DUNE FD DAQ Observability

Brett Viren

(draft v0.1)

Contents

1 Introduction **1**

2 Observability **2**

3 Applicability of Observability Model to DUNE FD DAQ **3**

4 Goals and requirements **4**

5 Tour of observability components **5**
5.1 DIY . . . 5
5.2 OpenTracing . . . 6
5.3 Jaeger . . . 6
5.4 Elastic Stack . . . 8
5.5 `fluentd` . . . 8
5.6 Graphite . . . 9
5.7 Grafana . . . 11
5.8 InfluxDB . . . 11
5.9 Prometheus . . . 11
5.10 Comparisons . . . 13
5.10.1 LogStash vs fluentd . . . 13
5.10.2 ElasticSearch vs PostgreSQL vs Graphite . . . 13
5.10.3 Kibana vs Grafana vs. Graphite-web . . . 13

6 Prototype Architecture **13**

1 Introduction

The DUNE FD, as a whole system, has a requirement for the minimum uptime allowed for producing data. The maximum allowed downtime is a few percent. The DAQ is obviously on the critical path for meeting this requirement. The DAQ is predominately a software system and production software systems of similar scale and complexity achieve this level of uptime. It is reasonable to expect that the DUNE FD DAQ can also achieve high uptime by emulating and adopting elements from these production systems. One ingredient in their success is to implement a concept called *observability*.

This note begins with a description of the concept of *observability* as used in the development and operation of complex distributed software systems and applicability of the observability model to DAQ design is given. Some goals and requirements to guide development of a DAQ observability

system are presented. Then, a tour of a select subset of existing systems is given. Finally a prototype architecture is proposed as an initial sketch of the major components that might eventually be put into place for a DUNE FD DAQ osbservability system.

2 Observability

In the formalism of control theory, *observability* is a measure of the amount of information about a vector representing an internal state of a complex system which can be inferred given another vector representing measurements of that internal state. This term has been borrowed by a broad community of developers to refer to the techniques and practices they have developed to report measurements and attempt to determine the internal state of their complex distributed software systems.

Typically, *observability* is important for large scale systems which must interact with external entities (eg, humans) reliably and promptly. Observability is intended to provide information which may predict and determine the cause of failures or delays in the system. In practice, the external system to such systems follow a request-reply communication pattern. Between these two endpoints the complex system operates.

Observability in complex software systems is not yet well defined. It can be thought as some more organized form and a superset of "logging" or "monitoring". Observability systems typically operate on the following classes of data:

event an occurrence at a point in time observed within a local software context and emitted as a set of descriptive values including a timestamp, a context identifier and a data identifier that associates the primary data involved in the occurrence.

metric a summary over a set of events.

span a period of time, measured in a local software context, over which that context performs a set of operations and to which a set of events and metrics may be associated.

trace a graph (usually a tree) structure formed by connecting all spans that hold a common *data identifier*.

Events and spans must be formed inside the system being observed. Metrics may be formed in the observed system or may be formed from events and spans collected by an observability system. As traces represent correlations across both (real) time and local software contexts throughout the distributed application they must be formed by the observability system after aggregating the spans to some central storage.

The basic architecture of an observability system may be described in terms of these parts or aspects:

instrumentation adding code to application software or programs to the OS which produce and output the observability objects of type event and span and possibly metric.

transport movement of observability objects between parts of the system.

transformation operations applied to observability data objects to produce new data objects, such as producing metrics and traces fro events and spans.

storage recording observability data objects to nonvolatile media for later use.

query functionality to recall or select a subset of observability objects, either from in-flight transport streams or from at-rest storage.

visualization production of visual representation of the observability data objects or their derivatives.

analysis operations to model the state of the system under observation through consumption of the observability objects.

alarming production of interrupt messages to alert humans or software systems when an exceptional state or behavior is detected.

3 Applicability of Observability Model to DUNE FD DAQ

The model described above and as typically implemented is not a perfect match for the entirety of the DUNE FD DAQ functionality. At its core is the *trace* which requires *spans* to present a *data identifier* from which the *spans* may be correlated.

The predominate use of observability is in complex "web applications" which present a request/reply interaction pattern to the external world or between divisions of its internal functionality. For such systems it is easy and natural to assign a unique identifier for each request and to carry that throughout the internal "business logic" and thus allow any *span* to record and present it as the *data identifier*.

This request/reply pattern partially matches portions of the DUNE FD DAQ design. The self-triggering mechanism of the DAQ culminates in a trigger decision (TD) formed by the data selection (DS) system. The TD being consumed by the back-end (BE) DAQ system can be interpreted as a request. The TD is expected to be received by the BE via a pub/sub protocol so there is no explicit (or otherwise) reply. However, the TD does provide an obvious *data identifier* which can be used by spans produced by software downstream of TD production. An obvious form would be a combination of the partition, run and trigger numbers provided by the TD.

The BE then goes on to make multiple requests to front-end DAQ buffer interface (BI) services (asynchronously) and from each receives an explicit reply containing the requested detector data fragments (or potentially an error). This subsystem communication pattern exactly follows the request-reply model and so can produce meaningful spans in a natural way. The unique ID here might be formed by adding the BI identifier to the unique identifier formed from the TD in the above example.

The model breaks down for the part of the DAQ which actually produces a TD. These upstream DAQ processes operate following purely a feed-forward pub/sub pattern. The general flow is one of aggregating information across detector space and time and dropping data identified as containing no useful information. Thus, there is no meaningful *data identifier* which can be a'priori defined. It is not possible to identify and assign a *data identifier* to the pre-TD objects which is a'priori known to survive to the TD.

However, accessible to all data through this part of the DAQ is a common hardware clock time. This may be used as a soft identifier in spans produced from the applications comprising this part of the DAQ. It is soft because it is also high resolution. The standard *trace* correlation can not be used as it requires an exact match of *data identifiers*. Any equivalent to a *trace* must take this softness in mind.

As an aside, it is noted that the DS system itself has the same characteristics of an observability system in that it has many of the aspects listed above. The DS system can be viewed as an

observability system for the detector activity (and noise) where the sources of events and spans are the producers of trigger primitives ("hit finders"). The trigger candidate (TC) and pinnacle TD producers can be identified as the middle element in the pipeline model of an observability sytem described above. At the very least, this conceptual symmetry must be kept in mind, if not exploited, when observability is applied to the DS system.

4 Goals and requirements

Careful architectural design, prototyping and validation, high standards for code quality are all critical to achieving high uptime for a complex software system. But these are not enough to assure zero problems with the resulting system. The goal of developing an observability system for the DUNE FD DAQ is thus to detect, and possibly predict unanticipated failures and to enable quick diagnoses of the cause of failures and their remediation.

Toward these goals, and with understanding of the larger goals of the DAQ, an initial set of requirements and allowances for an observability system are introduced. These requirements are necessarily not quantitative as much can not be known about the problem that observability tries to solve until the system is in operation and failures are encountered. Nonetheless, the intention is that these may help guide decisions toward planning, selection among the potential solutions and eventual development and deployment.

compatibility A solution must bring value without requiring unwanted changes to existing and planned DAQ software design and overall DAQ architecture. Any parts that require direct integration must support the language of the application (typically this means C/C++ and possibly Python).

non-blocking As an observability system is intended to increase the reliability of the primary DAQ operation and so it must not itself pose limitations on or risk toward those primary functions. Specifically, a primary DAQ process shall not be made to block or otherwise be significantly delayed or burdened in its normal operation by contributing to the observability system.

lossy The rate of observability objects emitted by a DAQ software component is expected to reflect, in some way, the rate of the primary data flow through the component. Even from components in high rate areas of the DAQ the system should attempt to not drop observability objects. However, some loss of observability objects may occur and the system should be robust against such occurrences. Intentional downsampling of observability objects is explicitly allowed.

comprehension The information required to solve future and as yet unknown problems can not be fully specified. Thus, the system should attempt to produce and make available as much information as possible, while honoring other requirements. Where practicalities limit the amount of data collected, these limits shall be imposed as downstream in the observability system as feasible in order to allow novel solutions to be inserted upstream where the data may be richer.

promptness For high uptime, prompt discovery of problems is required to assure prompt resolution. A maximum allowed latency can not be quantitatively derived from the maximum downtime requirement as the rate and duration of downtime occurrences can not be known until the system is put in operation and failure modes are understood. Thus relative requirements are stated. Streaming analysis should be preferred over (or added to) query-based or offline analysis performed on stored data. Humans and expert systems should receive alarms

of exceptional behavior, instead of (or in addition to) polling analysis results in order to deduce problems. Automated response shall be preferred over (or in addition to) human detection and intervention.

no-lock-in As we do not yet know all problems let alone their solutions we shall retain a loose coupling between the parts of any solutions we select. If a proposed solution clearly locks the system out of accepting future solutions to novel problems, known and unknown, then the proposal shall be rejected.

5 Tour of observability components

This section presents a tour of a select subset of observability system elements which may be beneficial for adoption by a DAQ observability system. It begins with a comment about the relative need to develop vs adopt. It then presents the tour. Inclusion in the tour requires that the potential technology:

- is consistent with the goals and requirements listed above,
- may be self-hosted (ie, are not solely "cloud-based" service),
- is distributed under a Free Software or Open Source license,
- posses an active user community.

5.1 DIY

Many large corporations make grand public-relations statements about their observability systems and they have excited fan-bases in industry and in their development communities. However, after studying the available systems it is clear that the majority of features provided with great fan-fair already exist in some form in the current DAQ software running at ProtoDUNE.

The following discusses aspects covered by the current DAQ software and some of the important deficiencies are identified potential solutions are proposed.

Instrumentation for the production of objects following the event/metric model is already implemented in parts of the DAQ although the coverage is not yet as complete as it will need to be. Transport and transformation components already exist. For example, PTMP can collect metrics from all threads in an application and publish them in aggregate. These can be further aggregated across processes. At the thread, application of network level, these metrics can be converted and fed into Graphite time-series database and visualized in Grafana (both described below in the tour). Metrics may also be produced from *artdaq* layers and aggregated in Graphite or other stores such as CERN's DIM.

There is not yet any equivalent to the correlation embodied by the span/trace model. As the event/metric instrumentation is already under the control of the DAQ software developers, extending it to cover the production of spans would require incremental effort. This effort could be in the form of instrumenting the code with OpenTracing API calls (described in the tour below) which would have a force magnifying effect.

One of the largest gaps in the current DAQ is a system for performing ongoing prompt analysis and alarming on any exceptional results. Of course strong effort goes into offline analysis which is in part responsible for the fast and successful development of the current DAQ system. As the DAQ goes into production, detection and analysis of its failures must produce actionable results in

seconds or minutes and not hours or days. As the analysis is very unique to the data, this may be one area that must be developed in-house, at least in part. A system to support this must allow for continual analysis development, including ad-hoc or temporary inclusion of novel algorithms, changes in configuration parameters, alarm thresholds, etc.

5.2 OpenTracing

OpenTracing[1] provides the *instrumentation* aspect. The project defines an abstract application programming interface (API) for the production of *spans* and associating *events* and *metrics*. An API is defined for nine languages including Python and C++. The application developer adds code using this abstract API and at run time the interface may be bound to a concrete implementation. This allows application software that uses the OpenTracing API to emit observability data objects to any system which either provides an OpenTracing API implementation or which can accept message from such an implementation. Switching between OpenTracing implementations need not require the application software to be further developed or even recompiled.

Instrumenting application software with the OpenTracing API require code to be added as illustreated in the following code sketch. This sketch is mean to show the level of programming effort and is not exact syntax which of course depends on the given language:

```
tracer = some_factory(...)
span = tracer.start("operation_name")
// usual application operation....
child_span = tracer.start("child_name", parent=span)
// usual application "sub" operation...
child_span.set_tag("somekey", somevalue)
// ..."sub" operation ends
child_span.destroy()
// ... parent operation ends
span.destroy();
```

Destruction of a span is typically scope-aware so the explicit `destroy()` may be omitted. This example is close to the level of coding required for instrumenting Python. Attaching key/value pair "tags" to a span in C++ requires somewhat more verbose syntax.

There are nine OpenTracing *tracer* implementations officially supported[2]. Of them, two are considered in the subsequent sections (Jaeger and Elastic APM).

In researching OpenTracing one ongoing development in the community was found which bears mention here. While OpenTracing provides an abstract API another project, OpenCensus, provides a full concrete implementation of the same concepts. As one is abstract and one is concrete the two are seen as orthogonal by their respective communities and the two projects are in the process collecting themselves under a common umbrella project named OpenTelemetry.

5.3 Jaeger

Jaeger[3] provides the aspects *instrumentation*, *query*, *visualization*. The project provides an implementation of the OpenTracing API. The resulting messages may be sent to Cassandra (3.4+) or ElasticSearch (5.x/6.x but not 7.x) for storage. A web based visualization provides:

[1] `https://opentracing.io/`
[2] `https://opentracing.io/docs/supported-tracers/`
[3] `https://www.jaegertracing.io`

- scatter plot of traces as a function of time,
- Gantt-chart of spans in a trace to browse spans,
- data driven discovery of the span graph,
- weighted span directed graph.

The Gantt-chart, an example of which is shown below, provides of spans could be useful for visually identifying unusually long latency spans and potentially unusually short spans. It may also be useful in contextualizing any explicit errors communicated through a span. Individual span data may be visualized through the Gantt-chart view. The remainder of the web UI does not appear particularly useful on initial evaluation. Limits on number of traces to display indicate the web UI is suitable for TD level tracing but possibly not for traces attempted at the TC or TP level.

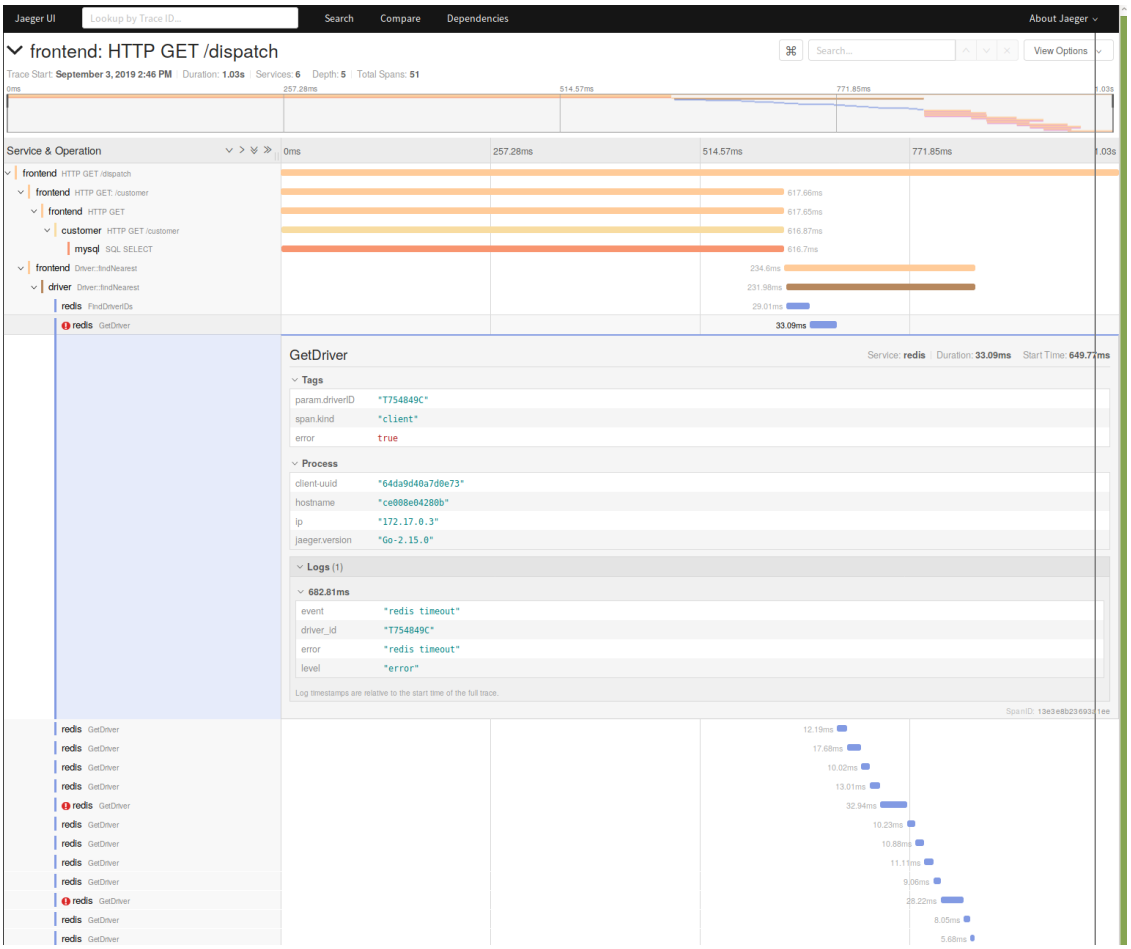


Figure 1: Example Gantt-chart view from Jaeger web UI fed by the provided demo HotROD web application. The data from a span indicating a (contrived) error is expanded.

As an OpenTracing implementation, Jaeger may be useful even without the web UI visualization merely to provide a conduit to storage. Support for the latest version of ElasticSearch 7 is just now arising[4] but only ES 5 and 6 are officially supported as of this writing. Cassandra is currently not considered in this note but is also supported.

[4] `https://github.com/jaegertracing/jaeger/issues/1474`

5.4 Elastic Stack

The Elastic Stack comprises a suite of components that span most of the observability aspects with the notable exclusion of *instrumentation*. However, given the functionality of LogStash, many forms of instrumentation can be connected. The stack is sometimes called ELK after the initials of the primary components:

ElasticSearch a noSQL JSON-based database

LogStash a service multiplexer/transformation/demultiplexer

Kibana a web-based visualization and UI to ElasticSearch

ElasticSearch is a database of documents provided in JSON format. A document is added to an "index" which is roughly similar in concept to a "database" in an RDBMS. A "mapping" from JSON object attributes names to data types (which may have structure) may be optionally specified and may be thought of as roughly corresponding to a table schema in an RDBMS. Prior to ElasticSearch 7 an index may hold a number of "document types" each with a distinct mapping. Current ElasticSearch 7 allows only a single document type (called `_doc`) for each index. As mappings are specified based on object attribute name this restriction may only matter for documents with different types for the same attribute name. These details are only required if the DAQ group develops code which directly provides input to ElasticSearch. At some cost, this option should nonetheless be considered as it could negate the need for substantial middleware such as LogStash described next.

LogStash provides central broker system with many powerful features. Essentially it accepts data from a large variety of types of sources, applies highly configurable transformations, and emits output to a large variety of sinks. Inputs of note include ElasticSearch, Graphite, HTTP (push or poll), pipe, stdin, syslog, tcp/udp. Outputs of note include ElasticSearch, Graphite, HTTP, pipe, stdout, syslog, tcp/udp as well as human oriented sinks such as email and IRC. Of note is that some inputs are also outputs allowing for cycles through the LogStash transformation. LogStash can enact transformations on its data by applying filter plugins through a configuration language (which is reminiscent of JSON but differs). Filters consist of low level functions such as event or metric aggregation, general (`grok` function) and specific (XML, JSON) parsing.

Kibana is a web-based UI for visualizing data held in ElasticSearch and provided through other components in the ELK ecosystem. It presents "dashboards" filled with various types of plots drawn from the data in elastic search. It allows browsing data and creating plots through the web UI. The "flights" demo data in Kibana 7.3 is shown below.

The ELK stack is implemented in Java with the memory and CPU usage that entails. ElasticSearch 7 has rewritten parts to replace a Java implementation of Ruby with pure Java and there are claims this reduces the resources required. With the demo "flights" data loaded, ES 7.3 uses 1.7 GB resident and 18.4 GB virtual. At idle, LogStash requires 1 GB resident, 10 GB virtual and 10% CPU.

5.5 `fluentd`

`fluentd`[5] provides a central broker system with seven types of plugins: input, parser, filter, output, formatter, storage and buffer. The main `fluentd` application require substantially less memory (at idle, 30 MB resident, 53 MB virtual, less than 1% CPU) than LogStash in part to many of the

[5] `https://www.fluentd.org/`

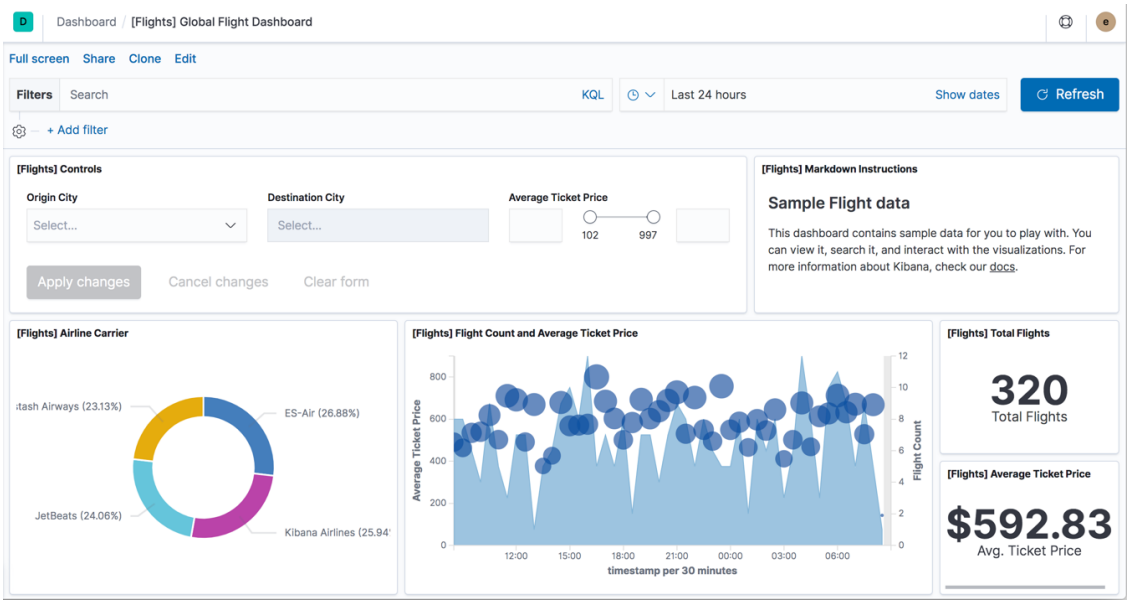


Figure 2: Sample dashboard showing Kibana taken from `https://www.elastic.co/guide/en/kibana/7.3/images/tutorial-sample-dashboard.png`

plugins being implemented in C/C++. The plugin API ("Fluent Bits") is available for developing user plugins.

input syslog, TPC/UDP, pipe as well as application level input from PostgreSQL logs, GlusterFS. Plugins provide input of operating system level information such as CPU, RAM, disk usage, network traffic and syslog. And JSON messages can be input over TCP or stdin.

parser regular expressions, JSON parser, syslog, Apache and nginx logs.

filter a `grep` like matching, a general record transformation, and application of parser plugins.

output ElasticSearch, InfluxDB, files, stdout MySQL and PostgreSQL, Graphite and ZeroMQ PUB sockets as well as human oriented systems like Email and IRC.

formatter JSON and msgpack.

buffer actually provides aggregation over time and can buffer in memory or file.

storage a single storage plugin exists to save data to a JSON file. Note, output plugins are more likely useful.

5.6 Graphite

Graphite[6] provides a time-series database and web-based UI for visualization of the data. Data is ingested through a component called Carbon. The primary backend storage is named Whisper which is a fixed-file-size, round-robin format. The web-UI based visualization is named graphite-web and queries the time-series database to produce plots. An example is shown below.

Graphite is meant to hold metrics and not spans nor traces and not particularly events. It has a trivial means to ingest metrics. Input line-oriented and formatted as:

[6] `https://graphiteapp.org/`

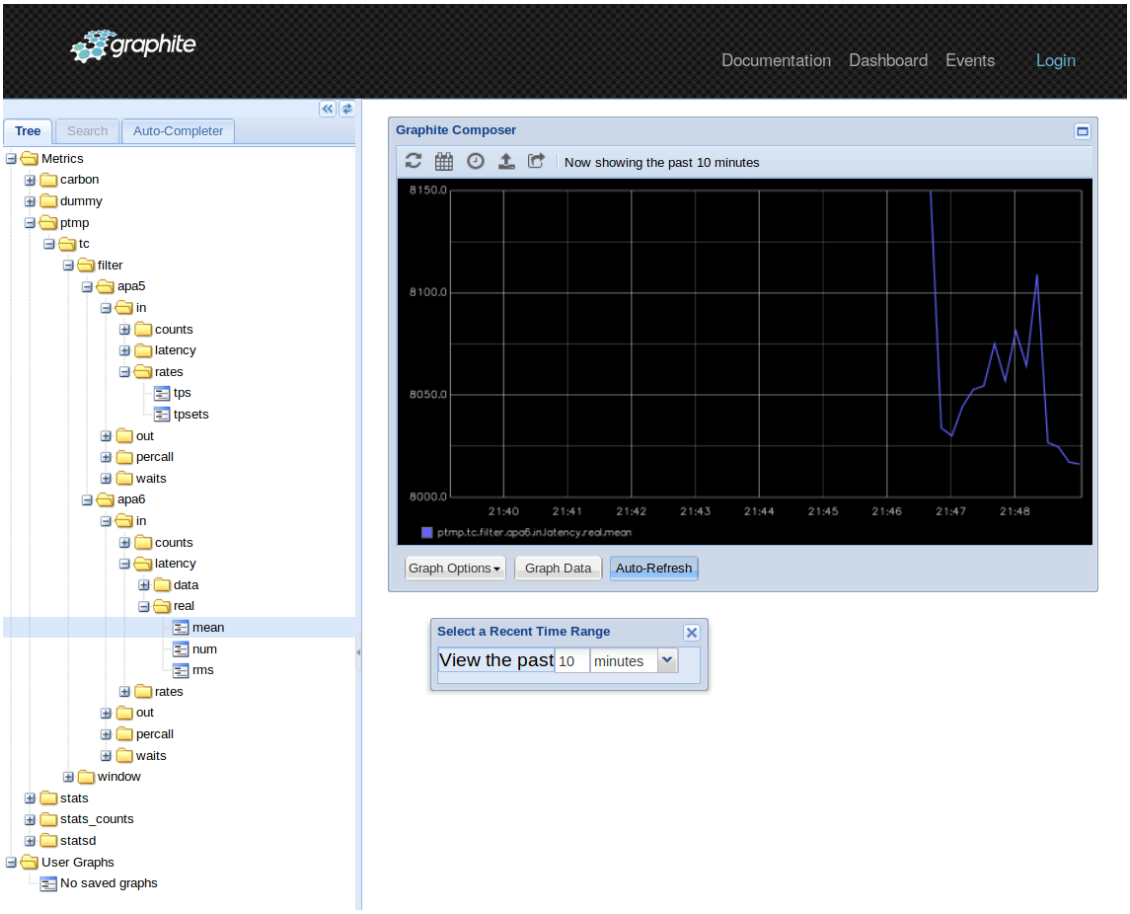


Figure 3: Example screen shot of Graphite-web displaying a metric from PTMP as part of Proto-DUNE self-triggering run in "replay" mode.

```
dot.separated.metric.name <numeric-value> <unix-epoc-seconds>\n
```

Input may be sent directly to a TCP socket. TCP connections may be opened for a single send or may be kept open for multiple sends. UDP may also be used. Single or multiple lines may be sent together. More sophisticated input is supported including AMQP and Python pickle serialized data however the line-oriented text is simple and general enough for practical use.

As in the example, the name of a metric is dot-separated. Thus, in aggregate, metrics are located in a hierarchical name space. As described next, this name space is directly exposed to using Graphite from Grafana. It frees the data source from having to have a predefined schema for its metrics. Exploiting this flexibility can however lead to problems as obsolete metrics can be seen as polluting the name space. Removal of metrics require removing the underlying Whisper files.

The simplicity, generality and low resources required of Graphite (about 50 MB resident and virtual, no CPU at idle) are its main selling points as are its ease of use with Grafana. As Whisper is a fixed-sized round-robin database, it should not be relied on for long-term archival storage.

5.7 Grafana

Grafana[7] provides web-UI based visualization for time-series data. It has plugins supporting a variety of data sources as well as for defining visualization dashboards. Links to specific views of the data can be created which allows for easy sharing. Entire views can be uploaded as JSON to servers to allow capturing of information in a form that lets other have some level of interactivity. An example visualization is shown below.

Input plugins of note include Graphite, Elasticsearch, InfluxDB, MySQL, PostgreSQL (including JSON entries), Prometheus as well as direct JSON over HTTP.

5.8 InfluxDB

InfluxDB[8] provides a time-series database with a query language. Writing directly to the DB uses an HTTP API. In addition a broker called Telegraf provides an input adapter. Inputs of note include PostgreSQL and syslog.

5.9 Prometheus

Prometheus[9] is a time-series database which supports metrics as sets of named values ("multi-dimensional" and similar to an n-tuple) and a metric query language. Metric samples are interpreted according to their type: counter, gauge, histogram or a "summary" (which might be better named a "statistic").

Metrics are ingested by Prometheus core by querying (called "scraping") a data source. Sources of data are called "exporters". A "push gateway" is provides an exporter that forwards data which has been pushed to it via HTTP. The project provides a Python library to provide an exporter interface and a community developed C++ library exists.

[7] `https://grafana.com/`

[8] `https://www.influxdata.com/products/influxdb-overview/`

[9] `https://prometheus.io/`

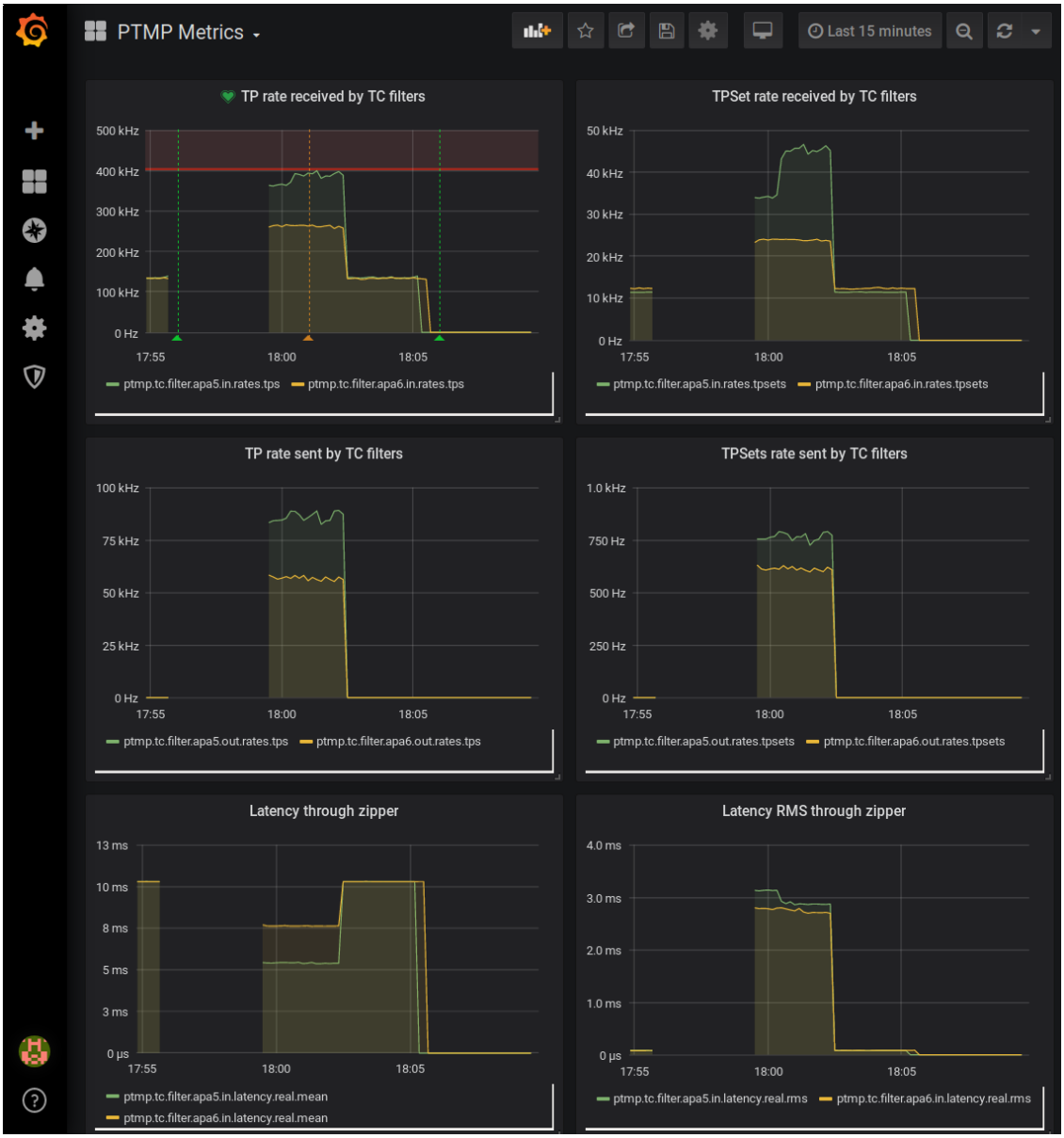


Figure 4: Grafana screen shot showing example visualization of PTMP metrics from ProtoDUNE self-triggering produced through file replay. The upper left visualization includes an example of an alert level defined.

5.10 Comparisons

5.10.1 LogStash vs fluentd

5.10.2 ElasticSearch vs PostgreSQL vs Graphite

5.10.3 Kibana vs Grafana vs. Graphite-web

6 Prototype Architecture

This final section proposes a architectural sketch to initiate a potential plan towards a design and implementation of an observability system for the DUNE FD DAQ. The final system is expected to differ from the plan presented here. Rather, the goal of the proposal is to show one possible example of how something might be put together. The proposal is summarized in the following figure.

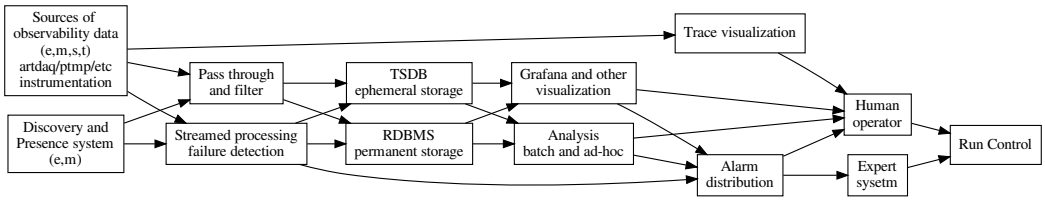


Figure 5: Sketch of potential elements of full observability system for DUNE FD DAQ. Nodes are major components and edges represnet the flow of observability data objects (events, metrics, spans and traces) and derived objects.

Walking through the directed graph starting its inputs, existing and new code will have instrumentation that will produce events, metrics and spans from the artdaq, PTMP and other software layers and applications. The operation of the "state machine" of artdaq is a natural place to generically implement events and spans. As discussed above, PTMP is currently used in a domain that does not fit the model for producing spans, as defined but can produce events and metrics which can be softly correlated based on hardware time.

A source of "meta" events and metrics is provided by the discovery and presence system. This will provide input about components of the DAQ appearing and disappearing on the network. It will augment similar information that is explicitly sent by the components themselves but will provide a backup monitoring for when those components fail in a way that their own notifications are not produced.

The next layer involves streamd processing of observability data objects as well as pass-through and filtering. The processing will execute algorithms which attempt to immediately detect abnormal or erroneous behavior in order to raise alarms. Both nodes in this layer will output information to storage in the next layer. Pass-through and filter will immediately send select observability data directly to storage while streamed processing will send derived information.

The storage layer consists of an RDBMS for lower-rate, permanent storage of data and a TSDB (eg Graphite) for higher-rate ephemeral storage. Selecting, specifically, PostgreSQL for the RDBMS would allow observability data and its derivatives to be stored under both schema and schemaless forms.

In any case of storage technology, new data sent to storage is promptly (well under one second) is available for use either in visualization systems such as Grafana or analysis jobs that execute in

a batched or ad-hoc basis. Both may produce alarms.

Alarms are centrally aggregated into a distribution system. Other sources of alarms other than what are shown may be excepted (eg, from slow control or sources external from the DAQ). This alarm distribution acts as a pub/sub broker. It gives a common point for many providers to send alarms and a common point for many consumers to subscribe to alarms of interest.

Two main classes of alarm consumers will be an expert system and human operators. Humans operators will also take input from visualization as well as the results of analysis jobs. Both may then instruct changes to DAQ via a Run Control interface.