Fermi National
Accelerator Laboratory

Summer internship 2018

Final report

Data Management for High Energy Physics – Monitoring the SAM system

Author:
Francesco Paciolla

Supervisor:
Robert Illingworth

September 26, 2018

Abstract

Current and future Fermilab experiments are using the Sequential Access via Metadata (SAM) system to manage their data. SAM is a highly automated data management system that provides file metadata and location cataloguing, uploading of new files to tape storage, dataset management, file transfers between global processing sites, and processing history tracking.

As for any other computing system, SAM requires constant monitoring to ensure a correct functioning of the system itself and a smarter and more efficient use of the available resources.

In order to lay the groundwork for a new monitoring system, this work shows how to extract monitoring data from the SAM system, how to collect them using open-source software and how to visualize them using different types of charts.

This report also presents a monitoring system prototype that can be used for the early tests.

Introduction

This report presents the work I performed during the summer 2018 at Fermilab where I worked as a Summer Student in the Computing Division section. I worked on a data management system called the SAM system.
The SAM system activity is monitored by an old monitoring system that has some intrinsic limitations. As will be described later, the aim of my project is to lay the groundwork for a new monitoring system that must have four key differences with respect to the old one, as shown in table 1.

Old	New
Queries the SAM database	Queries the SAM station
Limited set of information	Wider set of information
Slowly updating	Fast updating
Stand-alone system	Easy to integrate

Table 1: Differences between the new and the old monitoring systems.

This report shows how I addressed this task and the results that I achieved.

The first chapter describes the SAM system and its core ideas. Particular emphasis is given to the concepts of metadata and dataset to better explain the role of the SAM database.

The second chapter outlines the functioning of the SAM system. Here are shown all the various parts of which the SAM systems consists of and how they interact among each other and with the outside world. The SAM station is also introduced and its role is explained.

The third chapter consists of three main sections. The first explains how I extracted the monitoring data from the SAM station and how I stored them in a log file. The second shows how I collected the data using different open-source software. The data have been collected in a database shared by other monitoring systems. The latter deals with the visualization of the data. Some examples of charts are presented.
Each section contains a subsection that discuss some ideas for future improvements.
Finally, an extra section presents a prototype for the monitoring system that I created. This prototype has a simple graphical user interface; it is completely automatic and therefore can be used for the early testing sessions.

Contents

1 The SAM data management system

1.1 Introduction

The Fermilab Scientific Computing Division has to face multiple challenges while supporting the computing requirements of the many different experiments currently running or planned to start in the next few years at Fermilab. Each experiment produces a huge amount of data that may range from tens of TB per year up to many PB per year spread over a number of files that may be well over the million even for a small experiment. For example, the NOνA experiment produces about 5000 $\div$ 7000 files per day with peaks of 12 000 files [1]. These raw files, as well as the extensive amount of data coming from the simulations, must be processed through several stages in order to reconstruct the physical events.
In this environment, it is not practical to store every single file (raw or offline processed) in a single location and therefore, they are usually spread all over the world and are stored using a combination of different storage systems as [2]:

- Hierarchical storage
  - dCache/Enstore at FermilabdCache/HPSS at BNL
  - CASTOR at CERN
  - GPFS/TSM at INFN
  - etc.
- NFS Filers
  - BlueArc at Fermilab
- Distributed Filesystems
  - LUSTRE
  - CEPH
  - AFS
  - etc.

The location of a file changes over time as it may be moved from tape to a fast cache disk to be processed or from BlueArc onto to tape for long-term storage.
To manage the data on a so complicate and vast system each experiment should provide dedicated experts. This is often not possible given the smaller size of the experiments compared to the large colliding beam experiments that used to run at Fermilab.

After evaluating different approaches, a centralized system was thought to be the easiest way to handle all of the above listed problems at once. Therefore the decision to adopt the SAM system.

1.2 The SAM system

The Sequential Access via Metadata (SAM) system was originally used as a data handling system for the Run II of the Fermilab Tevatron [3]. Afterwards it was adopted by the D0 and CDF experiments. Having acquired a lot of experience from a decade of operation, SAM was a good choice as a unique data management system for the experiments at Fermilab.
Nowadays SAM is in use by the Minos, Minerνa, NOνA, μBoone, DarkSide, and LBNE experiments and is planned to be used at the Muon g-2 experiment [3].

SAM is designed as a highly automated data management system that requires minimal routine intervention. Data staging from tape and from storage element to storage element is completely automatic. This is good for the experiments, which cannot provide dedicated expertise for operating data management systems.
The original version of SAM was designed as a stand-alone system and it required a heavyweight client implementation. This made difficult its integration with other experiments and with the variety of standardized services that have become available as part of the Grid infrastructure. The Fermilab Scientific Computing Division has progressively updated SAM trying to combine modern, standardized, technologies with the past experience of the Run II in order to obtain a lightweight system with an easy to use interface and capable of being used by multiple experiments.

Although the updates and the changes in the SAM architecture, the ideas at its core have been remained the same and are tightly linked to the concept of metadata as will be shown in the next section.

1.3 SAM metadata

One of the main goals of the SAM system is to allow the user to get the files he needs without having to specify, or to know at all, the specific details about its name or position. In order to achieve this, every file stored in SAM has metadata associated with it. The metadata is usually added when the file is first uploaded into the database. As the file is processed or moves from a storage element to storage element the metadata is updated with it.

Metadata is data about the data. It mostly consist of a dictionary of key-value pairs and contains information useful in understanding what is in a file, how the file was generated or how it should be grouped with other files. Metadata may often include:

- physical data such as file size, data and time of creation, format, etc.;
- physics metadata such as run number, detector configuration, simulation parameters, etc.;
- provenance that is the complete history of a file. It stores information about the parent files from which it was derived and the application and version that was used to create it;
- quality check that states whether a data file has passed a quality check.

A list of predefined metadata is available [4].
Some fields are mandatory but the experiments can freely define their own fields, called parameters, depending on their needs. Most experiments have usually one person or few people that define the parameters to be used. The general user needs only to know the complete list of the parameters in use.
As for the fields, the values can be freely defined by the experiments and may be anything, e.g., integer numbers, strings or lists.
Table 2 shows a metadata example from the NOνA experiment [3].

File Name	reco r00013501 s00 t00 numi.root
File Id	4079577
File Format	root
File Size	18931848
Crc	4220712658 (adler 32 crc type)
Content Status	good
Data Tier	reconstructed
Data Stream	0
NOVA.DetectorID	ndos
NOVA.HornConfig	LE010z185
NOVA.HornPolarity	FHC
NOVA.Label	preMeta
NOVA.Special	none
NOVA.SubVersion	1
Online.RunEndTime	1329248111
Online.RunNumber	13501
Online.RunStartTime	1329244506
Online.SubRunEndTime	1329248111
Online.SubRunStartTime	1329244506
Online.Subrun	1329244506
Reconstructed.base release	S12.02.14
Runs	S12.02.14
Parents	ndos r00013501 s00 t00.raw

Table 2: Example of metadata from the NOνA experiment.

SAM stores the metadata in a database that can be queried to obtain the needed files.

1.4 Metadata queries, datasets and snapshots

Users often want to analyze files that share common characteristics, e.g., run number, time range, trigger setup. A group of files that share common characteristic is called a dataset. There can be hundreds, thousands, or hundreds of thousands of files in a single dataset.
A dataset can be created querying the SAM database where all the SAM metadata are stored. There is a specific language to query the catalogue that has been improved thanks to the experience gained from the past implementations. Examples of queries are:

- `run_number 13501 and file_format raw` – this returns all the raw data files from run 13501. Here *run_number* and *file_format* are the fields whereas *13501* and *raw* are their values;

- `run_number 13501 and file_format raw and not isparentof:(application reconstruction and version S12.02.14)` – this returns all the raw files from run 13501 which do not have a derived file reconstructed with the specified version of the software.

Since the database is queried whenever the user requests the dataset, a query may return different datasets when run multiple times. If more files with matching criteria are added into the system since the query was created, they are automatically included in the dataset when the query runs. The resulting dataset may vary depending on many factor and in particular on the query itself. For example, querying the database for all the files created since August 1, 2018, may return a growing dataset. Instead querying the database for files created from August 1, 2018 to October 1, 2018 may return a static dataset. However, if files are backdated later, and fall within that time range, then the dataset will change.
When a static dataset is needed, a direct query of the database is not the recommended solution since it may lead to different sets of files. A snapshot may be used instead.

A SAM snapshot is the actual list of files that satisfy the metadata query at a particular point in time. Being static, the snapshots can be used to compare current analysis with earlier ones. Moreover, the use of snapshots reduces the load on the central database. This is because the snapshot stores all the information about the list of the required files and therefore it does not search the database when it is used.

Having covered the basic ideas behind the SAM system, now it will be shown how these concepts are implemented in order to make the SAM system work.

2 How SAM works

2.1 SAM overview

Figure 1 shows a simplified diagram of the SAM system and its interactions with the outside world.
There are five main parts:

- the user: it may be a physicist that wants to run an analysis on some data collected by an experiment;

- the project: it is created by the user and contains information about the physical analysis to be carried out and about the dataset or snapshot on which the analysis has to be performed;

- SAM: it represent the SAM system as a whole and it is connected to the SAM database, which stores the metadata;

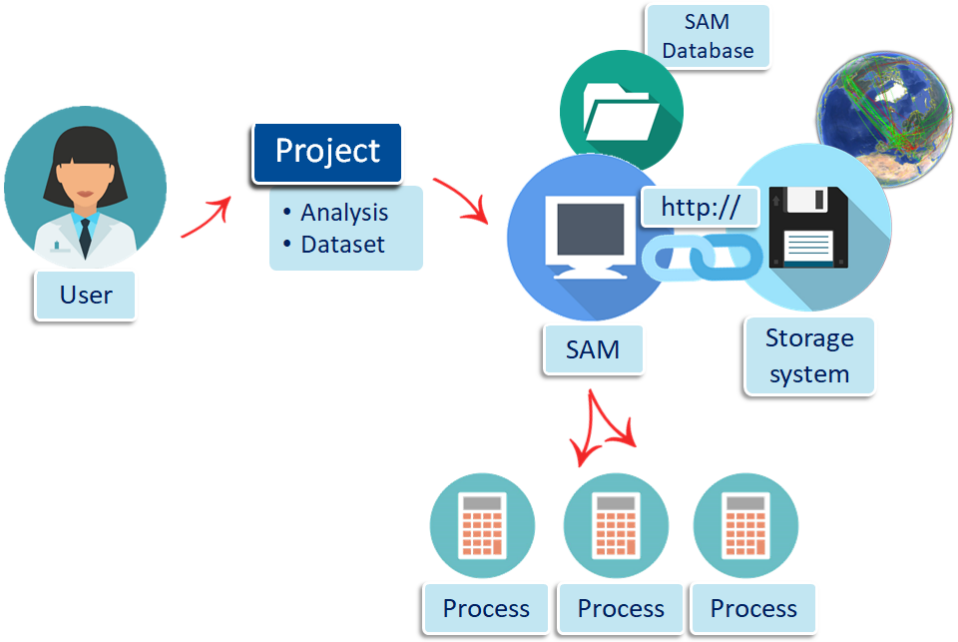

Figure 1: The SAM system and its interactions with the outside world

- the storage system: it is where all the actual files are stored. As already said, they may be stored all over the world. The storage system is connected to SAM through an HTTP protocol;
- the processes: they are the consumers that carry out the analysis on the files.

In the diagram, the SAM system has been oversimplified on purpose. In reality, the SAM system is much more complex than what shown in figure 1. It consists of many subsystems that make it easier to integrate SAM with the outside world and fulfil other different specific tasks. Many of these subsystems have not been used directly and are not relevant to the work done. Therefore, they have been omitted in the diagram to highlight the external connections of SAM with respect to the internal ones.
Among these subsystems, the samweb server and the SAM station need to be mentioned because they were relevant to the creation of the new monitoring system.

The samweb server allows the user to interact with the SAM database. Whenever it is needed to add, delete or modify metadata entries in the database, the user sends an HTTP(s) request to the samweb server. This is generally done via a command line application or a Python or C++ API. In this work, a Python API was used. Through the same interface many more action can be performed, e.g. start or kill a project or process, check the state of a project, recover a project. A complete list of samweb commands can be found here [5].

The SAM station is an application that coordinates all communication between projects, processes and the database, as well as all file delivery activities. Each experiment has his own station on which it runs his projects. Other stations are

used for testing and development. The SAM station can be considered as the core of the SAM system, meaning that it can hold information about projects, processes and files, at the same time. Therefore, this project aims at creating a monitoring system that works querying the SAM station, as it will be explained in the section 2.4.

2.2 Running a project

When a physical analysis has to be carried out, the operation of the SAM system can be schematized as follow:

1. The user creates a project as, e.g. a Python script. In the project, the user must specify the SAM station to use, the dataset and the actual physical analysis. Other optional information may be specified, e.g. the maximum number of processes to start or the actions to perform when errors occur;

2. The script, i.e. the project, is sent to the SAM system through the samweb server via command line and it is interpreted;

3. Information about the dataset are looked up in the SAM database and the actual list of file is created, as well as the snapshot;

4. The list of files is passed to the storage system. The storage system begins preparing the needed files. For example, if a file is on tape, it is moved to the disk;

5. While the storage system gets ready, the chosen SAM station starts the project. Later, the processes are started as well[1];

6. The SAM station asks and receives the URLs that can be used to retrieve the files from the storage system. Each URL is unique and it is directly linked to a specific file;

7. The processes ask for files and the SAM station gives them the URLs one at a time. Once a process receives a URL, it can access the file in the storage system and it can carry out the physical analysis. Individual processes have no direct control over the order the files are delivered;

8. When a process completes the analysis on a file, the SAM station will give it another URL. If there are no more files to be processed, the process is left to die;

9. Once all the processes are dead, the SAM station closes the project.

2.3 SAM advantages

One of the biggest benefits of SAM is that it is not directly coupled with any experiment's framework. Thanks to the samweb server, there is no need to integrate any code into the experiment framework to communicate with SAM. Instead, communication is performed via an HTTP protocol. Therefore, SAM can be modified

[1]The processes are not started by the SAM station but for the sake of simplicity, the actual work chain will be omitted. More information can be found here [2].

and updated or completely replaced with some other data management system without having to touch either the experiment framework or the data handling code.

Requiring no direct coupling, SAM can be also easily coupled with any experiment without having to know its framework. Both the size and the location of the experiments do not matter. Scaling up the SAM system is easy because multiple instances of its component as the SAM station and the samweb server can be created. Moreover, nothing has to be deployed at remote sites since the SAM servers can remain at Fermilab and they can be accessed from everywhere thanks to the HTTP protocol.

The SAM system frees the user from having to know the location of the file almost at no cost; the user is not required to have an intimate knowledge of SAM but he needs only to know how to make certain HTTP requests and deal with responses. Moreover, because SAM works with datasets rather than individual files, the user is not required to manual determine the order with which the files have to be processed. At the same time, SAM can command pre-staging of files from tape, or transfers from storage system to storage system, before the process needs to access the file. This enables more efficient file access than a purely access driven system.

2.4 The need for a (new) monitoring system

As for any other computing system, SAM requires constant monitoring. A good monitoring system allows for a quick solution of the problems that may occur, a correction of possible coding errors and a smarter and more efficient use of the available resources. In fact, many times projects do not proceed as smoothly as presented in section 2.2 for different reasons, e.g. a file transferring may not yet be completed and a process has to hold, a file may be corrupted and could be impossible to process, a project script may contain errors making the analysis impossible to carry out.

To check the SAM operation, a SAM monitoring system is available and it is accessible here [6]. Since it works querying the SAM database, it has some limitations:

- limited amount of available information: since the database stores only the metadata about the files, some information are not passed to the database. For example the file URL cannot be retrieved querying the database;
- slow updates: even though a specific query of the SAM database takes few seconds, a general-purpose monitoring query may take several minutes. The monitoring system updates its data every half an hour;
- stand-alone system: during the last few years, Fermilab has been trying to unify all the monitoring systems created over time for the different experiments. The current SAM monitoring system has not been yet integrated with the others.

To solve these and other smaller issues, it has been decided to create a new monitoring system that queries the SAM station instead of the SAM database.

Extracting the monitoring data from the station has some nice advantages. For example, it allows to collect more information about the system because the SAM station is tightly connected with all the other systems, as it can be seen in figure 1, and holds information about projects, processes and files. Moreover, the station can produce the monitoring data as soon as something of interest happens. There is no need for a database query and consequently there is no need to wait for the query to finish. Therefore, the data can be collected as soon as they are created increasing dramatically the updating speed of the monitoring system.
Creating a new monitoring system comes also with the freedom of choosing where to collect the data making it easier to integrate this monitoring system with all the others.

The goal of the present work is to create such a monitoring system and the next chapter will shows this task has been carried out.

3 Creating a new SAM monitoring system

3.1 Setting the goals

A new monitoring system based on the SAM station instead than the SAM database has multiple advantages as shown in section 2.4. The new system would have extended information on SAM, it would reduce the workload on the SAM database, it would have reduced updating time and it would be easier to integrate it with all the other monitoring systems. These and other advantages come at the cost of having to create a new system from scratches. To get a perfectly functioning system requires time for coding, testing and in particular for making different system to work together smoothly.
Therefore, the main goal of this work is to lay the groundwork for a new monitoring system.

In order to create a working prototype of the monitoring system in the shortest time possible, I decided to focus on three specific tasks:

- extracting metrics from the SAM station;
- collecting the data on a platform shared by other monitoring systems;
- visualizing the data using different types of graphs.

To complete these tasks I borrowed some ideas from the SAM system itself.
As said in section 2.1, the samweb server couples loosely SAM to the other systems in such a way that if SAM needs to be changed, nothing else has to be touched. Following the same idea, I treated each task as a standalone problem. The tasks are connected to each other by the use of dictionaries. The dictionaries, specifically the Python dictionaries, are at the core of the SAM database and I found them greatly useful since I needed to deal with different systems. Their simple structure as a list of `key:value` pairs makes it easy to avoid confusion

when passing information from system to system. This is because every value is always paired up with its brief description, i.e. the key.
Thanks to this approach, I was able to try many different solutions having always a functioning system and clear results. Most importantly, I managed to avoid time losses. This was a key element for the success of the project since many of the choices I made were dictated by the amount of time left.

Before starting tackling the main tasks, I had to complete a training session aimed at learning how to set up a working SAM station and how to interact with it. Therefore, besides studying the functioning of the SAM system already explained in the first two chapter, I had to:

- undergo a Python training. In particular, I focused on classes [7] and dictionaries [8]. The first were fundamental to understand the SAM station code; the latter are at the core of both the SAM database and the new monitoring system;
- set up a virtual machine where to run the SAM station. This was probably the most time consuming task due to compatibility problems between various programs, e.g. Putty, Kerberos and different Operating Systems;
- learn how to use the samweb server interface to create, run and stop a new project;
- learn about the Fermilab monitoring systems: Landscape [9], Kibana [10] and Grafana [11].

3.2 Extracting metrics

The key difference between a monitoring system based on the SAM database and one based on the SAM station is the way the data are collected. The SAM database being a collection of data has to be queried to extract the needed data. Instead, the SAM station is an actual software written on Python and therefore there is no need for a query. Whenever a significant event happens, it is possible to store on a file all the relevant information just by changing the code of the SAM station.
Examples of significant events are:

- the start/end of a project;
- the start/end of a process;
- the opening/closing of a file;
- the update of a project/process/file state;
- the dataset retrieval.

I added a class `WriteToLogFile` to the SAM station code. The class has a `__init__` method that checks for the existence of a log file. If there is none, it creates a new one. Among the other methods, the main one is the `send`. The `send` method requires a dictionary when called. It takes the dictionary, adds a

time information to it and sends (writes) it to the log file. The dictionary is written to the file using the *json* format.
The time information is added by calling the `add_time` method. The time is added as an ISO 8601 formatted UTC time. The time zone information has been explicitly added to avoid confusion.

When a significant event happens, relevant information about the event are stored in a dictionary. The dictionary does not need to have a predefined list of data; only the `event` key that defines the type of event is mandatory. Once the dictionary is created, the `send` method is called to write the dictionary in the log file. An example of a dictionary is shown in table 3.

event	open_file
project_id	74469
project_name	paciolla_ifdh_test_2018091714_30452
station_name	paciolla_python_station
snapshot_id	24288
username	paciolla
process_id	78010
process_state	active
file_name	ifdh_test_file_0
file_id	6387608
file_url	gsiftp://fndca1.fnal.gov:2811/pnfs/fnal.gov /usr/nova/users/bjwhite/ifdh_test_file_0
file_state	delivered
file_size	1457546274
time	2018-09-17T19:33:17.777201+00:00

Table 3: Example of dictionary stored in the log file. In this example, the event that triggered the logging is the opening of a new file.

When a station runs, hundreds of projects may be processing thousands of files at the same time. Between an `open_file` and a `close_file` event there may be hundreds of thousands unrelated entries. Therefore, the dictionaries contain seemingly redundant information to allow for a precise reconstruction of the events.

Sometimes the SAM station discards information not relevant to its functioning. For example, the `station_name` value is only used by the station when it first chooses where to run a project. The value is discarded as soon as the project starts although for monitoring reasons it needs to be stored until the project stops. To avoid losses of vital monitoring data, I changed some of the classes' definition, as well as their respective `__init__` modules, in the SAM station code.

3.2.1 Future improvements

In order to make it easier to improve the code later, I found convenient to create the `WriteToLogFile` class. Classes are easy to modify by simply adding new methods.

For example, a new method may be created to set a limit on the log file size. With the current implementation, every new event is added to the same log file. With time, its size may increase so much to make it difficult to handle. A limit could be set on the byte size or on the maximum number of lines to write. Once the limit is reached, a new log file can be created.
A different approach may instead not require at all the creation of a log file. The processes of opening, writing and closing a file are usually slow. Therefore, a method may send the monitoring data directly to the collection system, skipping the log file creation.
These solutions could result useful when scaling up the system. However, they were not adopted yet because the system was tested with few simple projects that did not require storing more than a MB of data per project. Moreover reading a log file was easier than querying a monitoring system database when checking for coding errors.

3.3 Collecting the data

Fermilab, during the last few years, has been trying to collect all the data that come from its many monitoring systems in a unique place. The purpose of this program is to provide a comprehensive framework to allow the various teams and experiments to monitor services and jobs running on the Fermilab servers. The database that is collecting all the monitoring data is Elasticsearch.
Elasticsearch [12] is a distributed real-time document store where every field is indexed and searchable. It is possible of scaling it to hundreds of servers and petabytes of structured and unstructured data. It provides a distributed search engine with real-time analytics. Elasticsearch comes with a collection of other open-source products designed to help the user managing the data.

The metrics extracted from the SAM station ore stored in a log file. In order to be properly analysed, they have to be uploaded to the Elasticsearch database. Since Elasticsearch does not come with an integrated capability of reading log files, another solution had to be found. Therefore, I adopted a "chain of software" capable of reading the log file and sending the information to Elasticsearch.
The software used are:

- Filebeat [13]: it is a lightweight shipper for forwarding and centralizing log data. It reads and forwards log lines and, if interrupted, remembers the location of where it left off when everything is back online. Filebeat forwards the data to Kafka;

- Apache Kafka [14]: it is an open-source stream-processing software platform written in Scala and Java. It aims to provide a unified, high-throughput, low-latency platform for handling real-time data feeds. Additionally, Kafka can aggregate and process data and connects to external systems as Elasticsearch;

- Kibana [10]: it is an open-source analytics and visualization platform designed to work with Elasticsearch. It aims to makes it easier to understand large volumes of data. It can search, view, and interact with data stored in

Elasticsearch indices. It can perform data analysis and visualize the data in a variety of charts, tables, and maps.

Figure 2 sketches how the software are connected. Filebeat reads the entries in the log file and sends them to Kafka. Kafka aggregates the data and forward them to Elasticsearch for storing. Once the data reach Elasticsearch, Kibana can visualize them.

Figure 2: The "chain of software" needed to bring the monitoring data to Elasticsearch from the log file.

This second task of collecting the data stands a bit apart from the other two because it did not require extensive coding to be completed. It mostly consisted in forcing different software to work smoothly with each other.

3.3.1 Future improvements

An easy improvement may consist in removing Kafka from the "chain" and making Filebeat forward the data directly to Elasticsearch. Other solutions may require avoiding the creation of the log file. The data may be sent directly to Elasticsearch and, consequently, there would not be any need for Filebeat or Kafka.
The way the system is built allows for these and other changes. Filebeat, Kafka, Elasticsearch and Kibana were used just because they were already in use at Fermilab. In theory, any of the software used can be substituted or removed at any time. As long as the dictionaries do not change passing from software to software, it is still possible to visualize the data with the visualization system that will be discussed in the next section.

3.4 Visualizing the data

As said in section 3.3, Kibana allows visualizing the data stored in the Elasticsearch database. Figure 3 shows an example of a graph obtained with Kibana.

Even though Kibana has many built-in functions designed to make it easy to query the Elasticsearch database and to aggregate the results, it has two big downsides. Firstly, Kibana does not provide anything to process raw data, apart from some simple functions as counting or taking the average. This problem could be solved if the data were pre-processed before being uploaded. This solution is impractical for many reasons. It would require a real-time reconstruction of thousands of events, many of which incomplete, whose time duration may be as long as a

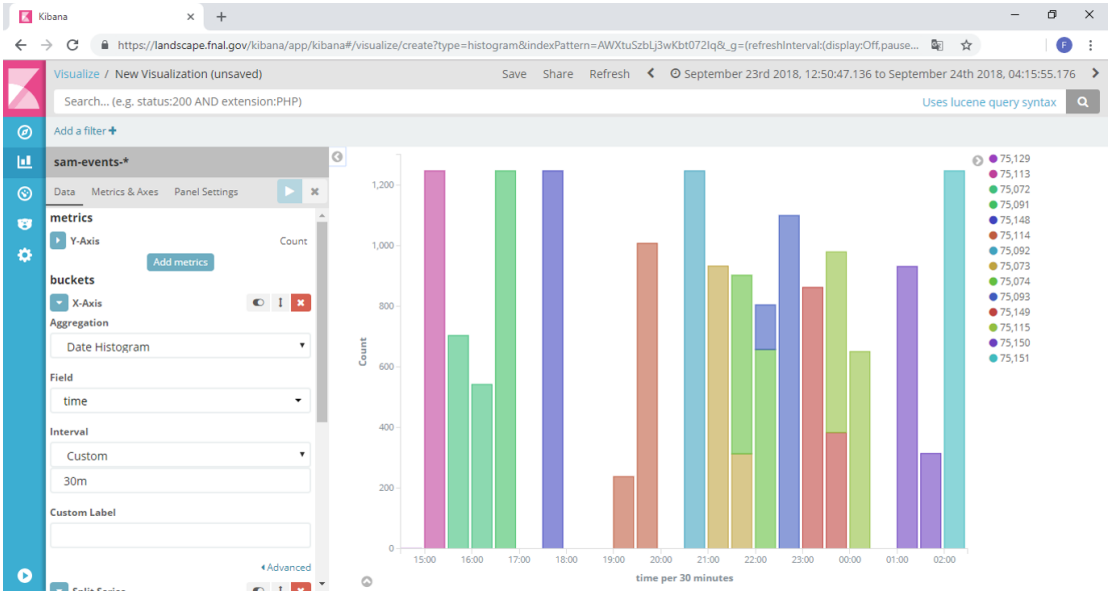

Figure 3: Example of chart drawn by Kibana. It shows the number of events per project occurred in the time interval chosen.

week. Even if this was possible, changes in Kibana may make the old data unusable. Moreover, another step would have to be added at the beginning of to the "software chain" depicted in figure 2. A second downside of Kibana is the limited amount of available types of charts. For example, a Gantt chart like the one in figure 4 is, right now, impossible to draw using Kibana. A possible workaround would require the use of a third-party software resulting in an even longer "software chain".

Different software have been considered as possible alternatives to Kibana. In the end, I decided to use Python and its `matplotlib` library [15]. This solution has the advantages of require no pre-processing of the data and no third-party application. A Python code is also relatively easy to integrate in other systems and the matplotlib library allows drawing virtually any kind of graph. Finally, coding with Python was time-wise cheaper than having to learn how to use any other software.

The code consists of two classes and four main functions. The two classes are `Query` and `DataManagement`. They retrieve and process the data in Elasticsearch. The four functions plot four different types of graphs.

The `Query` class queries the Elasticsearch database through its `elastic_query` method. The method search for all the events happened in a defined time interval. The end of this time interval is defined by the first five arguments (year, month, day, hour, minute) required when calling the `elastic_query` method. Other two optional arguments are available. The first defines the duration of the time interval. The second one is a Boolean value: if true, the end of the time interval is set to be the running time of the method.
The `elastic_query` method returns a list of dictionaries called `log_file`. The `log_file` content does not differ in any way from what it would be obtained reading line by line the log file created by the SAM station. Therefore, just by reading

the log file, the data can still be visualized even when Elasticsearch cannot be reached.

The `DataManagement` class processes the `log_file` to reconstruct the events. Its `extract_info_project` method takes in the `log_file` list and returns a nested dictionary called `info_project`. The `info_project` resembles a forest in its structure. Each tree in the forest represent a project. Attached to a project, there are the processes with their files in the same way as brunches and leaves are attached to the tree trunk.

The `info_project` stores all the information needed to plot the graphs. Therefore, it is a required argument for each of the four functions. Each function has two parts. The first picks out the information needed to plot the graph from the `info_project` dictionary and stores them in some lists since lists are easier to digest for the matplotlib functions. Moreover, this allows adding any kind of information to the `info_project` without having to change the plot functions; if an information is not relevant to the graph, it is simply ignored. The second part contains the actual code to plot the graph.

The next paragraphs will show the features of the four graphs obtained. The data shown in the graphs have been obtained by simulation running a test project with a dataset consisting of 301 files. When a file was correctly processed, it was marked as *consumed*. Randomly, some of the files were marked as *skipped* to simulate a processing error.

3.4.1 Overview

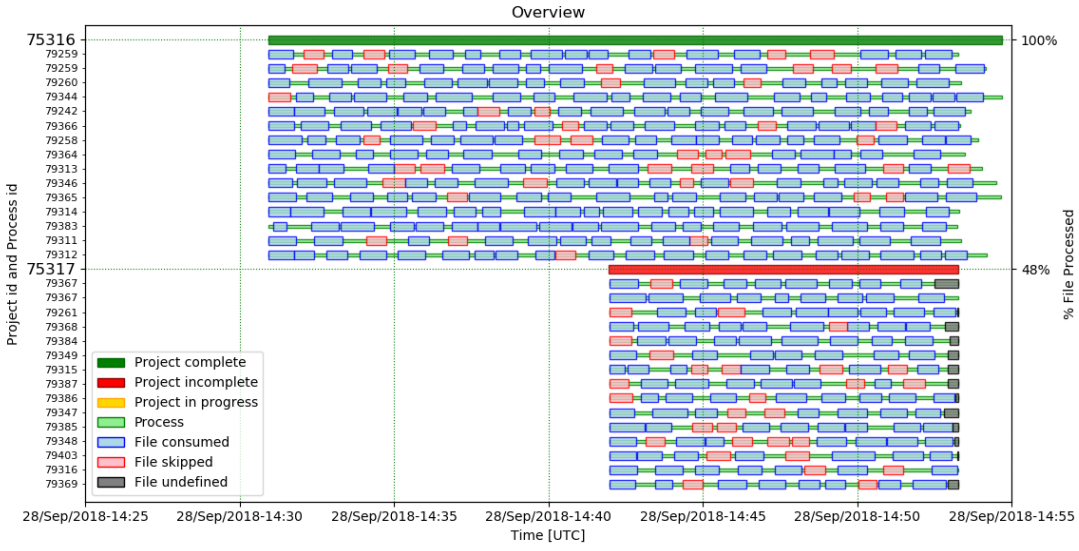

Figure 4: A Gantt chart that gives an overview of the projects running on the SAM station.

The *Overview* is a Gantt chart that shows what the SAM station was doing during the specified time interval. The example in figure 4 shows that, during the thirty

minutes interval considered, two projects were running. The projects are represented by the long wide bars. A project, the green one, was completed successfully; the other, the red one, was interrupted midway. The processes are represented by the narrow light green bars placed below the project that started them. The projects started fifteen processes each. The files are shown as small blocks placed above the process that processed them. Not all the files were correctly processed. Some, the red ones, contained errors and were skipped after some time. Others were left incomplete when the project was stopped and therefore are grey coloured. When a file is consumed or skipped, it may take some time to download the next file from the storage system. That is why sometimes there is some space between two consecutive files.

Thanks to the colour coding and to the use of the Gantt chart that emphasize the father-sons relation existing between a project and its processes or between a processes and its files, it easy to spot different problems. For example, a project that skips all the files may contain an error in its script, whereas a project that takes too much time to process the files may indicate a shortage of resources.
Failed projects and processes can be identified by their ID number shown on the left y-axis of the chart. The right y-axis shows the percentage of files processed by each project. It can be used to estimate the time needed to complete a project.

3.4.2 File Processing Time

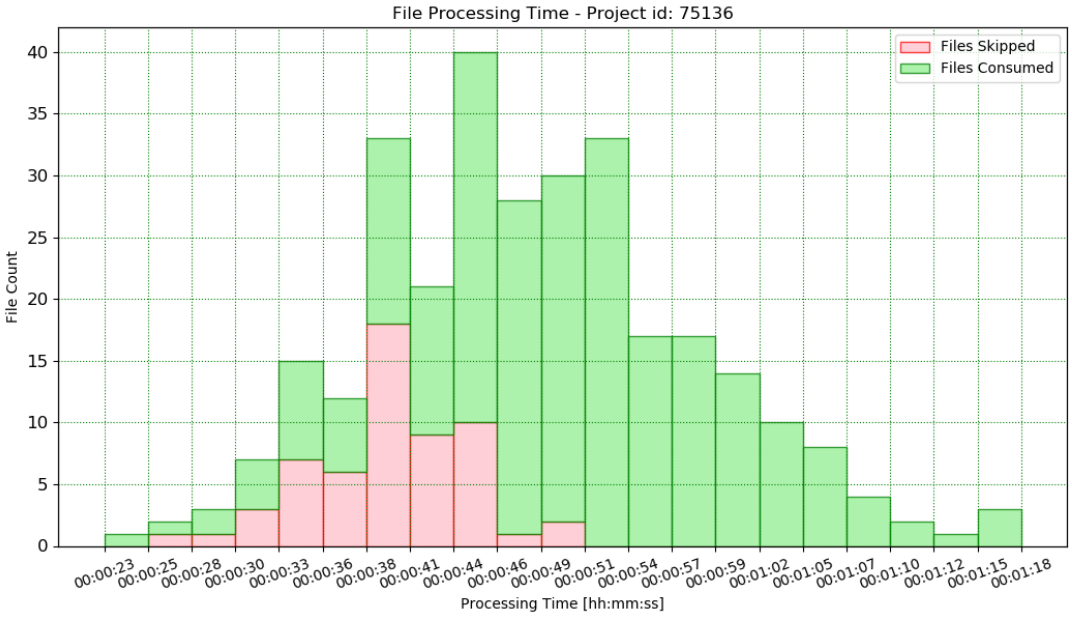

Figure 5: A stacked histogram type of chart that shows the amount of time spent processing the files.

For each running project, a *File Processing Time* chart is available. It is a stacked histogram that shows the time needed to process the files. Files that are currently being processed are not included in the chart. The example in figure 5 shows that the average processing time for consumed files is about fifty seconds. Instead, it takes about forty seconds to skip a file.

The difference between the two time averages provides information on the issue that caused the files to be skipped. In the example, it may reflect an error found towards the end of the files. Instead, an average time of few milliseconds for the skipped files may indicate that the files could not be opened at all.
This chart does not take in account file size.

3.4.3 File Processing Speed

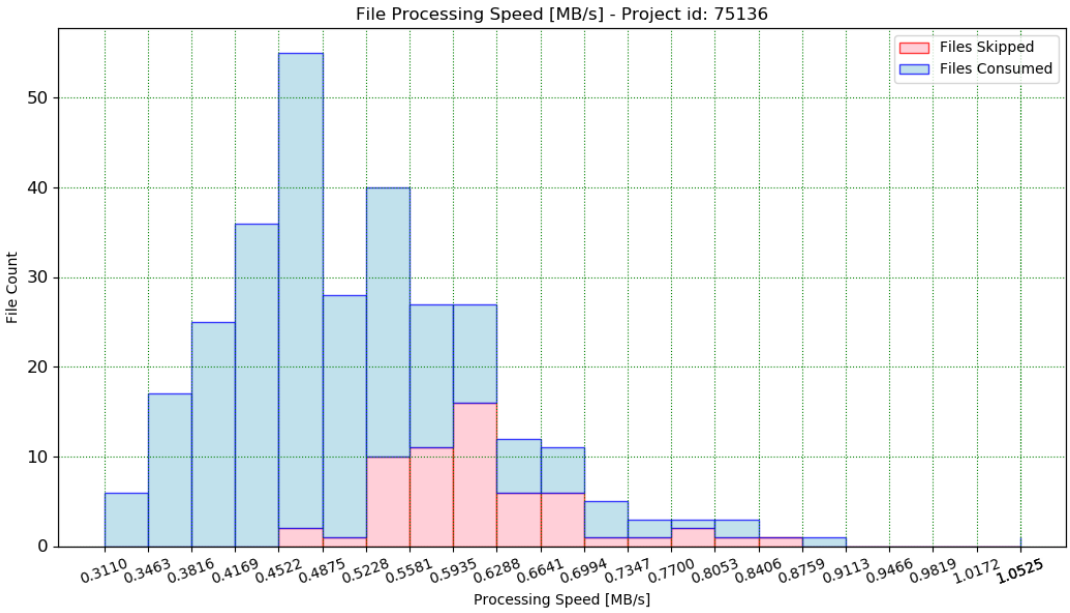

Figure 6: A stacked histogram type of chart that shows the processing speed of the files.

The *File Processing Speed* chart shown in figure 6 is quite similar to the *File Processing Time* one with the only difference being that it takes into account the size of the processed files.
Both the *File Processing Time* and the *File Processing Speed* graphs can be particularly useful for testing the storage system since files hosted in different places have in general different processing speed.

3.4.4 Files Opened, Closed and Skipped

This chart is a cumulative histogram that shows how many files have been consumed or skipped with respect to the number of opened files. A red dotted line represents the total number of file to be processed. The *consumed* and *skipped* histogram are stacked and overlaid on the *opened* histogram so that they touch the red line when the entire project is completed. In the example shown in figure 7, about 190 files have been opened but only about 140 files have been correctly processed. About 30 files were skipped whereas other 20 were still being processed when the chart was made.

In the example, the slope of the "skipped" line is constant and may indicate a problem related to the files. Instead, the presence of jumps may reveal an issue

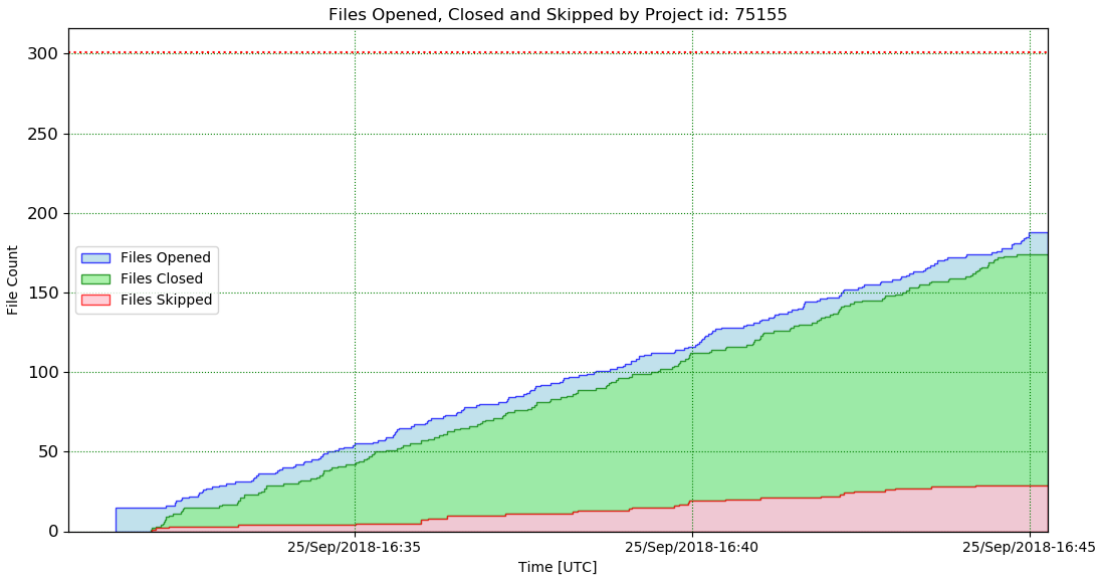

Figure 7: A cumulative histogram type of chart that shows how many files have been *opened*, *consumed* and *skipped* over time.

with the system itself, e.g. the machine that was processing the files has suddenly gone offline.

3.4.5 Future improvements

The four charts presented are just few of the many that can help identify the different issues that prevent a correct functioning of the SAM system. A functioning monitoring system would probably require many more. The same reasoning is also true for the query performed on the Elasticsearch database. Even tough all the charts shown above have been creating using the same query, using different and more specific queries may result in a better visualization of the data and in an overall increase of the speed of the monitoring system.

Foreseeing the need for new types of charts and queries, classes and functions have been used in order to make the process of adding new features easier. For example, more charts can be added by simply defining a new function with the `project_info` dictionary as an argument. In the same way, specific queries can be added to the one in use just by adding other methods to the `Query` class.
Apart from adding more charts and queries, the way the code is written makes also easier to add new features to the charts themselves. For example, in the figures 4, 5, 6, 7 the only two available end states for a file are *consumed* and *skipped*. If a new end state becomes available in the SAM station code, an alert message informs the user of the change. Moreover, to show the new end state in the charts, the user needs only to define a name and a colour.

In order to choose the type of features, charts and queries to add, a first testing session is needed. However, testing the system using the charts alone would be cumbersome. Switching from chart to chart requires changing slightly the code

and selecting a particular project is even more difficult. Therefore, I embedded the charts in a software with a graphical user interface that will be presented in the next and last section.

3.5 A prototype for the monitoring system

To check whether my work could be used for a real monitoring system, I embedded the four charts in a program with a graphical user interface. The graphical user interface makes it easier to switch between the different charts and projects and consequently makes the testing process easier.
An important feature of this monitoring system prototype is the automatic update of the data shown. The program queries the Elasticsearch database and redraws the charts every thirty seconds. This has to be compared with the thirty minutes needed to update the old monitoring system.

Figure 8: The GUI interface for the monitoring system prototype.

The graphical user interface has been realized combining the `matplotlib` [15] and the `tkinter` [16] Python libraries and it is shown in picture 8. It consists of a single window divided in three parts: a menu bar, a canvas and a project sidebar.

The menu bar, shown in detail in picture 9, has four drop-down menus:

- File: it allows to save the current shown graph and to close the program;
- Data Time Frame: it allows to change the duration of the time interval;
- Chart Type: it allows to choose the chart to draw;
- Update: it performs a new query of the Elasticsearch database and updates the chart.

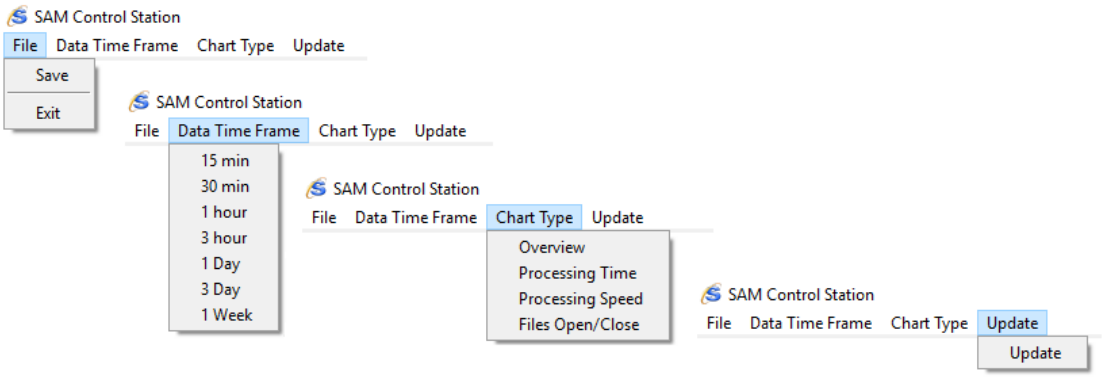

Figure 9: Menu bar detail of the GUI.

The canvas shows the selected chart. It contains the standard `matplotlib` sidebar that allows the user to interact with the chart performing simple actions like zooming and panning. A save button is also available.
The project sidebar contains a clickable button for each project that was running during the chosen time interval. The buttons allow changing the project shown by the chart.

4 Conclusions

The three tasks of extracting, collecting and visualizing the monitoring data have been completed and therefore the backbone of a new monitoring system has been successfully created.
The monitoring data are extracted from the SAM station and saved on a log file. Using different open-source software the monitoring data are then collected in the Elasticsearch database. Specifically, Filebeat reads the entries in the log files and sends them to Kafka. Kafka aggregates and forwards them to Elasticsearch. Querying the Elasticsearch database, the data can be finally visualized using a Python script. Four different graphs have been obtained so far.
Furthermore, using the `matplotlib` and `tkinter` Python libraries, a prototype for the new monitoring system has also been created. The prototype satisfies all the four requirements listed in table 1 in the introduction section:

- it works querying the SAM station;
- it has a wider set of available information regarding the SAM system;
- it stores its data on Elasticsearch as other monitoring systems already do;
- it shows live data since it updates every thirty seconds whereas it takes thirty minutes to update the old monitoring system.

References

[1] A. Aurisano *et al.*, "Data handling with SAM and art at the NOvA experiment", J. Phys. Conf. Ser. **664** (2015) no.4, 042001.

[2] User Guide for SAM,
https://cdcvs.fnal.gov/redmine/projects/sam/wiki/User_Guide_for_SAM

[3] R. A. Illingworth, "A Data Handling System for Modern and Future Fermilab Experiments", J. Phys. Conf. Ser. **513** (2014) 032045.

[4] Metadata format,
https://cdcvs.fnal.gov/redmine/projects/sam-web/wiki/Metadata_format

[5] Samweb Client Command Reference,
https://cdcvs.fnal.gov/redmine/projects/sam-main/wiki/Sam_web_client_Command_Reference

[6] SAM Station Monitoring,
http://samweb.fnal.gov:8480/station_monitor

[7] Python 3.6 Classes,
https://docs.python.org/3.6/tutorial/classes.html

[8] Python 3.6 Dictionary,
https://docs.python.org/3.6/tutorial/datastructures.html#dictionaries

[9] Landscape,
https://landscape.fnal.gov/d/000000097/landscape?refresh=10s&orgId=1

[10] Kibana,
https://www.elastic.co/products/kibana

[11] Grafana,
https://grafana.com/

[12] Elasticsearch,
https://www.elastic.co/products/elasticsearch

[13] Filebeat,
https://www.elastic.co/products/beats/filebeat

[14] Apache Kafka,
https://kafka.apache.org/

[15] matplotlib,
https://matplotlib.org/

[16] tkinter,
https://docs.python.org/3.6/library/tkinter.html

https://docs.python.org/3.6/library/tkinter.html