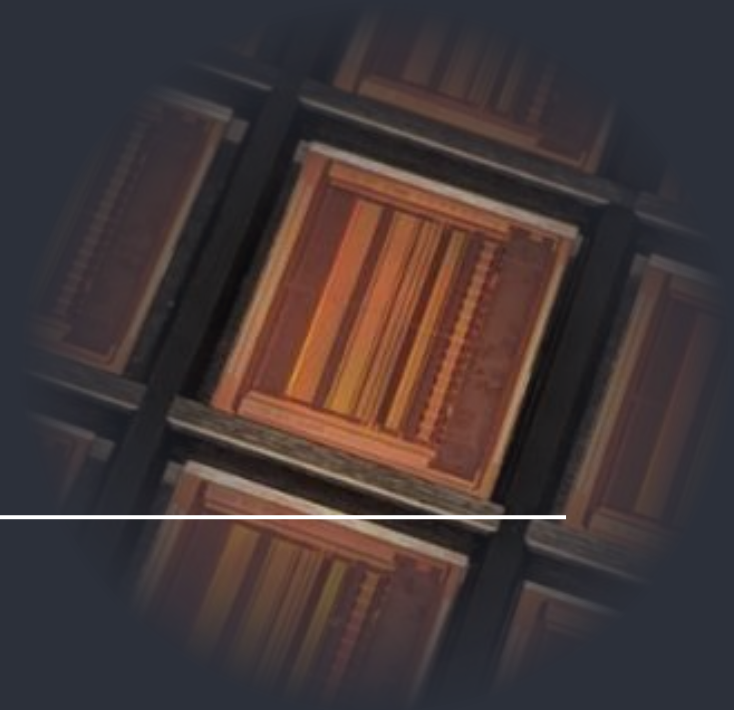
Rogue: Back-End Integration



Omar Moreno, Ryan Herbst, Larry Ruckman, Ben Reese, J.J. Russell

What Is Rogue?

Rogue is an open source C++/Python hybrid architecture platform that facilitates the communication with hardware modules

- Most of the C++ base classes expose their methods via Python
  - Allows for rapid prototyping in python with the ability to use underlying C++ for performance
- For most use cases, development is done in Python

Low level C++ interfaces

- Memory API - Interfaces with hardware register space
- Stream API - Interface for bulk data movement and asynchronous messages

High level Python interfaces (PyRogue)

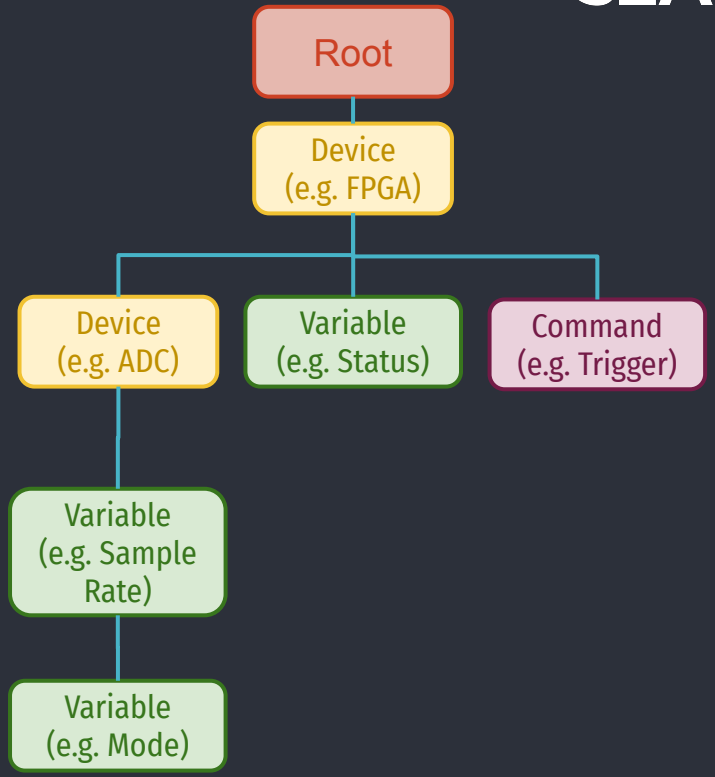
- Uses a hierarchical tree structure to organize hardware components in a system
- Layered design docouples tree from communication interface used to access hardware

Open source and easily deployable

- https://github.com/slaclab/rogue
- Conda recipes and Docker images are available

PyRogue Management Tree

- Top level API for writing Rogue applications which provides a mechanism for organizing elements into a tree
- Tree describes configuration and status registers that exist in connected hardware, such as FPGA firmware, ADCs, DACs and ASICs.
- Tree Node Types:
  - Devices
  - Variables
  - Commands
  - Root



Device

- Containers for logical groups of Variables, Commands, and other Devices
- Represent an organizational unit of hardware or other logical element.
- Examples:
  - An I2C or Serial Peripheral Interface (SPI) ADC with a set of configuration registers
  - An FPGA with many registers, divided into sub-Devices by functionality
  - A sensor ASIC with a set of configuration registers
- Each Device is instantiated with an address offset relative to its parent
- Each Device “owns” a segment of remote address space in hardware
- A special Root Device must be instantiated at the base of every Device Tree
  - Has specific APIs to manage the tree and expose it to external systems

Variables

- Variables describe a discrete values in the system
- Mostly they are used to describe FPGA, ADC, DAC or ASIC registers.
- RemoteVariable fields allow for a wide range of register types to be described
  - Name – Every node needs one
  - Mode – Read/Write, Read-only, Write-only
  - Address offset – Byte address offset relative to parent container
  - Bit size – Number of bits in register (common to have weird sizes)
  - Bit offset – Bit shift required within byte address
  - Data Model – Unsigned int, signed int, endianness, float, string

- LinkVariables allow for creation of Variables that are transformations of other Variables
  - Simply define a set of functions to convert to and/or from and number of dependent Variables
  - Example: Convert a raw ADC value to Volts, temperature, or whatever unit it ultimately represents

Commands

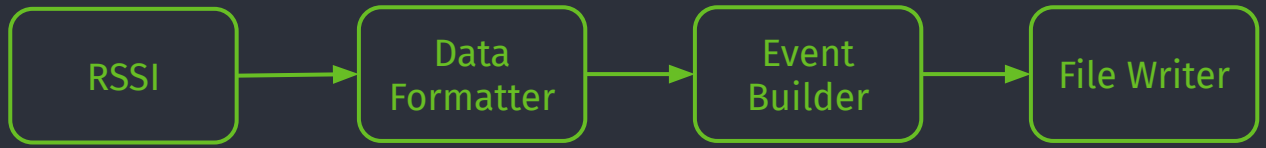
- A Command is essentially a wrapper around a Python function and is used to encapsulate common actions
- RemoteCommand
  - Simple actions on a single hardware register
  - Supports a number of optional arguments
- LocalCommand
  - Perform a sequence of operations within software
  - Can include sequences of variable writes or other Command executions
  - Ex: Complex initialization sequence across several Devices

Memory API

- Used to build memory access bridges to custom hardware while keeping a local mirror of register sets
- Can be used to build abstractions for any number of memory busses with arbitrary word alignments, address sizes and access sizes
- Devices in the Device Tree use the Memory API to reach hardware registers
- Several useful memory bridges are included with Rogue:
  - For Zynq SoCs, Rogue includes a Memory bridge for accessing FPGA registers on the AXI-bus
  - VCS RTL firmware simulation bridge
  - Local memory

Stream API

- Used to move bulk data between rogue processing elements and between hardware and software
- Can describe complex data processing pipeline, or simply direct data to a file
- Cascadable nature of API allows processing to be added to data chain by inserting new modules anywhere in the chain.
- Stream modules consist of two main interfaces
  - Sender (Stream Master) which request the frame and sends the data to other modules in the chain
  - Receiver (Stream Slave) which is used to process the data in a frame



Reliable SLAC Streaming Protocol

EPICS

EPICS support is provided via the python library P4P: https://mdavidsaver.github.io/p4p/overview.html

- P4P is a wrapper around PVAccess, a high-performance network communication protocol meant to succeed EPICS Channel Access
  - Supports Get, Put, Monitor and RPC operations

When enabled, a Process Variable (PV) is auto-generated for every variable in the pyrogue tree

- Commands are handled using Remote Procedure Calls
- Variables can be filtered or renamed if desired

Back-end Integration

In addition to serving as a stand-alone data acquisition solution, Rogue has been successfully integrated into several back-ends including

- CEBAF Online Data Acquisition (CODA)
  - C++/Java framework developed by Jefferson Lab for implementing data acquisition solutions at large (and small) scale
- EUDAQ
  - C++ data acquisition framework designed to be modular and cross-platform
- Psdaq
  - C++ data acquisition framework used by LCLS-II
- Observatory Control System (OCS)
  - Python based distributed control system for astronomical observatories

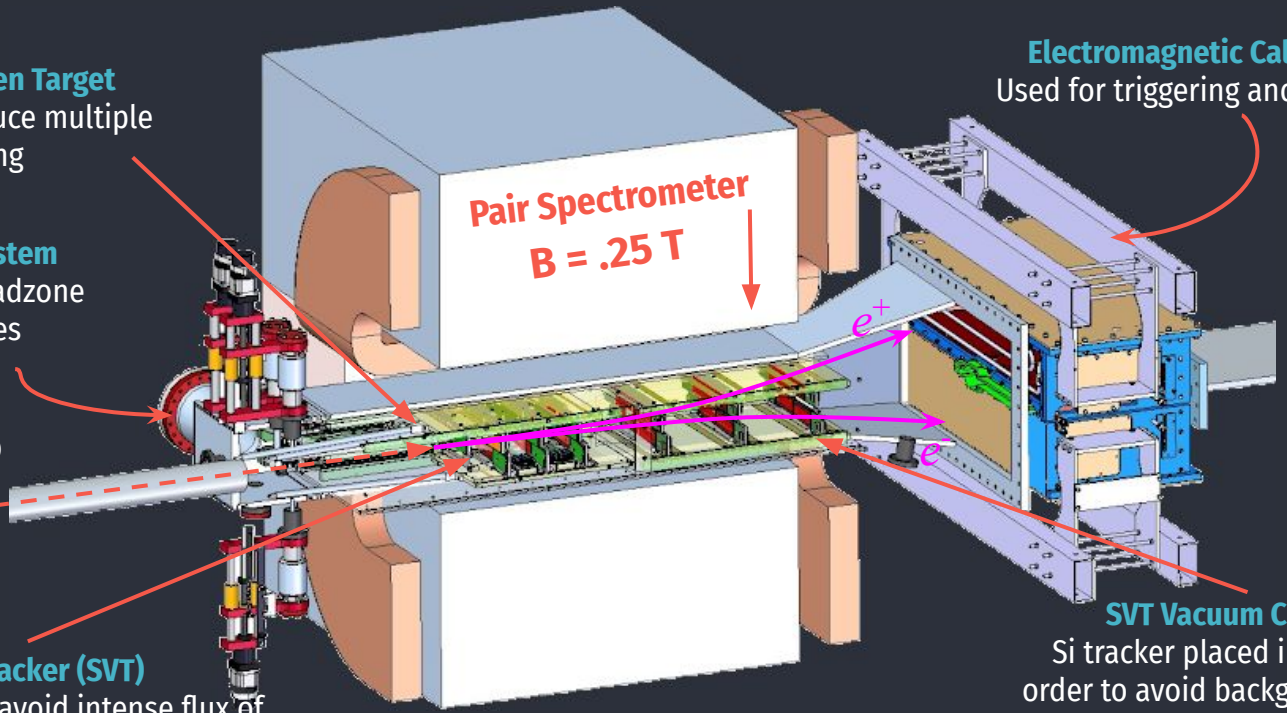
The Heavy Photon Search Experiment

~10^{-3} X_0 Tungsten Target
Thin target to reduce multiple scattering

Linear Shift Motion System
Allows adjustment of deadzone between SVT volumes

High intensity e^- beam
Courtesy of CEBAF @ JLab

Silicon Vertex Tracker (SVT)
Split into two volumes to avoid intense flux of scattered beam electrons. Used for precise momentum and vertex determination



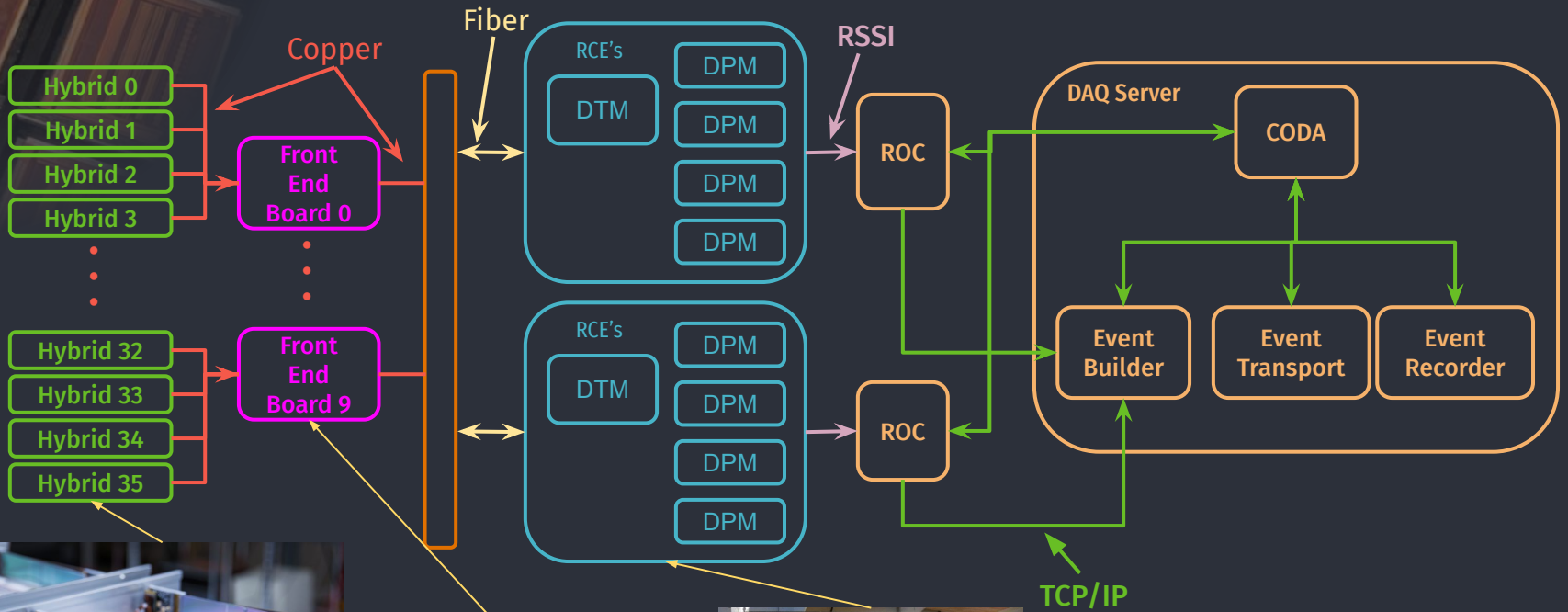
Electromagnetic Calorimeter
Used for triggering and particle ID

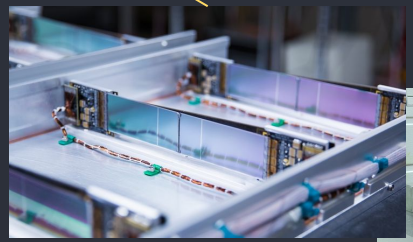
SVT Vacuum Chamber
Si tracker placed in vacuum in order to avoid backgrounds due to beam-gas interactions

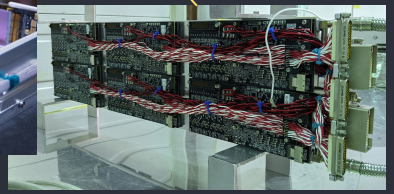
SVT + ECal DAQ capable of 50 kHz

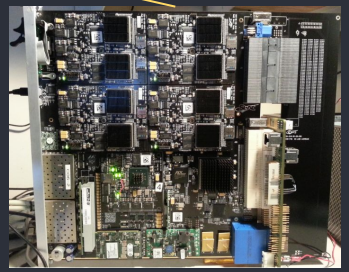
Installed within the Hall B alcove at Jefferson Lab upstream of the CLAS12 detector

Tracker Data Flow









SLAC TID TECHNOLOGY INNOVATION DIRECTORATE

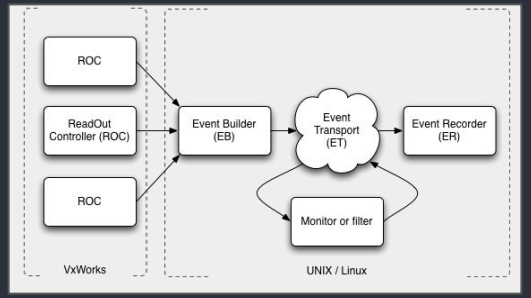
CODA Integration

The Readout Controller (ROC) app is responsible for collecting data from the front-ends and passing it to the event builder

- Each ROC will have an associated readout list which is a C library that defines how the ROC should be read out and what needs to be done during state transitions

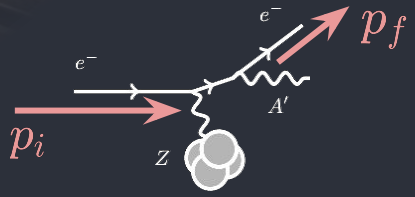
The tracker readout list instantiated an RSSI client which received the data from the RCE's, unbatched it and copied the events into CODA provided memory

State transitions were handled in a similar way except TCP/IP was used as the transport protocol



The Light Dark Matter eXperiment (LDMX)

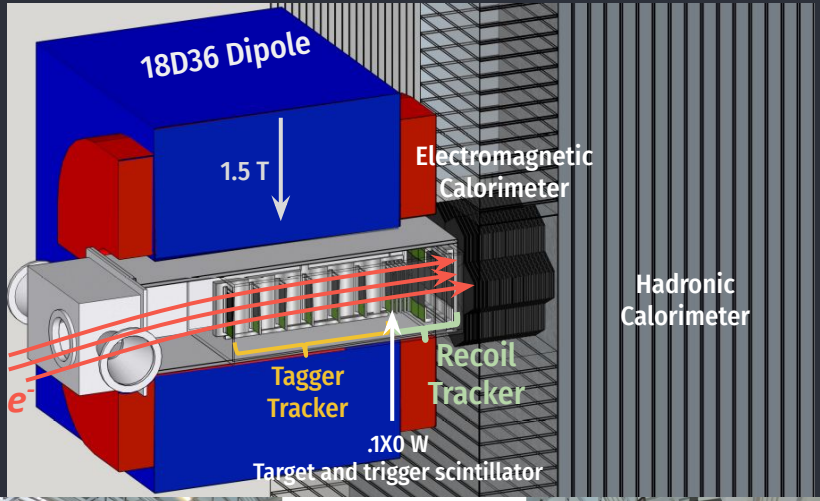
LDMX is a SLAC based experiment that aims to decisively test a variety of dark matter scenarios in the MeV-GeV mass range.

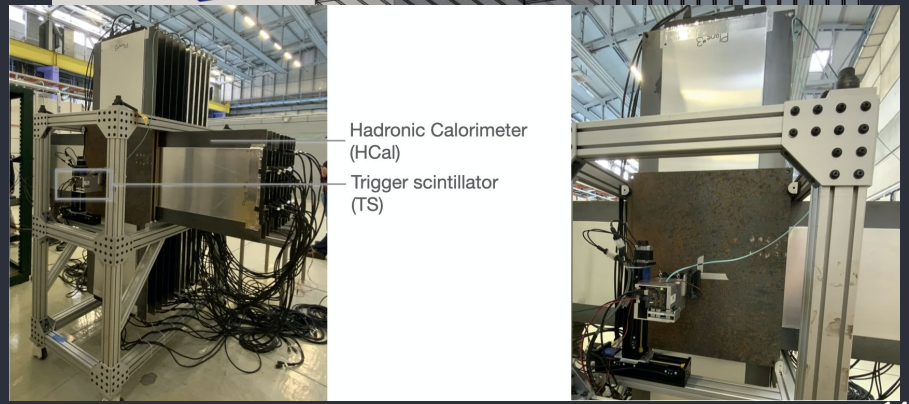


$$p_{\text{miss}} = p_i - p_f$$

Expected to run in FY27 but test beams will be used to understand several aspects of the detector

- First test beam took place in the spring of 2022 at CERN and used a simplified version of the Hadronic calorimeter and Trigger Scintillator
- Next test beam will be at SLAC in spring of 2024

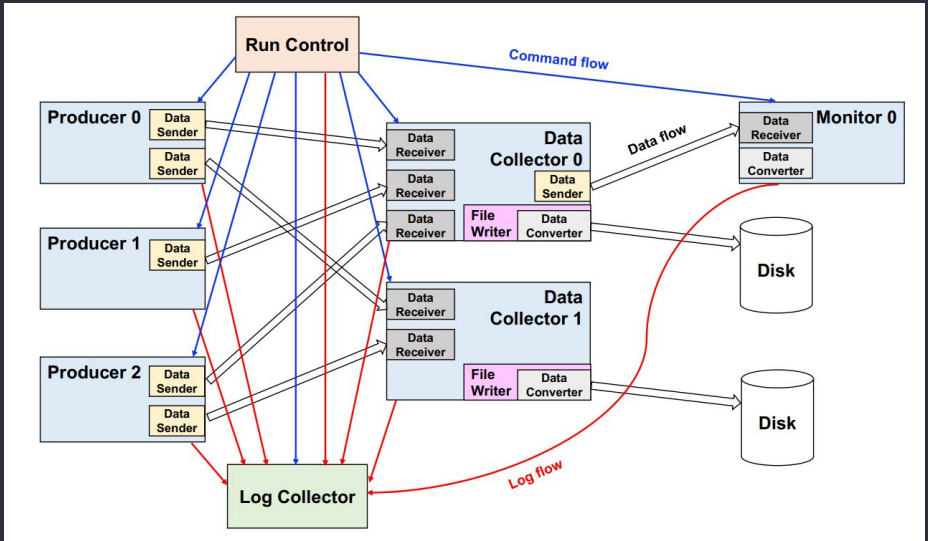




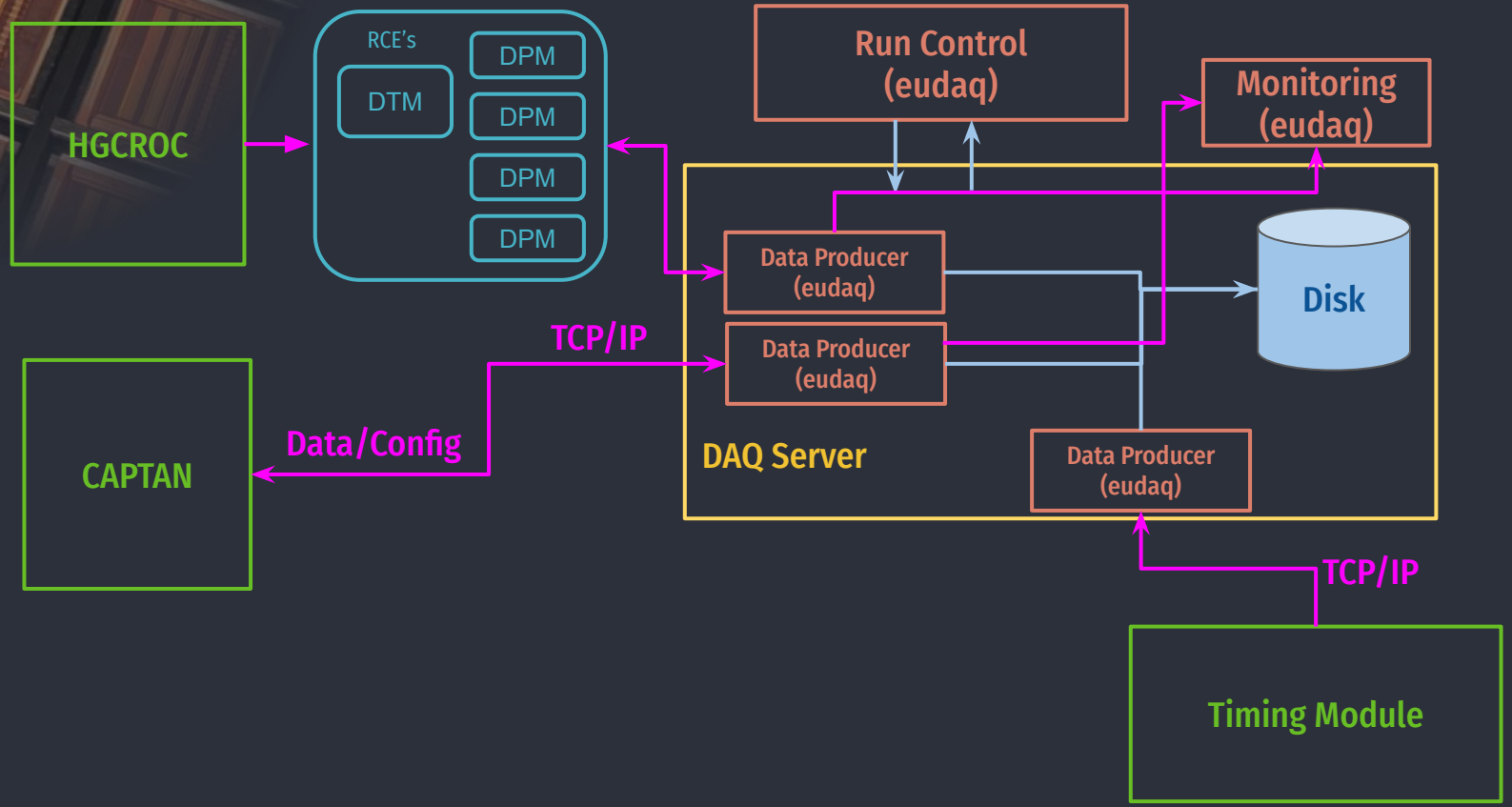
EUDAQ Overview

A central Run Control manages different processes via TCP/IP including

- Producers - process used to receive data, initialize, configure and start/stop hardware
- Data Collector - Receives data from all producers and does event building and data writing
- Log Collector - Compiles all logs and displays them to the user
- Monitoring - Reads data and creates online plots



Data Flow


HGCROC
RCE's
DTM
DPM
DPM
DPM
DPM
Run Control
(eudaq)
Monitoring
(eudaq)
Data Producer
(eudaq)
Data Producer
(eudaq)
Disk
DAQ Server
Data Producer
(eudaq)
TCP/IP
CAPTAN
Data/Config
TCP/IP
Timing Module

Rogue Integration

The stream API is used to establish a TCP/IP bridge between an RCE and the DAQ server

- The stream is extracted using the stream API within an eudaq producer and repackaged into a eudaq event
- The repackaged event is shipped to the data collector which writes the events to disk and sends it to the connected ROOT based monitoring app

The run control communicates state transitions to the eudaq producers, data collectors and monitors via tcp/ip

- The stream API was used to receive the state transitions within a producer and ship them off to the RCE server using the same bridge that was established to stream data
- The RCE server then listen for the specific transition string and would take actions depending on what state it was in

```
void RogueTcpClientProducer::DoConfigure() {

  // Get the configuration
  auto conf{GetConfiguration()};

  // Get the path to the output file
  output_path_ = conf->Get("OUTPUT_PATH", ".");

  // Get the file prefix
  file_prefix_ = conf->Get("ROGUE_FILE_PATTERN", "test");

  // Build the file name
  auto output_file{output_path_ + "/" + file_prefix_ + "_" + std::to_string(GetRunNumber()) + ".dat"};

  // First, make sure an existing file isn't open.
  if (writer_->isOpen()) writer_->close();

  // Open a file to write the stream
  writer_->open(output_file);

  EUDAQ_INFO("Writing rogue stream to " + output_file);

  // Send the config command to the rogue server
  tcp_command_->genFrame(rogue::commands::config);
}
```

```
void RogueTcpClientProducer::DoStartRun() {
  tcp_command_->genFrame(rogue::commands::start);
}

void RogueTcpClientProducer::DoStopRun() {
  tcp_command_->genFrame(rogue::commands::stop);

  // If open, close the file to which data is being written to.
  if (writer_->isOpen()) writer_->close();
}

void RogueTcpClientProducer::DoReset() {
  tcp_command_->genFrame(rogue::commands::reset);
}
```

Future Development and Conclusion

Rogue is an extremely extensible DAQ architecture that has been extensible used across several experiments

- Current doc: https://slaclab.github.io/rogue/
- Example: https://github.com/slaclab/rogue/tree/main/python/pyrogue/examples

Rogue is actively being developed with updates and enhancements pushed every week

- EPICS capabilities are being expanded
- Ongoing enhancements to online monitoring
- Expand run control capabilities
- Add memory bridges to new hardware
- Continue to explore integration with other DAQ systems
- Continue to identify performance improvements