PTMP Metrics

Brett Viren

Physics Department

BROOKHAVEN
NATIONAL LABORATORY

DUNE FD DAQ DFWG – 20 Nov 2019

PTMP in a nutshell

*A **network*** of trigger message passing components.*

Includes:

- Component library with CLI or for embedding (eg into *artdaq*)
- End-user configuration mechanism
  - **individual** component configuration,
  - **aggregation** of components into processes and
  - **connection** of components into a network.
- Optimized message processing algorithms
  - window, zipper, filter, query
- Extensible at run time
  - shared library plugins and dynamic factory object construction.
  - trigger algorithms incorporated as filter "engines" via this mechanism.

*Includes transport over TCP/IP, Unix domain sockets and cross-thread shared memory.

A post-hoc realization

*PTMP **is** a metrics system*

- The "system" being observed is LArTPC activity.
  - As represented by collection waveform streams.
- The "metric" message is a `TPSet` object.
  - Indicates that "something interesting" happened in the LArTPC.
- PTMP components are metric processors, aggregators.
  - As a whole, "self-triggering" is an expert-system for "anomaly" identification.
  - Including subsequent readout: automated anomaly response!

But, PTMP also has explicit metric messages to provide "observability" of its own operations.

But PTMP is not a **general** metric system*

*(it does however implement a fairly general metric subsystem)

- Now, PTMP has only one, monolithic message type in the system (`TPSet`).
  - Good for self-trigger use, but little else.
- What should be generic PTMP code is forced to be type-specific.
- Also, there are desires to migrate to new, richer message schema.
- Solution: factor and extend `TPSet`

  header — schema version (already there), payload ID, a representative time and detector ID, a message sequence number.

  payload — "application" data which is passed-through with no serializing most PTMP algorithms.
- PTMP, itself, will then be a fairly generic **metric**-passing system.

PTMP's software technology

- **ZeroMQ** ecosystem provides the basis for PTMP
  - **`libzmq`** is used for communication patterns and message transport.
  - **CZMQ** for simpler interface to `libzmq` and useful software patterns: **actor**, **reactor**, **poller**. Can provide **auth/auth** if/when needed.
  - **zproto** provides model-oriented protocol definition, used for the client/server part of the `TPSet` stream "query" component.
- **Protobuffers** used to define and serialize `TPSet` objects.
- **JSON** and `nlohmann::json` for configuration and metric serialization.
- **CLI11** for command line interface handling
- Built in `upif`, small plugin/factor method adapted from **Wire-Cell**.
- **Python** for various support modules, CLI scripts
  - Can implement `TPSet` processing nodes in Python.
- **Jsonnet** provides human-oriented configuration language
  - (optional, uses via CLI and Python module)
- **shoreman** for launching groups of related processes (mostly for tests).

Note: only libzmq, CZMQ and protobuf are "external" dependencies.

ZeroMQ features relevant to metrics
(and as used in PTMP)

- Communication pattern variety, all asynchronous N-to-M

PUB/SUB	one-way, send-to-all, PUB not delayed by slow SUB
PUSH/PULL	one-way, round-robin send, block on back-pressure
DEALER/ROUTER	two-way, round-robin send, directed reply
(REP/REQ)	(unused in PTMP, simple, synchronous send/receive)

- Transport variety

`inproc`	shared memory, thread-safe
`ipc`	Unix domain sockets (FIFO files)
`tcp`	TCP/IP network

- Configurable (no recompile) communication patterns and transports.
- Robust connections (endpoints can come/go)
- Distributed **discovery and presence** mechanism (Zyre)
  - Uses mix of UDP broadcast + TCP to discover and update peers.
  - Unlike "name services" there is no single point of failure.
  - Can also follow "service" pattern and with multiple redundant services.

PTMP features relevant to metric systems

- Configuration of components and whole system
  - Easy to insert new sources/sinks of metrics.
- File dump and paced replay
  - Useful for offline developing/testing of new metrics sources and processing.
- Stream query
  - Readout of recent messages (eg, supporting *artdaq* duties).
  - Prompt processing of metrics (eg, supporting an expert-system).

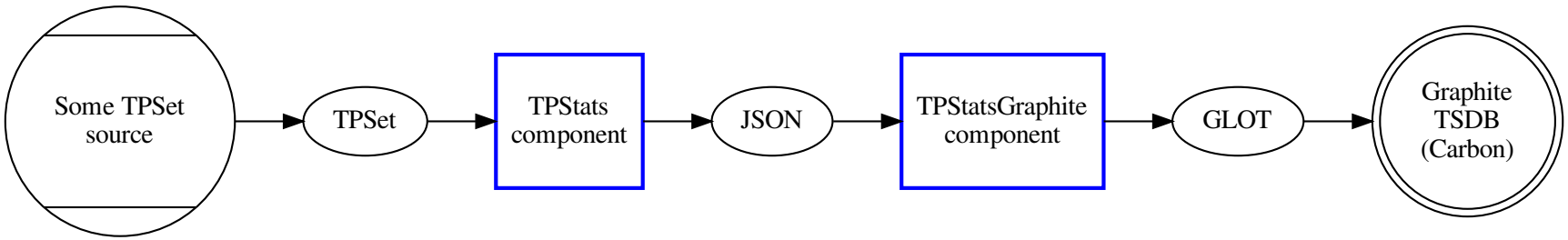
PTMP has two types of explicit metrics

- `TPStats` emits metrics about a `TPSet` stream.
- `ptmp::metrics::Metrics` emit arbitrary structured data from points throughout code.

Two source types, but with some "coherency":

- Same message formats used by both.
- Some "crossing of streams" supported.

TPStats

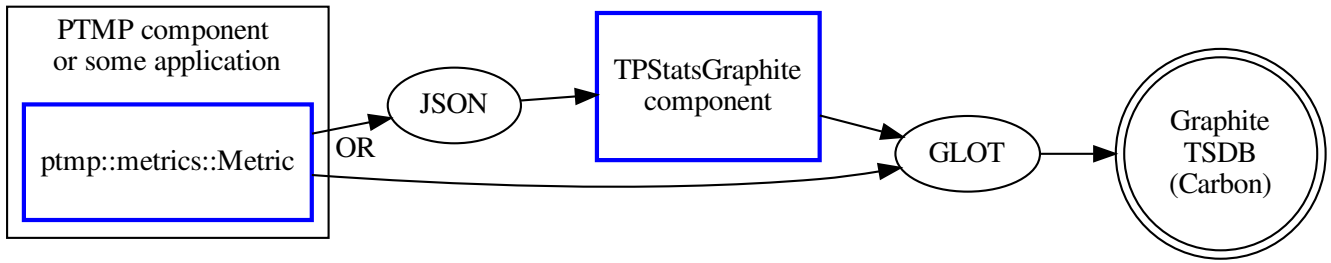


- The `TPStats` is a PTMP component object.
- Sinks `TPSet` stream, source of a summary metrics with content like:

times	received and created clock times, data times (`tstart`)
counts	`TPSets`, `TrigPrims`, bytes, skipped seqno, channels
rates	`TPSets`, `TrigPrims`, bytes, skipped seqno
ADC	per `TPSet`, per `TrigPrim`, rate, mean
latency	mean/rms/min/max comparing created/received and `tstart`/received times

- Output as ZeroMQ message with structured payload serialized with JSON
- `TPStatsGraphite` converts JSON to "Graphite lines of text" (GLOT) and may connect directly to a Graphite Carbon ingest socket.

ptmp::metrics::Metric



- C++ class fronting a ZeroMQ socket, flexible object lifetime.
- Presents a "logger" type interface but for structured data.
- Socket configured via usual PTMP mechanism.
- Message payload serialized directly to JSON or GLOT format.
  - `nlohmann::json` used for JSON, built-in GLOT support.
  - GLOT may use ZMQ STREAM socket, connect directly to Graphite/Carbon.
  - JSON/GLOT messages "located" under configurable structure prefix.
- Metric message is sent out immediately on call.
  - Can send individual scalar values or a composite structure.
- `ptmp::metrics::Metric` is independent from `TPStat`
  - But `ptmp::metrics::Metric` can send JSON to `TPStatsGraphite`.

`ptmp::metrics::Metric` usage example

```
void some_function(...) {
  std::string met_cfg = ...;
  ptmp::metrics::Metric::Metric met(met_cfg);

  while (...) {
    int something = ...;
    float other = ...;

    // Whole structure
    met({"something":something, "other":other});

    // One-shot scalar
    met("something", something);
  }
}
```

This example creates a `Metric` on the local stack. May also pass in prebuilt metric object or hold one as class member. A ZeroMQ socket lifecycle follows the `Metric` object so don't construct deep inside some fast loop. Each call is a `send()` so best to use "whole structure" rather than many "one-shot scalar" calls. 0.1-1.0 MHz message rate is achievable, see DocDB 16976.

Comments on Docker

- A Docker container is available for building and running PTMP, used by Travis-CI to test each PTMP commit.
- A `docker-compose.yml` file is available which brings together **Graphite** and **Grafana**
  - Easy, useful setup to see "live" results while developing new metrics.
- Independent of metrics, I think container usage is a good development $\rightarrow$ production deployment.
  - Usual Dev/Ops benefits like quick roll back of production "oops", documentation, reproducible, offline testing, development in "real" production environment.

Some Possible Next Steps

- Develop a "standard" but **general** DUNE metric system.
  - A "light-weight", independent (low-dependencies) core support library.
  - Applications in C++, Python CLI/`bash`, avoid barriers for other languages.
  - Standardize message schema, express as high-level, general model.
    - "moo" package: Jsonnet → protobuf/GraphViz playground
    - I will follow similar approach for PTMP migration to a v1 schema.
- Start thinking about **metric-consuming** applications.
  - "AI" / expert systems to diagnose source of problems
    - Leverage/reimplement ATLAS' BDT-based work?
  - Fast queries on recent metrics (PTMP `TPQuery`, ELK?, PipelineDB?)
  - Converters of metric streams from external sources (eg, slow control)
  - Sink converters (databases, email/SMS, Elog)
- Overall "observability" system(s).
  - See DocDB 16973 for a work-in-progress note.

CCM: thoughts on going beyond M

- Both **monitoring** and **control** are hierarchical systems
  - monitor A **fan-in** processing hierarchy, messages say what **has** happened
  - control A **fan-out** processing hierarchy, messages say what **should** happen
- **Monitoring** is rather connected to **control**.
  - Need connections at many points **between** the two hierarchies.
  - Solve local problems locally (and automatically if possible).
- Generalizing PTMP patterns is **one** solution for the **fan-in** problem. The **fan-out** requires at least one additional feature: **routing**.
  - High-level commands need interpreting into lower level commands and finally into some real-world action.
  - Interpretation depends on where in the hierarchy.
  - Interpretation logic is best defined by the end-user, not hard-coded.

I think we should read about SMI++ and discuss.

Some related links

`ptmp`	core package `https://github.com/brettviren/ptmp`
`tpquery`	TP buffer / streamed query `https://github.com/brettviren/tpquery`
`ptmp-tcs`	integration of external TP filter algorithms code `https://github.com/brettviren/ptmp-tcs`
`ptmp-docker`	docker and docker-compose files `https://github.com/brettviren/ptmp-docker`
PTMP CI	Travis-CI build status of PTMP `https://travis-ci.org/brettviren/ptmp`
`zperfmq`	ZeroMQ performance benchmark `https://github.com/brettviren/zperfmq`
`moo`	Model oriented objects playground `https://github.com/brettviren/moo`