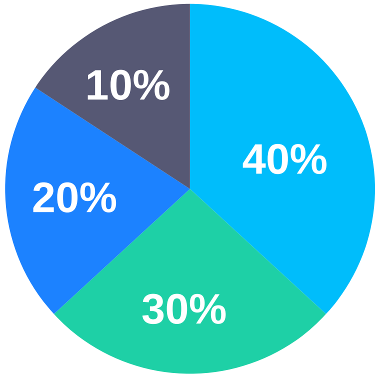
ProtoDune Operational Monitoring Roadmap

Lola Stankovic, technical student at EP-DT-DI department
CERN, 7.10.2020

Overview

- **Operational Monitoring for ProtoDUNE**
- **Target - first prototype and implementation**
  - Architecture and design
  - Visualization interface
- **Evaluation of different storages**
  - InfluxDB and VictoriaMetrics
  - Evaluation based on tests and results
- **Plans**
  - Functional Operational Monitoring library

What is the aim of operational monitoring?



Real-time analytics and clear picture of how our applications and infrastructure are working

More-appropriate strategies and understanding where improvements need to be made.

Following performance characteristics over time

Detecting issues before they impact infrastructure – workflow failures

Proposal of the ProtoDUNE monitoring system

- **Backend monitoring cluster**
  - Prometheus
  - Long term storage
    - InfluxDB, VictoriaMetrics

- **Visualization tools**
  - Grafana
  - Dashboards embedded in custom websites

- **Publish/Subscribe system** (if needed)
  - NATS, Kafka

- **Monitoring Library of APIs for providing monitoring support**

Prometheus

Prometheus is an open-source **monitoring and alerting toolkit.**

Time-series based numerical data

All data is stored as time series

Great visualization

Prometheus expression browser, Grafana integration and a console template language

Many client libraries

Easy instrumentation of services

Precise alerting

Alert Manager

Many integrations

Existing official and non-official exporters and remote endpoints and storages

Prometheus - use case

- **Prometheus software** is used to pull metrics very efficiently and is very easy to run and maintain in large deployments.

- not meant for long-term storage

- **Various options for the remote storage solutions for Prometheus**
    - Cortex
    - InfluxDB
    - Thanos
    - VictoriaMetrics

InfluxDB

- **Open source time series database**
  - fast, high-availability storage and retrieval of time series data
  - InfluxQL - the InfluxDB SQL-like query language with built-in functions to easily query the data
  - plugins support for other data ingestion protocols such as Graphite, collectd, and OpenTSDB
  - simple, high performing write and query HTTP APIs.

- **The InfluxDB open source version is free**
  - cluster solution is available in commercial enterprise version

VictoriaMetrics

- **The following versions are open source and free:**
  - single-node version
  - cluster version

- In some benchmarks it outperforms InfluxDB, TimescaleDB by up to 20x

- It supports Prometheus Querying API

- Can be used as Prometheus replacement in Grafana.

- **VictoriaMetrics/InfluxDB use-case:**
  - backend storage for Prometheus server
  - building Grafana dashboards using TSDB as datasource
  - querying data via HTTP APIs

Remote storage comparison - evaluation

- **Benchmarking tests:**
  - Setting InfluxDB / VictoriaMetrics on host server
  - Configuring Prometheus to remotely write data to InfluxDB/Victoria Metrics
  - Running Avalanche

- Avalanche is a simple metrics generation tool that can be used to test metric ingestion throughput. Before running the Avalanche, Prometheus configuration file had to be changed and set to scrape the Avalanche.

- Avalanche was then run setting flags for generating time-series workload.

CONFIGURATION FLAGS:

--metric-count=500	Number of metrics to serve.
--label-count=10	Number of labels per-metric.
--series-count=10	Number of series per-metric.
--metricname-length=5	Modify length of metric names.
--labelname-length=5	Modify length of label names.
--value-interval=30	Change series values every {interval} seconds.
--series-interval=60	Change series_id label values every {interval}

Remote storage comparison - evaluation

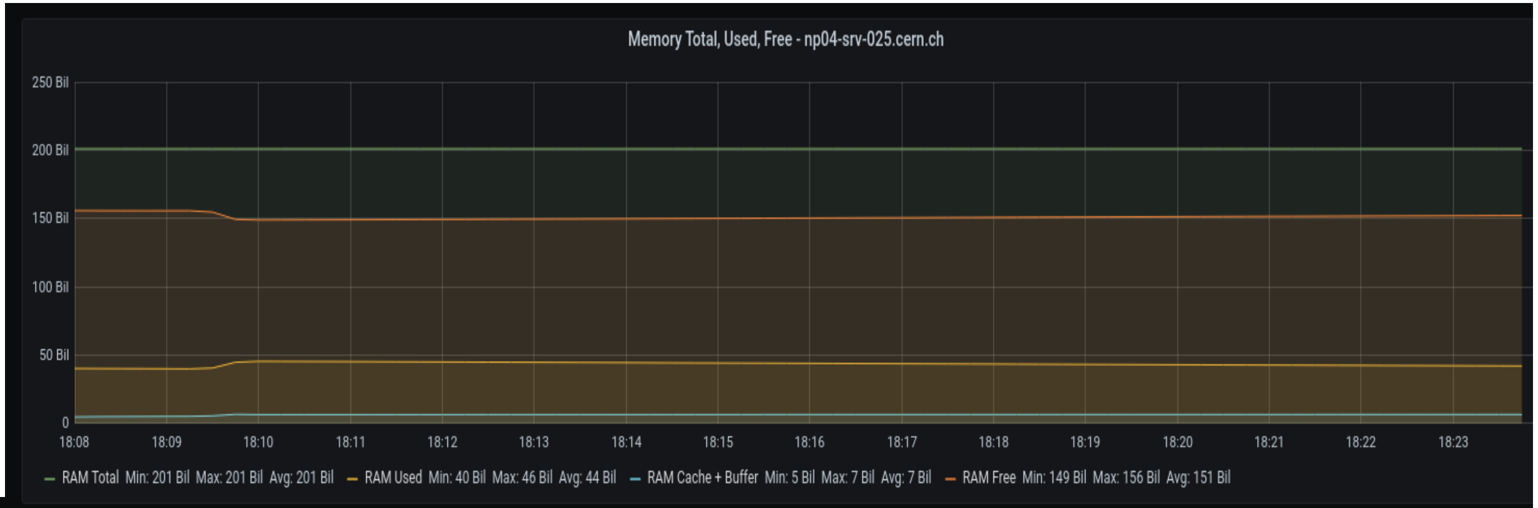
In the assessment, we had deployed Prometheus 2.17.1 version on NP04 machine.
The observations below are based on setting InfluxDB and VictoriaMetrics as Prometheus remote storage and then running Avalanche tool for generating big ingestion workload.

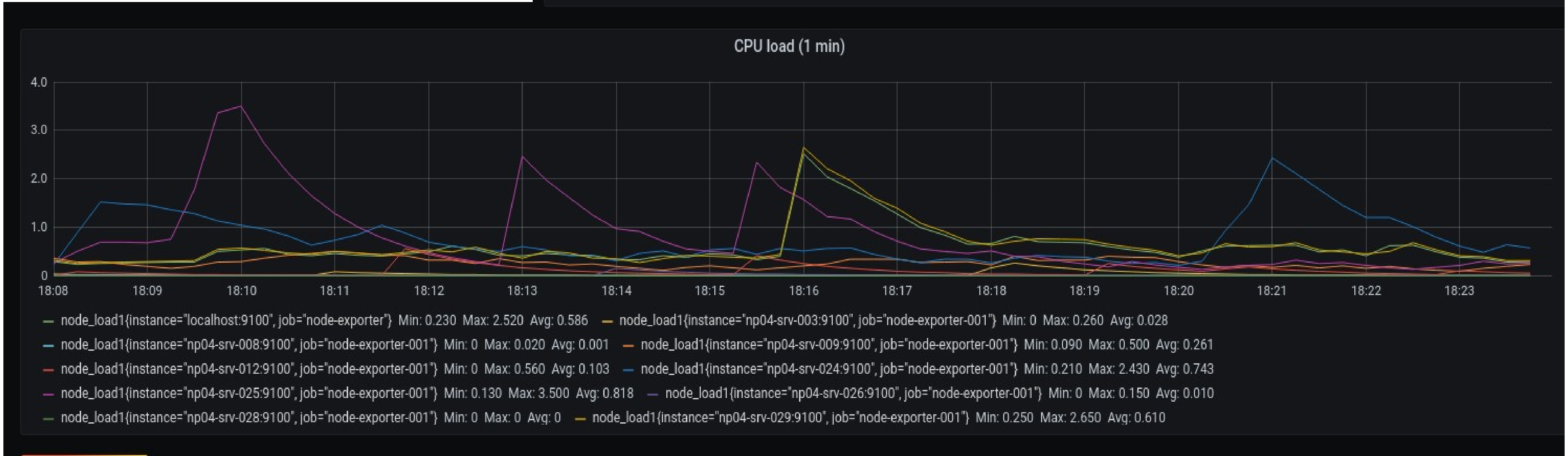
WORKLOAD INGESTION:
metric count:50 000
series count:10
500K TS
scraping interval:15s, 33.33K samples/s

	InfluxDB	VictoriaMetrics	VictoriaMetrics-different configuration
Prometheus CPU usage	1.46	0.85 cores	1.21cores
Prometheus Memory usage	27.23GiB	24.12GiB	44.21GiB
Node-exporter* RAM used	43.5Bil	32.8Bil	34.6Bil
Node-exporter* CPU load (1 min)	0.818	0.32	0.61

Remote storage comparison - evaluation

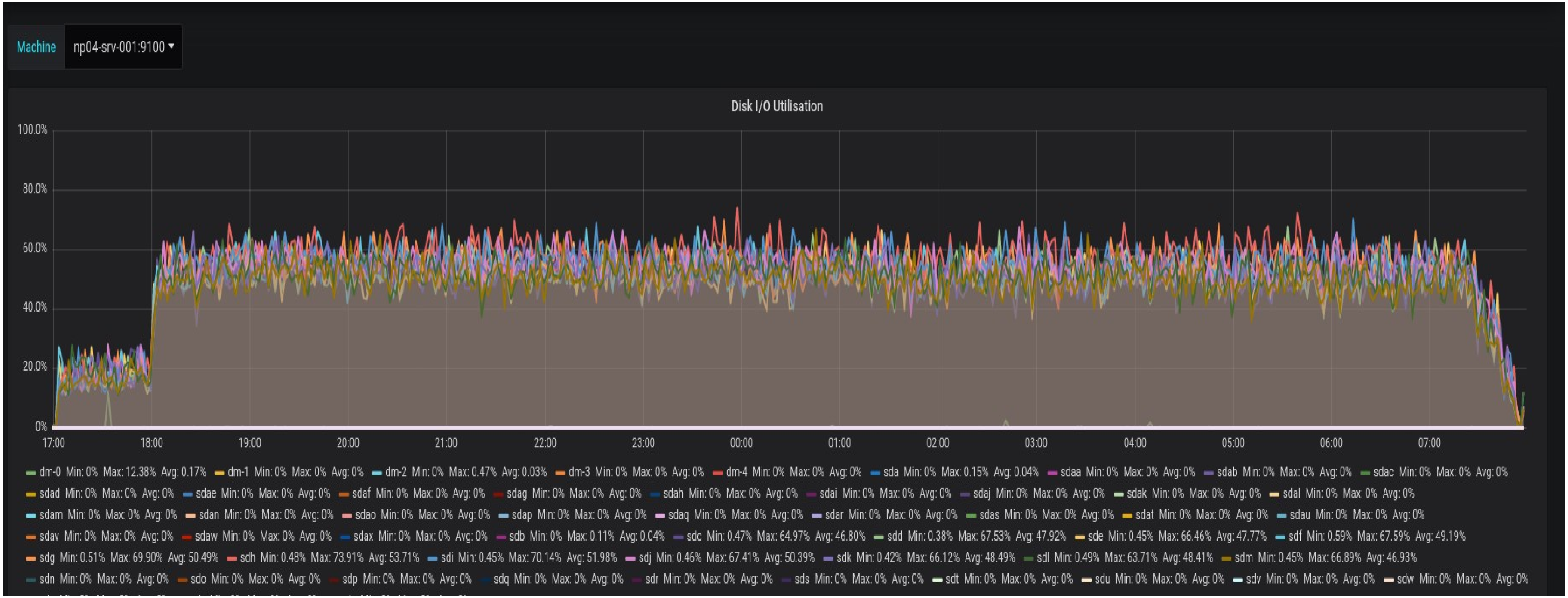
- Grafana dashboards
  - Not just stress-testing TSDB
  - Tracking system cpu and memory usage





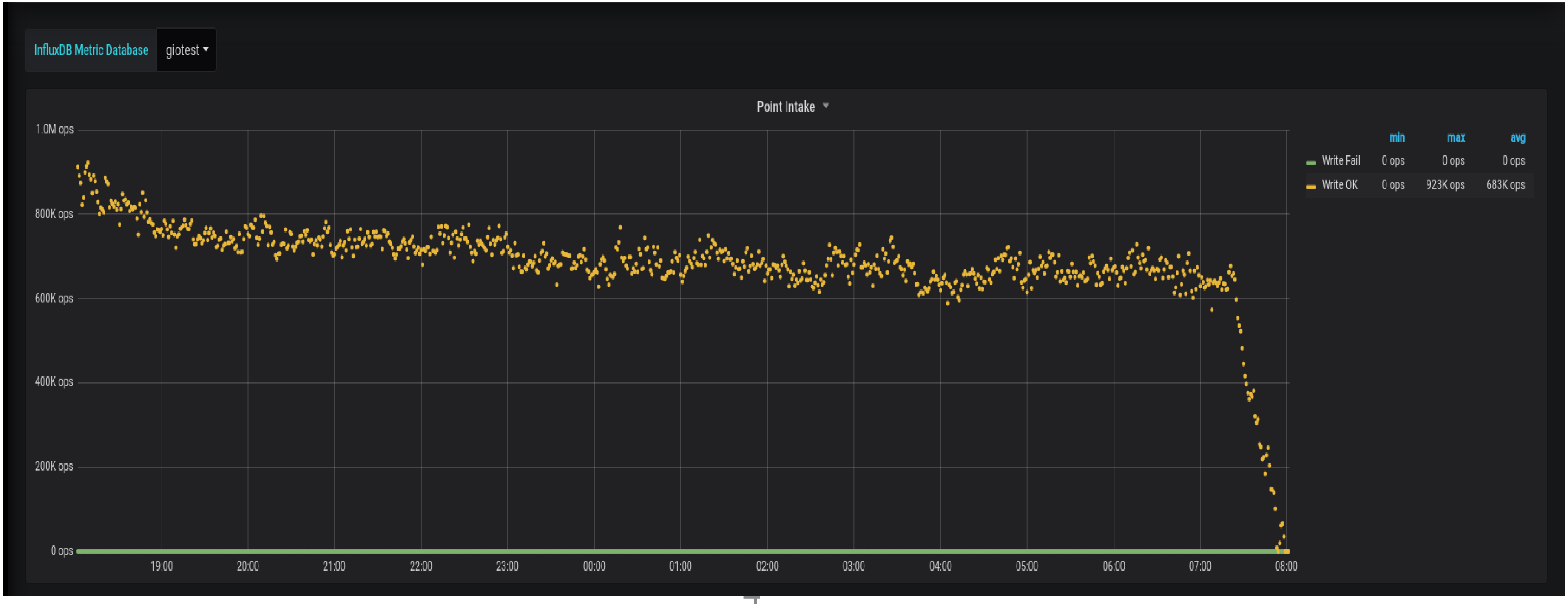
Testing InfluxDB

- Disk IO usage of Influx storage host



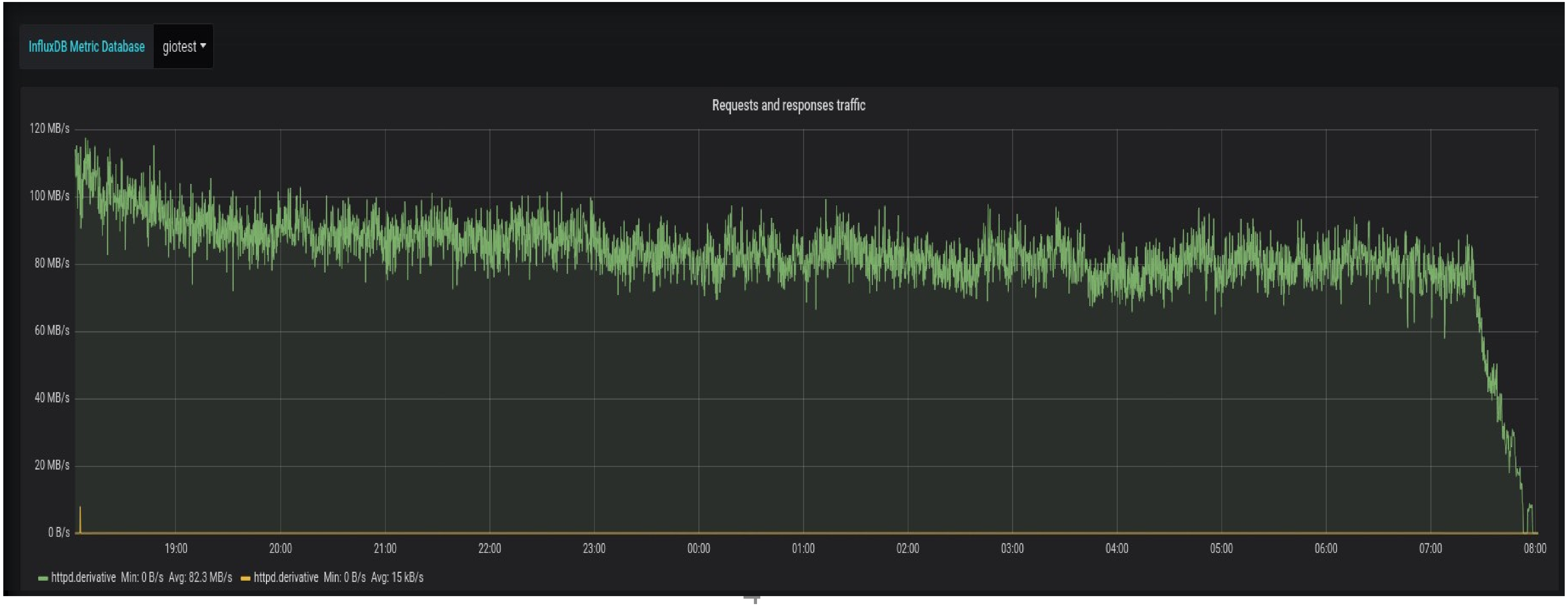
Testing InfluxDB

- Number of writing operations per second



Testing InfluxDB

- Number of requests per second during testing



Testing InfluxDB

- Representation of mean value of one stored measurement in the last 1 year

Testing Prometheus performance

To measure the performance of the native Prometheus TSDB, we have used the **ApacheBench software**.

Additional information:
Software Version
Prometheus 2.17.1
ApacheBench version: Server version: Apache/2.4.6 (CentOS)

Results on epdtdi103.cern.ch machine:

	Simple query	Query With one PromQL Function	Multiple PromQL Functions
Test Configurations	100 000 requests, 100 in parallel	100 000 requests, 100 in parallel	10 000 requests, 100 in parallel
Requests/second	110276.56	109283.77	42.76
Longest request (ms)	7	47	3209
Total test time (seconds)	0.907	0.915	233.842
Time per request (mean in ms)	0.907	0.915	2338.425
Time per request (mean, across all concurent requests in ms)	0.009	0.010	23.384

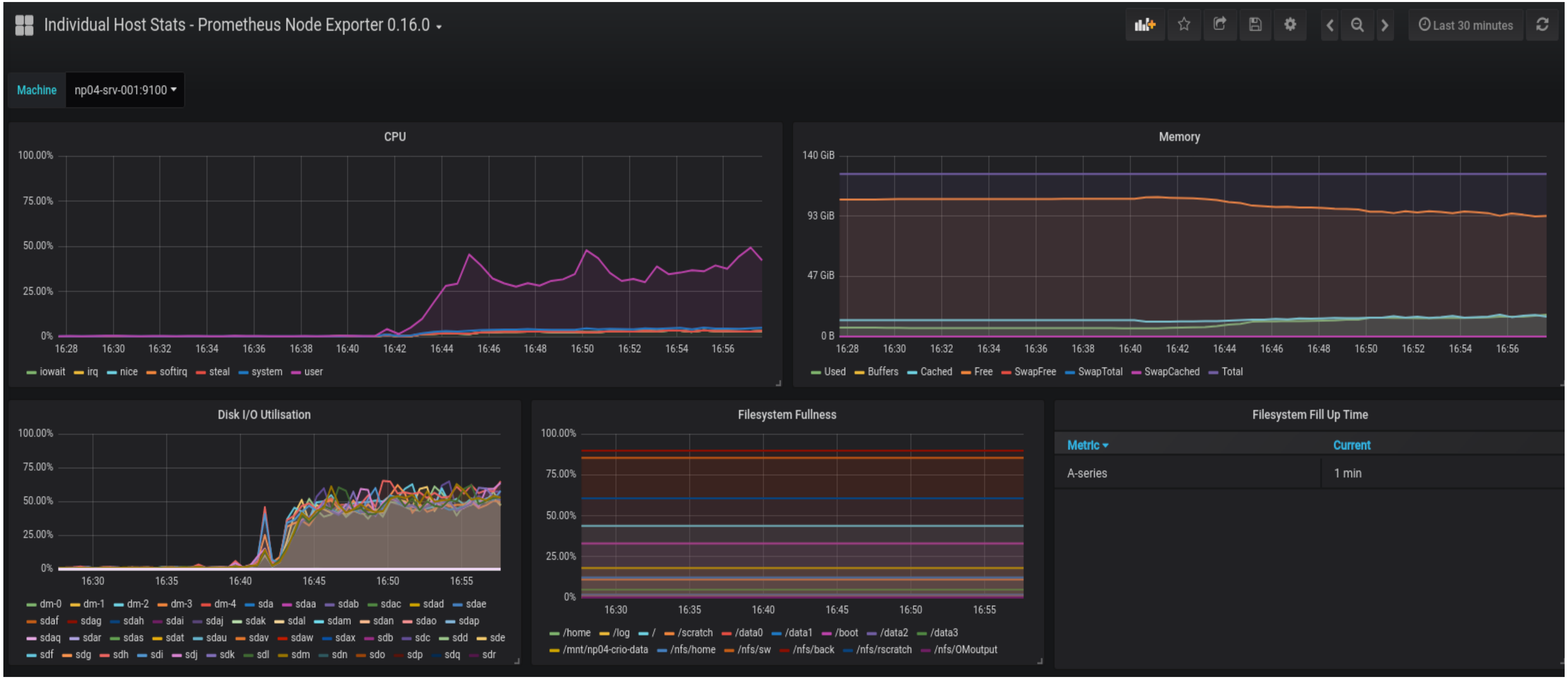
Grafana Data Visualization

- **Open-source visualization platform that allows querying, creating custom dashboards and exploring metrics from different sources**
  - tool for presenting TSDB data into meaningful graphs
  - supports both Prometheus and its long-term storage backends as a datasorce

- **Grafana custom dashboards for Prometheus**
  - dashboard graphs of Prometheus metrics with intelligent templating

- **Grafana custom dashboards for Node-Exporter**
  - tracking various machine resources such as memory, disk and CPU utilization

Grafana Data Visualization - Example



C++ Operational Monitoring Library

- **Operational Monitoring for ProtoDUNE considerations**
  - very light weight
  - generic
  - support regular collection of operational metrics

- **Solution -** Applicable C++ Operational Monitoring library
  - provide access to the API's REST interface
  - efficiently communicate with the backend
  - handle all low-level details of communication

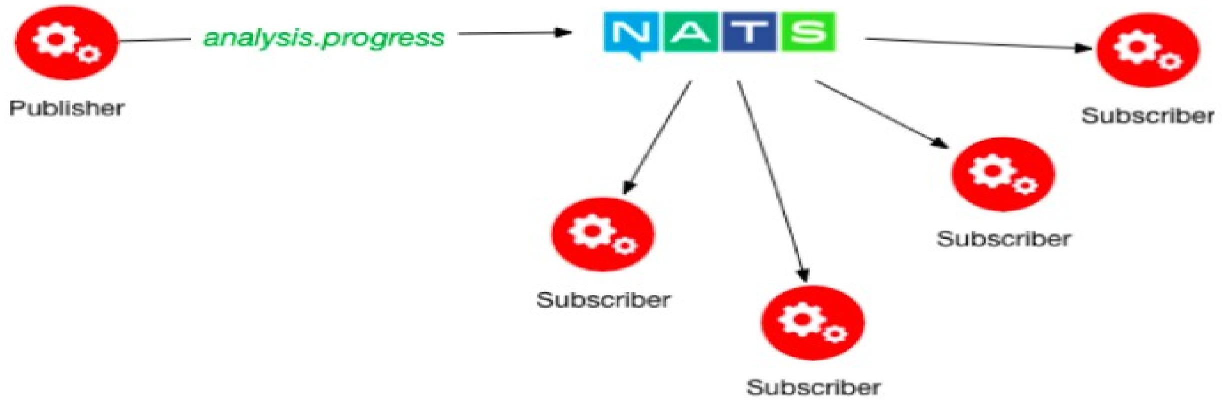
C++ Operational Monitoring Library

- **Simple design and implementation**
  - registering user-defined metrics
  - publishing metrics directly using official client libraries and by posting HTTP requests
  - data retrieval
- **Provides by default**
  - system-level monitoring for the host where it runs
  - application monitoring
- The library should be used by DAQ services and processes

C++ Operational Monitoring Library

- **Publish/subscribe system**
  - Introducing a lightweight messaging system
    - Low CPU-consuming
    - High availability
    - High scalability

- **High performance and low latency are critical**



Publish/Subscribe Pattern

C++ Operational Monitoring Library

- **The C++ Operational Monitoring Library of API's provides some basic functionality:**
  - Registering user-defined metrics
  - Publishing metrics via HTTP posts and directly using official InfluxDB Client
  - Retrieving metric values

C++ Operational Monitoring Library

- ## Metric registration

The starting point for metrics is **MetricRegistry class** that enables registering metrics using template function *registerMetric()* which accepts a metric ref. wrapper, creates a smart pointer to that wrapper and then inserts it into a map.

```
void registerMetric(const std::string& metricName, std::reference_wrapper<T> myMetric)
```

where:

- _ myMetric - reference wrapper of the metric’s value
- metricName-name of the metric that will be registered and published

The user modules will be able to register what is their metric using registerMetric() function.

For example:

```
std::atomic<float> myMetric_float{0.1};

std::atomic<int> myMetric_int(5);

MetricRegistry mman;

mman.registerMetric<std::atomic<float>>("FPGA Temperature", std::ref(myMetric_float));

mman.registerMetric<std::atomic<int>>("Humidity", std::ref(myMetric_int));
```

C++ Operational Monitoring Library

In the registerMetric() function, the metrics will be stored in a map:

metric_set.insert(std::make_pair(metricName, std::shared_ptr<MetricRefInterface>(new MetricRef<T> (myMetric))).second;}

The acceptable types of metric’s values are only of atomic type (: std::atomic<int>, std::atomic<float> ,std::atomic<double>, std::atomic<bool> , std::atomic<size_t>).

- ## Metric monitoring

MetricMonitor is a component responsible for periodically looking(scraping) at the metrics and their values and then activating corresponding publishing components. The class has following private members:

- rate- integer number set to 1 second by default
- collector_threads- vector of threads that will at every 1 second (rate limiter threshold) look at the value of metric and publish it
- stop() - operation that will stop monitoring
- monitor() - operation responsible for creating threads
- publish_metrics(std::map<std::string, std::shared_ptr<MetricRefInterface>> metrics) - operation that will threads execute by going through the map of registered metrics, taking their values and publishing them

C++ Operational Monitoring Library

Each metric is registered within a MetricRegistry, and has a unique name within that registry. The publish_metrics() function will redirect the publishing of the metrics either directly to influxDB or by sending HTTP requests. With the value and name of the metric additional flags such as the application name, host name and name of the thread responsible for publishing will be passed.

Example of the code:

```
for(std::map<std::string, std::shared_ptr<MetricRefInterface>>::iterator itr = metrics.begin(), itr_end = metrics.end(); itr != itr_end; ++itr) {

std::string metric_name= itr->first;

double metric_value=0;

//casting to std::atomic<float>

std::reference_wrapper<std::atomic<float>> value =dynamic_cast<MetricRef<std::atomic<float>>&>(*itr->second).getValue();

metric_value= (double) value.get();

std::cout<< "Metric name:" << metric_name << "\n";

std::cout<< "Metric value:" << metric_value << "\n";

metric_publish.publishMetric(metric_name, application_name, host_name, metric_value, “HTTP_request”);

metric_publish.publishMetric(metric_name, application_name, host_name, metric_value, “InfluxDB_client”);
```

C++ Operational Monitoring Library

- **Metric publishing**

The metrics can be sent to influxDB directly in a line protocol format with the support of the InfluxDB Client Library, or by posting http requests.

In this case, we will have two classes HTTPPublisher and ClientPublisher that will implement the MetricPublish interface.

For writing data points to influx, we must specify an existing database in the db query parameter. Points will be written to db’s default retention policy.

The following parameters are required:

- influxDbName - name of the influxDB database
- influxDbAddress - ip address of the host
- influxDbUrl - url used for sending data points to influxDB example: “http://localhost:80086/write?db=”+influxDbName

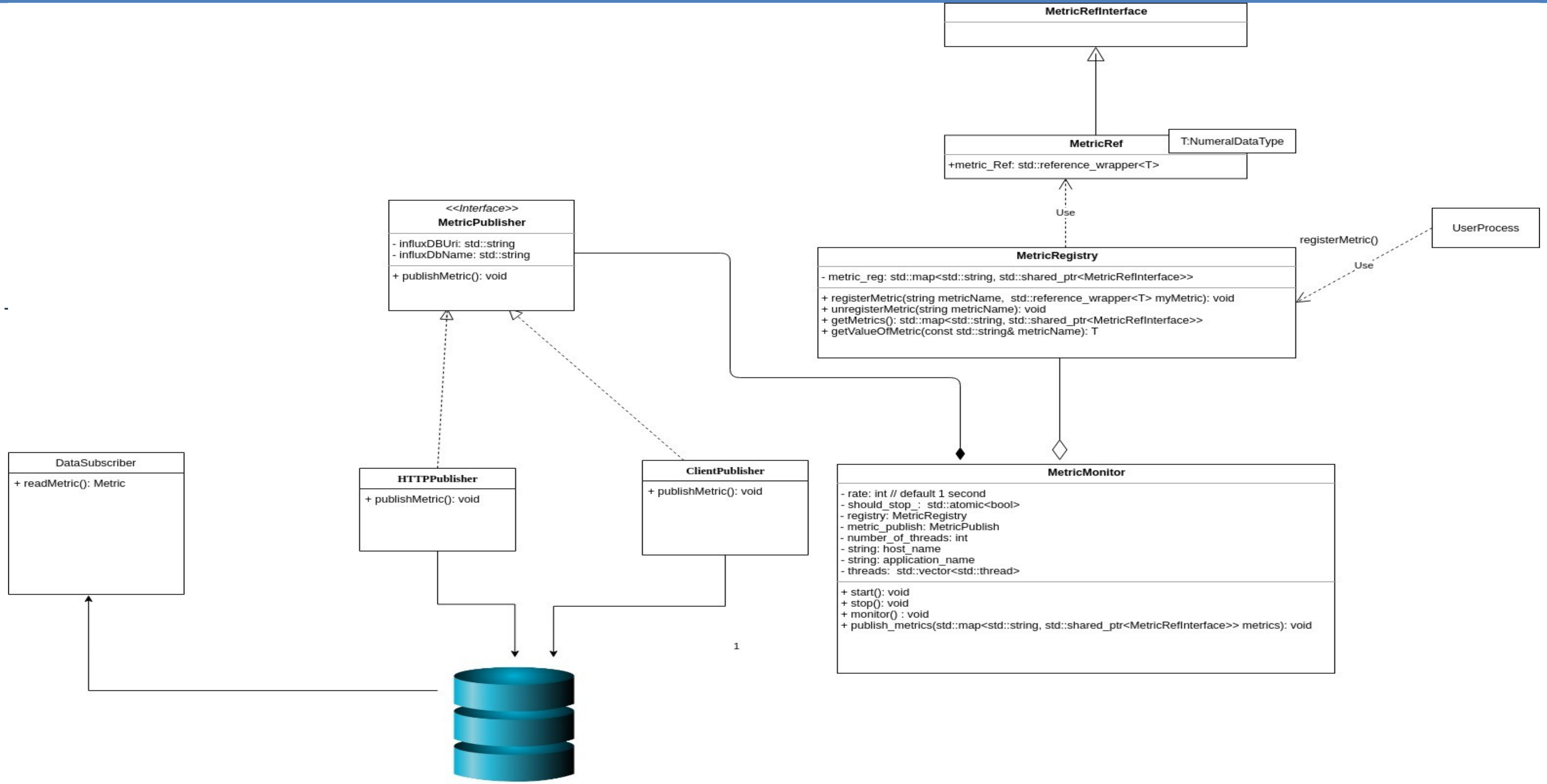
C++ Operational Monitoring Library

Example of publishing using official influxDB Client Library:

```
void publishMetric(const std::string& metricName, const std::string& application_name, const std::string& host_name,double metric_value, "InfluxDB_client"){
influxdb_cpp::server_info si(influxDbAddress, 8086, influxDbName);
influxdb_cpp::builder()
.meas(metricName)
.tag("host", host_name)
.tag("application", application_name)
.field("x", metric_value)
.timestamp(1512722735522840439)
.post_http(si);

}
```

C++ Operational Monitoring Library



Final overview

- Exploring message broker systems for publishing/subscription of metrics and choosing the best solution for Prometheus long-term storage
- Evaluating InfluxDB and VictoriaMetrics cluster performance on ProtoDUNE’s machines and comparing their performance
- The final product - lightweight operational monitoring library that can be used to monitor jobs, services and hosts

Thank you for your attention

.