Fermilab | U.S. DEPARTMENT OF ENERGY | Office of Science

art in *artdaq*

Eric Flumerfelt, for the *artdaq* team

Frameworks Workshop

05 June 2023

How *artdaq* uses *art*

Since *artdaq* version 3, *art* is started as a subprocess (fork/exec) by the *artdaq* system. These “companion *art* processes” are fed events via shared memory and custom input and output modules are used to decode and encode *art* events stored in *artdaq* Fragments.

artdaq contains library code co-developed with the artists that handles the actual conversion of art Events from and to Fragments, and handles the necessary pieces of art initialization such as process history and parameter set registry. We generally refer to this library code as “art goo” due to the opaqueness of this code.

How *artdaq* uses *art*: Services

Our shared memory interactions are isolated to the ArtdaqSharedMemoryService, which provides artdaq::Fragment objects to the ArtdaqInput implementation code.

We also have an ArtdaqFragmentNamingService, which translates the numeric Fragment type ID field to a string suitable for the art instance names for the Fragment products stored in an art Event.

How *artdaq* uses *art*: Online Monitoring

For “online monitoring” we have a second set of input and output modules which allow two art processes to communicate directly with one another (the artdaq Dispatcher starts a “companion” art process when connected to from the “monitor” art process; this allows pre-filtering to be performed before the “companion” art process transfers data, potentially over the network, to the “monitor” process).

How *artdaq* uses *art*: Modules

artdaq provides a near-copy of RootOutput with a few DAQ-specific customizations. RootDAQOut uses POSIX system calls to ensure that written data does not persist in buffer cache.

Several analyzer modules are provided by *artdaq* for use in DAQ systems, they read Fragments in the art events and look for empty events, missing data, and other DAQ-related issues.

artdaq currently maintains HDF5-based example input and output modules (https://github.com/art-daq/artdaq_demo_hdf5), though we do not believe that they are currently used by anyone. These produce and consume “raw data only” HDF5 files (Fragments and Event Headers only, no history or provenance information).

artdaq's "stretch goal" for *art*

Currently, *artdaq* is responsible for the maintenance of the *art* Event serialization and deserialization code ("*art* goo"), and on occasion this has led to very subtle and difficult to diagnose issues when the internals of the art framework change.

It would be better if art provided a set of framework functions which use a standard flat data format (e.g., TBufferFile) which the *artdaq* code could then use to construct *art* Events from the data stored in Fragments and create Fragments from already-serialized *art* Events.

If this is not possible, it would at least be helpful if the artists continue to provide sufficient warning whenever framework internals change in a way that might disrupt our "*art* goo", and possibly assist in developing sandboxed unit test cases that could quickly identify potential issues.

Summary

Our usage of art is possibly unique in that we do not use a file-based interface, which leads to our somewhat unusual method of communicating with and between art processes. Our current solutions are dependent on ROOT for providing the serialization and deserialization methods (via TBufferFile) for both events and art process initialization (things like history, process registry, and more).

artdaq also heavily uses the plugin facility provided by *cetlib* and has MessageFacility endpoints defined in *artdaq_core* and *artdaq_mfextensions.* These framework pieces support the basic functionality of *artdaq.*

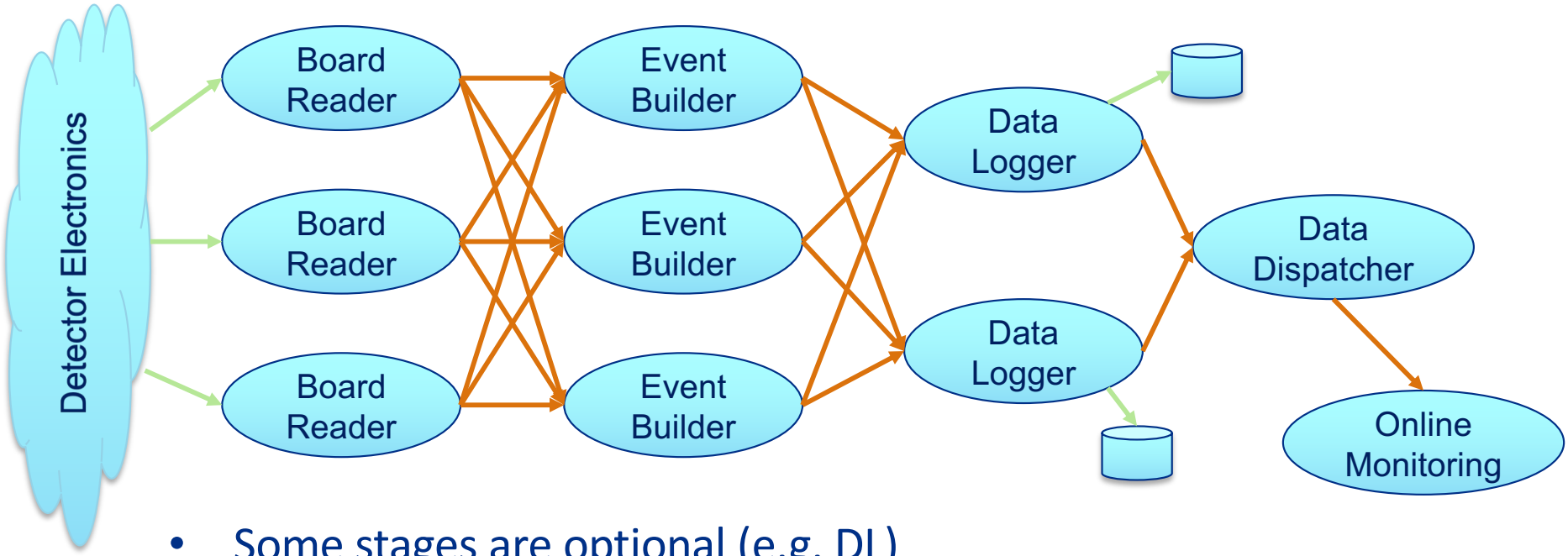
Backup slides

Introduction

Reminder that *artdaq* is a data acquisition framework developed and maintained withing CSAID. It is used by multiple experiments at Fermilab, including Mu2e, ICARUS, and SBND.

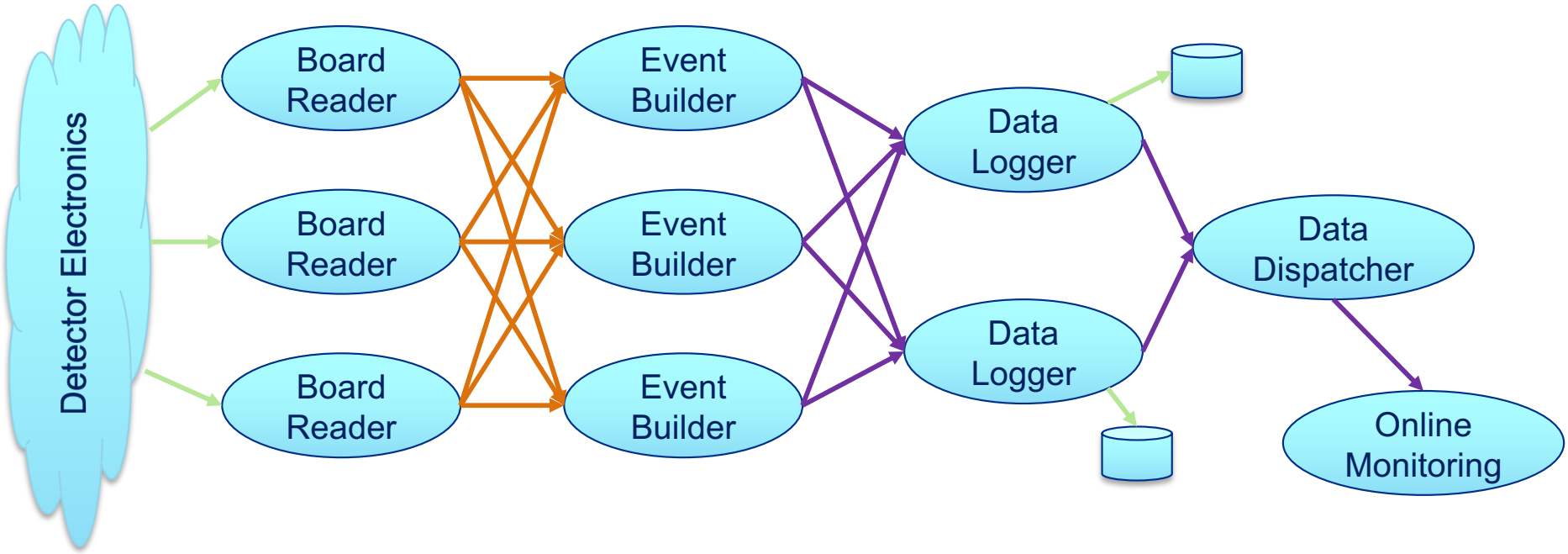
It uses *art* for optional event filtering/transformation, and for writing raw data on disk in art/ROOT format. Etc.

Reminder of data flow in *artdaq*



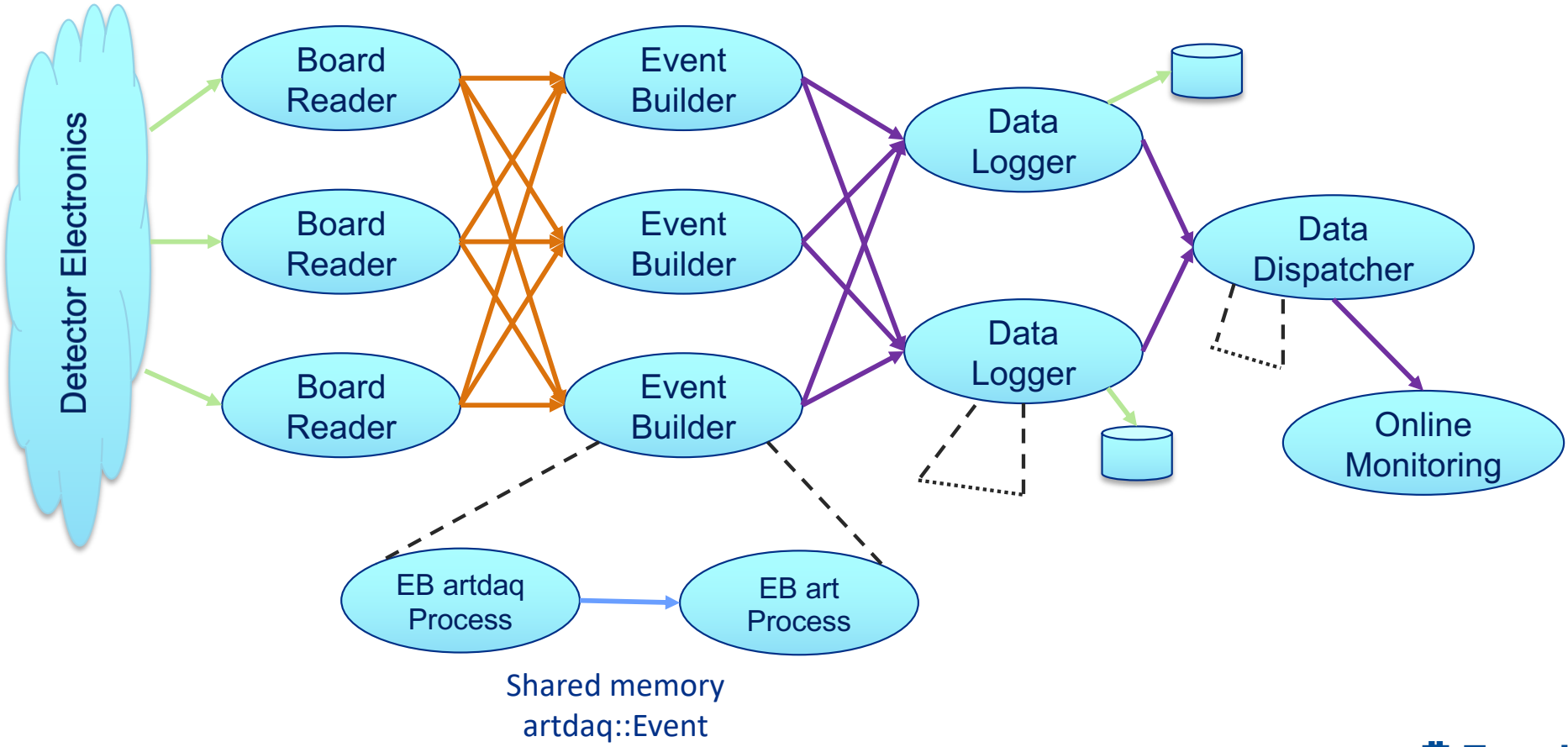
- Some stages are optional (e.g. DL)
- Connection details are highly configurable
- Data transfers between stages in *artdaq* downstream of BRs use the artdaq::Fragment class

Contents of artdaq::Fragment change downstream



- Initial contents are raw data from electronics (plus artdaq-added header)
- Once the data has been touched by *art*, the payload of the artdaq::Fragments becomes serialized art/ROOT objects

Separate art processes at EB, DL, Disp stages…



Fermilab

Input and output modules inside *artdaq art* processes

ArtdaqInput

- *art* input source reads Fragments and determines if they contain a serialized art event, reconstituting it if so. If they are data fragments, then it creates a new *art* event to hold them. (This gives us the flexibility to have systems like Mu2e where there are multiple “layers” of EventBuilders…events which pass the filtering done in the first layer then request CRV data which is added to the events)

BR -> (?:EB -> EB art (ArtdaqInput & RootNetOutput))+ -> DL -> DL art (ArtdaqInput & RootDAQOut & RootNetOutput) -> DI -> DI art (ArtdaqInput & TransferOutput) -> OM art (TransferInput)