Design and Implementation of a Monitoring System

Serguei Kolos,

University of California, Irvine

ISOTDAQ

International School of Trigger and Data Acquisition

What you are expected to learn from this presentation

- Monitoring is indispensable component of a complex system
- Use Monitoring System from the very beginning of a complex project:
  - This will be rewarded
- Implementing a good Monitoring System requires time and effort:
  - Just use the good API from the beginning
  - Implementation will evolve in parallel with the main project
- What is a “good API” for Monitoring System?
- How to make a simple implementation
- How to scale it up towards the final one

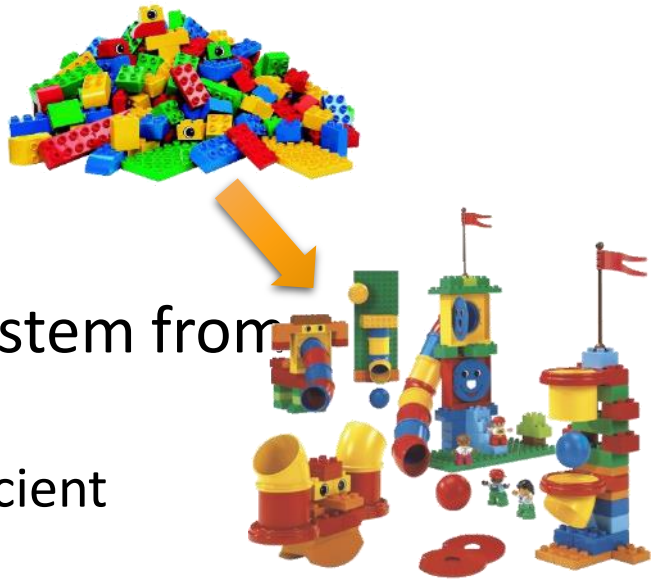
How Higgs boson discovery would have looked like in ideal world

What Happens in Reality

- A complex project has a chance to success only if it is ready to deal with problems
- Monitoring System provides the first line of defense:
  - Detects, Reports, Helps to Investigate

Can a ready-made solution be used?

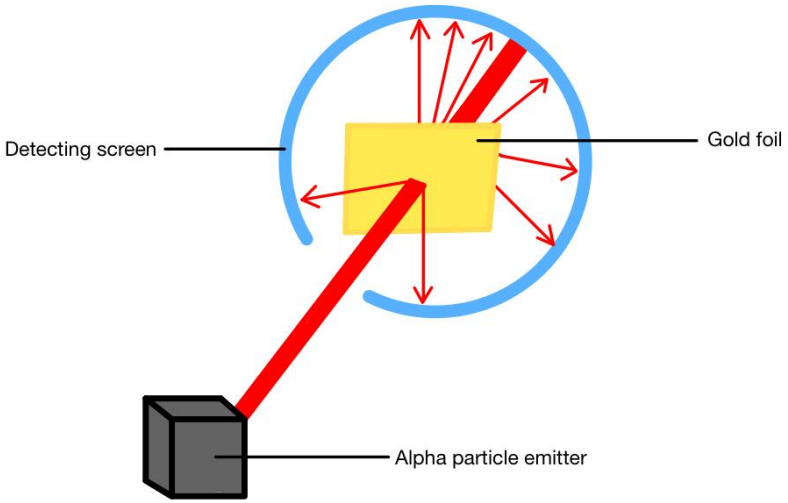
- Off-the-shelf solutions for monitoring of complex custom-built systems don't exist:
  - Fully applies to a DAQ system for physics experiment setup
- They can be constructed using:
  - Commodity tools/frameworks
  - System specific drivers/adapters
- Not worth building a Monitoring system from scratch:
  - Using existing tools is much more efficient



Two Main Approaches for Monitoring

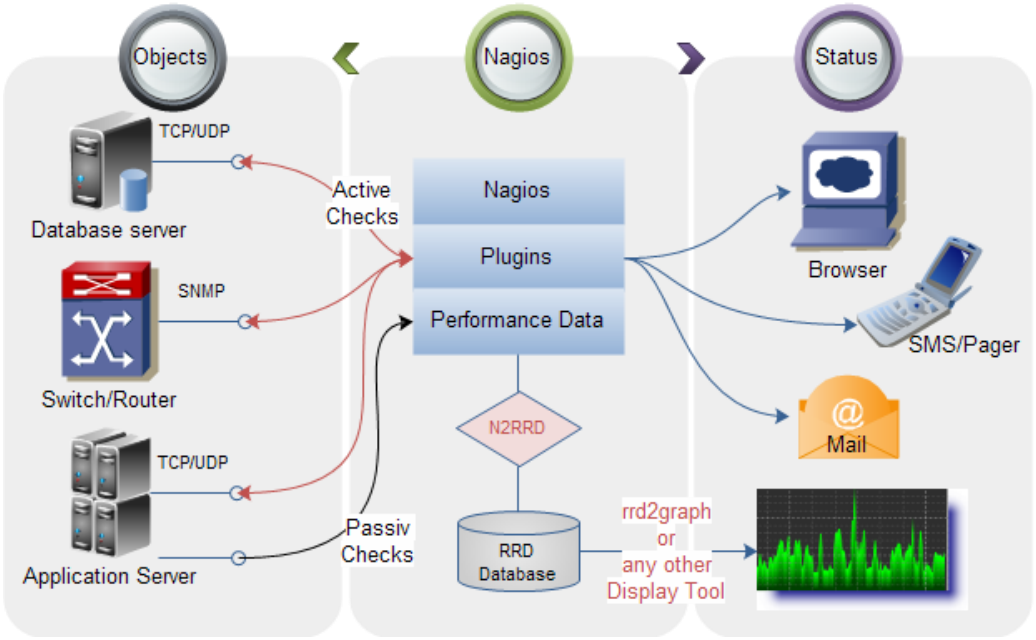
The **Black Box** Monitoring Approach

Rutherford gold foil experiment



- System to be monitored is a ***Black Box***
- Use well-known procedures as probes for the ***Black Box*** and measure the result

The **Black Box** Monitoring Example



- Nagios is a classical example of the ***Black Box*** monitoring
- A framework that:
  - Provides checks for commodity HW and SW
  - Allows to integrate custom checks

Black Box Approach for a DAQ system?

- **D**ata **A**c**Q**uisition is an heterogeneous field
  - Boundaries not well defined
  - An **alchemy** of physics, electronics, networking, computer science, …
  - Hacking and experience
  - …, money and manpower matter as well

- Many DAQ components operate at high rate:
  - Polling for monitoring information is inefficient
- DAQ system has many custom HW and SW components:
  - An opportunity to do monitoring in a better way…

What if the Universe was created by a Computer Scientist?

The White Box Approach

- Objects expose information about their states:
  - E.g. coordinates and velocities of the particles
- The Monitoring system merely takes care of visualizing it in an appropriate way

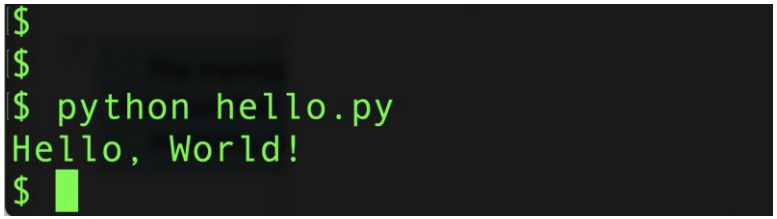
The Simplest White Box Monitoring Example

The monitoring **API** function that is used for incidents reporting

The ***Message*** explains the incident

```
print("Hello, World!")
```

The program exposes its state to external world



The Architecture of a **White Box** Monitoring System

- **API**
  - **The critical component**
  - DAQ system see only this **API**
  - It must be independent of the **Communication** and **Visualisation**
- **Communication** and **Visualisation** are fully flexible:
  - Can be changed multiple times throughout the project's lifetime

Choosing a Good Monitoring API

print("Hello, World")

- The ***print*** function has many flaws:
  - It implements both the **API** and the **Communication** layers:
    - No customization will be possible
  - It adds no meta-information to the message:
    - Severity, timestamp, function name, file and line number, etc…
- Do better solutions exist?

Logging API to the rescue

```
import logging
logging.basicConfig(level=logging.INFO,
 format="%(asctime)s %(levelname)s\
 [%(filename)s:%(lineno)s%(funcName)s()] %(message)s")

logging.info("Hello, World!")
```

Use the standard well-designed API

The output format can be easily customized

```
$
$
$ python hello.py
2022-06-14 14:57:37,493 INFO [hello.py:5<module>()] Hello, World!
$
```

Timestamp

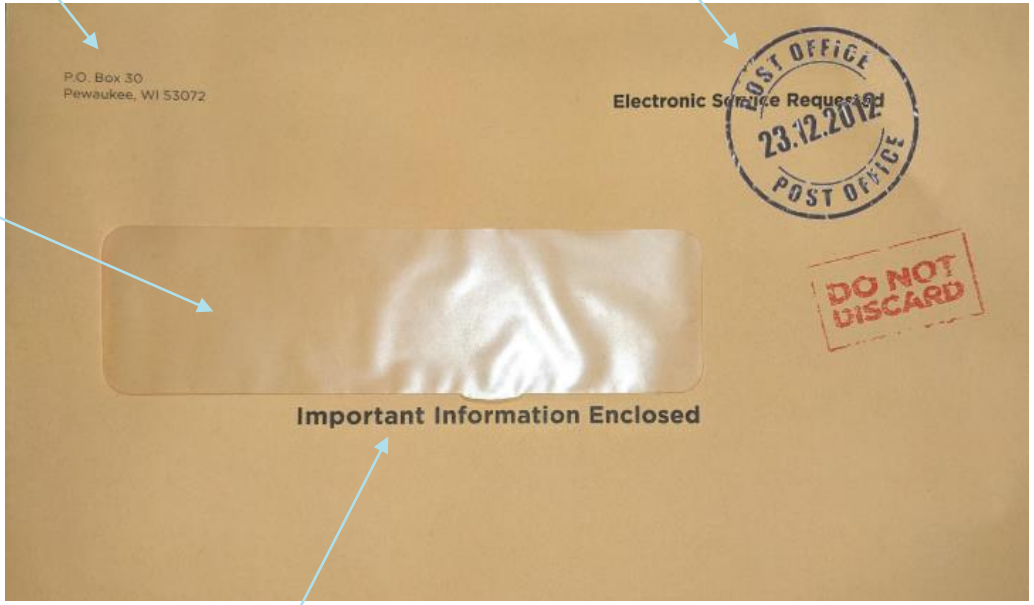
Severity

Origin

Standard properties added automatically

Message Properties

- *Destination Address is not provided by the sender*
- *All messages are public – anyone can read them*



Message Severity

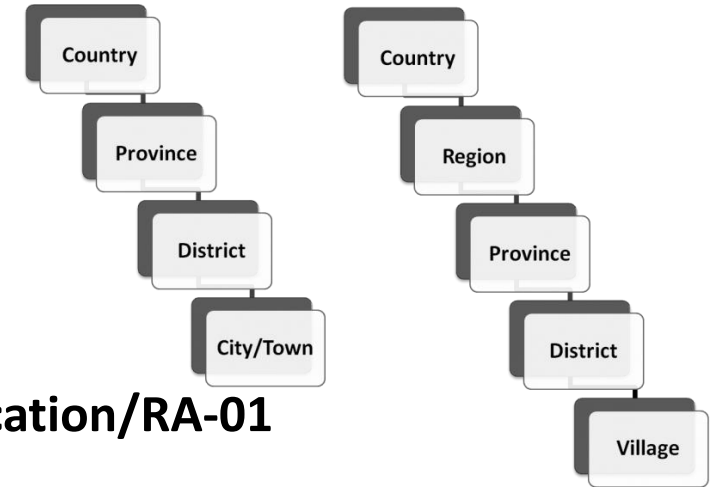
- **CRITICAL**
  - A fatal failure has occurred. This indicates that the component can not do its work any more without external intervention
- **ERROR**
  - A recoverable error has happened
- **WARNING**
  - Nothing is bad so far but the system is close to a certain limit
  - Do not neglect warnings as they tend to become errors
- **INFO**
  - Something that is expected has just happened
- **DEBUG**
  - Detailed information used for testing and debugging
  - Is usually compiled out in production SW for performance

Timestamps

- Used to correlate information from different sources
- Used to correlate information with real life events
- The time stamp guidelines:
  - Use NTP service on all computers
  - Use the best possible precision (nanoseconds) when creating time stamps
  - Use UTC time
  - Conversion to the human readable local time shall be done by the message displaying applications

Message Origin

- Usually is split into two independent properties:
  - Component (process) that produced the message
  - Code (file name + line number) where the message was generated
- Component ID must be unique
- Use a structured string as ID:
  - Make it human-readable
  - Easy grouping and filtering of messages
- Possible examples:
  - **/ATLAS/MuonDetector/ReadoutApplication/RA-01**
  - **ATLAS-MUON-ReadOut-01**



Existing Logging APIs

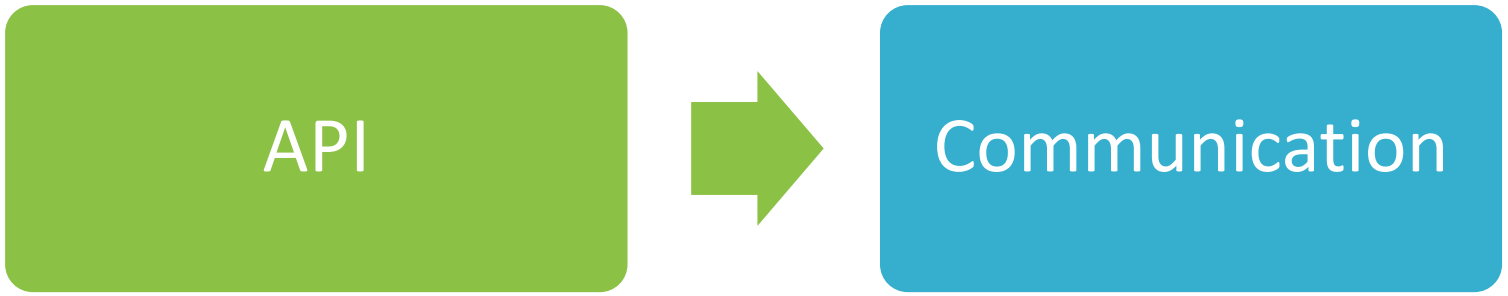
Python

```
import logging


class Logger:
  def critical(msg, *args, **kwargs):
  def debug(msg, *args, **kwargs):
  def error(msg, *args, **kwargs):
  def info(msg, *args, **kwargs):
  def warning(msg, *args, **kwargs):
```

Java

```
import java.util.logging.Logger


class Logger {
  void severe(String msg);
  void fine(String msg);
  void error(String msg);
  void info(String msg);
  void warning(String msg);
}
```

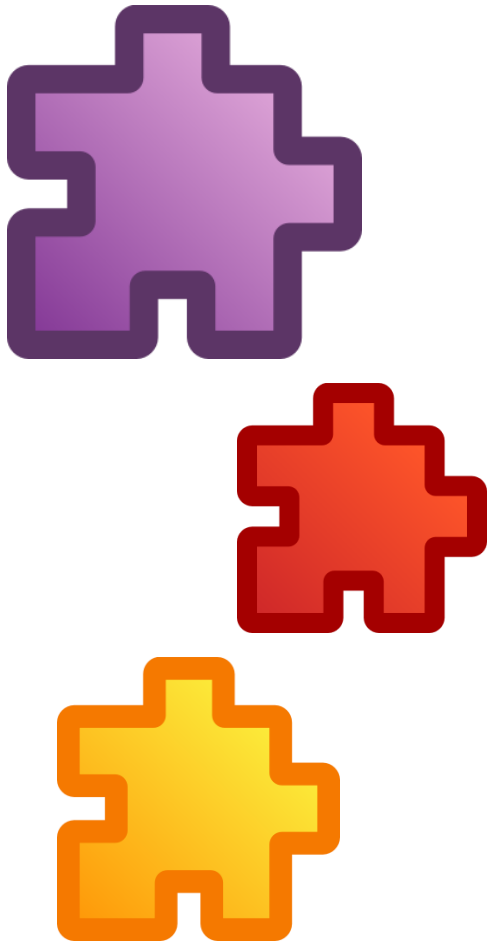
The Main Advantages of a Logging API



- **Logger API**
  - Well-designed and mature
- **Communication:**
  - Different implementations exist on the market
  - They are easily interchangeable
  - Transparent for the applications that are using a particular API

Example: Existing Appenders for Java Logger

- **CassandraAppender** - writes its output to an Apache Cassandra database
- **FileAppender** – writes events to an arbitrary file.
- **FlumeAppender** - Apache Flume is a distributed, reliable and highly available system for efficiently collecting, aggregating, and moving large amounts of log data
- **JDBCAppender** - writes log events to a relational database table using standard JDBC
- **NoSQLAppender** - writes log events to a NoSQL database
- **SMTPAppender -** sends an e-mail when a specific logging event occurs, typically on errors or fatal errors
- **ZeroMQAppender -** uses the JeroMQ library to send log events to one or more ZeroMQ endpoints



What about C++?

- Rare case where using MACRO for the public API is a viable option

```
DAQ_LOG_CRITICAL("File '" << file_name << "' not found")
DAQ_LOG_ERROR(…)
DAQ_LOG_WARNING(…)
DAQ_LOG_INFO(…)
DAQ_LOG_DEBUG(…)
```

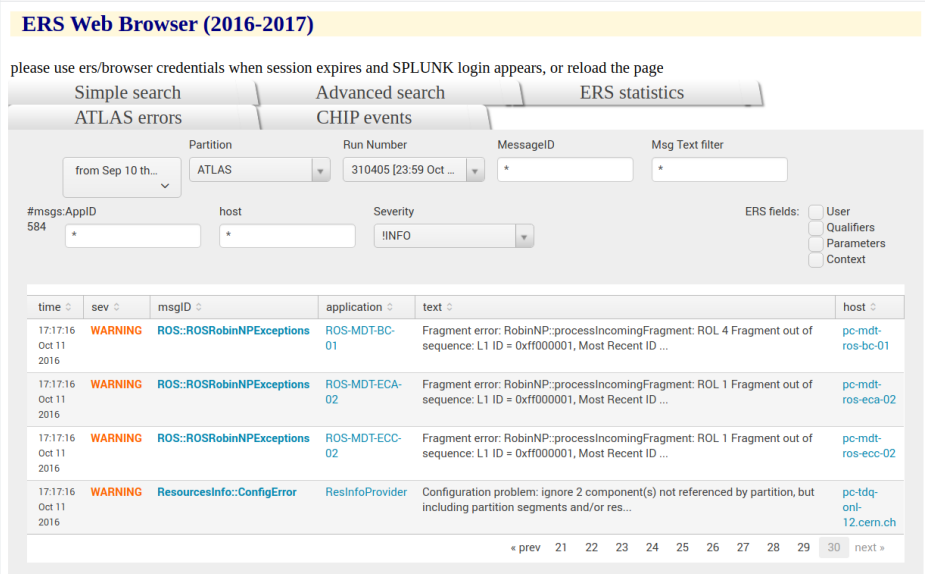
- Initial implementation may be trivial:

```
#define DAQ_LOG_CRITICAL(m) std::cerr << m << std::endl;
```

- A scalable implementation can be provided later:
  - Will not affect users' code

Example: The ATLAS Error Reporting System

C++ MACRO → CORBA[1] → Splunk[2]



[1] Common Object Request Broker Architecture – inter-process communication technology

[2] Splunk – A software platform to stream and collect data

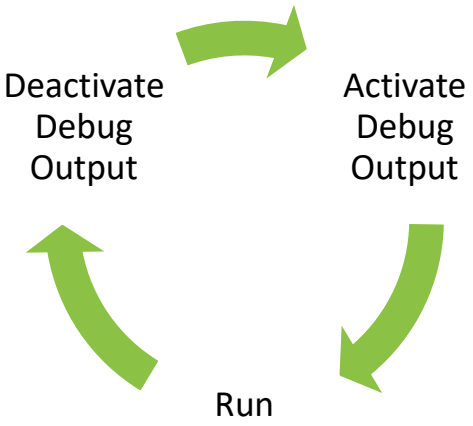
Set Priorities Properly

- Choose (or implement) a Monitoring **API** before starting to implement the DAQ system:
  - The Monitoring must be used by all components of the DAQ system
  - Changing them later will be a pain
- Can take care about **Communication** and **Visualization** implementations later:
  - Using simple output to terminal would be sufficient for the beginning

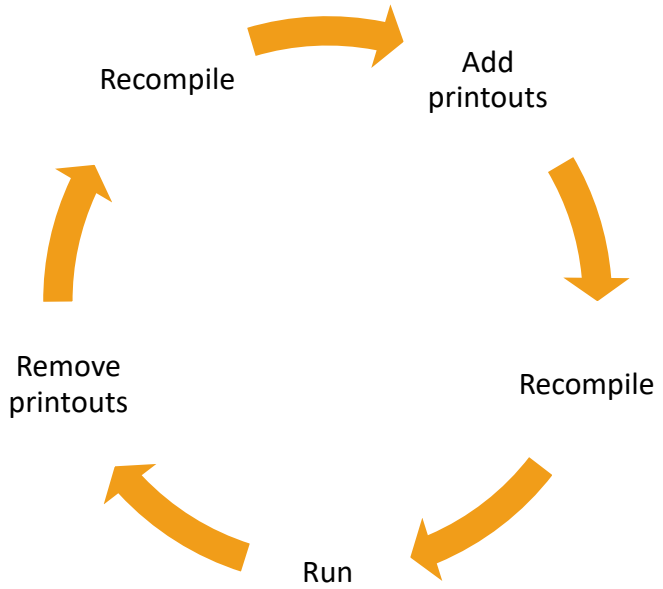
- Advantages:
  - Using the monitoring system will exercise its functionality and performance
  - Learn the best ways of presenting information
  - Speed up the DAQ system development

How Monitoring System can speed up DAQ System Development

Efficient debugging cycle using Monitoring API



"Traditional" debugging cycle



- *Reduces time for debugging*
- *Optimizes the placement of DEBUG output in the code*

Metrics – Another Kind of Monitoring Information

- Values of properties of the software and hardware system components
- Expressed as integer or floating point numbers

Metric Types

Counter

- Monotonically increasing integer number
- Simple to monitor:
  - Last value for the last time period
- Examples:
  - Cumulative totals: number of triggers, number of bytes sent/received, etc.

Gauge

- Arbitrary changing value:
  - Integer or floating point
- Monitoring can be tricky:
  - Last value
  - Mean value
  - Min/Max values
  - Frequency distribution (histogram)
- Examples:
  - Resources usage: CPU, memory, buffe
  - Rates: triggers/s, bytes/s, etc.
  - HW Properties: voltage, current, temperature, etc.

Metrics Monitoring Requirements

- Produced by the DAQ System components

✓Shall be displayed as time series

✓Shall be accessible in real-time

✓Shall be recorded to be checked later

The Architecture of a Metrics Monitoring System

- **API**
  - **The critical component**
  - DAQ system see only this **API**
  - It must be independent of the **Communication** and **Visualisation**
- **Communication** and **Visualisation** are fully flexible:
  - Can be changed multiple times throughout the project's lifetime

A Common API for Metrics?

- There is no commonly accepted API for Metrics:
  - SW tools for metrics collection and analysis use their proprietary APIs
- This may not be a problem for a small short-living project:
  - Directly using a specific SW API is a viable option
  - Be careful to choose a SW with the live-time going beyond your project time-scale
- For example HEP experiments have a life-time of O(10) years:
  - It’s difficult to find a SW system that is likely to survive that long

A Simple Private API for Metrics Monitoring

```
package Atlas.Monitoring;

interface Gauge {
    void setValue(double v);
}

interface Counter {
    void increment();
    void reset();
}

interface Metrics {
    Counter createCounter(String name)
                throw (AlreadyExistsException);
    Gauge createGauge(String name)
                throw (AlreadyExistsException);
}
```

*Makes it independent of the **Communication** implementation*

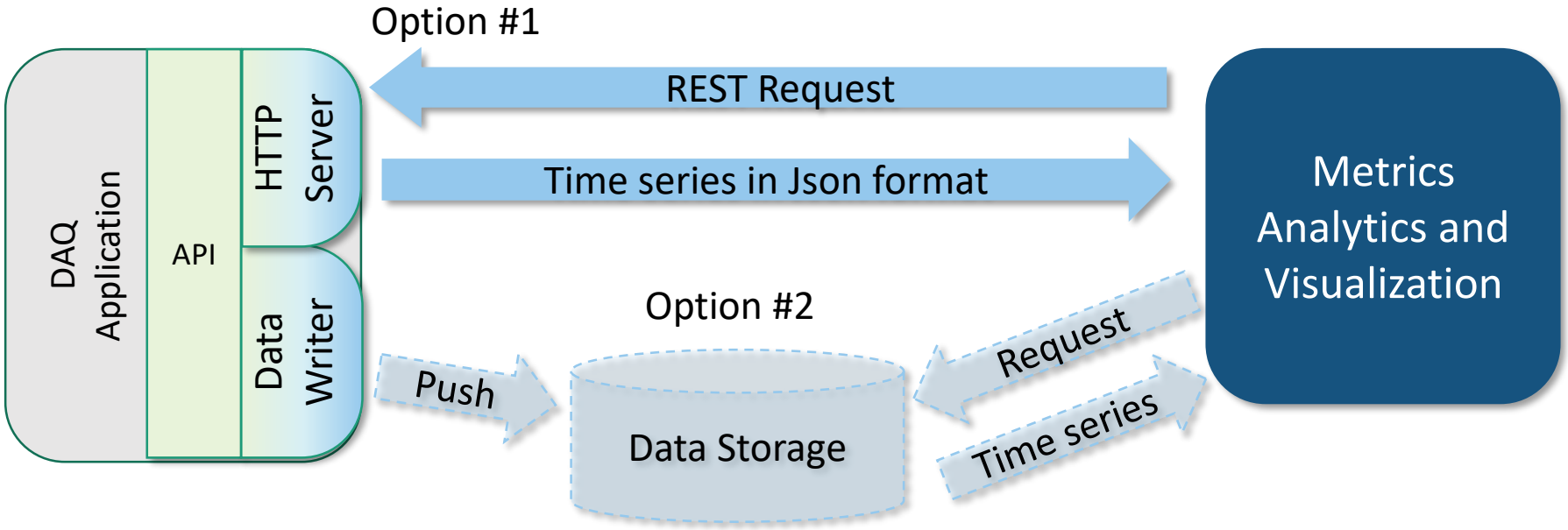
Supports different treatment for Counters and Gauges

Enforces uniqueness of Metrics IDs

Metrics IDs

- All Metrics must have unique IDs
- Uniform naming schema greatly simplifies Metrics handling:
  - Finding required Metrics is straightforward
  - Easy selection and filtering using regular expressions
- A possible approach:
  - Component Name + Metrics Name
- Example:
  - */ATLAS/Dataflow/EventRecoder/**EventsNumber***
  - */ATLAS/Dataflow/EventRecoder/**RecordingRate***

Monitoring System Implementation Options



- The underlying implementation can be updated as the main project evolves:
  - Does not affect the DAQ applications
  - The same Analytics and Visualization tools can still be used

RESTful Protocol

- REST – **Re**presentational **S**tate **T**ransfer
- Client-server HTTP-based stateless communication protocol
- Supported by most of the modern information storage as well as Web-based Visualisation systems:
  - Supports seamless interoperations
- Makes it easy to switch from one Storage or Visualisation platform to another

REST Protocol Example

- Request:

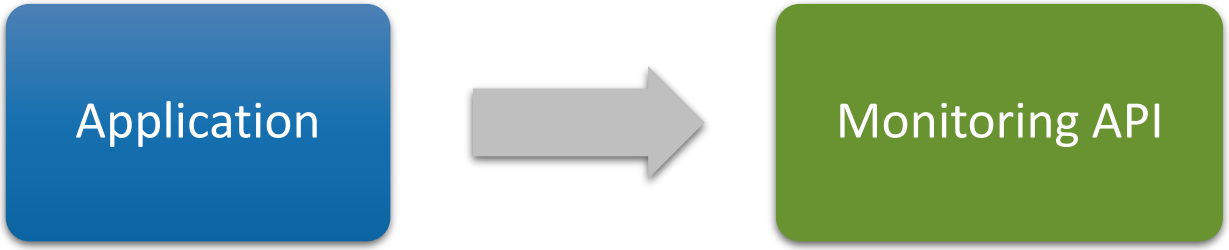
```
https://atlasop.cern.ch/monitoring/
    ? id=ATLAS.Dataflow.RecordedEvents.Rate
    & from=now-30d
    & to=now
```

- Response:

  Json Time Series, e.g.:

```
[
    {t:1579104640,v:12345},
    {t:1579104645,v:12346},
    {t:1579104650,v:12347},
    {t:1579104655,v:12348}
]
```

Metrics Values Update Rate

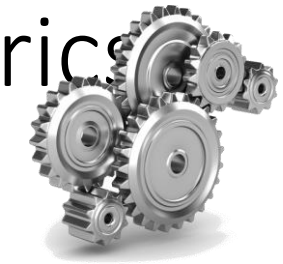


- Metrics update rate is defined by the data handling rate:
  - E.g. rate of triggers for the ATLAS experiment is 100 kHz
- High update rate must be scaled down:
  - Takes too much space in the data storage
    - 100 kHz of event rate => $(8 + 8)*3600*10^5$ = ~6 GB data per hour per single metrics
  - Cannot be directly visualized:
    - 4K displays have 3840 pixels along X axis
    - Can display data for 40ms only

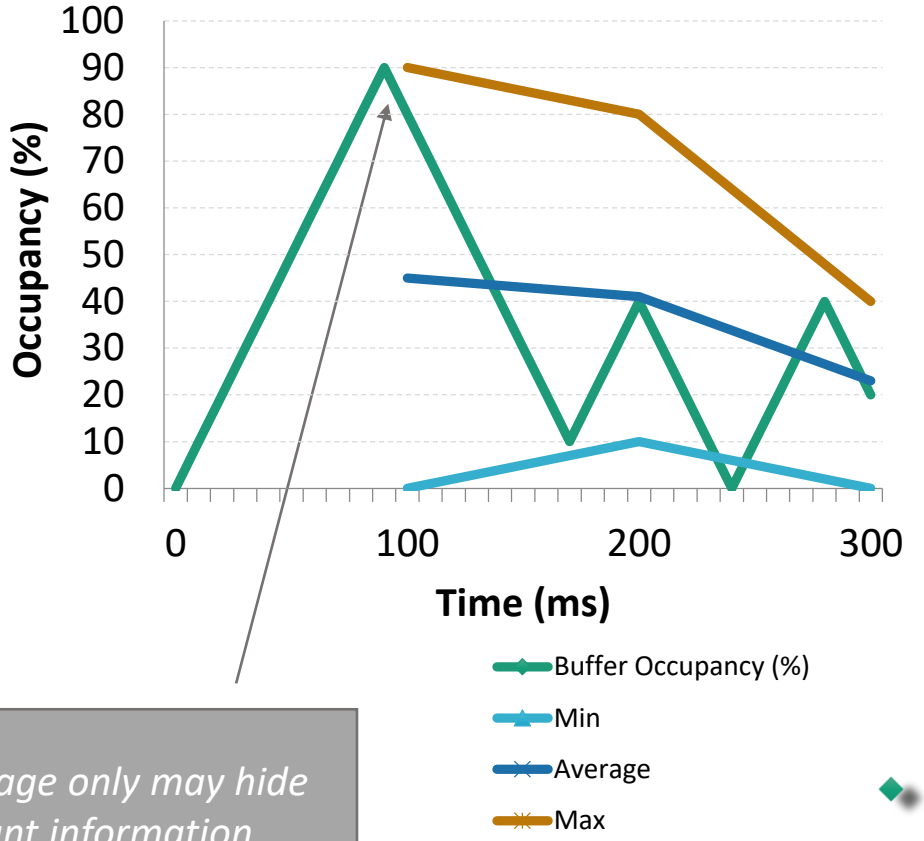
Metrics Rate Down-sampling

- Metrics values can be down-sampled by the API implementation:
  - Reduces recording rate
  - Simplifies storage requirements
- Output update interval can be made configurable:
  - A default value for all metrics
  - Individual values per specific metrics
- Transparent for the **Applications** and **Communication** components

Down-sampling for Different Metrics Types

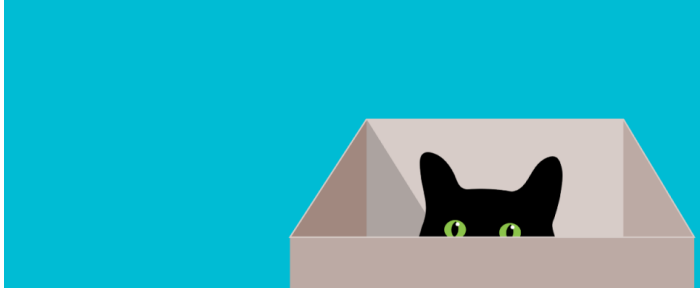


- Counter:
  - Publish the last value for each output update interval
- Gauge:
  - Publish three values for each output update interval:
    - Min, Average, Max



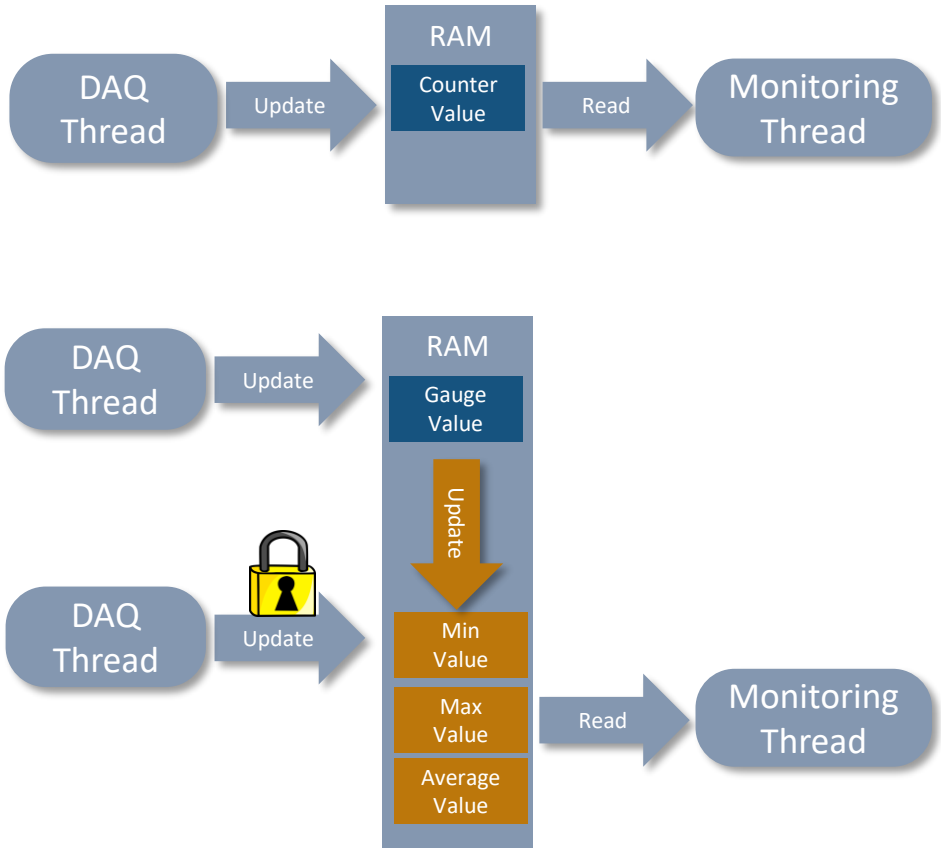
Using Average only may hide important information

Observation Effect



- An observation produces an overhead:
  - It consumes resources (CPU, memory, network bandwidth)
  - It may affect performance of the DAQ applications
- Information must be passed to the **Communication** component **asynchronously**:
  - Monitoring information is updated by the DAQ thread
  - Down-sampling and publishing must be done by another thread
- Thread-safety must be taken into account:
  - But excessive thread-safety measures may hit the DAQ system performance

Thread-safety for Monitoring



- Memory read/write operations are atomic
- For a single DAQ thread:
  - No need for mutexes or even atomics
- Using multiple update threads requires a mutex per Gauge:
  - Monitoring Thread must not keep it locked when passes values to the **Communication** component

Thread-Safety Overhead

- Locking an unlocked mutex takes ~50 CPU cycles => less than 50ns:
  - If the mutex is locked this may lead to arbitrary delay
- Example: monitoring a buffer occupancy:
  - 10 kHz input rate:
    - Mutex locking takes 0.5ms => 0.05% overhead
  - 1 MHz input rate:
    - Mutex locking takes 50ms => 5% overhead

Scaling up the Monitoring System

The HEP Experimental Realm

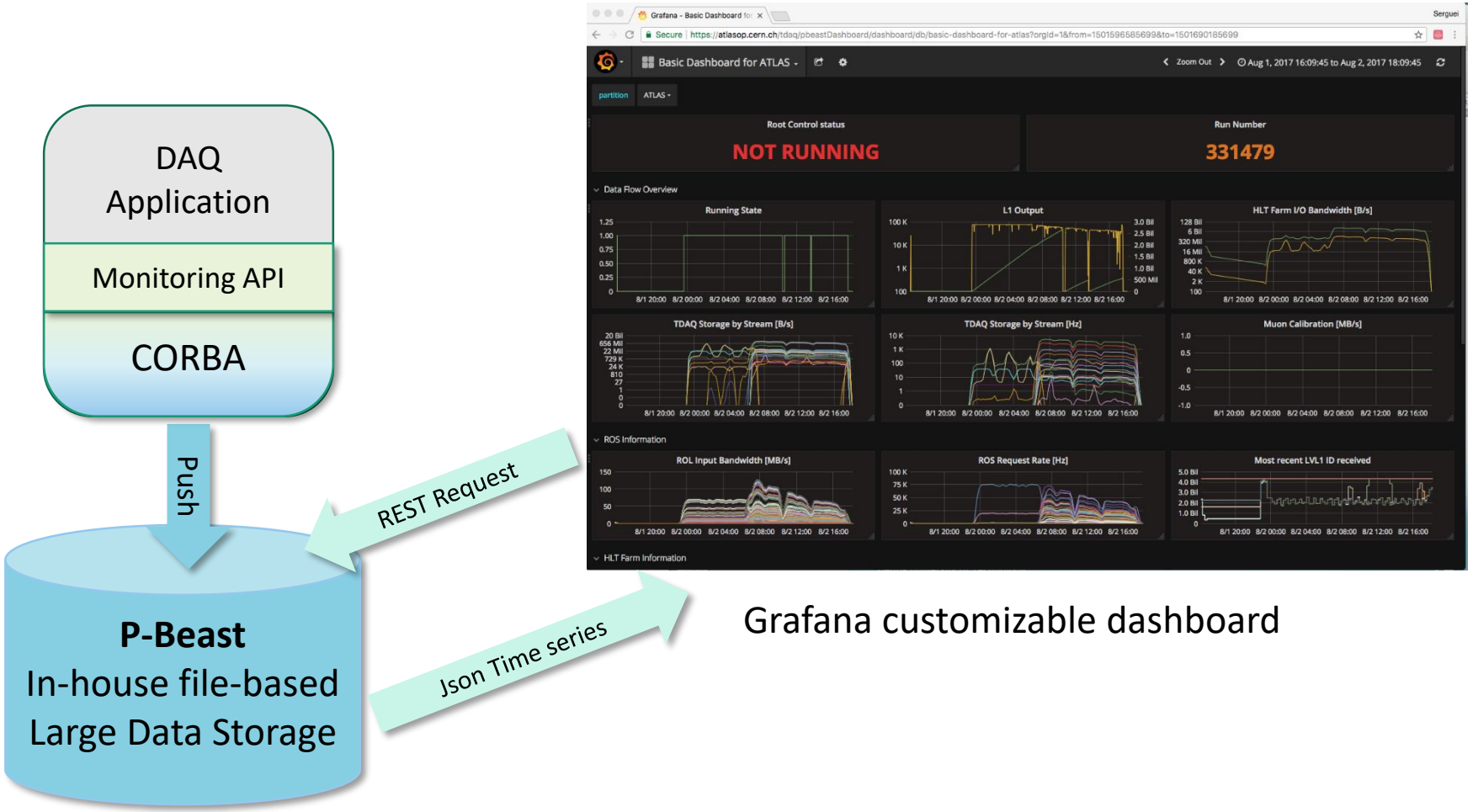
- A DAQ system of a modern HEP experiment consists of:
  - O(1K) computers and network devices
  - O(10K) SW applications
  - O(100K) Metrics

- A single gauge metric for 24h run requires:
  - (8 + 8*3)*360*24=**280**kB of RAM
- 100K Metrics => 28GB per day => 200GB per week => **10TB per year**

Large Storage Implementations

- Traditional relational databases will not work well for large-scale projects
- NoSQL distributed alternatives:
  - **Whisper** – a lightweight, flat-file database format for storing time-series data
  - **InfluxDB** – a time-series database written in Go
  - **Cassandra** – scalable, high availability storage platform
  - **MongoDB** - a general purpose, document-based, distributed database

Example: The ATLAS Web-based Metrics Monitoring

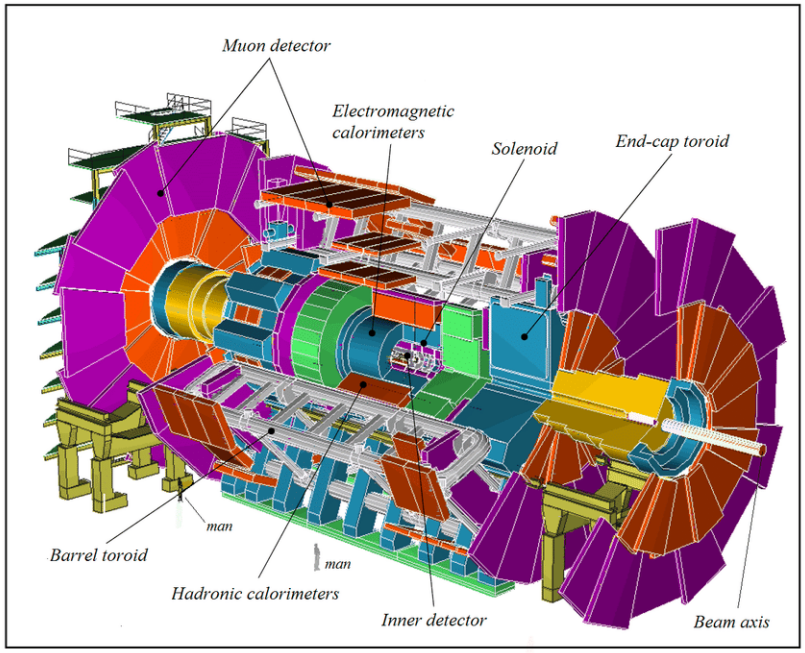


Grafana customizable dashboard

DAQ Specialty:
Data Quality Monitoring

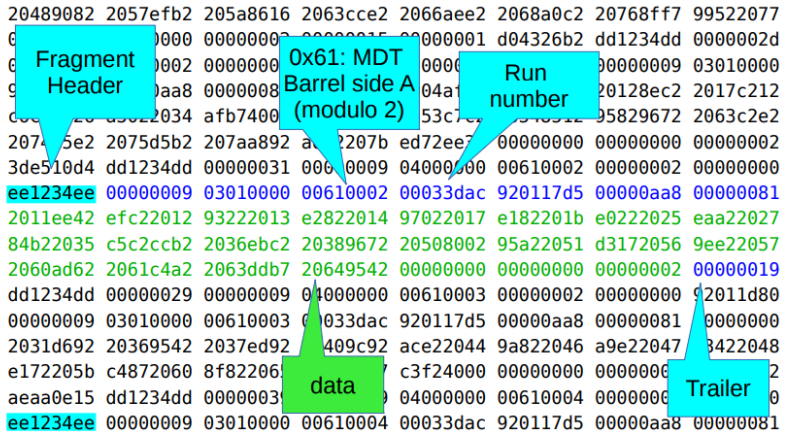
How to Monitor the Detector?

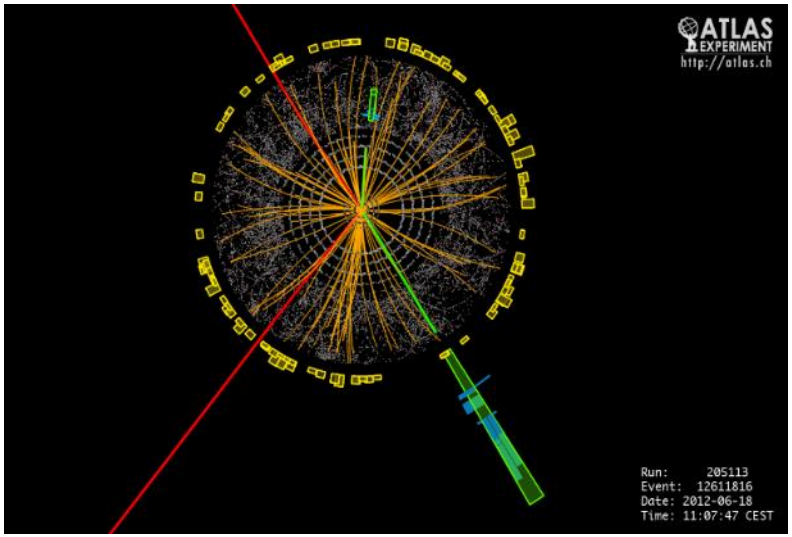
- Detectors of LHC experiments are incredibly complex devices:
  - Up to 10^8 output data channels
  - Mostly custom electronics
  - 40 MHz operational frequency
- Traditional monitoring would yield in O(1) PHz (petahertz) of metrics update rate:
  - These metrics are not even attempted to be produced explicitly
- However DAQ system has a handle on these metrics…



Detector Metrics

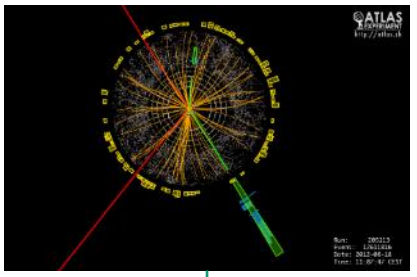
- Each **Physics Event** taken from the detector by the DAQ system contains metrics for a sub-set of detector channels:
  - An expert can spot problems by looking into a graphical event representation
  - This is of course very difficult and unreliable





Automated Data Quality Analysis

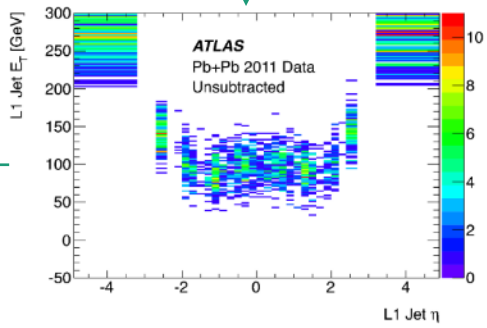
- Dedicated DAQ applications apply standard physics analysis algorithms to a statistical sub-set of the Physics Events:
  - Extract Detector Metrics and build their statistical distributions(histograms)
  - Analyze histograms and produce a new set of Metrics – Data Quality statuses



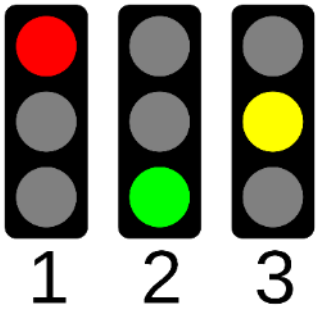
Samples of Physics Events

Physics Event Analysis Algorithms

Statistical Distribution



Statistical Analysis Algorithms



Summary: The Key Points

- ☑ Have your Monitoring System API ready from the beginning of the main project
- ☑ Use standard Monitoring APIs whenever it is possible:
  - e.g. Logging API
- ☑ Think carefully when designing a custom API:
  - It should not depend on a particular technology
- ☑ Monitoring System implementation will evolve in the course of DAQ system development for the mutual benefit
- ☑ Use existing solutions for Communication and Visualization components:
  - In-house development must be well justified