Carolina Lindqvist

An alarm aggregation framework for critical IT service monitoring at CERN

Aalto University School of Science

Thesis submitted for examination for the degree of Master of Science in Technology.

Esbo 10.6.2015

Thesis supervisors:

Assoc. Prof. Keijo Heljanko
Assoc. Prof. Magnus Lie Hetland

Thesis advisor:

M.Sc. Alvaro Gonzalez Alvarez

Aalto University
School of Science

AALTO UNIVERSITY
SCHOOL OF SCIENCE

ABSTRACT OF THE
MASTER'S THESIS

Author: Carolina Lindqvist

Title: An alarm aggregation framework for critical IT service monitoring at CERN

Date: 10.6.2015 Language: English Number of pages: 8+71

Aalto University School of Science

Professorship: Security and Mobile Computing

Supervisors: Assoc. Prof. Keijo Heljanko, Assoc. Prof. Magnus Lie Hetland

Advisor: M.Sc. Alvaro Gonzalez Alvarez

The emergence of cloud computing has transformed the architecture of Information Technology (IT) services during the last decade. In consequence, monitoring solutions are actively adapted and developed to support the management of the virtualized, scalable, and dynamic environments hosting services now. This thesis work was done as part of an internship at CERN, the European Organization for Nuclear Research, where users and staff rely on several IT services as a support for their daily work. For software development such services include issue tracking, software repository hosting and version control. Scientific computing relies on services such as batch processing or volunteer computing portals. All of these services run on a common cloud infrastructure with a common monitoring system in place.

This thesis presents the development of a monitoring framework that was created to improve the detection of service degradation and to reduce the amount of noise that a service responsible is exposed to, as they maintain a service, which is equipped with a monitoring system that creates and sends a large amount of alerts. Two of the main challenges were to take appropriate advantage of generic and existing monitoring components in the planning and implementation phases, and to follow the ongoing development of the monitoring and cloud management tools at CERN.

Two different use cases are considered: a large service where negligible alarms occur, and a small service where single alarms are important. The conclusion from testing the framework prototype on real data in a large service, as well as an interview with two service responsibles, indicates that the filtering capability provided by this framework can be an influential tool for efficient fault detection and correction in real-world scenarios. In addition to these cases, the thesis describes the existing infrastructure, large-scale monitoring in general, as well as recommendations for future work and extensions for the CERN IT monitoring infrastructure.

Keywords: Alarm aggregation, Monitoring, Alerting, Cloud computing, CERN, European Organization for Nuclear Research

AALTO-UNIVERSITETET
HÖGSKOLAN FÖR TEKNIKVETENSKAPER

SAMMANDRAG AV
DIPLOMARBETET

Författare: Carolina Lindqvist

Titel: Ett alarmaggregeringsramverk för kritisk IT-tjänstemonitorering vid CERN

Datum: 10.6.2015 Språk: Engelska Sidantal: 8+71

Högskolan för teknikvetenskaper

Professur: Security and Mobile Computing

Övervakare: Prof. Keijo Heljanko, Førsteamanuensis Magnus Lie Hetland

Handledare: M.Sc. Alvaro Gonzalez Alvarez

Den ökade uppkomsten av molntjänster under det senaste decenniet har lett till förändringar inom informationstekniska tjänstearkitekturer. Som en följd av detta håller man även på att omskapa de existerande lösningarna för tjänsteövervakning (monitorering) samt vidareutveckla dessa, i avsikt att underlätta driften av sådana virtualiserade, skalerbara samt dynamiska miljöer. Detta diplomarbete är utfört som en del av författarens tid som praktikant vid CERN, den Europeiska organisationen för kärnforskning, där användare och personal förlitar sig på en mångfald av IT-tjänster som ett stöd för sitt dagliga arbete. Alla dessa tjänster körs på en gemensam molnplattform och övervakas av en gemensam programvara.

Detta diplomarbete presenterar utvecklandet av ett monitoreringsramverk som skapades för att göra det lättare att upptäcka försämrad tjänstekvalitet. Ett annat nämnvärt syfte var att minska informationsflödet från det existerande alarmeringssystemet. Många tjänsteansvariga utsätts nu för en flod av alarm då något allvarligt sker i en servergrupp, då de istället kunde notifieras en gång. Två av de största utmaningarna med arbetet var att tillgodogöra sig existerande monitorerings- och förvaltningsverktyg som finns i bruk vid CERN, samt följa med utvecklingen av dessa verktyg.

Två separata användningsfall beaktas: en stor tjänst där oväsentliga alarm förekommer, samt en mindre tjänst där varje enskilt alarm är viktigt. Slutsatserna från att ha experimenterat med prototypen av ramverket på äkta data i en stor tjänst, samt en intervju med två serviceansvariga, tyder på att ramverkets förmåga att verka som ett informationsfilter kan vara ett betydelsefullt verktyg för att effektivt uppdaga och korrigera fel. I tillägg presenterar arbetet den existerande infrastrukturen, storskalig monitorering i allmänhet, samt rekommendationer för framtida arbete och vidareutveckling av infrastrukturen för monitorering vid CERN.

Nyckelord: Monitorering, aggregerade alarm, notifiering, molntjänster, CERN, European Organization for Nuclear Research

Preface

The work that this thesis describes was carried out as part of a studentship at the European Organization for Nuclear Research, CERN. I am eternally grateful to yet again be granted the chance to conclude my work towards pursuing a degree, at this inspiring, intriguing and international organization.

I would like to thank my thesis supervisors Assoc. Prof. Keijo Heljanko and Assoc. Prof. Magnus Lie Hedland for the wise guidance and advice they patiently and regularly offered as I worked on this thesis. I am very grateful to M.Sc. Alvaro Gonzalez Alvarez, my thesis advisor and supervisor at CERN, who always kindly answered my questions and seeked to make time for guiding me, even as he was buried in work. I could not have wished for better supervisors than all of you. I also wish to thank the batch team and the monitoring team for patiently explaining and discussing the monitoring infrastructure, as well as pointing me in the right direction with my own work. Last but not least, I wish to thank my section for supporting me in this work, and several colleagues, fellow students and activity clubs for making my time here even more enjoyable.

Furthermore, I proceed to express my gratitude towards everybody involved in the Master's programme I study in, NordSecMob. It has been a great journey the last two years, from Finland to Norway, and a huge learning experience in many ways. I have met the most amazing friends and co-students from literally all around the world during my studies. I am sure we all remember the meeting that was organized for NordSecMob students in Røros, and the conference in Tromsø as we gathered once again to explore topics on information security ... and to hunt for the aurora borealis in a desolated swamp area. I wish you all a bright and adventurous future! I would like to thank my family, as well, for encouraging me to study diligently, learn languages and travel abroad to explore the world. Thank you also, my dear friends in Finland, and many other countries, for the merry moments we spend together. Finally, a heartfelt thank you to Olav, for being the wonderful person you are.

Regarding the work, during these four-five months, I sometimes struggled, but each day I managed to find a novel concept or a beautiful piece of code to enjoy and learn from. It has been a long way, and I am happy to have reached this milestone, to be able to dive into the future work I have outlined in the concluding section. This is also a significant step for earning my Master's degrees from Aalto University (formerly Helsinki University of Technology) and the Norwegian University of Science and Technology. My desire for knowledge has not been stilled yet, and I hope to continue my studies after graduating from this excellent programme.

Geneva, Switzerland, 30.5.2015

Carolina Lindqvist

Contents

Symbols and abbreviations

Abbreviations

AI	Agile Infrastructure group
ALICE	A Large Ion Collider Experiment, an LHC experiment.
API	Application Programming Interface
ATLAS	A Toroidal LHC ApparatuS, an LHC experiment.
BOINC	Berkeley Open Infrastructure for Network Computing
CC	CERN Computing Center
CEP	Complex Event Processing
CERN	European Organzation for Nuclear Research
CLI	Command Line Interface
CMS	Compact Muon Solenoid, an LHC experiment.
DCS	Detector Control System
DB	Database
FE	Functional Element
FSM	Finite State Machine
GNI	General Notification Infrastructure
GUI	Graphical User Interface
HDFS	Hadoop File System
HG	Host group
HTTP	Hyper-Text Transfer Protocol
ID	Identification
IT	Information Technology
IO	Input/Output
JSON	JavaScript Object Notation
LEMON	LHC Era MONitoring, part of the CERN monitoring infrastructure.
LHC	Large Hadron Collider
LHCb	Large Hadron Collider beauty, an LHC experiment.
MQ	Message Queue
QA	Quality Assurance
REST	Representational State Transfer
SCOM	System Center Operations Manager, management system for datacenters.
SLA	Service Level Agreement
SNMP	Simple Network Management Protocol
SNOW	Service NOW, a commercial service management software.
SSO	Single Sign-On, an access control property where the user only authenticates once and can access several services with the same authentication.
SQL	Structured Query Language
TDD	Test-Driven Development
UPS	Uninterruptible Power Supply
VCS	Version Control Systems
VM	Virtual Machine
YAML	YAML Ain't Markup Language

Glossary

Metric	A class that will measure or calculate one or more values which are sent to the lemon agent. The metric is identified by a unique number and its name.
Sensor	A script (Perl, Python) written by a user. It contains a collection of measured metrics.

1 Introduction

This Master's thesis presents a framework for monitoring and attempted automatic failure correction based on preconfigured policies. The work was done as part of a year-long internship in the Infrastructure Services section in the IT Platform & Engineering Services group at CERN, as an effort to improve the section- and group-wide monitoring. The framework was designed and created to solve the problem of aggregating notification data from several hosts and to detect failures based on the aggregation.

The current monitoring infrastructure generates notifications for errors in single hosts, and it is possible to manually query for aggregation data in the presentation layer of the monitoring infrastructure. However, the created framework is able to automatically monitor and collect the notification messages as they are sent and group the notifications by the host group they are sent from. For each host group, a policy can be defined containing information about the notifications that are ignored or counted, as well as actions that are taken if a defined amount of hosts in the same group are sending notifications. The actions include arbitrary shell commands or scripts whose execution results can be sent as e-mails to the service responsible or simply e-mails that are sent stating that something is wrong.

The added benefits of this thesis work include a possible reduction of the time spent investigating the impact of a service failure, the chance for a service owner to gain an overview of the failing machines in a host group in a single e-mail message, the opportunity to automatically attempt to correct the failure by configuring an action, and the convenience of configuring an individual policy for each host group. Challenges were also encountered during the development process. For example, the framework design was constrained by the architecture and software used in the current monitoring infrastructure, and limitations had to be implemented to avoid accidental damage or e-mail floods caused by a misconfigured framework. The final version of the prototype was tested on real data by allowing it to run for several days while collecting both the received messages and the actions in log files that the framework generates. This was done in order to investigate the ways in which the framework behaves both on busy and calm days. The conclusion was that the framework can act as a configurable filter and aggregation tool for the notifications.

The thesis is structured as follows: Section 1 briefly describes the problem and research goals, as well as discusses research limitations and similar work. Section 2 presents the background to monitoring in large-scale computing environments and a brief overview of the different meanings the concept of monitoring holds within. In Section 3 the existing CERN IT monitoring infrastructure is presented, broken down into the different stages and components that exist in the monitoring system. Section 4 explains the architecture of the framework that was created as an extension of the existing monitoring infrastructure. In Section 5 the automation capabilities of the framework are discussed, with Section 6 showing the practical tests and experiments that were done with the framework. Finally, Section 7 explores future work to be done, a brief evaluation of the used methodologies, and the conclusions drawn from the current work.

1.1 Problem definition and research goals

This thesis work is an attempt at solving the problem of gathering notification data from several groups of hosts, and aggregating that information to form a conclusion, whether the notifications indicate a problem with a single machine or a group of machines. In the currently existing monitoring system it is possible to monitor the state of individual machines and automatically notify the service owner in the case of a service malfunction. Since it is considered better to have more false positives than false negatives, the alarms can lead to copious amounts of e-mails and can give rise to automatically opened tickets in the incident ticket system that is used. It is for example possible that a machine is under high load (as it is supposed to be) during which time the current monitoring system cannot reach the machine. If the state of being unreachable persists, there is a risk that the monitoring system believes the machine to be malfunctioning and as a consequence sends out a notification. As the service owner receives the notification, valuable time is spent investigating a possible problem, only to conclude that the single notification was a false alarm.

The aim of this thesis is to develop a framework that is able to aggregate alarms from groups of machines and trigger different actions based on sums of individual notifications. The framework will be able to keep track of the state of a group of machines, which improves the diagnosing process. It will be possible to configure whether to inform the service owner when a single machine in a group has an issue or whether to inform the service owner only when a significant number of machines are sending notifications. The framework will also be configurable to run a command, for example, collecting information from log files, which is sent out along with the notification e-mail, or sending a ping to another service to see if the main service is malfunctioning as a consequence of another service malfunction.

The research goal is to develop a framework that is able to correctly identify and discern between situations when there is a small temporary disturbance in a group of machines, and when a larger problem has built up, and only then notify the service owner. Challenges related to this work include, but are not restricted to, the following list: storing and knowing the state of the host groups at a certain point in time, the ability to extend the framework with new actions, keeping the framework simple, exploring the available technologies and programming libraries that are appropriate to use, as well as creating a software that is easily can be integrated to the existing systems and still be maintained when the software is finished.

1.2 Research limitations and similar work

One clear limitation is that there needs to be a policy defined for each host group that is to be monitored. It means that it will take a certain amount of time and knowledge to be able to know which notifications are important for a certain host group, and which notifications can be safely ignored. It also means that the granularity is on a host group level. It is for example not possible to ignore one machine that is behaving differently from the others in the host group.

Since the idea of a host group is to combine similar machines, e.g., hosting a

service, this granularity level is deemed to be sensible. On the other hand, if one machine has several errors, this may not be noted, since the host group is regarded as a whole. This may require some special consideration as the configuration files for smaller host groups are defined, in case it is important to differentiate between knowing that one machine has multiple errors and knowing that there is some error in one machine. Another limitation is that it is important to follow what the CERN monitoring team is working on, in order to avoid duplicated work. This is resolved through meetings and discussions. It is also not possible to freely choose and implement any software that might be found suitable for the task that is at hand, since the seamless integration of this framework with tools that exist and are used in the CERN IT infrastructure is important.

Furthermore, it is worth noting that the CERN Computing Center (CC) is currently working on renewing their workflow for partial shutdown of the CC in the case of a cooling problem or power incident [41]. If an incident occurs, there is an emergency UPS (uninterruptible power supply) system that will provide the machines with power for a short amount of time. In this case it is important to be able to shut down less important nodes, in order to save UPS power for the critical infrastructure. The proposal now, is to automate the shutdown process by creating a collection of scripts that would react if a cooling or power cut alarm is received, and the situation does not seem to resolve itself after an amount of time has passed. The problems are similar to the problems in this thesis work; essentially assuring that no false shutdown is triggered, notifying the owners of the affected (virtual) machines, authenticating the notifications to prevent malicious shutdown, and ensuring the simplicity of the scripts to prevent unmaintainability. Similar work from other academic areas that is related to monitoring and autonomous repairing action is presented in Section 2.

In 2014 an analytics working group was formed at CERN IT to unify efforts towards understanding and using the vast amount of information that can be gathered from various sensors and log files in machines running services at CERN. The goal is to gather common data to a single store and create common tools to facilitate the work between groups in the IT department for aggregating and analysing such as usage data and investigating trends. One of the four large experiments, the ATLAS experiment, has also formed an analytics working group, which for example is concerned with analysing and possibly predicting which datasets are accessed more often than others, in order to be able to grant high availability for such datasets.

Lastly, it is worth mentioning a monitoring and alerting solution, Metis, that monitors the messaging infrastructure at CERN. This software was released in April 2015[1]. The author of this thesis found the work that was done on Mesis in late April 2015, and stayed in touch with the autor of Mesis to learn the similarities and differences between this framework and Mesis. Mesis is based on Esper[2] which is an engine for complex event processing (CEP). CEP is an independent area of research involving processing individual events by, for example, creating an aggregation that calculates a real-time average of a value that consists of several other streams of

[1] http://lucamag.web.cern.ch/lucamag/

[2] http://www.espertech.com/esper/index.php

values [7]. In real applications, the calculations can of course be more complex than an average value.

The most notable difference is that Metis monitors the status (metrics) of the CERN messaging system [57], whereas the framework is meant for notification aggregation, on a host group level for machines that are part of the IT infrastructure at CERN. Other notable differences are that Mesis has been under development for a significantly longer time and that the author has previous experience from working on monitoring systems at CERN, and using the technologies that the software is based upon [57]. A similar goal is the plan to integrate Mesis with the ticket creation infrastructure which is described in Section 3.5 and to analyse logs for gaining information on the current state of the messaging system.

Regarding limitations, with respect to the three properties that define Big Data, "volume, variety, and velocity" [46], the framework itself will be designed to handle notification messages from potentially around 3000 host groups of which around 90 host groups are of interest. In reality, it is considered fairly unlikely that every single host would trigger a notification message, since in such cases it is likely that the monitoring system that transmits the messages would also be impacted by the same problem (e.g., a large-scale power outage). The messages have been observed to arrive at a pace of around 400/minute depending on how busy the system is. The number includes duplicates of the notification message, since notification messages are continuously sent out until the initiator of the notification is resolved, whereupon a (single) message indicating that the notification is closed is sent out.

2 Background

This section contains a short discussion on service monitoring in computing clouds, and monitoring in general on a large scale. Academic work with different approaches for monitoring as well as four different popular monitoring tools are described. The different types of monitoring at CERN, as identified by the author, are presented. Finally, the CERN IT services which are targeted by this thesis work are shown in relation to the use cases and needs that were discovered before the development of the monitoring framework was commenced.

The services and the monitoring which are discussed in this thesis work, are hosted in computing clouds. By the NIST defintion [61], cloud computing is a pattern that allows for sharing a collection of "configurable computing resources" which can be requested, used, and liberated by the users with as little work and intervention as possible being required by the maintainers and providers of the service. One of the five dictinctive characteristics NIST lists for cloud computing, is that it is a "measured service" [61]. This is meant in a sense that the use of resources should be measurable for accounting purposes, which adds transparency for the service user, and allows the service provider to monitor the usage (e.g., for service development purposes). The definition also describes four different cloud models; private and public clouds for private and public use, community cloud, which is shared by a community with a common cause, and hybrid cloud which is a combination of at least two other cloud types which are joined by common technology to ensure data compatibility.

The cloud infrastructure which the services discussed in this thesis are based on, is a private cloud that partially can be described as a community cloud, namely the CERN Cloud Infrastructure[3]. This infrastructure is described in more detail in Section 2.3. It is worth noting that the CERN cloud is distinguished from the Worldwide LHC Computing Grid (WLCG) which is used by the High Energy Physics (HEP) communities at CERN and globally, notably, for analyzing data from LHC, the Large Hadron Collider. The Grid computing model is different; a predecessor to cloud computing, which enables sharing of computing resources across (virtual) organizations in a fashion similar to cloud computing, but is oriented towards solving specific computational problems rather than providing a general service [34]. Clarifying the concept and comparing computing resources to cake batter, the grid can be seen as a whole chocolate cake whereas a cloud can be seen as a plate of individual brownies. This facilitates understanding that a cloud also can be part of a grid, which for example has been demonstrated by the CMS experiment. The computing farm at the experiment, whose nodes are normally used for pre-processing detector data as the LHC is in operation, can be turned into a cloud which is attached to the WLCG at times when data is not available [22] (e.g., during large shutdowns). The reason for this solution is, that the machines are too valuable a resource to leave partially unused during the time when no data can be taken from the LHC. The same idea has also been cultivated in the LHCb experiment [16].

Automated monitoring in cloud environments such as the ones discussed above,

[3] http://clouddocs.web.cern.ch/clouddocs/

is becoming increasingly more important, as more and more data is starting to be collected for analytics purposes. The service providers can be interested in information regarding how their services are used by the users, the cost-effectiveness of the service, automated scaling based on the usage, automated problem solving for known occurring issues, or simply gathering statistics that can be provided to users in order to improve the transparency of the service from a user's point of view. Since the amount of data that can be collected from large-scale computing environments is quite vast, it can also be classified as "Big Data", which the Oxford English Dictionary [1] explains as data of such a large size, that the handling of that data is a challenge of its own.

There is also an academic interest in quantifying metrics gathered from cloud environments. As can be seen in Fig. 1 the amount of articles found in the DBLP computer science bibliography, which are related to analytics and big data as well as cloud monitoring has increased steadily since 2009-2010. It can be noted that at the moment of writing (April 2015) almost three quarters of the data for 2015 is yet to be produced, thus 2015 was left out. The data was gathered by performing a search, downloading the results in JSON format, and parsing it through a very simple script. Five results on cloud monitoring younger than 2009 were related to Meteorology and thus omitted. The purpose of this figure is to illustrate the rising interest in monitoring and data gathering in relation to the concept of cloud computing and large amounts of data.

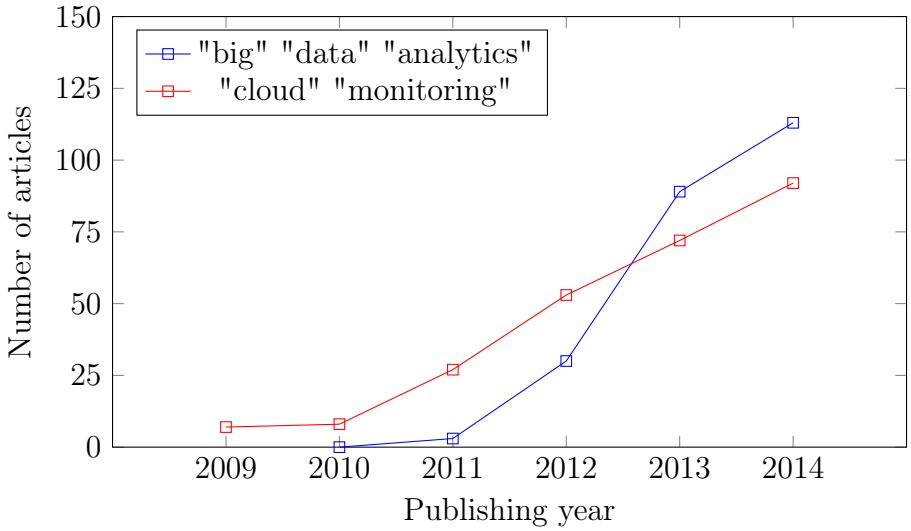

Figure 1: Amount of articles found when searching for the terms shown in the plot legend from the DBLP[5] computer science bibliography on 15.04.2015.

[5] http://dblp.uni-trier.de/

2.1 An introduction to large scale monitoring

The need for monitoring rises from the necessity of being able to determine the state of a service or hardware that an organization, company or private entity is maintaining. Whether it be providing a service for a large user community or several customers, or for personal use, it is essential to understand and be able to discover if a service is degraded, unresponsive or has other critical issues before the error affects the user experience. Taking this one step further, monitoring is also an important part of controlling Service Level Agreements (SLAs) which are part of the Quality Assurance (QA) process for a service [23]. An SLA is a type of contract between service providers and service consumers, which explains what the service is supposed to accomplish for the user (e.g., describing the expected level of availability).

Monitoring can be divided in three stages [81]: data collection, data analysis and decisions. The three stages can be implemented as automatic or partially automatic functions which require active human intervention to function. Collection and analysis can be done by a program that collects information and does the analysis, for example based on controlling that predifined threshold values are not exceeded. Decisions or actions can also be programmed, but in several cases the decisions are made by humans, which are notified by the system as a threshold is breached.

In cases of simple monitoring, the analysis stage can also be done by a human. One common example is for a user to study the system state using a UNIX tool like the `top` task manager. In more complex systems which consist of several thousands of values and several sensors which measure these values, it can quickly become difficult to grasp the whole view of a system. In this case it is more efficient to start offloading the task to a programmatical solution which will alert or alarm a human when unusual values appear.

Automated monitoring can be done on a small scale using a centralized service which collects monitoring information from a number of hosts. Modern monitoring systems are evolving away from centralized solutions as large scale systems become more common. When monitoring on a large scale, it is important to eliminate single points of failure, such as those which can exist in a centralized solution. One of the simplest methods for preparing to scale up is to structure the data collection as a tree [81]. But, as the scale increases, there will be a need for having several levels in the tree structure and a need to implement a distributed solution that eliminates single points of failure.

This has caused peer-to-peer communication to appear as a new medium [81]. One such example is the proof-of-concept monitoring system presented in [51] which mainly aims to present an example architecture for a peer-to-peer-based monitoring system. The motivation for exploring peer-to-peer technology is that it can scale and adapt efficiently, in particular in elastic systems where hosts are spawned and terminated frequently. The research in [51] shows that the proof-of-concept is able to scale, but the benchmark is limited to a cloud containing 17 virtual machines (VMs), which cannot be compared to an infrastructure of thousands of VMs.

There are more complex monitoring systems, for example, HOLMES [76], which collects metrics, and uses the previously mentioned CEP engine Esper as well as

machine learning techniques to automatically learn the normal behavior of a cluster. The aim of HOLMES is to reduce the time that a service responsible spends on configuring a monitoring tool, by allowing the monitoring tool to identify a baseline for the monitored machines and alerting human operators when an anomaly is detected.

One can also find monitoring systems with self-healing properties, e.g., SHOWA [56]. It is a modular framework that monitors web applications, conducts an analysis searching for anomalies, and attempts to repair the defect if any is found. The analysis is based on the idea of correlating the web server response time with degraded performance. This is done by an anomaly detection module, which can launch a module for analysing three different types of failures, for example analysing whether the anomaly affects the performance of the service at all. Taking the self-healing property one step further, one novel approach, [26], suggests tagging streams of monitoring data with annotations and using Semantic Web technologies (for reference see, e.g., [10]) in combination with Stream Reasoning (see, e.g., [80]) for analysing the flow of monitoring information during certain intervals (i.e., time windows). The concept of inspecting streams of data is illustrated in Fig. 2. However, this technology has problems regarding performance and scalability, notably regarding the speed at which large amounts of data can be analysed [26].

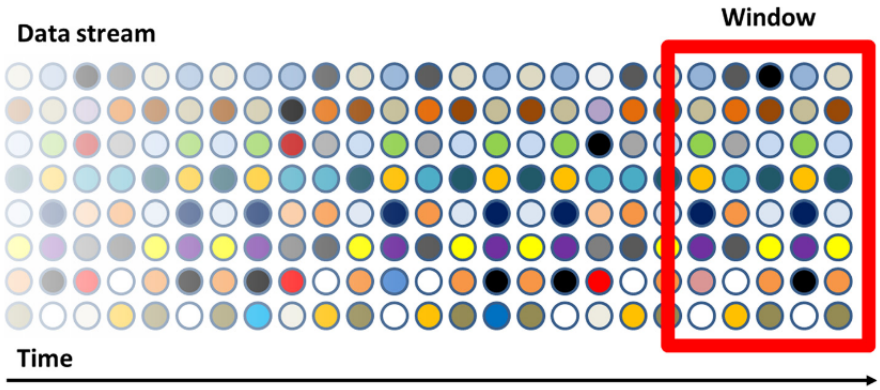

Figure 2: Inspecting a window of diverse streamed (monitoring) data [26].

Monitoring large data streams is by itself not a recently emerged topic. For example, at CERN, detector data streams of a size of 100 GB/s are not exceptional [2]. Other applications which are known to handle and generate large amounts of data are large sensor networks, high-frequency trading, and real-time sales analysis [32]. Much academical effort has also been invested in doing research on monitoring (and anomaly detection) from a computer security point of view, but these themes are out of the scope for this thesis. Moving from academical research to investigating the features of four popular monitoring tools: Nagios, Cacti, Zabbix and Ganglia, one can observe differences in the features they offer, which shows why it is important to be able to choose the right tool when attempting to monitor a set of machines.

An elementary Nagios installation consists of a monitoring server that performs checks on a set of machines. A check is a measurement of a define value. The

monitoring server has configuration files, which for example define the checks to be performed, the time during which checks are performed, and whom to alert in case a check has an erroneous value. Configurations can be shared by machine (through inheritance in the configuration file), which reduces the amount of necessary configuration. Various plugins and extensions exist due to the large user community. Since having a single monitoring server is an obstacle for large scale monitoring, there are Nagios extensions and installation architectures which distribute the monitoring work among several nodes or create a hierarchy of monitoring servers [81]. As an example, Yahoo Inc, is listed[6] on the Nagios home page having an installations consisting of 2000 Nagios servers monitoring 100000 hosts and 200000 services.

The Cacti monitoring tool is a graphing tool which mainly is meant for monitoring network equipment, by using the Simple Network Management Protocol (SNMP) [79]. The configuration is done in a web interface containing relevant fields to be filled in. This can be a tedious and manual task, but templates exist to facilitate the process. The configuration specifies the data that will be polled for from the device, and graphed in the Graphical User Interface (GUI) of Cacti. Different notifications and reports can be constructed and sent to the service responsible using plugins. It is also possible to use Cacti in a large scale deployment, with the largest known amount of possible data sources in 2012 being 1 million [70].

Zabbix consists of a complete monitoring system with graphing and notification capabilities [74]. It can work using remote checks or by having an agent running on the monitored client. The monitored values can be configured to trigger an action if they exceed a certain threshold. The thresholds can be set to different levels of severity with different actions. A properly set up Zabbix installation (e.g using Zabbix proxies) is capable of operating on a large scale, handling over 9000 readings per second [24].

Ganglia differs from the three other chosen monitoring tools by more specifically being a cluster or grid monitoring tool [60], thus aimed at highly distributed systems, and by originating from an academic environment; the University of California, Berkeley. It is made for monitoring large-scale environments consisting of several clusters, with an ability to handle clusters containing over 2000 nodes [78]. The early idea was a monitoring tool for a single cluster, but it later transformed to cover computing grids and other models for distributed computing.

These four tools are among the older and traditionally used monitoring tools. For an extensive survey the reader is advised to study [81], which also presents a taxonomy for monitoring tools that can be seen in Appendix A. As can be concluded, the term "monitoring tool" is used in a fairly general manner, including full-featured tools which provide functionality for many different stages of monitoring (e.g., collecting, analysing, decisions) or simple tools which cover only one of the stages. It is also possible to combine several tools which perform individual tasks to create a customized monitoring infrastructure, and even introduce customized components such as plugins or software to suit the existing needs. This is one reason for why monitoring can turn out to be a complex task. As a result, certain tools and

[6] http://users.nagios.org/directory/Yahoo%2C-Inc/details

combinations that are observed to work well become popular and emerge as so called de facto standards.

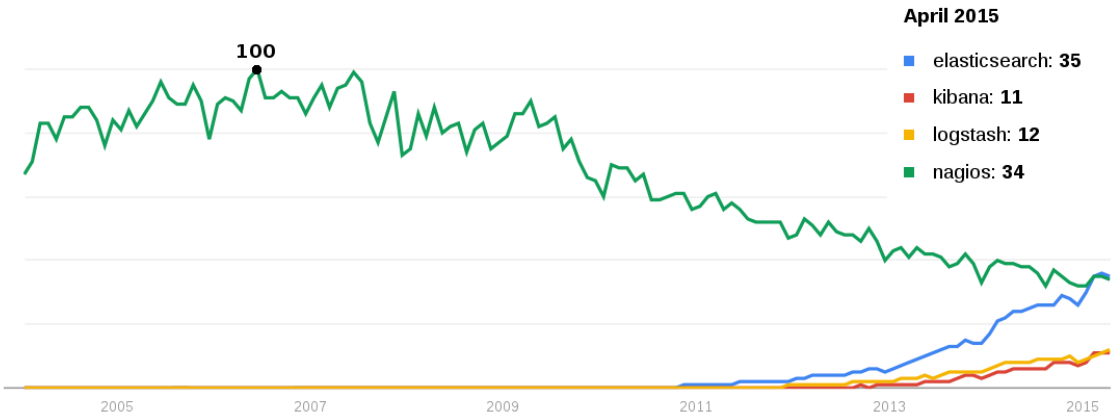

Figure 3: Google Trends[8] analysis in April 2015 for new and old monitoring tools.

One such combination whose popularity has increased steadily is the Elasticsearch, Logstash, Kibana (ELK) stack that will be presented in detail in Section 3, as it is partially (but substantially) used in the CERN monitoring infrastructure. A graph illustrating the rising interest in these three softwares as search terms in Google Trends as of April 2015 can be seen in Fig. 3. The $y - axis$ on the graph represents the amount of searches for a specific term in a normalized form as a percentage of the highest spot on the graph. The point marked '100' represents the highest amount of queries and the legend numbers represent a percentage (i.e., the amount of searches for 'elasticsearch' is 35 percent of the amount of searches for 'nagios' in late 2006).

2.2 CERN IT monitoring in general

The department of IT at CERN is divided in groups, which are divided in sections [30]. The groups are separated based on the different functions and services they maintain and develop (e.g., storage, databases or datacenter facilities). As an example, the section for IT Monitoring Management & Tools develops and maintains the general monitoring infrastructure that is described in Section 3.

In addition to this infrastructure, groups and experiments may have their own customized monitoring solutions. The reason for this is that the general infrastructure provides services for monitoring system statistics on a machine level, whereas a group or experiment may need to monitor specific hardware, for example physical sensors for measuring a voltage, or a specific service, for example version control services.

The different levels of monitoring at CERN as described previously is illustrated in Fig. 4. The author has illustrated a rough classification on top of the background image depicting an aerial view of CERN [18]. The yellow quarter of a ring visible in the photograph illustrates the Large Hadron Collider (LHC) ring, and three of the

[8]Data Source: Google Trends (www.google.com/trends)

four LHC experiment sites are shown in the picture (LHCb, ATLAS, ALICE). The main CERN site in Meyrin, where the datacenter and IT department are located can be seen to the right of the ATLAS experiment. The remote CERN Prévessin site can be seen below the LHCb experiment.

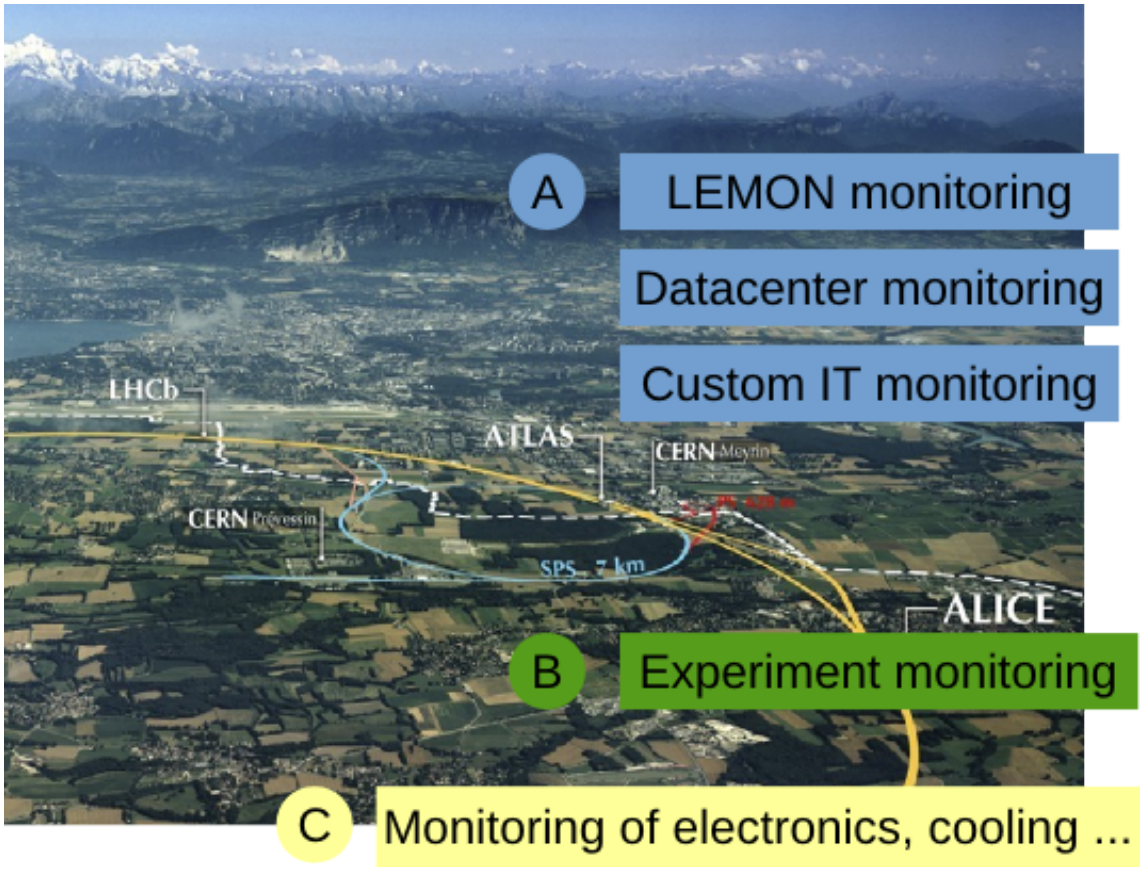

Figure 4: A rough classification of monitoring at CERN, drawn upon a photo [18] by CERN.

The monitoring in *A* refers to the monitoring infrastructure used in the IT department and the datacenter which are located physically close to each other on the main site. This monitoring can be divided in three parts: The LEMON (LHC Era Monitoring) monitoring infrastructure provided by the monitoring team, datacenter monitoring solutions, and custom IT monitoring developed in groups at the IT department. The reason for this grouping is that the datacenter monitoring may have different requirements for example for monitoring the 100 PB tape storage [65], compared to monitoring a web service that is provided by a group. One example of such a project is Metis (the monitoring system for the messaging infrastructure at CERN) which was discussed in Section 1.2. However, the aim of the LEMON monitoring infrastructure is to provide a service that limits the need for custom solutions and implements common monitoring scenarios, which also makes it easier to share and combine monitoring data between different services [3].

The monitoring in *B*, experiment monitoring, refers to the various types of

monitoring that the experiment sites require. The monitoring is a mixture of IT monitoring and monitoring for the experiment infrastructure; such as particle detectors[9], electronic devices, pressure levels, cooling and other relevant measurements. Each of the four LHC experiment has a small datacenter (computing farm) [2], [12], [28], [67] on the scale of a few thousand nodes for rapid on-site detector data processing. These nodes give rise to the need for IT monitoring, for which a monitoring system such as LEMON or another monitoring software may be used. As an example, the ALICE experiment has been using LEMON, but has chosen to start using Zabbix[10] in 2015 after an evaluation of six monitoring tools [77]. Zabbix is also used for monitoring systems related to safety at CERN, such as access control for the CERN premises, servers for surveillance cameras and personnel safety systems [44].

The experiment infrastructure monitoring is similar to the monitoring in C. The C indicates the monitoring of electronic devices installed in the LHC tunnel. One example of such a monitoring system is a radiation level monitoring system [73], [82] consisting of hundreds of radiation measuring devices which the LHC tunnel is equipped with [68]. This data can be collected for creating real-time dashboards shown in the control room for the LHC [82]. Another example is monitoring the various gas delivery systems for the LHC [43], which is done by gas analysis modules measuring (e.g., the flammability and gas mixture composition), and alarming on threatening values.

On the experiment site, there is additional monitoring for the detector, which itself is completely different depending on the site. Monitoring the detector includes monitoring the state of the hardware and the data quality. For instance, the detector control system (DCS) in ALICE which controls 18 different detectors with 15000 different nodes (electrical devices and data channels) divided in subsystems and child-parent hierarchies [21]. The system state is based on threshold logic, one of the most common monitoring system paradigms [81]. The parts are modeled as finite state machines (FSM). If a certain amount of child nodes show an erroneous value, the parent will change its state to an error state. The system presents the current state on a dashboard as seen in Fig. 5. These dashboards are monitored 24/7 in a control room by trained operators, who will be alerted by an alert system to investigate the error state.

Out of these three types, this thesis focuses on the first type (A) of monitoring. The monitoring infrastructure provides services for creating customized sensors for gathering statistics from a machine. These statistics are automatically transported, stored and graphed on a dashboard. Several default statistics are also gathered from the machines without a need for the user to write code for a sensor. These statistics mainly describe basic CPU, IO, memory and network usage. The infrastructure is described in detail in Section 3.

[9] http://home.web.cern.ch/about/how-detector-works
[10] http://www.zabbix.com/

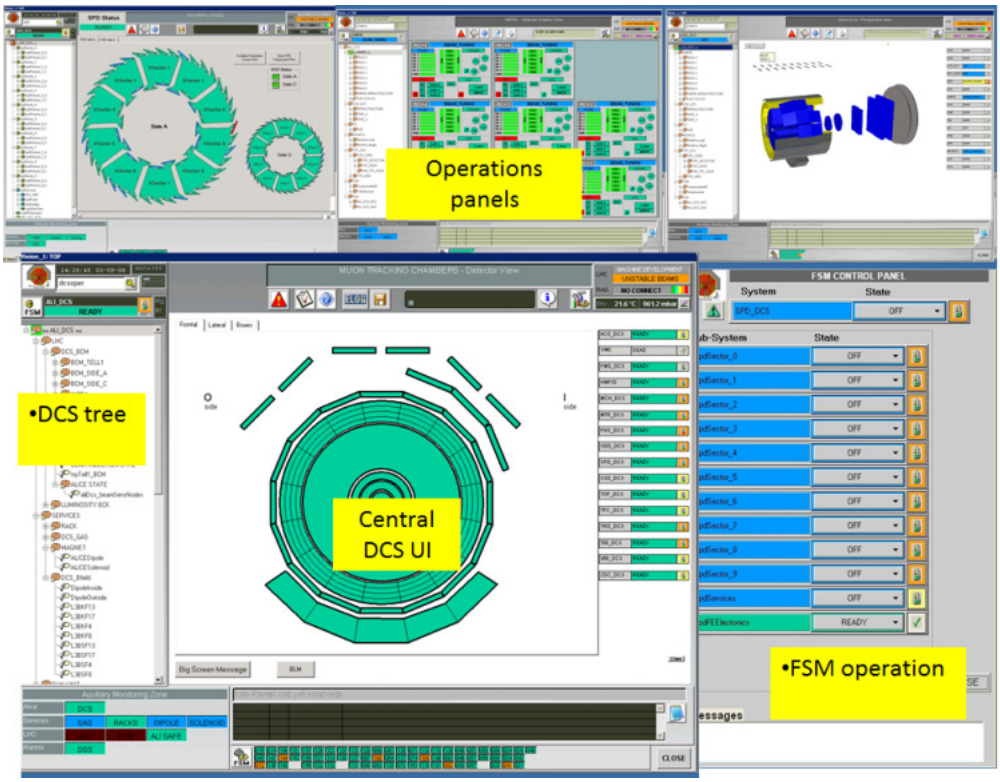

Figure 5: A view of the dashboards in the ALICE detector control system [21].

2.3 Business critical IT services at CERN

This thesis work is carried out at the IT-PES (Platform and Engineering Services) group [31], in the IS (Infrastructure Services) section. The main services that the group provide for the CERN users include the batch service, version control systems and issue tracking (e.g., GIT and JIRA), configuration management services (e.g., Puppet), the BOINC (Berkeley Open Infrastructure for Network Computing) volunteer computing service, and a number of other services managed by three separate sections. The services are hosted on both physical and virtual machines (VMs), the choice depends on the service. This thesis will focus on virtual machines, as the majority of the services concerned run on VMs.

The virtual machines in the CERN cloud infrastructure are OpenStack[11] machines optionally managed by the Puppet[12] configuration management software [4]. The machines are organized in host groups with each host group having a corresponding configuration file (puppet manifest) or being configured using a parent host group's configuration. The host group configuration can be a folder containg other host group configurations, a file containing the host group's configuration, or both. The concept of host groups and configuration files is illustrated in Fig. 6. To facilitate the configuration of the notification aggregation framework that is described in this thesis, the framework configuration files for each host group are also grouped in the

[11] https://www.openstack.org/
[12] https://puppetlabs.com/

same manner.

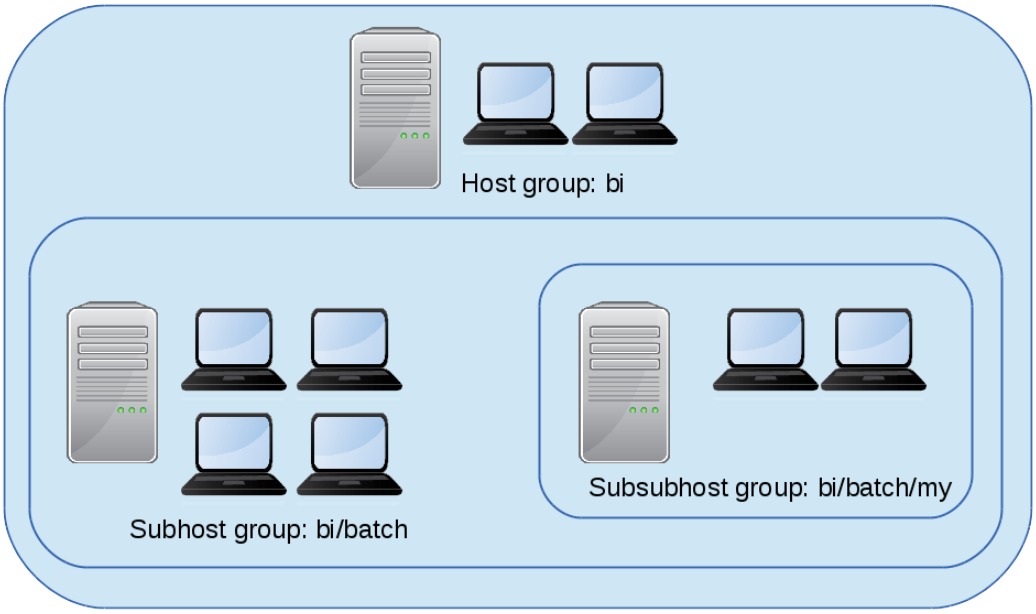

Figure 6: The host group is the parent of the subhost group.

2.4 Use cases and needs

Service monitoring at CERN is important not only for ensuring that the users are able to benefit from a service that works as expected, but also for identifying possible performance issues and optimizations such as bottlenecks, and for accounting purposes [4]. Efficient storage, data extraction (e.g., performing a search), and data graphing abilities are helpful for these tasks. Previously, it was difficult to use the data gathered from the LEMON monitoring tools, since data was stored in separate storages and highly customized code was necessary for accessing this data. Resource accounting data was available for certain services, whereas other services required several different data gathering tools [4]. To improve the situation, a new architecture for the monitoring infrastructure was designed and implemented, which consists of a unified storage for monitoring data, a search feature, data graphing features, and an automated alert and ticket creation system. This architecture is described in Section 3.

One issue that can be identified in the architecture, is that a service which is hosted in a large host group can give rise to several alerts and tickets, which in

turn will create a storm of e-mails and an overflow of information for the service responsibles, in addition to communication from users, who express their concerns when the service does not work. Important time that could be spent solving the problem has to be used for updating the users and analysing new e-mails or tickets. The same issue is present in monitoring that is based on cronjobs which send alerting e-mails as they fail. These scenarios are not uncommon at the appearance of an issue. One possible solution is to shut off the alarms while resolving the issue, and re-enable them later, but this imposes a certain risk. It is also a concern that service responsibles will start ignoring every notification in case most of them are unimportant. Theoretical findings showing this phenomenon has been seen in the case of [69], a study on human behavior and information overload.

A need to dampen unimportant notification and rise awareness of important notifications was soon identified when defining the use cases for the monitoring framework described in this thesis. The first attempt at a solution, by the author, was to investigate the existing monitoring framework and look for tools that could fulfill this task. After an initial analysis, it was found that an aggregated view of the data is available, but only by visiting a dashboard, manually creating the aggregation by querying the data in the dashboard, and visually analysing the generated graphs in search for anomalies or warnings.

A larger investigation was undertaken by the author, to identify tools that automatically could analyse the same data that the monitoring framework was storing, and rise an alarm in case a serious issue had happened (e.g., several hosts in the same host group going down). One of the promising candidates was the Riemann monitoring tool[13] that is able to aggregate data from several machines and send notifications by e-mail if errors are encountered. Another issue was log analysis, which is often done manually as the service responsible has discovered that there is a problem in some service. One idea was to filter such logs (e.g., Apache web server logs) using Logstash[14], which had the added benefit of being compatible with the Riemann monitoring and alerting tool.

Theoretically, using these tools, it would be possible to identify certain types of log messages, which indicate problems, that would trigger an alert in Riemann. Doing only this, log file parsing and alerting, is possible in the existing framework, for example by creating a sensor script for the parsing, and enabling an alert. However, there are downsides to this approach. First, there is currently no direct way of automatically aggregating notification information in real-time (from a users point of view) from several different machines. It can be done afterwards, by manually creating a query that searches for the wanted notification in a specified set of machines and draws a graph, but this is not that useful, since the error has been detected at that point and a certain amount of notifications have been dispatched. Secondly, to create a sensor, it is necessary to write a script, test it, and maintain it during its lifetime in order for the notifications to work properly. This can be an error-prone and time-consuming task. Having tools that automatically would give the same end results as these two steps, would be very beneficial.

[13] http://riemann.io/
[14] http://logstash.net/

Still, introducing new tools have downsides such as the effort necessary to install, configure, integrate, and maintain them. This can, however, be a more straightforward solution with long-time stability if the tools are chosen with care and properly configured, compared to software development from scratch. But, a disadvantage is that the tools might contain unnecessary or duplicate features compared to the existing framework, whereas a highly customizable script may be able to do what is needed and nothing more. These were some points that were considered before the final decisions considering the chosen solution.

The final solution was to create a light Python-based framework to solve the task. The reason for this was that a small, well-developed and suitably customized framework brings less of a maintenance overhead, than creating a small cluster of VMs on which the new monitoring tools would be installed. The VMs would not only need maintenance and quota, the software itself would also need to be upgraded and configured with regular intervals. The framework would tap into the stream of notifications as they are collected from all machines to one queue that sends them to storage. As this stream contains every notification, it would be fairly easy to collect the relevant notifications for an aggregated view, and notify the service responsible by a chosen mean. The benefits of this solution are that it is easily integrable to the existing framework, customizable in a desired way, and the development process can be steered by requirement. A more in-depth description of the architecture and implementation of this framework is found in Section 4.

Before the development work started, two main use cases were identified: using such a framework for monitoring a small service (e.g., GIT) where one notification may mean that the whole service is down, and monitoring a large service (e.g., the batch system[15]) where a 'high load' notification may mean that the service is operating normally, but for certain machines that the service is degraded or inefficient. After discussion, the latter case was chosen to be investigated initially. The reason for this was that a large environment produces more noise (uninteresting notifications) and thus proves to be more suitable for testing purposes. After investigating the fundamental needs that a service responsible may experience while maintaining a service, the most important use cases and needs that the framework would help solve were identified as described in the list below:

- To know the service status:
  - This can be done by configuring limits for when a service is degraded.
  - It is better to be able to warn users about service degradation, rather than receiving complaints and then discovering the degradation.
- To suppress unimportant alarms without turning them off:
  - By configuring a threshold for sending, e.g., 5 of the same alarms/hour.
- To inform users/service responsibles of service degradation:

[15] http://information-technology.web.cern.ch/services/batch

  - It could be done by opening or automatically responding to a ticket.

- To automatically close tickets when the alarm is resolved.

- To try to pinpoint or give a hint about the reason for failure:

  - For example by defining a log file that a `grep` command would be executed on, on a machine.

One concrete example are notifications which do not open tickets, since they are insignificant by themselves, but a large amount of these exceptions in several machines require an action. This can for example happen if the notification metric measures a version of a certain software and all machines are not updated at the same point or the update is not completed. In that case an aggregated view and an e-mail which is sent to the service responsible can be useful. However, these notifications are fairly rare in the monitoring framework.

A short analysis of the found use cases and the objectives was performed. Among the conclusions are, that the framework should be configurable to hold a value of what the definition of service availability is (e.g., percentage of machines required to barely run a service). For a service responsible, it is important to be alerted about errors before the user complains. One of the main features, as mentioned before, is to filter away redundant information being careful that any important alert is not missed. It would also be beneficial to be able to automatically update several tickets with relevant information, or mark them as resolved, letting users or watchers know that the error is recognized without spending human time on such communication. Finally, the service responsible could define a command or action as an attempt to automatically correct the error. This would be beneficial in cases where a known error can appear and a known solution can be applied. Ideally, this framework would fit in a niche of the existing framework where it could reduce the time that is spent on repetitive tasks and on analysing information that contains an abundance of noise.

3 The CERN IT monitoring infrastructure

This Section describes the architecture and components of the CERN IT monitoring infrastructure. At CERN, the main software infrastructure for monitoring is provided, maintained and developed by the monitoring group (IT-MON). Information is collected, processed, stored and published by a framework consisting of different softwares which are responsible for their own distinct part of the design [3]. As a comparison, the old architecture of the monitoring infrastructure is illustrated in Fig. 7, and the current architecture is illustrated in Fig. 8.

Two important parts of the monitoring infrastructure are the LEMON (hereafter Lemon) monitoring tool and the General Notification Infrastructure (GNI) (described in Section 3.5) for handling notifications, which can be seen as a part of the monitoring tool. Lemon was created and deployed at CERN at the start of the new millenium [4]. The logo, whose design is closely related to the tool name, is depicted in Fig. 9. As a clarification, the term "Lemon monitoring framework", when used in this thesis, comprises both software that includes the actual Lemon monitoring software (e.g., `lemon-agent`), and other software that has been added later such as the Kibana dashboard which by itself is not connected to the original Lemon monitoring framework in any way. This reflects how the term "Lemon monitoring framework" is commonly used, and is distinct from the term "monitoring framework", which in this thesis describes the new framework that the author developed.

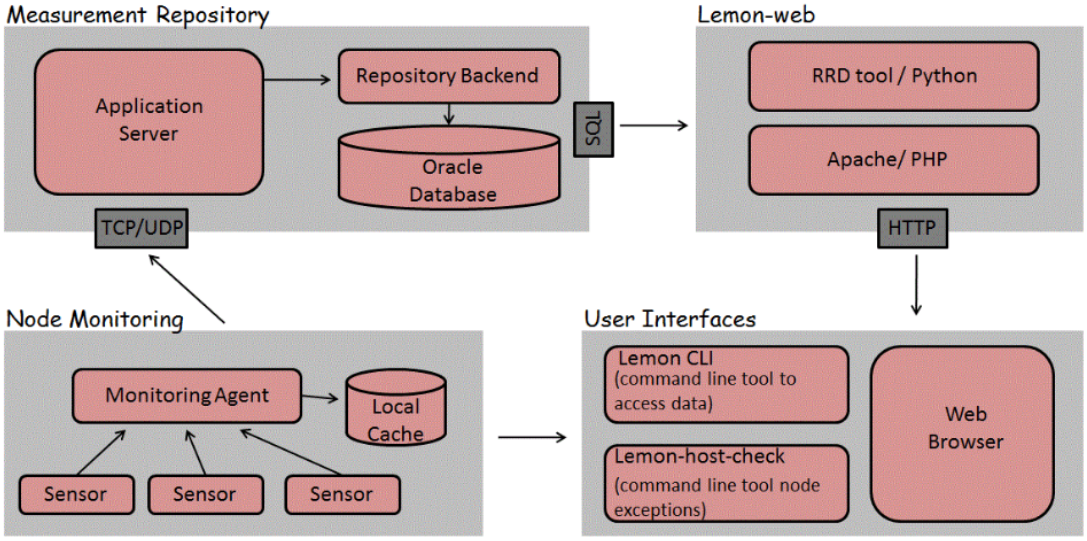

Figure 7: The old Lemon monitoring system architecture [58].

Originally, the Lemon monitoring framework had custom-built components for handling visualization and the storage was a database, this can be seen in Fig. 7 as a comparison to the current architecture. It is important to recognize the components in the old version, since some of them continue to be used, with CERN having around 15 years of experience using that component. Most notably the "Node Monitoring" component containing the monitoring agent remains in use. The visualisation (Lemon-Web) and storage (Measurement Repository) have been upgraded to other improved

solutions. The user interfaces Lemon CLI and Lemon-host-check are still available and commonly used when testing a newly created sensor. Nowadays there is a Python API (and Perl API) for developing sensors, whereas before the official APIs were C/C++ and Perl [58]. It is worth noting that the concept of notifications also have been part of the old Lemon monitoring tool, even if they are not depicted, but the implementation has changed over the years.

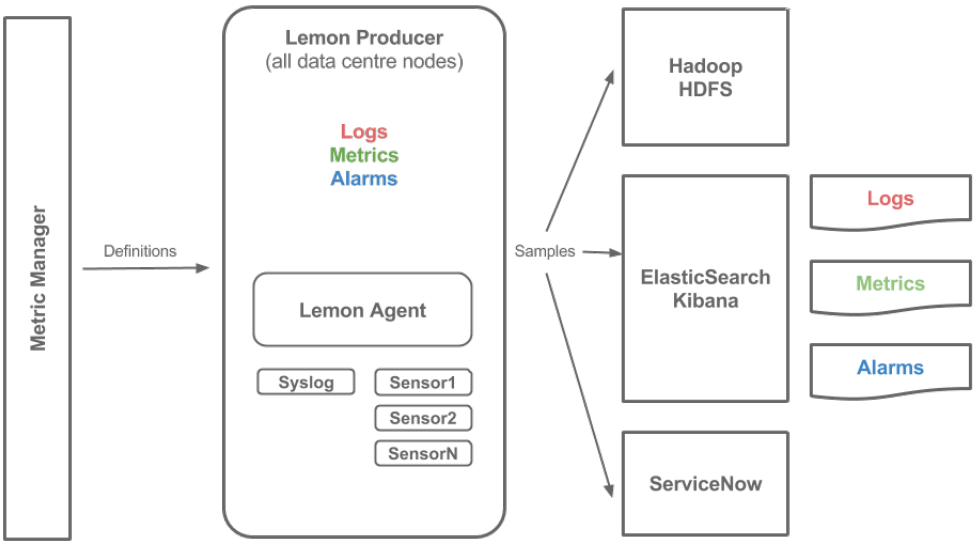

Figure 8: The existing monitoring infrastructure architecture [29].

Since the first release, the framework has been extensively updated, changed and expanded. The latest major architectural change started in the early 2010s [3], which is the currently used architecture. The reason for changing the architecture, was not only to update the Lemon monitoring framework, but also an attempt to exchange independent (but similar) monitoring softwares that were used for one single tool [3]. In that case it would be easier to share monitoring data, perform comparisons of different systems, and as a whole take a step towards unifying the infrastructure in order to gain a better overview of resources. Below follows a list of components and softwares that are currently used in the Lemon monitoring framework ([3], [58]) which will be described in turn in this section.

- Metric Manager
  - A web interface for defining metrics.
- Lemon Agent
  - A software that runs on the monitored machine gathering readings.
- Lemon Forwarder

  - A software that runs on the monitored machine, processing readings from the Lemon Agent and sending messages to the Messaging System.

- Flume
  - Transports the samples from the Lemon Agent to their destinations.
- Hadoop HDFS
  - The storage for the metric and notification data.
- Elasticsearch
  - A full-text search engine for the metrics and notifications.
- Kibana
  - A dashboard for visualizing the metrics and notifications.
- Service Now (SNOW)
  - A service management/ticketing system.

The Lemon monitoring framework is based on sensors which measure one or more metric. More specifically, a sensor consists of metrics (which implement metric classes). This is illustrated in Fig. 10. The monitoring is done by the `lemon-agent` software which runs in every machine that is monitored [58]. It communicates with the sensors using a protocol that was specifically created for this purpose[16]. An entity which produces monitoring data (i.e., a machine) is called a Lemon Producer. The Lemon Forwarder is a component that processes messages with metrics from the Lemon Agent, and decides whether to create a notification or not.

Figure 9: The characteristic logo of the Lemon monitoring system [17].

A sensor is a Perl or Python script that implements one or more metric classes which calculate or read some value that one wishes to measure [58]. Each sensor is a separate process. The measurement is recorded by the monitoring framework using functions imported from the Lemon programming library (module) that are called upon inside the metric script. The measurement frequency is decided by the creator of the metric. Some other parameters that can be set when creating the metric are seen in the Metric manager screenshot in Fig. 11. Each metric is associated with a unique numeric metric ID. There are both standard and custom sensors. The standard sensors are managed by the monitoring team, whereas custom sensors are developed by users of the monitoring infrastructure.

[16] http://lemon.web.cern.ch/lemon/doc/sensorAPI.pdf

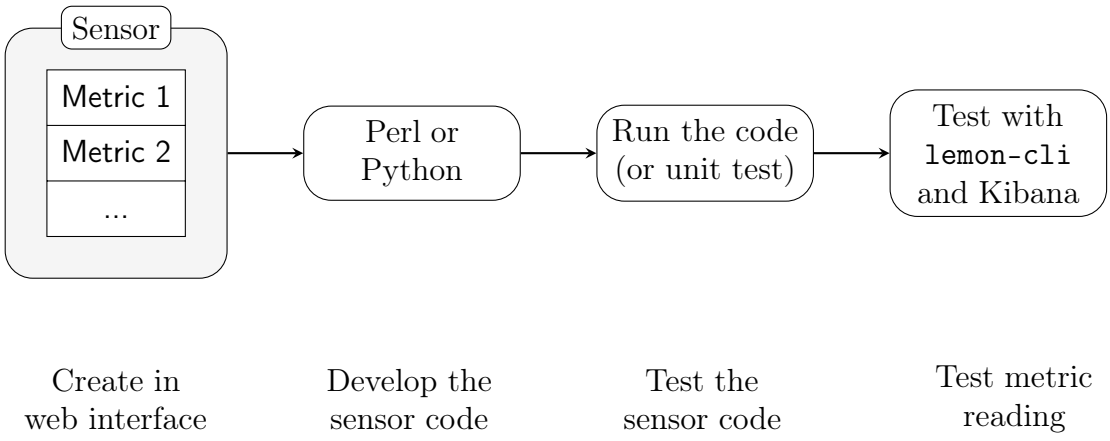

Figure 10: Workflow for creating a metric.

A notification is special kind of metric. It contains a state, e.g., `open`, `active`, or `close` which reflect the lifetime of the notification. It monitors values of other metrics. An arbitrary amount of metric IDs to monitor and thresholds for their values can be defined. When the value that is defined as the threshold is exceeded as many times as the metric creator has defined, a notification is sent. Optionally a ticket can be set to open as the limit is exceeded. The workflow for creating a metric and notification that the user or service responsible is required to follow is shown in Fig. 10. Basic metrics exist for reuse, but customized metrics have to be created for many services. This can be compared to the similar workflow for the monitoring tools described in Section 2.1 where the users can use either a template or a pre-made plugin for monitoring, or alternatively create their own.

The Lemon monitoring framework collects, transports and filters data through various software before it reaches the final destination - graphs and numeric values on a dashboard. Metrics and notifications are transported and handled in different manners. Sections 3.1 to 3.4 discuss the transport, storage, processing and publishing of metric data, whereas the last section, Section 3.5, presents how notifications are handled as a comparison to metrics, since only some steps in the procedures are shared between metrics and notifications.

3.1 Data sources and transport

Apache Flume is used as the transport medium for the metric values [3]. It is a Java software for distributed gathering and aggregation for extensive volumes of data that is sent in streams (e.g., log data). It is based on the concepts "source", "channel" and "sink" which are contained in an "agent" [39]. The received data is called "events". A source is a consumer component to which data is sent. One example is the `exec` source that executes an arbitrary command and collects the output data, or a server port whose traffic is collected. In the case of Lemon metrics, a Directory Queue and rsyslog are the two used sources.

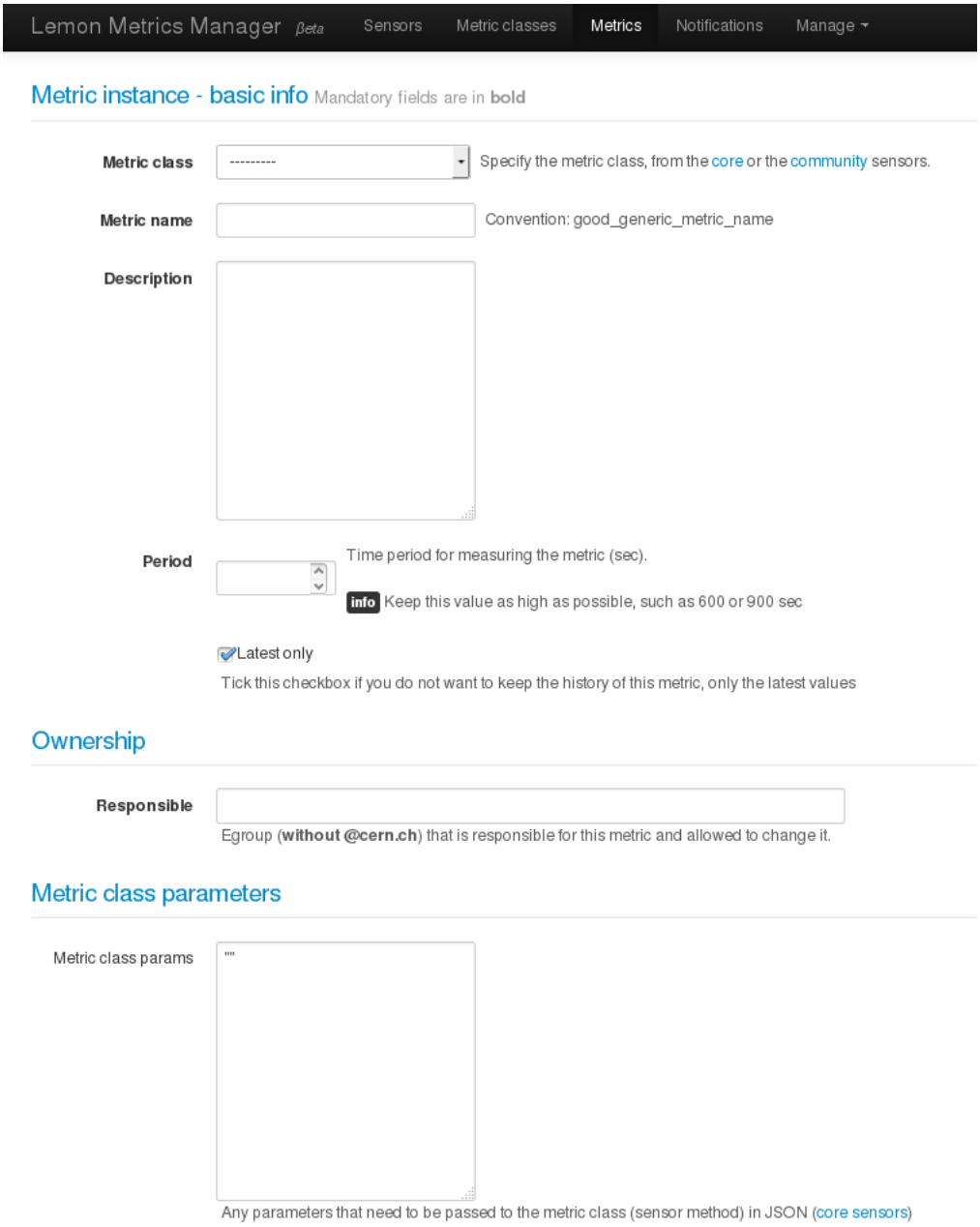

Figure 11: Part of the Monitoring group's Metric manager web interface where the user can define a metric [75].

The Directory Queue[17] is a locally stored queue to which the lemon-producer (metrics or notification producer) writes messages[18]. The second source, rsyslog[19], is a software implementation of the syslog protocol (RFC 5424) for transmitting messages containing log information. Syslog has existed since the 1980s [52], but has been used as a de facto standard until the first standardization in 2001 (RFC 3164). Flume is thus configured to listen for and subsequently capture messages which are either appearing in the Directory Queue or sent out by rsyslog. The channel is an intermediate temporary storage for the events (messages). The sink reads the events from the channel and forwards them to an endpoint.

In this case an Apache Avro sink is used to forward the events to a central gateway instance for final storage. The information is sent to a cluster of ten nodes that forwards the information to a third layer which is the storage. Apache Avro[20] is a software for data serialization based on schemas that are applied to read and written data. This minimises the need for processing and storage, as the schemas can be used instead of appending tags or other information to the log data.

3.2 Data storage

The following section provides a description of where the information from the monitoring framework is to be stored. As can be seen in Fig. 8, every Lemon Producer generates measurement samples which are sent using Flume to be stored both in an ElasticSearch cluster and a Hadoop Distributed File System (HDFS) [3]. Hadoop is a distributed framework for processing enormous data sets[21]. As the abbreviation reveals, HDFS [13] is a distributed and highly scalable file system. It consists of a NameNode (master node) which controls the storage and several Datanodes (slave nodes). The NameNode is responsible for splitting data and organizing the distribution of the data pieces. Each piece has a configurable number of replicas stored, to ensure that data can be restored even if a part of the storage system fails. HDFS also has a rack awareness feature which for example can be used for storing replicas in different racks. The Lemon monitoring framework uses a 500 TB part of a HDFS which is part of a larger Hadoop cluster [3]. This is the data archive, the final data storage destination.

The data is also stored in an ElasticSearch cluster that is available throughout CERN. Here the metric data is stored only for 30 days. ElasticSearch is a searchable document storage with analytics features based on the Apache Lucene search library [49]. Given a document encoded in JSON (JavaScript Object Notation), ElasticSearch can index and store the detected keys, making the document searchable. An example is presented in Fig. 12, where the index is the first row, `logstash-2015.02.17`. The figure contains a notification that has been read from the message queue and processed by the authors installation of Logstash (a software similar to Flume for

[17] http://lemon.web.cern.ch/lemon/projects/dirq-consumer/
[18] http://lemon-monitoring.web.cern.ch/lemon_producer_configuration
[19] http://www.rsyslog.com/
[20] https://avro.apache.org/docs/current/
[21] https://hadoop.apache.org/

automatic processing, tagging, and sending of log files) and stored in ElasticSearch. As can be seen, ElasticSearch has received and identified the JSON structure, and created a timestamped index. This index can be used for example for searching by host group name, data aggregation by a specific notification state, time-based queries or application-specific queries and correlations, in case several applications store data with different indices.

The data from ElasticSearch can be queried over HTTP using a REST API. The Lemon monitoring system users have access to the metric and notification data by a graphical interface, Kibana, presented in Section 3.4. It is not possible to configure customised indices, since this would require administrator access to the installation. Users at CERN are currently recommended to launch an own minimal ElasticSearch cluster (1 master and 3 data nodes) for customised use of either monitoring data or log files which contain sensitive data (e.g., access logs), since the common dashboard data is visible inside the CERN network. Detailed instructions for this procedure exist[22]. In that way, it is possible for users to create their own solutions for analysing monitoring data.

As a comparison to the HDFS storage, the monitoring infrastructure's ElasticSearch cluster consists of ten nodes, of which one is the master node, one is the search node, and eight are the data nodes [3]. This installation uses indices which are separated by date, as seen in the index in Fig. 12.

```
"logstash-2015.02.17": {
  "mappings": {
          ...
        "notification": {
          "properties": {
            "hostgroup": { ... },
            "metric_id": { ... },
            "metric_name": { ... },
            "snow_assignment_level": { ... },
            "state": { ... }
          }
          ...
  }
```

Figure 12: An example of a JSON-encoded index in ElasticSearch.

The storage model which is implemented in the Lemon monitoring system architecture can be compared to the Lambda architecture [59] for handling Big Data, originating from Nathan Marz a few years back. In fact, a presentation in late April 2015 [6] revealed this architecture to be the inspiration for the design of the current monitoring infrastructure. The Lambda architecture proposes the use of a separate batch storage for archival and a separate storage for analytics purposes. The batch

[22] https://itmon.web.cern.ch/itmon/recipes/elasticsearch_and_kibana_a_basic_setup.html

storage is permanent, and can produce output that is based on the full, organized, amount of data that stems from a certain point in time. The analytics storage is meant to provide fast access to new data, whereas the results or conclusions may not be as accurate due to the fact that the data may only be a partial view of a situation. In this case the batch storage represents the HDFS cluster, and the analytics storage represents the ElasticSearch cluster.

3.3 Data processing

In this section the data processing in the existing monitoring system is briefly discussed. Currently, the data that is gathered by the monitoring system is mostly processed only by ElasticSearch queries. The query is written in the Apache Lucene query language, which, by the authors experience, can prove to be a usability issue for an average user. As a solution to this, many basic dashboards which present monitoring data have been made available by the monitoring team. The future plans of the monitoring team include integrating a data processing engine, such as Apache Storm[23], notably for metrics aggregation and use cases involving machine learning [3].

At the moment, one significant component in the GNI that will be presented in Section 3.5 processes one type of notifications (`no_contact`) which indicate that contact has been lost with a machine [33]. These notifications are very important, since they can indicate that a service or a part of the infrastructure is not functional. However, this is done by an individual software component that does not have the power of a data processing engine behind it. This thesis work, which involves creating a framework for processing the notifications acting as a filter, can be seen as a similar individual architectural component. Possibly, it can prove to be a useful initial case study for future work regarding the processing of monitoring data.

3.4 Data publishing

This section explains how the metric data is published in the Lemon monitoring framework. As can be observed from Fig. 8, the logs, metrics, and alarms (notifications) are stored in ElasticSearch and published in Kibana. Kibana is a frontend for ElasticSearch that provides dashboards and analytics functionality which is designed to work based on indexed ElasticSearch data. The data can be plotted over a timeline since it contains timestamps, and it is possible to construct a dashboard with plots simply by clicking, querying, and selecting the relevant data [3]. Writing the query is likely to be the most difficult step. The Kibana installation at CERN allows for permanent custom dashboards upon request, and users can freely create temporary dashboards. As will be discussed in the following subsection, notification data will not only be available on the dashboards, but can also trigger the creation of a ticket, which results in an e-mail being sent to the service responsible, or since April 2015, simply an e-mail alert.

[23] https://storm.apache.org/

Finally, it can be noted that during the course of this thesis work, changes were happening in the ElasticSearch and Kibana projects. Most notably the company behind these softwares changed the name from ElasticSearch to Elastic performed a rebrand in March 2015[24]. In addition to this, new versions of both ElasticSearch and Kibana were released. Different versions of Kibana require a specific ElasticSearch version to be installed on all nodes in the cluster. If even one machine with a different version joins the cluster, Kibana will show an error message indicating this and not work as expected[25]. This was the case for the author, as the new beta version of Kibana 4 was tested in a small ElasticSearch installation. In the new version, Kibana 4, it is possible to create a more versatile set of visualizations, and a step-by-step functionality for creating graphs was introduced (as opposed to writing a query and saving the resulting graph). In April 2015, the CERN monitoring infrastructure uses Kibana 3 for visualisation, and is evaluating Kibana 4.

One of the large changes between the versions were that Kibana had been distributed as a plugin to ElasticSearch in previous versions, whereas it now exists as a separate software. It can also be noted that ElasticSearch does not have any in-built access control, nor does Kibana. One alternative is the commercial Shield plugin, offered by Elastic [15]. Another alternative provided by the official Elastic blog [62] shortly before the Shield plugin was available, is to use a lightweight web server such as Nginx[26] as a proxy for ElasticSearch and Kibana, allowing the web server to handle the authentication and additional features such as load balancing and access logs. This was the approach the author used in a test evaluation of Kibana 4.

3.5 General Notification Infrastructure (GNI)

The GNI is a part of the monitoring infrastructure, and uses data from the Lemon Producers as is illustrated in Fig. 13. The SCOM (System Center Operations Manager) producers are Windows servers, which will not be discussed, since the focus lies on Lemon. The Lemon Producer metrics are processed by the Lemon Forwarder component [53], as previously mentioned. This component runs in every monitored machine. It performs an aggregation of the metrics that are measured inside the machine, in order to decide whether to raise a notification or not. If a notification is triggered, the forwarder sends it to the messaging system. The aggregation is done since a notification can be triggered by a combination of metrics.

It is relevant to note that the metric aggregation is performed inside the machine. As far as the author knows, there is no program available that automatically aggregates Lemon metric information or notifications, and create an alarm as an informed decision based on the state of multiple machines. One feature of the Lemon notifications is that an arbitrary command can be executed on a machine if a notification is raised, but again, there is no tool that would have a complete view, for example of the state

[24] https://www.elastic.co/about/press/elasticsearch-changes-name-to-elastic-to-reflect-wide-adoption-beyond-search

[25] https://github.com/elastic/kibana/issues/2513

[26] http://nginx.org/

of all hosts in a host group, and be able to act based on such information. This is a challenge that the new framework which will be presented in Section 4 aims to solve.

The messaging system in Fig. 13 consists of ActiveMQ messaging brokers [5]. Apache ActiveMQ is a asynchronous "message-oriented middleware" [72] for enterprise messaging. The function of enterprise messaging software is to deliver messages, using message brokers, between different applications, as a mean to provide interoperability, and to facilitate the integration of softwares in an enterprise environment comprising a diversity of softwares [72]. In the case of Lemon, the message broker connects the Lemon Producers (machines) with the GNI dashboard and the SNOW (Service Now) ticketing system.

The GNI dashboard is similar to the metrics dashboard that was discussed in the previous subsection. It is also a searchable Kibana 3 dashboard that shows the total top count of active notifications per host group, per host, and per notification type on the front page. Similarly, one can search for notifications and graph the amount of notifications for a certain time interval. SNOW [71] is a incident management system used at CERN. The incidents are managed as tickets which contain relevant information such as the caller name, a timestamp, the functional element (FE) that is concerned, the location, etc. As an important definition, an FE is a group of persons which are responsible for a certain service, and the FE also has its own e-mail list containing these persons. Most services are managed with the help of these tickets, ranging from requesting a parking permit to reporting a computer security incident. The ticketing system is also accesible by an API allowing the SNOW publisher to optionally open a ticket for reporting a failure, in case a notification is activated.

As a final part of the GNI, the `No contact` processor is presented. It is a component that analyses if the metric readouts are timing out, which can indicate that there is a problem with a machine [33]. If the machine is unresponsive for a distressing amount of time, the `No contact` processor will create a SNOW ticket assigned to the operator on duty at the computer center. The ticket can be automatically closed by the processor if it observes that the machine has come online again after a moment. Eliminating false positives can be difficult, since some nodes, for example in the batch system, can be under a temporary high load, not having enough resources to send a metric readout but still not having a problem.

With this as a background, it was found that a way of aggregating notification information from a group of machines could be done by developing a framework which will be described in Section 4. During discussions with the involved teams, it was found that there already had been ideas to develop a framework similar to this. It was also deemed to be easier to integrate a custom-built framework with the existing monitoring system, than choosing, installing, and maintaining a software that would duplicate many of the features that already exist in the monitoring system in addition to aggregating notifications. The placement of the framework in relation to the existing infrastructure can be seen in Fig. 13. It would be a component similar to the `No contact` processor, since both components read notifications from the MQ, processes them, and re-emits notifications into the messaging system which end up on the dashboard or in the SNOW ticketing system.

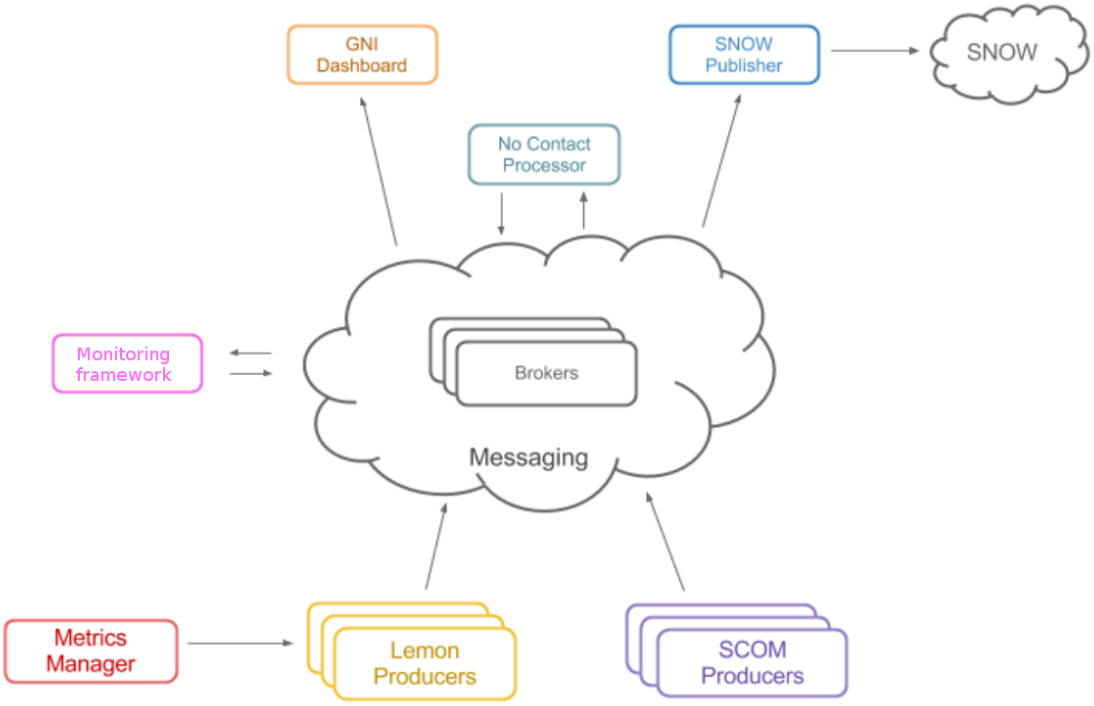

Figure 13: The GNI architecture [33] with the "Monitoring framework" component added on the left.

4 Expanding the existing monitoring infrastructure

This section illustrates how the information collected by the Lemon monitoring framework has been taken advantage of to create a framework for aggregating notification data from several hosts. The first subsection shows the architecture of the new framework, the second subsection presents the logic behind the configuration files that the framework uses, the third and last subsection described how data is stored and processed in the framework. Each subsection presents new technologies in relation to how they have been used, often with a comparison as a motivation as to why certain choises were made.

The current monitoring infrastructure was presented in Section 3, and it was found that there is a shortcoming regarding the aggregation of data on a larger scale (e.g., per host group) as opposed to per machine before a notification is created. As far as the author investigated, there were no tools available at CERN for this task that would do any kind of basic analysis in real time and create an alert that would be triggered upon reception of notifications from several machines. This was the reason to investigate possible solutions, which resulted in the creation of this framework. As had been observed in Section 2.4, this framework was designed and implemented to automatically handle unimportant notifications and raise awareness of large-scale failures. The objective was to make it easy to configure which notifications and host groups are important, and to whom the notifications are to be sent.

4.1 Architecture of the new framework

The framework itself is divided in roughly ten separate scripts containing classes. The main parts can be seen in Fig. 14 which shows the main scripts and folders as well as explanations for each component. It is developed in and for Python (version 2.6.6) as this is the version that is available in the installations of Scientific Linux Cern 6 (SLC6) which at the moment of writing is running on the majority of the machines this thesis work considers. The logic for consuming notification messages from the message queue uses the `twisted_consumer.py` proof of concept Consumer class[27], created by a colleague, with some customization done for the sake of the framework.

As seen in Fig. 14 the framework consists of five core parts which are marked in red (`main, actions, database, config` and `logger`). The actions that are launched from the `actions` script are defined in their own scripts, as an example `openstack` and `send_mail` are shown (marked in light red). The configuration files are stored in their own folder, `yamlconf`, marked in blue. The framework also contains unit tests (marked in green).

The basic unit that is handled in the framework is a `message`, which is parsed from the body and headers of the message that is received from the MQ. In general there are a total of between 400–600 active notification messages of different importance (also from different host groups) arriving per minute. This was measured simply by counting the messages in a function as part of the framework and logging this

[27] https://github.com/gmccance/twisted-gni/

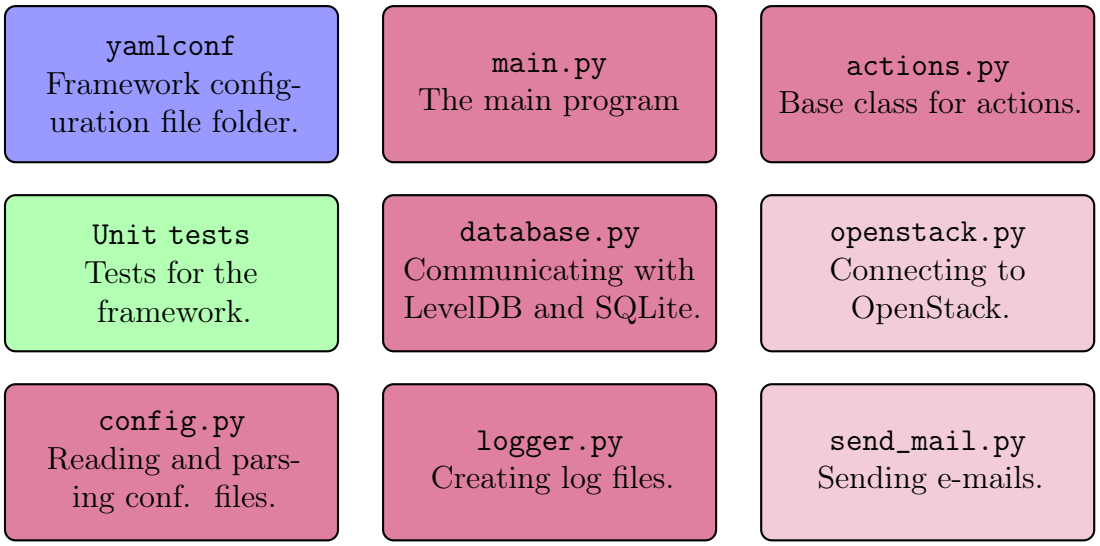

Figure 14: The main parts of the framework architecture.

information as the framework was running. Of these, tens of alarms (per minute) belong to the host groups that are under investigation in this thesis work. This, again, is based on information extracted from the log files.

The program flow of the framework is presented in Fig. 15, which shows the different stages of how a MQ message is handled in the framework. In short, important fields from the MQ message is parsed to form a `message`, which is stored. The policies are inspected, which may or may not trigger a notification. The `message` contains the following fields: `header_timestamp, body_timestamp, value, metric_id, entity, description, environment, hostgroup` and `fe_name`. The field `entity` is the hostname, and the field `environment` describes whether the machine is part of the production or development enviroment. These `message` units are used for storing the relevant information into a local database and for passing the relevant information from a MQ message to an action. As can be seen, the `messages` contain the host group names and can thus be mapped to a policy.

If there is a policy for an exact host group, it is examined to determine whether it should trigger or not. Currently the specifications of the framework contain two types of policies; number and fraction. The number policy simply counts the number of machines with notifications in the host group and compares it with the threshold. The fraction policy counts the number of machines with notifications and compares it with the total amount of machines in that host group. In this case the threshold acts as a percentage. This can be especially useful in environments with a large amount of machines where, for example, having 90 percent or more of the machines running is considered to be a sufficient service level.

The framework will also match the host group in the notification against policies for upper level host groups (parent host groups). The reason for this is that a parent host group counts notifications from its subhost groups. I.e., a notification from host groups such as `zone5/important/computing` or `zone5/important/idle` may also be counted in a policy for the host group `zone5/important`. This means that

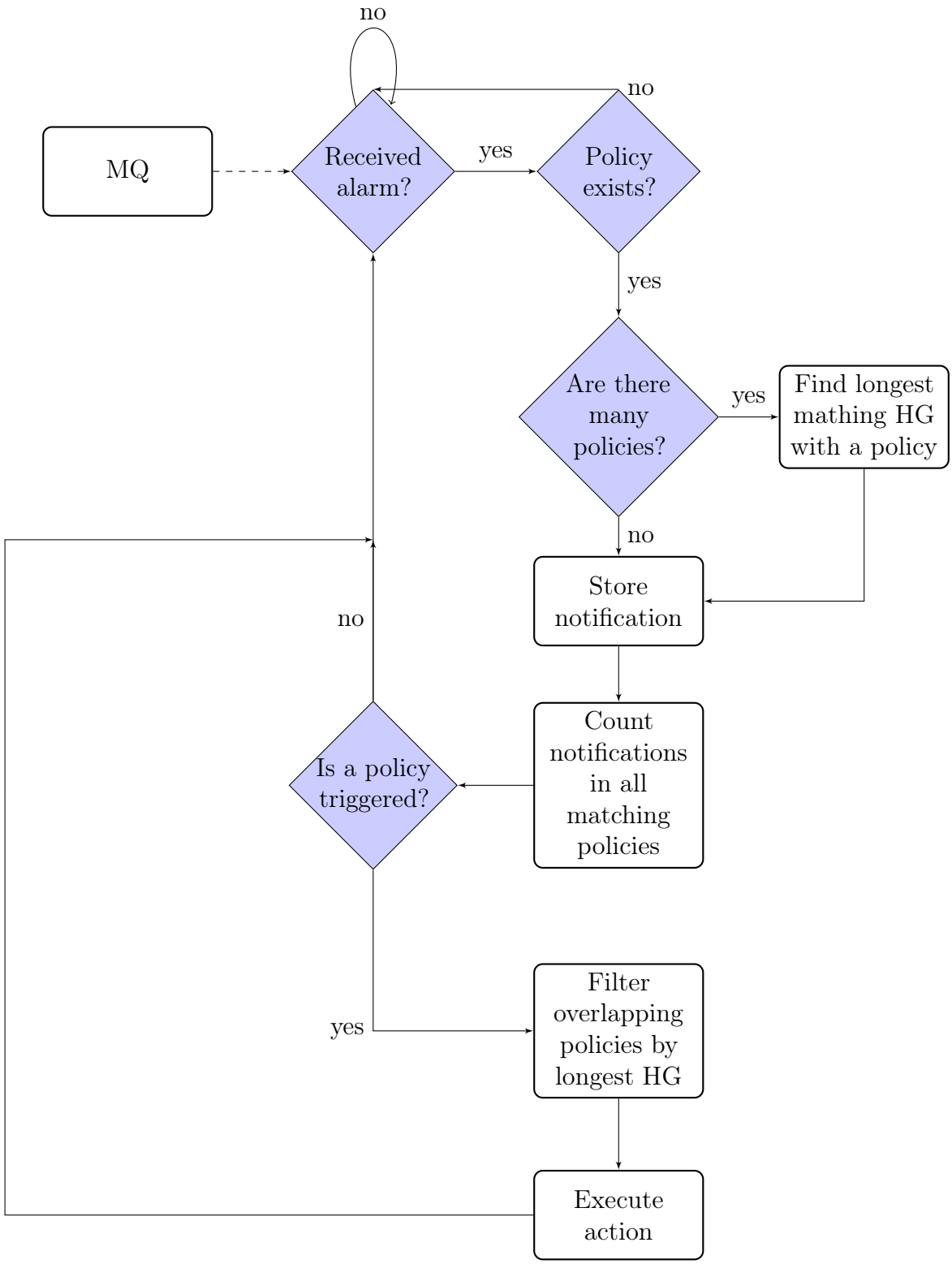

Figure 15: The program flow.

several policies can trigger an action upon reception of one message. At the moment of writing, the framework will execute all triggered actions, since that was a request from the users, but as seen in Fig. 15, one way to decide which policy would be preferred, would be to always choose the longest matching policy. However, this can lead to conflicts if, say, one policy wishes to restart a certain software and another policy wishes to restart the whole machine. This, and other related risks that need to be accounted for in the framework, are discussed in Section 6.4.

When the policies to trigger have been collected, the framework executes the configured action(s). These actions are described in more detail in Section 5. After this, the framework processes the next notification that has arrived and goes through the same process. This will go on until the framework is closed down, after which it is possible to restart the framework and load the latest state that the framework was in, or to start counting the aggregated notifications from zero. As part of the specifications imposed on the framework, the notifications are only removed from the error dictionary when the framework receives a message saying that the notification is closed. The count is, e.g., not zeroed when a threshold is reached.

4.2 Configuration files

The configuration files for the framework are written in YAML [9]. An example of a configuration file can be seen in Section 5, Fig. 18, where the configuration is used to illustrate the how the actions work. The layout of the configuration file and the choice to use YAML was made, since the majority of VMs at the department of IT have their configuration managed by Puppet as explained in Section 2.3, and it was deemed to be a good choice from a usability point of view, to have both types of configuration files follow the same logic and format. Thus, the configuration files are stored in folders named by the parent host group, and the configuration file name is the subhost group name. This idea was previously illustrated in Fig. 6 which describes the relation between host groups and subhost groups.

Some issues that might arise from this choice are, that it can be complicated to manually create these configuration files, the correct folder, and to ensure that the configuration is correct, parsable and so on. To solve such issues, it is part of the future work to create a short script that would check the configuration file for errors or syntax mistakes, or even a tool for automatically creating these configuration files if there is a need for it.

The configuration file must contain at least the following information: the host group name on the first row, the type of policy which is used and one or more triggers (actions). It can also be specified which types of exceptions to include or exclude when counting. Regarding the host group name, it is important to not add the parent host group names, i.e., the full path on the first row. This is an example of a misconfiguration that is currently difficult to spot and which goes undetected in the framework until the service responsible starts wondering why no notifications are collected from this certain host group. However, some of the large benefits with the configuration files are that they are written in a format that is familiar for most of the personnel, and easily can be changed and added as necessary.

4.3 Storage and processing of data

As messages arrive to the framework, they are parsed and put in a Python dictionary of the format that can be seen in Fig. 16. The figure shows the structure of the error dictionary that contains the received notifications and the policy dictionary that contains information parsed from the configuration files. The fields that were chosen as relevant to store are parsed into a `message`, from the message that is received from the message queue. This parsed `message` is stored in a local SQLite[28] database as a table row with columns and values as described in Fig. 17.

```
1   {hostgroup{error:set([machine])}}
```

The error dictionary.

```
1   {hostgroup{
2       exceptions:{'include':set([include]),
3                   'exclude':set([exclude])},
4       policy:'number' OR 'fraction'
5       triggers:{
6                   email:{number:n, threshold:t}
7                   snow: {fe: f, threshold: t}
8                   cmd: {do: script, threshold: t}
9                 }
10  }}
```

The policy dictionary.

Figure 16: The error and policy dictionary structures.

```
{'body_timestamp': 1425993887,
 'description': u'exception.ipmi_wrong',
 'entity': u'lxbsu0625',
 'environment': u'production',
 'fe_name': u'LXBATCH',
 'hostgroup': u'batchsystem/batch/gridworker/aishare/share',
 'header_timestamp': u'1425993894',
 'metric_id': 30540
 'value': 5}
```

Figure 17: In this dictionary which represents data parsed from a MQ message, the keys represent database table column names, in which the values are stored.

The error dictionary is stored using LevelDB[29]. LevelDB is a storage engine for

[28] https://sqlite.org/
[29] https://code.google.com/p/py-leveldb/

storing ordered key-value pairs in string format [27]. The notification dictionary had to be serialized to string format in order to be stored. First, an attempt was made to store the dictionary in JSON (JavaScript Object Notation) format, which failed. The reason for this is that the dictionary contains a `Set` of machines which are sending notifications, and the conversion for a `Set` to JSON string is not available in the Python JSON library [38], unless additional encoding and decoding code is added. The second attempt was to convert the dictionary to YAML which offers native support for sets [8]. This conversion was successful, and thus, the dictionary is stored with the current time (in seconds) as a key, and a YAML encoded string representing the dictionary as a value.

LevelDB supports fetching of records from a defined range with optional upper or lower bounds, which, e.g., can be time ranges. This makes it possible to store the dictionary as the framework is shut down, and resume to the old state immediatly by reloading the error dictionary from the LevelDB storage. Such a feature is useful if the framework crashes for some reason (e.g., unexpected power outage) or if it is closed just before an action would have been triggered. It is also possible to replay the actions of the framework since the actions (including a timestamp) are stored in log files, and the notifications are stored in LevelDB. Taking it one step further, it would be possible to create simulations of how different configurations would react with different message flows based on the existing components in the framework.

5 Framework actions

This section presents the actions that the framework is capable of performing after a policy has been triggered as previously described. The actions are configurable in the host group configuration file, and in the case that two overlapping policies are triggered, both actions are executed. This could cause issues, but it was a conscious choice to do so for now, based on discussions with the future users.

When an action is triggered, the framework calls a function with the following information:

- Exception name.
- Policy (a dictionary containing the whole policy).
- Message (a dictionary containing the parsed MQ message).

The function executes as a separate process using the `multiprocessing module`. The process terminates as the action returns from being executed and the program ends. If the process execution hangs for some reason, there exists a possibility, that the computer where the framework is running will start to contain zombie actions that are not terminated. In this case it would be beneficial to set a time limit for how long an action is allowed to run, and run a cleanup process with a certain interval that would kill processes which have been left hanging for longer than the defined time limit allows.

There also needs to be a limit on the amount of actions the framework can take. The following two limits were considered: a limit for actions the framework can launch during a certain time interval or a limit for the actions per host group. It was decided to start with inserting a limit per host group, deciding how long the framework has to wait before being able to launch a new action. This is the "cooldown time" for an action. However, if the child host group has recently executed an action, it is still possible for the parent host group to execute an action. This logic follows from the decision that was previously discussed, to allow all actions in policies which are triggered by a notification to be executed simultaneously on the same host group.

5.1 Shell commands and scripts

The framework supports the execution of shell commands and scripts. The syntax for this is a trigger name starting with `cmd_`. An example can be seen in the `cmd_ping` trigger in the configuration file shown in Fig. 18, where the command in the `do` field pings the host `monitoring.cern.ch` twice. Such a command can for example be useful when the framework needs to determine whether the received exceptions are a result of another service malfunction. As can be seen in the same `cmd_ping` trigger, it is also possible to use piped commands.

It is necessary to provide the absolute (full) path to the command, otherwise the command will not be found. Since any arbirtary command will be accepted, it is also possible to launch a script that will execute as the threshold is reached. However,

```
carolinatest:
  policy: number
  triggers:
      email:
        threshold: 5
        recipient: carolina.lindqvist@cern.ch
      snow:
        threshold: 1000
        fe: fake_test_functional_element

      cmd_ping: &PING
        threshold: 33
        do: /bin/ping -c 2 monitoring.cern.ch | \
            grep rtt | cut -d / -f5

      cmd_trace: &TRACE
        threshold: 4
        do: /bin/traceroute monitoring.cern.ch
        recipient: carolina.lindqvist@cern.ch

      analysis:
        threshold: 3
        tasks:
          cmd_ping: *PING
          cmd_trace: *TRACE
        recipient: carolina.lindqvist@cern.ch

  exceptions:
    include:
      - exception.carolina_incl
      - exception.carolina_ping
      - exception.carolina_study
    exclude:
      - exception.carolina_excl
```

Figure 18: A sample configuration.

launching scripts introduces problems such as ensuring that the script is in the correct place, that the script is executable, and most importantly, that the contents of the script corresponds to what the person who writes the configuration imagines the script to contain. This is a possible risk that needs to be kept in mind, even though it is unlikely to be a severe problem, and this practice is already known, for example from using cronjobs[30]. Practical problems such as having to maintain both the configuration files and the script files that are associated with the configurations may arise. Long commands and analyses which only fit into scripts should therefore be avoided, unless they are necessary.

Due to the nature of YAML, it was possible to introduce the `analysis` trigger. In YAML it is possible to specify a reference to a previously defined node and use the reference to indicate the contents of the referenced node. This is described in the YAML specification Sections 6.9.2 and 7.1 [9]. An example of this referencing can be seen in Fig. 18 where the rows 11 and 16 contain an "anchor", which is the name of the reference preceded by an ampersand (&) sign and the "alias" on rows 24-25 where the reference is used, in form of the reference name preceded by an asterisk (*). As the configuration is loaded, the alias is expanded to contain the node content, for example row 24 would contain rows 12-14 (not 11).

The analysis trigger takes advantage of the referencing technique by enabling the combination of several actions to be run. This could, for example, be useful when the result of single commands are sent to one user, but the result of actions with higher thresholds are sent to another user. It is also an alternative to have 2-3 commands combined to an `analysis` trigger instead of writing and maintaining a script that would execute several commands and return a parsed output of them.

After any script or command has been executed, the result is returned to the `action` script which is responsible for the actions. If there is a recipient e-mail adress that has been configured for the trigger, the command and the result will be sent to the recipient along with a timestamp which indicates the time when the e-mail was dispatched. The subject line contains the timestamp, the host group name, and a tag surrounded by brackets (`[ismon]`) for facilitating e-mail filtering. The tag "ismon" is shorthand for Infrastructure Services MONitoring. In the case that a script or command hangs, this will not directly influence the framework, since the `action` script runs as a separate Python process.

5.2 Creating and modifying service tickets

The commercial SNOW (Service NOW) incident management and ticketing system[31] is the main system that is used at CERN for handling incidents and tickets. This system was described in Section 3.5 as it is one endpoint for the notification infrastructure. One of the new features that the framework is planned to be used for is, automatic opening, editing and closing of SNOW tickets. This feature would save time, e.g., if there are several tickets that are generated by the same machine in

[30] http://man7.org/linux/man-pages/man8/cron.8.html

[31] http://www.servicenow.com/products/it-service-automation-applications/incident-management.html

which case these tickets could be aggregated to a single ticket containing all the notifications that the machine (or machines) were showing.

During the development of the framework it was important to be able to have a test environment where multiple incidents could be created and viewed. It was not feasible to use a real functional element, e.g., inside the section, since any test tickets would be sent to the several persons that belong to the functional element's mailing list. Instead, the possibility to use a test instance that would be provided by the SNOW team was investigated. This posed a small challenge, since the access to the ticket creation API had to be justified and discussed, which had to be done not only with the SNOW team, but also with the monitoring team, since this work is based on their infrastructure. As a reminder, as was seen in Fig. 13, the infrastructure contains a SNOW publisher component which reads notifications and opens tickets if it is necessary.

One suggestion from the monitoring team was, to reuse the SNOW publisher component and simply create a new notification (based on the aggregated information) in the framework, and submit it to the MQ broker. Then the broker would submit the ticket to the GNI dashboard and create a ticket using the SNOW publisher if it was necessary. This would also solve the testing and access issues, since it could be confirmed that the ticketing works by examining the GNI dashboard and seeing the framework's notification. The author was able to send a test notification that is created by the framework, written to the queue, and that appears in the Kibana dashboard. The notification can be seen in Fig. 19. To open a ticket, using the GNI, it is only required to insert a flag in the notification message (the `header.m_snow` field set to 1), and the GNI will open a ticket.

Compared to the existing infrastructure, it is an improvement to be able to programmatically create new notifications that consist of aggregated data instead of information from single hosts. What remains to be studied in the future, is to be able to edit and close tickets. Integrating the framework with SNOW will need minor changes to be done also on the SNOW side, as well as agreements between all involved teams on the specifications for the interface, i.e., the meaning of different fields in the notification, in order for SNOW to be able to parse the notification properly. The major change is that notifications will not belong to single entities, but to functional elements (services). Discussing these changes, and implementing them is a time-consuming task that will be continued after the thesis has been presented.

5.3 OpenStack actions

The infrastructure at CERN was presented in Section 2.3. As was mentioned, the foundation for the CERN cloud is based on OpenStack, which at CERN is currently running the Juno (10th) version [14]. The latest (11th) version, Kilo, was released recently, at the end of April 2015 [37]. The services that the focus lies on are hosted on OpenStack VMs, and the framework can interface with the OpenStack API, which is the reason for briefly presenting OpenStack.

OpenStack is essentially a collection of several tools with different purposes, which each represents a component or service in a computing cloud [35]. The components

can be combined fairly freely depending on the architecture and purpose of the cloud, but there exists a set of recommended core services [35] that should be included in a general cloud architecture. The components are also named in a manner that does not always reflect their purpose, for example, some of the core services are as follows: `nova` (computing), `neutron` (networking), `keystone` (identity and authentication) and `horizon` (a dashboard). The services are accessible using the dashboard, if that component is installed, or by a command-line tool that bears the same name as the component one wishes to use, e.g., `nova`. The OpenStack components are also accessible by a Python API, which again, is different for each tool and bears the name of the tool. As an example, the package `python-novaclient` can be used to implement a custom version of the `nova` tool or, e.g., to programmatically list the VMs in a hypervisor or start migrating a VM, which is also doable by using the `nova` command in the CLI.

It is worth noting that the APIs have been dubbed "The best-kept secret of OpenStack"[32] and the OpenStack development team also notes that the Python API "has not yet been documented"[33]. These are practical issues a developer faces as they wish to programmatically access OpenStack components. Since the CLI tools themselves are fairly well documented, it is also possible to use a Python library such as `subprocess` that is able to run arbitrary CLI commands and instead run a `nova` command as a subprocess. The author has subjectively observed that this is a common shortcut when there is little time for the developer to familiarize themselves with the OpenStack Python libraries and discover the sometimes undocumented features, or when it is simply more practical to use subprocesses instead of creating a dependency on the native library.

In the case of the framework it seemed clearer from an architectural point of view to use the native library for the actions that involve OpenStack, most notably, for the action to restart a VM. The author also had some previous experience using the API which proved to be an advantage, and, additionally, the IT-PES group had already developed tools and a customized client whose functionality could be inherited in this project. One of the main advantages from the customized client was the Kerberos authentication method[34] that is not (yet) included in the official OpenStack release.

Kerberos is an authentication service, or more exactly a protocol, whose most recent version is defined in RFC 4120. The authentication is based on a client, an authentication server, and a ticket-granting service [64]. The basic idea behind Kerberos is that a user authenticates once, and receives a special, temporary, and cryptographically protected ticket that can be used for authentication instead of the original credential. Some of the main benefits of this are, that the original credential is passed only once, which reduces the dangers of having it captured by a malicious user, the ticket can be used for transparent authentication in services which accept this kind of authentication, and the ticket has an expiration time after which it cannot be used even if it somehow is captured.

[32] http://www.ibm.com/developerworks/cloud/library/cl-openstack-pythonapis/

[33] https://github.com/openstack/python-novaclient

[34] http://openstack-in-production.blogspot.ch/2014/10/kerberos-and-single-sign-on-with.html

The Kerberos support is important for using OpenStack itself, since CERN uses Kerberos tickets as Single Sign-On (SSO) credentials for OpenStack, which in practice means that the credentials that are used for interacting with OpenStack are the same credentials which are used to access all of the main services at CERN (such as e-mail, administrative services, wiki pages, meeting management, etc.). However, when accessing OpenStack from the CLI or API, the credentials are usually stored in cleartext, either in environmental variables or inside a script (`openrc.sh`) that places the credentials inside the environmental variables. This is the easiest way to authenticate [36] which is built in, and as a consequence an often used solution. In many cases this is a good thing, e.g., for testing out OpenStack or in environments where the OpenStack credentials are not of significance. But, as explained, SSO credentials are very significant. Since the main OpenStack installation is accessible through a common cluster (which can also be logged in to from outside CERN given that the user is registered at CERN), it is possible for anyone with root access to this cluster, to malisciously search for and obtain SSO credentials from the environmental variables. This, and its implications are clearly explained in the user guide for the cloud service [48] to make users aware of this fact and the associated risks.

The root access is restricted to administrators, but a vulnerability or an inside attack could potentially leak a user's password. In any case, an administrator of a cloud service has no reason to be able to access a credential that can be used universally. Even worse, if, by human error, an unknowing user stores the password in cleartext and places the file in a publicly accessible location, e.g., a public repository or a public storage, it is possible even for unaffiliated persons to obtain a password. As a solution, it is safer to use the Kerberos authentication method by first requesting a Kerberos ticket, and subsequently use the ticket to authenticate to OpenStack. This way there is no need to handle the credentials in cleartext at any point, and it is one of the commonly used techniques to safely authenticate CERN users and staff to internal services. For these reasons, it was very beneficial to be able to use the OpenStack client (`novaclient`) that is patched with Kerberos support in the notification aggregation framework.

The OpenStack actions that in general can be imposed on a VM are, for example, terminating of the VM, creating a VM, rebooting a VM or migrating a VM. The action that was implemented in the framework is the rebooting action, which can be useful if a service is malfunctioning and it is known by the service responsible that restarting the machine (or machines) will remove the problem. It is fairly easy to extend the framework with further actions. It is only necessary to add a call for the corresponding function from the `novaclient` instance with a few lines of wrapper code that is common for all functions.

View: **Table** / JSON / Raw

Field	Action	Value
@timestamp		2015-06-03T08:46:10.000Z
_id		AU24mWVUJtTljc-v0zMl
_index		gni-2015-06
_type		notifications
body.metadata.description		This is a test notification from the aggregation framework.
body.metadata.entity		test_host
body.metadata.hostgroup		carolinatest
body.metadata.metric_id		123456
body.metadata.metric_name		exception.carolina_test
body.metadata.state		active
body.metadata.timestamp		1433321170
body.metadata.uuid		123456-1433321170
body.payload.entity		test_host
body.payload.metric_id		123456
body.payload.timestamp		1433321170
body.payload.values		[1, 2, 3]
header.JMSXUserID		pesalarm
header.destination		/topic/monitoring.notification.generic
header.expires		1433925970233
header.m_producer		aggregation_framework
header.m_submitter_environment		production
header.m_submitter_host		test_host
header.m_submitter_hostgroup		carolinatest
header.m_type		notification
header.m_version		2.0
header.message-id		ID:mb107.cern.ch-56236-1430135839101-1:10845437:-1:1:1
header.priority		4
header.subscription		1b14770-553e27a3-2e66-1e63-2
header.timestamp		1433321170233

Figure 19: A notification that is sent by the framework to the Kibana dashboard.

6 Experiments done with the framework

The framework has gone through three quite different types of evaluations which will be described in this section. The first type is unit testing, which is a fairly common part of software development. The second type consists of testing the framework in use on real data, gathering logs of the actions and the received messages, and analysing these logs to be able to compare the behavior of the framework compared to the existing infrastructure. The third type of evaluation is an interview that was conducted with representatives for a both a small and a large distributed service. The important difference between these two use cases, as discussed before, are that a large service can ignore certain notifications and a small service need to be more strict when detecting possible service degradation. In the final phases of the thesis work, the interview was also extended to a survey that reaches out across several groups at the CERN IT department which hold similar interest in a framework like this, such as the monitoring and the cloud infrastructure team. The reason for choosing interviewing as a method, was to learn how the service responsibles are using the monitoring tools in practice, and to learn which features are missing, that the framework could provide. It was also a mean to justify the measured results from the experimentation, by providing an account of concrete user experiences.

6.1 Unit tests

The framework is tested using unit tests that take advantage of the Python `unittest2` module. At the moment the unit tests cover the basic functionality of the main program: receiving and counting notifications as well as tests for triggering or not triggering the different actions. The unit tests are set up by calling methods, which do not correspond to a real scenario when the framework is listening to a message queue. It would be possible to simulate a real environment for the tests, either by replaying old traffic using the error dictionaries that are stored in LevelDB as was conjectured in Section 4.3, or by simulating a message queue that mock messages would be pushed in to. Either of these were deemed to not be necessary, since the architecture was deemed to be modular enough for creating tests that test single funcions instead of a giant test that would simulate a real environment.

The idea of constructing modular and easily testable code is supported by the fact that test-driven development (TDD) can be used as a tool for guiding the design and proper refactoring of the software and classes in the right direction [42]. In any case, it is often helpful to have stable tests, since the developer can know at which point part of the code breaks, and start debugging from there. Previous studies have found both positive and contradictory evidence when reviewing the use of TDD, such as improved code quality but decreased productivity, depending on the context [63]. It can be seen that the project described in [42] has close to a third more lines of test code than production code, which could indicate decreased productivity. However, the author explains that there is less of an effort needed when developing test code, which would indicate that it does not explicitly influence the productivity in this case.

During the development of the framework described in this thesis an approach that mixes TDD with traditional methods was used. A test was usually constructed before adding any large features, as a design model for the interfaces, but afterwards more tests were added in case it was found that the old test did not catch certain errors. The focus was not on rigorous testing, but instead on spending time developing the framework. Tests were used as a help during development, not as a tool that dominated the development. The code was also briefly profiled with the `cPython` module to gain a quick overview of whether the framework contains obvious bottlenecks at this stage of development. The results of a single two hour run can be found in Appendix C. Based on this profiling data, it looks like the SQLite database library takes most of the time to store messages in the database. At a later point, the feature of storing messages in SQLite may be removed if it is found to not be necessary. It is worth noting that the intention of doing the profiling was to detect if there are any serious problems in the framework, and to see how the load is divided among the different components of the framework. It was not an extensive investigation, since this project is limited in terms of time. Also, the framework itself is not showing bad performance, which makes profiling a low-priority task that was done simply out of curiosity.

6.2 Testing the framework in use

To investigate the impact of using the aggregation framework as opposed to when not using the framework, the framework was started and run during a six-day interval, to illustrate any variations in the traffic over several days. The test was planned, executed and evaluated based on principles which describe how to conduct experiments in computer science [83]. The main principles that were used are as follows: making the test situation as realistic as possible, finding an independent source for verification, and collecting the results in a reliable manner.

Two features that were looked for in the experiment can be highlighted as a hypothesis statement for the tests. The first is that the same machine may have several open notifications, which each will trigger a SNOW ticket, even if there is only a single cause. The hypothesis in this case is that there are several cases when a service responsible receives several tickets from the same machine regarding the same problem, and that the framework would help minimizing the amount of such cases. One example that was seen when running the framework, was a machine that was showing a notification for an unsupported configuration and soon after that a `no_contact` notification. Using the framework, the service responsible would have been notified only once instead of twice.

The second idea to be investigated was to consider different tunable parameters for the framework. The hypothesis was, that the cooldown time and the threshold limits, are the most tuneable parameters which exist in the framework, and that there should be visible differences in having long or short cooldown times. The intention was to investigate the limit for how long the cooldown time can be, before a service responsible starts losing information. Knowing how to tune the cooldown time is especially useful in cases where there are sudden bursts of notifications, which may

be closed in a short time. As a reminder, part of the specification for the framework is to always keep the amount of received open notifications in memory, and not, for example, zero the counter as an action is triggered. A notification is removed from the error dictionary only when the framework receives a message that the notification is closed. The test would also show if there are notifications which suddenly appear in large quantities, and therefore either need to be excluded if they are less important, or if the configuration needs a higher threshold.

Regarding the first experiment principle, being realistic, it was found during the experimentation phase, that there are certain CERN-developed notifications which do not give rise to SNOW tickets, that thus had to be excluded from the notifications that the framework was counting for the threshold. This was done in a few minutes, by reading the notification name, visiting the Metric Manager portal (shown in Fig. 11) to investigate whether the notification was set to open a ticket or not, and finally to enter the name of the notification in the proper host group configuration file. This cut down the counted amount of notifications from 113 to 1 on a certain host group which had many occurrences of a single notification, even when running the framework for 10 minutes each time. The notification in question triggered if there were less than 50 running instances of a certain batch client process, and the notification's actuator was set to try to start such a process. To ensure realistic measured values, all observed notifications which did not create a ticket were excluded in the above described manner before commencing with the following measurements. The sole exceptions were cases when these notifications had been specifically included by the service responsibles when specifying the configuration files for each host group. As mentioned before, one intention of the experiment was to single out notifications that can be excluded when tuning the framework.

The independent source that was to be used for cross-checking is the Kibana dashboard described in Section 3.4 that stores all notifications and allows a user to perform aggregation of notifications by manual queries. The collection of results was done by using the log files that the framework writes to. Each triggered action was noted, as well as the opening and closing time of the notifications. The log files were used to generate graphs that will be shown, explained, analysed and compared to results that can be discovered from the Kibana dashboard. During the extraction of this information from the log files, the author discovered and fixed minor flaws in the logging, e.g., that the log message for closing a file did not contain the host group (policy) that was concerned. The message only wrote the notification and machine name. However, the log files were still usable, since the machine name was logged when a notification was opened, and it thus was possible to easily deduce where a machine belonged simply by keeping note of the amount of notifications that belonged to a machine name.

The Python script that was developed to parse the log files into graphs use the `matplotlib` module's plotting functionality. For the sake of simplicity two host groups are compared. Host group 1 (HG1) contains rules for *including* four notification types. Host group 2 (HG2), a subhost group of HG1, contains rules for *exclusion* of eight notification types (thus including the notifications which are not excluded). The script reads the log file, and creates a point when a notification is

opened or closed in either of the host groups, or both, since we keep in mind that HG 2 is a subhost group of HG 1. The log files also contain the amount of notifications that have arrived to each host group. This information is stored each minute in the logs and is basically a count of the notifications per host group from the error dictionary (see, e.g., Fig. 16 as a reference for the error dictionary contents). Below follows five graphs showing different aspects of the notifications that HG 1 and its subhost group, HG2, received during a six-day interval. All graphs show the same time interval. The graph data is taken both from Kibana, which is deemed to be the reliable source, and the framework. One comparison is performed between Kibana and the framework to motivate that the framework functions correctly.

In Fig. 20 the total amount of active and opened notifications can be seen. The reader may note that in the current monitoring system, an alert is sent only as the notification is opened. The messages which state that a notification is active is simply helpful for knowing that a notification is active, i.e., not closed. In order to catch notifications which have been opened before the program was started, the aggregation framework also listens for active notifications. As seen in Fig. 20, the peak value is between 25000 and 30000 notifications, which may sound alarming. It is worth noting that this number contains uncritical notifications which do not create a ticket, and also that the same machine may have several active notifications. The host group also contains thousands of nodes which reflects in the amount of notifications. The total amount of notification messages (over 1.7 million) is also not surprising, since the active notifications are included and the timespan is several days.

If the active notifications are removed, and only opened notifications are considered, the graph changes. Roughly 13.000 out of 1.7 million messages represent active notifications. In percents, this number is around 0.75%. This is illustrated in Fig. 21 with the same data as in Fig. 20, but only showing opened notifications. It can still be observed that there are over 300 notifications opened at the peak, and four bursts of more than 100 notifications have been observed during two days in the beginning.

When HG1, the same host group as in the two previously discussed figures, was observed by the monitoring framework, the result was considerably different. This can be seen in Fig. 23. There is at most three relevant notifications which according to the rule for that host group would trigger an alert. The reason for the differences in the notification peaks (300 vs. 3) can mainly be explained as follows: there are only four notification types that are considered to be of importance. Even if the graphed amount of notifications in Fig. 21 would not generate tickets, they can still be considered to be a kind of noise that needs to be filtered out when inspecting the graphs in Kibana. The amount of notifications which are opened and create a ticket are seen in Fig. 22. It can be noted that two notifications account for much of the opened tickets. Using the framework to automatically filter a certain amount of these notifications away from the start can be considered to be a significant improvement to the workflow. And still, the data does not disappear, but will be available in Kibana if it is necessary to study it at some point.

To confirm that the notification framework would work in a consistent manner, a comparison was made with the data that is available in Kibana. The author created

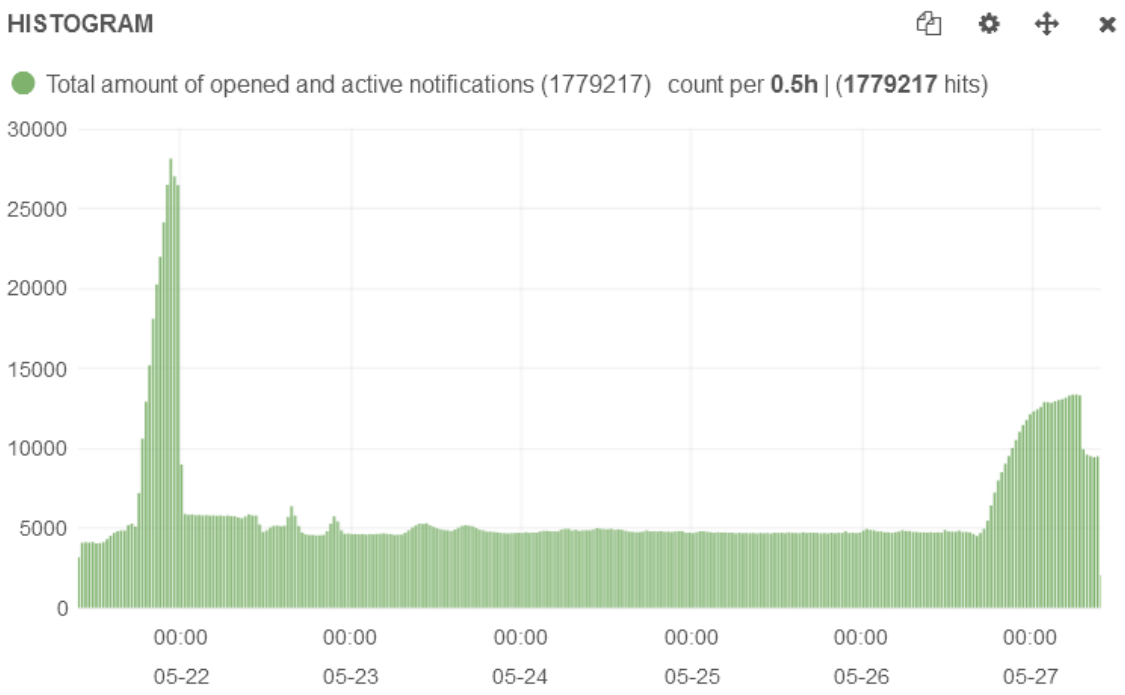

Figure 20: The total amount of *active and opened* (mostly) uncritical and critical notifications in Host group 1 as shown in Kibana.

a graph in Kibana, which contains the occurrence of the same four notifications that the framework observes for HG1. The graph is presented in Fig. 24. The framework graph, seen in Fig. 23, was created by parsing the log files of the framework, since at the point the experimentation was done, the framework's ability to send actual notifications to Kibana was still under development. The expectation from the author was, that the graphs would be more or less the same. However, the author observed three key factors that can be made accountable for the minor, but visible, differences in Figures 23 and 24. The first remark is that the graph from Kibana shows only opened notifications, whereas the framework observes both open and active notifications. This leads to a slightly higher notification count in Fig. 23 around the peak. The reason for not graphing the active notifications in Fig. 24 is that the same machine could be counted twice, since, to the authors best knowledge, it is not possible to out of the box create a graph in Kibana 3, which shows a unique count of a certain field. If both active and opened notifications would be shown, the graph would thus look completely different and not reflect how many new notifications have appeared. This is the reason for emphasizing in the figure captions that the notifications are opened in Fig. 24, and unique active and opened in Fig. 23.

Lastly, the third reason for differences was found to be the binning (value grouping) of the histogram. The first Kibana graph that the author created was very different, since the measurements were grouped on a hourly basis. The framework graph in Fig. 23 is based on physical width, which made it difficult to create a suitable

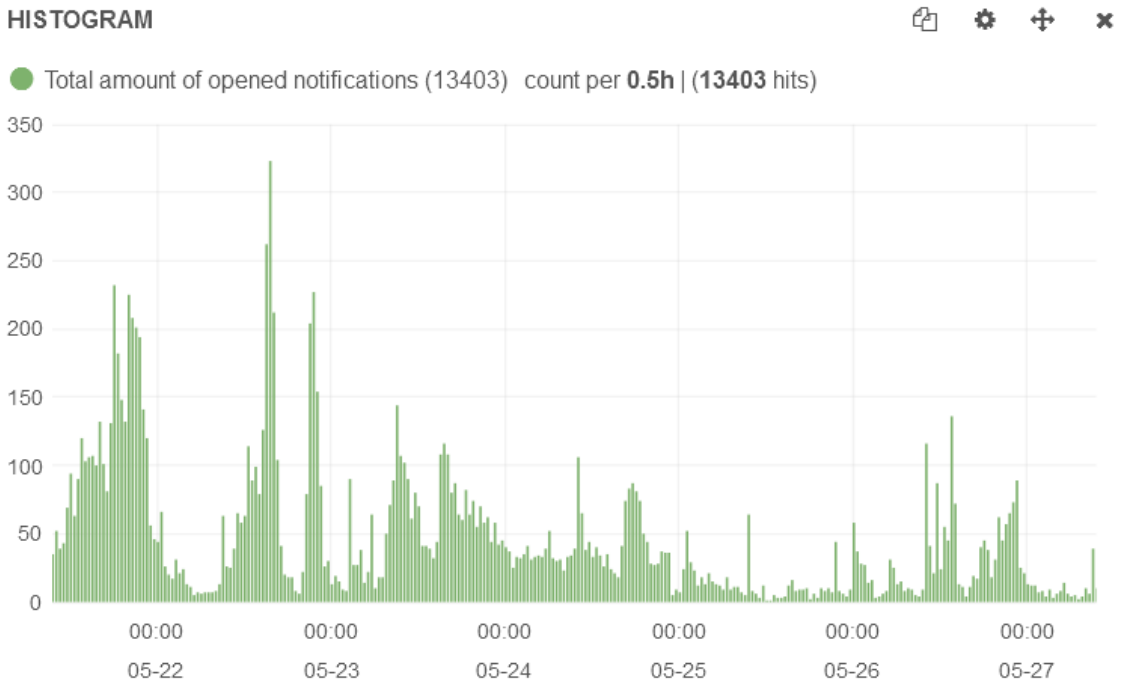

Figure 21: The total amount of *opened* (mostly) uncritical and critical notifications notifications in Host group 1 as shown in Kibana.

binning for both graphs. It would be possible with additional calculations in the log-file parsing code, but was deemed to be out of the scope for this thesis since the graphs are merely used as illustrations of indications, not proofs. A matching binning was quickly found in Kibana by using a 0.5 hour interval, and it can be seen that there is a correspondence between the notifications that the framework noted and the notifications that actually were opened during the same time interval. Finally, comparing the relevant amount notifications seen in Fig. 23 to the total amount of opened notifications in Fig. 21 shows the framework's capability to filter effectively and can be seen as evidence for confirming the first experiment hypothesis that was discussed previously.

The second experiment was to investigate tunable parameters, having the hypothesis that the cooldown time and thresholds would be most important. However, as seen in Fig. 25 which shows two graphs, choosing which notifications to include or exclude in the configuration phase can create significant differences in how the framework performs. The upper graph shows the notification count with all eight notification types that were specified as interesting for HG2 and the lower graph the notification count when a single notification type is excluded. The notification in question is a `YUM_error` notification which is a custom notification (i.e., not provided by the monitoring team, but created by a service responsible) that looks for errors in the YUM packet managers log files and creates a notification if there is something to investigate. It can be seen as an insignificant notification, but was included by

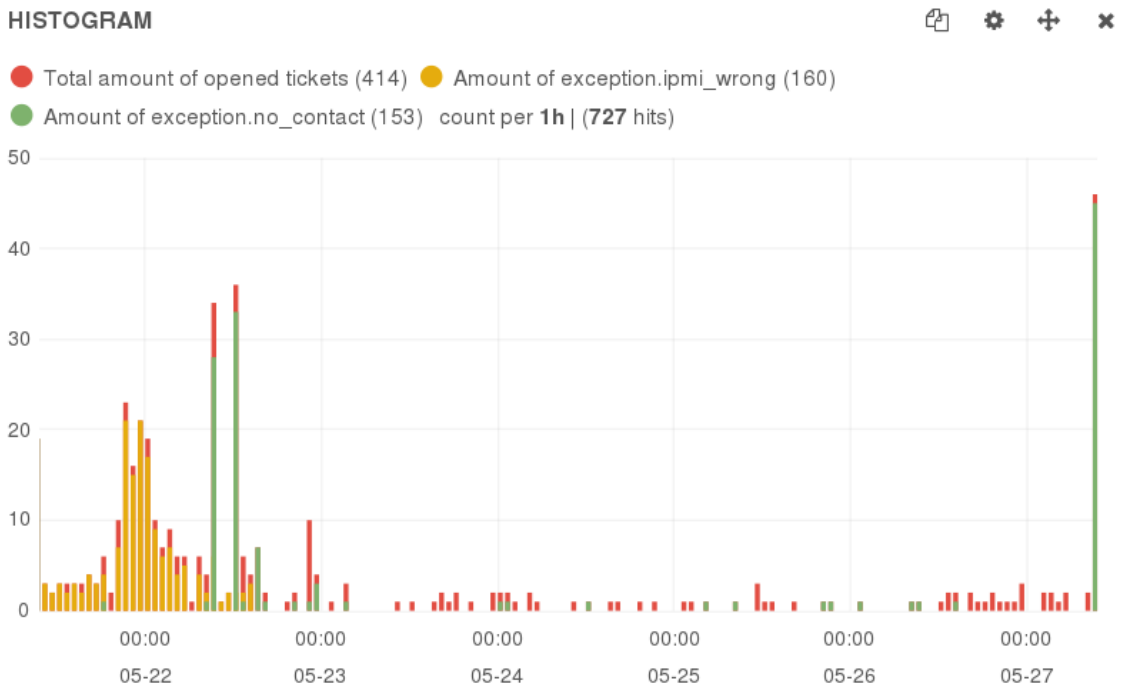

Figure 22: The total amount of *opened* notifications in Host group 1 (red) during a 6-day run of the framework which actually open a *ticket*, showing that two specific notifications (yellow, green) have caused sudden bursts.

specification. Having a high enough threshold can make such notifications very useful for determining when something is wrong on a large scale rather than in independent machines.

This brings up the point of tunable parameters. Fig. 25 shows that HG2 would need a threshold slightly above 40, or it would need to be investigated if unsignificant notifications are getting caught and account for added noise in the graphs. The influence of cooldown time cannot really be observed in the figure. At the visible smaller pikes in the beginning of the graph, it mostly rises and stays high for intervals as long as a day. Having a short cooldown time could cause too many alerts, and it would not be feasible to have day-long cooldown times. Further investigation is necessary to be able to take a stance on the significance of the cooldown time. Lastly, regarding a general limitation of the framework to avoid potentially harming host groups with an excessive amount of actions, the difference of including or excluding a certain notification is very visible in Fig. 25.

If the threshold is set to a conventient limit of around 40 for HG2 as previously discussed, a sudden burst of a single notification (which may even have been placed in the configuration as a test or by mistake) can cause the highest alert to be set of and cause a false alarm. In the case shown in Fig. 25 the notification burst was active for at least half a day. Neither a high threshold or a long cooldown time would be good solutions, since the high threshold would silence every other notification in

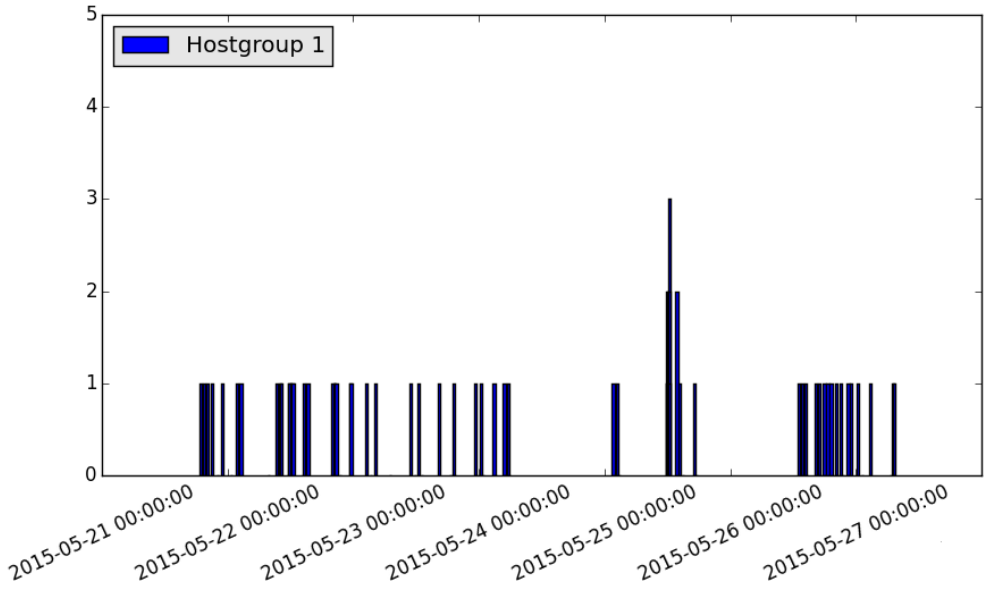

Figure 23: The amount of noted (*unique active and opened*) notifications in Host group 1 during a 6-day run of the framework.

the figure, and a long cooldown time still would create alerts if the situation goes on for more than half a day. Thus, as a remedy to false alarms, tuning and knowledge of how the host group behaves is necessary. This task and this knowledge can be done and acquired manually, but automating this step or aiding the manual process are candidates for future work.

A summary of the number of notifications that were active, opened, which opened a ticket, and which the framework noted can be seen in Fig. 26. Four permille of the active or open messages that passed through the framework were such that they were noted. Comparing the amount of notifications which created tickets (414) and the amount of notifications noted by the framework (70), it can be seen that less than a fourth of the tickets needed immediate attention. When comparing the distribution of the notifications that the framework noted, seen in Fig. 23, it is seen that there is often one machine which has a problem, and only on one day out of six, two or three different machines have a problem. Several tickets that were noted, concerned machines which sent a `root_full` notification, inicating that the root storage partition was full. This is an example of an error that may or may not be temporary and resolve itself. It can also be argued that, since the notification created a ticket, someone might have resolved the problem manually during the course of the experimental measurements. This type of self-resolving errors may differ from service to service, and reflects the kind of environment-specific knowledge that is required from a service responsible as they write a framework configuration file.

In conclusion, the framework was through experimentation shown to observe data that was very similar to the information stored in Kibana, and to be able to filter out

Figure 24: The amount of *opened* notifications in Host group 1 during the same time as in Fig. 23 as seen in Kibana, when applying filters to exclude the same notifications as those which the framework suppressed.

a significant amount of irrelevant notifications using the inclusion and exclusion lists in the configuration files. It was also observed that the thresholds need to be adjusted in order to have a fully functional framework, and that sudden bursts need to be taken into account when setting the cooldown time and threshold limits. Finally, it can also be noted that the framework can be used to create alerts when there are several notifications that individually are insignificant, but in large amounts are significant. This is useful in cases where a standard notification (that cannot be changed easily) is interesting for a service responsible. It is then possible to add this notification to the configuration file, and receive alerts for a notification that normally would not create a ticket or send an e-mail.

6.3 Interviewing future users

To gain a better understanding of what functionality the framework is expected to provide, a short interview was held with two main service responsibles which belong to the group of possible future users. The six questions that were asked can be found in Appendix D. The questions were designed based on the theoretical background found in [66], which desecribes interviewing as a research methodology. The objective of the interview was to determine in which kind of scenarios the sensors and notifications (described in Section 3) are used, if the current notifications provide

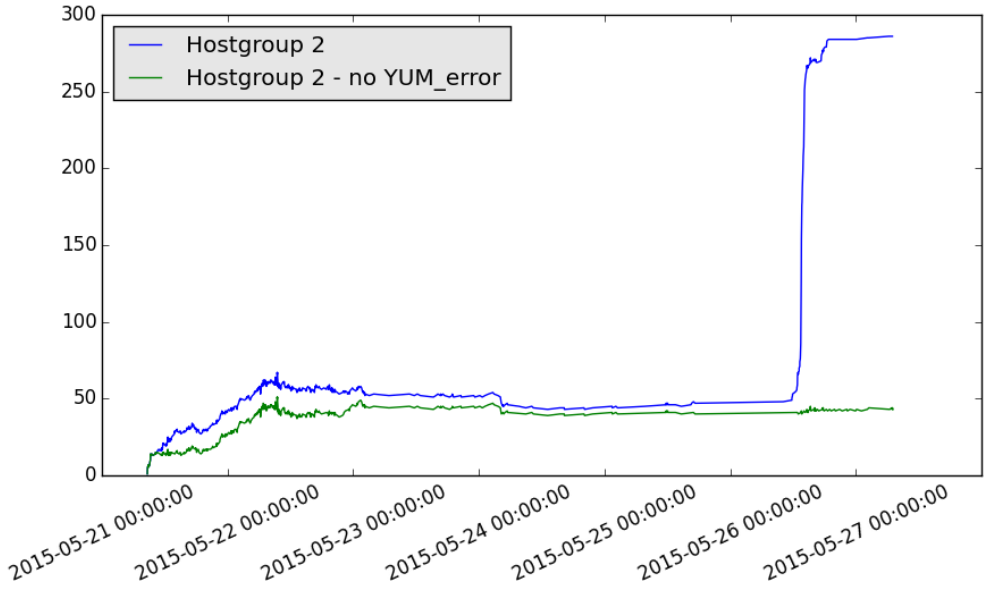

Figure 25: The amount of notifications in Host group 2 with and without an uncritical notification.

	Number	Perc. of total
Active or open	1779217	100%
Open	13403	0.753%
Open, created tickets	414	0.023%
Active or open, noted by framework	70	0.004%

Figure 26: Amount of notifications that were sent for Host group 1 in the measured interval, discerning important notifications from less important notifications, as seen in Figures 20– 24.

the desired functionality, and to investigate how filtering such notifications would benefit the service responsibles.

Both interviewees confirmed, to begin with, that they use sensors, notifications, as well as the other services provided by the monitoring team (e.g., visualisation tools). In addition, one mentioned the use of Nagios for service level monitoring, which it would be interesting to replace with Lemon sensors instead. The Kibana dashboard was praised for saving time by automatically visualizing metric readings. There was also a common need to monitor the infrastructure on machine level. This functionality has been provided by the Lemon monitoring tool since the start. In a large service, the aggregation is needed for filtering purposes, to be able to set a threshold in order to avoid a constant flow of the same information to a service

responsibles inbox or list of tickets. The small service focuses on monitoring as an important tool for providing a better service by not wasting time on false alarms and noticing real problems in time. In general, no case for these two services could be identified where monitoring could not be done by a Lemon sensor.

The preferred methods for alerting vary between the two cases. In one interviewees opinion, e-mails are acceptable, in the other case they are to be avoided. Urgent issues are wished to be relayed by tickets in the small service and by SMS or other fast methods in the large service, since the small service is less critical than the large service. One interviewee notes that e-mail notifications have become less and less interesting in favor of spending more time on having a better design to start with, depending on how critical a problem is. Indeed, it is a matter of both priorization and available possibilities. If the service is based on a commercial product, as some of the small services are, it is difficult to make changes for a better design. If the service is architectured and maintained in-house, it is easier to make improvements if a problem is found.

When receiving alerts, it is agreed upon that filtering and the possibility to set thresholds are necessary features. On one hand, if a flood of pointless problems gets reported, or if the same problem is reported several times, it disturbs the service responsible instead of helping them solve the problem. One interviewee explains that they use text messages (SMSs) sent to the phone as an urgent method, SNOW tickets for relevant issues and the Kibana dashboard for less urgent problems. They also continue to say that tuning is important, to know which problems define a broken service, and to start aggregating such problems.

6.4 Lessons learned and a discussion on automated actions

Certain risks and limits regarding the use of the framework need to be acknowledged and understood before using it in a production environment. One main risk is that a wrongly configured framework could either start flooding the service managers with e-mails if the threshold is set too low, or the wrong notification is included or excluded. To mitigate this problem, the cooldown time limit was put in place. There is also the risk that now, as the framework executes all found policies for a single host group, there are cases when one notification can trigger several policies which may have internal conflicts. These conflicts can arise if the framework is badly configured, or if the user is simply not aware of the fact that a single notification can be counted in several (matching) host groups.

Regarding these problems, there can be parallels drawn to one of the large outages in the Amazon Web Services public cloud [47] in 2011 that caused storage volumes which lost the network connection to their replicas to try to replicate themselves all at the same time as they regained the connection, which caused too much traffic, and a large decrease in service availability. The problem also spread in a chain reaction-like manner since the aggressive replication traffic started blocking other storage volumes. Scenarios such as these are important to consider when partially or completely automating actions in any environment.

Other outages in public clouds are discussed in a survey which also finds a lack

in discussion and literature around cloud service outages [54]. The survey examines outages among six services provided by five large cloud service providers and analyses the downtime and availability, as well as lists the root cause for the outage. The survey finds the largest causes of outages to be system issues, power outages, network problems, hardware or software issues, or human mistakes. More exotic causes such as natural disaster or vehicle accidents are found to be equally or more common than hacking. Still, the sample of the study is very small (78 separate events) which makes it difficult to draw conclusions. It is also difficult to classify a network routing problem caused by human mistake as being in only one of these categories. The survey discusses power outages as one of the most important issues, since it is the second most common cause following "Other system issues". In relation to the framework, software errors and human mistakes are deemed to be the most likely cause of any malfunction, i.e., rebooting or killing machines due to a bug or malconfigured policy, but as presented, best-effort attempts have been made to prevent such errors and mistakes from happening.

Similarly, automated stock trading algorithms as well as bugs in such software are believed to have caused, for example, a NASDAQ shutdown for three hours in 2013, a trading error which cost $460 millions, and a stock market crash in 2010 [40]. A government report was later published on this so called 2010 "Flash Crash", which investigates its causes and effects [19]. The report states that "... the interaction between automated execution programs and algorithmic trading strategies can quickly ... result in disorderly markets", which further reflects the risks of using software with automated actions on a large scale. Recent developments in this case caused charges to be pressed against an individual [20], in April 2015, who was arrested and accused for modifying a trading program to execute an algorithm that created large amounts of selling orders whose prices were subsequently modified, leading to the cancellation of orders. This was later suspected to have caused an imbalance that partially influenced the chain of events that lead to the crash. With this in mind, it can still be noted that trading algorithms run in an entirely different environment compared to the environment in which the framework is running, where no obvious personal benefits can be gained by attempting to use the framework in a fraudulent manner.

When analysing the interview responses, and reading that some services require "prompt notification methods" for alerting a service responsible at, e.g., 3 AM, other than SMSs, the author tried to consider the existence of commonly available communication methods other than e-mail, dashboards and other Internet-related technologies, SMSs and phone calls. Excluding more domain-specific devices such as handheld radio transceivers, which also often are operated manually and not automatically, the author attempted to investigate the use of pagers. They are considered to be an almost obsoleted technology due to the introduction of other mobile communication technologies, but remain in scarce use, in sectors where instant alerting is a critical issue, e.g., in healthcare and in emergency situations such as natural disasters where cellular networks are not a stable enough solution [50], [11].

The result of this brief investigation turned up an interesting article published in 2014 that conceptually is related to the research and development in this thesis,

namely the problem of experiencing too many alarms that are intertwined with several false alarms. In medical devices, up to 80-99 percent of alarms appear to be false [25]. As a remedy, the authors of the article, [25], have developed an "Alarm Escalation System" based on pagers, to reduce the amount of alarms and filter insignificant alarms. Currently the monitoring is done by humans who analyse and manage waveforms of medical data. The article proposes using "filters, threshold delays, fuzzy logic, and/or technical validation before alerting" which are techniques that are either the same or very similar to the techniques in this framework. An algorithm was developed that benefited from a categorization of alarms that had been done using alarm data log files. A study was conducted that showed a decline in alarms, and a shorter mean duration of the alarms, when all nurses carried a pager connected to the alarm escalating system. The authors acknowledge that further studies are necessary, but conclude that it is a viable option for hospital units that do not have human monitors at all. While this concerns a totally different area of industry compared to the aggregation framework described in this thesis, and while the alarms are of higher criticality since human lifes are at stake, it is still interesting to note that the basic and underlying principles in the end have a high similarity.

7 Results and conclusions

This section describes the results of the thesis work as well as future work that will be conducted. One of the major tasks is to fully implement the frameworks capability to edit and close SNOW tickets, as was described in Section 5.2. As was seen, it is currently possible to submit notifications to the dashboard, but the actual opening of ticket has not been tested, since it would require minor changes in the ticketing system. The current section explores future possible exensions of the framework, as well as the direction this work will be taken in. As a note, when the framework has evolved enough, it is planned to release it in the public domain, thus following the CERN licencing guidelines[35]. Finally the conclusions are presented.

7.1 Results from the survey

The survey described in Section 6.3 was during the final weeks of thesis work extended to collect responses and opinions from a larger audience, including the rest of the section and the group, as well as other sections such as the monitoring team and the cloud service team, upon whose services this framework is based. The questionnaire was opened after a presentation held by the author, for this audience, during a weekly topical meeting [55]. The intention was to familiarize the audience with the new features that the framework is able to provide, before inviting these future users to respond to the survey which would help the author understand the needs of the users, to guide the future work. A specific empty checkbox was added to the survey, that the respondents needed to check, asking for their permission to use their response in an anonymous summary of all responses, and publishing it in this publicly available thesis work as follows below.

Few responses were collected. The general opinion from the extended survey seems to be that the Lemon sensors are comprehensively used and "get the job done", but that there is a true lack for an aggregation feature such as the one this framework provides. In a sense, it is encouraging to see that different sections are expressing the same needs, which can be taken as an indication that this framework truly will fill a gap that currently exists in the monitoring infrastructure. One respondent also explicitly wishes for the threshold functionality, i.e., knowing only when 5 nodes start failing instead of getting notified when a single node fails. Regarding the notifications, there is an opinion that more work should be aimed towards providing a stable service than creating alerts, but that self-healing properties and notification systems that create less noise are wished for. Also, the majority of the respondents experience most kind of alerts as "annoying" which hints at a need to develop strategies for further minimizing the amount of alerts that are sent out. It is a challenge for future work to investigate possible solutions for fulfilling these requests which the user community made in the survey.

[35] https://legal.web.cern.ch/licensing/software

7.2 Extending the framework to cover other services

The first version of the framework was developed following the specifications and needs that were discovered in a large use case, the host groups managed by the batch team. The reason for focusing on this case, was that there are more notifications and problems to be found in a large case which makes it easier to test that the framework is functioning as expected. This amount of errors can be trivially concluded from the fact that when errors occur with a certain frequency on a machine, it is more likely to always have some errors in a very large set of machines. As a reminder, there are around 3000 machines in the batch host groups, several of them are under high load, which should not make it surprising that contact may be lost with one or two machines every day.

When considering other services, especially inside the author's section, it can be noted that most of these are small-scale services. As an example of a new service to cover, currently the version control services (VCS) have notifications for machine stats (e.g., CPU and memory consumption), but the actual service monitoring is done, for example, by cronjobs which send emails in case they fail. To integrate the VCS service monitoring into the existing framework, it is necessary to create suitable notifications and work out a policy that can be entered into the framework. It will be interesting to see how a small service can benefit from the framework, since the amount of notifications will be significantly fewer.

It is also worth noting that no special adaptation is required from the framework point of view when employing it to monitor a new service. The only thing that is required, is a configuration file, such as previously described, which indicates which notifications are significant and which are not.

7.3 Integration with Rundeck

Several softwares are being evaluated at CERN for use cases related to both monitoring as well as other topics, such as configuration management or VM deployment. The author has considered some of these softwares as candidates for integration into the framework. Rundeck[36] is one of these. It is a service for automating predefined workflows which are performed in steps. It can be seen as a collection of commands or scripts that are defined and executed in the specified order. Each step or command can be rolled back if it fails, and the whole process can be triggered either using a REST API or manually in a web interface. One methaphorical description of Rundeck that is known to the author from discussions with service responsibles, is that it is a "cronjob on steroids".

The reason for this description is that Rundeck can perform similar tasks to what a collection of scripts could be made to do, but saves the user from writing the code that surrounds actual commands, for example, code for checking the output of a command and determining whether the command was successful or not. Rundeck manages to provide a framework that performs the general steps that are involved in a service responsibles daily workflow, e.g., periodically running the same script

[36] http://rundeck.org/

on multiple machines, following up that the script does what it is expected to do, indicating if the script worked or not and providing the capability to rewind what the script did if it failed. This makes it a useful tool for service responsibles, since they can work on a higher level of abstraction, thinking of workflows, instead of, e.g., writing code for distributing the actual code across machines. The service is still being investigated and tested at CERN [45], even though there is a steady interest in using the software for various automated tasks.

7.4 Future visions and collaborating with the monitoring team

Another topic that is being explored by the monitoring team, is the use of stream computing software for aggregating metrics. The technology is also of interest to the analytics working group that was discussed previously in Section 1.2. At the moment of writing, in May 2015, the monitoring team is taking their first steps in testing out Apache Spark[37]. However, their first use case and needs are different from the goals that were outlined for this thesis work. The monitoring team receives large streams of metric data from all machines which belong to the Lemon monitoring framework and hopes to be able to compute and filter out relevant values from these streams using stream computing technology [6].

If the monitoring team decides to pursue stream computing as a core technology for metrics aggregation, it would be interesting to attempt to perform the same tasks as the newly developed framework does, but using stream computing. The benefits of a successful implementation could be an application that is more closely integrated to the monitoring infrastructure than a separate framework. If possible, an environment that would be similar to the metric and sensor infrastructure could be created, but for notification aggregation.

In the same way that users today write their custom Lemon metrics (and notification for the metrics), a centralized service based on stream computing technology could allow users to write their own notification (and why not metrics) aggregation code, that would be managed and run in the same manner as the Lemon monitoring framework is run. From a user point of view, writing the aggregation code for the stream computing service would replace the work spent on writing configuration files for the framework. This can be less error-prone, since choosing one of the major stream computing softwares instead of the framework, means choosing a tool that has existed for longer, has a larger user community, more documentation, and that has been tested and used more extensively.

7.5 Evaluation of the used methodologies

The thesis work was done focusing on solving a common problem in information technology management: ensuring that a service works, while spending as little time as possible debugging and searching for the error, and instead being able to understand and address the issue directly. In order to understand the problem domain, documentation and available literature was investigated, the author gained

[37] https://spark.apache.org/

insight on the current situation from discussions with the involved teams, and different solutions were explored, as discussed in Sections 3 and 4, before choosing to develop a framework for this task.

From the authors point of view, experience was gained in several areas. OpenStack was the only software that the author had any previous and mentionable experience with, as well as general software development experience. The configuration management tools, the CERN monitoring framework, several state-of-the-art cloud monitoring solutions that were evaluated, Perl, YAML, and a number of Python modules are among the concepts that the author learned during this thesis work.

As a conclusion regarding the methodologies for the software development, it cannot be stressed enough that it is important to be able to choose the right tools and adhere to good software development practices from the beginning, since having a fluent workflow will save time during the whole process. The author wished to find a Python IDE (Integrated Development Environment) of similar quality as the ones available for Java or the C language family, but did not succeed. PyCharm[38] was the best IDE the author discovered, but the author found it resource-consuming to use. Instead the author resorted to another popular alternative, using a text editor (`vim`) with various plugins to imitate the shortcuts and benefits an IDE provides, which proved to be a decent alternative. Writing and having unit tests for testing the framework also proved to be helpful when debugging or ensuring that the framework works as intended.

The framework was evaluated by conducting experiments as outlined in Section 6.2. The tests were planned with two hypotheses in mind that were evaluated and discussed. As a whole, information gathered from the experiments could be shown to motivate the advantages of using the framework compared to not having such a tool. As can be observed from the figures in Section 6.2, using Kibana to perform manual aggregation of the notifications proved to be a powerful mean of creating visual representations of the notifications that are sent in the CERN monitoring infrastructure. Gathering data from the framework in form of log files, and being able to compare these measurements with trustworthy data stored by components in the existing monitoring infrastructure added to the credibility of the experiments.

It is also an advantage to be able to repeat the measurements, since the involved data is stored and easily available for a reasonable amount of time. In the case of the data that is graphed in Kibana (and stored in ElasticSearch) it is for example deleted after one month, but retrievable afterwards in a different format by loading it straight from the HDFS storage that is part of the Lemon monitoring framework. Regarding the interview, it was hoped that there would be more answers available. Significant reasons may be the effort required to write down answers which describe the current use of the monitoring infrastructure, and the effort to evaluate how a service responsible would use it differently, and which functionality they would wish that an improved infrastructure would bring.

[38] https://www.jetbrains.com/pycharm/

7.6 Conclusions

Quoting one response from the extended survey on the future of the framework, "It's a real problem that we only have node monitoring and absolutely no framework for application/service monitoring", the author is confident that the framework that was developed and described in this thesis can be an asset for service managers in the department of IT at CERN. The framework is also intended to be released in the public domain as previously mentioned. This means that the public can benefit from the alerting capabilities of the framework and freely extend it for their own use, by primarily making sure their MQ software is supported by the Twisted library, and changing the code for parsing the messages from the queue, since the messages are very likely of a different format.

It would have been interesting to investigate the existing tools deeper by installing and evaluating them in comparison to the framework. In the beginning, various methods such as creating a specific policy definition language, and using a rule engine to perform the aggregation were discussed. However, it was an explicit wish to have a lightweight, and easily maintainable framework that could be integrated in the existing infrastructure without significant dependencies, and the described framework was deemed to be the best solution. The author is grateful to have gained much experience from planning, implementing and testing the framework and meanwhile learning to understand the different pieces of the existing infrastructure as well as several related technologies.

During the course of the project at least three different projects at CERN were identified which have similar ideas and goals in mind. First, the data center emergency shutdown project which is still being planned. Secondly, the project for monitoring the messaging brokers and automatically sending an e-mail if a broker malfunctions done by the distributed computing (i.e., grid) group (see Section 1.2). Finally, the third project is the monitoring team's pursuit towards gaining a deeper understanding of the metrics they collect (see Section 7.4). This indicates that the topic explored in the thesis work is of general interest in several areas of IT at CERN.

In the future, more sensors and notifications will be added to the monitoring solution of the section in order to take advantage of the framework. The ability to automatically submit, modify, and close tickets will be fully implemented. It also stands as an interesting challenge to pursue the new ideas and the possible collaboration that were outlined in Section 7.4. As a whole, this thesis work consists of a study and implementation that can be used as a helpful model for any future notification aggregation tool at CERN, whether it be based on the framework itself, or developed based on another technology.

References

[1] *OED Online*. Oxford University Press, March 2015. Big Data.

[2] A. Afaq, W. Badgett, G. Bauer, K. Biery, V. Boyer, J. Branson, A. Brett, E. Cano, A. Carboni, H. Cheung, M. Ciganek, S. Cittolin, W. Dagenhart, S. Erhan, D. Gigi, F. Glege, R. Gomez-Reino, M. Gulmini, J. Gutleber, C. Jacobs, Jin Cheol Kim, M. Klute, J. Kowalkowski, E. Lipeles, J.A.L. Perez, G. Maron, F. Meijers, E. Meschi, R. Moser, E.G. Mlot, S. Murray, A. Oh, L. Orsini, C. Paus, A. Petrucci, M. Pieri, L. Pollet, A. Racz, H. Sakulin, M. Sani, P. Schieferdecker, C. Schwick, E. Sexton-Kennedy, K. Sumorok, I. Suzuki, D. Tsirigkas, and J. Varela. The CMS High Level Trigger System. *Nuclear Science, IEEE Transactions on*, 55(1):172–175, Feb 2008.

[3] P Andrade, J Ascenso, I Fedorko, B Fiorini, M Paladin, L Pigueiras, and M Santos. Agile infrastructure monitoring. In *Journal of Physics: Conference Series*, volume 513, page 062001. IOP Publishing, 2014.

[4] P Andrade, T Bell, J Van Eldik, G McCance, B Panzer-Steindel, M Coelho dos Santos, S Traylen, and U Schwickerath. Review of CERN Data Centre Infrastructure. In *Journal of Physics: Conference Series*, volume 396, page 042002. IOP Publishing, 2012.

[5] Pedro Andrade. Exploiting open source tools to realize a new monitoring infrastructure at CERN. CNAF Seminar. Available: https://agenda.cnaf.infn.it/getFile.py/access?resId=1&materialId=slides&confId=623. Accessed: 2015-04-30.

[6] Pedro Andrade. Building a data lake. Presented at the AI Review meeting at CERN. http://indico.cern.ch/event/390892/. Accessed: 2015-04-30., 2015.

[7] Darko Anicic, Paul Fodor, Sebastian Rudolph, Roland Stühmer, Nenad Stojanovic, and Rudi Studer. A Rule-Based Language for Complex Event Processing and Reasoning. In Pascal Hitzler and Thomas Lukasiewicz, editors, *Web Reasoning and Rule Systems*, volume 6333 of *Lecture Notes in Computer Science*, pages 42–57. Springer Berlin Heidelberg, 2010.

[8] Oren Ben-Kiki, Clark Evans, and Brian Ingerson. Set Language-Independent Type for YAML™ Version 1.1. http://yaml.org/type/set.html. Accessed: 2015-04-14.

[9] Oren Ben-Kiki, Clark Evans, and Brian Ingerson. YAML Ain't Markup Language (YAML™) Version 1.2 (2009). http://www.yaml.org/spec/1.2/spec.html. Accessed: 2015-04-01.

[10] Tim Berners-Lee, James Hendler, Ora Lassila, et al. The semantic web. *Scientific American*, 284(5):28–37, 2001.

[11] Glenn Bischoff. Paging maintains relevance when message matters most. Urgent Communications Magazine, January 2012. Available: http://urgentcomm.com/mobile-data-commentary/paging-maintains-relevance-when-message-matters-most. Accessed: 2015-05-21.

[12] F. Bonifazi, A. Carbone, D. Galli, C. Gaspar, D. Gregori, U. Marconi, G. Peco, V.M. Vagnoni, and E. van Herwijnen. The Monitoring and Control System of the LHCb Event Filter Farm. *Nuclear Science, IEEE Transactions on*, 55(1):370–378, Feb 2008.

[13] Dhruba Borthakur. HDFS Architecture Guide. https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html. Accessed: 2015-04-29.

[14] Sebastian Bukowiec. Cloud service. Presented at the AI Review meeting at CERN. https://indico.cern.ch/event/392955/. Accessed: 2015-05-12., 2015.

[15] Elasticsearch BV. Getting Started (Long Version). http://www.elastic.co/guide/en/shield/current/getting-started.html, April 2015. Accessed: 2015-04-30.

[16] L G Cardoso, P Charpentier, J Closier, M Frank, C Gaspar, B Jost, G Liu, N Neufeld, and O Callot. Control and Monitoring of the Online Computer Farm for Offline Processing in LHCb. *14th International Conference on Accelerator and Large Experimental Physics Control Systems, San Francisco, CA, USA, 6 - 11 Oct 2013*, Mar 2014.

[17] CERN. LEMON - LHC Era Monitoring logo. http://lemon-monitoring.web.cern.ch/sites/lemon-monitoring.web.cern.ch/files/lemon_logo.gif. Accessed: 2015-04-22.

[18] CERN. Aerial View of the CERN taken in 2008. https://cds.cern.ch/record/1295244?ln=en, Jul 2008. Accessed: 2015-04-01.

[19] SEC CFTC and US SEC. Findings regarding the market events of May 6, 2010. *Report of the Staffs of the CFTC and SEC to the Joint Advisory Committee on Emerging Regulatory Issues*, 2010.

[20] U.S. Commodity Futures Trading Commission (CFTC). CFTC Charges U.K. Resident Navinder Singh Sarao and His Company Nav Sarao Futures Limited PLC with Price Manipulation and Spoofing. Press Release. April 21, 2015. Available: http://www.cftc.gov/PressRoom/PressReleases/pr7156-15. Accessed: 2015-06-02.

[21] P. Chochula, A. Augustinus, M. Boccioli, P. Bond, L. Jirden, A. Kurepin, M. Lechman, O. Pinazza, and P. Rosinsky. The design and operation of the detector control system of the ALICE experiment at CERN. In *Real Time Conference (RT), 2012 18th IEEE-NPSS*, pages 1–6, June 2012.

[22] David Colling, Adam Huffman, Alison McCrae, Andrew Lahiff, Claudio Grandi, Mattia Cinquilli, Stephen Gowdy, Jose Antonio Coarasa, Anthony Tiradani, Wojciech Ozga, Olivier Chaze, Massimo Sgaravatto, and Daniela Bauer. Using the CMS High Level Trigger as a Cloud Resource. *Journal of Physics: Conference Series*, 513(3):032019, 2014.

[23] M. Comuzzi, C. Kotsokalis, G. Spanoudakis, and R. Yahyapour. Establishing and Monitoring SLAs in Complex Service Based Systems. In *Web Services, 2009. ICWS 2009. IEEE International Conference on*, pages 783–790, July 2009.

[24] Corey Shaw, Zabbix Blog. Scalable Zabbix – Lessons on hitting 9400 NVPS | Zabbix Weblog. http://blog.zabbix.com/scalable-zabbix-lessons-on-hitting-9400-nvps/2615/. Accessed: 2015-04-22.

[25] Maria M Cvach, Robert J Frank, Pete Doyle, and Zeina Khouri Stevens. Use of pagers with an alarm escalation system to reduce cardiac monitor alarm signals. *Journal of nursing care quality*, 29(1):9–18, 2014.

[26] Rustem Dautov, Iraklis Paraskakis, and Mike Stannett. Utilising stream reasoning techniques to underpin an autonomous framework for cloud application platforms. *Journal of Cloud Computing*, 3(1), 2014.

[27] Jeff Dean and Sanjay Ghemawat. Google Inc. Google Open Source Blog: LevelDB: A Fast Persistent Key-Value Store. http://google-opensource.blogspot.ch/2011/07/leveldb-fast-persistent-key-value-store.html. Accessed: 2015-04-14.

[28] M. Dobson, U.A. Malik, and H.G. Elejabarrieta. Management of Online Processing Farms in the ATLAS Experiment. *Nuclear Science, IEEE Transactions on*, 55(1):411–416, Feb 2008.

[29] European Organization for Nuclear Research. Exercise 1: Introduction | IT Monitoring. http://itmon.web.cern.ch/itmon/tutorial/tutorial.html. Accessed: 2015-03-10.

[30] European Organization for Nuclear Research. Organisation | IT Department. http://information-technology.web.cern.ch/about/organisation. Accessed: 2015-03-09.

[31] European Organization for Nuclear Research. Platform & Engineering Services | IT Department. http://information-technology.web.cern.ch/about/organisation/platform-engineering-services. Accessed: 2015-03-09.

[32] Jianqing Fan, Fang Han, and Han Liu. Challenges of Big Data analysis. *National Science Review*, 2014.

[33] Ivan Fedorko. Alarming with General Notification Infrastructure (GNI). Presented at the AI Monitoring meeting at CERN. http://indico.cern.ch/event/265105/. Accessed: 2015-04-28., 2013.

[34] I. Foster, Yong Zhao, I. Raicu, and Shiyong Lu. Cloud Computing and Grid Computing 360-Degree Compared. In *Grid Computing Environments Workshop, 2008. GCE '08*, pages 1–10, Nov 2008.

[35] OpenStack Foundation. OpenStack Architecture Design Guide. Available: http://docs.openstack.org/arch-design/arch-design.pdf. Accessed: 2015-05-18.

[36] OpenStack Foundation. OpenStack Operations Guide. Available: http://docs.openstack.org/openstack-ops/openstack-ops-manual.pdf. Accessed: 2015-05-18.

[37] OpenStack Foundation. OpenStack Roadmap » OpenStack Open Source Cloud Computing Software. https://www.openstack.org/software/roadmap/, May 2015. Accessed: 2015-05-12.

[38] Python Software Foundation. 18.2. json — JSON encoder and decoder — Python v2.6.9 documentation. https://docs.python.org/2.6/library/json.html. Accessed: 2015-04-14.

[39] The Apache Software Foundation. Flume 1.5.2 User Guide — Apache Flume. https://flume.apache.org/FlumeUserGuide.html. Accessed: 2015-04-27.

[40] Michael A. Goldstein, Pavitra Kumar, and Frank C. Graves. Computerized and High-Frequency Trading. *Financial Review*, 49(2):177–202, 2014.

[41] Anthony Grossir. CC shutdown working group . https://indico.cern.ch/event/375503/, March 2015. Accessed: 2015-04-17.

[42] E. Guerra. Designing a framework with test-driven development: A journey. *Software, IEEE*, 31(1):9–14, Jan 2014.

[43] R. Guida, M. Capeans, F. Hahn, S. Haider, and B. Mandelli. The gas systems for the LHC experiments. In *Nuclear Science Symposium and Medical Imaging Conference (NSS/MIC), 2013 IEEE*, pages 1–7, Oct 2013.

[44] T Hakulinen, P Ninin, R Nunes, and T R Riesco. Revisiting CERN Safety System Monitoring (SSM). *14th International Conference on Accelerator and Large Experimental Physics Control Systems, San Francisco, CA, USA, 6 - 11 Oct 2013*, Mar 2014.

[45] Akos Hencz and Daniel Fernandez Rodriguez. Automation with Rundeck. Presented at the Weekly topical meeting at CERN. https://indico.cern.ch/event/373698/. Accessed: 2015-05-28., 2015.

[46] IBM, Paul Zikopoulos, and Chris Eaton. *Understanding Big Data: Analytics for Enterprise Class Hadoop and Streaming Data*. McGraw-Hill Osborne Media, 1st edition, 2011.

[47] Amazon Web Services Inc. Summary of the Amazon EC2 and Amazon RDS Service Disruption in the US East Region. http://aws.amazon.com/message/65648/. Accessed: 2015-05-18.

[48] CERN Cloud Infrastructure. Environment Options. http://clouddocs.web.cern.ch/clouddocs/using_openstack/environment_options.html. Accessed: 2015-05-19.

[49] Kevin Kluge. Introduction to elasticsearch. Presented at Scale 12x The Thelfth Annual Southern Calilfornia Linux Expo. https://www.socallinuxexpo.org/scale12x/presentations/introduction-elasticsearch.html. Accessed: 2015-04-29., 2014.

[50] Verne Kopytoff. Where pagers haven't gone extinct yet. Fortune Magazine, News. Available: http://fortune.com/2013/07/16/where-pagers-havent-gone-extinct-yet/. Accessed: 2015-05-21.

[51] B. König, J.M. Alcaraz Calero, and J. Kirschnick. Elastic monitoring framework for cloud infrastructures. *Communications, IET*, 6(10):1306–1315, July 2012.

[52] Samuel J Leffler, Mike Karels, and Marshall Kirk McKusick. *Measuring and improving the performance of 4.2 BSD*. University of California, 1984.

[53] CERN Lemon team. Welcome to Lemon forwarder's documentation! — Lemon forwarder 0.14 documentation. http://lemon.web.cern.ch/lemon/projects/forwarder/. Accessed: 2015-04-30.

[54] Zheng Li, Mingfei Liang, Liam O'Brien, and He Zhang. The Cloud's Cloudy Moment: A Systematic Survey of Public Cloud Service Outage. *International Journal of Cloud Computing and Services Science (IJ-CLOSER)*, 2(5):321–330, 2013.

[55] Carolina Lindqvist. Alarm aggregation. Presented at the Weekly topical meeting at CERN. https://indico.cern.ch/event/392961/. Accessed: 2015-05-28., 2015.

[56] Magalhães, João Paulo and Silva, Luis Moura. SHOWA: A Self-Healing Framework for Web-Based Applications. *ACM Trans. Auton. Adapt. Syst.*, 10(1):4:1–4:28, March 2015.

[57] Luca Magnoni. Advanced monitoring with complex stream processing. Presented at the IT Technical Forum at CERN. http://indico.cern.ch/event/382420/. Accessed: 2015-05-29., 2015.

[58] Babik Marian, Fedorko Ivan, Hook Nicholas, Lansdale Thomas Hector, Lenkes Daniel, Siket Miroslav, and Waldron Denis. LEMON-LHC Era Monitoring for Large-Scale Infrastructures. In *Journal of Physics: Conference Series*, volume 331, page 052025. IOP Publishing, 2011.

[59] Nathan Marz and James Warren. *Big Data: Principles and Best Practices of Scalable Realtime Data Systems*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2015.

[60] Matthew L Massie, Brent N Chun, and David E Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817 – 840, 2004.

[61] Peter Mell and Timothy Grance. The NIST Definition of Cloud Computing. Technical Report 800-145, National Institute of Standards and Technology (NIST), Gaithersburg, MD, September 2011.

[62] Karel Minařík. Playing HTTP Tricks with Nginx | Elastic. https://www.elastic.co/blog/playing-http-tricks-nginx, April 2015. Accessed: 2015-04-30.

[63] Hussan Munir, Misagh Moayyed, and Kai Petersen. Considering rigor and relevance when evaluating test driven development: A systematic review. *Information and Software Technology*, 56(4):375 – 394, 2014.

[64] B.C. Neuman and T. Ts'o. Kerberos: An authentication service for computer networks. *Communications Magazine, IEEE*, 32(9):33–38, Sept 1994.

[65] Fotios Nikolaidis and Daniele Francesco Kruse. Transaction aware tape-infrastructure monitoring. *Journal of Physics: Conference Series*, 513(3):032070, 2014.

[66] Sabine Mertens Oishi, editor. *How to Design an Interview for In-Person Administration*. SAGE Publications, Inc.

[67] M. Richter, K. Aamodt, T. Alt, S. Bablok, C. Cheshkov, P.T. Hille, V. Lindenstruth, G. Ovrebekk, M. Ploskon, S. Popescu, D. Rohrich, T.M. Steinbeck, and J. Thader. High Level Trigger Applications for the ALICE Experiment. *Nuclear Science, IEEE Transactions on*, 55(1):133–138, Feb 2008.

[68] K. Roed, V. Boccone, M. Brugger, A. Ferrari, D. Kramer, E. Lebbos, R. Losito, A. Mereghetti, G. Spiezia, and R. Versaci. FLUKA Simulations for SEE Studies of Critical LHC Underground Areas. *Nuclear Science, IEEE Transactions on*, 58(3):932–938, June 2011.

[69] Reijo Savolainen. Filtering and Withdrawing: Strategies for Coping with Information Overload in Everyday Contexts. *J. Inf. Sci.*, 33(5):611–621, October 2007.

[70] Reinhard Scheck. Cacti overview. In *6th TF-NOC (Task Force on Network Operation Centres) meeting*, June 2012. http://www.terena.org/activities/tf-noc/meeting6/slides/Cacti_TF-NOC_2012.pdf.

[71] ServiceNow. ServiceNow | Incident Management | ITIL Service, Incident Management. http://www.servicenow.com/products/it-service-automation-applications/incident-management.html. Accessed: 2015-04-30.

[72] Bruce Snyder, Dejan Bosanac, and Rob Davies. *ActiveMQ in Action*. Manning Publications Co., Greenwich, CT, USA, 2011.

[73] Giovanni Spiezia, Julien Mekki, Sergio Batuca, Markus Brugger, Marco Calviani, et al. The LHC Radiation Monitoring System - RadMon. 10th International Conference on Large Scale Applications and Radiation Hardness of Semiconductor Detectors. *Proceedings of Science*, RD11:024, 2011.

[74] Paul Tader. Server Monitoring with Zabbix. *Linux Journal*, 2010(195), July 2010.

[75] IT Monitoring Team. Metrics metadata manager. Accessible inside CERN.

[76] Pedro Henriques dos Santos Teixeira, Ricardo Gomes Clemente, Ronald Andreu Kaiser, and Denis Almeida Vieira Jr. HOLMES: An event-driven solution to monitor data centers through continuous queries and machine learning. In *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*, pages 216–221. ACM, 2010.

[77] Adriana Telesca, F Carena, W Carena, S Chapeland, V Chibante Barroso, F Costa, E Dénes, R Divià, U Fuchs, A Grigore, et al. System performance monitoring of the ALICE Data Acquisition System with Zabbix. In *Journal of Physics: Conference Series*, volume 513, page 062046. IOP Publishing, 2014.

[78] The Ganglia Project. Ganglia Monitoring System. http://ganglia.info/. Accessed: 2015-04-22.

[79] Jonathan Thurman. *5 Unsung Tools of DevOps*. O'Reilly Media, Inc., 2013.

[80] Emanuele Valle, Stefano Ceri, Davide Francesco Barbieri, Daniele Braga, and Alessandro Campi. Future Internet — FIS 2008. chapter A First Step Towards Stream Reasoning, pages 72–81. Springer-Verlag, Berlin, Heidelberg, 2009.

[81] Jonathan Stuart Ward and Adam Barker. Observing the clouds: A survey and taxonomy of cloud monitoring. *Journal of Cloud Computing*, 3(1), 2014.

[82] T Wijnands, Christian Pignard, and R Tesarek. An on line radiation monitoring system for the LHC machine and experimental caverns. *12th Workshop on Electronics For LHC and Future Experiments, Valencia, Spain, 25 - 29 Sep 2006*, pages pp.113–118, 2007.

[83] Justin Zobel. Experimentation. In *Writing for Computer Science*, pages 197–215. Springer London, 2014.

A Taxonomy of monitoring tools

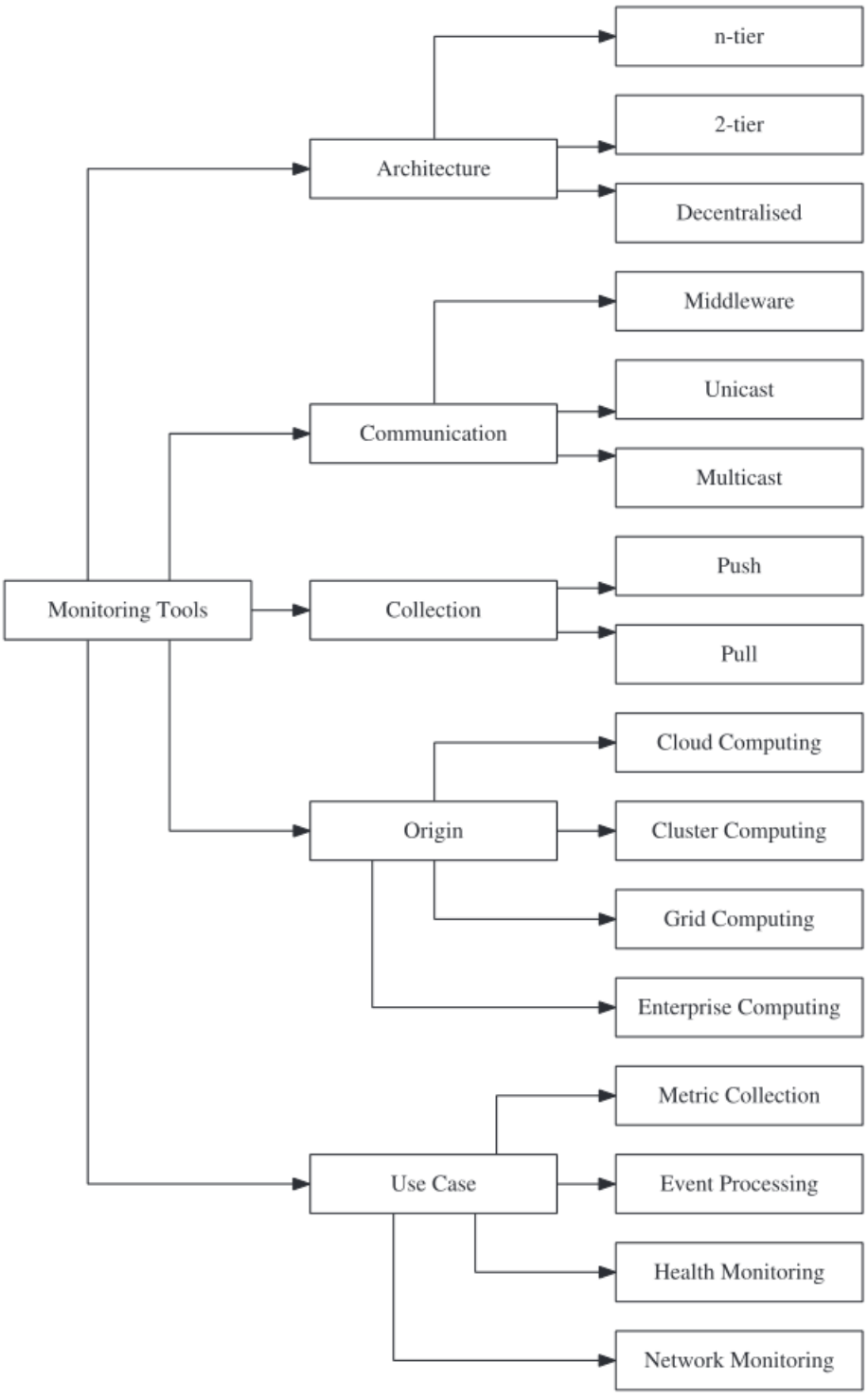

Figure A1: Taxonomy of cloud monitoring tools in [81].

B An excerpt from a sample notification dictionary

```
{ ...
u'batchsystem/batch/gridworker/aishare/share':
   {u'exception.YUM_Transactions': set([u'b6119a5b37']),
   u'exception.ipmi_wrong': set([u'lxbsu0625',
                                 u'lxbsu2307',
                                 u'lxbsu2406']),
   u'exception.pool_full': set([u'b627bfb4f8']),
   u'exception.puppetd_wrong': set([u'b6b2d448b4',
                                    u'b6cfdb6fed']),
   u'exception.swap_io': set([u'b6db383634']),
   u'puppetd_run_errors': set([u'b60c202d9b',
                               u'b61e5f7dbb',
                               u'b62dc5e259',
                               u'b646ad7d82',
                               u'b6548cce50',
                               u'b67571e248',
                               u'b67dac7ec7',
                               u'b681202474',
                               u'b697d50ba2',
                               u'b69b36936e',
                               u'b6bcd0f48c',
                               u'b6d3bcc1a4'])},
u'batchsystem/batch/gridworker/cms/cmscaf':
   {u'puppetd_run_errors': set([u'b657c5255a'])},
u'batchsystem/batch/gridworker/terminate':
   {u'exception.root_full': set([u'b67128f9a4']),
   u'puppetd_run_errors': set([u'b5026d70ea',
                               u'b52a1fc2f1'])},
u'boinc/vm':
   {u'exception.YUM_error': set([u'boincaivb07'])},
... }
```

C Profiling the framework

A short investigation was done with the profiling module `cProfile` using the following parameters: `python -m cProfile -s time monni.py`, letting the framework run for two hours, and showing the top 40 results with the total accumulated time that has been spent calling a certain function. The main observation is that much time is spent communicating with the database and polling for new messages.

```
   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
    15404 4061.570    0.264 4061.570    0.264 {method 'commit' of
                                               'sqlite3.Connection' objects}
    55859 3018.900    0.054 3018.900    0.054 {method 'poll' of 'select.epoll'
                                               objects}
 48653340   77.011    0.000  103.920    0.000 parser.py:134(_parseBody)
    38870   64.758    0.002  263.922    0.007 parser.py:62(add)
 25906145   42.967    0.000   89.965    0.000 parser.py:117(_parseHeader)
 74017158   39.144    0.000   39.144    0.000 {method 'write' of 'cStringIO.StringO'
                                               objects}
181268/60421   21.678    0.000   81.394    0.001 decoder.py:162(JSONObject)
3747978/120842   16.816    0.000   82.636    0.001 scanner.py:38(iterscan)
    15405   14.554    0.001   14.554    0.001 {method 'execute' of 'sqlite3.Cursor'
                                               objects}
  2945089    9.505    0.000    9.505    0.000 {_json.scanstring}
  5734178    9.195    0.000    9.195    0.000 {isinstance}
483546/60458    9.061    0.000 4249.272    0.070 defer.py:1088(_inlineCallbacks)
906943/906877    6.572    0.000   12.812    0.000 defer.py:503(_runCallbacks)
  1813208    6.461    0.000   12.779    0.000 parser.py:156(_unescape)
  1087919    6.340    0.000   27.049    0.000 frame.py:86(<genexpr>)
  1934086    6.309    0.000   12.911    0.000 frame.py:111(_escape)
  1027492    5.714    0.000    9.988    0.000 parser.py:149(_decode)
  2054962    4.976    0.000    8.322    0.000 frame.py:108(_encode)
  1312769    4.846    0.000   10.772    0.000 decoder.py:152(JSONString)
483516/60462    4.718    0.000 4249.902    0.070 defer.py:1241(unwindGenerator)
   680180    4.467    0.000    4.467    0.000 frame.py:114(headers)
  3428442    4.409    0.000    4.409    0.000 {built-in method scanner}
  8567071    4.360    0.000    4.360    0.000 {built-in method end}
  3747294    3.619    0.000    3.619    0.000 util.py:26(__call__)
  1027496    3.575    0.000   16.407    0.000 {map}
   241711    3.457    0.000    5.484    0.000 commands.py:317(_checkHeader)
4059829/4059685    3.001    0.000    3.406    0.000 {getattr}
  1873989    2.629    0.000    2.629    0.000 {built-in method match}
  3747294    2.603    0.000    2.604    0.000 util.py:14(get)
  1027492    2.577    0.000    2.577    0.000 {_codecs.ascii_decode}
  1934086    2.511    0.000    3.870    0.000 util.py:5(escape)
   306471    2.473    0.000    3.885    0.000 decoder.py:60(JSONNumber)
  2054962    2.406    0.000    2.406    0.000 {_codecs.ascii_encode}
  1148383    2.364    0.000    3.402    0.000 parser.py:78(_flush)
  1813208    2.334    0.000    3.579    0.000 util.py:8(unescape)
   906604    2.281    0.000    2.281    0.000 {method 'split' of 'unicode' objects}
  4170409    2.243    0.000    2.243    0.000 parser.py:162(version)
429098/429094    2.218    0.000   29.394    0.000 {method 'join' of 'str' objects}
  1148383    2.211    0.000    5.613    0.000 parser.py:93(_transition)
1571411/181363    2.124    0.000 4242.761    0.023 {method 'send' of 'generator'
                                                    objects}
```

D Interview questions for service responsibles

1. Do you use the monitoring framework tools (Sensors/Notifications/Kibana)? When? What do you think about them?
2. (a) Describe your monitoring scenario for which you want to use or create a sensor or notification.
   (b) Describe your monitoring scenario for which you do not want to use or create a sensor or notification.
3. How do you want to be notified that your service does not work? Has your opinion on this changed during your lifetime/career?
4. What is your reaction when you receive one or many SMSs, e-mails, SNOW tickets, or other alarms notifying you that a service does not work?
5. Would it help to filter or aggregate such alarms? Which of these alarms? How?
6. Anything you wish to say regarding service monitoring? Any question that was unexpected or left out?