artdaq Metric Reporting Infrastructure

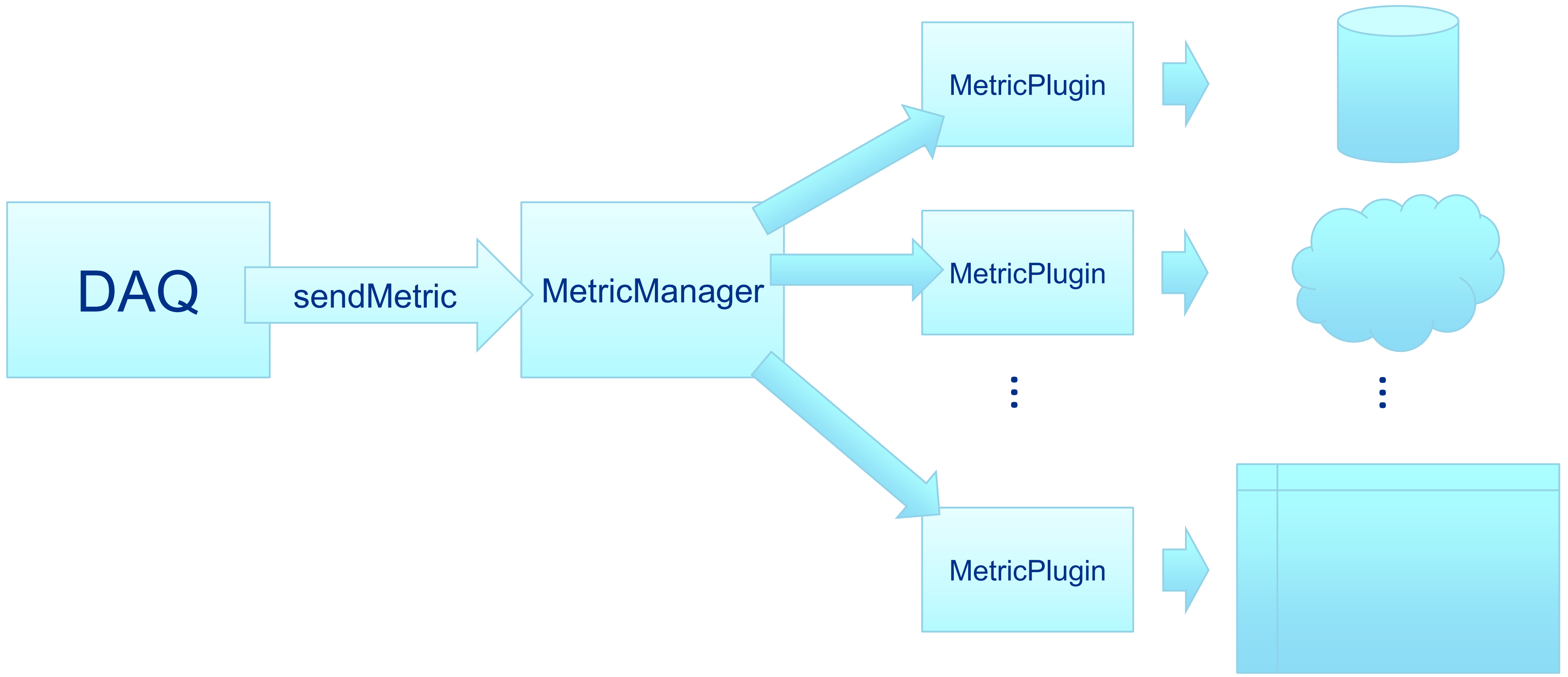
Eric Flumerfelt
DUNE Dataflow Working Group
20 November 2019

Overview

artdaq has a built-in plugin-based metric reporting system, which is designed to accept metric data at high rates from the DAQ, aggregate this data in a reasonable fashion, and send it to any number of metric back-ends. An *artdaq* metric sent to the backend consists of a name, value, and unit. Metrics are indexed and combined using the metric name as a key, and metric names and units are expected to remain constant through an entire run of the DAQ software.

The metric reporting implementation is entirely contained within the artdaq_utilities package, which depends on messagefacility. (artdaq_utilities uses functionality from fhicl-cpp, cetlib, and cetlib_except.)

artdaq Metric Reporting Infrastructure



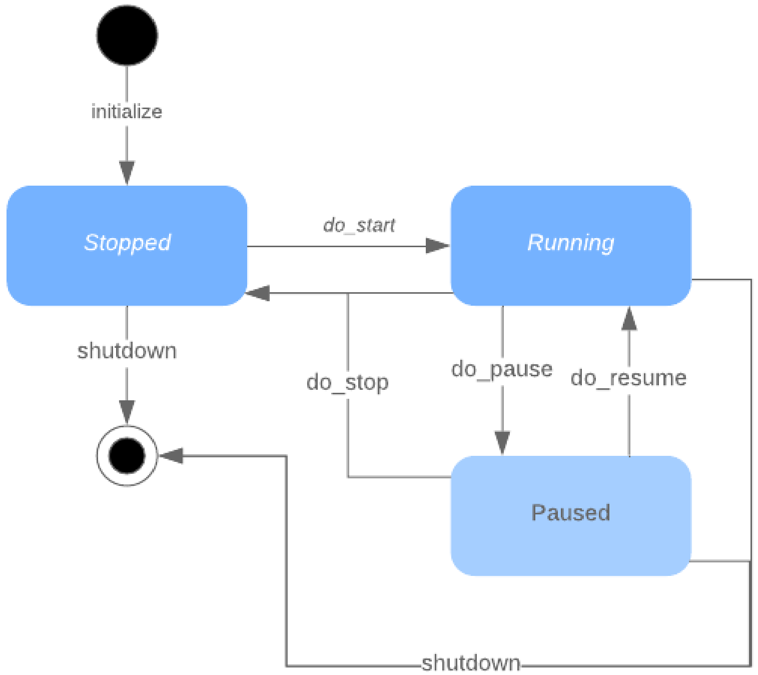
MetricManager

The MetricManager class instantiates and controls the configured MetricPlugins and serves as the central switchyard for metrics. It is designed to accept new metric data points from *artdaq* and return control in the smallest amount of time possible. These metric data points are placed onto a queue where a dedicated thread distributes them to the metric plugins. Metrics are aggregated while sitting in the queue to reduce the memory footprint in high-rate situations.

MetricManager

MetricManager maintains a state machine and will not allow metrics to be reported when it is not in its “running” state

- MetricManager state may be distinct from the *artdaq* running state
- Some MetricPlugin implementations use the various transitions to connect to their back-ends or to zero out the metrics at the end of the run.



MetricPlugin

Each metric plugin instance is characterized by its plugin type, reporting interval, and enabled metric mask.

Some metric back-ends (such as Ganglia) are limited in the amount of granularity they provide. The reporting_interval parameter ensures that updates are not sent more frequently than the backend can handle, conserving CPU and network resources.

The MetricPlugin takes care of aggregating metric data points over the reporting interval, and each metric is given a set of mode flags (Min, Max, LastPoint, Average, etc.) which control this aggregation. If more than one mode is specified, individual metric instances are automatically created for each mode (e.g. “MyMetric - Min”, “MyMetric - Max”)

MetricPlugin

Each metric call is also assigned a level, and these levels determine which metrics are reported to which plugins.

Several plugins of the same type may be instantiated with different metric masks and intervals to control reporting.

```
metrics: {
    brFile: {
        metricPluginType: "file"
        level: 2

fileName: "boardreader/br_%UID%_metrics.log"
        absolute_file_path: false
        uniquify: true
    }
    brVerbose: {
        metricPluginType: "file"
        level: 5

fileName: "boardreader/br_%UID%_verbose_metrics.log"

        absolute_file_path: false
        uniquify: true
    }
}
```

MetricPlugin

MetricPlugin is also the base class that defines the interface that metric plugins for new backends should implement.

Several example metric plugins are included in artdaq_utilities, and several more with additional dependencies are available in separate packages (artdaq-ganglia-plugin, artdaq-dim-plugin, artdaq-epics-plugin).

Metric Reporting Example

```
#include "artdaq-utilities/Plugins/MetricManager.hh"
#include "artdaq/Application/LoadParameterSet.hh"
#include "fhiclcpp/types/TableFragment.h"
#include <iostream>
struct Config { fhicl::TableFragment<artdaq::MetricManager::Config> metricmanager_config; };

int main(int argc, char* argv[]) {
    auto config_ps = LoadParameterSet<Config>(argc, argv);
    artdaq::MetricManager mm;
    mm.initialize(config_ps, config_ps.get<std::string>("application_name", "SimpleMetric"));
    mm.do_start();
    int level = config_ps.get<int>("metric_level", 1);
    std::cout << "Enter metrics in <name> <value> <units> format. Ctrl-D to end" << std::endl;
    std::string name, unit; double value;
    while (std::cin >> name >> value >> unit) {
        mm.sendMetric(name, value, unit, level, artdaq::MetricMode::LastPoint);
    }
    mm.do_stop();
}
```

- Note that in *artdaq*-based code, MetricManager is instantiated and configured by the Globals class (#include “artdaq/DAQdata/Globals.hh”), and should be called via:

```
if (metricMan) metricMan->sendMetric(...);
```