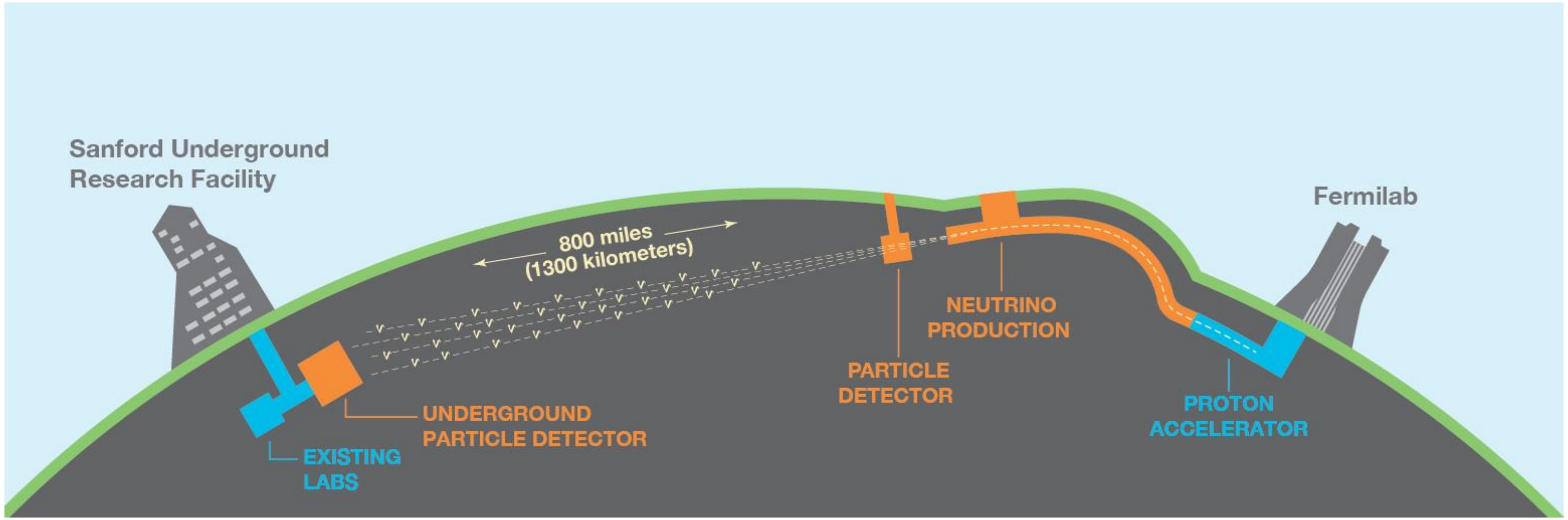
The DUNE-DAQ Application Framework

Eric Flumerfelt, on behalf of the DUNE DAQ Consortium
IEEE Real-Time Conference 2024
Day Month Year

In partnership with:

Introduction

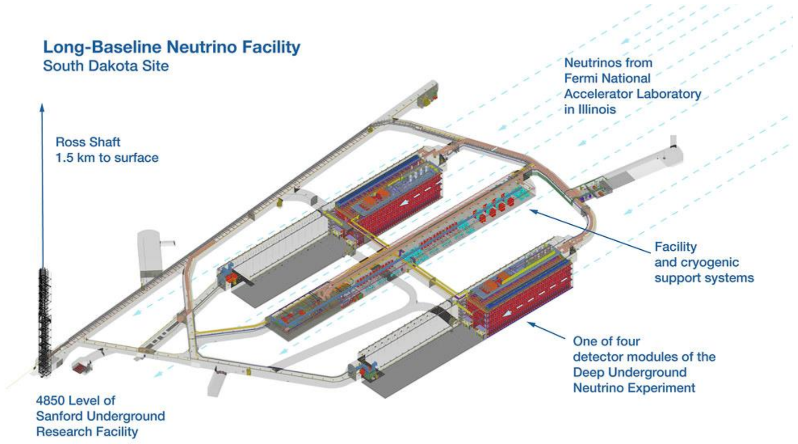
The Deep Underground Neutrino Experiment (DUNE) is a next-generation neutrino oscillation experiment. It will receive neutrinos sent from the Fermilab accelerator campus to a set of detectors located at the Sanford Underground Research Facility (SURF) in Lead, South Dakota. It will also act as a neutrino observatory in the case of a nearby supernova event.


Sanford Underground Research Facility
800 miles (1300 kilometers)
Fermilab
NEUTRINO PRODUCTION
PARTICLE DETECTOR
PROTON ACCELERATOR
UNDERGROUND PARTICLE DETECTOR
EXISTING LABS

DUNE DAQ Requirements

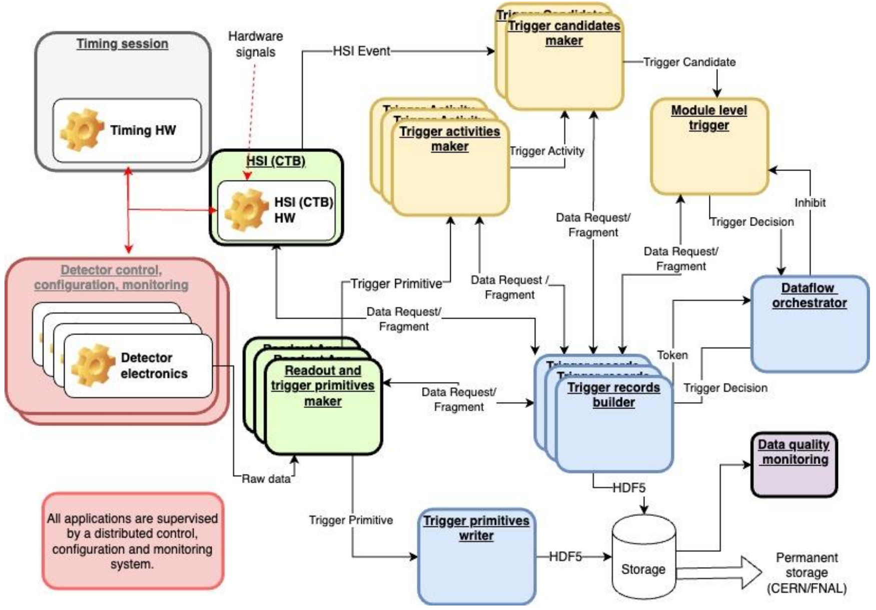
Due to the dual nature of the experiment as both beamline detector and astronomical observatory, there are very strict requirements on the DAQ system

- >99.5% uptime requirement for SNB physics
  - Less than 2 days per year total downtime
  - Includes maintenance of detector elements and computing
- Ability to stream ~100s of raw detector data to tape
  - SNB event requires collecting all relevant data
  - Must maintain normal beamline operations during this time
- Support different trigger streams, including data-driven ones
  - Allow for *in-situ* calibrations and additional physics studies


Long-Baseline Neutrino Facility
South Dakota Site
Neutrinos from Fermi National Accelerator Laboratory in Illinois
Ross Shaft 1.5 km to surface
Facility and cryogenic support systems
One of four detector modules of the Deep Underground Neutrino Experiment
4850 Level of Sanford Underground Research Facility

DUNE DAQ Architecture

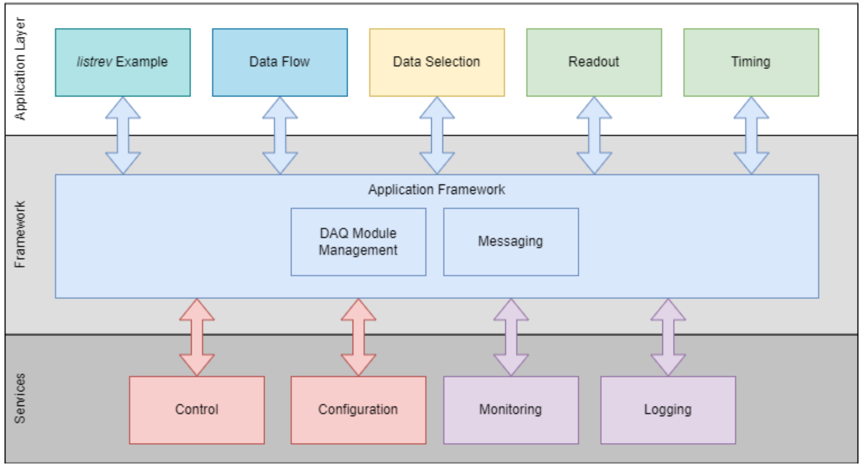
- The DAQ consists of a number of application components which communicate with one another
- Readout applications receive and buffer data from detector electronics
- Data Selection applications create trigger decisions based on configurable algorithms
- Data Flow applications receive decisions, request data from the readout applications, and build Trigger Records which are saved to disk



Framework Architecture

The Application Framework provides a common base for all DUNE-DAQ applications

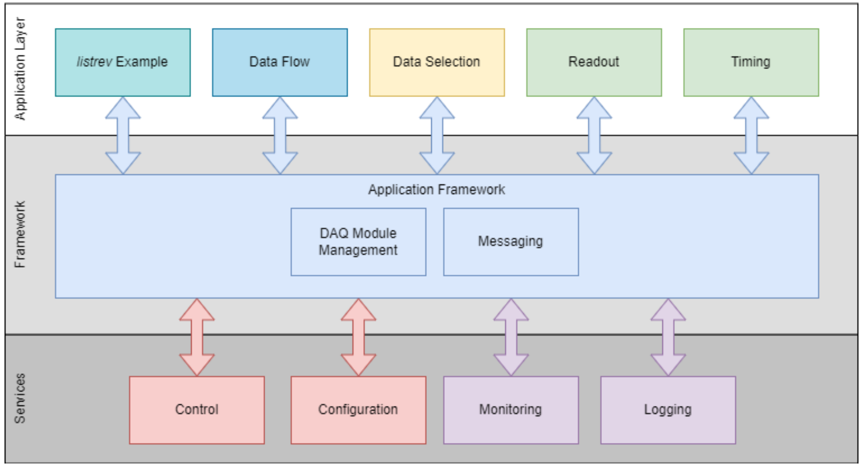
- Initially conceived and designed at a DUNE DAQ workshop at CERN in February 2020.
- Applications are built up from “DAQ Modules”, which implement application logic
- Framework provides the API for modules and a message-passing interface for inter-module communication
- External interfaces for control, configuration, and monitoring are provided by the framework and managed on the modules’ behalf



Service Interfaces

The Application Framework provides a plug-in interface for these external interfaces, allowing for both custom and commercial off-the-shelf implementations to be used

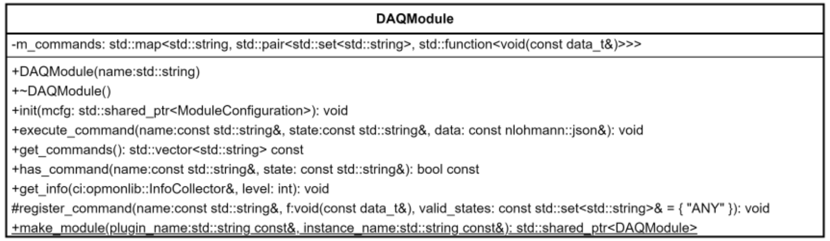
- Application state and command validation occur in Control interface, which can use a REST or gRPC-based communication implementation
  - Commands are dispatched through the framework to the DAQModule instances
- Configuration information is provided to the application at startup using the OKS system adapted from ATLAS DAQ
- Monitoring is implemented via Kafka, influxDB, and Grafana dashboards
- Log messages are handled using the ERS package, also adapted from ATLAS
  - Fermilab's TRACE package provides high-speed low-level debug printouts



DAQ Module API

Each DAQ Module is responsible for implementing a single piece of the overall DUNE-DAQ functionality.

- During module construction, the application framework provides a handle to the configuration so that the module can retrieve its parameters
- Methods which implement module response to commands are registered by command name
  - The DAQModuleManager component of the framework finds modules with registered methods for a given command, when that command is received from the Control interface
- Monitoring information is collected from DAQ Modules by the application and forwarded to the monitoring backend

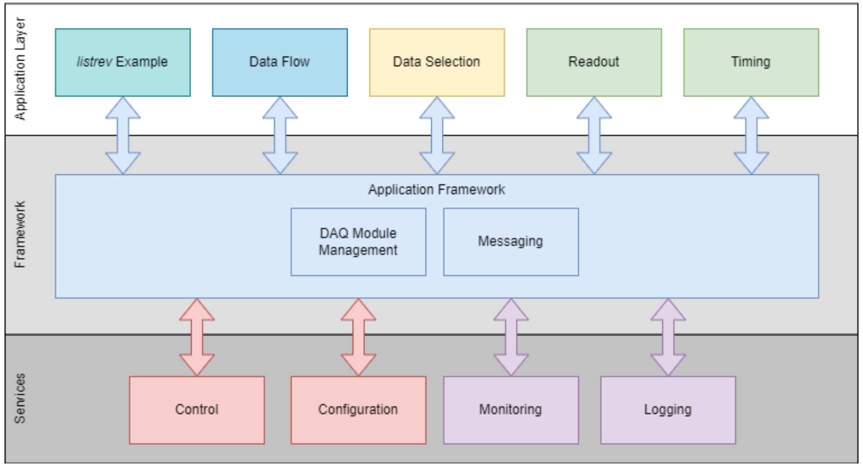


```
DAQModule
-m_commands: std::map<std::string, std::pair<std::set<std::string>, std::function<void(const data_t&)>>>
+DAQModule(name:std::string)
+~DAQModule()
+init(mcfg: std::shared_ptr<ModuleConfiguration>): void
+execute_command(name:const std::string&, state:const std::string&, data: const nlohmann::json&): void
+get_commands(): std::vector<std::string> const
+has_command(name:const std::string&, state: const std::string&): bool const
+get_info(ci:opmonlib::InfoCollector&, level: int): void
#register_command(name:const std::string&, f:void(const data_t&), valid_states: const std::set<std::string>& = { "ANY" }): void
+make_module(plugin_name:std::string const&, instance_name:std::string const&): std::shared_ptr<DAQModule>
```

Messaging API

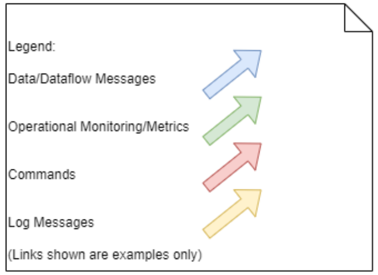
One of the core functionalities of the Application Framework is its messaging API

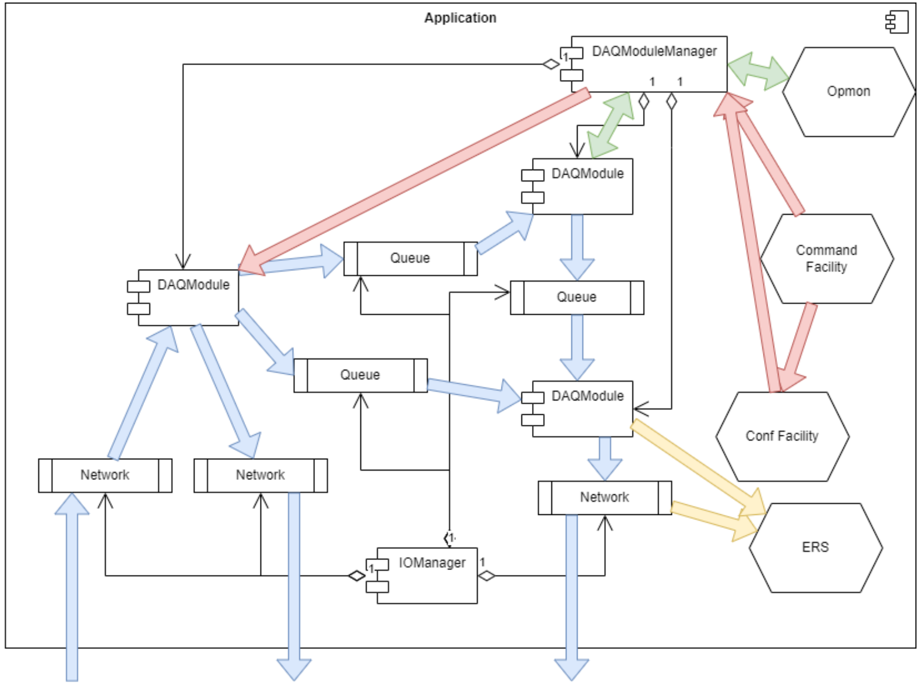
- DAQ Modules send and receive strongly-typed messages
- API is agnostic for whether messages are internal to an application or sent over the network
  - Network transport and serialization are also plug-in, currently implemented via ZMQ and MsgPack
- This system allows for application composition to be changed as needed
- A “connection service” is used to register network links for automatic reconnection



Applications

- DAQ Modules are grouped into applications, which are controlled and configured as a single unit

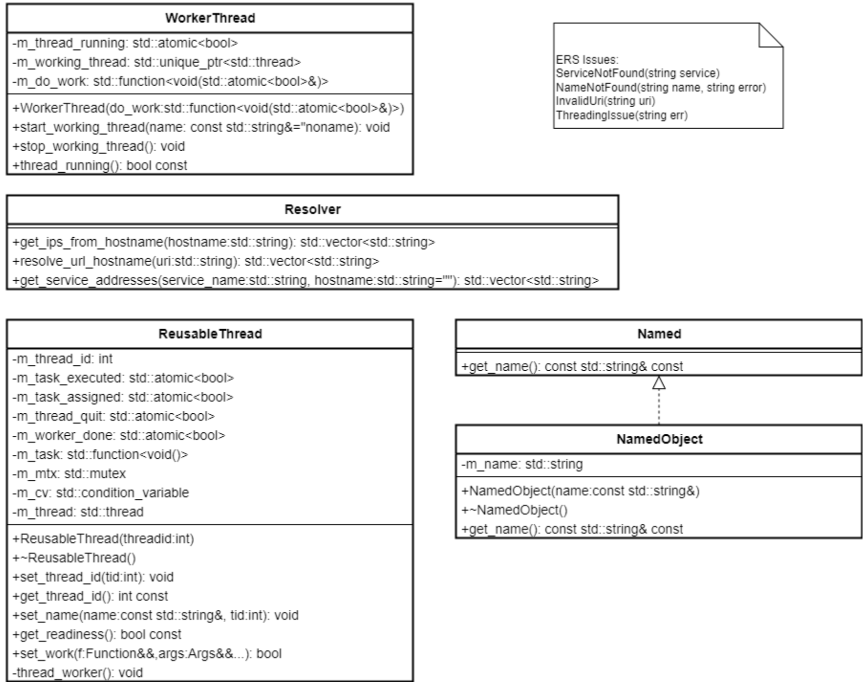

Legend:
Data/Dataflow Messages
Operational Monitoring/Metrics
Commands
Log Messages
(Links shown are examples only)



Threading and Utilities

Several utility classes are provided by the Application Framework

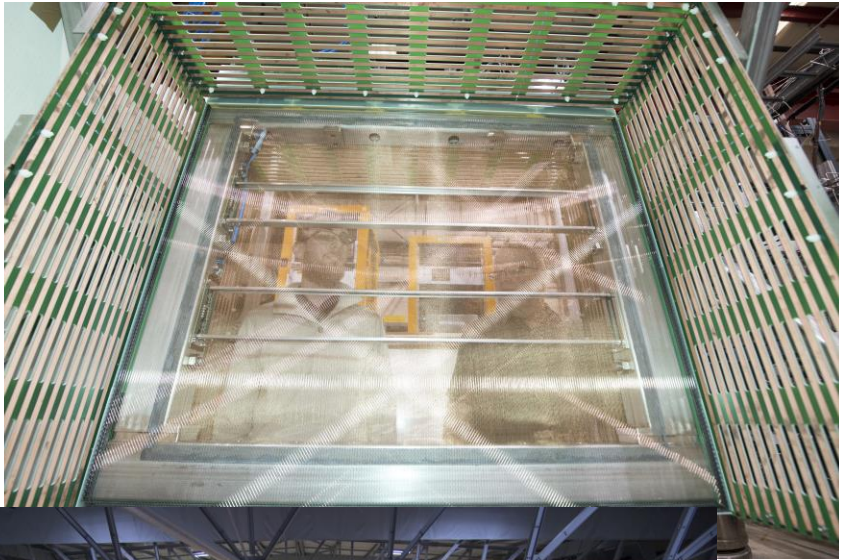
- Most notable are the WorkerThread and ReusableThread interfaces, which provide long-lived tasks and a thread pool implementation, respectively.
- The framework provides a naming hierarchy which is used for monitoring and logging
- DNS resolution utilities are used in the network messaging implementation and have the ability to interface with the Kubernetes DNS for service-based endpoint resolution.



```
WorkerThread
-m_thread_running: std::atomic<bool>
-m_working_thread: std::unique_ptr<std::thread>
-m_do_work: std::function<void(std::atomic<bool>&)>
+WorkerThread(do_work:std::function<void(std::atomic<bool>&)>)
+start_working_thread(name: const std::string&="noname): void
+stop_working_thread(): void
+thread_running(): bool const

ERS Issues:
ServiceNotFound(string service)
NameNotFound(string name, string error)
InvalidUri(string uri)
ThreadingIssue(string err)

Resolver
+get_ips_from_hostname(hostname:std::string): std::vector<std::string>
+resolve_url_hostname(uri:std::string): std::vector<std::string>
+get_service_addresses(service_name:std::string, hostname:std::string=""): std::vector<std::string>

ReusableThread
-m_thread_id: int
-m_task_executed: std::atomic<bool>
-m_task_assigned: std::atomic<bool>
-m_thread_quit: std::atomic<bool>
-m_worker_done: std::atomic<bool>
-m_task: std::function<void()>
-m_mtx: std::mutex
-m_cv: std::condition_variable
-m_thread: std::thread
+ReusableThread(threadid:int)
+~ReusableThread()
+set_thread_id(tid:int): void
+get_thread_id(): int const
+set_name(name:const std::string&, tid:int): void
+get_readiness(): bool const
+set_work(f:Function&&,args:Args&&...): bool
-thread_worker(): void

Named
+get_name(): const std::string& const

NamedObject
-m_name: std::string
+NamedObject(name:const std::string&)
+~NamedObject()
+get_name(): const std::string& const
```

Deployments

- The DUNE DAQ Application Framework has been used as the basis for the DAQ system for ProtoDUNE horizontal and vertical drift, as well as the ICEBERG prototype detector at Fermilab and the TOAD Near Detector prototype.
- The system has evolved with each iteration, with different DAQ Modules being used in different configurations. Readout has transitioned from Proto-WIBs to DUNE-WIBs and from FELIX-based readout to WIBEth Ethernet-based readout.



Current Status & Outlook

- The DUNE-DAQ Application Framework is being deployed at ProtoDUNE II for data-taking activities
  - Support general readout of the detector for hardware debugging
  - Run framework and CCM tests to validate requirements
  - Develop system layouts for full DUNE
- The Framework is in the final stages of development
  - As the backbone of the DUNE DAQ, any changes in the framework API require large amounts of effort to reconcile with module implementations
  - We are reviewing and finalizing functionality to move the framework into "maintenance mode" as a complete product

Participants of the Application Framework Review Workshop held at CERN, February, 2024. In the background is the ProtoDUNE Vertical Drift prototype detector

Acknowledgement & Disclaimer

This work was produced by Fermi Research Alliance, LLC under Contract No. DE-AC02-07CH11359 with the U.S. Department of Energy, Office of Science, Office of High Energy Physics. Publisher acknowledges the U.S. Government license to provide public access under the DOE Public Access Plan DOE Public Access Plan

Fermilab report number: FERMILAB-SLIDES-24-0063-CSAID

Backup

Why make a new Framework?

The first DUNE prototype detector, DUNE 35T, used the *artdaq* data acquisition framework.

- *artdaq* has a rather strict system layout and did not support the desired topology for DUNE DAQ
- The existing command & control system in *artdaq* did not support dynamic reconfiguration of applications, which would be needed to meet the uptime requirement

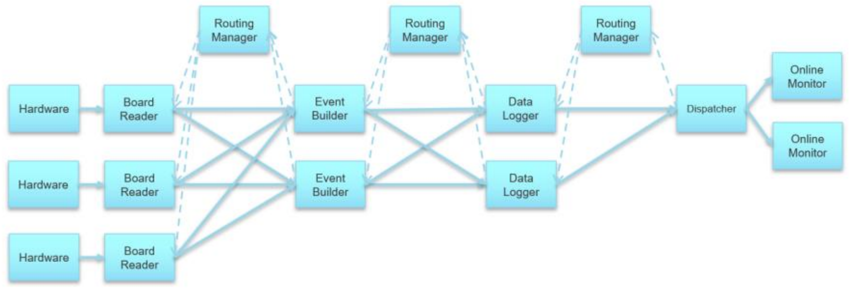


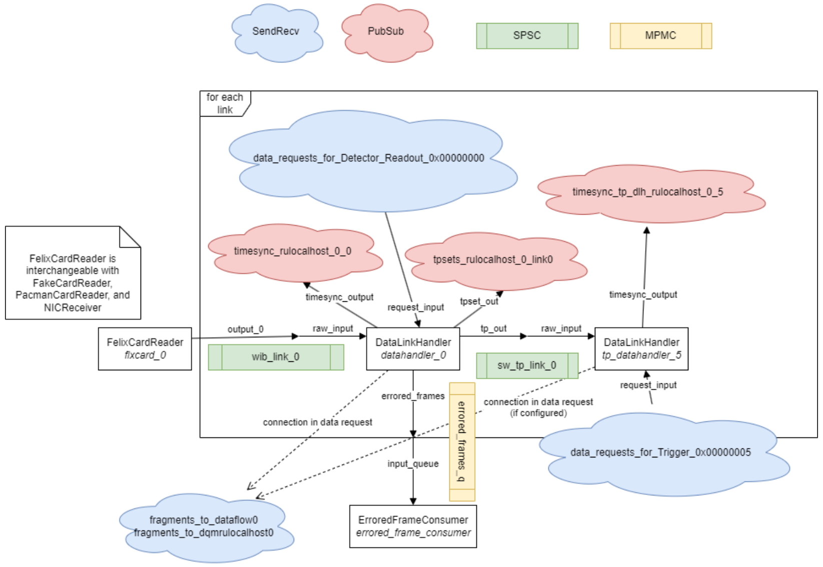
Diagram of the *artdaq* data flow

- Other frameworks were either too task-specific or did not provide suitable flexibility for implementing the DUNE DAQ.

More about the Messaging API

Applications can be of arbitrary complexity, with many links between modules both internal and external

- Messaging connections come in two main flavors, point-to-point and broadcast
- Broadcast connections are used for multi-app synchronization and replicating messages to multiple interested receivers
- Point-to-point connections are used for data flow
- Connections are indexed by name and data type and can be retrieved from the connectivity service using regular expression lookup

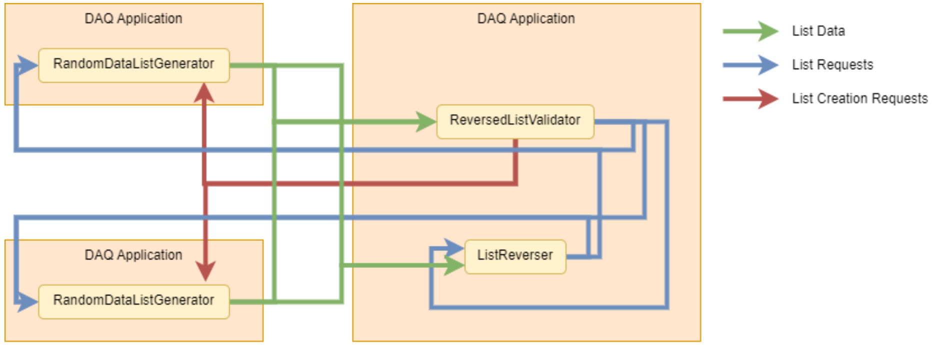


Schematic representation of a readout application

The *listrev* Application Framework Functional Demonstration

The *listrev* example configuration consists of several applications, each with one or more DAQModules. It demonstrates the use of the DAQModule API, several flavors of messaging, and how the configuration, control, monitoring and logging interfaces interact with DAQ Applications

- The ReversedListValidator module sends a "CreateList" message to the Generators
- The Generators create a list of integers, using information in the CreateList (#, pattern) and wait for requests
- The ReversedListValidator requests a list with a given index from the Generators and the ListReverser(s)



Schematic of the "listrev" example configuration

- Upon receiving a request from the Validator, the Reverser sends a request to the Generators, reverses the order of the list, and sends it to the validator