The DUNE event processing framework

David Adams
Brookhaven National Laboratory
DUNE-doc-20099

July 21, 2020

1 Introduction

In November of 2019, DUNE established the "DUNE Software Framework Requirements Task Force" with part of the charge being to propose requirements for "DUNE framework(s) that meet the needs of the DUNE collaboration for signal processing, pattern recognition and large scale data analysis." This document presents some of my thoughts on how to address this charge.

Much of the discussion in task force meetings thus far has focused on an *event* or *trigger record* processing framework such as that provided by art[1] and Gaudi[2]. At present, DUNE along with many other neutrino experiments makes use of art and many of the larsoft[3] components layered on top of art. Art provides scheduling for a collection of *modules* whose input is obtained from an event data store and output may be new event data products written to that store or other products such as plot files or Root histograms and trees. In Gaudi, the analog to the art module is the *algorithm* and most of the comments here about modules apply equally well to those algorithms. The art event data products are limited to C++ vectors of Root-persistent classes. No mechanism is provided for an object in one container to point to those in other containers (e.g. a track to its hits) but separate association products can be and are used to provide such information. Art also provides means for modules to make use of *services* to access non-event information such as detector geometry and data-taking conditions (voltages, channel status, beam status, etc.).

Some sort of framework is needed for production data processing (e.g. generating large simulation samples or reconstructing all the data from a run period) but there are many disadvantages to embedding code directly in modules and services with interfaces dictated by a particular framework. Code inside a module is difficult to develop and test because one must make use of the framework. In addition, code inside a module is not accessible to analyzers who do not want to work inside the framework or at all when the collaboration decides to switch to another framework. The art developers have recognized this and their module design guide[4] recommends that user code be placed in external libraries. The granularity of the modules is driven by the persistent event data model and is often coarser than that steps that are naturally carried out during processing. The requirement to have configurable and plug-in options for these steps is a natural one to place on the framework and is partially anwered by *tools* in both art and Gaudi.

After the event-processing code is outside the module, there are still the problems of accessing event and non-event data. In a simple case, an art module might pass an event data product to a tool and receive back another product which is then added to the event store. But the tool must still be called from a module or some other code that is able to retrieve and store event data. And there will be cases where the tool want to follow links to other data products, e.g. the hits or even raw data to refit a track. Art gallery[5] provides some support for reading the event data but none for writing.

Art provides access to non-event data via services but does not support the use of services for jobs running without the framework. This means that the event-processing tools requiring such data and only be used in framework jobs or DUNE must provide another mechanism to access the data (possibly via the service interface). This is also a problem in other contexts as other DUNE software needs this data.

2 Overview

Instead of placing requirements on the processing framework that carries out bulk event data processing and simulation, this document puts requirements on the infrastructure that supports that framework. In the typical case

that those requirements are also relevant to other activities such as data acquisition and analysis, any implementation should be easily used in all domains. At the cost of some up-front planning, this ultimately reduces effort for infrastructure developers by avoiding duplication. Most important, it allows code developers and analysts to choose the framework or environment most convenient to the task at hand while retaining easy access to a wide range of shared algorithms and services.

3 Software infrastructure

A high-level and efficient programming language is needed for event data processing but also for data acquisition and for analysis, e.g. machine learning and statistical sampling. The obvious choice today is C++ supplemented with experiment-specific and external tools and libraries discussed here and below. Root [6] provides many commonly used facilities (histograms, ntuples, fitting, ...) to the C++ developer. Larsoft provides C++ algorithms and data structures shared with other neutrino experiments. These have been very important in development of code for the DUNE protoypes and we should try to ensure future development there is consistent with the choices made by DUNE.

DUNE already has a very large (and growing) code base, mostly in dunetpc [7] that includes many of the algorithms used in event data processing. Bulk data processing is based on a specific version of this (and other code) and is run at multiple sites on machines with different operating systems. Thus the framework is required to make use of a selectable version of the DUNE code. The framework infrastructure should provide means for DUNE to build and package its software. DUNE scientists will contribute to the bulk reconstruction code base and also develop analysis code individually, in analysis groups and DUNE-wide. These imply that the build system and packaging be user friendly and not just for use by software experts.

4 Tools

The algorithms used in data processing often have many parameters (e.g. thresholds) that are tuned to particular values to use at a specific point in the processing for a particular job. We require that the framework provide means to include the specification of these parameters as part of the overall job configuration. The class that implements the algorithm is called a *tool* and that along with a particular set of parameters is a *tool configuration*. A given tool configuration may be useful in other domains and so we require the same configuration be accessible in the same way when the code using the algorithm is run outside the framework.

There may be multiple tools for a given task and a client should be able to use them through a common interface with compile-time dependence on that interface but no compile- or link-time dependence on the tools. Thus each tool has unique type name (can be the same as the class name) and the full tool configuration includes that name and the parameter values. The framework infrastructure is required to provide means at run time to locate the tool library and create an instance of the configured with those parameters.

A particular tool configuration might be used many times in a job and clients of a given implementation should have the option to use or not use the same tool instance. This is relevant where tool carry *processing state*, i.e. state beyond that derived the configuration, e.g. a histogram filled each time the tool is called. This and the requirement nested configurations be easy for users to understand will likely lead to the design choice that tool configurations are named and accessed by those names, e.g. with a *tool manager*.

The build system provided by the framework infrastructure should make it easy for developers to add their own tool interfaces and tools. It should also be easy to provide named tool configurations.

Although one might (as art did) construct an event processing framework and add tools later, the modules and services in that framework will likely benefit from using the infrastructure developed for tools. They might even be tools. In any case, we require the framework be composed of a graph of modules and impose on the modules, the same requirements imposed on tools above.

5 Event data

Like many HEP detectors, DUNE will stream data at a very high rate and use triggers to select blocks of data (time slices in selected channels) to record for immediate and/or later processing. The data acqusition system typically builds numbered *events* from this data and that raw output and the data derived from constitute the *event data*. The ultimate requirement on the framework is to provide means to process these data and reduce them to a form suitable for the next round of processing or for analysis.

The raw and derived event data are very much a concern for the scientists associated with the experiment and a very important requirement is that data be easily accessible to the algorithms those scientists develop. This is obviously a fundamental requirement for the (bulk event processing) framework but this access is also needed for developing and tuning of the processing algorithms. In addition, those studying detector performance or examining events of scientific interest need to examine or visualize the data for individual events.

Like many experiments, DUNE will have large number of events, on the order of a few Hz data acquisition over a period of years with multiple processing passes. This implies bookkeeping to organize the data into runs, files and datasets. Bulk processing will need to access all events in a dataset and ensure each is processed exactly once. Others will need means to locate individual specified events.

In many or most experiments, the events are processed independently, e.g. for a collider, an event may be associated with a triggered beam crossing. This assumption greatly simplifies the design of an event-processing framework but forces processing outside that framework if it is violated. And that assumption may lead the data acquisition to ignore triggers or duplicate data when there is partial overlap between the data associated with multiple triggers of the same or different types. For DUNE, one can envision a neutrino trigger to read out a full detector, a cosmic trigger to read out a few APAs and a speck/radiological trigger to read out a few channels, each with a different time window and/or compression (including zero suppression). In addition, a supernova trigger will require the association of data over a much longer time window, on the order of tens of seconds.

In the current single-event processing, the framework loops over events and for each, populates an event data store, schedules a graph of modules that read and write from that store, and then writes out some or all of the contents of the updated store. In a multi-event model, there will be multiple event stores. Or, in either case it may be more natural (from an OO point of view), to introduce an event class and pass event objects to the modules. The event class is then provided by the framework infrastructure. That infrastructure is also expected to provide means to locate, read and write events. Following the discussion above, these capabilities should be available both inside and outside the framework.

The framework infrastructure should provide means for objects in the data store to (persistently) point to one another. In the common case that the stored object is itself a container, one should be able to point to individual objects in that container or to the container itself. At minimum, contained references should be supported for vectors and maps indexed by integers or strings. Both direct pointers, i.e. to an object, and associations, dedicated objects that point to two or more objects should be provided.

Art makes use of Root dictionaries to allow users to store a wide range of experiment (or larsoft) defined types and containers of those types. This is likely sufficient. However it only supports associations, i.e. no direct pointers, and only between objects stored in vectors. To go beyond this, experiments are left to create their own mechanisms or make use of those in Root.

6 Services

In addition to the data associated with the current event, event processing requires other data such as geometry or *conditions* information. The latter may include channel status (on/off, good/bad), voltages, trigger menu, beam status, LAr purity, pedestals and much more. It is often desireable to cache this data because it is expensive both to retrieve (e.g. access a database or web service) and then to upack the data for many elements. One set of data is often relevant to a range of events and we would like to use this cached representation for multiple events.

The framework is expected to provide infrastructure supporting *services* where such code can reside including means for user code to retrieve named service instances. The services should support interface and be configurable and locatable at run time as described for tools above. Many of the experiment-provided services will also be useful outside the bulk-processing framework and the framework infrastructure should ensure clients can use services in the same manner in those jobs. There should be means to notify conditions services about which events are currently being processed and when that processing is complete. It may also be useful to know about future events.

In the current art implementation [8], the framework provides means for the service to register a callback at the start of an event (and many other places [9]). The framework creates the service at initialization and makes registered during processing. Services are difficult or impossible to use outside the framework. Services are accessed by interface class name implying that a job may only access one instance for each interface.

Many of the requirements for services can be achieved by making them tools as defined above and then adding interface to specify the event.

7 Multithreading

The evolving relative costs of memory and processors suggests a significant cost reduction in running jobs where multiple threads process the same chunk of memory. There is the possibility to split jobs at the framework level, e.g. to process different events or modules in different threads, but these *high-level threads* are likely to produce only modest reductions memory/cpu usage. The biggest reductions im memory/cpu are likely made with *low-level threads*, in the experiment-specific algorithms inside modules, e.g. splitting on channel or FEMB boundaries or by step within a module. However, it is likely that processing within a module will have bottlenecks where it can only make effective use of a small number of threads and so there will be some advantage to also have a few high-level threads.

The framework and framework infrastructure must be thread safe. There will be many threads and subthreads active at any given time and the framework infrastructure should provide a thread manager that limits the total number and allocates them according to need and task priority. In framework jobs, the framework sets the priority (e.g. FIFO) and, in other contexts, those priorities could be specified in the run time configuration of the thread manager. Experiment code would obtain threads or permission to create threads from the thread manager. The manager could also assign each thread an ID that is unique within the lifetime of the job.

References

[1] The art Event Processing Framework. `https://art.fnal.gov`.

[2] Gaudi project documentation. `https://gaudi-framework.readthedocs.io`.

[3] LArSoft. `https://larsoft.org`.

[4] Art Module Design Guide. `https://cdcvs.fnal.gov/redmine/projects/art/wiki/Art_Module_Design_Guide`.

[5] Gallery. `https://art.fnal.gov/gallery`.

[6] ROOT Data Analysis Framework. `https://root.cern.ch`.

[7] dunetpc. `https://cdcvs.fnal.gov/redmine/projects/dunetpc/wiki`.

[8] Art services guide. `https://cdcvs.fnal.gov/redmine/projects/art/wiki/Guide_to_writing_and_using_services`.

[9] art/ActivityRegistry.h. `https://nusoft.fnal.gov/larsoft/doxsvn/html/ActivityRegistry_8h_source.html`.