Synchronous Data Service (SDS) EPICS module based on PVXS and Normative Types for ESS

Fall 2024 EPICS Collaboration Meeting – ORNL, Oak Ridge TN, USA

João Paulo Martins
Juan F. Esteban Muller
Integrated Control Systems Division (ICS)

2024-09-23

Agenda

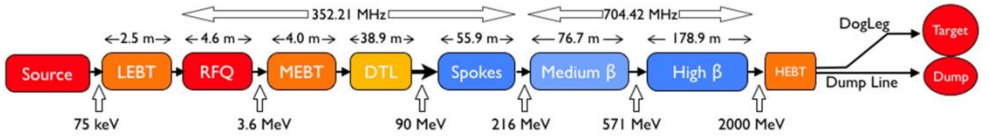
1

Introduction

Introduction

European Spallation Source (ESS) Status

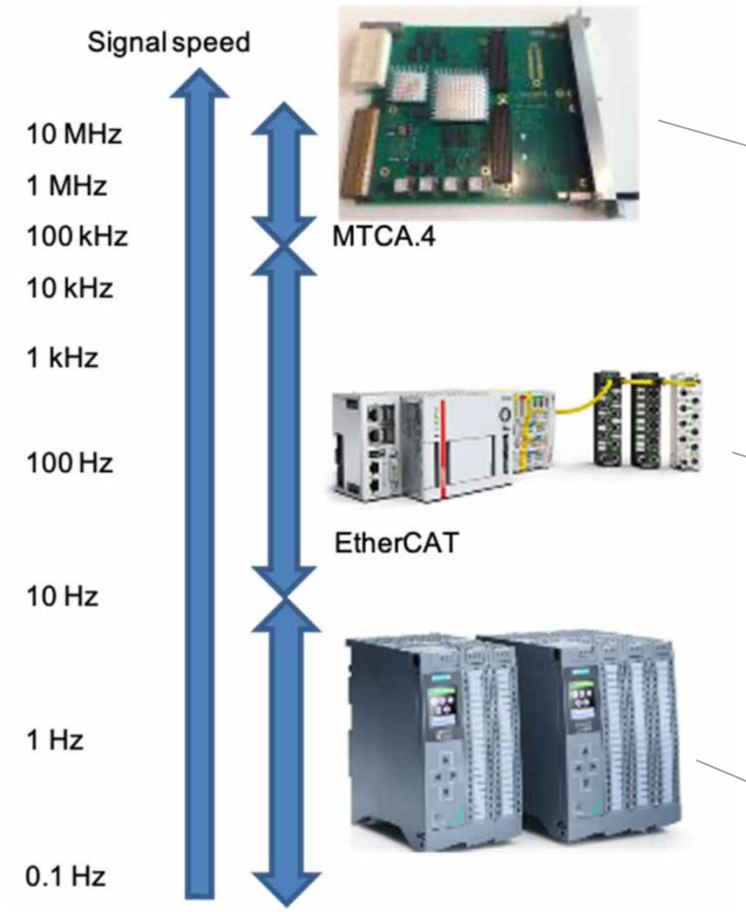
- ESS in a nutshell:
  - Linear Proton Accelerator designed for a beam average power of 5MW;
  - Rotating tungsten target station to produce neutrons;
  - (up to) 22 neutron instruments (beamlines);
- Current Status:
  - Installation and conditioning of the latest cryomodules;
  - Beam on dump scheduled to January/25;
  - Beam on target in September/25;
  - User program to start in 2027;



Parameter	Value
Ave power (design) [MW]	5
Max energy (design) [MeV]	2000
Peak current [mA]	62.5
Pulse length [ms]	2.86
Rep rate [Hz]	14
Duty factor [%]	4
RF freq [MHz]	352.21/704.42

Introduction

Control System Hardware Strategy defined by ICS



MTCA.4 for applications with data acquisition exceeding few kHz;
These applications require a FPGA and custom, high-speed signal processing in system;

For slower signals, EtherCAT will be used as a real-time fieldbus with good price/performance ratio;
Synchronization and event information are key for applications where a full custom platform solution would be too costly;

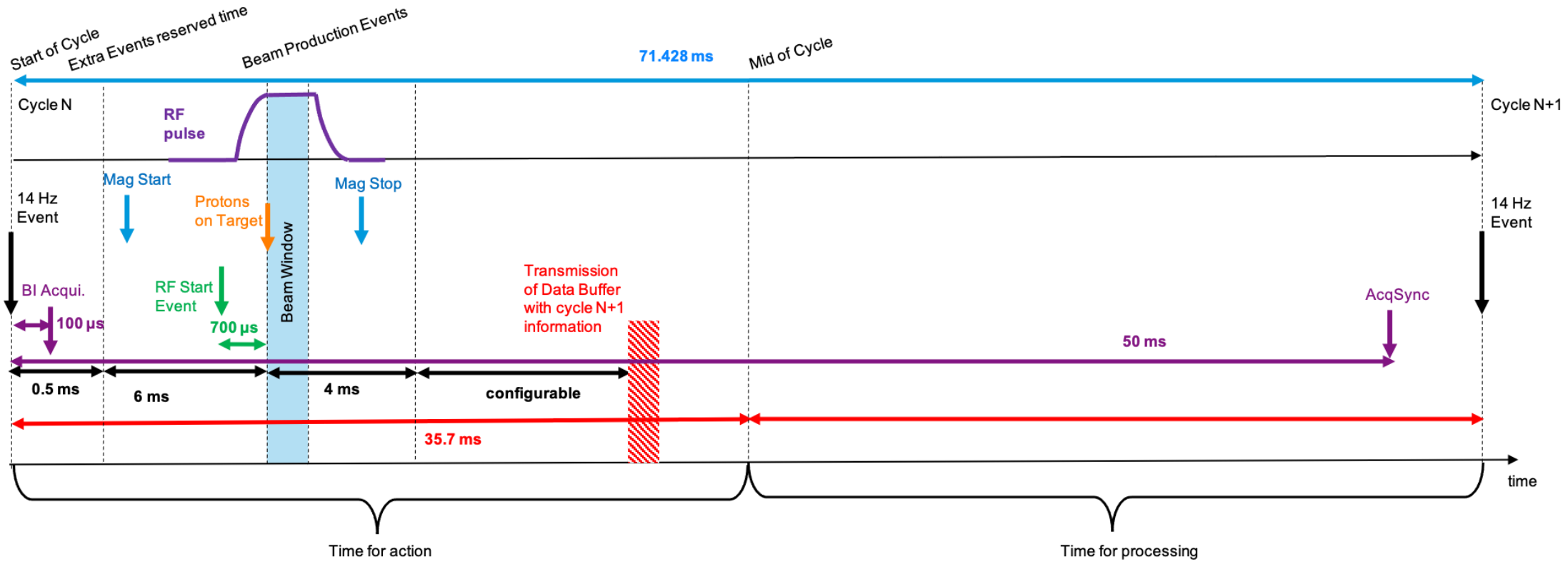
Low speed signals are handled with commercially available PLC systems;
This is a cost-effective solution that addresses ESS reliability and maintainability requirements;
The PLCs are connected to EPICS for further integration into the control system;

2

High-speed Data Acquisition Systems at ESS

High-speed DAQ Systems at ESS

Overview of the Timing Structure of the ESS Linac



High-speed DAQ Systems at ESS

Overview of the MTCA applications

- MTCA is used in multiple systems:
  - LLRF, RFLPS
  - BPM, BCM, BLMs
  - other PBI systems (FC, WS, EMUs, …)
  - Fast Beam Interlock System

- Timing Distribution (MRF)
  - MRF MTCA Event Master
  - MRF MTCA Event Receiver

High-speed DAQ Systems at ESS

Basic workflow of a MTCA data acquisition IOC

- Timing System events arrive to the EVR which then generates electrical triggers on the MTCA backplane
- FPGAs in the AMCs receive triggers and take actions:
  - Start acquisition on ADCs
  - Real-time tasks
- FPGA interrupts CPU via PCIe
- Worker thread on IOC device support reads data from FPGA

High-speed DAQ Systems at ESS

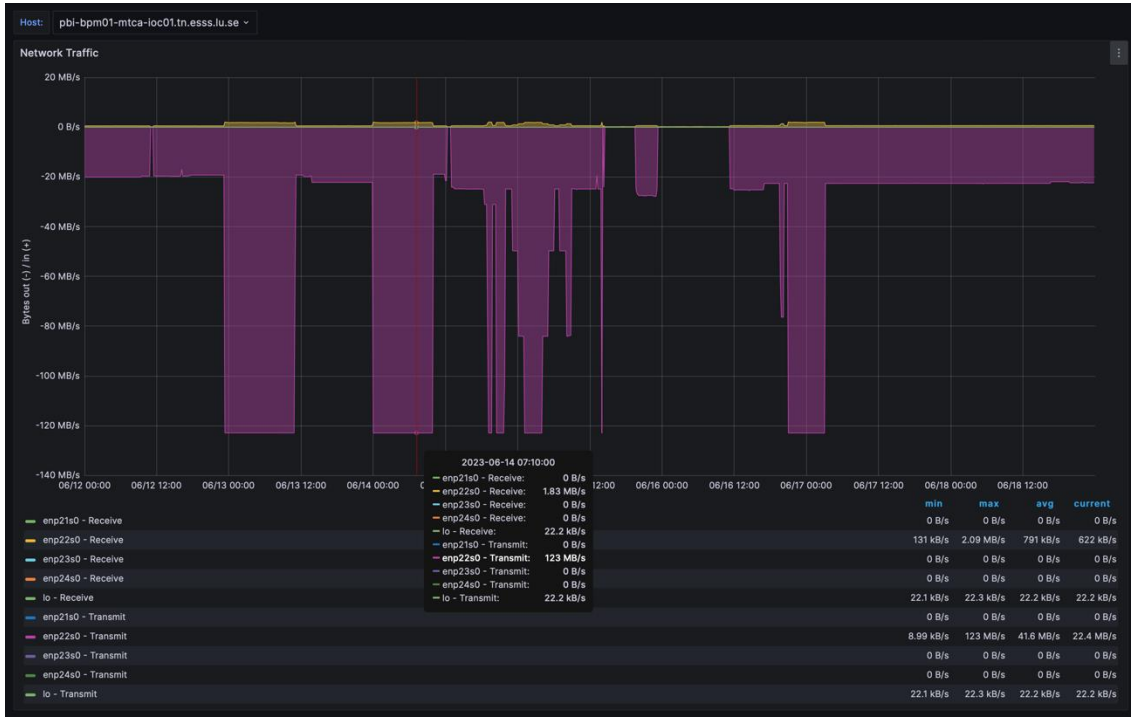
Basic workflow of a MTCA data acquisition IOC

- All readout parameters are extracted from the FPGA and published to EPICS records (I/O Intr)
- Array datasets are published as EPICS waveform/aai records (I/O Intr)
- Acquisition parameters are defined per system:
  - Sampling rate
  - Acquisition length (number of points)
  - Decimation
- Other modes also exist (circular buffer)
- Update rate: 1 Hz – 14 Hz

High-speed DAQ Systems at ESS

Archiving the waveforms

- DAQ waveforms are valuable data for many stakeholders
- We lack a standard way to store the datasets, besides the Archiver Appliance
  - Some systems implement a feature to save the data locally (NDPluginHDF5 or raw binary files)
- Waveforms are being monitored by OPIs and in the Archiver Appliance
  - Potential risk to overload the network
  - Indiscriminate use of permanent storage
  - Cumbersome to extract and correlate data



3

Synchronous Data Service (SDS)

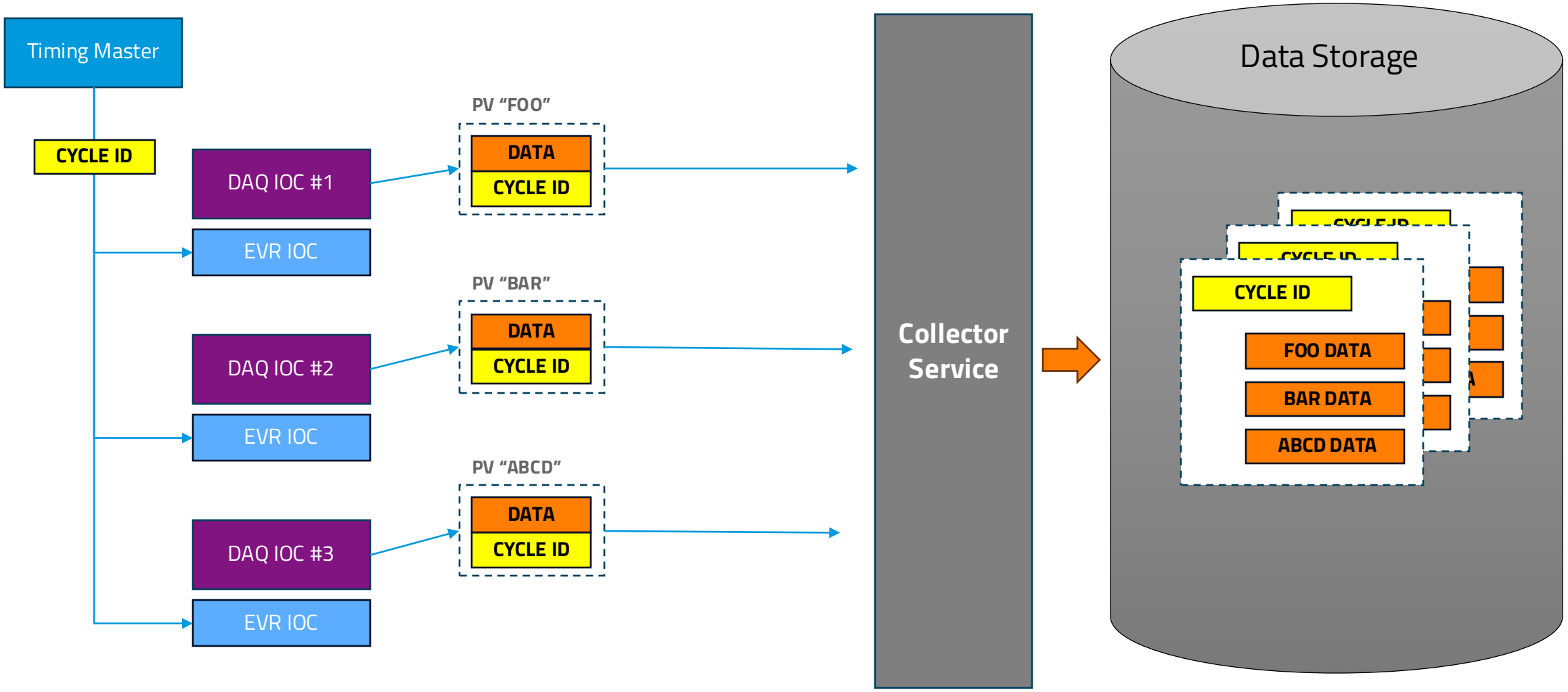
Synchronous Data Service (SDS)

Overview

- Given that ESS is a pulsed machine, SDS aims to collect and store data indexed by an unique cycle ID instead of the time series fashion
- It is NOT a new Archiver but rather a complementary service
- The main objective is to facilitate the post-analysis and correlation of data from different accelerator systems
- An opportunity to implement a global post-mortem and on-demand data collection system

Synchronous Data Service (SDS)

General Architecture



Synchronous Data Service (SDS)

First attempt to implement at the IOC level

- Since EPICS Base release 7.0.6 every record has the UTAG field (64 bits)
  - UTAG is copied together with TIME using the TSEL field
  - UTAG value is mapped to userTag field of a Normative Types PV
- Contribution to mrfioc2 to enable a custom UTAG field on the event counter records
  - Event counters are used as timestamp source
  - Since release 2.5.0

```
record(int64out, "$(EN)-SP") {
  field(DTYP, "EVR Event Utag")
  field(SCAN, "I/O Intr")
  field(OUT , "@OBJ=$(OBJ),Code=$(CODE)")
  field(VAL , "0")
  field(TSE , "-2") # from device support
  field(FLNK, "$(EN)Cnt-I")
  info(autosaveFields_pass0, "OUT")
}

record(calc, "$(EN)Cnt-I") {
  field(DESC, "TS and UTAG source")
  #field(SDIS, "$(EN)-SP")
  #field(DISV, "0")
  field(CALC, "A+1")
  field(INPA, "$(EN)Cnt-I NPP")
  $(SFTSEN=)field(TSEL, "$(EN)-SP.TIME")
}
```

```
> dbgf LabS-ICS:Ctrl-EVR-411:EvtF14HzCnt-I.UTAG
DBF_UINT64:          15336819666 = 0x392254bd2
```

Synchronous Data Service (SDS)

First attempt to implement at the IOC level

Issues:

- UTAG is 64-bit but userTag is 32-bit
- TSEL mechanism does not copy UTAG when link is between two separate IOCs

This idea is not out of the table yet:

- We will use the newer versions of mrfioc2 with UTAG on event counters

Another approach: use QSRV Group PV

```
#- Cycle ID params
record(int64in, "$(P)$(R=)#SDSMDataCycleIdIn") {
    field(DESC, "IDCycle")
    field(INP, {pva:{pv:"$(PEVR)IdCycle",proc:false,time:true}})
    field(TSE, "-2")

    info(Q:group, {
        "$(P)$(R=)SDSMetadata":{
            "value":{+type:"plain", +channel:"VAL"},
            "cycleId":{+type:"meta", +channel:"TIME"},
            "cycleId.value":{+type:"plain", +channel:"VAL"},
            "sdsInfo.cycleId":{+type:"plain", +channel:"VAL"},
        }
    })

    field(FLNK, "$(P)$(R=)#SDSMDataSdsEvtInfo")
}
```

4

SDS EPICS Module

SDS EPICS Module

Motivation, requirements and constraints

Motivation

- Facilitate the offline analysis and correlation of data from different systems
- Handle array datasets in a sensible way
- Explore the capabilities of EPICS 7 and PVA

Requirements

- Generate PVs at the IOC level that contains data and metadata (cycle ID)
- Accumulate consecutive acquisitions locally and upload to a collector service for post-mortem or on-demand analysis
- Store this data on permanent storage indexed by the cycle ID

Constraints

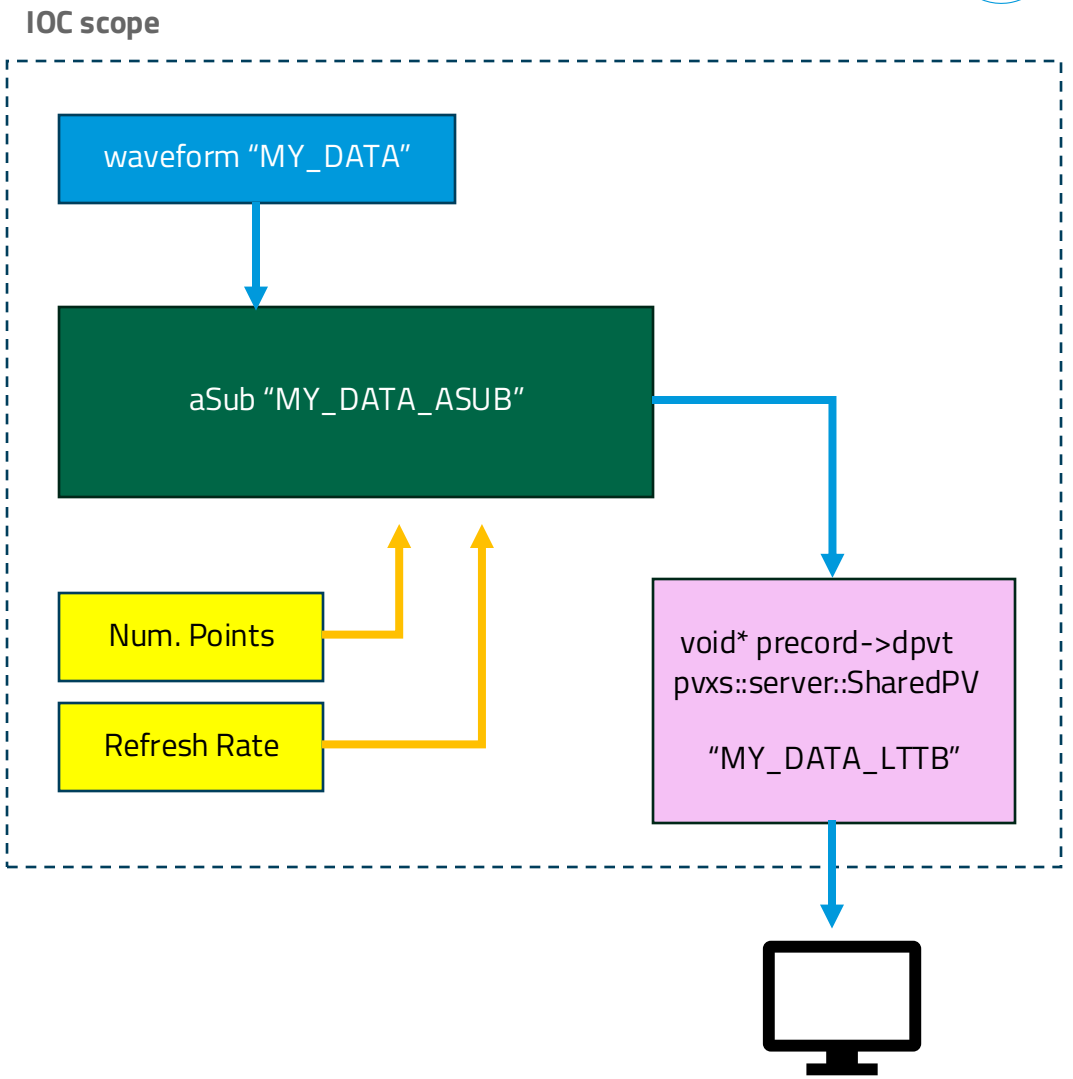
- Don’t overload the network (in fact, try to reduce the traffic)
- Don’t modify existing IOCs (as much as possible)

Off-topic – PVXS in production

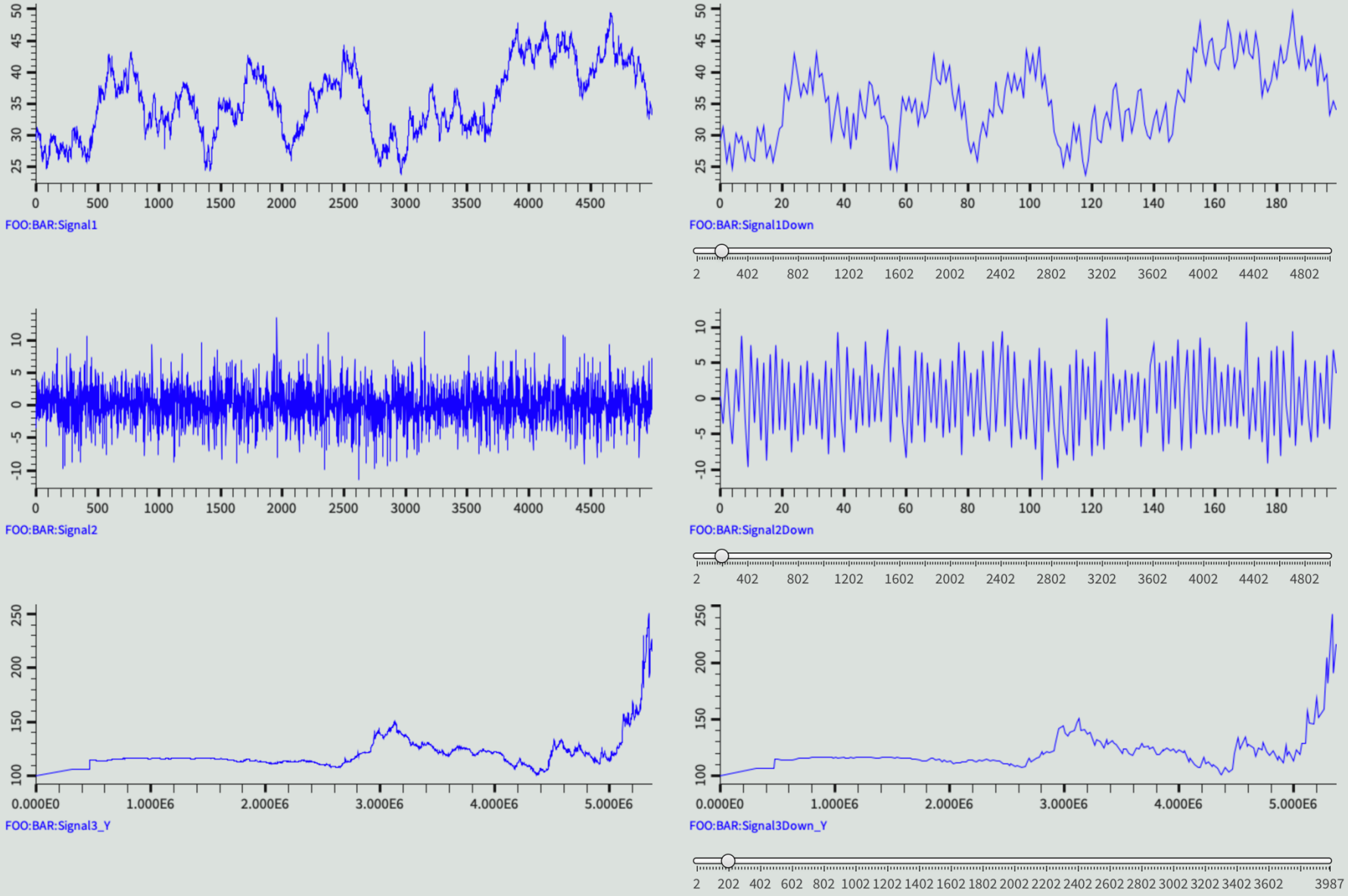
Down-sampling waveforms

- We have poor control of OPIs monitoring waveforms
- Archiver is also monitoring waveforms
- Huge traffic when running at 14 Hz

- (Partially) tackle the network congestion with down-sampled waveforms at the IOC level
- LTTB Algorithm
- Using asub record with pvxs to create a new PV with down-sampled array and throttled updates



LTTB Downsampling Algorithm Test



SDS EPICS Module

Architecture

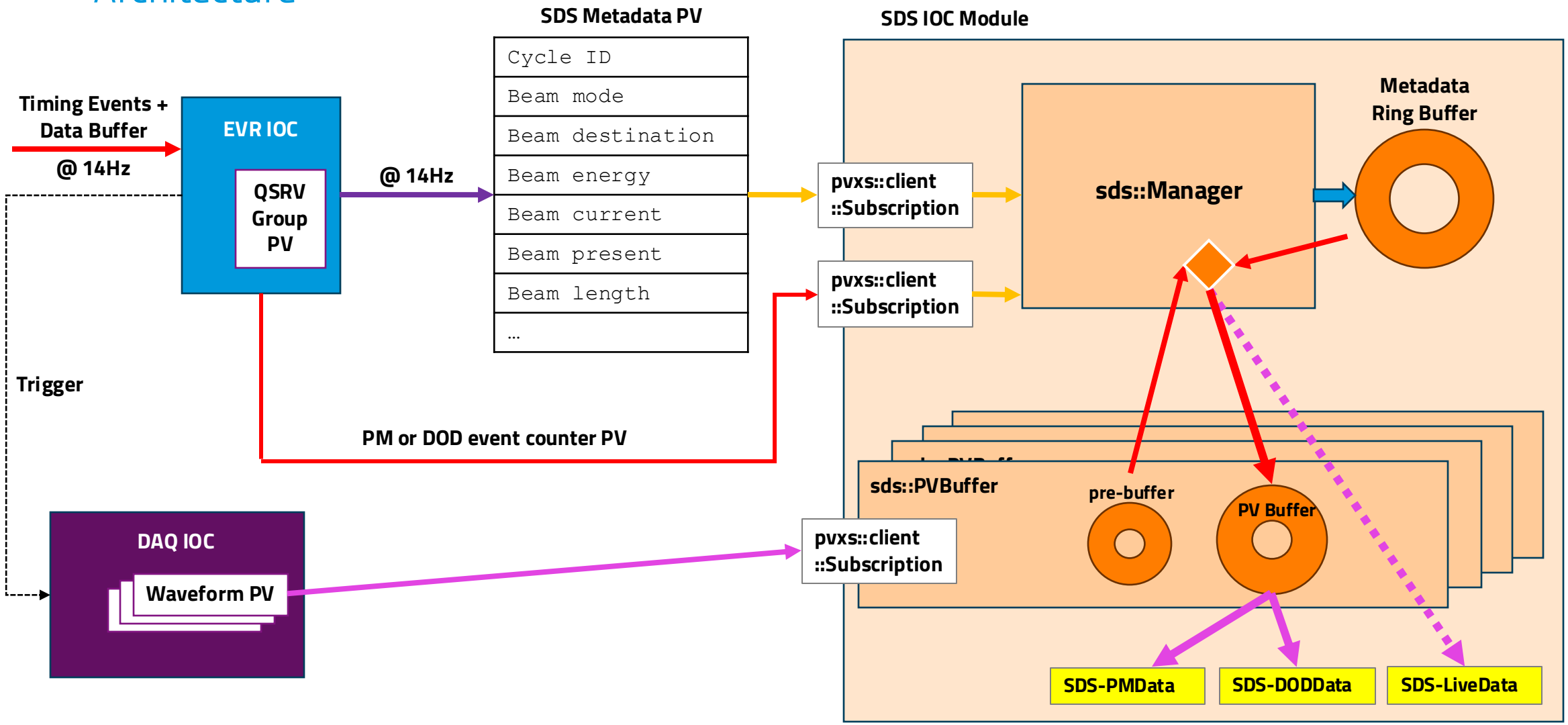
Disclaimer: this is still an on-going development!

- We have tried different architectures, we now have an optimal solution
  - EVR and DAQ are separate IOCs but need to exchange information
  - SDS can run as a separate IOC
  - Try not to assume anything, latencies may be higher than expected
  - Consider that everything is asynchronous, despite the timing interruptions
  - Some aspects still need to be better implemented

- Goal: ingest waveforms, match data with metadata, buffer it locally while also making it available to clients as Normative Types PV

SDS EPICS Module

Architecture



SDS EPICS Module

Generating the SDS Metadata PV with QSRV Groups

Very straightforward once you understand the info tags

- Processing chain with only one “trigger” field defined
- Defined as NTScaler with “value” equal the Cycle ID
- Copies all metadata from the timing data buffer into custom fields
- Updates at 14Hz even without beam

```
record(calc, "$(P)$(R=)#SDSMDataFetchDBuf") {
    field(DESC, "FWD process chain")
    field(CALC, "A+1")

    field(INPA,  "$(P)$(R=)#SDSMDataFetchDBuf NPP")
    field(VAL,   "0")
    field(INPB,  {pva:{pv:"$(F14HzCnt)",proc:"CPP",time:true}})
    field(TSE,   -2)

    info(Q:group, {
        "$(P)$(R=)SDSMetadata":{
            +id:"epics:nt/NTScalar:1.0",
            "": {+type:"meta", +channel:"TIME"},
            "cycleId":{+type:"meta", +channel:"TIME"},
            "sdsInfo":{+type:"meta", +channel:"TIME"},
        }
    })

    field(FLNK, "$(P)$(R=)#SDSMDataBStateIn")
}

#- Beam State
record(ai, "$(P)$(R=)#SDSMDataBStateIn") {
    field(INP,  {pva:{pv:"$(PEVR)BState",proc:false}})

    info(Q:group, {
        "$(P)$(R=)SDSMetadata":{
            "beamInfo.state":{+type:"plain", +channel:"VAL"},
        }
    })

    field(FLNK, "$(P)$(R=)#SDSMDataBDestIn")
}
```

SDS EPICS Module

Generating the SDS Metadata PV with QSRV Groups

```
LabS-ICS:SC-IOC-411:SDSMetadata
Server: 172.30.7.114:5075
Type:
    epics:nt/NTScalar:1.0
        structure record
            structure _options
                int queueSize
                boolean atomic
        alarm_t alarm
            int severity
            int status
            string message
        time_t timeStamp
            long secondsPastEpoch
            int nanoseconds
            int userTag
    long value
```

```
    structure beamInfo
        int curr
        double dest
        int energy
        int len
        double mode
        double present
        double state
    structure cycleId
        alarm_t alarm
            int severity
            int status
            string message
        time_t timeStamp
            long secondsPastEpoch
            int nanoseconds
            int userTag
        long value
```

```
    structure diagnostics
        alarm_t alarm
            int severity
            int status
            string message
        time_t timeStamp
            long secondsPastEpoch
            int nanoseconds
            int userTag
        double count
    structure sdsInfo
        alarm_t alarm
            int severity
            int status
            string message
        time_t timeStamp
            long secondsPastEpoch
            int nanoseconds
            int userTag
        structure buffer
            int idx
            int size
        long cycleId
        int evtCode
        double evtcount
```

SDS EPICS Module

pvxs::Value as the standard data container

The `pvxs::Value` class objects are used all over the code

- All ring-buffers are of the type `std::deque<pvxs::Value>`
- All message queues are of the type `pvxs::MPMCFIFO<pvxs::Value>`

Field names are hardcoded

- Ex. “cycleId.value”

```
Value top = TypeDef(TypeCode::Struct, {
    members::Int32("fldname"),
}).create();

top["fldname"] = 1;

fld = top["fldname"];
fld = 2;
```

Is analogous to the following pseudo code.

```
// pseudo-code
struct anon {
    int32_t fldname=0u;
};
void* top = new anon;

static_cast<anon*>(top)->fldname = 1;

void* fld = &static_cast<anon*>(top)->fldname;
static_cast<int32_t*>(fld) = 2;
```

SDS EPICS Module

How to create a custom structure in PVXS

```
pvxs::TypeDef SDSData::build() const {
    using namespace pvxs::members;

    auto time_t(pvxs::nt::TimeStamp{}.build());
    auto alarm_t = {
        Int32("severity"),
        Int32("status"),
        String("message"),
    };

    pvxs::TypeDef def(pvxs::TypeCode::Struct, "epics:nt/NTNDArray:1.0", {
        Union("value", {
                        BoolA("booleanValue"),
                        Int8A("byteValue"),
                        Int16A("shortValue"),
                        Int32A("intValue"),
                        Int64A("longValue"),
                        UInt8A("ubyteValue"),
                        UInt16A("ushortValue"),
                        UInt32A("uintValue"),
                        UInt64A("ulongValue"),
                        Float32A("floatValue"),
                        Float64A("doubleValue"),
        }),
        time_t.as("timeStamp"),
        Struct("alarm","alarm_t",alarm_t),

        StructA("dimension", "dimension_t", {
            Int32("size"),
            Int32("offset"),
            Int32("fullSize"),
            Int32("binning"),
            Bool("reverse"),
        }),

        Struct("arrayInfo","arrayInfo_t", {
            Float32("sampling"),
            Int32("size"),
            Int32("decimation")
        }),
```

```
        }),

        Struct("sdsInfo", "sdsInfo_t", {
            Int64("cycleId"),
            Float32("evtcount"),
            Int32("evtCode"),
            Struct("buffer", "buffer_t", {
                Int32("size"),
                Int32("idx")}),
                time_t.as("timeStamp"),
                Struct("alarm", "alarm_t", alarm_t)
            }),

        Struct("cycleId", "cycleId_t", {
            Int64("value"),
            Struct("alarm", "alarm_t", alarm_t),
            time_t.as("timeStamp")
        }),

        Struct("acqEvt", "acqInfo_t", {
            String("name"),
            String("evr"),
            Float32("delay"),
            Int32("code"),
            Struct("alarm", "alarm_t",alarm_t),
            time_t.as("timeStamp")
        }),

        Struct("diagnostics","diagnostics_t", {
            Int32("uniqueId"),
            // Int32("missedFrames"),
            // Int32("dupCycleId"),
            time_t.as("timeStamp")
        }),
    });

    return def;
}
```

SDS EPICS Module

The structure of the typical SDS PV (data + metadata)

```
epics:nt/NTNDArray:1.0
    union value
        boolean[] booleanValue
        byte[] byteValue
        short[] shortValue
        int[] intValue
        long[] longValue
        ubyte[] ubyteValue
        ushort[] ushortValue
        uint[] uintValue
        ulong[] ulongValue
        float[] floatValue
        double[] doubleValue
    time_t timeStamp
        long secondsPastEpoch
        int nanoseconds
        int userTag
    alarm_t alarm
        int severity
        int status
        string message
```

```
    dimension_t[] dimension
        dimension_t
            int size
            int offset
            int fullSize
            int binning
            boolean reverse
    arrayInfo_t arrayInfo
        float sampling
        int size
        int decimation
```

```
  beamInfo_t beamInfo
      float mode
      float state
      float present
      int len
      int energy
      int curr
      float dest
  sdsInfo_t sdsInfo
      long cycleId
      float evtcount
      int evtCode
      buffer_t buffer
          int size
          int idx
      time_t timeStamp
          long secondsPastEpoch
          int nanoseconds
          int userTag
      alarm_t alarm
          int severity
          int status
          string message
```

SDS EPICS Module

The structure of the typical SDS PV (data + metadata)

```
    string message
cycleId_t cycleId
    long value
    alarm_t alarm
        int severity
        int status
        string message
    time_t timeStamp
        long secondsPastEpoch
        int nanoseconds
        int userTag
```

```
acqInfo_t acqEvt
    string name
    string evr
    float delay
    int code
    alarm_t alarm
        int severity
        int status
        string message
    time_t timeStamp
        long secondsPastEpoch
        int nanoseconds
        int userTag
diagnostics_t diagnostics
    int uniqueId
    time_t timeStamp
        long secondsPastEpoch
        int nanoseconds
        int userTag
```

SDS EPICS Module

Pushing data out to the PVXS PVA server

- `sds::PVBuffer` objects are created for each data channel
  - Holds the circular buffers of `pvxs::Value`
  - Holds three (3) `pvxs::server::SharedPV` objects that effectively create new PVs directly from C++

- New PV names are derived from original data source PV
  - SDS-LiveData
  - SDS-PMData
  - SDS-DODData

```
m_ArrayDataLivePV = pvxs::server::SharedPV::buildReadonly();
std::string pvname = m_Name + "SDS-LiveData";
pvxs::Value livedatainit = sds::SDSData{}.create().cloneEmpty();
pvxs::ioc::server().addPV(pvname, m_ArrayDataLivePV);
m_ArrayDataLivePV.open(livedatainit);

m_ArrayDataPMPV = pvxs::server::SharedPV::buildReadonly();
pvname = m_Name + "SDS-PMData";
pvxs::Value pmdatainit = m_ArrayDataLivePV.fetch().cloneEmpty();
pvxs::ioc::server().addPV(pvname, m_ArrayDataPMPV);
m_ArrayDataPMPV.open(pmdatainit);

m_ArrayDataDODPV = pvxs::server::SharedPV::buildReadonly();
pvname = m_Name + "SDS-DODData";
pvxs::Value doddatainit = m_ArrayDataLivePV.fetch().cloneEmpty();
pvxs::ioc::server().addPV(pvname, m_ArrayDataDODPV);
m_ArrayDataDODPV.open(doddatainit);
```

SDS EPICS Module

Pushing data out to the PVXS PVA server

- SDS-LiveData [`pvxs::server::SharedPV`]
  - Copy of the `pvxs::Value` is pushed to SharedPV which triggers subscription updates
  - Use with caution!

- SDS-DODData [`pvxs::server::SharedPV`]
  - Acts upon a timing event arrival (Data-on-Demand event)
  - Updates the SharedPV with the circular buffer values with an interval of 1 second

- SDS-PMData [`pvxs::server::SharedPV`]
  - Acts upon a timing event arrival (Post-mortem event)
  - Updates the SharedPV with the circular buffer values with an interval of 1 second

SDS EPICS Module

How to integrate on your IOC ?

- In the EVR IOC: simply load the database that creates the QSRV Group

```
# ------------------------------------------------------------------------
# Configuration of the SDSMetadata (EVR IOC)
# ------------------------------------------------------------------------
dbLoadRecords("$(sdsioc_DB)/metadataGroup.template", "P=SDS:, PEVR=EVR:, F14HzCnt=EVR:F14HzCnt-I")
```

SDS EPICS Module

How to integrate on your IOC ?

Dependencies:

- EPICS Base
- pvxs

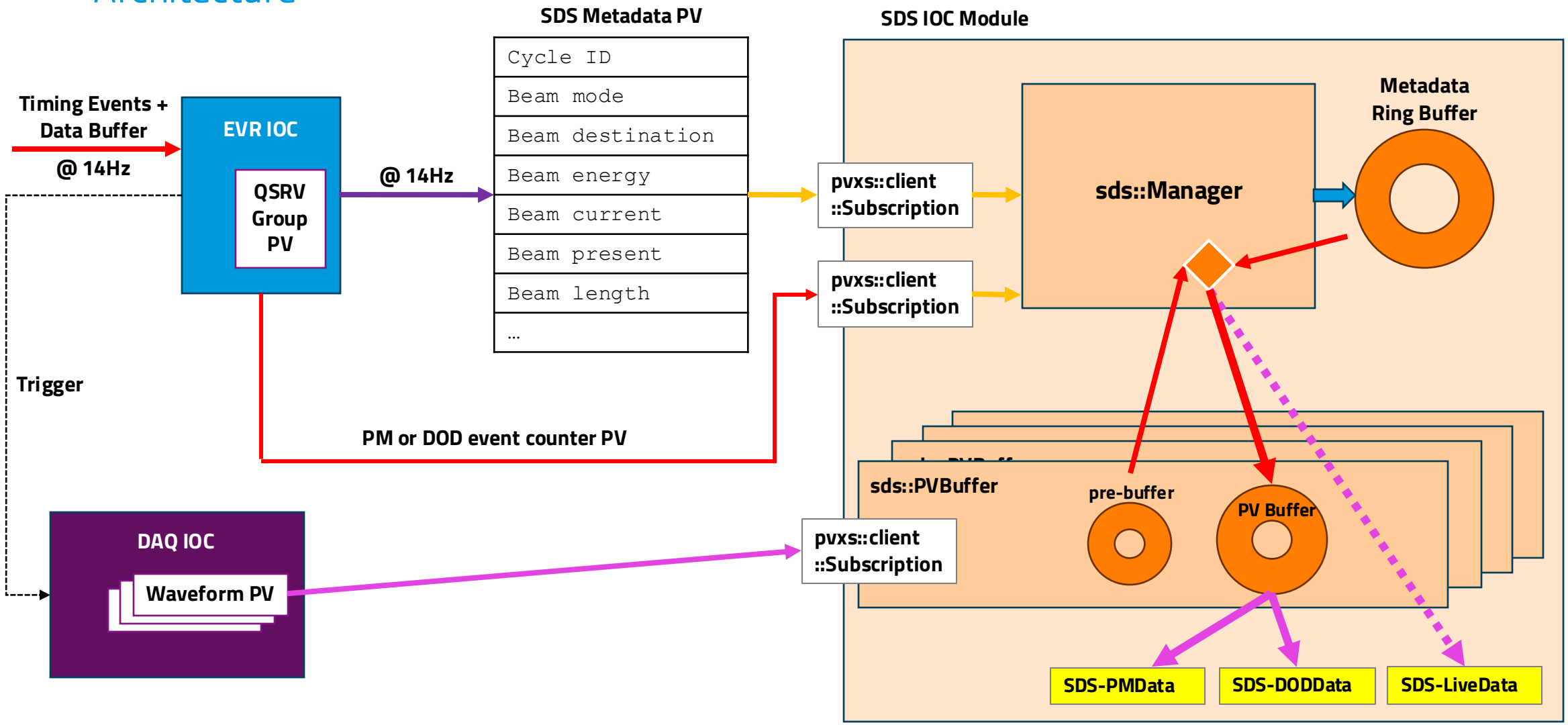
Modes:

- Inside the DAQ IOC
- Stand-alone IOC running in the same host
- ~~Stand-alone IOC running in another host~~

```
# ------------------------------------------------------------------------
# SDS Configuration
# ------------------------------------------------------------------------
epicsEnvSet("SDSPREFIX", "FOO:BAR:SDS")
sdsConfigure("$(SDSPREFIX):","SDS","FOO:BAR:SDSMetadata","$(EVR_NAME)PMortemCnt-I", "$(EVR_NAME)DoDCnt-I")

#- ------------------------------------------------------------------------
#- SDS Waveform Capture
#- ------------------------------------------------------------------------
#- int sdsCreateChannel(const char *pvname, int prebufsize, int postbufsize, const char *sampfreqpv, const char* decimationpv,
#                       const char *evrname, const char *evrpulser, const char *evroutput, const char *evrevtnumpv,
#                       const char *evrevtdelaypv, const char *evrevtcntpv)

epicsEnvSet("EVR","EVR")
epicsEnvSet("EVR_PUL","DlyGen-6")
epicsEnvSet("EVR_OUT","RX17")
epicsEnvSet("EVR_EVTPV","EVR:DlyGen-6-Evt-Trig0-SP")
epicsEnvSet("EVR_DLYPV","EVR:DlyGen-6-Delay-RB")
epicsEnvSet("EVR_CNTPV","EVR:BIAcqStCnt-I")
epicsEnvSet("SAMPRATEPV","$(DAQDEV):SamplingFrequencyR")
epicsEnvSet("DECIMATIONPV","$(DAQDEV):Decimation-RB")

sdsCreateChannel("$(DAQDEV):CH1-TRC1-ArrayData" "15" "1" "$(SAMPRATEPV)" "$(DECIMATIONPV)" "$(EVR)" "$(EVR_PUL)" "$(EVR_OUT)"
        "$(EVR_EVTPV)" "$(EVR_DLYPV)" "$(EVR_CNTPV)")

sdsCreateChannel("$(DAQDEV):CH2-TRC1-ArrayData" "15" "1" "$(SAMPRATEPV)" "$(DECIMATIONPV)" "$(EVR)" "$(EVR_PUL)" "$(EVR_OUT)"
        "$(EVR_EVTPV)" "$(EVR_DLYPV)" "$(EVR_CNTPV)")
```

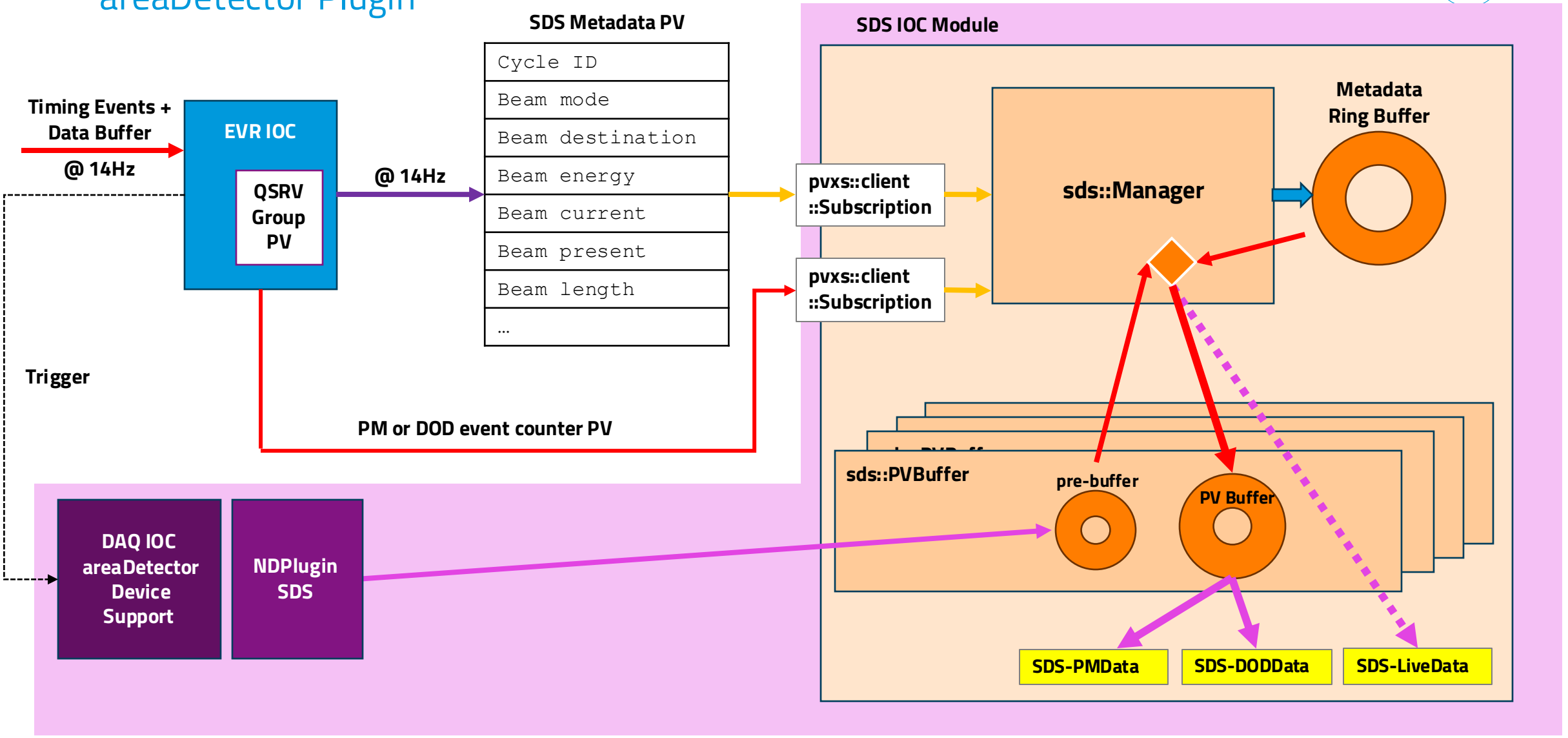
SDS EPICS Module

Architecture



SDS EPICS Module

areaDetector Plugin

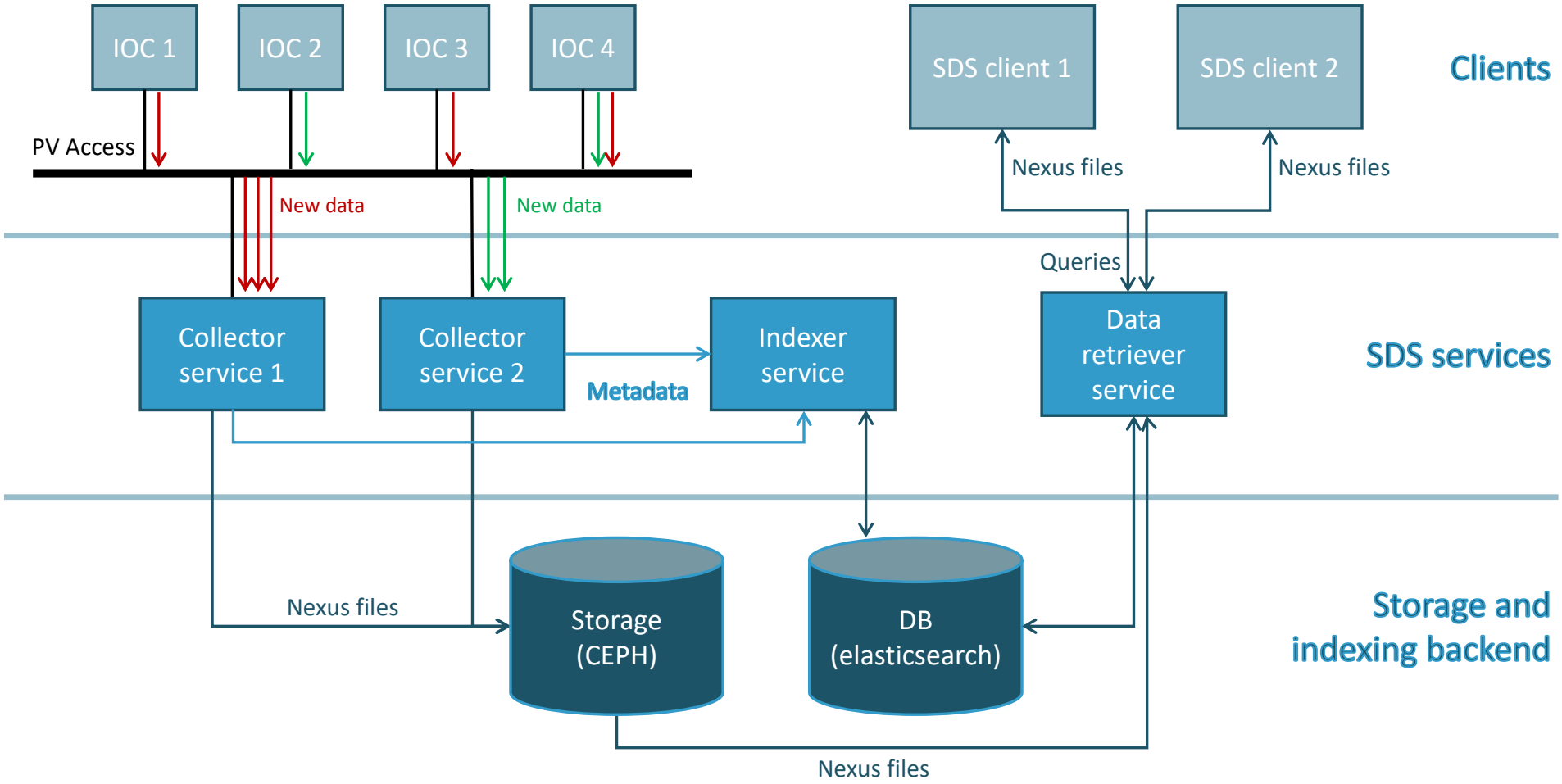


5

SDS Collector

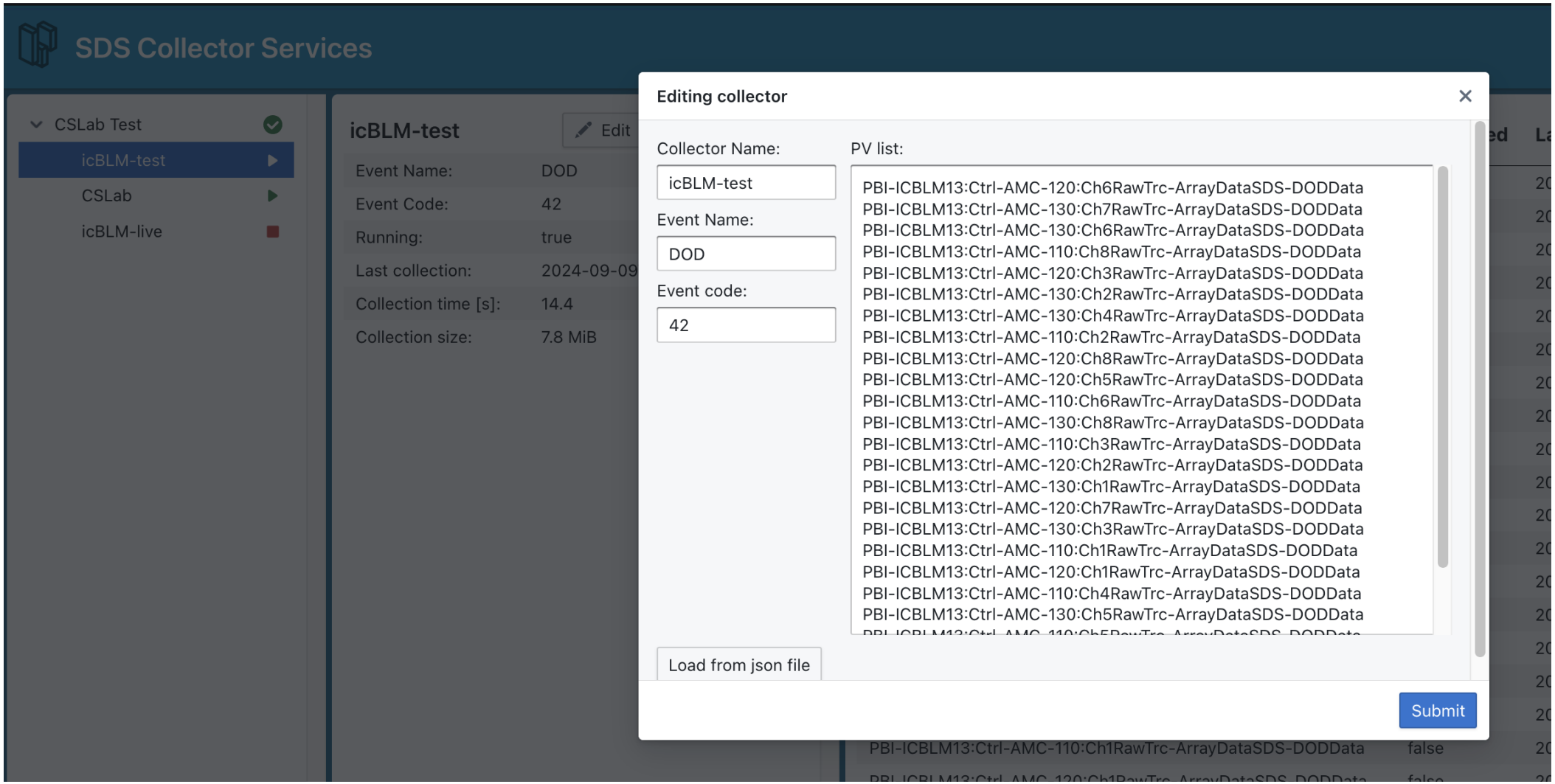
SDS Collector

Architecture



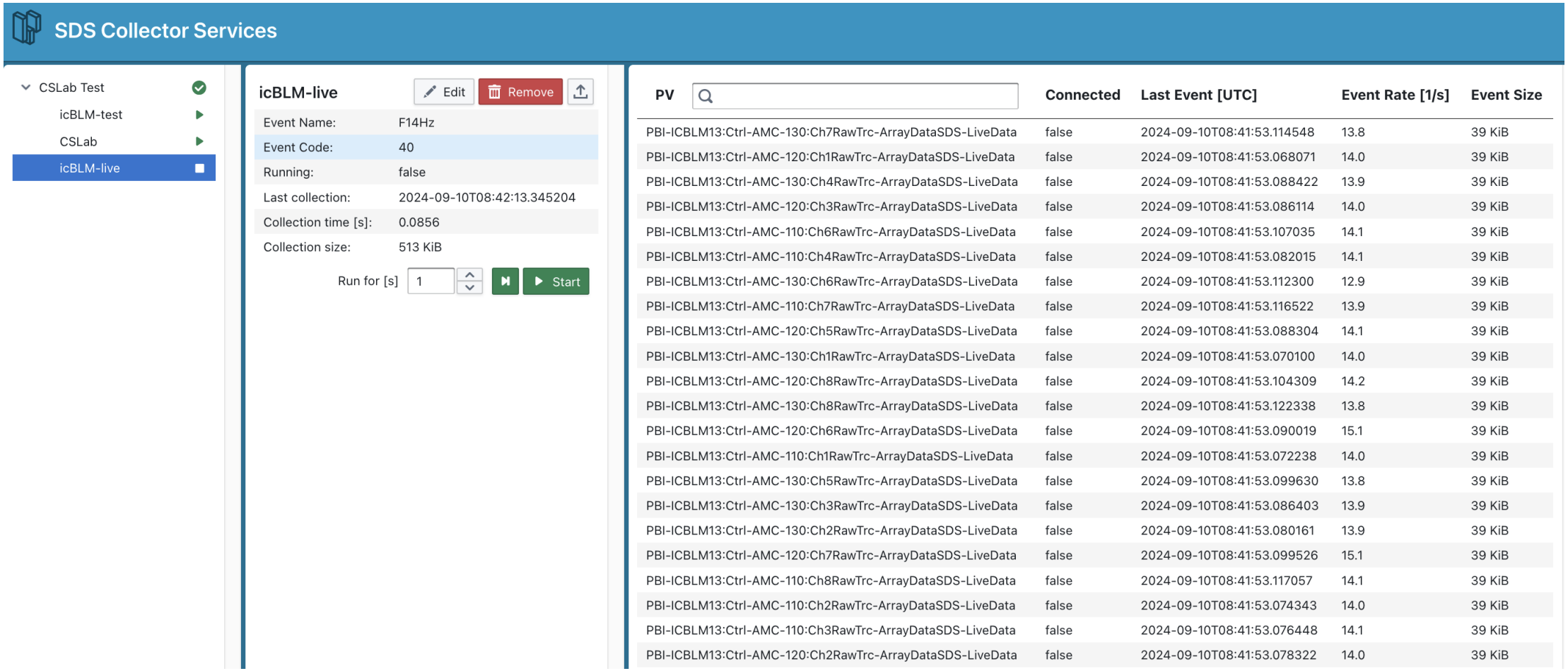
SDS Collector

Web interface to configure new collectors



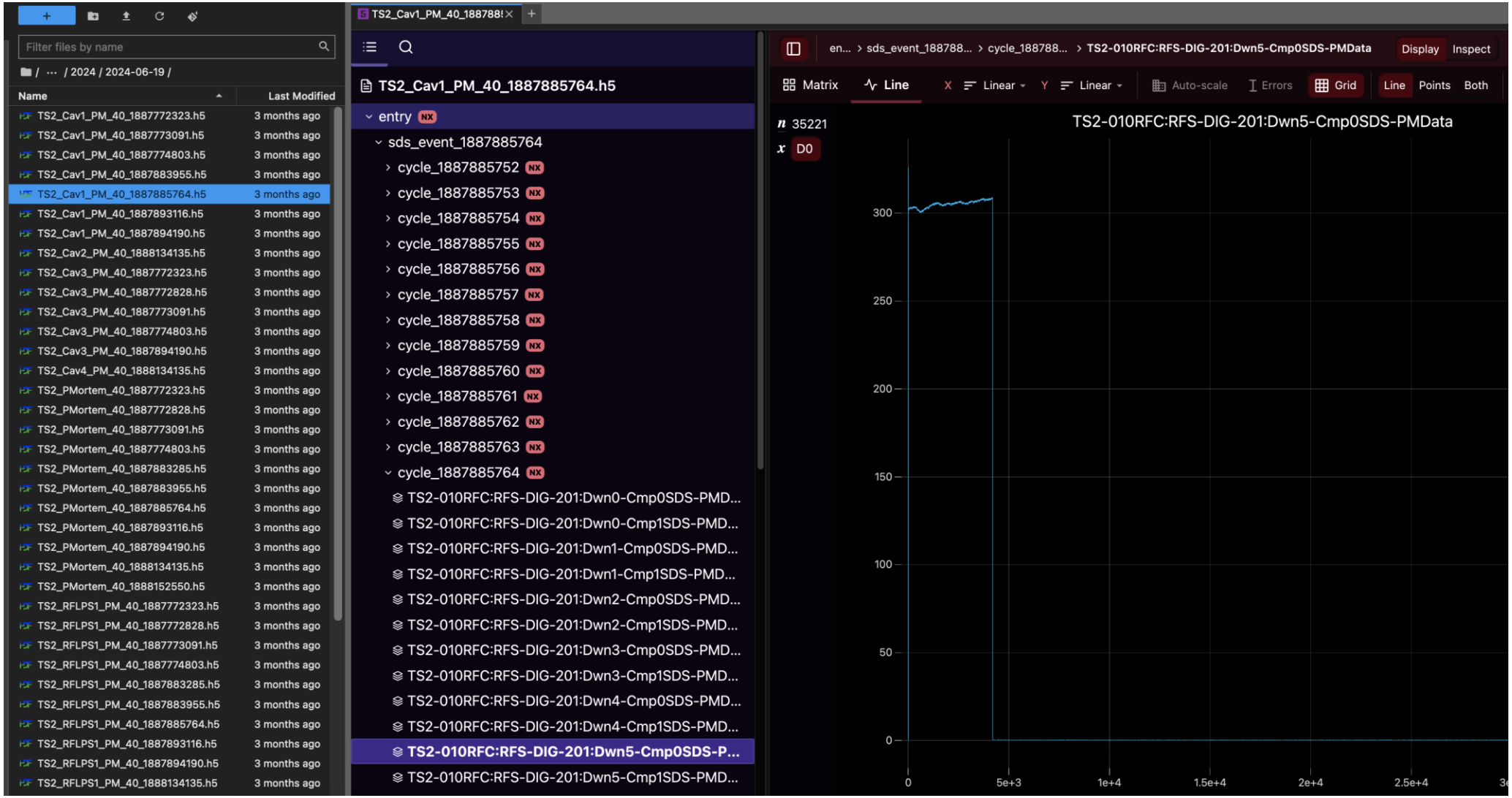
SDS Collector

Configuring the stream acquisition



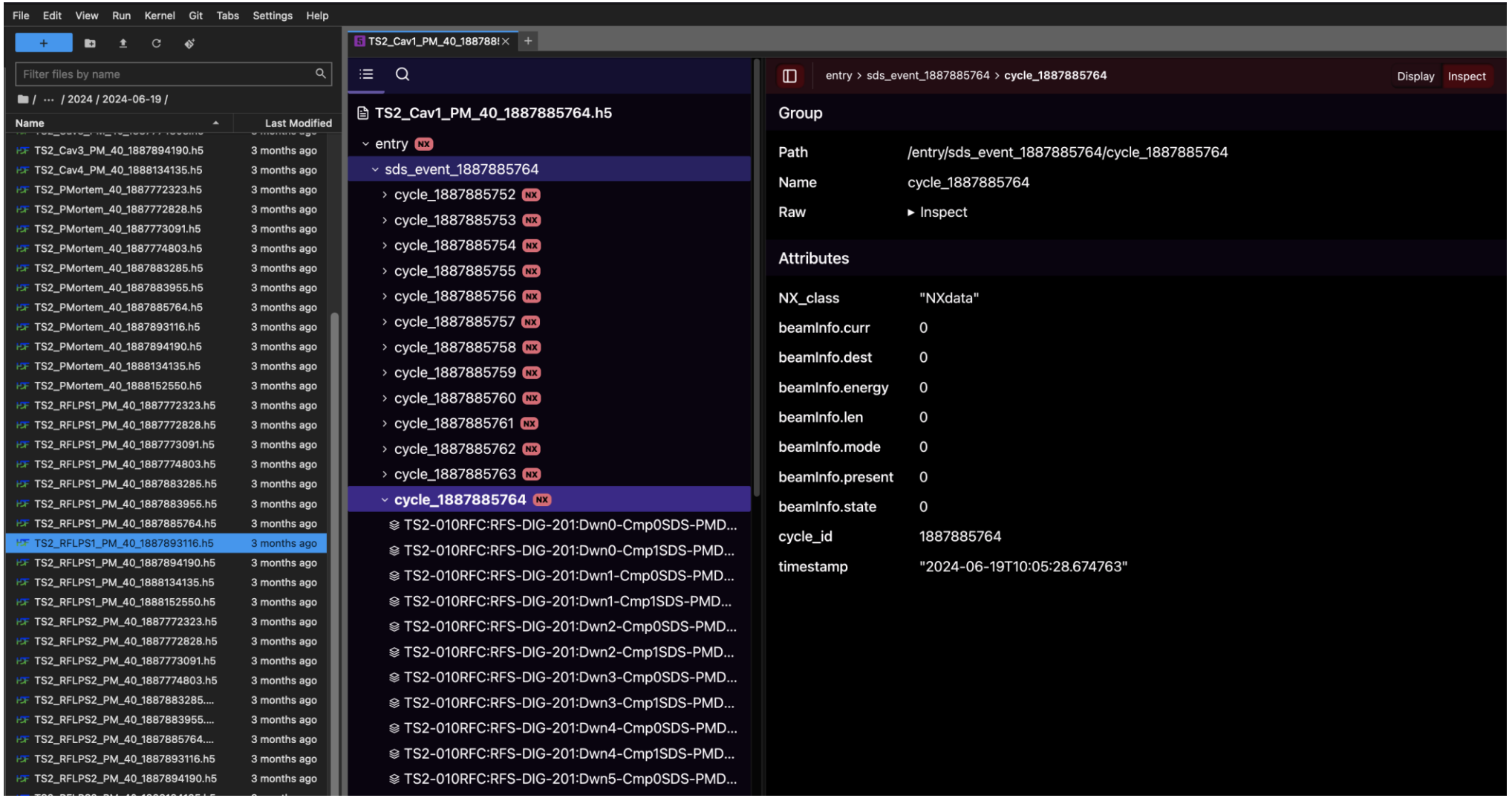
SDS Collector

NExUS files available in Jupyterhub



SDS Collector

NExUS file metadata collection



6

Conclusions

Conclusions

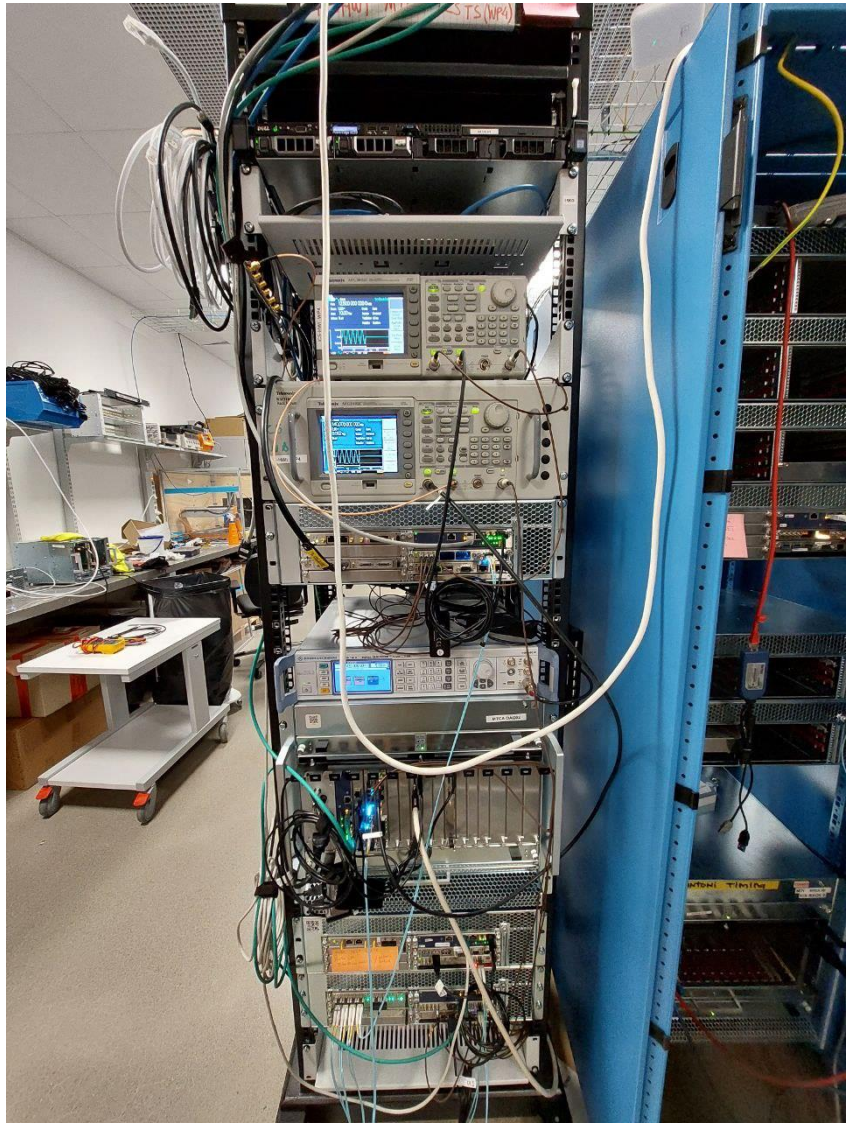
Validation Tests

Test setup in the lab with different MTCA systems:

- MTCA 1 – Generic acquisition with Struck SIS8300 (FC, WS) and IFC1410 (EMU)
- MTCA 2 – LLRF and RFLPS-FIM
- MTCA 3 – BCM (SIS8300)
- MTCA 4 – Event Master (superCycle)
- MTCA 5 – BPM (4x Struck SIS8300) *

Testing with LLRF and RFLPS in Test Stand 2

- Successful demonstration for RF stakeholders
- Not connected to the Timing Network



Conclusions

Performance Impact

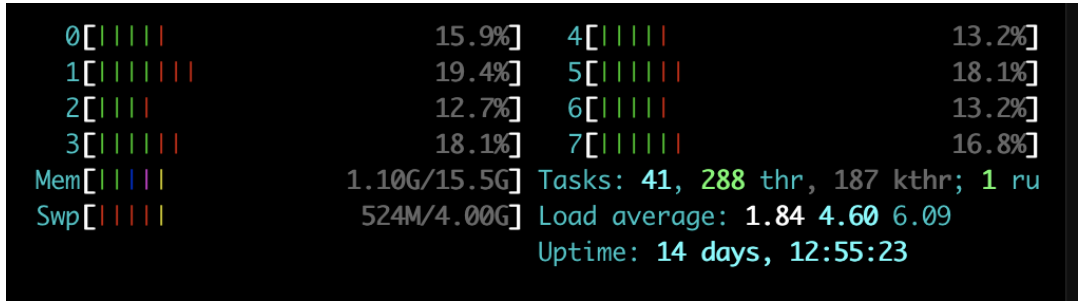
Performance is relative

- Depends on the system (number of channels, sampling rate, number of points)
- Depends on the hardware

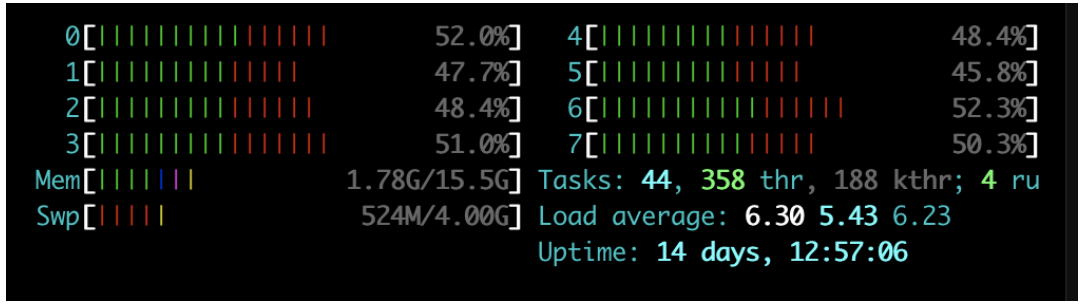
Obvioulsly the current scenario is not optimal

- Arrays are being copied too many times along the process
- PVA links within the same host
- NDPlugin performance impact is lower

BCM IOC with 20 channels 400k samples (32-bits) running acquisitions at 14 Hz



BCM IOC with 20 channels 400k samples (32-bits) running acquisitions at 14 Hz + SDSIOC monitoring all 20 waveforms



Conclusions

Performance Impact

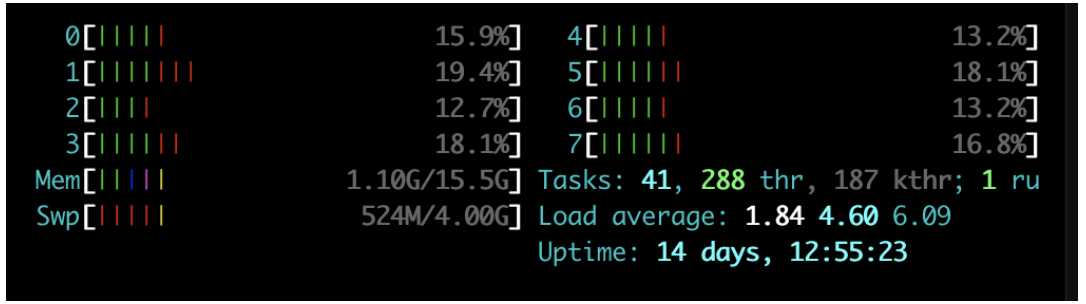
Scenario with separate IOC did not degrade the performance of the majority of the systems

- BCM
- WS, EMU
- LLRF
- RFLPS-FIM

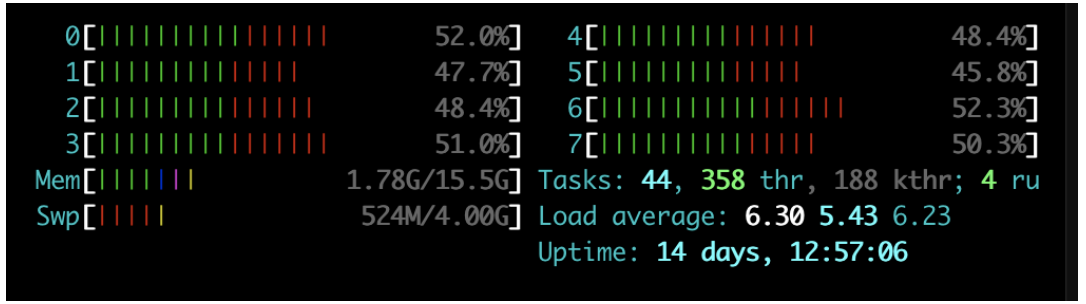
The heaviest system is the BPM with four digitizers

- Using the NDPlugin made the SDS IOC feasible

BCM IOC with 20 channels 400k samples (32-bits) running acquisitions at 14 Hz



BCM IOC with 20 channels 400k samples (32-bits) running acquisitions at 14 Hz + SDSIOC monitoring all 20 waveforms



Conclusions

Next Steps

- Finish the development of the SDS IOC module and release the first version
- Deploy the SDS IOC module for LLRF, RFLPS and some PBI systems before the next commissioning
- Test the SDS Collector at a larger scale
- Re-evaluate the technical solutions and explore other options to optimize performance:
  - Array management should be done at the device support layer

- Data analysis framework: work in progress
  - Systems Engineering document to describe the workflow
  - Improve current solutions for data retrieval
  - Enable DASK as the Python library for parallel and distributed computing

- The dream: “Given these PVs from this period to that period, calculate something and show me the results”

Thank you / Obrigado / Tack