artdaq: DAQ software development made simple

Kurt Biery, Eric Flumerfelt, John Freeman, Wesley Ketchum, Gennadiy Lukhanin and Ron Rechenmacher
Fermi National Accelerator Laboratory, Batavia, IL, USA
E-mail: jcfree@fnal.gov

Abstract.
For a few years now, the \textit{artdaq} data acquisition software toolkit has provided numerous experiments with ready-to-use components which allow for rapid development and deployment of DAQ systems. Developed within the Fermilab Scientific Computing Division, \textit{artdaq} provides data transfer, event building, run control, and event analysis functionality. This latter feature includes built-in support for the \textit{art} event analysis framework, allowing experiments to run \textit{art} modules for real-time filtering, compression, disk writing and online monitoring. As \textit{art}, also developed at Fermilab, is also used for offline analysis, a major advantage of \textit{artdaq} is that it allows developers to easily switch between developing online and offline software.

\textit{artdaq} continues to be improved. Support for an alternate mode of running whereby data from some subdetector components are only streamed if requested has been added; this option will reduce unnecessary DAQ throughput. Real-time reporting of DAQ metrics has been implemented, along with the flexibility to choose the format through which experiments receive the reports; these formats include the Ganglia, Graphite and syslog software packages, along with flat ASCII files. Additionally, work has been performed investigating more flexible modes of online monitoring, including the capability to run multiple online monitoring processes on different hosts, each running its own set of \textit{art} modules. Finally, a web-based GUI interface through which users can configure details of their DAQ system has been implemented, increasing the ease of use of the system.

Already successfully deployed on the LArIAT, DarkSide-50, DUNE 35ton and Mu2e experiments, \textit{artdaq} will be employed for SBND and is a strong candidate for use on ICARUS and protoDUNE. With each experiment comes new ideas for how \textit{artdaq} can be made more flexible and powerful. The above improvements will be described, along with potential ideas for the future.

1. Introduction
In an era of tightening high energy physics budgets and a concentration of resources on the LHC experiments, many other experiments which might have been larger in the past have now scaled down, with perhaps the equivalent of only a few FTEs dedicated to a given experiment. In such an environment, the avoidance of duplication of effort becomes ever more critical. The development of \textit{artdaq} at Fermilab over the last few years has played a crucial role in the accomplishment of such a goal, providing users with out-of-the-box tools to put together an experiment’s DAQ software without needing to build it from the ground up. An overview of \textit{artdaq} is provided in this Proceeding. It is recommended, however, to also go to the \textit{artdaq-demo} wiki page [1], where there are instructions for how to install, build and use \textit{artdaq-demo}, a “toy” \textit{artdaq}-based DAQ system designed to educate users on \textit{artdaq}’s features and use.
2. art overview
The original motivation for artdaq was to allow the features of art [2] to operate in a realtime data acquisition environment. Already a mature software package at the time artdaq was conceived, art had been designed for use in offline analyses. art is based on the concept of “modules”, which are C++ classes deriving from a set of art-provided base classes that can provide event filtering, reconstruction and analysis. Modules have many strengths, among which are reusability (a given module can be used in common by many experiments) and configurability (it’s possible to set parameters which control a module without the need for code recompilation). For this latter feature in art, a Fermilab-developed language called FHicL (Fermilab Hierarchical Command Language) [3] was developed. FHicL is analogous to JSON or XML, though syntactically more terse than either (particularly XML) and allowing for C-style #includes. It provides a simple, intuitive syntax providing support for key-value pairs where values can be integers, floats, sequences or even tables of more key-value pairs. art has been a big success and is now widely used by neutrino experiments at Fermilab. Clearly, although art was developed with offline analysis in mind, many of its benefits could also be used for online analysis.

3. artdaq overview
The artdaq package provides different types of processes which serve different roles in a running DAQ system. Below, these process types are described in order of upstream-to-downstream. As art modules are configurable via FHicL, it was seen as logical for FHicL be the configuration language to use for the processes themselves. With this feature, it’s therefore possible to change quantities like buffer sizes, timeout settings, etc. without needing to recompile. Furthermore, as artdaq depends on the art package, for certain process types it’s possible to embed art workflows in the FHicL code which configures these processes. In this way, it’s possible to perform similar filtering, analysis, etc. with incoming data events as can be performed offline.

While this capability was the original motivation for the development of artdaq, there are many other features provided by artdaq which reduce the DAQ software development effort required by experiments which use it. These include support for a state machine model of data acquisition (where artdaq processes respond to “initialize”, “start”, “pause”, “resume”, “stop” and “shutdown” transitions), data transfer, color-coded severity-categorized messaging, and non-blocking online data analysis. These features will now be described in further detail, within the context of a description of the different process types.

3.1. BoardReaders
The BoardReader process is intended to interact with a subset of an experiment’s hardware; a given experiment may use many BoardReaders. Each BoardReader contains an object called a “fragment generator”. Like art, artdaq is written primarily in C++, and a fragment generator is an instance of an experiment-defined C++ class which specifies how data is to be read in from a particular subsection of a detector, using sockets, vendor-supplied libraries and so forth. Each such class derives from the artdaq::CommandableFragmentGenerator base class, and needs to implement some virtual functions called at specific times in the lifecycle of the DAQ process. These include start(), called when the start transition is sent to the processes and intended to initiate data taking, stop(), called at the stop transition and intended to stop data taking, and getNext_(), which is called repeatedly during the running state, i.e., during data taking. How the data is obtained from the hardware in getNext_() is of course experiment-specific. However, every experiment needs to wrap the data as a payload in an instance of an artdaq::Fragment object, henceforth referred to as a “fragment”. A fragment contains a header before the payload containing a sequence ID, used to uniquely identify the event to which the fragment belongs, and a fragment ID, used to identify the particular fragment generator from which the fragment originated. A given call to getNext_() can return zero, one, or even many fragments, which
**artdaq** then sends downstream for assembly with fragments from other fragment generators into complete events.

By default, this sending of fragments downstream is based on a “push” model, in which the fragment generator (and thus the BoardReader) sends a fragment once it’s been created. However, a new and powerful feature provided by **artdaq::CommandableFragmentGenerator** is the ability for it to run in a “request” mode, where even though the fragment generator has created the fragment, it only sends it downstream if it’s received a request via a port on which it’s listening for request messages. A request is an instance of a class called **artdaq::RequestPacket**, which contains as a member the sequence ID of the requested fragment. Both requests and fragments are buffered in memory, with the requests arriving on a separate thread in the fragment generator than the one in which the fragment is created. A request may arrive either before or after the creation of its corresponding fragment. The number of fragments which can be buffered, as well as their permitted duration in memory before being discarded, are settable parameters.

### 3.2. EventBuilders

The assembly of fragments from different fragment generators into completed events is performed by processes called EventBuilders. Currently, in a running **artdaq** system consisting of \( N \) EventBuilders, a given EventBuilder will be in charge of \( 1/N \)th of the events, in round-robin fashion - e.g., in a system with two EventBuilders, one EventBuilder will assemble fragments with even-numbered sequence IDs and the other will assemble those with odd-numbered sequence IDs. The round-robin approach is quite simple but not necessarily optimal and ongoing improvements to it are described in Section 6.0.2. Every EventBuilder is told how many BoardReaders are running in the system, and hence what set of fragment IDs to expect for the fragments from a given event before it can assemble the complete event. Unlike BoardReaders, EventBuilders can be configured to run a set of **art** modules on the completed events - typically filter modules designed to eliminate unwanted events from the DAQ dataflow. After an event has been assembled and has passed any filter module(s) provided, it’s then sent downstream to be written to disk. Another function provided by EventBuilders is the ability to request fragments from BoardReaders, when the BoardReaders are running in the request mode described in the previous section.

### 3.3. Data Loggers

Completed events received by Data Loggers are written to disk using the **art**-supplied **RootOutput** module, which produces files which can be provided later to **art** processes for offline analysis. They also send out the events for potential online analysis, typically through shared memory (though other options are available, as will be described in Section 6.0.3). Crucially, this sending is performed in a non-blocking manner - meaning that the Data Logger prioritizes writing each and every event to disk, and will skip the sending of the events out to the Dispatcher if the sending will impede the disk writing function.

### 3.4. Dispatchers and online monitoring

The process in charge of distributing events to be used for online analysis is called a Dispatcher. Standalone **art** processes configured to run a certain set of **art** analysis modules on the events can register themselves with the Dispatcher and request a subset of the events the Dispatcher receives. As an example of this feature, on installation, **artdaq-demo** can run an example in which plots are made for every 100th event. An **art** process can register itself with a Dispatcher at any point before or during a run, and killing a given **art** process de-registers it with the Dispatcher, freeing the Dispatcher from the need to send any further events to that process. The **art** processes need not know about each other. In this way, the Dispatcher allows different
Multi-layer artdaq Design for Delayed CRV Readout

Figure 1. The planned mu2e deployment of artdaq processes. See Section 3.5 for details.

groups within an experiment to monitor different subsets of data depending on their needs. Indeed, as support for a multicast method of the Dispatcher sending events to the art processes already exists, it’s possible to have art processes performing online monitoring on hosts separate from those on which the main DAQ is running.

3.5. A real-life example
An interesting example of an actual artdaq-based system is that planned for Mu2e [4]. The basic concept of its layout is shown in Fig. 1, and it’s designed to handle a very large volume of data coming from the Mu2e detector by only requesting the full amount of data in an event after successfully filtering on a subset of the data. This subset arrives at BoardReaders, labeled “Distro” on the plot, and is then sent to EventBuilders labeled “Filter”. Unlike as is often the case with EventBuilders, the “Filter” EventBuilders only expect fragments from one “Distro” BoardReader - i.e., its concept of an event actually only encompasses one fragment. The data in that single-fragment event is filtered on via art modules, and if the event passes, it’s sent downstream to a second layer of EventBuilders (labeled “EVB”) which will then send a request up to the hardware via a dedicated BoardReader listening for requests (labeled “Broker”) to get the full set of data for the event, which is then relayed to the EventBuilder via that dedicated BoardReader. The EventBuilder will then send the event downstream to the Data Logger to be written to disk and possibly monitored.
4. Features

`artdaq` contains numerous features which apply to all process types. Some of these (e.g., state machine support) have already been mentioned. However, the following features are also available:

4.1. MessageViewer

One option during installation of `artdaq` is the parallel installation of a process called the MessageViewer. A Fermilab-provided library called MessageFacility [5] comes with `artdaq` and is intended for use in `artdaq`-based systems for text output purposes, in place of the standard C++ `iostream` library. MessageFacility provides support both for labeling messages by their process (or even class) of origination, and by ranking them according to severity (Error, Warning, Info and so on). MessageViewer will print MessageFacility-based messages in a window (Fig. 2) with buttons which allow the user to interactively filter messages by severity level and/or by the process from which the messages come. Additionally, MessageViewer presents the messages in a color-coded fashion, displaying error messages as red, warnings as orange, and so on. In this manner, it becomes relatively easier for users to investigate the behavior of fragment generators, `art` modules, etc., in a running DAQ system.

4.2. Metric reporting

`artdaq` processes are continually monitoring and reporting metrics describing the behavior of a running DAQ - events per second, MB of data per second, the number of incomplete events (i.e., events for which not all fragments have yet been received) currently in a given EventBuilder, and so on. Along with these native metrics, users can report their own quantities inside of the fragment generators and `art` modules they write by using the metric manager API provided by `artdaq`. Similar to MessageFacility, there's a separation between the information provided and the method through which it's displayed. Plugins already exist to show metric quantities using Ganglia, Graphite, syslog or flat ASCII files, and an experiment’s users can write their own plugins as well. Fig. 3 shows the data flow rate in MB/s through the Data Logger used by the DUNE 35ton experiment as reported using the Ganglia metric plugin.

4.3. TRACE

Also provided along with `artdaq` installation is Fermilab’s TRACE package. Designed for developer use, TRACE provides a circular shared memory buffer into which users may very rapidly store debug messages called using the TRACE API, thereby not affecting the timing of the code (obviously an important consideration when troubleshooting a DAQ system!). Support for categories and severity levels is provided, providing further convenience during the debugging process.

5. Experiments which use `artdaq`

Many experiments have already profitably used `artdaq`-based systems, and many more are currently developing them for when data taking begins. Those experiments which have used `artdaq` include DUNE 35ton [6], Darkside-50 [7], and LArHAT [8]. Additionally, Mu2e, protoDUNE Single Phase [9], SBND [10] and ICARUS [11] are at various stages of developing and deploying `artdaq`-based DAQ systems. In Table 1, you can see the flexibility and scaleability of `artdaq` systems as used in these different experiments - using the same set of tools, DAQ systems with very different throughput and filtering requirements can be built.
Figure 2. An example of a Message Viewer window showing both Info and Warning messages (green and orange, respectively).

6. Recent and upcoming improvements
Efforts are ongoing at Fermilab to both improve on existing features of artdaq as well as add new features where warranted. The concepts of flexibility and reusability, already employed during prior development of artdaq, are paramount when looking to find ways to improve the software. Additionally, the developers of artdaq take an active role in seeing the deployment of artdaq-based DAQ systems on Fermilab experiments and are constantly on the lookout for feedback from its users. Some improvements are described here.

6.0.1. FHiCL configuration database Traditionally, the FHiCL parameters used to configure the artdaq processes (including the fragment generators and of course the art modules) have simply been stored in flat text (ASCII) files, one per artdaq process. The manner in which FHiCL variables controlling the DAQ systems are changed is done via manipulation of these
Figure 3. An example of metrics reporting using a Ganglia-based metric plugin from DUNE 35ton. The intervals with zero rate correspond to times between data taking runs.

Table 1. Statistics on various experiments’ DAQ systems. “Data rate” refers to the peak incoming data rate, and “Reduction factor” refers to the data reduction factor due to filtering in the EventBuilder(s).

| Experiment      | Data rate (GB/s) | # BoardReaders | # EventBuilders | Reduction factor |
|-----------------|------------------|----------------|----------------|-----------------|
| DUNE 35ton      | 0.1              | 24             | 16             | 1               |
| Darkside-50     | 0.5              | 12             | 16             | ~5              |
| LArIAT          | 0.3              | 1              | 1              | 1               |
| Mu2e            | 33               | 36             | ~500           | ~100            |
| protoDUNE-SP    | 3                | ~20            | 10-20          | 1               |
| SBND            | 0.4              | ~20            | 10-20          | 1               |
| ICARUS          | 0.4              | ~20            | 10-20          | 1               |

files, meaning either manual edits or perhaps noninteractive editing via a script. Recently, an effort has been underway to store the FHiCL parameters configuring a DAQ run not in ASCII files, but rather, in a MongoDB. A major advantage of this approach is that it becomes possible to set parameters over the web with all the flexibility that entails, using the convenience of an online GUI. For security purposes, it’s only possible to do so if the user has an account on the host containing the database and either launches the browser on the host, or launches the browser on another system but port-forwards a connection over ssh to the host. Additionally, support is provided to save the FHiCL parameters during a particular run to the MongoDB, making it possible to easily retrieve the configuration of the DAQ used.

6.0.2. Event supervisor While a round-robin technique as described earlier in this Proceedings has traditionally been used to assign events to EventBuilders, this isn’t necessarily the optimal approach. In particular, if some EventBuilders in a system fall behind - perhaps due to being on hosts with inferior hardware or a poor network connection as compared to EventBuilders on other hosts - it makes sense to assign events only (or at least primarily) to those EventBuilders which are running smoothly. As such, work is ongoing to develop an “event supervisor”, a process whose job it will be to monitor the performance of the EventBuilders and assign events
to EventBuilders based on EventBuilder performance.

6.0.3. Data transport flexibility via plugins  Traditionally, the manner in which artdaq fragments and events were moved between processes was using MPI. Recently, other methods for moving data around have been investigated and in some cases developed, including RTI-DDS, shared memory (like that already existing between the Data Logger and Dispatcher) as well as basic TCP socket transfer. In the future, support not just for MPI but these other methods of physical data transfer will be provided as a plugin specified via FHiCL. Furthermore, users will be able to develop their own data transfer plugins.

6.0.4. OTSDAQ  While, strictly speaking, OTSDAQ (“Off-the-Shelf DAQ”) [12] isn’t a feature of artdaq so much as it is an extension of artdaq, it’s a very innovative and powerful use of the package which deserves mention here. In essence, while artdaq contains software that eliminates much of the work otherwise needed in an experiment’s DAQ software development, OTSDAQ takes this and extends it into the realm of hardware. For certain supported hardware readout components - currently including BeagleBone, PicoZed and CAPTAN - OTSDAQ provides off-the-shelf firmware which can run on them. Furthermore, it provides fragment generators designed to read out these supported hardware components. In this way, a small beam test experiment, for example, could get a DAQ system (both software and hardware) up and running on a timescale of a day, or perhaps even more quickly with on-hand expert support. For the most up-to-date information on this project, see the OTSDAQ wiki page [12].

7. Conclusions
Reducing unnecessary duplication of effort is always a valuable goal, but especially so when ambitious experiments are facing tight budgets and limited manpower. Like the art package against which it’s built, artdaq relieves an experiment’s developers from needing to spend a great deal of time writing code to perform such standard DAQ software actions as routing data through the DAQ, supporting state machine transitions, dealing with translating online software conventions to offline, and myriad other challenges. Successfully used on a number of experiments to date, artdaq continues to be chosen by new experiments which wish to take advantage of its strengths, and a symbiotic relationship between the developers of artdaq and its users exists whereby user feedback results in continuous improvement of the package.

References
[1] https://cdcvs.fnal.gov/redmine/projects/artdaq-demo/wiki
[2] https://cdcvs.fnal.gov/redmine/projects/art/wiki
[3] https://cdcvs.fnal.gov/redmine/projects/fhicl/wiki
[4] Morescalchi L 2016 The Mu2e experiment at Fermilab Preprint arXiv:1609.02021 [hep-ex]
[5] https://cdcvs.fnal.gov/redmine/projects/messagefacility/wiki
[6] Hahn A et al 2014 The LBNE 35 ton prototype cryostat Preprint FERMILAB-CONF-14-420-PPD
[7] Agnes P et al 2015 Phys. Lett. B 743, 456
[8] Paley J et al 2014 LArIAT: liquid argon in a testbeam Preprint arXiv:1406.5560 [physics.ins-det]
[9] https://twiki.cern.ch/twiki/bin/view/CENF/DUNEProtSPH
[10] sbn-nd.fnal.gov
[11] Antonello M et al 2013 ICARUS at FNAL Preprint FERMILAB-PROPOSAL-1052
[12] https://cdcvs.fnal.gov/redmine/projects/otsdaq/wiki