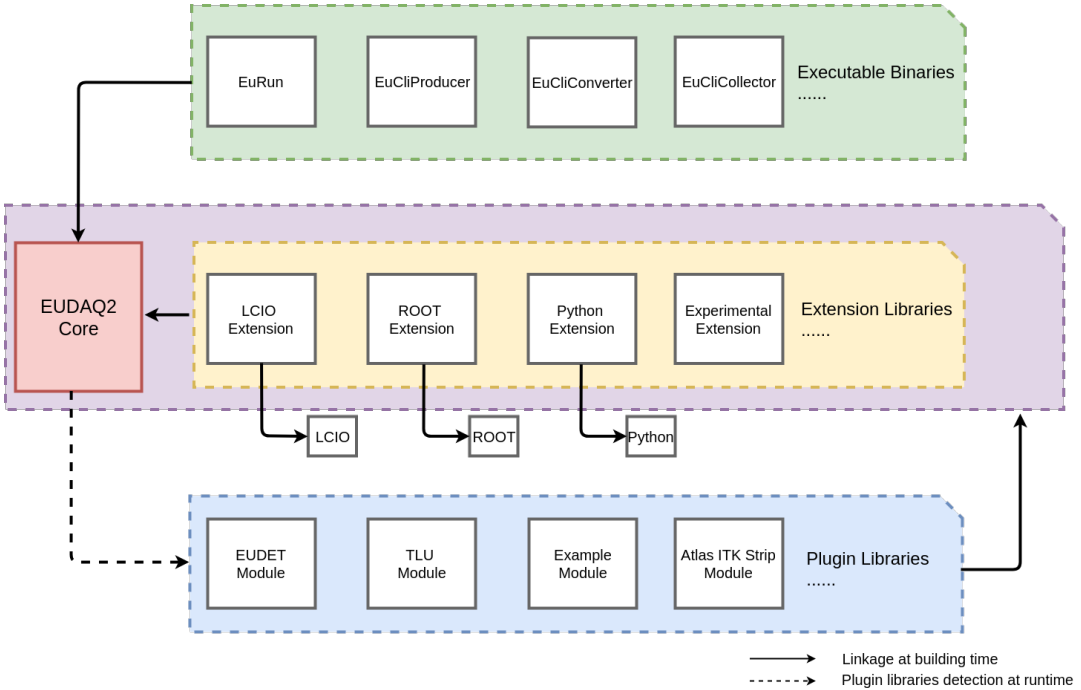
EUDAQ2 - Concept

- One central runcontrol instance to steer all other components within the same network
- Runs on MAC,Windows and Unix – developments focused on unix (no request for other OS by users since years)
- Finite state machine defining behavior
- Communication with different computers via custom TCP/ stack
- Each (set of) physical devices is implemented as a ‘Producer’
- Offers data sorting based on trigger Ids or event numbers. Alternatively direct data storage
- Provides basic data quality monitoring
- Central instance to collect log messages/errors etc
- Synergies with the AIDA TLU (see next talk by D. Cussans)





EUDAQ2 Architecture/Linking



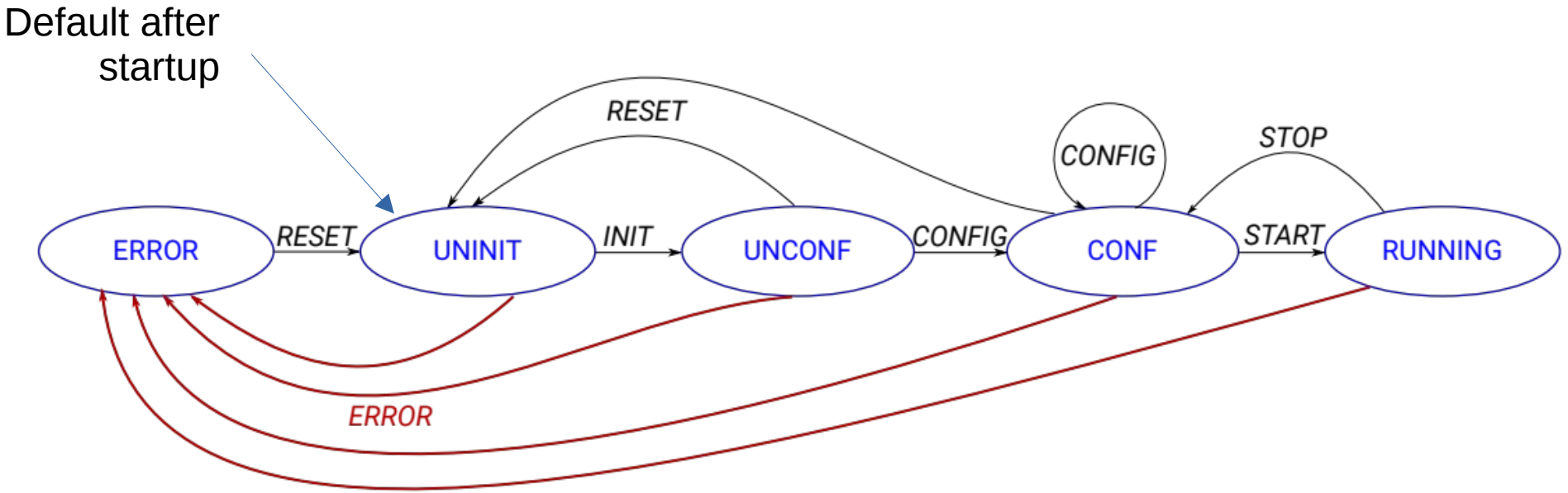
EUDAQ2 is split into several components:

- Core: All central components discusses before
- Executable
- Plugin library: User code, which is loaded at run time if requested

→ Only the core and the plugin to control the detector needs to be installed on the readout-PC. Reduces build complexity

Finite State Machine

Runcontrol state is defined as the lowest state of any component
Each component reports its state to the run control

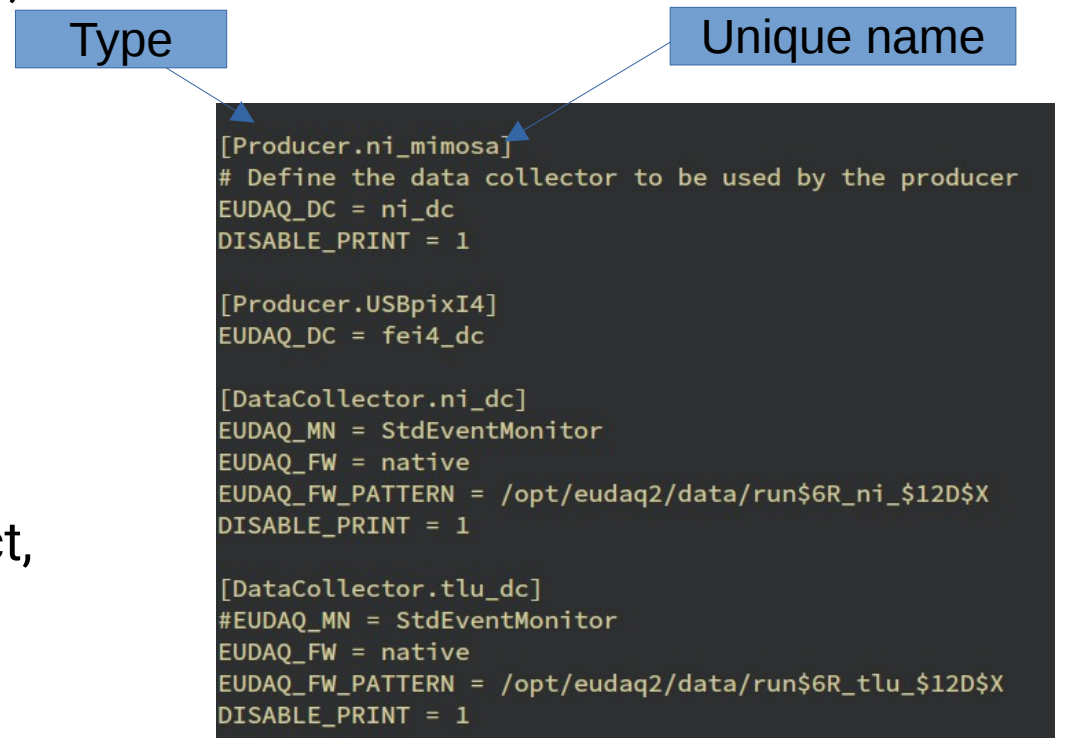


Time consuming once after power up steps: At initialize → Frequent steps: At configure

Communication & Configuration

- All components (but the runcontrol) inherit from a CommandReceiver, which implements all required communication protocols
  - Connection procedure
  - Virtual function to react on state changes
  - Connections to a logging instance
  - TCP/IP connection
- A central logger is used to collect, display and store Log messages from all connected components

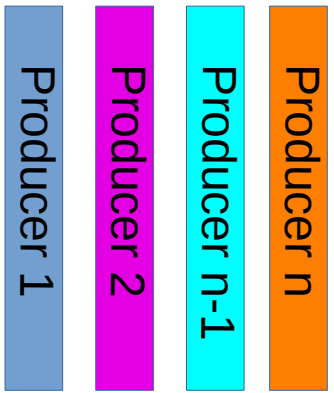
Configuration/Initialization are passed by simple plain text files:



Data Collectors

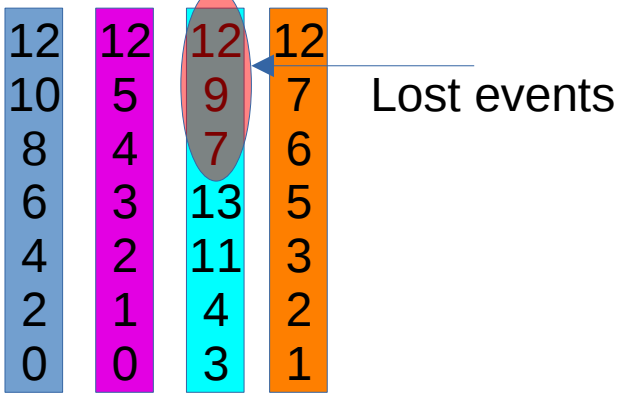
Any data collection method can be implemented – currently three versions are provided

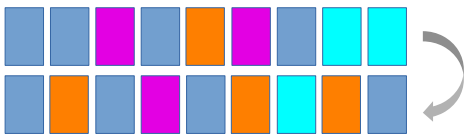
Direct data storage

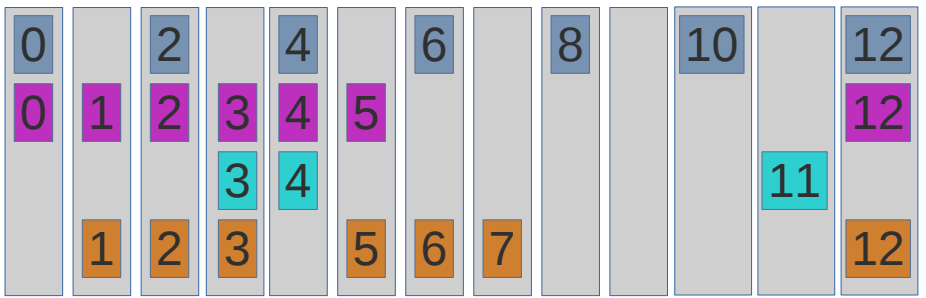



```
[Producer.ni_mimosa]
# Define the data collector
EUDAQ_DC = ni_dc
DISABLE_PRINT = 1

[Producer.USBpixI4]
EUDAQ_DC = fei4_dc

[DataCollector.ni_dc]
EUDAQ_MN = StdEventMonitor
EUDAQ_FW = native
```

Triger/EventID sorting

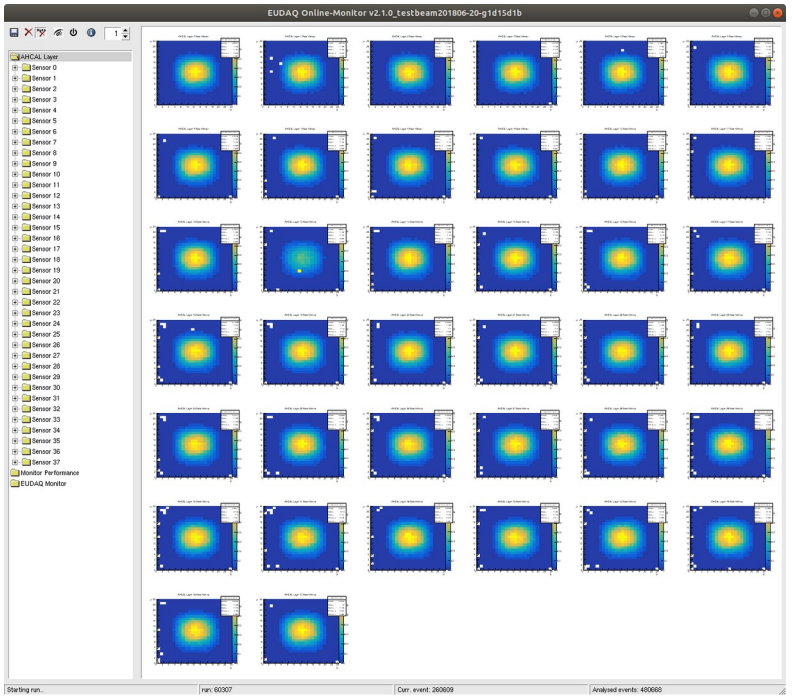






Default Online Monitor

- Root(6) based gui
- Plots inter event correlations and hitmaps of all devices in a single data stream
- Requires sorted data collection
- Fixed fraction of events will be forwarded to monitor
- Monitored values are limited to x/y position
- More flexible root monitor implemented by testbeam users: RootMonitor (would be a talk on its own:BTTB8 Talk)



User Code - Producer/Converter

Producer

```
class DLLEXPORT Producer : public CommandReceiver{
public:
  Producer(const std::string &name, const std::string &runcontrol);

  virtual void DoInitialise(){};
  virtual void DoConfigure(){};
  virtual void DoStartRun(){};
  virtual void DoStopRun(){};
  virtual void DoReset(){};
  virtual void DoTerminate(){};
  virtual void DoStatus(){};

  void SendEvent(EventSP ev);
  static ProducerSP Make(const std::string &code_name, const std::string &run_name,
            const std::string &runcontrol);

private:
  void OnInitialise() override final;
  void OnConfigure() override final;
  void OnStartRun() override final;
  void OnStopRun() override final;
  void OnReset() override final;
  void OnTerminate() override final;
  void OnStatus() override;

private:
  uint32_t m_pdc_n;
  uint32_t m_evt_c;
  std::mutex m_mtx_sender;
  std::map<std::string, std::shared_ptr<DataSender>> m_senders;
};
//----------DOC-MARK-----ENDDECLEAR-----DOC-MARK----------
}
```

Implement these functions

Converter

Producer stores data as char vector – converter implements the re-conversion to a std event

```
namespace eudaq{
  template <typename T1, typename T2> class DataConverter;

  template <typename T1, typename T2>
  class DLLEXPORT DataConverter{
  public:
    using ConverterUP = std::unique_ptr<DataConverter<T1, T2>>;
    using T1SP = std::shared_ptr<T1>;
    using T1SPC = std::shared_ptr<const T1>;
    using T2SP = std::shared_ptr<T2>;
    using T2SPC = std::shared_ptr<const T2>;

    DataConverter() = default;
    DataConverter(const DataConverter &) = delete;
    DataConverter& operator = (const DataConverter &) = delete;
    virtual ~DataConverter(){};
    virtual bool Converting(T1SPC d1, T2SP d2, ConfigurationSPC conf) const = 0;
  };
}
#endif
```

Getting started and Summary

EUDAQ2 provides:

- A finite state machine to steer a set of subsystems
- Common Data structure
- Network communication
- Logging
- Basic DQM
- Data storage

EUDAQ2 is a community project and actively used within test beams → User input always welcome

- Check the repository:

  eudaq2

- Run the examples and get familiar with the structure
- Implement your own devices
- Test beam campaign to study your developments under realistic conditions :)

Backup

Basic principles

In general

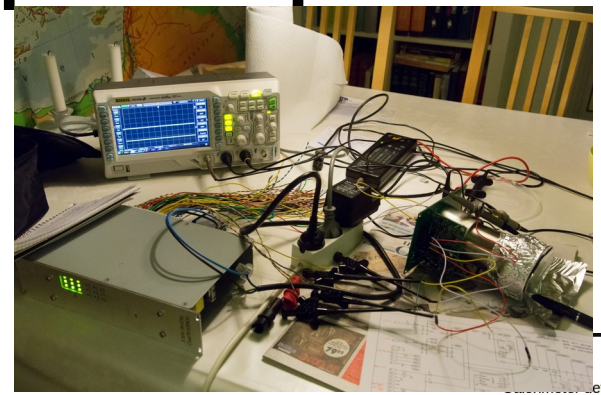


image credit

- Three core components:
  - The detector
  - Readout hardware
  - Readout software
- A very simple example: Reading out a photomultiplier with an oscilloscope.
- This talk will focus on the last component: readout software (and some level of detail about detector hardware)
- Starting and stopping data taking
- State machines to control readout

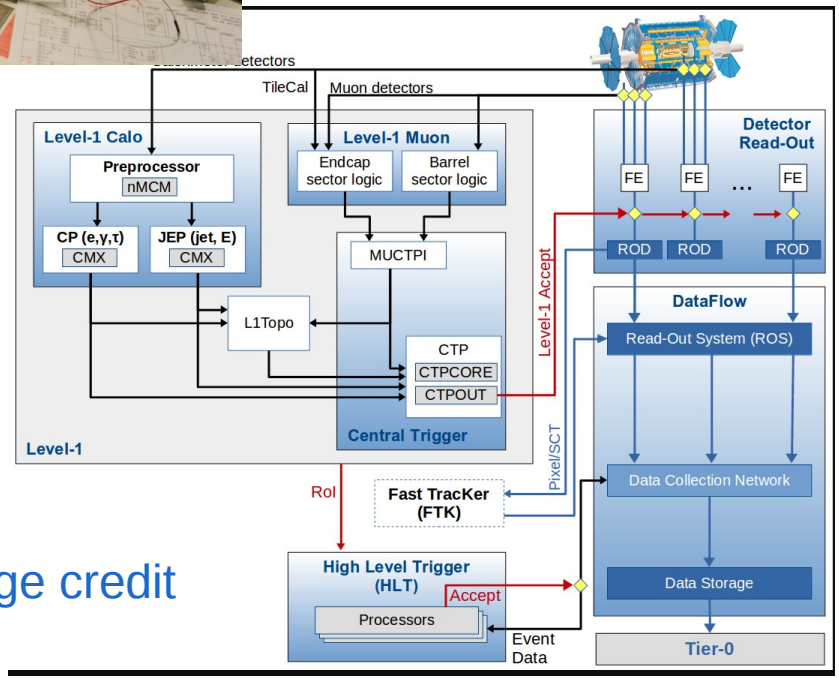


Image credit