FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Scalable High-Performance Platform for e-Science

Carlos Miguel Morais Moreira de Sousa Carvalheira

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: João Correia Lopes

July 26, 2014

Scalable High-Performance Platform for e-Science

Carlos Miguel Morais Moreira de Sousa Carvalheira

Mestrado Integrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

July 26, 2014

Abstract

Research data management is increasing in importance due to the research value that lies in the truly huge amounts of data collected during scientific investigation.

Technological advances and proliferation of cheaper hardware components has led to the creation and usage of vast networks of computers working cooperatively in distributed systems, with the current technological trend being the cloud. This work proposes a scalable architecture that is capable of meeting the demands of large-scale data storage and processing of datasets. Its data model suits the research data management platform to the needs of a variety of scientific fields of study, so long as the data is in the form of a time series.

To allow for this abstraction, we implemented an HTTP API which separates the internal workings of the platform from the business logic of the particular field of science it is used for. This business logic and presentation layer is outside the scope of this work. The API hides the complexity and scalable nature of the platform, simplifying the design of the business logic applications. In addition to storing it, data may also be processed according to standard procedures so as to expedite the scientific process undertaken by researchers. Automated field agents collect scientific measurements and upload them to the platform for processing and archiving. Custom map-reduce jobs ingest the raw data and store the derived data in the platform for future reference. All of the artifacts managed by the platform may be connected together using typed links in triple structures of the form subject-predicate-object. This allows for easier data discovery, both by automated users as well as human users. We also implemented a configuration management solution that automates and manages node deployment, with each node having different roles. In this work we also present and implement a novel approach to decentralized service discovery and load balancing, instead of the traditional centralized approach.

When performing load tests we achieved a combination of components that approximates an optimal resource usage related to the amount of load introduced. We found the main bottlenecks of the platform at various configurations and load profiles, as well as what hardware most influences each component of the system. This informs administrators as to what the optimal combination of resources is as features are introduced, bugs corrected and optimizations implemented. Future work should be directed to the study and implementation of autoscaling features in order to further automate the configuration and management of the platform. The load profiles, data and custom procedures implemented were artificial in nature and approximate expected production conditions. Testing the platform with production data will deliver better results than with artificial data.

Resumo

A gestão de dados de pesquisa científica tem crescido em importância devido ao valor existente nas enormes quantidades de dados recolhidos durante a investigação científica.

Os avanços tecnológicos e a proliferação de *hardware* mais barato levou à criação e utilização de grandes redes de computadores que trabalham cooperativamente em sistemas distribuídos, sendo a actual tendência tecnológica a *cloud*. Este trabalho propõe uma arquitectura escalável capaz de cumprir a procura de armazenamento e processamento de dados em larga escala. O modelo de dados é apropriado para várias áreas de estudo científico, desde que os dados estejam na forma de séries temporais.

Implementámos uma API HTTP que abstrai e separa o funcionamento interno da plataforma da lógica de negócio da área de estudo para o qual é usada. A lógica de negócio e a interface com o utilizador estão fora do âmbito deste trabaho. A API esconde a complexidade e escalabilidade da plataforma, simplificando o design das aplicações de lógica de negócio. Além de armazenados, os dados podem ser processados de acordo com procedimentos *standard* de forma a acelerar o processo científico. Agentes automatizados recolhem dados científicos no campo e armazenam-nos na plataforma para processamento e armazenamento. Trabalhos *map-reduce* processam os dados em bruto e guardam os dados derivados na plataforma para uso futuro. Todos os artefactos geridos pela plataforma podem ser ligados entre si através de estruturas de triplos da forma sujeito-predicado-objecto. Isto facilita a procura de informação, tanto por utilizadores automatizados como humanos. Também implementámos uma solução de gestão de configurações que automatiza e gere os vários recursos de hardware, tendo cada nó de computação um papel diferente. Neste trabalho apresentámos e implementámos uma abordagem nova de descoberta de serviços e balanceamento de carga descentralizada, contrastando com a abordagem tradicional centralizada.

Ao fazermos testes de carga, chegámos a uma combinação de componentes que aproxima uma utilização óptima de recursos, relativo à quantidade de carga introduzida. Encontrámos os *bottlenecks* principais da plataforma relativamente às diversas configurações e perfis de carga introduzidos, além de qual o hardware que mais influencia cada componente do sistema. Isto informa os administradores sobre qual a combinação de recursos óptima à medida que são introduzidas funcionalidades, corrigidos bugs e implementadas optimizações. Trabalho futuro deverá ser direccionado para o estudo e a implementação de *autoscaling* de forma a automatizar ainda mais a configuração e gestão da plataforma. Os perfis de carga, dados e procedimentos foram artificiais e aproximam as condições esperadas em produção. Testar a plataforma com dados de produção deverá trazer melhores resultados do que com dados artificiais.

“If is worth make infrastructure scalable,
is worth make it infinite scalable.”[sic]

DevOps Borat

Contents

List of Figures

List of Tables

LIST OF TABLES

Abbreviations

API	Application Programming Interface
CQL	Cassandra Query Language
DSL	Domain-Specific Language
DRY	Don't Repeat Yourself
IO	Input/Output
LiDAR	Portmanteau of "light" and "radar". Sensor technology that measures distances by illuminating the target with a laser
LTS	Long-Term Support. Refers to certain Ubuntu Linux versions which have a 5-year support guarantee
ORM	Object-Relational Mapper
RDBMS	Relational Database Management System
RDF	Resource Description Framework
RPC	Remote Procedure Call
SPOF	Single Point Of Failure
Time series	A data format in which every entry has an associated timestamp
UUID	Universally Unique Identifier
YAML	Yet Another Markup Language

Chapter 1

Introduction

In this chapter we will introduce the work to be done during the course of this project. First we will describe the context of this project and why it is relevant. Then we will present its goals, expected outcomes and contributions. Finally, we will briefly outline the structure of the document.

1.1 Context

Science is changing its ways of doing research. Before, for most of human history, one would derive conclusions and formulate hypotheses based on empirical observation of natural phenomena. Once the appropriate tools were sufficiently developed, tools like the various branches of mathematics, one would build models that explained current known phenomena, whether previously unexplained or not, and possibly predicted future unknown phenomena. Once automated tools like computers became available, phenomena could be measured *en masse* and simulated in order to exercise current models and create new ones. This is simply science taking advantage of new or improved tools; the basic methodology of formulating hypotheses, collecting and analyzing data and deriving conclusions remains unaffected. The changes to the scientific process lie in the "collecting and analyzing" part.

Another related subject is the sharing of these data and findings with the scientific community. With the ever increasing influx of information, making sense of structured research data, results and sharing them become a challenge. Efforts, often enforced by governments in legal ways, have been made to increase the sharing of research results to the public domain [HTT09, p. 25]. Acceptance of such policies has been poor at best. One may speculate this is due to laziness. If this process of sharing were automatic, we might see more adherence to those kinds of policies[1].

[1]It is my belief that science and engineering provide tools that solve problems but do not dictate under which conditions it is to be solved. In other words, as Eric Raymond put it: "mechanism, not policy"; `http://www.faqs.org/docs/artu/ch01s06.html`; this is the reasoning behind the property of "automatic" and not "compulsory". One is a mechanism, the other is a policy. As scientists and engineers, however, our tools must support whichever policies are in effect at any given time.

1.2 Motivation

The problem science faces today is in structuring, storing, curating, connecting and analyzing the true deluge of data coming from ever more complex simulations and the myriad of sensors and data collectors. Anecdotal evidence suggests [HTT09, p. 18] current tools are not suited anymore to the tasks at hand. This problem will only worsen as more data is collected and needs to be processed. This way of doing science has been dubbed the “Fourth Paradigm of Science” [HTT09].

Therefore, new tools are necessary to store the data and help scientists to curate, connect, analyze and share it. This work is an effort in that direction, of automating tasks and managing the data which originates from scientific experiments and measurements. These are often in the form of time series, which will be the primary data format address by this work.

While data is important in this context, so is its associated metadata. Being able to describe data itself is invaluable in scientific contexts. Because the metadata generated and stored is essential for searching, categorizing and utilizing data, this aspect of metadata management cannot be overlooked. More so in the context of the Fourth Paradigm, where truly huge amounts of data are handled on a regular basis.

1.3 Goals

This project will address the problems stated above. We will produce the specification of a platform and develop a working prototype that will handle the required large volume of scientific data. The work will also aim to be generic in the sense that the same platform specification may be used in various fields of science, not just the one used to validate this project, as long as the data conforms to the platform’s formats.

The platform will receive time series input data from automated field sensors via an API defined and documented in Chapter 4 and store it for later processing. Human researchers will be able to consult this data and perform simple operations on it, like 2- or 5-second averages of measurements or computation of other derived values. Custom procedures may be allowed, but were not implemented in our prototype. This feature raises security issues, addressed in Chapter 4. Researchers will also be able to store research reports or other digital objects (summit presentations, photographs of equipment or locations, etc) and connect these various artifacts, time series and digital objects, following a suitable ontology[2]. The semantic connections between the artifacts will allow both human and automated users to harvest and process them in an efficient fashion [BHBl09]; users may determine privacy policies regarding the data stored to allow for both closed and open data access.

The platform will be designed in a way that its storage and processing capacity may be increased effortlessly to account for growing demand of its services. Ideally, it will be able to scale to an arbitrarily large capacity, both in storage and computational capabilities, while maintaining

[2]That is, a set of descriptors tailored to the problem’s domain.

a reasonable ratio of resource allocation and effective capacity. This ratio will be computed and presented as an output of this project.

In summary, this work will provide the specification of a platform with the above properties, along with performance metrics that will help determine its behavior and feasibility.

1.4 Document Structure

In Chapter 2 we will make a literature and previous work review in order to assess the state of current tools and paradigms. We will then have a baseline to build upon, going forward.

In Chapter 3 we will define the requirements of the system, what it should do and what services it will provide.

In Chapter 4 we will introduce the architecture of the platform, enumerate its different components and decide on what technologies to use.

In Chapter 5 we will make considerations about the platform's development and implementation details, such as what internal machinery is necessary to run and coordinate the system and what the relationships among the different components are.

In Chapter 6 we will explain how the work was carried out: what methodology was followed, what steps were performed in what order and how we measured the success of the work.

In Chapter 7 we will discuss the data we collected following the methodology defined in Chapter 6. We will also determine how well we accomplished the goals we put forward earlier in this chapter.

In Chapter 8 we will reflect on this work as a whole and expose what insights were gained throughout. We also suggest improvements to the platform and what other avenues are opened for study in future work.

Chapter 2

Research Data Management

In this chapter we will present a review of the literature and previous works which are in the scope of this dissertation work. First we will define some key concepts in the area, then we will analyze other previous works related to this one and finally we will reflect on how the current technological trends in distributed system affect our overall work.

2.1 Introduction

"Research data management is the management of data produced by an individual or an institution in the course of its research." [LG13, p. 5]

Furthermore, this research data is very valuable, while also being expensive due to the specialized manpower and equipment required to produce it. Its value lies in the possibilities that the knowledge extracted from this data have an impact on society. However, someone has to take responsibility of managing the infrastructures necessary to host the data [Ber08]. It is undesirable, in at least an economic perspective, to push that responsibility to a single company or institution. Instead, companies and institutions should be encouraged to take collective responsibility of stewardship of data that is, in one way or another, relevant to their interests. These interests may be interpreted in three categories of increasingly larger audience. The data which has individual value, that is, data that is interesting to an individual or family. The data which has community value, that is relevant to a targeted or specialized audience. This is the emphasis of this work, data that is important to scientific communities of various fields of study. Finally, there is data that has social, political or historical significance. Repositories of this kind of data are generally managed by governments and its associated institutions [Ber08].

The costs of hardware necessary to run the infrastructures have been declining since their inception. This has lead to the increased use of cheaper and more available computational capabilities. Since more data can be stored and processed, more data has been able to be produced and stored for future use. Paradoxically, the costs of running such infrastructures have not declined but have instead shifted from hardware acquisition and maintenance to human operators that manage the systems and use them. This is explained by the higher level of expertise and training needed by both types of users[1]. Increased availability of greater computational power in networked environments enables researchers to manipulate increasingly larger and more complex datasets. This is the foundation of the Fourth Paradigm, that scientific discovery stems from the automated processing and sharing of massive amounts of data collected systematically [HTT09].

Sharing data, releasing it for use by others, in a reusable and interpretable format is a difficult problem because of economics, research incentives, intellectual property, public policy and, most importantly, the competing interests of its stakeholders [Bor11, p. 2]. This contrasts with the notion that sharing research data has the potential to increase the visibility of individual researchers and institutions as well as benefiting other research studies that use the same data. Publications that provide access to their base data have been shown to yield higher citation counts [PDF07].

In this chapter, we will explain some additional concepts about research data management: e-Science, the Research Object and Linked Open Data. Following that, we will review what other efforts are being made towards the Fourth Paradigm and finally we will survey the applicable technologies to this project.

2.2 e-Science

e-Science is the application of information technologies to the scientific process, with particular emphasis in automation and collaboration [HT02]. In particular, it takes advantage of computer science topics such as algorithms, processes networks and the proliferation of digital hardware due to its increased availability as the costs of acquiring it decreases over time, as per Moore's Law [M^{+}98, fig. 2]. Such availability and cost reduction has also enabled greatly improved interconnection and networking between hardware components, creating the opportunity for machines to communicate autonomously among themselves in a cooperative fashion. These factors increase the efficiency of the scientific process by automating simulations, data collection and processing activities which would not be feasible otherwise. The outcome of this approach seems obvious: that scientific progress becomes more expedite and reliable.

Therefore, appropriate information technology tools and practices must be adopted in order to support this approach of data- and computationally-intensive research to scientific investigation. These tools and practices must be suitable for the long term storage, access and processing of massive amounts of research data generated through the scientific process; what we might call an e-Science platform [Boh13].

[1]The users require training and expertise because of the increased amount of specialized data produced and stored.

2.3 Research Objects

A new term has arisen which is "better suited to the emerging practices of data-centric researchers": the Research Object [DR13]. A Research Object is the result of scientific investigation and may take many forms: a dataset, an article submitted to a conference, photographs of equipment and experimental setup, maps of geographic features, or other kinds artifacts which are relevant. The Research Object has the important feature of being a digital artifact, that is, it must be suitable for storage in digital devices such as computers.

It has the following properties [De 09]:

- Replayable — go back and see what happened. Whether observing the planet, the population or an automated experiment, data collection can occur over milliseconds or months. The ability to replay the experiment, and to focus on crucial parts, is essential for human understanding of what happened.
- Repeatable — run the experiment again. Enough information for the original researcher or others to be able to repeat the experiment, perhaps years later, in order to verify the results or validate the experimental environment. This also helps scale to the repetition of processing demanded by data intensive research.
- Reproducible — an independent experiment to reproduce the results. To reproduce (or replicate) a result is for someone else to start with the description of the experiment and see if a result can be reproduced. This is one of the tenets of the scientific method as we know it.
- Reusable — use as part of new experiments. One experiment may call upon another, and by assembling methods in this way we can conduct research, and ask research questions, at a higher level.
- Repurposable — reuse the pieces in a new experiment. An experiment which is a black box is only reusable as a black box. By opening the lid we find parts, and combinations of parts, available for reuse, and the way they are assembled is a clue to how they can be reassembled.
- Reliable — robust under automation. This applies to the robustness of science provided by systematic processing with human-out-the loop, and to the comprehensive handling of failure demanded in complex systems where success may be the exception, not the norm.

From these properties identified, we would like to emphasize three which are the most important, in our opinion: Reliable, Reusable and Reproducible.

Reliability, or the amenability to being automatically processed is central to our goals and the goals of the Fourth Paradigm. If a Research Object cannot be managed and processed using computer systems, its usefulness is greatly diminished. Data-management platforms will be able to make full use of Research Objects that lend themselves to being systematically processed, increasing the efficiency of interpreting it. Contrasting, if Research Objects can only be interpreted by researchers, the process takes more time, and consequently more money, and is more prone to human error.

Reusability is important in the sense that the quality of scientific investigations may be greatly improved by using, not only data gathered for a specific experiment, but also by comparing and contrasting with other related data. If a Research Object can fill both these roles, one would expect more accurate scientific results by cross-referencing multiple related data and making this process easily executable via automation of these tasks.

Reproducibility is central to the scientific process in itself. It cements or refutes the discoveries of a given investigation, essentially determining its validity. If a Research Object is reproducible, the credibility of its findings can be attested by others. This property is very similar to the property Repeatable. The difference is that in Repeatable, the very same process is taken, whereas in Reproducible, a copy of the process is used. The first ensures the Research Object is consistently produced using the same setup and methodologies[2], the second ensures that someone else can verify the results using the description of the setup and methodologies.

2.4 Linked Open Data

This topic comprises two sub-topics which are commonly presented together: Open data and Linked data. In the following sections we will describe each of this topics and how they relate to our understanding of research data management.

In this work we will use the combined concept of Open Data and Linked Data: Linked Open Data.

2.4.1 Open Data

Open data refers to the principles by which data should be shared. Sharing research data enables scientific communities to reproduce or verify each others findings. It also allows for transparent operation and making publicly funded research and initiatives available to the general public. We see this both in research contexts [Bor11] as well as political and public interest domains [KR14].

In [BK12] a set of guidelines is provided that when followed and applied to data qualifies it as open. These guidelines were initially made for government data and not research data, but they may easily applied to data in our context.

We present the list below, taken directly from [BK12], with slight adaptation:

Data must be complete: all public data is made available. The term “data” refers to electronically-stored information or recordings, including but not limited to documents, databases, transcripts, and audio/visual recordings. Public data is data that is not subject to valid privacy, security or privilege limitations, as governed by other statutes.

[2]And is not the result of an unaccounted for factor.

Data must be primary: data is published as collected at the source, with the finest possible level of granularity, and not in aggregate or modified forms.

Data must be timely: data is made available as quickly as necessary to preserve the value of the data.

Data must be accessible: data is available to the widest range of users for the widest range of purposes.

Data must be machine-processable: data is structured so that it can be processed in an automated way.

Access must be non-discriminatory: data is available to anyone, with no registration requirement.

Data formats must be non-proprietary: data is available in a format over which no entity has exclusive control.

Data must be license-free: data is not subject to any copyright, patent, trademark or trade secrets regulation. Reasonable privacy, security and privilege restrictions may be allowed as governed by other statutes.

Following Open Data policies is important in research data management contexts because of the importance of data reuse, not limited to verifying investigation results, but also because implementing Linked Data policies in a closed data environment would result in unnecessarily complex data access restrictions, possibly nullifying the advantages of Linked Data [LG13].

2.4.2 Linked Data

Linked Data is a method of publishing structured data so that it can be interlinked. This is useful because it allows not only human users to search and use it as well as automated agents. This last aspect is important to our context in which data must be machine-processable, which would become intractable otherwise.

It uses common web technologies and concepts such as HTTP and URIs. These resources are meant to be consumed by computer agents and not, as traditional of these technologies, by human users. Linked Data is simply about using the Web to create typed links between data from different sources [BHBl09].

In this scenario, one may think of linked data as connecting metadata associated with the data. These connections are made according to a defined, controlled vocabulary (ontology) and the usage of this common vocabulary allows for comprehensible metadata collection and management, ensuring data quality. This structure is amenable to the construction to a straightforward user interface for metadata curation, for example using a web interface. Additionally, the vocabulary may be expanded to incorporate more concepts, as needed.

On a final, and important, remark, the structured nature of linked data lends itself to automatic search, retrieval and processing, in a machine-to-machine environment. This is very useful for our purposes, as explained above. Automated data management activities enable researchers to collect and understand ever increasing amounts of data. Because researchers only enter the process after data has been processed by computers, creating the necessary tools and infrastructure to support the automated agents will only improve the efficiency with which they work, and therefore, the efficiency of the scientific process itself.

2.5 Related works

In this section we will describe the more prominent platforms, initiatives and efforts made towards e-Science platforms.

2.5.1 DataONE

DataONE, Data Observation Network for Earth, is a platform supported by the U.S. National Science Foundation that enables environmental research in a distributed and sustainable cyber-infrastructure that meets the needs for science and society for open, persistent, robust and secure access to well-described and easily discovered Earth observational data [LG13].

Its mission is to "Enable new science and knowledge creation through universal access to data about life on earth and the environment that sustains it."[3]. It provides a federated approach to scientific dissemination in that member nodes from various fields of research may join the DataONE infrastructure. Once a node joins the network, its information is accessible using the DataONE tools and services. The federated nature of the platform allows for high availability, fault tolerance and diversity of content.

2.5.2 ANDS

ANDS, the Australian National Data Service[4] is a platform that enables researchers to effectively manage and share the increasingly larger and more complex data created in scientific investigations. It aims at data that is better described, more connected, integrated, organized and accessible. It is a national-level repository of research resources that attempts to make better use of Australia's research output, allowing Australian researchers to easily publish, discover and use data, with the objective of enabling new and more efficient research.

It uses a "controlled vocabulary", an ontology, allowing for better management of research data. This creates the opportunity for a multi-disciplinary use of the platform in which various fields of study may formalize their own vocabulary allowing the platform to be used for many scientific topics.

[3] `http://www.dataone.org`
[4] `http://www.ands.org.au/`

2.5.3 PaNdata

PaNdata[5] is a European initiative for proton and neutron research data and aims to create a fully integrated, pan-European information infrastructure supporting the scientific process. Its Open Data Infrastructure project has started to work on implementing the software framework including data catalogs, user management and data analysis methods, seamlessly integrating the existing infrastructure of its thirteen European partners.

2.5.4 EUDAT

The EUDAT[6] project aims to create a cross-disciplinary and cross-national Collaborative Data Infrastructure that enables shared access to stored research data. Multiple research communities from various fields of study are supported by the platform; examples are climate, biological and medical, linguistic and Earth sciences. EUDAT provides services for storage and searching of research data, sharing mechanisms and data staging for off-site processing. It uses a federated approach to the distribution of its computing resources, in which users may opt to contribute their resources for a more integrated usage (referred to as "joining"[7]). Users may instead use the platform without "joining" the infrastructure.

The data staging service warrants special consideration. This service exposes data, resident in EUDAT systems, to authorized agents outside of their systems for consumption. This model imposes heavy network utilization as it must ship large amounts of data, expected to be produced and stored, across potentially great geographic distances. It is easier and more efficient instead to move the computation closer to the data and not the other way around [FGJ+09]. They have recently started using the Dublin Core (DC)[8] vocabulary as the only metadata the platform manages.

2.5.5 WindScanner.eu

The WindScanner.eu [LG13] project's goals are to provide a set of services to store and disseminate time series atmospheric measurements by means of LiDAR sensors. It is a European initiative, led by the Technical University of Denmark in compliance with the European Strategic Energy Technology Plan [MSS+12].

In more detail, the sensors will measure and generate detailed maps of wind conditions in the proximity of wind farms, covering several square kilometers. After collecting data and processing it for quality assurance on the field, the data is shipped to the central infrastructure for long-term storage, management, analysis and processing. Data may then be manipulated using procedures created by human researchers. In accordance with Linked Open Data principles, data is linked together in a semantic web of data relationships, which makes search and structuring tractable to both humans and machines. This will not also improve sharing of research objects by human

[5] http://pan-data.eu/
[6] http://www.eudat.eu/
[7] http://eudat.eu/EUDAT%2BPrimer.html
[8] http://dublincore.org/

users, but facilitates the automatic harvesting by future mechanisms of scientific research and dissemination. This project has a strong focus on open data access, although mechanisms to define access policies are yet to be implemented.

While the possibility of data processing *in situ* mitigates the need to ship data back and forth, this particular model of operation, of allowing custom procedures is a potential security risk, even with sandbox policies in place. A user, by mistake or malice, may perform procedures which are, at best, outside the scope of scientific research[9] and, at worse, degrade the performance and service standards of the system. However, a set of standard procedures, frequently used by the researchers of the domain, may be chosen to be implemented, tested and then integrated into the platform. On a final note, the design of the system was not amenable to the large scale operation needed and expected of such systems.

2.6 Technological influences

This project is heavily based on the WindScanner.eu project's features. The greatest contribution and improvement to it will be the definition of a platform specification that will easily increase in capacity to store and process large volumes of data.

In recent years, the advances in distributed systems have been staggering, from Beowulf clusters[10], to grid computing, to cloud computing. The platform takes advantage of these technological advances which will provide the infrastructure on which this platform will reside. The building block of the infrastructure is the computing instance. It may take many forms, but the most common are bare metal servers, containers and virtual machines. For our purposes and those of our platform the differences are not relevant; each type of instance behaves as if it were an independent physical machine.

In our opinion, cloud computing is a confusing term. One may only talk about cloud computing from the perspective of the end-user. The user does not really care where its data is stored or processed and instead trusts the service he uses that the data will always be available anytime and, more importantly, anywhere. However, from the perspective of developers and infrastructure architects, the term makes little sense. A distributed system design will always have to deal with different components[11] and will not look like a cloud to the designer. Quite the opposite, the cloud will only look like one from the outside, not from within, where the architect is standing.

This work has been carried out "inside the cloud", so to speak. The system behaves like a cloud to the end-user; the end-user might be a developer, a user of the system or an automated agent. It is curious to note that the infrastructure on which the platform resides may also be a cloud itself.

[9] And is therefore against the spirit, if not the letter, of the utilization policies.

[10] http://yclept.ucdavis.edu/Beowulf/aboutbeowulf.html

[11] In which some components may in fact behave like clouds.

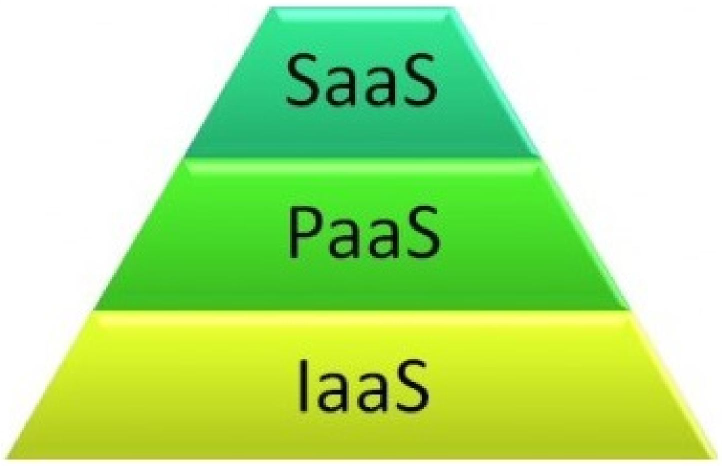


Figure 2.1: Cloud services stack

We should mention that there are three main cloud computing services: Software-as-a-Service, Platform-as-a-Service and Infrastructure-as-a-Service [LS10, p. 35]. Each provides different functionality for different use cases. This work is in the scope of SaaS. The users store, manage and process data using the platform. These actions are performed using a well defined application interface. The data may be consulted and otherwise managed anytime, anywhere and by simply following the application interface which hides the complexity of the underlying system. Likewise, the business logic and user interface application are an example of SaaS, which happens to consume services from another SaaS application (our platform).

Figure 2.1 demonstrates a simple representation of the hierarchical stack of these Services, with each layer consuming the services of the layer directly below it.

In this scenario, computing instances are commissioned and retired as needed and the platform will not worry whether it was given a bare metal instance or a virtual machine, nor where the physical host is. It's the logical connection, not the physical, that is relevant to the consumer of the infrastructure service (in this case, our platform). Therefore, computing instances and resources are a fungible commodity: one may substitute one computing instance with another and continue normal operations.

It is worth noting that these types of distributed systems favor increasing capacity by increasing the quantity of hardware devices in them [MMSW07]. This is called "horizontal scaling", as opposed to "vertical scaling" where hardware components are built with more individual capacity.

For reference, we include two widely known examples of Iaas and PaaS providers.

Amazon Web Services (AWS)[12] is a popular IaaS provider in which virtual machines of different capacities may be commissioned, both manually and automatically. It also provides a number

[12] http://aws.amazon.com/

of other custom services that ease the administration of the systems that developers build upon. The underlying networking and provisioning of all manner of physical hardware is done by AWS so that developers will focus on deploying their platforms.

Heroku[13] is a popular PaaS provider in which developers may deploy their web applications. The underlying system administration and HTTP routing, as well as scaling capabilities, are already implemented by the platform, so developers will only need to focus on developing their business logic and web user interface.

In this particular work, we used the FEUP Grid cluster as our IaaS provider, as further explained in Chapter 4. No PaaS solution was used in order to prevent vendor lock-in and to allow us the freedom to develop our architecture without the constraints of a PaaS offering. This last point is particularly important because the goal of this project is precisely the design of the platform and its performance metrics. We instead "rolled" our own solution to configure the environment in which the components operate.

2.7 Summary

In this Chapter we identified the major concepts in the context of this work and what relationship they bear. We have established that research data management is a worthy, debated and growing topic in modern society. How computers and their ability to automatically process and link research data are important tools in scientific investigations.

After, we have seen what efforts towards the Fourth Paradigm are being undertaken worldwide and how those efforts and knowledge may impact and direct our own efforts.

In the final section we discuss the technological advances in the last years that enabled such widespread dissemination of processing and storage power and the new paradigms of distributed systems that are designed to handle truly vast amounts of data. These technological and scientific advances will provide us with solid foundations for our project, not only in terms of actual technologies and software but also on conceptual and architectural paradigms.

[13] https://www.heroku.com/

Chapter 3

Problem Description

In this Chapter we will describe the features of the platform and the functionality it is expected to expose to its users.

We will heavily base the platform's requirements on WindScanner.eu [LG13]. The main difference is that this platform will be designed to be arbitrarily scalable in terms of processing and storage capacity.

We will define the broad system requirements, the relevant actors and their user stories.

3.1 General System Requirements

We will now define the broad characteristics of the system so that we may understand its function and a solution may be proposed.

- The system must store research data in the form of time series. This research data may come from various fields of study.
- The system must store the respective metadata and link it together in a format that is easily understood and accessible by human users and automated agents.
- The system must store other unstructured data in the form of BLOBs.
- The system must perform calculations and other data processing activities over the time series data. These operations should be defined and implemented by the system maintainers or may be defined by the researchers that use the platform[1].
- The system must be able to easily increase its capacity, both in terms of storage and of processing, in a cost-effective manner. This is the main focus of the work. A way to determine or approximate cost-effectiveness is described in Chapter 6.

[1]This last point of allowing *ad hoc* procedures developed by the researchers constituted a matter of debate throughout our work, eventually leading us to not implementing it for security reasons addressed in Section 5.5.

3.2 Requirements

In this section we will define the actors, entities that interact with the system and identify their specific requirements. They will be identified in the form of User Stories, a format characteristic of Agile Methodologies [Coh04].

3.2.1 Actors

In this system there are two actors: the Service Application and the Administrator.

The *Service Application* is a web service that communicates with the outside human or automated users and with the internal API via HTTP. In this sense it mainly acts as a proxy between the outside world and the underlying platform. Any interaction from the outside, such as field agents collecting data, must submit their requests through the *Service Application*.

The *Administrator* is a human user that oversees and cares for the operation of the system.

3.2.2 User Stories

In this section we will list the User Stories pertaining to this platform. The main difference with the specification in [LG13] is that this platform will provide functionality for the two actors identified above and not for human researchers directly. In our case, the *Data Provider*, *Visitor*, *Campaign Manager* and *Researcher* actors identified in [LG13] will perform their actions through the *Service Application* actor.

Table 3.1 lists the User Stories, adapted from [LG13, p. 23].

Table 3.1: User Stories

Identifier	**Name**	**Description**
US01	Register	As a *Service Application*, I want to register a new user account so that it has access to profile data
US11	Create campaign	As a *Service Application*, I want to create a new campaign so that a *Automated Agent* can upload its associated data
US12	Set access permissions	As a *Service Application*, I want to set the access permissions to campaigns so that defined users can access them
US13	Publish dataset	As a *Service Application*, I want to validate and publish datasets and associated metadata so that researchers can access them
US14	Curate dataset	As a *Service Application*, I want to curate the published datasets and associated metadata so that they are up-to-date
US21	Upload raw data	As a *Service Application*, I want to upload raw data and associated metadata so that the platform stores them

Continues on next page…

Table 3.1 – continued from previous page

Identifier	**Name**	**Description**
US22	Upload processed data	As a *Service Application*, I want to upload processed data and associated metadata so that the platform stores them
US23	Upload metadata	As a *Service Application*, I want to upload device metadata so that the platform stores them
US31	Search	As a *Service Application*, I want to filter the existing datasets by date, place and other attributes so that I can find the datasets easily
US32	Upload resource	As a *Service Application*, I want to submit new resources so that the platform stores them
US33	Describe resource	As a *Service Application*, I want to describe a resource with given attributes so that researchers can search for it
US34	Annotate Relation	As a *Service Application*, I want to annotate a relation between two resources so that Researchers get better results from searches
US35	Publish resource	As a *Service Application*, I want to publish new resources so that researchers can view them.
US41	Create subset	As a *Service Application*, I want to create a new resource associated with a subset of an existing dataset so that the platform can provide access to it
US42	Download dataset	As a *Service Application*, I want to download a dataset on behalf of a researcher so that he can analyze its contents
US43	Submit computation	As a *Service Application*, I want to submit code to be executed server-side on a particular dataset so that I get results from remote computations (optional)
US44	Review computation	As a *Service Application*, I want to view my computation status and results so that I can vouch its progress
US45	Publish results	As a *Service Application*, I want to publish my results as resources so that researchers can access them
US46	Insert comment	As a *Service Application*, I want to provide feedback on a resource so that the scientific discussion is enhanced
US47	Track comments	As a *Service Application*, I want to track the comments on my resources so that I can assess their relevance
US51	Accounts management	As an *Administrator*, I want to create accounts and grant permissions so that data providers and researchers can perform their roles

Continues on next page…

Table 3.1 – continued from previous page

Identifier	**Name**	**Description**
US52	Monitor hub	As an *Administrator*, I want to monitor the system status so that I ensure a correct operation

3.3 Summary

In this Chapter we defined the requirements of the platform to be developed during this work. This will provide us with guidance as to what features should be implemented and how to divide the platform into more manageable components.

Technological and architectural considerations about the platform as a whole, along with its individual component may be read in Chapter 4.

Chapter 4

Architecture

In this chapter we will outline the architectural and technological decisions made for this work. Whereas in Chapter 3 we defined the problem to be solved and in Chapter 2 we surveyed current trends and best practices, in this chapter we will determine how we implemented the solution to the problem.

4.1 Introduction

For each component identified in Section 4.2 we will explain their purpose in the system. We will base our technological decisions on the following characteristics, ordered from most important to least important: how adequate the technology is to the component at hand, how easy it is to integrate with the other components and our familiarity with the technology in question. In the final section of this chapter we will present the low-level architecture of the platform and how the components communicate with each other, which is an essential part of a distributed system.

4.2 High-level architecture

In Figure 4.1, the components that form the system are laid out and their connections made apparent. Each component will be explained in detail on the following sections.

As a general implementation guidance, the platform will attempt to use Open Source/Free Software[1] as much as possible. The main drivers to this decision have been identified in [CR06]. Furthermore, Open Source software may be freely audited for security or performance issues and may be modified to suit the particular interests of its users as they see fit. Being free as in "free beer" [Sta13] may not be as important in a philosophical perspective, but certainly lowers the barrier of entry in using and deploying Open Source technologies.

[1]For our purposes, we will use both terms interchangeably.

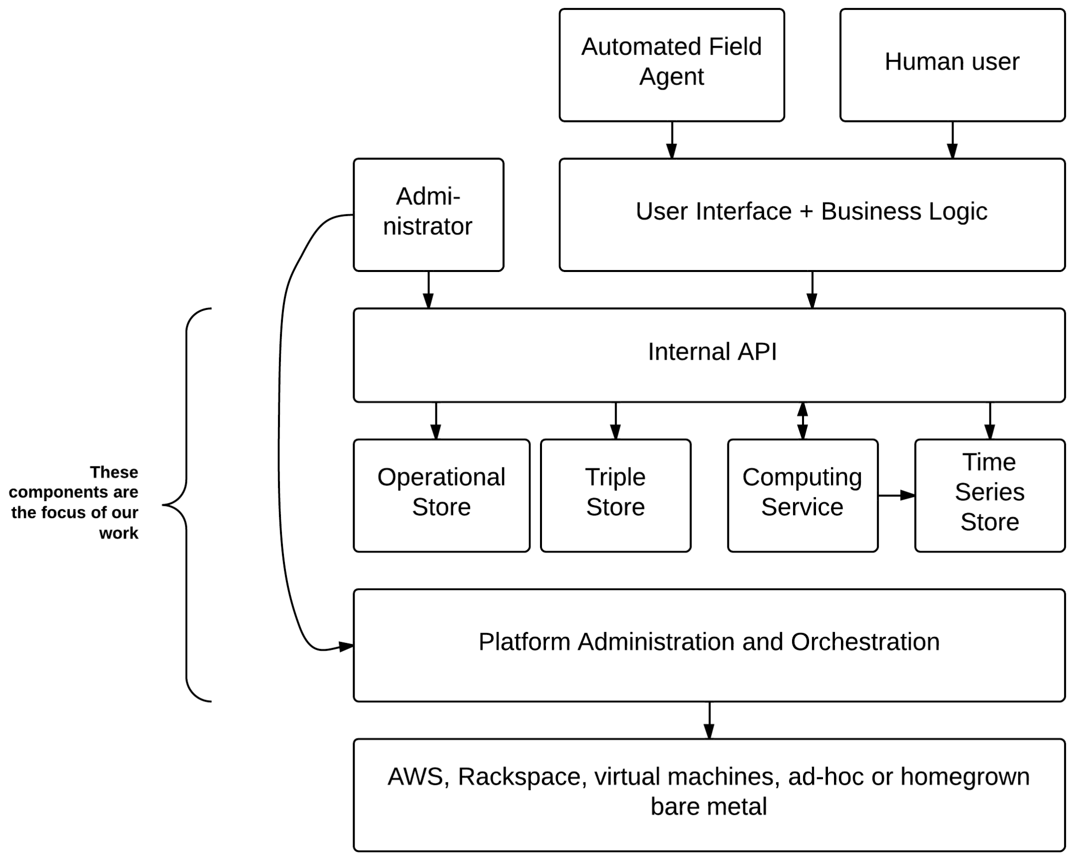

Automated Field Agent
Human user
Admi-nistrator
User Interface + Business Logic
Internal API
Operational Store
Triple Store
Computing Service
Time Series Store
These components are the focus of our work
Platform Administration and Orchestration
AWS, Rackspace, virtual machines, ad-hoc or homegrown bare metal

Figure 4.1: System architecture

All hosts will use the GNU/Linux distribution Ubuntu Server[2]. This is mainly due to our familiarity with the platform; different distributions have different ways of accomplishing the same goals and differ mainly in the philosophical approaches they take.

4.3 User interface and business logic

This component is not part of the system to be developed *per se*, but is an important part nevertheless. It implements the user interface and business logic of a specific instance of this platform. Both automated and human users will use this component to interact with the platform. Because of this, the component itself must be able to communicate with the lower layers of the system by following its exposed API.

This component is outside of the scope of this project. For reference, this component was the main focus of the Winds@FEUP [GLPR14] project[3], which consisted in developing a prototype

[2] http://www.ubuntu.com/server/
[3] http://winds.fe.up.pt/

for the WindScanner.eu project. It uses the Spring framework[4].

4.4 Internal API

This is the main entry point to the platform. All communications between the upper service layer and the platform layer must route through this component. The API will be implemented using the HTTP protocol and therefore an appropriate web stack must be deployed. Preferably, the web stack will allow rapid development of features, timely response time and easy integration with the platform. We expect response times to be strongly dominated by network activity and communication with remote services[5]. Further, since all modern frameworks provide a suitable foundation of development resources from where to start, we expect the velocity of feature completion to be a function of the developer's proficiency and familiarity with the framework. Therefore, the question of whether the chosen software stack is suitable is mainly dependent on its ability to be integrated with the other components and its ease of administration.

The proposed solution is a Django application implementing the HTTP API. Based on the Python language, it is a capable and mature web framework, surrounded by an ecosystem of solid supporting software, like Gunicorn(an application server)[6] and *supervisord*(a process manager)[7], which we plan to use. Python also has support for scientific computing with libraries like SciPy[8], RPy[9] and oct2py[10]. These libraries are widely deployed and increasingly recognized. This makes Python a good candidate for the chosen language. While this component will not perform scientific calculations, it is worth noting that there exists a framework for workflow definition and asynchronous task running, Celery[11], which uses Python. We can then easily integrate Celery, Django and scientific Python libraries into a suitable and cohesive solution to our problem.

We implemented this component using Django and various supporting libraries. The full code and documentation may be consulted elsewhere. The API provides CRUD access to the various resources made available by the other components.

We found the need to use an HTTP server that takes requests from the outside and passes them on to Django. The two most prominent alternatives are Apache HTTP server[12] and nginx[13]. Both are time-proven, solid choices and both are easily interconnected with Django. Therefore the choice is reduced to developer preference. We then chose nginx, for its simpler configuration directives and low resource usage when compared to Apache.

[4] http://projects.spring.io/spring-framework/

[5] Remote from the point of view of the API, not from the point of view of the whole system. In other words, remote here refers to services that live on machines other than the machines on which the application implementing the API lives but are still part of the platform.

[6] http://gunicorn.org/

[7] http://supervisord.org/

[8] http://www.scipy.org/

[9] http://rpy.sourceforge.net/

[10] https://pypi.python.org/pypi/oct2py

[11] http://www.celeryproject.org/

[12] http://httpd.apache.org/

[13] http://nginx.org/

4.5 Operational Store

The Operational Store database will house information regarding the system itself. This includes user permissions policies, administrative information such as known hosts, procedure metadata[14], job metadata[15] and, importantly, dataset metadata.

A Relational Database Management System (RDBMS) is suitable for this component, and MySQL[16] and PostgreSQL[17] are the more popular choices. Both systems are capable, performant and suitable to our needs. Since they are both as easy to integrate into the platform and administer, even in a highly available distributed setup, the choice is reduced to developer familiarity and preference. We then choose PostgreSQL.

We will now discuss the data model of this particular component. The figures were generated automatically via introspection of the tables in the database and are grouped by the *apps* implemented in Django. *apps* are simply autonomous modules of a Django project[18]. The tool used was *django-extensions*[19].

The connections between the tables represent relationships. A One-To-Many relationship is represented with a line and a circle on one end. The circle denotes the Many multiplicity. A Many-To-Many relationship is represented by a line with a circle on both ends. Lines are annotated with the foreign key field on the target table and the original table, in brackets.

Bold or black attributes are mandatory, whereas gray attributes are not. Italicized attributes are inherited from another class. Below the name of the table, in italics, there are the classes from which that class inherits. With the exception of CPermission all tables inherit from ResearchObject, an abstract class. The abstract classes defined are ResearchObject and Procedure and they are described below. Representing them in the diagram would reduce its legibility and so they have been left out.

Figure 4.2 (on page 38) shows the first part of the data model for the Operational Store. We will explain each class, their relationships and how they contribute to the implementation. This data model was derived from the domain model defined and described in [LG13]. All classes have an *id* attribute which is an auto-incrementing integer. This is the default behavior in Django and we saw no good reason to override it.

First we will explain the abstract class *ResearchObject*. It is a simple class that provides the basis for all the others. It has a *name* attribute which is mandatory and is a human-readable short name for the object. The *description* attribute is also a human-readable long name or description of the object. The *metadata* attribute is a machine-readable text field and represents a part of the trade-off we made between a tight fit for a given field of science and generality for any field of study. Because we cannot know in advance what attributes each field of study would like to represent, we present the option of string-encoding arbitrary data (using JSON, perhaps) in each

[14]Procedures are computations the users may apply to datasets in order to derive new datasets.
[15]Jobs are instances of procedures applied to datasets.
[16]`http://www.mysql.com/`
[17]`http://www.postgresql.org/`
[18]`http://docs.djangoproject.com/en/dev/ref/applications/`
[19]`django-extensions.readthedocs.com/en/latest/`

object of this database. It is, however, inelegant and cripples the ability to search objects on these attributes. In newer versions of PostgreSQL (9.3 and up) searching on JSON columns is supported, however, to our knowledge, it is not natively supported in Django.

The class *Archive* represents the metadata of a BLOB. It may or may not be related to a *Campaign* or a *Dataset*, as evidenced by the foreign keys defined. The only mandatory attribute is the *file_field*. It holds metadata relative to the BLOB: size, name, URL. Note that the BLOB is not actually stored on the database, but lives instead in the BLOB store; this attribute is only an alias.

Further down, *Sites* are objects that represent a specific geographic location where *Campaigns* are carried out. Campaigns are the catchall name for groups of *Datasets*. The nomenclature originated from the Winds@FEUP project, but it is general enough that other fields of science will find it amenable for use. *Event* objects denote some notable occurrence in the course of a *Campaign*. The *Device* class stores metadata about the devices used to generate a *Dataset*. It is the first example so far of a Many-To-Many relationship. Finally, the *Dataset* object is of central importance to the whole platform. It stores metadata about a group of *Datapoints* such as the time interval of the *Dataset* (with *highest_ts* and *lowest_ts*) and the count of points. The behavior of the *published* attribute is not implemented. We may summarize the meaning of *apps.campaign* as: *Datasets* are collected and grouped in *Campaigns* which are carried out in *Sites*. These *Datasets* are generated by one or more *Devices* and sometimes *Events* occur. Notably, the *Datapoint* class is absent from this model. It was implemented in the Time Series Store, explained in Section 4.8. Conceptually each *Dataset* has a relationship with *Datapoint* in a One-To-Many fashion.

To the right of the figure we have *apps.dataviewer*. This nomenclature is also a legacy of Winds@FEUP and represents the views or transformations the platform will apply to the *Datasets*. *Dimension* is the simplest of the classes. It represents individual columns (as noted by the *ts_column* attribute) in the Time Series Store, the input data coming from field sensors and also result data from procedures applied to the data. So, for example, if a field sensor collects temperature data in Centigrade, one would define such a *Dimension*. The same would occur if another sensor collects pressure in Bar. If we were to implement a *Calculator* (see below) that would convert Centigrade to Kelvin we would need to create another *Dimension* relative to temperatures in Kelvin. Note that if some field sensor collects data in Kelvin it could very well reuse this *Dimension* for its *Datasets*. The *units* attribute is a human readable unit identifier; in the example above we could use “C” or “K” as the units. *datatype* is a hint to the Time Series Store as to the type of column it should define.

Filters, *Calculators* and *Aggregators* inherit from an abstract class *Procedure* (and so they are *Procedures*) which in turn inherits from *ResearchObject*. *Procedure* simply defines an attribute, *async_function*, that is the name of the function defined in the Compute Service code that must be called whenever the given *Procedure* is scheduled. *Aggregators* perform aggregation functions, like averages or medians or counts or whatever *async_function* is set. The *interval_in_seconds* attribute determines over what time span the aggregation should be calculated. The prototypical use of this feature is to calculate averages over a given interval. In essence, *Aggregators* take one or more *Datapoints* as input and return a single *Datapoint* as output.

Calculators, on the other hand, take a number of values from a single *Datapoint* as input and return a single value as output. The input values are the ones defined in the Many-To-Many relationship with *Dimension* and the output value is the One-To-Many relationship with the same table. So, taking the example above, we would define a *Calculator* whose input is the *Dimension* Centigrade and the output is the *Dimension* Kelvin. When we apply this *Calculator* to a *Dataset*, the code would go over every *Datapoint*, take the *Centigrade* input, perform the calculation and write it as the output in the Kelvin column. We could similarly (albeit in a more contrived example) determine that another *Calculator* takes as input the *Dimensions* Mass and Acceleration and the output is Force. Note that this object is just the metadata of the *Procedure*; the actual code is defined elsewhere in the Compute Service.

The last entity, *Filters*, were not implemented in our platform. Their goal is to filter the *Datapoints* and determine which ones should be passed on to the next *Procedure* in the chain. They take one or more *Dimensions* as arguments (as per the relationship) and return either True or False, whether the *Datapoint* should be filtered or included. The applications of this are mainly data cleaning or bounding (if the input *Dimension* is *time*, a hardcoded *Dimension*). If the *Dataset* is to be processed in this way, but the *Datapoints* should remain visible and marked in some way, perhaps it is better to define a *Calculator* that accepts the same input columns and outputs to a *Dimension* State (or Annotation maybe) whose contents may be something like: “Good”, “Over recommended temp” or “Device malfunction suspected” or any such comments suitable to the data at hand.

To make an overview using an analogy with the Linux shell, the *Procedures* are piped to one another and performed in succession. *Aggregators* downsample the *Dataset* to a more meaningful or manageable amount, the *Calculators* add values to the *Datapoints* but do not alter their number and the *Filters* work like the *grep*[20] program, filtering out *Datapoints* for the next *Procedure*.

The model of the final *app*, *apps.wsuser*, is present in Figure 4.3 (on page 39). *apps.wsuser* is used to manage the users of the platform and their permissions, as per the table *CPermission* (Custom Permissions, to differentiate from *Permission* which is already another relation defined by Django for other purposes). *WSUser* inherits both from *ResearchObject* and *AbstractUser*. This last one is defined by Django and it eases extension of the basic *User* class, also defined by it. This class stores all information pertaining to a given user of the platform, be it a researcher or an automated agent.

The *CPermission* implements the permissions a given user has, as per the relationship between the tables, and a *Resource*[21], stored in the *object_url* attribute. The attributes *object_id* and *object_type* are used internally to ease search of the target entity. The boolean fields determine which permissions the given WSUser has on the given *Resource*. Notably, the platform does not enforce permissions when a user requests a *Resource*. This decision is intentional. The *Service*

[20] `http://pubs.opengroup.org/onlinepubs/9699919799/utilities/grep.html`

[21] A *Resource* is a concept implemented by the API that layers on top of this data model and is described in the API documentation.

Application is the one responsible for enforcing the policies it sets. The rationale for this is that the *Service Application* is the final arbiter of anything that happens in the platform as a whole. This allows it to ignore or otherwise circumvent permissions at its discretion.

Since *Resource* is not a concept existent in this model[22] it does not make sense to have referential integrity between the *CPermission* table and whichever table holds the *Resource*. Furthermore, the *Service Application* only knows about the concept of Resources and this data model is agnostic to that concept. As such, when it requests the permissions of a user in relation to another *Resource* it queries for the URI for the *Resource*, not a primary key, as happens with the entities described earlier in this section. If we were to implement referential integrity the complexity of the model would increase beyond the gains we would have (if any). If no permission may be found for a given pair of *WSUser* and *Resource*, the permission is undefined. If the *Resource* does not exist yet a *CPermission* for it does, it does not matter as the *Resource* may not be found anyway. Having some amount of useless data (as in the previous situation) is tolerable and it can be easily cleaned up with maintenance scripts that test the existence of the *Resource*.

For more discussion on the architecture of this component, see Section 4.7.

4.6 Triple Store

This component will store the triples data structure according to the Linked Data principles. It supports fast query and relationship traversal suitable to the high read-operations demand expected of this component.

Both RDBMS and specialized systems could be used in this situation. Since little time has been dedicated to the study of specialized solutions like Virtuoso[23], we opted instead to use PostgreSQL. In fact, this has a number of benefits: it is a familiar technology, it is easily integrated with Django (via *rdflib_django*[24] and *rdflib*[25]) and it eases administration and deployment because there is less variety of technology to manage.

We will now discuss the data model for the Triple Store. All considerations made in Section 4.5 about the figures apply here.

Figure 4.4 (on page 40) shows the data model for the Triple Store. As the label *rdflib_django* suggests, it was created by that library with no intervention on our part. It supports literal URI triples of the form subject-predicate-object, with the first type of statement having a literal field as the object. It also features support for quadruple statements in the form context-subject-predicate-object, as per the NamedGraph class and its relationship with the other tables. Statements are not normally searched through the *id* field, but instead offer search functionality by simple equality on the attributes of the Statement tables. Note that predicates are URIs to other Resources which may be further described using RDF or other formats applicable. This allows more complex queries and inferences over the Statements using SPARQL, which is interpreted and processed by the library

[22] We know it exists through the *object_url*, but as far as the model is concerned, it is just a character field.
[23] http://virtuoso.openlinksw.com/
[24] https://github.com/publysher/rdflib-django
[25] http://rdflib.readthedocs.org/en/latest/

we used. In other words, the library we used allows for and actually implements a SPARQL endpoint. This is an important feature as it allows automated agents to perform complex searches and better find relevant information for its purposes, which would otherwise be a tedious and error prone process.

For more discussion on the architecture of this component, see Section 4.7.

4.7 Architectural considerations of the Operational and Triple Stores

Since these components are very similar, the architectural details are essentially the same, we have a single section to explain both.

These components must be performant under load, scalable to meet capacity demands, resistant to node failure and, if at all possible, transparent to the client applications. This is so the clients view the component as a single entity, easing the application's administration. We must then find a balance between ease of use, ease of administration, scaling potential and failover procedures. The first thing to do is to analyze the access patterns of the applications using the component. Next, we discuss what major options there are to scaling and service availability. Given the choices and the respective strengths and weaknesses, we will make a recommendation as to what solution should be implemented.

The usage of the Triple Store and the Operational Store is predicted to be read-heavy[26]. We may suspect this empirically if we reason that most user actions in a session correspond to consulting triples and comparatively few inserts of such triples. We may make the same assumptions regarding the Operational Store: other system components will consult it more than they will update it.

The *naïve* solution is to have one node with the database. It is trivial to setup and manage, but is not at all fault tolerant as a node failure will interrupt service. Therefore, more complexity is needed. The logical continuation is to add a second node (slave) which mirrors the data in the first (master), but is not part of the operations themselves (cold standby), either synchronously or asynchronously, for hard consistency or eventual consistency, respectively. This solves failures of the master node, but may interrupt service while the slave is being promoted and, while we added robustness to the system, we did not add capacity because the node is unused until such time it is promoted. Adding more cold slaves is administratively easy. In order to make the slave a hot standby, that is, part of the operations, the applications must be configured or modified to do so. Furthermore, to maintain consistency, the application must perform write operations only on the master, while read operations can be performed on any node. The master node is an obvious bottleneck in write operations, but there is more capacity for read operations. Adding another master requires additional locking complexity to retain consistency such as using an external tool like Zookeeper[27]. It provides increased write capacity which may be somewhat countered by the

[26] There are more read queries than write queries.
[27] http://zookeeper.apache.org/

need to synchronize with the Zookeeper external service. That task would be left to the client applications which would need to be modified.

The final solution we present is sharding. This requires an enormous coordination between all the pieces of the architecture and greater insight into usage patterns and the specific relational model, as already mentioned previously. Each shard would also need to be replicated, for failover scenarios.

The balanced approach of master/slave hot standby is to be preferred, as write capacity is not as important as read capacity. Using PostgreSQL we setup Streaming Replication in order to implement the master/slave part of the solution. In essence, the slave connects to the master and receives all writes that are requested on it. The slave then applies those writes to its own instance of the database. To implement node failure resistance, load balancing and application transparency we will use *pgPool-II*[28]. There is no other software other than *pgPool-II* with the features needed by this platform. However, at the time of writing we have discovered a tool that looks promising in helping manage a cluster of PostgreSQL nodes: repmgr[29]. Further investigation of this tool is required in order to assess its usefulness.

The *pgPool-II* processes live in the same machines as the API. This means that the API connects to *localhost* as if it were the actual database process, unaware of the distributed nature of these stores. When configuring a new API node, the orchestration software configures *pgPool-II* to loadbalance requests, aware of what machines are masters and what machines are slaves.

4.8 Time Series Store

This component will store the time series data originating from field sensors and also any derived datasets. These derived datasets are created when a user or the system itself schedule operations to be performed on base datasets. The machinery to perform the operations and process the data resides in the Computing service component (see 4.10).

Since the store is expected to service high volumes of both read and write operations we recommend the usage of a NoSQL database [HAMS08, 5.4][Str10, 2.1]. Of the ACID properties guaranteed by RDBMS, (strong) Consistency is the one that is less critical to the system. Eventual Consistency is a reasonable trade-off which allows us to further scale the capacity of the system. There are many systems available to choose from, but we will narrow down the analysis to the following software: MongoDB[30], HBase[31] and Cassandra[32].

MongoDB stores what it calls documents in collections[33] using a format similar to JSON called BSON. Since no schema is enforced at the collection level, this affords the developer a greater degree of freedom in utilizing the system. This provides a suitable foundation from which

[28]`http://www.pgpool.net/mediawiki/index.php/Main_Page`
[29]`https://github.com/2ndQuadrant/repmgr`
[30]`http://www.mongodb.org/`
[31]`http://hbase.apache.org/`
[32]`http://cassandra.apache.org/`
[33]Collections are analogous to tables in RDBMS. Documents are analogous to entries in tables.

to model our problem. Furthermore, all interactions with the database are made using JavaScript which is a widely known programming language, decreasing the barrier of entry to users new to MongoDB. It implements internal machinery that allows a user to perform map-reduce [DG04] operations over data, over network-connected nodes.

However, scaling MongoDB requires substantial coordination of hosts. There are two modes of distributed operation which may be combined: replica sets and sharding. Replica sets are groups of nodes that mirror each other's data in a master-slave configuration. Write operations are only made to the master and read operations may be directed to any node. This increases read throughput and data safety because of mirroring. MongoDB implements mechanisms for fault tolerance and failover by automatically promoting a slave, should a master node became unusable. When the data stored is larger than one node can accommodate, sharding is necessary. In this configuration, each shard has a portion of the entire data, allowing more data to be stored; also, each shard may itself be a replica set.

There are a number of consequences that arise from implementing a sharded environment. There are two additional operating system (OS) process types to administer, apart from the regular *mongod*[34]: *mongos* and the *config servers*[35]. The config servers index the location of the documents by their shard key, assigning ranged of shard keys to different shards. *Config servers* also need their own failover mechanisms, not implemented by MongoDB. The documentation recommends using three such nodes.

Mongos are simply the processes that interface between the user[36], the config servers and the shards. They abstract the underlying distributed nature of the system from the application using it.

Shard keys are the indexing and distribution criterion for each document. They uniquely identify a document and the shard in which it resides. Choosing a good shard key is a challenge in itself and involves good knowledge of the particular data access patterns of the system.

HBase is an Apache Foundation project and its main influence is Google BigTable [CDG+08]. It uses the Hadoop File System (HDFS), which in turn is similar in concept to the Google File System [GGL03]. It integrates well with other Apache tools and platforms and uses SQL for general interaction with the system. HBase guarantees immediate consistency and allows joins over tables, which is not a requirement of this project. Its model of distributing the data is as complex as using sharding with MongoDB[37], added to the fact that HDFS is itself complex to setup and administer[38].

Cassandra is heavily based on Amazon Dynamo [DHJ+07] and shares many of its features. It was initially developed by Facebook and subsequently made open-source and is currently under the tutelage of the Apache Foundation. It exposes an interface and concepts similar to RDBMS but is not at all relational. Keyspaces[39] exist in which tables are created. There is a concept

[34] The process which serves actual read/write requests.
[35] http://docs.mongodb.org/manual/sharding/
[36] In this case, the user is the client application.
[37] http://hbase.apache.org/book/architecture.html
[38] http://hadoop.apache.org/docs/stable1/hdfs_design.html
[39] Analogous to databases in RDBMS.

of primary key, but not of foreign key. Tables may be read from and written to using an SQL-like language, CQL (Cassandra Query Language). Notably, join operations are not permitted or supported natively; this design choice is intentional. Replication, failover and data distribution is done in a much simpler way than with the above systems. Cassandra nodes form a peer-to-peer network that takes care of those issues, with identical configurations on all nodes. Read and write operations may be directed to any node and no special logic or effort is needed of the application. This greatly simplifies the administration of the platform and the implementation of client applications. Because all nodes are read/writable, we expect good throughput for all operations.

The solution we found more suitable to our work and our data formats (namely, time series) is to use Cassandra. Its table structure is adequate to our needs and its ease of administration is a strong point to consider, in an otherwise quite complex platform [Wil10].

We implemented Cassandra in a peer-to-peer architecture. The configuration is very straightforward, with a simple caveat. Some nodes are known as "seed nodes" and new nodes entering the cluster[40] attempt to communicate through these nodes to discover what other nodes belong to the cluster. No special additional configuration is needed for a node to be a seed.

The data model of the Time Series Store is fairly simple. Cassandra exposes concepts for data definition and manipulation similar to those of RDBMS. There exists the concept of keyspaces which are similar to databases. There also exists the concepts of tables which exist in the context of a keyspace, and of columns which exist in the context of a table. Tables and columns behave very much like their counterparts in RDBMS. As previously mentioned, Cassandra deliberately does not support join operations on its tables. These concepts are exposed through CQL[41] but the internal storage format of Cassandra is quite different. This difference plays a role later on during data modeling.

This component will store the datapoints that make up the datasets managed by the platform. The data model is so simple that it does not warrant a picture of it. In other words, a picture of the data model would not provide any additional information. The model is described as follows: there exists only one keyspace, *ws*. In that keyspace only a single table exists, *tsstore*, in which all datapoints will be stored, one datapoint per row.

That table has the following columns: *bucket*, *dataset* and *time*. Other columns are added over time as the platform is used; this will be explained further below. *bucket* will be explained after that because it is essentially a workaround for the limitations imposed by Cassandra.

dataset contains the id of the dataset this particular datapoint belongs to. In a sense, it is a foreign key to the *id* column of the Dataset table in the Operational Store. Referential integrity is obviously not enforced. *time* is the string-encoded timestamp of the datapoint. We chose string-encoding because it is easily human readable and the timestamps may be directly used when an agent consumes the services of the API. We could also have stored the timestamp as UNIX time, and probably make more efficient use of storage space. Timestamps allow for granularity in the

[40] As per Cassandra nomenclature: `https://wiki.apache.org/cassandra/GettingStarted`.
[41] In particular, CQL3.

order of 10 microseconds; while the timestamp is actually stored in the order of 1 microsecond, the algorithms for manipulating data need more than 1 microsecond of difference between two consecutive datapoints of the same dataset to work effectively. We suspect that granularities of 2 or 3 microseconds may be possible, but 10 microseconds leaves enough room for safety. Storing the time as a string is a legacy from early architecture and development of this platform where timestamps were stored as Universally Unique Identifier (UUID) Version 1[42]. This is no longer the case. These two columns are the clustering keys[43] of the table. This means that Cassandra will store data ordered first by *dataset* and then by *time*, which enables fast queries on both of those columns. This modeling decision was highly influenced by the access patterns of the store, because when a read operation is requested the *where* clauses of the CQL statements implemented only mention these two columns.

Other columns are created whenever a new Dimension resource is added to the platform. This means all rows share the same set of columns, even though some datasets don't use some of those columns/dimensions. This is not a problem regarding storage or search capability, as Cassandra will not store null values. When the datapoints related to a dataset are requested, the CQL statement will state on which columns to project the search; these columns coincide with the Dimensions associated with the Dataset in the Operational Store, therefore only the relevant data is returned. Remember that derived values from Calculator procedures need a Dimension to be stored and therefore quite a large number of dimensions may exist at any given time.

The final column is *bucket*. It is the partition key of the table. The goal of the partition key is to determine on which node[44] of the Cassandra cluster this particular row/datapoint should be stored. To understand the need for *bucket*, we will go over why neither *time* nor *dataset* nor the two together are suitable candidates. *time* must be search-able as an interval. That is to say, we must be able to write something like:

```
where time < '2014' and time > '2000'.
```

Cassandra does not allow partition keys to be searched this way, therefore *time* alone is not adequate at all for a partitioning key. This also precludes *time* from being part of a composite partitioning key, therefore both *dataset* and *time* are inadequate. *dataset* will be searched on an equality basis; that is to say, we write queries like this:

```
where dataset=3.
```

This is suitable to a partitioning key and to the way we perform queries on the table. However, there is a limit to the number of entities that may be in a partition: 2 billion[45]. For reference, here is the relevant portion of that link: "The maximum number of cells (rows x columns) in a single partition is 2 billion." In this case, rows and columns do not refer to the concepts in CQL but to

[42] http://www.ietf.org/rfc/rfc4122.txt

[43] http://www.datastax.com/documentation/cql/3.0/cql/ddl/ddl_compound_keys_c.html

[44] Which "vnode", actually. Cassandra partitioning model assigns one or more vnodes to a node. A node is an instance of Cassandra.

[45] http://wiki.apache.org/cassandra/CassandraLimitations

the internal representation Cassandra uses. In this link[46] we may find an explanation of how the different concepts of the storage engine and CQL interoperate, in particular in the last figure of the article. It illustrates the fact that each partitioning key will create a number of rows equal to the number of CQL3 columns that are not null, each with two columns. Each row-column pair represents a cell; this is the cell concept the link on Cassandra limitations mentions. If we assume that the prototypical dataset has 100 dimensions/columns of raw and derived data, that leaves us with 10 million datapoints/CQL rows per dataset. Assuming a sample rate of 1 second, this allows for datasets of approximately 4 months worth of data. If the devices are to be left on the field indefinitely collecting data this approach is impractical. This is why we implemented *bucket*. When a dataset reaches a certain configurable size[47] a new bucket is created and all subsequent datapoints are inserted into that bucket. This allows for an arbitrarily large quantity of datapoints per dataset, at the cost of having to manage the buckets within the code which is fairly easy.

4.9 BLOB Store

This component will store the BLOBs that researchers may upload to the platform. Objects are not expected to be significantly large (probably no more than 1GB) and the majority to be in the single digits of MB. Therefore a general purpose file system is enough for our needs. These BLOBs will be served using the HTTP protocol, indexed by URI.

There are some distributed file systems for remote data access such as GlusterFS[48], NFS[49], NDB[50] and HDFS[51]. These have not been studied in depth and because this component is not the main focus of this work, a passable but not optimal solution will be proposed. We suspect data access patterns on this component to be read-heavy, but also that the amount of data will grow beyond the capacity of a single host. We then propose that the data be mirrored on a number of hosts, avoiding increased complexity and losing our focus, while assuring availability of service and data safety.

For the file system, the default configuration of the chosen operating system is good[52]. The mirroring mechanism to be used will be BTSync[53] for its ease of use and peer-to-peer dissemination strategy.

[46] http://www.datastax.com/dev/blog/thrift-to-cql3
[47] For testing purposes, this value is set at 10000 datapoints.
[48] http://www.gluster.org/
[49] http://tools.ietf.org/html/rfc5661
[50] http://nbd.sourceforge.net/
[51] http://hadoop.apache.org/docs/stable2/hadoop-project-dist/hadoop-hdfs/HdfsUserGuide.html
[52] For reference, the default file system on Ubuntu 12.04 LTS is ext4.
[53] http://www.bittorrent.com/sync

4.10 Computing Service

This component will perform data processing and manipulation on the datasets stored by the Time Series Store (see 4.8). The procedures to be applied to the data may already be defined by the system or the researchers themselves may define their own procedures. In this work we decided not to allow researcher to perform *ad hoc* procedures because it raises security issues, discussed in Section 5.5. Regardless, all tasks and procedures are scheduled by the API endpoints, triggered on user input and on autonomous decision by the platform[54]. Additionally, we expect the compute nodes to be undifferentiated, that is, every node may perform any task given to it.

Two sub-components are needed: the task queue manager and the computing nodes. The recommended software is RabbitMQ[55] to manage the task queues and Celery processes to perform the computations themselves.

RabbitMQ implements the AMQP[56] protocol in order to handle message passing between processes. It works in a publish/subscribe architecture in which producers publish their messages to queues which are subscribed by other processes for future consumption.

Celery is a Python framework for workflow definition and task processing. It integrates well with RabbitMQ and Django, therefore it is a natural choice for this component.

The main reasons to choose RabbitMQ over Gearman[57] or Beanstalkd[58] is its easy integration with Celery (and Django) and the ability to define complex workflows of tasks for data processing.

Before we make a brief consideration on Gearman and Beanstalkd we must explain a term we use further ahead: job ordering. It is closely tied to the job scheduling mechanisms we implemented. Every time a computation is requested by the API, one job is launched. That job partitions the relevant dataset in chunks, preventing one process from loading too much data into memory and being killed by the operating system. Now we have a number of jobs, each one operating on a manageable piece of data, very similar to the map-reduce paradigm. However, one requirement of this platform is that it should allow the user to define a number of computations (procedures) to be applied in succession to a given dataset in order to create a new, derived dataset. So the next procedure in the list must wait for all the small jobs related to the previous procedure to finish. This cannot be achieved using job priority alone because only when all the jobs are completed (as opposed to reserved for processing), should the next procedure start. The ordering of the small jobs is not relevant, they are by design able to be run in any order. The AMQP protocol (and therefore also RabbitMQ) supports this feature, but not the other solutions. In this sense, it is similar to instruction pipelining in CPU architectures [HP11]. We fill the pipeline with small jobs and must wait for the pipeline to be flushed before the next set of small jobs is scheduled because the scheduler (the initial job that schedules small jobs) assumes the next batch of small jobs has

[54] Researchers may define that a dataset should be processed according to a certain procedure as its data points enter the platform. On deployment, platform operators may define the same behavior on a global scope; for every dataset.

[55] https://www.rabbitmq.com/

[56] http://www.amqp.org/

[57] http://gearman.org/

[58] http://kr.github.io/beanstalkd/

a dependency in the first batch. Future work could look into the actual dependencies of jobs and mitigate or eliminate the need for this feature.

Gearman is a simple job server. It receives requests for jobs and dispatches them to the workers. While not tied to any particular language of framework, its simplicity necessarily precludes it from having a number of features as extensive as other solutions. There is little support for determining job priority, there is no concept of queues, one cannot determine that a job must be started after a preset delay and job ordering is not supported. In fact, only the last feature is relevant for our needs, but RabbitMQ and Celery support it.

Beanstalkd is yet another simple job server, very similar to Gearman. It is more feature rich in that it supports very granular job priority, queues and delays, but still cannot schedule job ordering.

During development we ran into shared memory issues while using Celery and we switched to Gearman, for its clearer process model which eliminated the shared memory problem[59]. We implemented the Gearman job server daemon *gearmand* on separate machines in order to give us a more fine-grained control over the components of the system.

Since only AMQP supports what we call job ordering, we had to implement this feature ourselves. We could simply let a process block while its subtasks run in other processes. This has the downside of potentially blocking too many processes while in production and causing a denial of service because no processes are available for job processing. The solution we implemented is a simple counting mechanism whose state is stored in the Operational Store. When a complex workflow (more than one procedure) is requested on the API, it stores it on the Operational Store and then schedules the first job. As explained above, this job creates a number of small jobs and calls back on the API to update the number of small jobs scheduled. Each small job also calls back on the API to announce that it has terminated, decreasing the count. When the count is zero, this means all small jobs have terminated (the pipeline is empty) and the API schedules the next procedure which repeats the cycle. This will explained more in depth in Section 5.5.

4.11 Platform administration and orchestration

The final component is the platform administration. It will manage the configuration of the various types of nodes existent in the system. When a new host is added to the platform, its role must be defined beforehand and registered in the administration component. This way, the component will know how to configure the host, taking into account information resident in the Operational Store (see Section 4.5).

For simple cases, using an *ssh* session to manually configure the machine is a good enough approach. For more complex scenarios, automation tools (also known as orchestration tools) are used for increased productivity and configuration control. Configuration drift is a phenomenon where, by unstructured modifications to the machines (circumventing the automation tools, if any), the configurations of machines with similar responsibilities differ from each other, possibly

[59]More specifically, the session object to connect to the Cassandra cluster was being shared with all the workers in a particular machine and when a thread shutdown the session, others would be denied the service.

causing intermittent erroneous states to surface, which are difficult to debug. Automation tools also relieve the user from knowing in detail the target platform (e.g. the operating system); this may be seen as a move to a higher level paradigm. This has two advantages: whereas before a system administrator would have to have a much more in-depth knowledge of the platform, now he does not necessarily have to. The second, and arguably more interesting, is that new platforms that build on top of the operating system (or other platforms) may be developed to support a wider range of needs for application developers using higher level constructs.

These tools often work in a client-server architecture where there is a master node and several slave nodes. The master node holds the configuration directives, the list of nodes that it controls and how to selectively target them for configuration changes. There are some variations to this theme, but it seems to be the dominant one, among the existent tools. The configuration directives are generally written using simple markup syntax or a Domain-Specific Language (DSL). Some have full support for writing configuration using common programming languages. These options provide flexibility, when configurations are simple (using markup syntax) or power, when the need arises (using programming languages). DSLs are a middle ground where the additional cost of learning a language is a factor to take into consideration.

In any case, the configurations are text files. As such, they are prime candidates for version control, both for ease of deployment/maintenance of master nodes and for disaster recovery. If the version control system supports hooks to certain events such as commits/pushes (whatever the nomenclature may be), configuration updates may be further automated by calling those hooks such that when a new update is committed/pushed it is also immediately applied. Another suggestion is to use Continuous Integration tools to test the updates before applying them, if the configurations warrant such complex procedures to be executed. In that case, project management techniques may be useful to manage the life cycle of the configurations.

While most of the tools require a daemon process of the appropriate type on each machine, some daemons are specialized to the tool and others are generic. Generic in the sense that they exist by default in most GNU/Linux distributions and are used for purposes other than orchestration and configuration management. Specifically the OpenSSH suite and its daemon, *sshd*. Other tools use custom libraries for networking and instead of setting up and tearing down *ssh* sessions, they use a publish/subscribe architecture. This implies long lived processes on all the machines which is another “moving part” in the system, increasing its complexity. Some tools forego this in exchange for no master long lived process and use *sshd* on the slaves; the master process only runs when configuration updates are necessary and *sshd* is based on solid and well known technology, mitigating the complexity issue.

When doing the communication between master and slave, there are some network transports to choose from, depending on the tool. We already discussed *sshd*, but ZeroMQ[60] is also used in publish/subscribe architectures. In this case the master creates a pub/sub socket with an interface similar to POSIX or BSD sockets and the slaves connect to it to receive updates. This greatly increases the throughput of update events by decreasing setup/teardown times of processes and

[60] http://zeromq.org/

connections and by communicating updates to all targeted slaves at once. It does not, however, implement security as data is transferred in plain-text. At least one tool has its own custom implementation of public-key cryptography, which mitigates security issues, but because it is not as mature as other implementations (e.g.OpenSSH) it may contain more unknown bugs and/or vulnerabilities.

SaltStack[61] is a tool for system orchestration. It works in a pub/sub client-server architecture, in which there is a master (the server) and the minions (the clients). Minions connect to the master and receive configuration directives, like what packages to install and what services to run. This configuration is held by the master node in YAML format, though other formats are supported. When configuration changes occur, they are pushed by the master to all the targeted minions, who apply them. Minions can be targeted by ID, regular expression or "grains". Grains are static pieces of minion configuration. An example would be the role the minion has on the system: application server, file server, database, etc. Administrators may create their own custom grains as they see fit, such as the datacenter or rack in which the machine lives. Initially, minions already have some grains defined, such as the OS they are running.

The master holds configuration files, be they YAML to be parsed by the minions or files the minions should keep in their filesystem[62]. If the configuration is of the YAML variety, the minion will parse it and use the operating system's resources such as *apt*, *rpm*, *init* scripts or the shell to apply the configuration. All communication is encrypted using a custom implementation of public-key cryptography over ZeroMQ transport. It is therefore resistant to communication interception. The ZeroMQ transport and its publish/subscribe architecture allow the system to behave much faster than using SSH for transport since it is well suited to broadcast communication and does not have the overhead of making a connection each time.

Chef[63] is a configuration management tool written in Ruby and Erlang. It uses a pure-Ruby DSL for writing system configuration *recipes*. Chef is used to streamline the task of configuring and maintaining a company's servers, and can integrate with cloud-based platforms, such as Rackspace and Amazon EC2, to automatically provision and configure new machines. The user writes *recipes* that describe how Chef manages server applications (such as Apache, MySQL, or Hadoop) and how they are to be configured. These *recipes* describe a series of resources that should be in a particular state: packages that should be installed, services that should be running, or files that should be written. Chef makes sure each resource is properly configured and corrects any resources that are not in the desired state. Chef can run in client/server mode, or in a standalone configuration named *chef-solo*. In a client/server mode, the Chef client sends various attributes about the node to the Chef server. The server uses Solr[64] to index these attributes and provides an API for clients to query this information. Chef recipes can query these attributes and use the

[61] `http://www.saltstack.com/`

[62] An example: the master may determine that a minion must have its own /etc/hosts synchronized with the resource `salt://generic_config/hosts`; this resource is held and served by the master node.

[63] `http://www.getchef.com/chef/`

[64] `https://lucene.apache.org/solr/`

resulting data to help configure the node. Traditionally, Chef is used to manage GNU/Linux but later versions support running on Windows as well.

Ansible[65] is another tool for system orchestration. Similar to other solutions, it keeps configuration files in YAML and a list of machines to connect to: the *inventory*. These machines are grouped into roles and may be targeted this way. No daemons other than OpenSSH need to exist on any machine in order to run Ansible. The footprint on the master machine is simply a folder structure, commonly a *git* repository, with a script that is invoked by the user to parse the YAML files and run remote commands. Since it uses OpenSSH for network transport, its messages are secured in the same way any other *ssh* traffic would be, which is a good security guarantee, given OpenSSH is among the most widely implemented and reviewed security suites. However, setting up connections, running commands and tearing down connections is time consuming over SSH, which means the delay between running the command on the master node and having it occur on all controlled nodes is higher than in other solutions.

Fabric[66] is a Python library that streamlines the usage of *ssh*, with emphasis on system administration and deployment. Instead of using markup to declare the state of the targeted machine, Fabric uses *ssh* in an imperative fashion, commanding the machine to apply a series of steps. These steps are often written in the target's shell program, but there it also provides shortcuts for easy file transfer (via *sftp*). This tool does not require any long lived process on the master machine and only the *sshd* daemon on the targeted machines. In this sense it is somewhat similar to Ansible, above.

The proposed software to use is SaltStack, for its optional use of Python for the definition of configurations and the usage of ZeroMQ to deliver fast updates to the controlled nodes.

However, during development of this component we ran into networking issues with SaltStack[67] that presented a serious obstacle to productivity. After estimating the cost of lost work due to trying to solve the above problem, we decided to switch to Fabric for our orchestration tool, mainly due to our familiarity with it and the technologies and paradigms around it (OpenSSH and *sshd*).

In our Fabric scripts we explicitly define roles in our system. These roles closely match the components of our platform.

For a more thorough explanation of the actual implementation of each component and how they communicate among themselves see Section 5.

As for the administration component of the platform, it provides simple CRUD access to the entities in the Operational and Triple Stores (but not the Time Series Store) circumventing the API. This is suitable to manual corrections should the need arise, in an otherwise autonomous interaction with the API. It is implemented using the *admin* interface provided by Django.

It also supports simple management of the machines in use by the platform. The administrator inserts new machines and determines their roles in the system. Then he may order the application

[65] `http://www.ansible.com/home`

[66] `http://www.fabfile.org/`

[67] More specifically, the salt-master process could only connect to *localhost* minions, defeating the purpose of its use. We could not identify the source of this problem.

to create an inventory containing the nodes entered via the *admin* interface. He then feeds this inventory file to the Fabric scripts and runs the appropriate command in a shell session in order to configure a given machine with a given role. Given that this work is a prototype, not all of this process is automated and in fact the configuration session is interactive; the administrator must attend to it. Future work in this component could consist on automating this process of commissioning new computational resources. Ideally, the administrator inserts a new node in the *admin* panel and the configuration proceeds autonomously without further intervention. This feature of managing the inventory of the platform was not implemented.

There is yet another, final, subcomponent for administration: the metrics collector. Every node is configured with collectd[68]. This software collects and transmits performance metrics from the local node to the metrics node. The metrics node receives this performance data and graphs it in a web interface using Graphite[69]. This allows the administrator to know at all times the load on the nodes and informs his decisions to commission or decommission nodes as demand for the services changes. Graphite also collects data from HTTP requests made to the API nodes[70] and even database query timings. This allows for a more high-level information feed; instead of showing CPU load or memory usage Graphite will graph request throughput, delay and erroneous response status, discriminated by resource requested on the API. This was implemented by using the library *django-statsd-mozilla*[71].

The only lacking software in this stack is probably a monitoring application like Nagios[72]. It monitors nodes and alerts system administrators to potential problems with the platform. It was not implemented because monitoring the health of the nodes was not a priority and the additional work of configuring a sane environment for it and adding it to our infrastructure would not add value to this project's main goal. However, Nagios or a software with similar features would be highly desirable in a production environment.

4.12 Summary

In this chapter we identified and described the components of the platform. These components are conceptual in nature and represent the baseline from which to derive their actual implementation. We also analyzed what technologies are available, how suitable they are and, with that information in mind, chose which ones we would use. The main technological choices were Django, Cassandra, Gearman, PostgreSQL, Fabric and Ubuntu. We hinted at some implementation details, namely in the data models for the various stores, but not at the architectural connections between the components and how they are actually deployed which will be described in Chapter 5.

[68]https://collectd.org/
[69]http://graphite.readthedocs.org/en/latest/
[70]They transfer it to Graphite via UDP.
[71]https://django-statsd.readthedocs.org/en/latest/
[72]http://www.nagios.org/

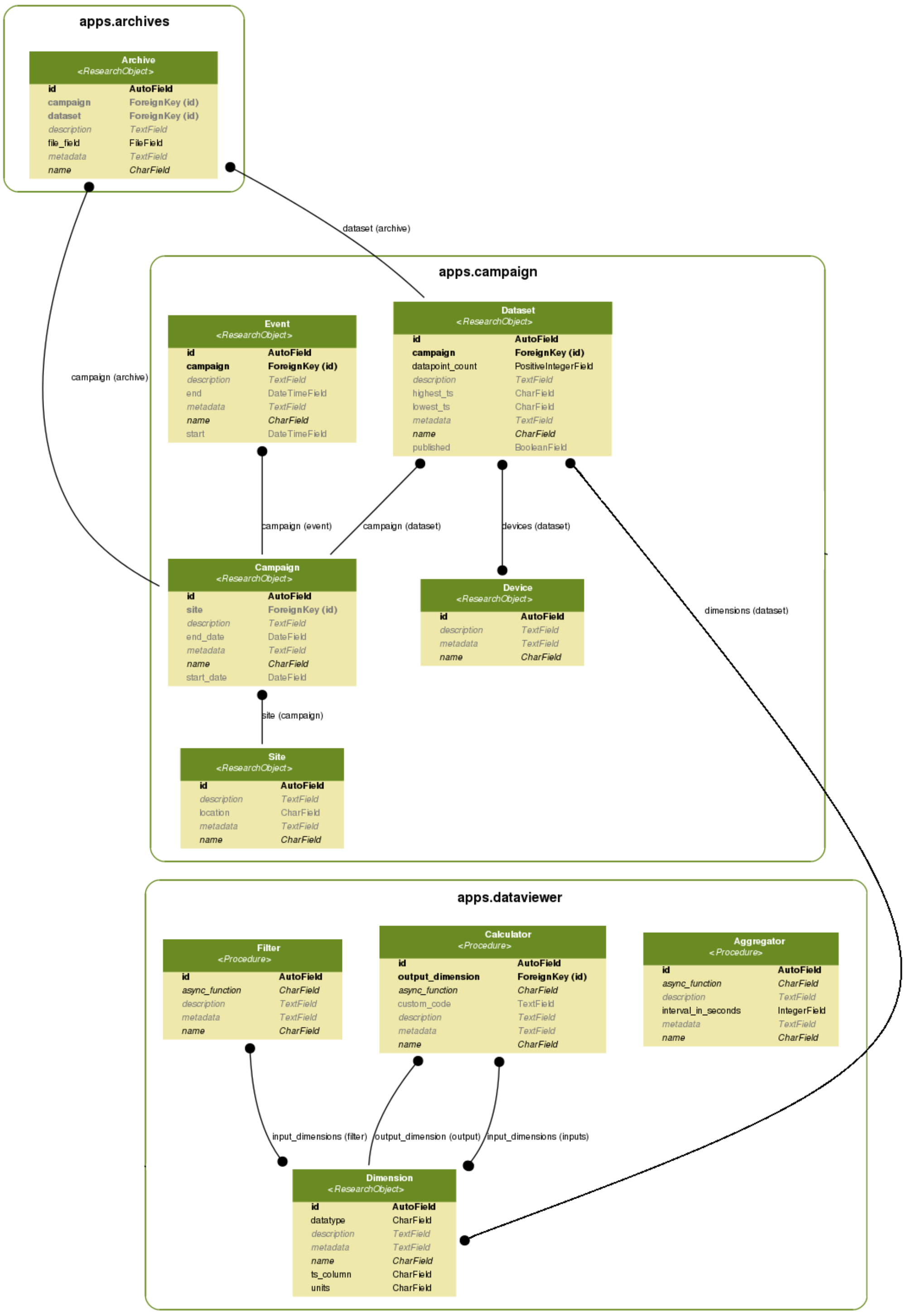


Figure 4.2: Data model for the Operational Store — First diagram

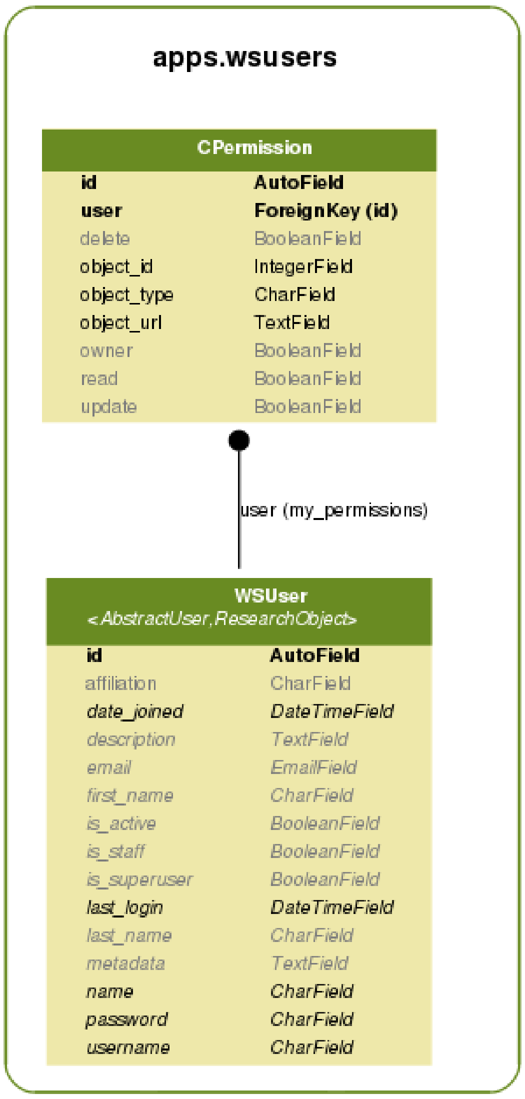


Figure 4.3: Data model for the Operational Store — Second diagram

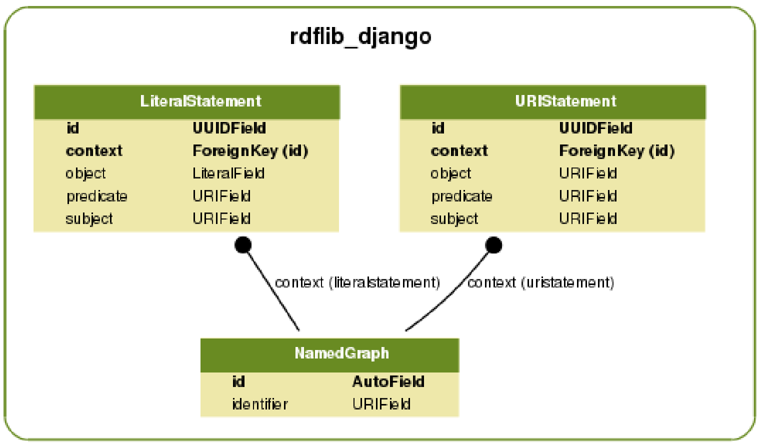


Figure 4.4: Data model for the Triple Store

Chapter 5

Implementation

In this chapter we will present the actual implementation of the components and how they translate into the roles that our nodes take. These roles are defined and configured using Fabric, as previously stated in Section 4.11.

5.1 Introduction

In our architecture one node can only have one role at a time. Implementing support for more roles in a node would have meant Fabric scripts more complex than necessary.

Every program that does not daemonize itself through package scripts (in the case of PostgreSQL or nginx, for example) is managed by *supervisord* (in the case of the workers or *gunicorn*, for example).

We start by presenting the general philosophy and comments about the implementation of the architecture. We then explain in detail how each node type was implemented. For each node type we will present a UML deployment diagram. For clarity, the communication paths between the artifacts show an arrow to indicate which artifact initiates contact. For brevity and clarity we did not show the transport layer protocol of the communication paths between the nodes. They are all implemented with TCP/IP.

5.2 General implementation remarks

We have made some important decisions regarding the platform. The first was to not modify default configurations on the components we did not implement ourselves. These default configurations come packaged with their respective applications. The only configurations we changed were the ones that allowed us to effectively deploy the architecture. While this is certainly suboptimal in terms of performance, controlling for and making experiments with these changes in place would have increased the complexity of this work to an unmanageable degree.

We did not implement any specialized caching layer. The various applications we use already implement some local caching as default behavior and, as per above, we did not modify this. There are various solutions for this use case such as memcached[1] and redis[2]. Caching expensive results in order to increase performance is a common architectural pattern, but it would have meant increased complexity beyond manageable. However, it is strongly advised this solution be studied in the future, should the need for more performance arise.

The major architectural decision we took was to configure all nodes with an instance of an appropriate load balancer for its needed services. The configuration of each of those instances is managed by Fabric. While the most common pattern for load balancing involves a central node through where all balanced traffic flows, it introduces a single point of failure (SPOF) and would probably constitute a network bottleneck in the future, when network activity increases. To combat this without having to reconfigure all the applications we opted to have them connect to *localhost* when they need external services (such as database or API access). These services are transparently contacted by the local instance of the load balancer. This introduces extra, albeit almost negligible, load locally and extra complexity in managing and synchronizing all configurations of load balancers. We posit that this extra complexity is offset by the threat of having a very real SPOF and the extra complexity of working around that SPOF with virtual IPs and other methods. We have not previously seen this pattern anywhere, whether in the literature or anecdotally.

Following the same thought process, we did not implement a load balancing mechanism between the API nodes and the *Service Application*. It is free to implement whichever strategy is most amenable to its interests. The responsibility to properly load balance between the pool of available API nodes rests on the *Service Application*.

Figure 5.1 shows the UML deployment diagram, providing a complete system view. It is meant to be informative of the deployment and communication among the nodes, which are explained in more detail in the next sections. Every node type with the exception of the Operational and Triple Stores masters may have any number of instances. These may only have one instance each on the whole platform. Note that the *Service Application* node is outside the scope of this work.

5.3 API Node

In Figure 5.2 we can see the representation of a single instance of an API node. It receives requests in its only outward-facing service, HTTP, implemented by nginx. nginx then routes the request according to its URL.

The BTSync daemon lives in the API machines and the application writes BLOBs uploaded to it to the local filesystem. The folder to which the application writes the files is mirrored to each other API machine via BTSync. In this way we achieve data replication and safety, along with expected good read-write throughput, at the cost of limiting the amount of data to the lowest disk

[1]http://memcached.org/
[2]http://redis.io/

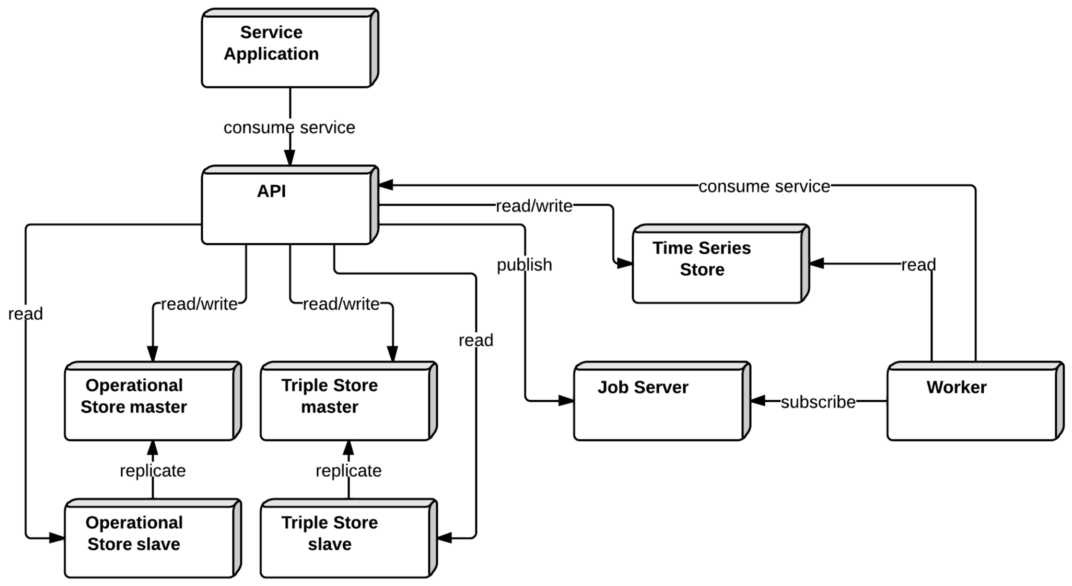


Figure 5.1: Deployment diagram

capacity among the API machines. Clearly this is suboptimal in terms of storage scalability and is likely a good topic for future work.

If an HTTP request that arrives on nginx requests a BLOB, it reads the respective file from the local file system and sends it. If that request should instead br processed by API, it reverse proxies it to Django, which is run by *gunicorn*. This is a common architectural pattern [HKM09]. If the request is the addition of a new BLOB, Django will write the uploaded file to the local file system. Otherwise it will contact *localhost* on the appropriate TCP port[3] for the external services it must consume, namely, the various Stores and the Job Servers.

If the port requested leads to the Operational or Triple Stores, *pgPool-II* will consult its configuration and route read/write requests to the master node and read requests to the slaves. This is completely transparent to Django and is reminiscent of the principle of Backing Services in 12 Factor App[4]. Likewise, if the request warrants consulting the Time Series Store, Django will contact *localhost* on Cassandra's default port. HAProxy[5] will route the connection to one of the Time Series Store nodes in a round-robin fashion. Since all Time series nodes form a peer-to-peer network and every one is capable of servicing any request, they behave much like a stateless application and the architectural pattern chosen for this case is the above. Further studies should be directed to other algorithms for loadbalancing supported by HAProxy, such as least-connection, so as to ascertain which is more suitable.

Finally, if the request requires a job to be scheduled, Django will contact HAProxy and publish

[3]Ports used are the default ports defined by the different external services.
[4]http://12factor.net/backing-services
[5]http://haproxy.1wt.eu/

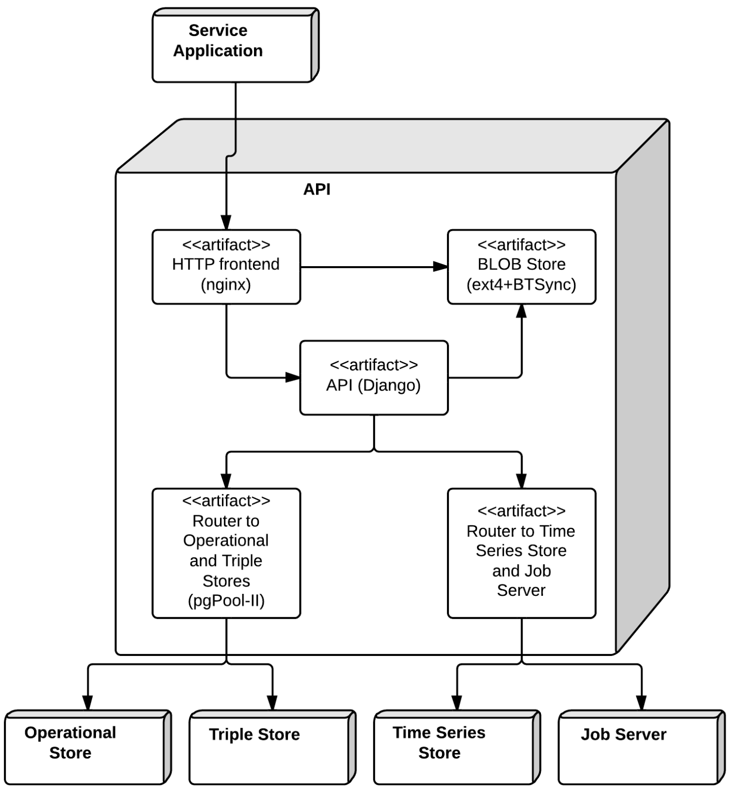

Service Application
API
<<artifact>> HTTP frontend (nginx)
<<artifact>> BLOB Store (ext4+BTSync)
<<artifact>> API (Django)
<<artifact>> Router to Operational and Triple Stores (pgPool-II)
<<artifact>> Router to Time Series Store and Job Server
Operational Store
Triple Store
Time Series Store
Job Server

Figure 5.2: API node implementation

a job. Note that job submission happens during the HTTP request-response cycle, but not its execution by the workers. Again, HAProxy loadbalances the job servers using round-robin. This means that exactly one job server now has a job to be run. Further below we will explain how the workers contact the job servers and consume jobs.

Because these nodes store no application state, servicing a node for repair or update is simply a matter of shutting it down and commissioning and configuring another one. Scaling this component is just as simple: commission and configure more nodes as needed.

5.4 Job server node

In Figure 5.3 we may see the implementation of the job servers. It simply comprises the *gearmand* daemon listening on its default port. The API nodes connect to it via HAProxy and publish jobs for future execution.

As previously stated, this component could be absorbed into either the API nodes or the worker nodes. We implemented it as a separate node type in order to give us more fine-grained control on scaling the platform. The same could be said for the proxy to PostgreSQL, *pgPool-II*, but this application pathway (connecting to the databases) is largely more utilized than job scheduling

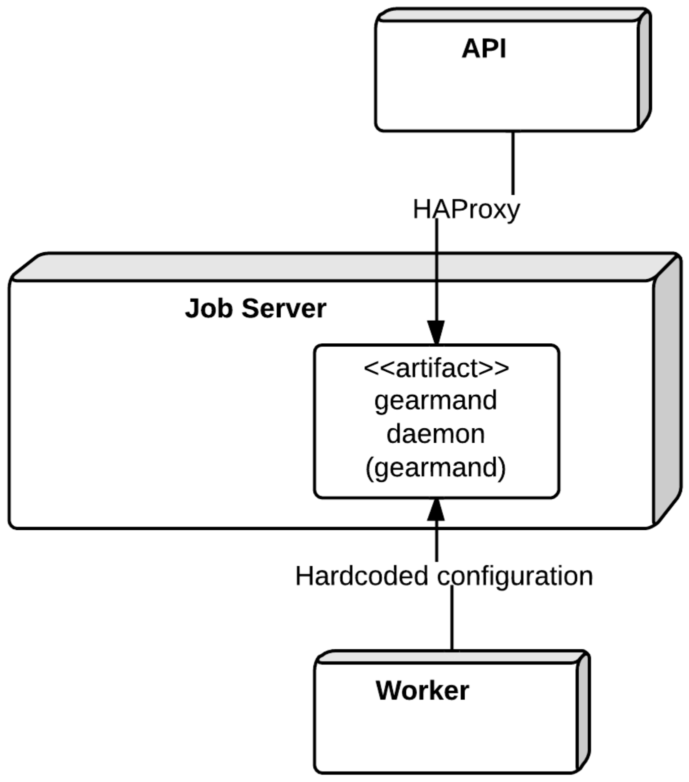


Figure 5.3: Job server node implementation

while also being more sensitive to network latency. That is to say, the user of the API would be more impacted by delays in accessing the stores than in scheduling jobs, as network delays have a larger impact on distributed web application performance than other factors [Gri13][6].

The worker nodes connect to the job servers via hardcoded configuration. This configuration is first set at spin-up time[7] and may be subsequently updated using Fabric. In this way, any worker node connects to all known job servers and therefore can consume jobs that are published on any of them. This configuration step could be avoided if the job server daemon resided on the worker nodes; they would simply consume jobs from *localhost* while the API nodes would publish jobs as usual, albeit with different configuration in its HAProxy instance. This is an interesting path for further study, as it simplifies the platform while losing some flexibility, as discussed above.

Scaling the job servers is done in the same way as the API nodes, with some additional complexity in order to update the relevant configurations in the API and worker nodes: configure another node and update the configurations. Node loss is somewhat tolerable. While the jobs on that particular node will be lost, they may be re-scheduled as the scheduling mechanisms were

[6]This reference has information on browser network latency while connecting to web applications. However, we may extrapolate that network latency inside our platform also governs response time in the same way.

[7]That is, when we are first configuring the node to join the platform.

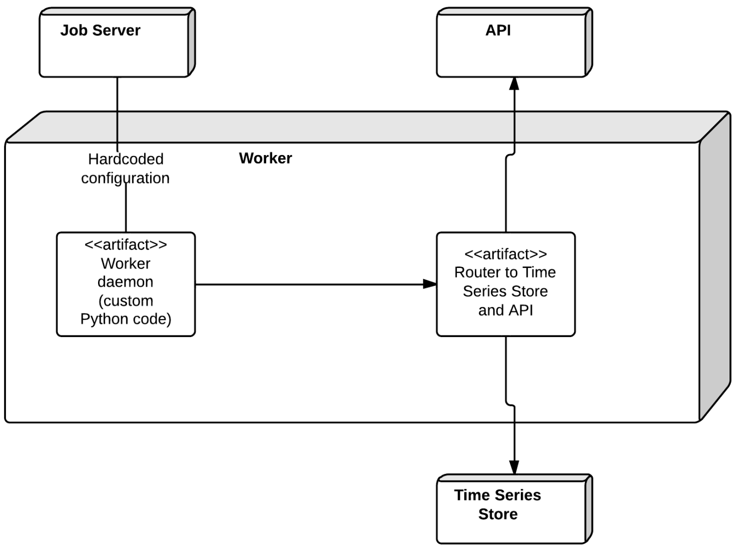

Job Server
API
Hardcoded configuration
Worker
<<artifact>> Worker daemon (custom Python code)
<<artifact>> Router to Time Series Store and API
Time Series Store

Figure 5.4: Worker node implementation

designed to create idempotent jobs. While the configuration is not updated in the API and worker nodes processing continues as usual as HAProxy and the *gearmand* client library (in use in the worker nodes) tolerate failing job servers.

5.5 Worker node

The implementation of the worker nodes is in Figure 5.4. The worker daemon uses the python-gearman[8] library in order to interact with the job servers, both as a publisher and as a consumer of jobs.

Any scheduled job starts with a *pre_schedule* job. It provides all the metadata necessary to process a dataset. The goal of *pre_schedule* is similar to that of a mapper in map-reduce: it splits the relevant dataset in manageable intervals taking into account the metadata it is given. For each interval it will schedule a new job with the dataset interval and the procedure to be applied to it. The interval is passed as a CQL string the next function will run against the Time Series Store, via HAProxy on *localhost*.

Before starting to schedule small jobs, the code will already know how many of them it will start. It contacts the API via HAProxy on *localhost* to update the job count, stored on the Operational Store. This prevents a race condition in which a small job would terminate before the number of jobs is determined[9]. *pre_schedule* then terminates.

[8]https://pythonhosted.org/gearman/index.html

[9]The API will act on the number of jobs reaching zero by scheduling the next procedure in the pipeline. This race condition would erroneously trigger that scheduling, leading workers to perform calculations on inexistent or erroneous data thereby either stalling the whole process or corrupting data.

When the worker receives a small job to be executed, it runs the CQL string on the Time Series Store to retrieve the data. It then runs the function it was passed as an argument which is the content of the attribute *async_function* in the Procedure object in the Operational Store. In other words, *pre_schedule* receives the dataset identifier and the Procedure object which is passed along to the job runner functions as applicable: *row_calculator* and *row_aggregator*, for *Calculator* or *Aggregator Procedures* respectively. The *row_** functions call the function *Procedure.async_function* with the data returned from the CQL statement. This function is defined in other modules of the worker code and may be easily extended to include more functions. This provides the basic framework for adding Procedures to the platform, be them user created or developed by the platform maintainers.

The contract for extension is simple. *Calculators* take one input variable, a *dict*[10], with the keys being the *Calculator.input_dimensions.ts_column*. The *dict* represents a single row. This means the columns and respective values of the *input_dimensions* of the *Calculator* are available in its scope. The return value expected is a single value of the type *Calculator.output_dimension.ts_column*. The code is free to do whichever else is necessary to compute the result.

Aggregators take a *list* of *dict*s (in effect, a *list* of rows) and a *list* of columns (strings) which are the columns of the row that are writeable. The function must return a single *dict* (row) whose keys are the *list* of columns and the values are the ones computed by the function.

User created *Procedures* should also follow this contract as it provides a clean and straightforward way to implement the computations. In the case of user created *Procedures*, the custom code would need to be serialised from the API/Operational Store to the workers. That would require the workers to properly isolate code execution and prevent abusive behavior, which are the main reasons for our declining in implementing this feature. We instead propose, for the time being, that any custom procedures be reviewed and "blessed" by the platform maintainers so as to mitigate any issues. This certainly impedes researcher freedom in using the platform. We propose that further ahead in development a safe and secure environment be devised to contain custom code in order to preserve platform integrity and prevent abusive utilization.

In any case, when a new *Procedure* is to be added, it must be inserted in the Operational Store via the API. Then, the respective code must be added to the worker repository. Finally, the code must be updated in all worker nodes using the Fabric scripts.

When the small job ends, the worker will contact the API once announcing its completion and again to insert the resulting datapoint in the Time Series Store. In an effort to keeping code DRY[11] (Don't Repeat Yourself), we chose to reuse the API and not re-implementing the datapoint insertion logic. Later on, after development, we noticed another opportunity to follow this principle which, combined with the previous tactic, allows for simpler code and architecture on the workers. Instead of performing CQL statements directly on the Time Series Store to retrieve the dataset interval the *Procedure* needs, we could request the interval from the API. This would

[10] A *dict* is a Python data structure. It is an associative array.
[11] `http://c2.com/cgi/wiki?DontRepeatYourself`

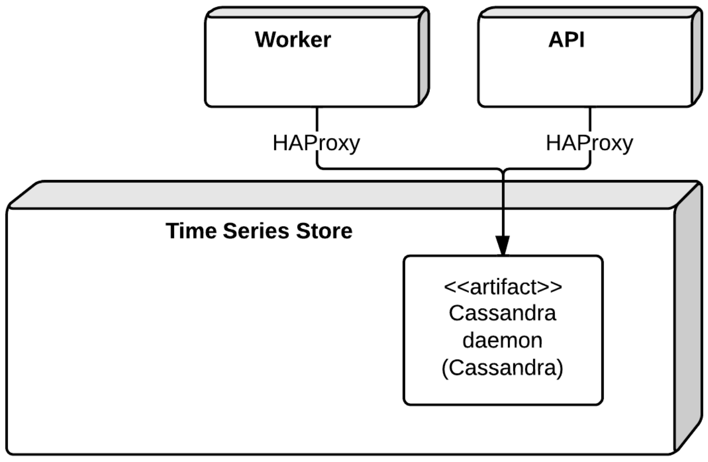


Figure 5.5: Time Series Store node implementation

enable the worker to completely drop the dependency on the Cassandra driver, *cassandra-driver*[12] and the HAProxy instance on the workers would no longer need to track information regarding the Time Series Store. The code would need to be adapted to do this, with little additional effort from the developers. It is strongly suggested this avenue be pursued.

Scaling this component is just as simple as the API nodes. Since they don't store application state, to increase capacity the administrator must commission and configure another worker node. No other configuration elsewhere needs to be updated. Node failure is just as tolerable as with the job servers: if a node fails the current job will be lost. However, the job servers track job reservations and completion in a way that if they notice a worker failing with a job halfway through they will return the job to the queue to be run by another live worker.

5.6 Time Series Stores node

Figure 5.5 represents the implementation of a Time Series Store node. The nodes form a peer-to-peer cluster in which every node is capable of servicing any read or write request. This peer-to-peer exchange is implemented by Cassandra with no intervention on our part. We use the default configuration[13], but further research in this area should be carried out.

When the API or worker nodes wish to contact the store they connect to HAProxy on their *localhost* which directs them, in a round-robin fashion, to one of the Time series nodes. Trying to find the particular node in which the desired data actually lives would defeat the purpose of using a peer-to-peer architecture. Cassandra also takes care of data partitioning and replication.

[12]http://datastax.github.io/python-driver/index.html

[13]http://www.datastax.com/documentation/cassandra/2.0/cassandra/architecture/architectureSnitchesAbout_c.html

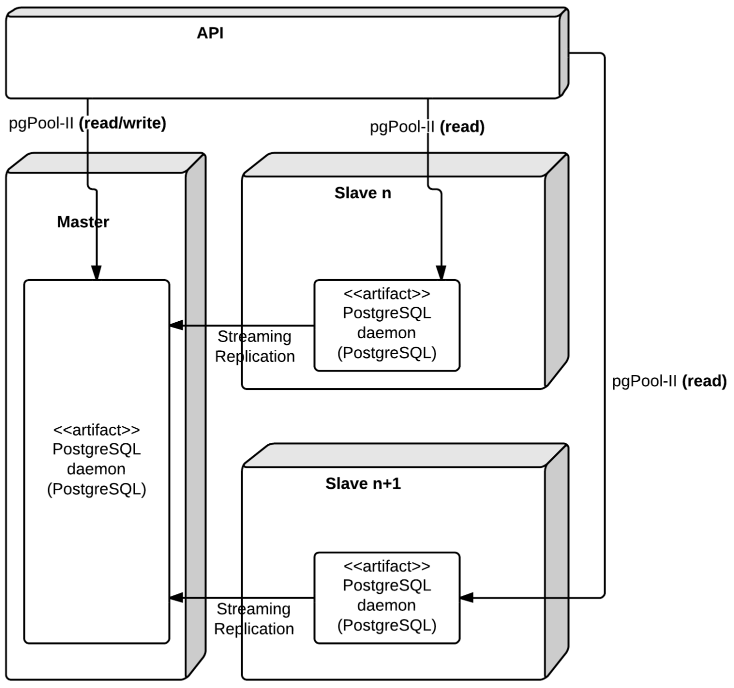

API
pgPool-II **(read/write)**
pgPool-II **(read)**
Master
<<artifact>> PostgreSQL daemon (PostgreSQL)
Slave n
<<artifact>> PostgreSQL daemon (PostgreSQL)
Streaming Replication
pgPool-II **(read)**
Slave n+1
<<artifact>> PostgreSQL daemon (PostgreSQL)
Streaming Replication

Figure 5.6: Operational and Triple Stores node implementation

The nodes know about each other and in fact exchange various bits of information via a gossip protocol[14].

5.7 Operational and Triple Stores nodes

Since the implementation of the Operational and Triple Stores is identical, the single Figure 5.6 shows how these components are implemented. Note that while the figure only shows two slave nodes, there may be any number of slave nodes running. The nodes may all be contacted by *pgPool-II* running on the API nodes, but while read and write operations may be directed to the master node, only read operations may be directed to the slave nodes. The configuration in *pgPool-II* allows it do determine which nodes perform which roles and the SQL generated by Django[15] hints *pgPool-II* to what kind of operation (read or write) it refers to. This implements the architecture defined in Section 4.7. In this way, all nodes mirror each other's data and the master node is the authoritative source of data for all the slaves, making Consistency a more manageable problem since all nodes will eventually converge.

[14]http://www.datastax.com/documentation/cassandra/2.0/cassandra/architecture/architectureGossipAbout_c.html

[15]Using its accompanying Object-Relational Mapper (ORM). We could also hand-generate our own SQL should we so desire.

The mechanism for mirroring is Streaming Replication, implemented by PostgreSQL itself. We simply direct the slave nodes at spin-up time to replicate whichever data lives in the master.

Should a slave node experience failure of any kind it may be safely decommissioned and replaced. Scaling slave nodes (and consequently read capacity for its particular store) is as simple as following the same pattern as in the API nodes. Scaling master nodes is, at best, impractical. It is however a deliberate decision, as explained in Section 4.7

Should a master server fail for any reason, the data in its particular store would become read-only. *pgPool-II* has mechanisms for promoting a slave node to a master node, but these have not been implemented or tested in our work. In practice, this is a SPOF for both stores and a mechanism for slave promotion or other repair mechanisms must be implemented. Further work and testing should be directed into this issue.

As a suggestion, we would propose an architectural change in which the nodes do not form a fanout structure but one similar to a linked list in which only slave number one connects to the master, slave number two connects to slave number one, slave three to slave two and so on. This is referred to in the PostgreSQL documentation as Cascading Replication[16]. This is supported by the current technological stack, but requires more coordination effort because when a slave fails the chain of Streaming Replication must be repaired. Failure of the master requires the first slave in the chain to become the new master which is somewhat simpler to accomplish than with the current solution.

5.8 Summary

In this chapter we analyzed how each node is actually implemented and how the nodes interconnect. We started by introducing an important high-level directive about the interconnection of nodes and service discovery, along with other limitations in scope we imposed. We illustrate the various network flows through the nodes the data may take during its life-cycle in the platform. We also thoroughly outline how the Workers process data and what the contract for extending functionalities is. This is relevant in the future when maintainers of the platform introduce more *Procedures* to meet users' demands and expectations. This implementation of the architecture we outlined in Chapter 4 allows us to actually configure the hardware resources so that we may perform tests on the platform. In Chapter 6 we determine how these tests will be carried out.

[16] http://www.postgresql.org/docs/9.2/static/warm-standby.html#CASCADING-REPLICATION

Chapter 6

Methodology

In this chapter we will define the methodology we used while undertaking this project. First we will determine how we generated the load on the platform by identifying the most representative load profiles we expect to see in production environments. We describe an outline of how we intend to introduce load and scale the platform as we perform the tests. Lastly, we enumerate and justify the set of metrics we plan to collect and analyze, in order to assess the performance and behavior of the platform.

6.1 Load generation

Because we did not have a sample or pattern of production traffic on the platform, we had to create our own patterns and behaviors. The behaviors we created were based on an educated guess about how the platform might be utilized in the future, how many components are used with each behavior and which components warrant most attention.

All behaviors mimic the actions of an actor in [LG13] as if it were interacting with the *Service Application*, which in turn interacts with the internal API. The actual implementation of all of them contact the API directly instead of contacting the *Service Application* as the *Service Application* does not exist in a form that can take advantage of the API.

While the available data from the Winds@FEUP project could be inserted into the platform, we could not implement procedures on it given we are not specialists in that particular field of study. Therefore, both the data inserted and the procedures applied to it are artificial in nature and are further explained in the respective behavior.

The four behaviors, Initializer, Data Provider, Triple and Researcher are explained below. All behaviours operate indefinitely until stopped, with the exception of the Initializer.

6.1.1 Initializer

The goal of this behavior is simply to setup the initial resources on the platform so the other behaviors may conduct their actions. It creates a Site, a Campaign and three Dimensions: Centigrade, Fahrenheit and Kelvin. These Dimensions mimic both the data collected by field sensors and data derived from procedures applied to the particular dataset in which they feature.

It also creates three Procedures: one Aggregator and two Calculators. These were implemented in the Worker nodes prior to their creation in the Operational Store, as determined by the protocol for adding Procedures to the platform.

The Aggregator computes arithmetic averages of 20 seconds. That is to say, the procedure takes a 20-second interval from the dataset and computes the average of all Dimensions that the Dataset is related to (in the Operational Store). This is the procedure that most illustrates production usage, as informally reported by specialists of the field.

One Calculator converts Centigrade to Fahrenheit. The second Calculator converts Fahrenheit to Kelvin. While these are conceptually simple, their implementation features a key aspect: since they consist of arithmetic operations, the workload is skewed to being CPU intensive. This is a pattern we suspect to see in production scenarios and these examples, while simple, approximate those scenarios.

This behavior need only run a single time when the platform is first deployed.

6.1.2 Data Provider

The Data Provider behavior mimics the access pattern of the Data Provider actor.

The agent connects to the API and creates two datasets, one raw dataset and one derived dataset. It will then proceed to insert datapoints in the raw dataset.

The timestamp of the datapoints is generated automatically. The agent takes the current time as a seed and, for each successive datapoint, increments the time by an integer value randomly chosen between 2 and 5 seconds. This discrepancy in collection interval attempts to mimic the insertion of data that has not gone through quality control on the field. The datapoints simulate the data collected by a thermometer. The temperatures are collected in Centigrade. The dataset is considered completely uploaded when the cut-off time is reached. This time varies between 90 seconds and 111 days, depending on the load profile chosen for each test run.

Then, the agent schedules the procedures in Section 6.1.1 on the raw dataset, whose results are to be stored on the derived dataset. It first schedules the Aggregator, then the Calculator from Centigrade to Fahrenheit, then the Calculator from Fahrenheit to Kelvin. These procedures must be applied in that specific order. Note that the last Calculator depends on the output values of the first Calculator. In this way, we test the platform's ability to properly schedule Procedures.

This behavior exercises the API, Operational Store (read/write), Time Series Store (read-/write), Job server and Worker nodes.

6.1.3 Tripler

The Tripler behavior attempts to recreate the usage pattern of an automated agent or a human researcher when inserting and searching for linked resources (as introduced in Section 2.4.2). These links are implemented via a triple data structure, stored in the Triple Store. The triples follow a subject-predicate-object structure in which every one of those attributes is an URI[1].

The agent selects three resources at random from the pool exposed by the API. It determines one to be the subject, another to be the predicate and the final to be the object. The predicate is an important part of the structure as it add meaning to the whole. In real situations, the predicate would probably be an URI/URL to a resource (a *ResearchObject*) that describes the predicate[2] so as to aid automated agents in searching the triples for relevant information. In this case, we simply insert a predicate with no meaning into the triple because there is no need (or way) to validate that behavior in this work.

After inserting the triple, the agent will select one of its elements and search the API for all the triples that contain that element. For example: the agent randomly selects the subject element. It will search the API for all triples in which the URI on the subject element is present as the subject. Then it will do a second search where the URI is present as the predicate and finally where the URI is present as the property. This simulates high read to write ratio pattern expected on the Triple Store.

This behavior exercises the API, the Operational Store (read) and Triple Store (read/write) nodes.

6.1.4 Researcher

The Researcher behavior attempts to mirror the behavior a human researcher using the platform.

It is the most simple behavior. The agent will contact the API and request a list of datasets. It will then select one dataset at random and select a subset of it, based on its time interval[3]. It will then request the subset of the dataset, with its associated datapoints.

While the Researcher actor is allowed to schedule procedures on datasets, this task is already carried out by the Data Provider behavior. Adding this action to this behavior would be unnecessary and impractical.

This behavior exercises the API and Time Series Store (read) nodes.

6.2 Validation

In this section we will discuss how we plan to validate the work performed, in essence how we measure the degree of success of it.

[1]Linked data principles allow for the object to be a literal value instead of an URI. We did not consider this course of action while designing the behavior as it adds little value to the experiment.

[2]Using RDF (Resource Description Framework), for example.

[3]Note that the list of datasets only includes its metadata and excludes their actual datapoints. Only when requesting a specific dataset are the datapoints also returned.

First, the final prototype must conform to the specifications stated in Chapter 3. Chapter 5 details the implementation of the platform which conforms to the specification.

The second, final and most important measure of success is in measuring how the platform behaves and increases its capacity. We will base our analysis of its scalability on the metrics defined and collected during the load tests. We have two available variables to modify: data size and compute instances. We will first deploy the platform with minimal storage and processing capacity, that is, using less computing instances per component. Then, we will perform load tests and collect their measurements. We will then increase the expected capacity of the platform by increasing the amount of computing instances available to use. We will proceed with evaluation and capacity increase activities until the performance gains are too low to justify the increase in resource (hardware) utilization. We cannot beforehand determine when this will occur.

When the system no longer benefits from increased resource usage, one of two things has happened: either there is too little data flowing through the platform and resources are under utilized or the system can no longer scale its capacity to account for increased demand. At this point, we will artificially, or otherwise, increase the load on the system in order to test the first hypothesis. If the resulting metrics collected reveal that computing resources are under utilized, we should see that performance indicators are not much worse than before additional data was introduced. When increasing data load again worsens performance measurements, it means that the previous configuration is close to optimal in terms of resource utilization and data intensity ratio. This is an important finding and will give insights on implementing auto-scaling features of the platform in order to optimize resource utilization and operation costs. This feature is outside the scope of this work and is a good topic to be assessed in future studies. Ideally we would not encounter the second hypothesis.

6.3 Relevant metrics

In this section we will list and explain the metrics that provide the most insight on the performance of the platform. Actual figures of these metrics may be consulted throughout Chapter 7. Collecting metrics and performance information also introduces load on the various nodes, but this is not really a problem. It is as important in production as is during testing in order to assess system health and behavior, so this additional load is not at all artificial and therefore does not skew results in any way.

Before we continue, we must explain the concept of the 90th percentile mean time, which is a measurement we mention below about HTTP request latency. The 90th percentile mean is calculated as follows: for all timing measurements, the higher 90% of them are taken and averaged. This means that 90% of the requests take less than this value. This value represents the average time of the worst 90% performers. It is more relevant and informative than the mean latency [Gol11].

The metrics we paid special attention to are the following:

Request throughput per status code — this metric shows the number of HTTP requests per second that were served by the API nodes. This includes requests made both by the *Service Application* and by the workers. We discriminate on the status code returned in order to have a more specific idea of what the platform is doing. In our case, we measure status codes 500 (server error), 200 (ok) and 201 (created). These are the most representative of the statuses: 500 indicates errors which should be attended to, 200 indicates a read operation was performed and 201 indicates a write was performed. It is measured in requests per second.

Request throughput — this metric shows the total number of HTTP requests per second the API is serving. It is an aggregate measurement of the previous metric and corresponds to total platform throughput. It is measured in requests per second.

General latency — this metric shows the aggregate latency of read and write operations on the API. It gives a general picture of the quality of service as measured by the time requests take to be satisfied. It is measured in milliseconds.

Dataset read list-detail latency — this metric shows the 90th percentile mean time a read request on the *Dataset* resource takes to complete. This compares the requests made to the endpoint for the list of *Datasets* as well as the endpoint for a singular dataset in which case the list of its datapoints is also fetched. This measure provides insight into the read capacity of the Operational Store and Time Series Store. Write latencies for *Dataset* write requests are negligible and not representative of expected load; they happen sporadically when creating a new *Dataset* for raw or derived data and consist of a single insertion in the Operational Store. As such, this last measure was left out of the figure presented in this document. It is measured in milliseconds.

Datapoint mean write latency — this metric shows the 90th percentile mean time a write request of a *Datapoint* takes to complete. Since *Datapoints* are not readable on their own, measuring datapoint reads is implemented by the previous metric. This metric provides insight into the write capacity of the Time Series Store. Note that both the *Service Application* and the workers insert datapoints via the API. It is measured in milliseconds.

Triple mean read-write latency — this metric shows the 90th percentile mean time that read and write requests take to complete against the Triple Store. This measures the performance of the Triple Store component. It is measured in milliseconds.

Outstanding tasks — this metric shows the number of tasks that are scheduled on the platform but are not completed yet. The trend of the line indicates whether or not more tasks are being added than completed, when the line rises or falls, respectively. This allows us to measure whether the current amount of workers can keep up with the tasks scheduled. A flat line indicates node allocation is close to optimal. A rising line indicates workers cannot keep up with demand and a falling line indicates Worker nodes are over-provisioned relative to task demand. It is measured in number of tasks.

We also collected lower-level metrics pertaining to hardware utilization of the different nodes. This informs our decisions on how to further scale the platform, along with the metrics above. It also provides us with insight on how the different types of nodes should be distributed over the available physical resources and how to tailor the physical machines' hardware specifications to the needs of the nodes. This is a topic for further study. We collected metrics from the API nodes, Operational, Triple and Time Series Stores, and the Worker nodes. All the metrics are identical for all the nodes, as follows:

CPU utilization — this measures the utilization of CPU in percentage of user, wait and system time. User time indicates how the processes in user space use CPU resources. We assume this metric is largely governed by the software we implemented in each node, as a bare-bones, freshly installed system has negligible CPU activity. Wait time indicates how long the CPU has been waiting for disk input/output (IO). This metric is relevant in order to assess hardware utilization and also helps diagnose application performance. System time is the time CPU spent in kernel space, performing system calls on behalf of other processes. It is measured in jiffies[4].

Memory occupation — this measures the amount of memory the node is consuming at any given time. It is the total memory occupied by all the processes in the node. We assume that the processes we started ourselves which are not part of the base operating system are the ones that are most visible in terms of memory consumption. It is measured in bytes.

Disk utilization — this measures the amount of data read from and written to the disk. This is important to know for all nodes, especially for the all the Stores. These are the nodes that will most likely write data to disk when doing an insertion or reading data from disk when responding to a lookup. While this measures IO throughput in bytes (and multiples thereof), CPU wait time measures IO latency. It is measured in bytes per second.

Network utilization — this measures the amount of data read from and written to the network. It compares data sent and received from this node through the network interface. The *loopback* interface was not considered because it has no actual effect on network traffic. This measures network throughput. It is measured in bytes per second.

Mid term load average — this measures the mid term trend load of the system. System load is a measure of the number of processes that await CPU resources for execution and provides a rough view of the general utilization of the system. As a rule-of-thumb, the optimal load average for a 1-CPU machine is 1.0. The nodes measure short, mid and long term loads, for 1, 5 and 15 minute averages, respectively. We chose the mid term, 5 minute average because it better illustrates the load during the relatively short test runs we made. It is dimensionless.

[4] http://man7.org/linux/man-pages/man7/time.7.html

These are the metrics we found that were more important and characteristic of the platform. Other metrics were not analyzed here, but are available elsewhere for offline analysis.

The first set of metrics were collected on the API nodes and sent to the machine responsible for collecting metrics with the exception of Outstanding tasks. Due to issues collecting this metric[5], we opted instead to implement a custom solution. We implemented an API endpoint that returns the Outstanding tasks on the platform. We sample it in 1-minute intervals. This introduces some load on the platform, specifically the API and Operational Store, but this metric is important enough, as explained above, that it warrants measurement and analysis.

The second set of metrics was collected on each individual node using *collectd* and routed to the metric collector. No special intervention from our part was performed or needed, apart from composing the graphs.

We call the first set of metrics "high level" metrics, to help differentiate from the second set of lower-level metrics or "hardware metrics".

The final set of measurements are simple counts of objects in the stores. Each of them gives a general magnitude of the data stored and processed:

Number of datapoints — this measures the total amount of data resident in the Time Series Store.

Number of tasks — this measures the number of tasks completed. Task metadata is stored in the Operational Store and processed by the Worker nodes.

Number of datasets — this measures the number of datasets inserted into the Operational Store.

Number of triples — this measures the number of triples inserted into the Triple Store.

6.4 Summary

In this chapter we defined our methodology for this work, how we will direct our testing of the platform. The client behaviors were described in some detail and, importantly, we have defined the way in which we will measure the success of the work. The method we prescribe allows us predict whether the implementation was successful and the architecture able to scale when facing increased demand. We have also established how we will determine cost-effectiveness and feasibility of the system. Lastly, we identified which measurements warrant study and why they do so. Analyzing and reflecting on these is what will allow us to understand the behavior of the platform.

[5]Specifically, Graphite was having trouble graphing gauges. We suspect the *bucky* daemon is erroneously reporting this datatype, but cannot confirm this. For more information on gauges consult: http://statsd.readthedocs.org/en/latest/types.html

Chapter 7

Discussion of results

In this chapter we will discuss the results from the data gathered during the course of this work. The process we followed for gathering and interpreting the data is the one outlined in Chapter 6. We start by deploying the platform on our physical infrastructure. Then we perform three test runs with different platform configurations and load profiles. At the end of each test run we perform a retrospective of it and summarize what knowledge we have gained throughout. At the end of this chapter we draw conclusions from the individual retrospectives, as well as from the whole of the experiment.

7.1 Initial setup and experiments

In this section we introduce the initial deployment of the platform and its base hardware and software configuration. We also discuss some issues that we encountered during initial testing that was made to ascertain the correct working of the platform. More issues were found than the ones reported here, but these are of particular importance because they affect the architecture and implementation.

The nodes were set up in the FEUP grid cluster. All the logical nodes are identical and reside in a single physical machine, managed by the KVM[1] hypervisor. The physical machine has the following hardware specification: 2x Intel E5-2450 CPUs[2] for 16 cores in total, 68GB of DDR3 RAM clocked at 1600MHz and 1x Seagate Constellation ES, model number ST500NM0011 500GB SATA HDD for local storage. Each virtual node has 1 CPU, 3GB of RAM, 10GB of local disk storage and 1GB swap. The operating system of every node is Ubuntu Linux server edition 12.04.4 LTS. The Linux kernel is version 3.11.0-15-generic, built for the x86_64/amd64 CPU architecture.

[1]http://www.linux-kvm.org/

[2]http://ark.intel.com/products/64611/Intel-Xeon-Processor-E5-2450-20M-Cache-2_10-GHz-8_00-GTs-Intel-QPI

It is a simple bare-bones disk image with no modifications made to the one present in the Ubuntu archives[3]. Further configuration of nodes is made using Fabric, as explained in Section 4.11.

Node usage is as follows:

- 1 command node from which to orchestrate the platform via Fabric
- 1 metric node to collect usage statistics both from the virtualized hardware and from the API
- 1 Operational Store master node
- 1 Triple Store master node
- 1 Time Series Store node
- 1 Job server node
- 1 API node
- 1 Worker node

In addition, we setup 1 client node that runs the behaviors defined in Section 6.1. This provides us with the bare minimum to have a working platform.

In our initial experiments while introducing load, we faced a memory leak in both the API and Worker nodes. We traced the problem to the usage of the Cassandra driver, but we could not determine whether it was caused by a real flaw in the driver or by our misuse of it. We suspect it was misuse, as the memory leaks reported by the development and user community occur in circumstances that are different from ours. We implemented measures to mitigate the memory leak by regularly recycling the offending processes, which seems to be effective, if crude.

Another minor problem surfaced in the Worker nodes. When contacting the API, the nodes instead contact HAProxy on *localhost*. Because the technological stack we used while developing is slightly different than the one used during testing[4], HAProxy was erroneously reporting the API nodes as being down[5]. The fastest way of correcting the issue was to use nginx as an HTTP proxy instead of HAProxy. Deploying a different version of HAProxy would have required more effort for the same value. This technological change can be justified further if the suggestion made at the end of Section 5.5 is followed because, in that case, no complexity is added while maintaining the same set of features. As it is implemented now, HAProxy routes requests to the Time Series Store and nginx routes requests to the API.

Yet another issue with the interaction between the API and the Workers was discovered. Intermittently, some scheduled jobs would not be terminated and the pipeline of a set of chained procedures would stall indefinitely. We attribute this issue to an implementation bug on the API.

[3] `http://old-releases.ubuntu.com/releases/12.04.1/`

[4] We used Django's development server while developing and *gunicorn* while testing.

[5] Specifically, the API nodes responded to health checks with chunked encoding (with the header *Transfer-encoding: chunked*), which the version of HAProxy we used does not support.

When a new datapoint is inserted, the algorithm on the API updates the time interval of the respective dataset. Sometimes, the interval is updated erroneously which means the Worker nodes read wrong timestamp metadata, crucial for its scheduling algorithm. Should the time interval be updated to correct values and the pipeline restarted again, everything works as expected. Attempts to correct the bug have proven fruitless[6]. The suggested workaround is to require the Data Provider/*Service Application* to insert the time interval of the dataset at the time of creation and update it, if/when needed.

7.2 First test run

This section provides an account of the first test run, done as if the system was in a production environment.

The platform setup was the one described in Section 7.1. We setup one client node running two instances of Data Provider, two instances of Tripler and four instances of Researcher, as characterized in Section 6.1. The client node also runs the task throughput poller in order to measure task completion rate. The Data Providers insert relatively short datasets. They are in the order of 30 datapoints each, corresponding to approximately 90 seconds worth of data[7].

We do not present data for the Job server as it is uneventful: resource utilization is minimal, further reinforcing the notion that this component could be absorbed into another one.

The first run was started at 16:45 on 2014/05/25 and ended at 19:50 of the same day. At 19:12 we stopped the clients from introducing more load to the system. Only the worker nodes continued consuming the API from that point onwards until 19:50, at which time they finished processing (almost) all scheduled tasks.

Figure 7.1 shows the high level metrics for the first run. Going left to right from the top left graphic, we see that read operations dominated the throughput of the platform for all of the test run, as evidenced by the number of responses with status code 200. Write operations, mainly dominated by datapoint insertion, came second with lower throughput. This is expected as the load introduced is fairly read-heavy. The residual status 500 (server error) responses are related to triple insertion and worker implementation bugs. When the Tripler behavior randomly attempts to insert a triple that is already present in the database, the API will throw an exception, hence the status code. Perhaps it would be better if the API returned status 202 (accepted), because while no data has changed, the data requested for insertion is present nonetheless.

Postmortem analysis of worker logs revealed they sporadically[8] throw exceptions when trying to connect to Cassandra. We suspect this is due to the limits on concurrent reads/writes set in its configuration file. This is likely a good configuration directive to tweak for further analysis. At the 19:12 mark we see a continuation of the above pattern, albeit with higher request throughput. This exposes the behavior of the workers when no other load is introduced. They insert datapoints

[6]Compounded by the fact that the Worker nodes also insert datapoints, not just the *Service Application*.
[7]As per Section 6.1.2, the Data Provider behaviour simulates sampling data with values between 2 and 5 seconds.
[8]This happened 2 times throughout the test run.

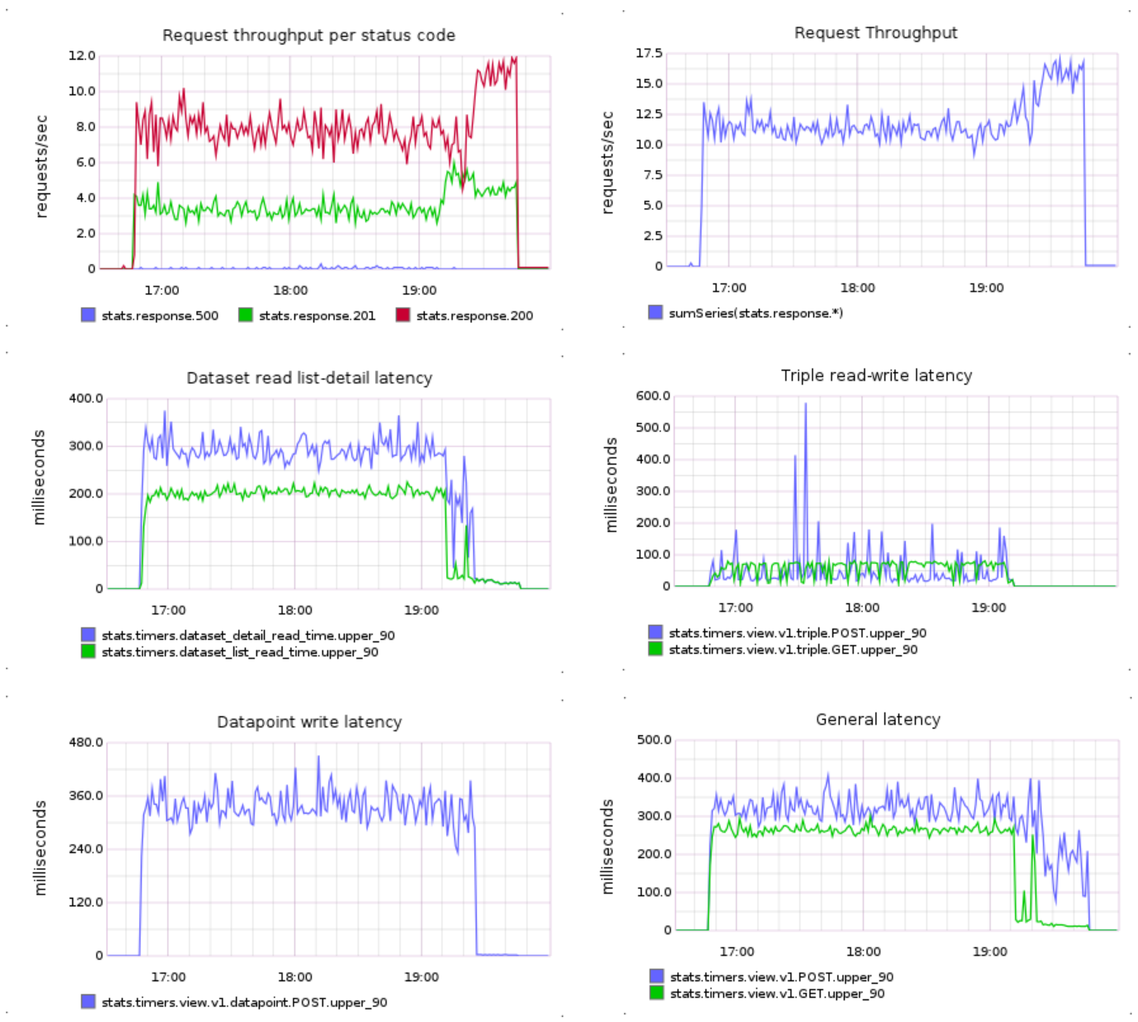


Figure 7.1: High level metrics — first run

along with job updates and query the API for metadata. Since the API node is only used by the workers, the resources of the entire platform are at the disposal of the workers.

The next graph is simpler. General throughput kept relatively constant at about 11 per second throughout the first part of the run until 19:12, at which point it climbed to 16. We were surprised by such a low number of requests per second, even though client logs are consistent with this number. We attribute this fact to CPU saturation in the API nodes for the first part, as explained further below. After 19:12, request throughput climbed due to the fact that the workers make many relatively lightweight read requests, mainly for metadata, while continuing to make datapoint insertions.

The next graphic compares read latencies of a list of datasets and the detail of a particular dataset. List read latency (green) is expectedly lower than detail read latency (blue) since list reads only return metadata from the Operational Store and detail reads return the metadata for a single dataset along with its respective datapoints from the Time Series Store. The existence

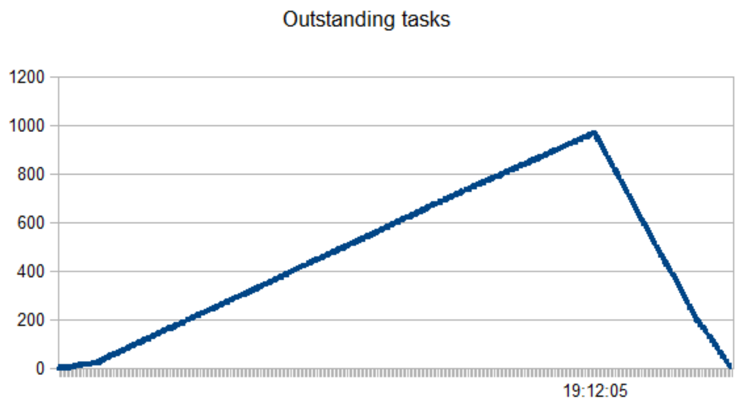


Figure 7.2: Outstanding tasks — first run

and sharp decline in detail latency at 19:12 evidence an implementation detail of the workers: they request the detail of datasets during normal operation, but provide a URL argument so that the API does not fetch the associated datapoints. They are unused by the workers in the current implementation and constitute additional pressure on the API and Time series nodes and so the workers specifically ask for them not to be fetched; this is a reasonable optimization at this point.

The next graphic shows the latency of reads (green) and writes (blue) of the triples. Reads and writes show a relatively low value, when compared to other latency values. The exceptionally high spikes at about 17:30 are unaccounted for and probably result from random disk contention in the physical host, as evidenced by an unusually CPU wait time for the Triple Store node, further analyzed below. Such a low latency is attributed to the indexes that *rdflib-django* creates on the *URIStatement* table.

The next graphic is relatively straightforward. It measures the latency of datapoint insertion. It is relatively constant throughout the whole run, even after 19:12, when the load on the Time Series Store is caused only by the activity of the workers. This is expected behavior, as the bottleneck of performance now shifts from the API nodes to the workers and, in a smaller amount, the Time Series Store.

The last graphic shows the aggregate latency for all the requests on the API, discriminated by GET/read requests (green) and POST/write requests (blue). As expected, read latency is lower than write latency, with read latency dropping significantly after 19:12 when the workers make many simple read requests that take very little time to complete.

Figure 7.2 is markedly different from the others and reflects the different mechanism by which data was collected. Nevertheless, it is informative of the general performance of the workers. It shows the tasks that were requested on the platform, but were not completed yet. The initial climbing shows that the workers could not keep up with job submission, as tasks are added at a faster rate than they are completed. The situation reverses at 19:12 where no new tasks are

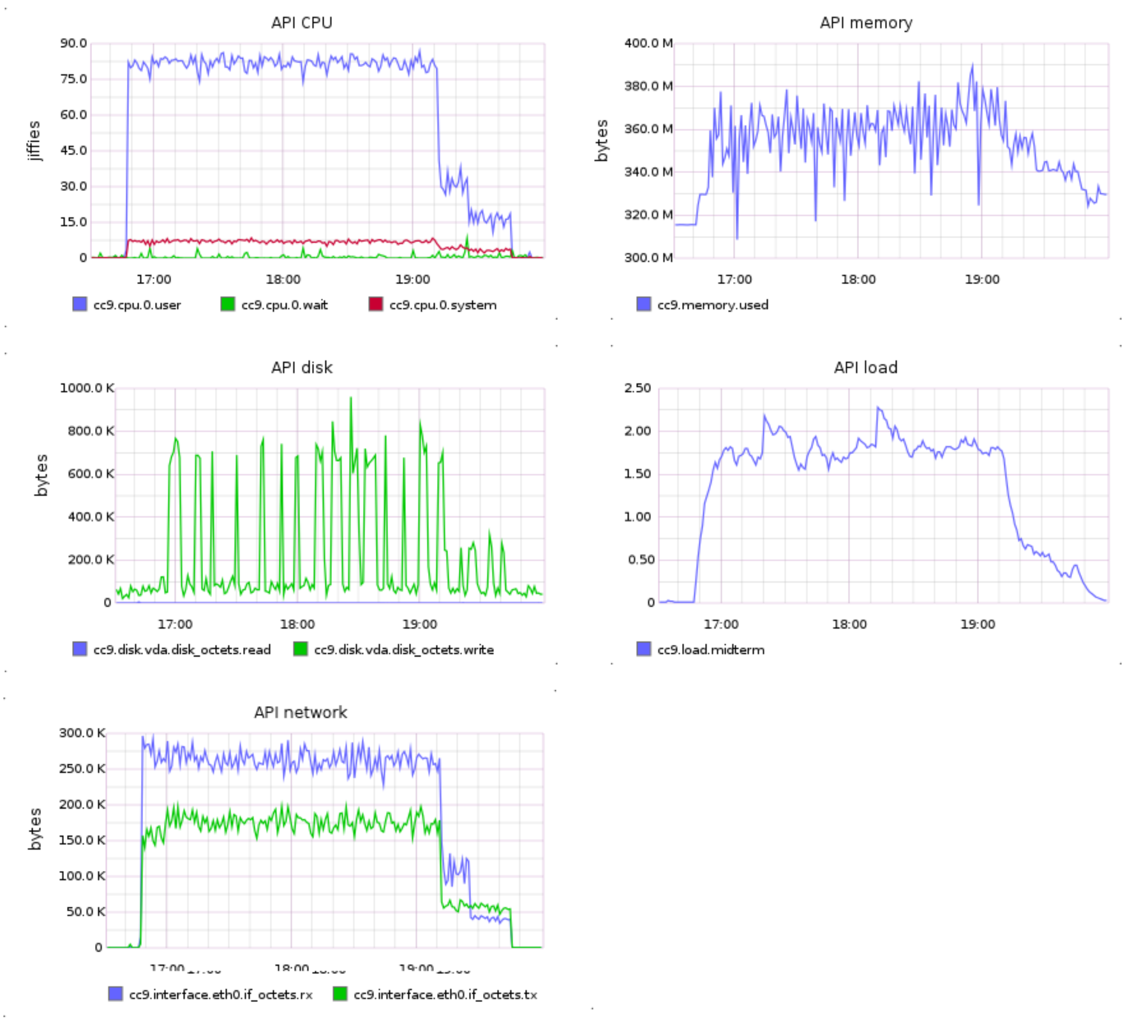


Figure 7.3: Hardware API nodes — first run

requested and the workers have all the resources of the system to themselves. The graph ends at 7[9] outstanding tasks and stays there indefinitely. Correlating this fact with the status 500 responses throughout, and the *postmortem* analysis of the workers' logs as well the *Gearman* driver code, we conclude that when the workers throw an exception the job is considered failed and is not returned to the queue. This is exceptionally undesirable behavior on the part of the driver, even for such a small rate of failure. Either customizing the driver code or fixing the worker to catching the exception and retrying the query on the Time Series Store after a back-off time are possible solutions to this problem.

Figure 7.3 shows the hardware utilization of the API nodes. We can see that CPU became saturated very early and continued heavily utilized until 19:12. This, combined with poor request throughput, hints at performance issues that can be optimized. After 19:12 CPU utilization drops,

[9]The workers threw 2 exceptions but 7 tasks were affected. This is due to the fact that when an exception is thrown, the API is not contacted for task completion and, therefore, does not trigger the next task in the pipeline. The other 5 tasks unaccounted for are the ones in the pipeline that never started.

along with system load. This indicates that one Worker node is not enough to saturate one API node, and the bottleneck at this stage is either the Worker node, the Time series node, or, more likely, a combination of the two.

Memory usage is very erratic in the short term, but predictable in the long run. This is due to the mechanism we implemented to mitigate memory leaks that recycles processes. The pronounced troughs in the graph indicate when a process gets recycled. Also, memory utilization is well below the system limit of 3GB. Expectedly, its utilization decreases after 19:12.

Disk utilization is very unexpected. The disk is written to in periodic and highly intensive bursts. After inspecting the contents of the node we attribute this disk usage pattern to the logging machinery of *pgPool-II*. We found *pgPool-II* had created about 500MB worth of text logs during this run. This is likely excessive logging, but because *pgPool-II* logs every SQL query we may still extract some knowledge from that: the API probably also makes excessive calls on the Operational and Triple Stores. Reducing the number of calls the API makes on these Stores is a good candidate for further optimization. Read bytes (blue) are practically non-existent and this is somewhat expected. There is no good reason for the processes on the API nodes to constantly read bytes from the disk.

System load kept well above 2.0 until 19:12. This indicates the node was saturated beyond its optimal capacity and indicates a possible bottleneck in the first part of the run. After 19:12 it decreases as expected, indicating a bottleneck exists elsewhere.

Network throughput (green for transmitted bytes and blue for received bytes) remains constant throughout the run and, expectedly, drops at 19:12 when there is less load.

In Figure 7.4 we may see the hardware utilization of the Worker node. User time kept steadily low and only climbed after 19:12. This clearly shows that worker processes are CPU bound and before 19:12 were starved for API resources. Once API resources became available at 19:12, the workers increased hardware utilization in all CPU and network aspects. The fact that CPU utilization in the API nodes after 19:12 decreases, yet request throughput climbs and worker CPU utilization is comparatively low reveals that the workers are making too many requests on the API. This is evidenced by relatively low CPU utilization on both nodes, yet worker system load climbs slightly above 1.0.

Memory utilization warrants the same comments as the API nodes'. Disk utilization behaves expectedly, with minimal write activity likely due to logging. System load increases over 1.0 after 19:12 which suggests the worker node becomes saturated after this point. Network utilization climbs expectedly at 19:12.

Figure 7.5 shows hardware utilization of the Time Series Store node. CPU utilization climbed after 19:12 which further reinforces the notion that a combination of Time series and Worker nodes constitute a bottleneck for general throughput at this time. Still, CPU is not saturated and system load is consistently just under 1.0 which informs us that the node can handle slightly more load than what we introduced. Memory usage is quite high compared to other nodes, for which the Cassandra daemon is responsible. Usage kept flat throughout the whole run, which we assume is because Cassandra allocates a maximum number of read/write sockets and keeps them in memory

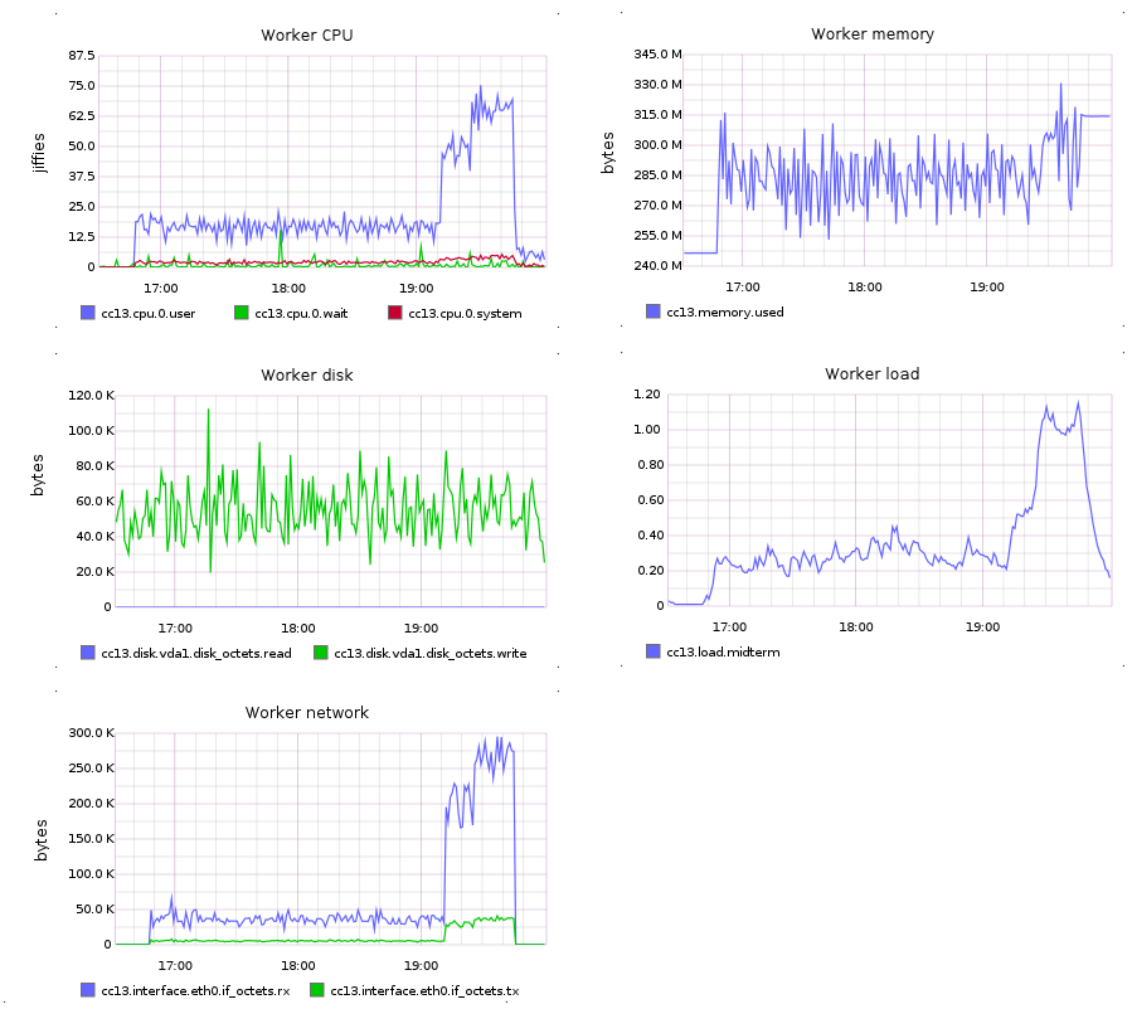


Figure 7.4: Hardware Worker nodes — first run

while load is introduced. We assume the drop at the end evidences garbage collection of those sockets when their timeout is reached and no inbound connections are detected.

Disk utilization is surprising. Write operations are lower than expected, for a node that handles so much write traffic. *Postmortem* analysis of the node reveals approximately 40MB of commit logs and 4.5MB of data. This is baffling information and we attribute it to compression Cassandra performs on the data. As per its default behavior, Cassandra keeps data in memory and flushes it periodically[10] to disk for durability. We assume these writes already comprise compressed data, whereas commit logs are in text format. Read operations are also inexistent throughout. We assume Cassandra keeps the whole data in memory and does not need to fetch it from disk. However, we cannot assume this will be true when data grows beyond memory capacity.

Network utilization mirrors the behavior of CPU utilization, as expected: when CPU utilization rises, so does the traffic generated. Transmitted bytes far surpass received bytes: whereas

[10]The default value is every 10 seconds.

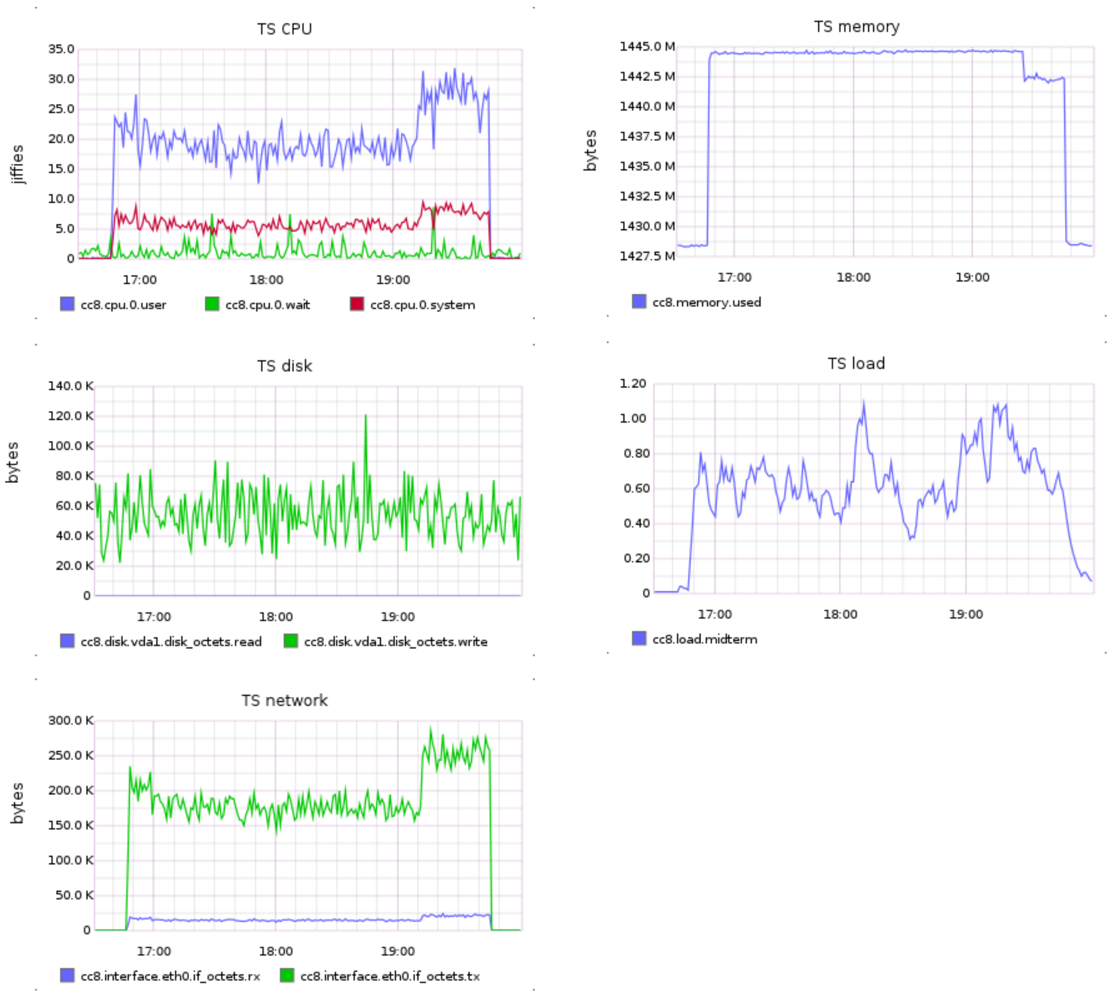


Figure 7.5: Hardware Time series nodes — first run

received bytes correspond mainly to CQL queries and datapoints insertion, transmitted bytes correspond to data read and sent.

Figure 7.6 shows hardware usage of the Operational Store master server. CPU time is dominated by wait time, suggesting the processes are kept waiting for disk IO. This is consistent with heavy write disk activity throughout and with the role as a database the node has been assigned. Both wait time and write operations expectedly climb after 19:12, evidencing the rise in request throughput, mainly dominated by small requests made by the workers.

Memory consumption drops at approximately 17:15, which we cannot explain. Remaining relatively constant throughout the test run, as it happens from that time to 19:12, is expected behavior. We suspect the drop at 19:12 is caused by a similar mechanism to the Time Series Store: garbage collecting connections. The connections that stay active are the ones used by the workers, which represent a smaller diversity of operations. System load climbs after 19:12 to account for the increased request throughput, as every request on the API typically involves consulting and writing operational data. Load is well below 1.0, indicating this resource is underutilized.

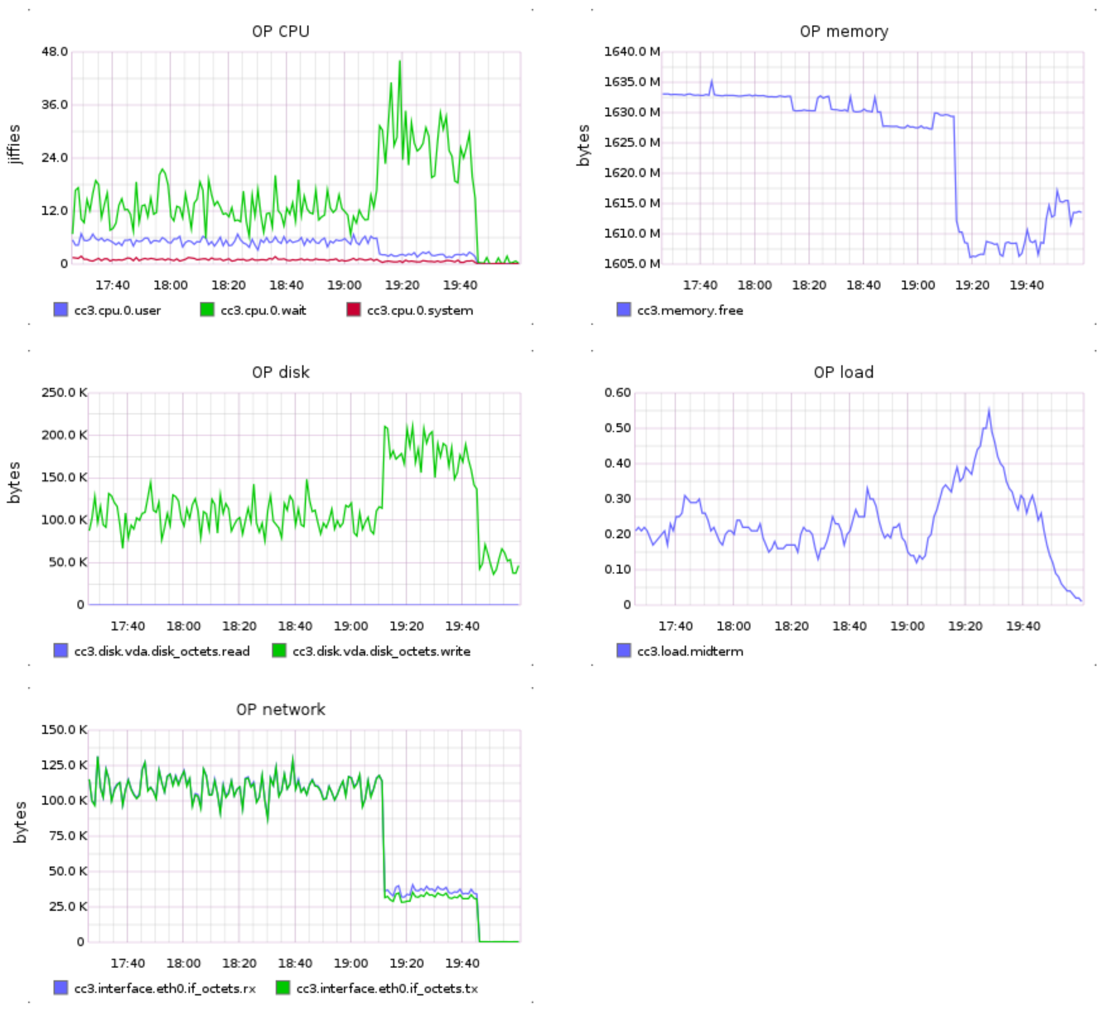


Figure 7.6: Hardware Operational Store nodes — first run

Network utilization is constant until 19:12, after which time it falls. This is apparently incongruent with CPU wait time, disk usage and request throughput patterns and we cannot at this time provide an explanation for decreased network activity. We suspect the very similar amounts of transmitted (green) and received (blue) data is attributed to the way SQL is generated by Django.

Figure 7.7 shows the graphs for hardware utilization of the Triple Store node. Hardware utilization is quite unremarkable. Similar to patterns observed in the Operational Store, CPU wait time is quite high compared to other times. Memory usage follows the same trend albeit at different time points. The small climb at 19:12 is unexpected and unexplainable because after that point the store is no longer in active use as the relevant client behaviors are no longer active. Network activity drops to base levels after 19:12 as expected. System load remains quite low throughout, indicating this node is easily keeping up with demand.

The general magnitude of the run, as defined by the third set of metrics in Section 6.3, is the following: 19653 datapoints, 1734 tasks, of which 5 never finished, 1164 datasets and 1850 triples. The number of datasets in particular appears high, but note that the each dataset comprises very

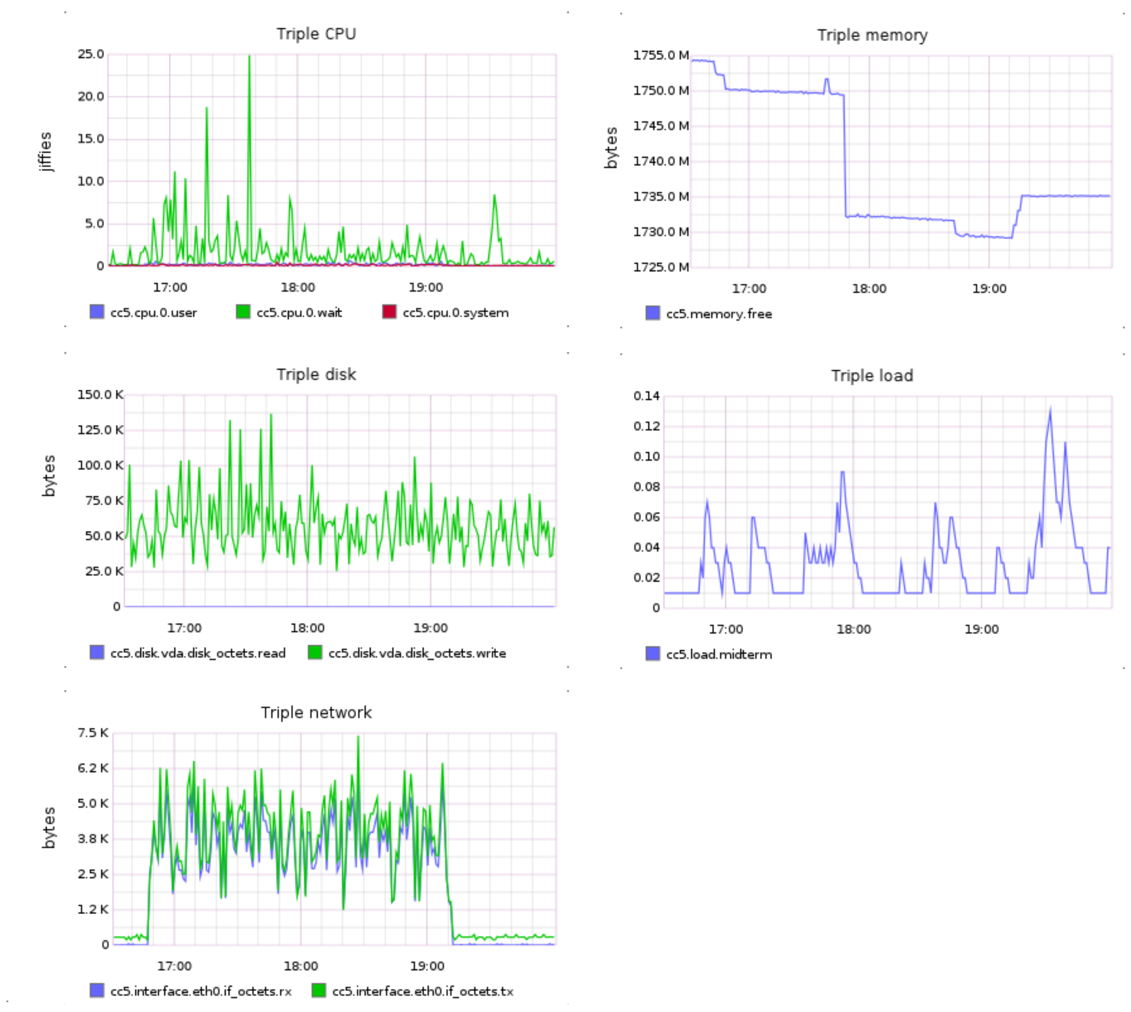


Figure 7.7: Hardware Triple Store nodes — first run

few datapoints.

7.2.1 Retrospective

During this test run we encountered both expected and unexpected behavior. The unexpected behavior was mostly due to implementation details and our own lack of deep experience with distributed systems.

However, many conclusions may be drawn from this test run. We found that the API nodes are a central point in the system and quickly become a bottleneck on the throughput and quality of service. Other components do not see as much usage as this one, in particular the Operational and Triple Stores, which seem to be able to handle substantially more load than introduced so far. We also found that the operations on the Worker nodes are delayed because of resource contention in the API nodes, as evidenced by the hardware utilization after 19:12. The Time Series Store utilization is somewhat close to maximum, in particular after 19:12. We also speculate that the implementation of the API and Workers in particular may benefit from optimizations, particularly in regards to connecting to the various stores. We propose that, were this platform actually in production, these optimizations be identified and implemented before commissioning further nodes so as to make better use of existing resources and not increasing operational costs unnecessarily.

For the next test run we will introduce more API nodes to share the load and hopefully achieve greater throughput. We suspect that the utilization on the Worker nodes will be consistently higher and the number of outstanding tasks to follow a less steep line. The Time Series Store will suffer greater load and will probably be the bottleneck on the second test run.

7.3 Second test run

In this second test run we made some bold decisions regarding the testing and the scalability of the platform. First, we cleared all data resident in the platform, so as to start the run with a "clean slate". The second decision was to add one API node and one Time series node, as per the conclusions drawn from the first run. We based this amount of nodes on the load average data collected in the previous run. The starting inventory for this test run is thus: 2 API nodes, 2 Time Series Store nodes and 1 Operational Store, Triple Store and Worker nodes.

We also tweaked some internal values in the worker nodes, namely: the recycling interval was increased in order to minimize process forking overhead and the number of datapoints each worker reserves for computations was also changed[11]. The new batch size we defined is conservative in that it attempts to strike a balance between data resident in memory and efficiency. This change places less pressure on the API and Time Series Store by making bigger batches of computations and not having to contact those components as often. It also increases CPU utilization, which we see as a more optimal usage of resources and better mirrors the expected CPU-bound nature of workers.

The biggest change was to the methodology used, which mirrors the one we defined in Section 6.2. In this test run we introduce more nodes as the run progresses. We base the decisions to scale the system on the data collected during the run.

Finally, the load profile was also changed, in order to approximate production conditions. Whereas in the first run the number of datapoints per dataset was rather small, we now increase and diversify that number. The load profile is as follows: 6 instances of Researcher, 3 instances of Tripler and 3 instances of Data Provider, all of which reside in a single node. Two of the Data Providers now insert datasets with approximately 3000 datapoints, which correspond to over 9000 seconds worth of data. The other instance inserts datasets with approximately 30000 datapoints, which correspond to a little over 1 day worth of data. We will not change the load profile throughout the run.

As per Section 6.2, the run ends when we determine that increasing the number of nodes does not yield a corresponding increase in high-level performance metrics. The test run started

[11]The previous value was 10, the current value is 200.

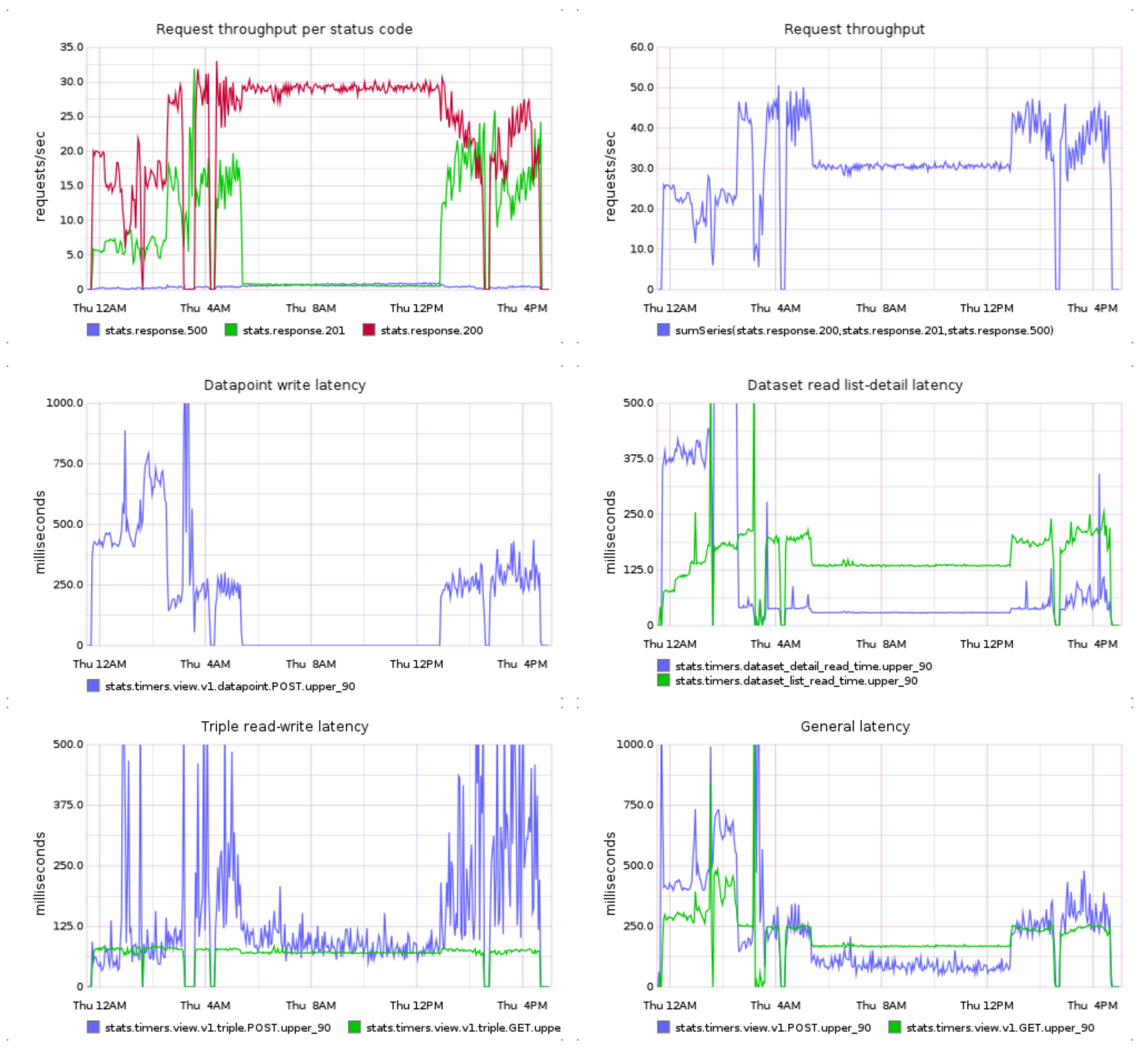


Figure 7.8: High level metrics — second run

at 23:42 of 2014-05-28 and ended at 16:39 of 2014-05-29. Some graphs are clipped at certain values in order to ignore aberrations, particularly in the timings. Typically, these unusually high spikes correspond to points in time where a node was being brought online and configured. This puts significant pressure on the disk of the physical machine and since all nodes share the same physical machine, all are affected.

In this section we will not analyze each graph in order, but instead we will provide a chronological account of what happened. We believe this better illustrates and explains the decisions we made along the way. In order to make decisions regarding the scalability of the platform, especially while it is in use, we must take into account all the relevant information at once. Analyzing a single source of information does not take into account the distributed and dynamic nature of the platform and may lead to erroneous conclusions, as it is the interactions of the components that makes up the platform and not each one individually. We then ask the reader to refer to the relevant figures as he or she reads along.

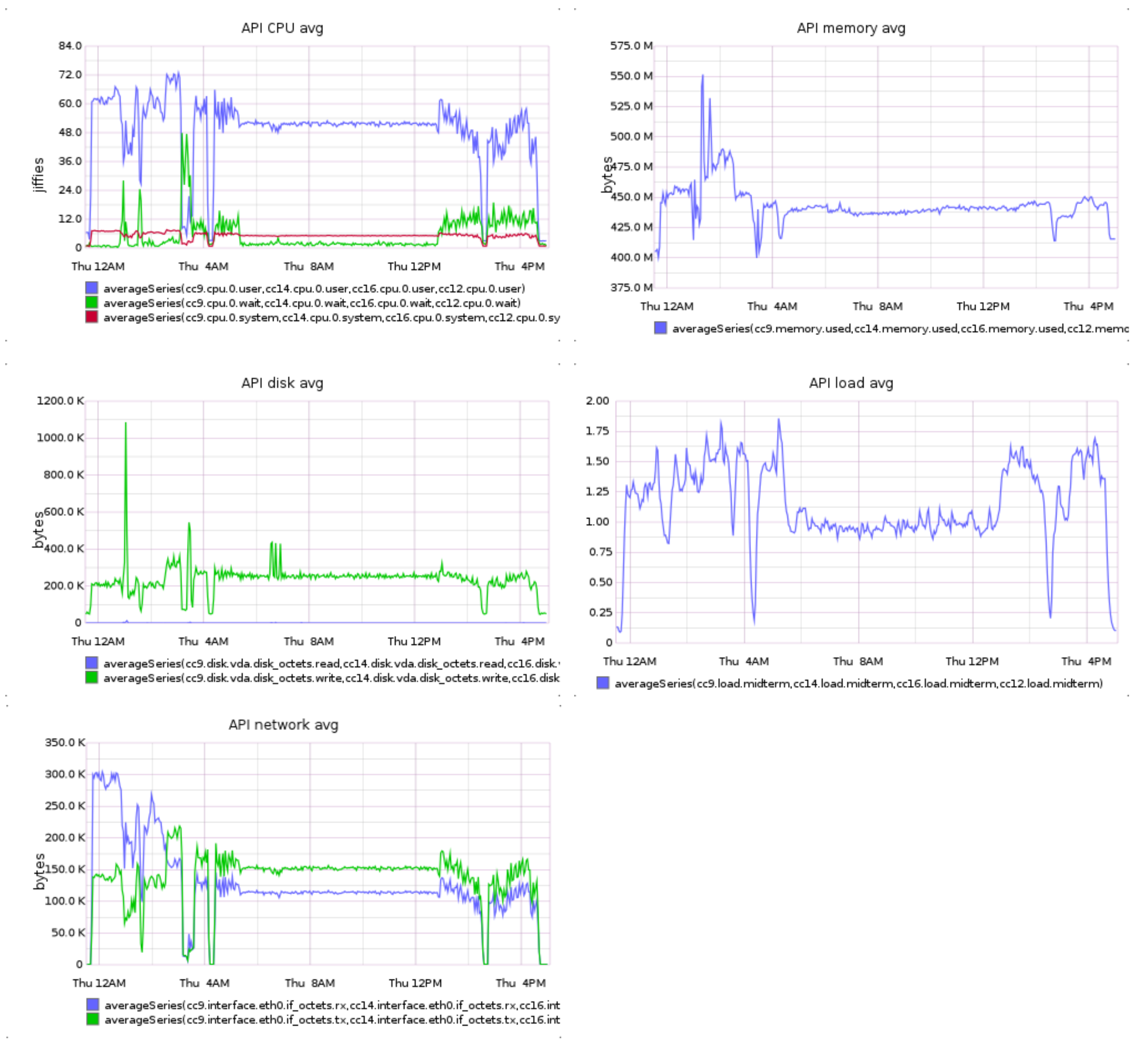


Figure 7.9: Hardware API — second run

Figure 7.8 shows the high-level metrics collected throughout the run. Comparing request throughput with Figure 7.1 we can see that our decision to commission one additional API and one additional Time Series Store nodes increased throughput roughly two-fold. This is obviously expected. However, the latency of datapoint insertion has climbed from about 350ms to about 400ms. We attribute this fact to the need of the Cassandra daemons to communicate and to the fact that the Time Series Store was now being used by two API nodes, therefore facing additional load.

On the Request throughput per status code graph, the first small increase in status 201 (corresponding to datapoint insertion) and the decrease in status 200 indicates the time that the first datasets were successfully inserted and the workers began processing the data. The different request profile stems from the worker activity that increase the share of requests inserting datapoints. This happened at about 0:12.

Analyzing Figures 7.9 and 7.10 we see that both stores are overloaded and they constitute good

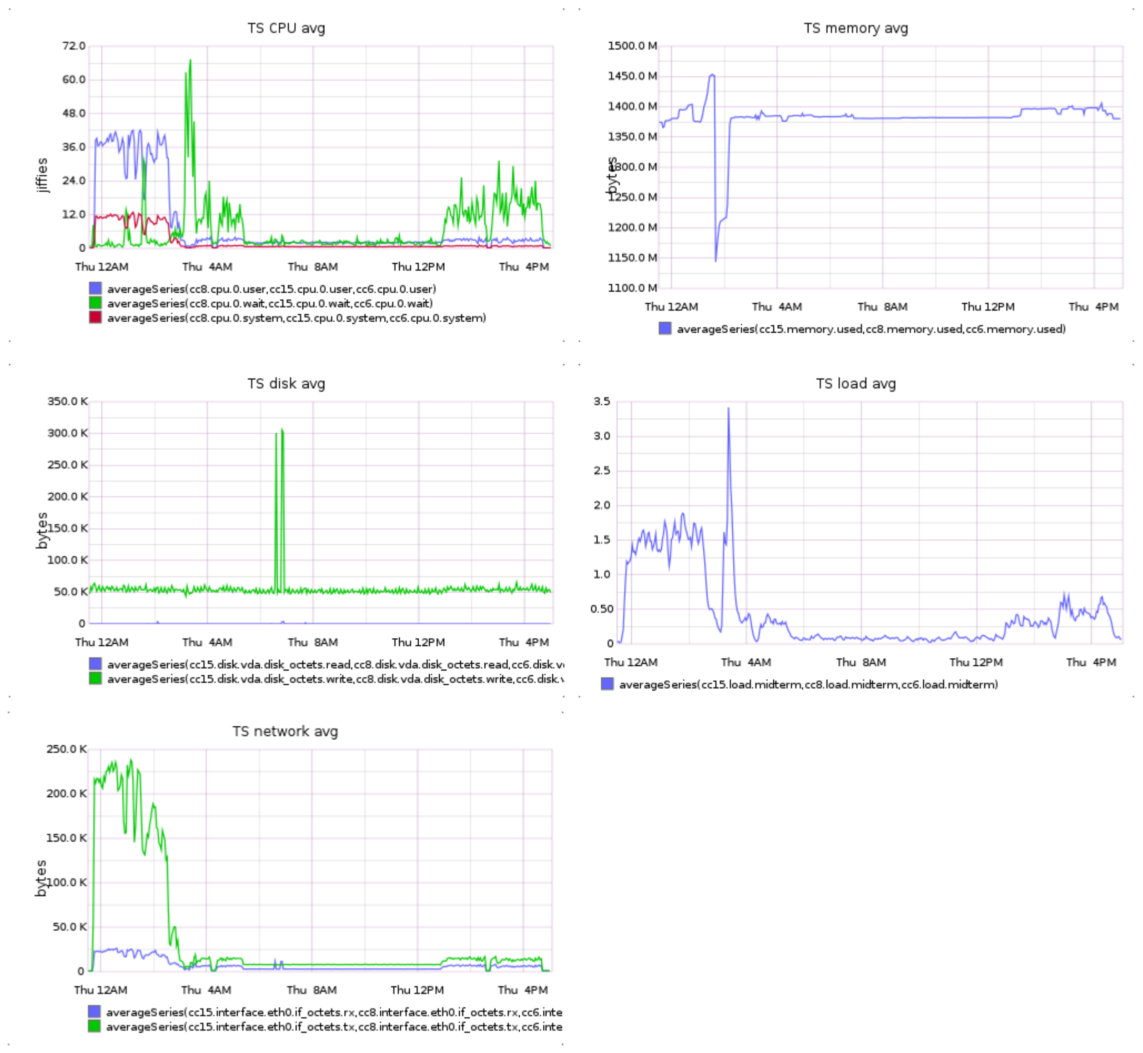


Figure 7.10: Hardware Time Series Store — second run

candidates for scaling. CPU utilization on the API is significantly higher and almost saturated, therefor we decided to scale this component first, at 0:51. Resource contention on the physical host[12] caused the platform to behave erratically and significantly reduced performance, to the point where the Operational and Triple Stores became almost unusable and caused some client instances to crash. Clients were restored at 1:10 and the new API node was fully online at 1:18. This decision hardly impacted throughput and latency.

At this point, the hardware utilization of the API and Time series nodes was still consistently high, hinting at bottlenecks in these components. Other components showed relatively low usage, as shown further below. At 1:31 we decided to add one Time Series Store node. Across all graphs we can see that resource contention on the physical host significantly affected the performance of the platform. At 1:50 the node was fully online. Again, this decision hardly changed performance and actually increased datapoint insertion and dataset detail read latencies. Clearly, scaling out the

[12]In particular disk usage, as evidenced by the unusual spikes in CPU wait time across all nodes.

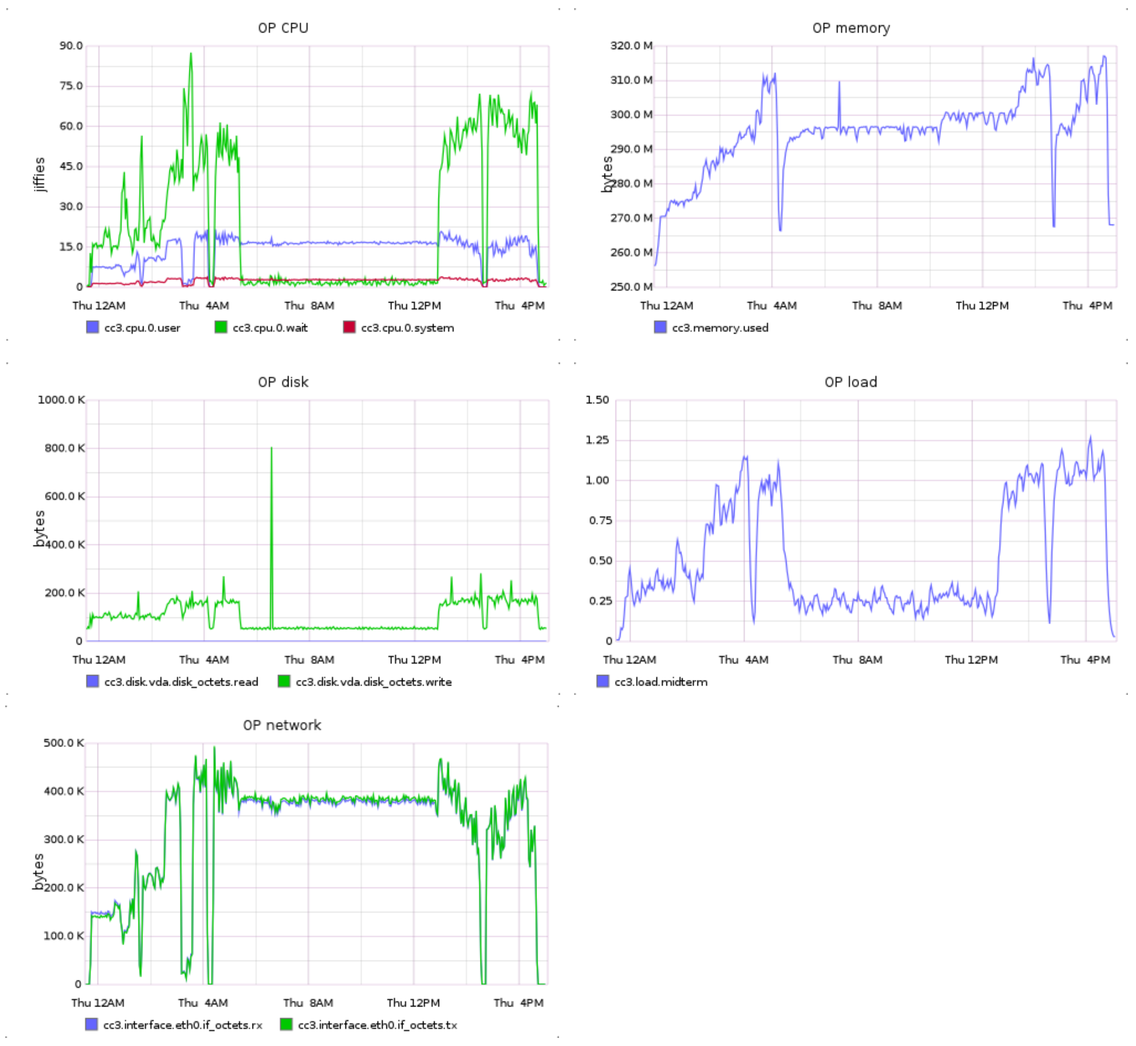


Figure 7.11: Hardware Operational Store — second run

Time Series Store was having the precise opposite effect of what was desired. This was extremely unexpected and further evidenced code inefficiencies, particularly in the API.

With this information at hand, we decided to pursue a different course of action. We implemented changes to the way the API connects to the Time Series Store, reusing sessions instead of opening a new one with every request. The changes were dramatic and may be seen in the increase in request throughput, sharp decrease in load and CPU time in the Time Series Store and also the decrease in datapoint insertion latency, bringing this value to par with the other stores. With this change the Time Series Store component became over-provisioned at 3 nodes.

Significantly, the Operational Store master node now climbs in load and CPU wait time, as can be seen in Figure 7.11. This evidences high disk contention on the physical node and, to a lesser extent, node saturation. This increase in resource usage is attributed to the increase in request throughput. Further evidence of physical disk saturation lies in consistently high CPU wait time, even across nodes that typically do not experience such a pattern, like the API nodes and the Worker node.

At 3:07, after we analyzed the impact of the optimizations, we decided to once again add another API node. This brought the inventory of API nodes to 4. Expectedly, performance suffered and became erratic until 3:33, when the node was fully brought online. This change slightly increased request throughput, but clearly not on the scale of what we expected. The trough at slightly after 4:00 was due to an operator mistake which killed all client processes. They were quickly restarted and operation continued unattended between that time and 12:53 of 2014-05-29.

At 5:30 some clients (the Data Providers) disconnected unexpectedly and *postmortem* analysis could not reveal the cause of this problem. This suggests better instrumenting and log analysis tools are needed for all nodes. The remaining clients during this period of time were one instance of Tripler and two instances of Researcher. At 12:53 we reinstated the expected load on the platform. Request throughput and write latencies were slightly lower than previously in the run while CPU wait times across all nodes increased.

At 16:42, when no further behavior changes were detected particularly pertaining to request throughput, the run was terminated and all clients ceased activity. This decision was based, as mentioned before, on the fact that adding the last API node did not increase performance metrics. We then conclude that the clients were not saturating the system with data and an approximation of an optimal point was found. Another reason to terminate the run was that disk contention on the physical host became a serious disrupting factor in interpreting the metrics, such that we can no longer attribute measured phenomena to their respective causes with confidence.

The general magnitude of the run is the following: 233555 datapoints, 171 tasks, of which 13 never finished, 152 datasets and 31943 triples. Notably, the amount datasets and tasks were much lower than in the previous run due to the fact that each dataset was proportionally bigger. The abnormally high rate of job failure stems from live changes made in the worker code which ungracefully terminated execution.

As seen in Figure 7.12 the Worker node remained rather underutilized throughout the run. Contrasting with the data from the previous run (per Figure 7.4) we conclude that the rate of task scheduling increases resource usage. In other words, when processing bigger datasets the workers are able outpace the data providers, whereas before they could barely keep up with demand. This is easily confirmed when contrasting Figure 7.2 with Figure 7.13 (on page 77). In the latter picture we see that tasks do not queue excessively (12:00-14:00 approximately) or at all (0:12) to be processed, as happened in the first test run. The first jump to 7 tasks and then again to 17 mark two events where tasks were added, but the workers stalled the pipeline by exiting ungracefully. Low hardware utilization is inconclusive due to noise introduced by disk contention.

As a final note, the Triple Store behavior seen in Figure 7.14 (on page 78) is fairly unremarkable, with the spikes in load average and CPU corresponding to other nodes' configuration procedures. The spike in disk writes remains unattributed. All hardware metrics report similar phenomena at that time, which suggests this event was scheduled by the operating system of the physical machine.

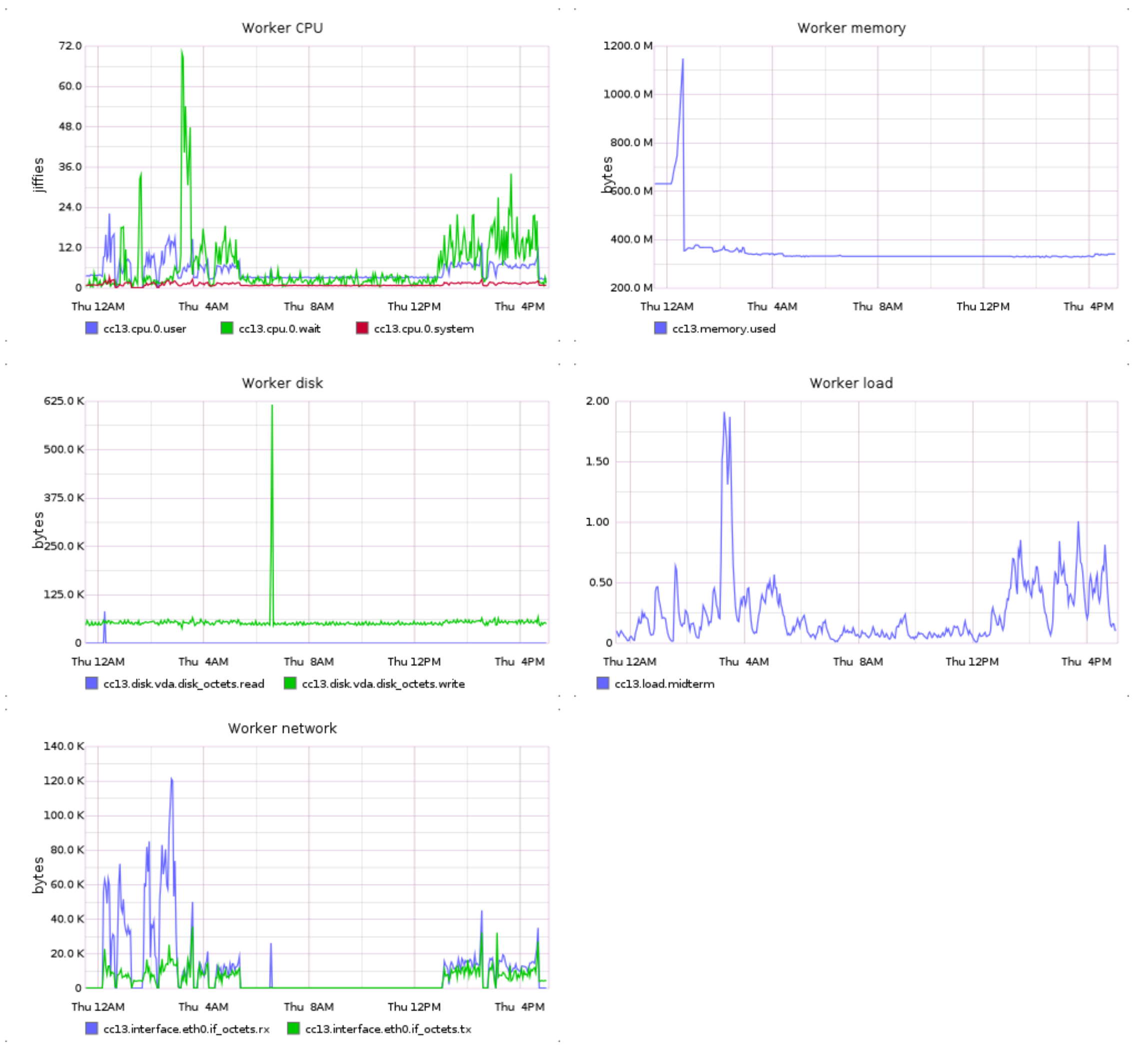


Figure 7.12: Hardware Worker — second run

7.3.1 Retrospective

In this test run we achieved significantly higher throughput of requests. This was both due to increased hardware allocation and optimizations made to our own code. We found that the Time Series Store did not need as many nodes as previously suspected and, in fact, adding more nodes worsened performance indicators. Perhaps the biggest contribution to performance was the way the API and Workers connect to the Time Series Store. This change both increased throughput and decreased resource utilization on that store.

The most important finding during this run was an approximation of the optimal balance between load introduced and hardware allocated, as defined in Section 6.2. The suggested hardware in order to sustain approximately 40 requests per second of mixed load is 3 API nodes and 1 node for each other component. All components showed load averages close to 1.0 with the exception of the Time Series Store which was over-provisioned and the Triple Store, which we assume was

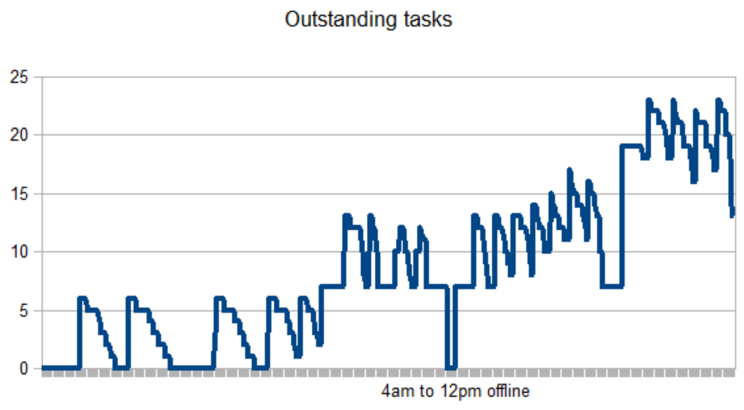


Figure 7.13: Outstanding tasks — second run

not stressed enough during this test run. The next probable candidates for further scaling would be the Operational Store and the API components. However, since all nodes share the same physical hardware, disk contention becomes a serious problem and skews results in ways we cannot fully determine. We attribute disk contention to the insertions in the Time Series Store.

Another sporadic factor in hardware saturation is node configuration. This is a result of our testing conditions and would not occur in production settings. At 26 and 27 minutes for an API node and 19 minutes for a Time Series Store node, this is an extremely long time when responding to sudden increase in demand. This long time is exacerbated by the fact that the hardware is not idle while making initial configurations. Again, this is a product of our testing environment. In either case, to combat this behavior we propose that instead of configuring a node from a base image we instead pre-configure disk images and install them on the new nodes joining the platform. These pre-configured images would then be further configured by the orchestrating subsystem, mainly for managing the configuration of the various routers: HAProxy, nginx and *pgPool-II*. This would significantly decrease deployment time and, consequently, response time to increased demand. If such a system is not feasible or still lacks response capability we propose that the existent nodes be run at less than optimal capacity (over-provisioned) to accommodate sharp increases in demand while new nodes are being configured.

In summary, we conclude from this test run that small localized optimizations may be as important in increasing the capacity of the platform as commissioning more hardware. More importantly, we conclude and foresee that optimal resource allocation will approximate a ratio of 3 API nodes and 1 node for each other component. This is important for autoscaling features that may be implemented for the platform. While no explicit heuristics are provided for autoscaling single nodes at a time, future implementers of this feature may reuse the reasoning behind our decisions to scale the components. Furthermore, we cannot guarantee that increasing hardware by a factor of two will double performance. Finally, the issue we encountered while configuring new nodes is relevant in our testing environment, but may not be as important or even manifest at all in

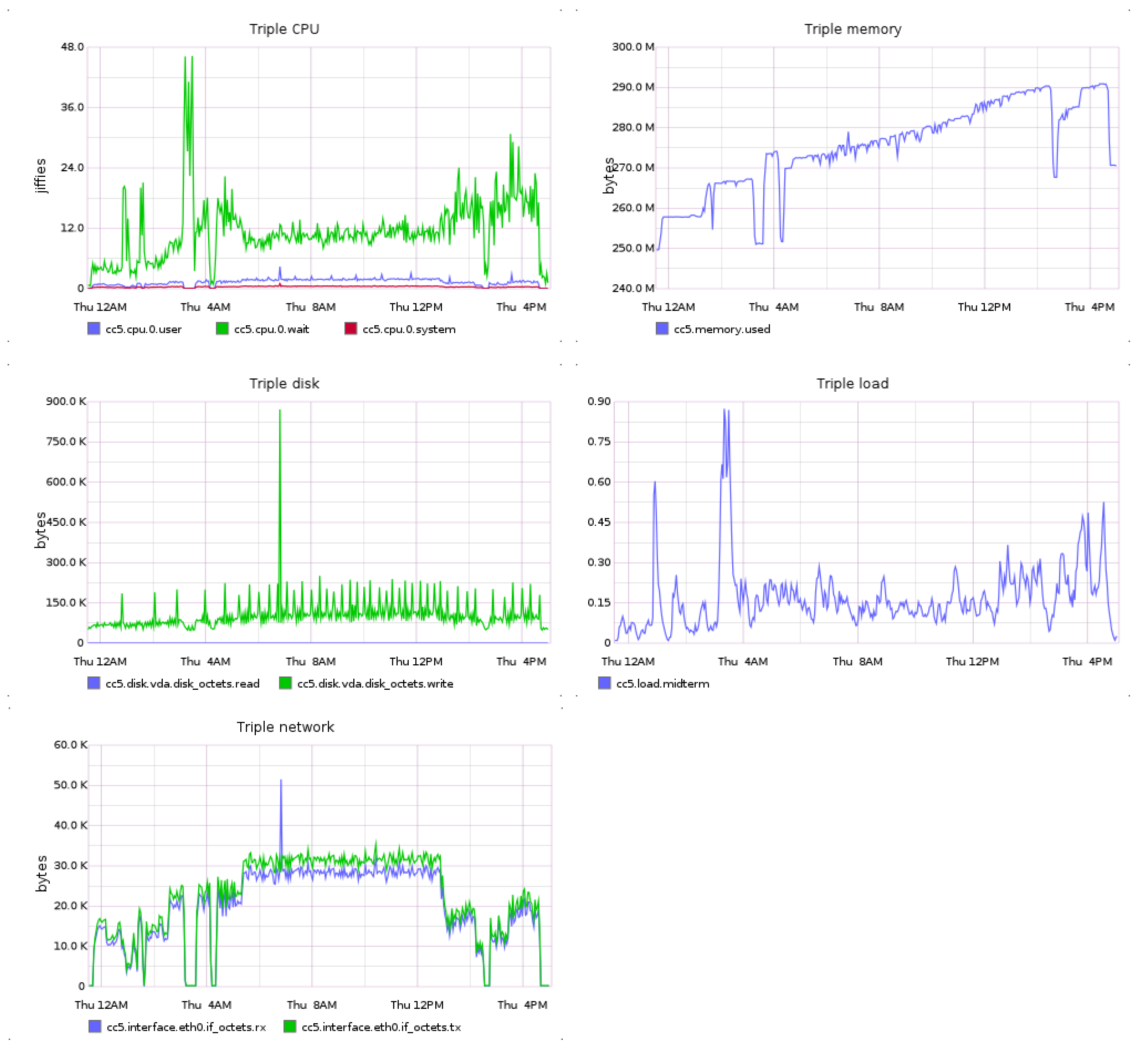


Figure 7.14: Hardware Triple Store — second run

a production environment.

7.4 Third test run

In this final test run we applied the combined knowledge of the previous test runs and introduced different load profiles that target specific components of the platform. Again, we cleared all data so it would not influence the behavior of the platform in this test run. However we did not clear data between load profile changes. Because we found that the Time Series Store was over-provisioned in the previous test run, we scaled down the instances of this component to just one for this test run. As per our methodology, we also posit that API component was slightly over-provisioned and we scaled it down to three instances. The inventory for this test run is the following: 3 API nodes and 1 node for each of the other components in the system. We did not change the inventory throughout the test run.

Again, we will provide a chronological account of how the events unfolded. The test run started at 0:34 of 2014-06-02 and ended at 22:05 of the same day.

The first load profile introduced was the Data Provider behavior. We commissioned three nodes to act as clients. One node had 6 instances of Data Provider that inserted datasets in the order of 30000 datapoints, or a little over 1 day worth of data. The second client node ran 6 instances of Data Provider that inserted approximately 11 days worth of data. The third client node ran 6 instances of Data Provider that inserted approximately 111 days worth of data[13]. Early on the system started showing instability and some of the clients disconnected, possibly due to timeouts. To prevent instability and allow the system to properly function unattended we instead deployed the second and third clients with 3 instances each. The first client node was left permanently disconnected. This load still saturated the platform due to disk contention on the physical host, as can be seen further below. This load profile started at 0:34 and ended at 14:47.

The second load profile introduced was the Researcher. We recycled the previous 3 nodes, stopping the Data Providers and starting the Researcher instances. Each node ran 3 Researcher instances. They started at 20:00 and ended at 20:54.

The third and final load profile was the Tripler. We followed the previous procedure of recycling the nodes and starting 3 instances of Tripler in each. We ran into an implementation error on the Triplers that incorrectly did not insert any triples. This was fixed and the load profile restarted again. It started at 20:54 and ended at 22:05.

In Figure 7.15 we see the high-level metrics for this run. We see that for the duration of the first load profile, which was left running unattended, metrics behave relatively constant. Request throughput hovers at a little over 40 requests per second and latency for datapoint insertion keeps at about 250 milliseconds which is consistent with the results from the second test run.

Figure 7.16 shows the hardware usage of the API nodes. CPU wait time is unusually high relative to other runs and user time is unusually low. Other metrics are relatively unremarkable. We will analyze the change in behavior after 20:00 further below. Unusually high wait times can be seen in all other nodes, notably in the Triple Store which was idle during this time period. This constitutes more evidence for extreme disk contention on the physical host. Figure 7.17 shows the hardware utilization of the Triple Store.

Figure 7.18 shows the hardware utilization of the Operational Store. As evident by the CPU time graph, wait times account for almost all CPU utilization. This means the node was busy most of the time waiting for an IO operation to complete. It is somewhat contradicted by the low disk throughput. We would only find out the cause for this extreme behavior further ahead during the test run. All other metrics are unremarkable, with the exception of load average. It keeps consistently at 1.0 and drops sharply after 19:12 when the load pattern changes.

At 14:47 we turned off the Data Providers in the client nodes and, until 19:45, only the workers were active processing the inserted datapoints. However, at 19:12 there is an extremely unexpected spike in datapoint insertion. The request throughput jumped to yet unseen 125 requests per second. At the same time, as can be seen in all hardware figures, CPU wait time dropped to almost zero.

[13]Note that the sampling rate of the behavior does not change from what we prescribe in Section 6.1.2.

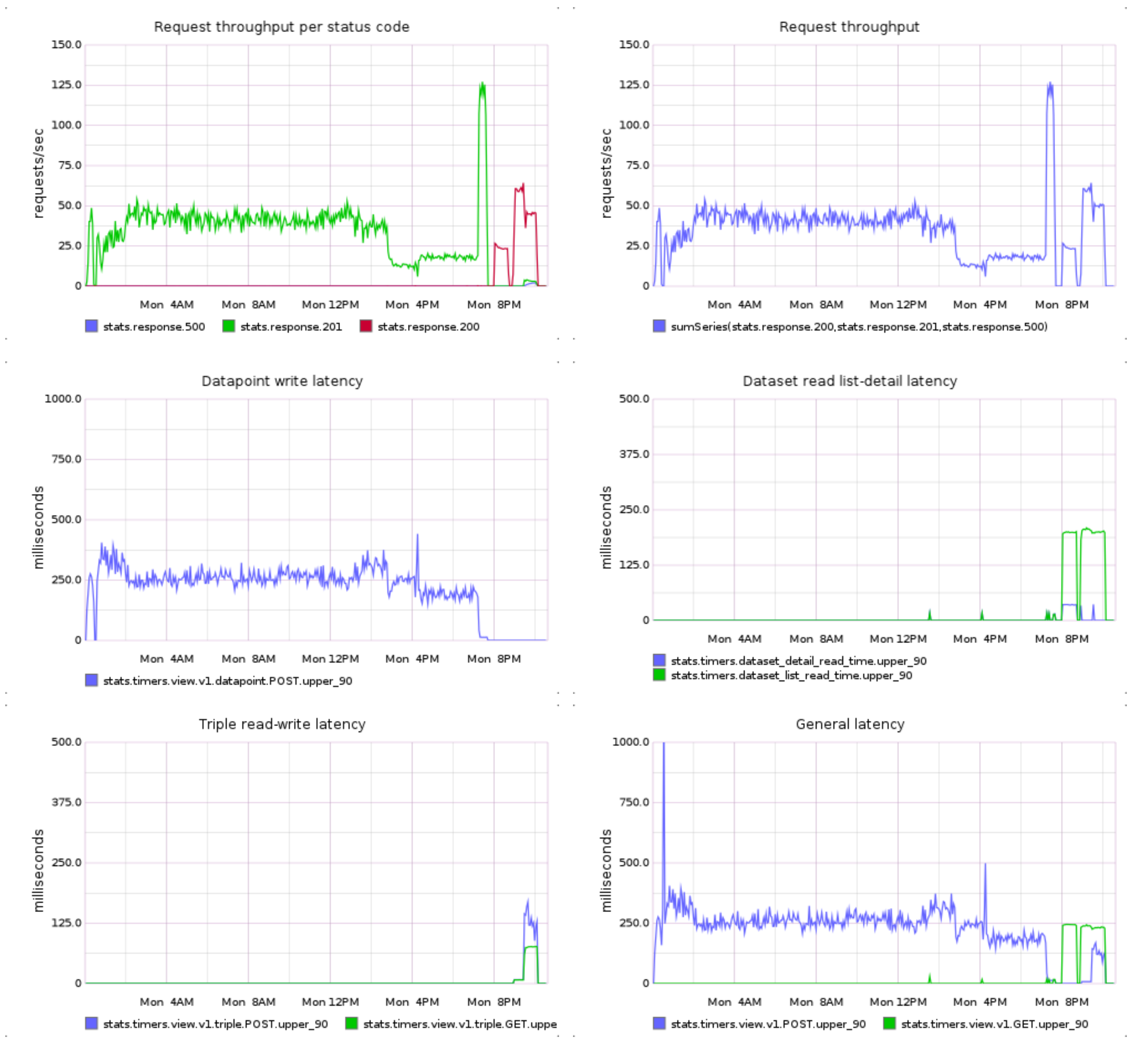


Figure 7.15: High level metrics — third run

Before this, we would think that disk contention was caused by high activity in the Time Series Store, graphed in Figure 7.19, even though the Operational Store showed significantly higher CPU times. We assumed the vast majority of write operations was directed to the Time Series Store.

However, the sudden and pronounced climb in request throughput was due to an implementation detail on the API. When any agent, external or Worker, inserts a new datapoint the count of datapoints per dataset must be increased. This count is stored in the Operational Store. The higher the throughput of datapoint insertion, the greater the number of updates to the Operational Store. As previously stated, this store is implemented using a RDBMS which guarantees ACID properties. Since it is constantly being requested to perform small writes, and all writes are synchronous, acknowledged and written to disk due to the Durability property, this drives disk usage up, in an inefficient manner. At 19:12, when the spike occurred, the workers were performing calculations which bypass the codepath on the API that updates the datapoint count[14]. Since the Operational

[14]This is the intended behavior.

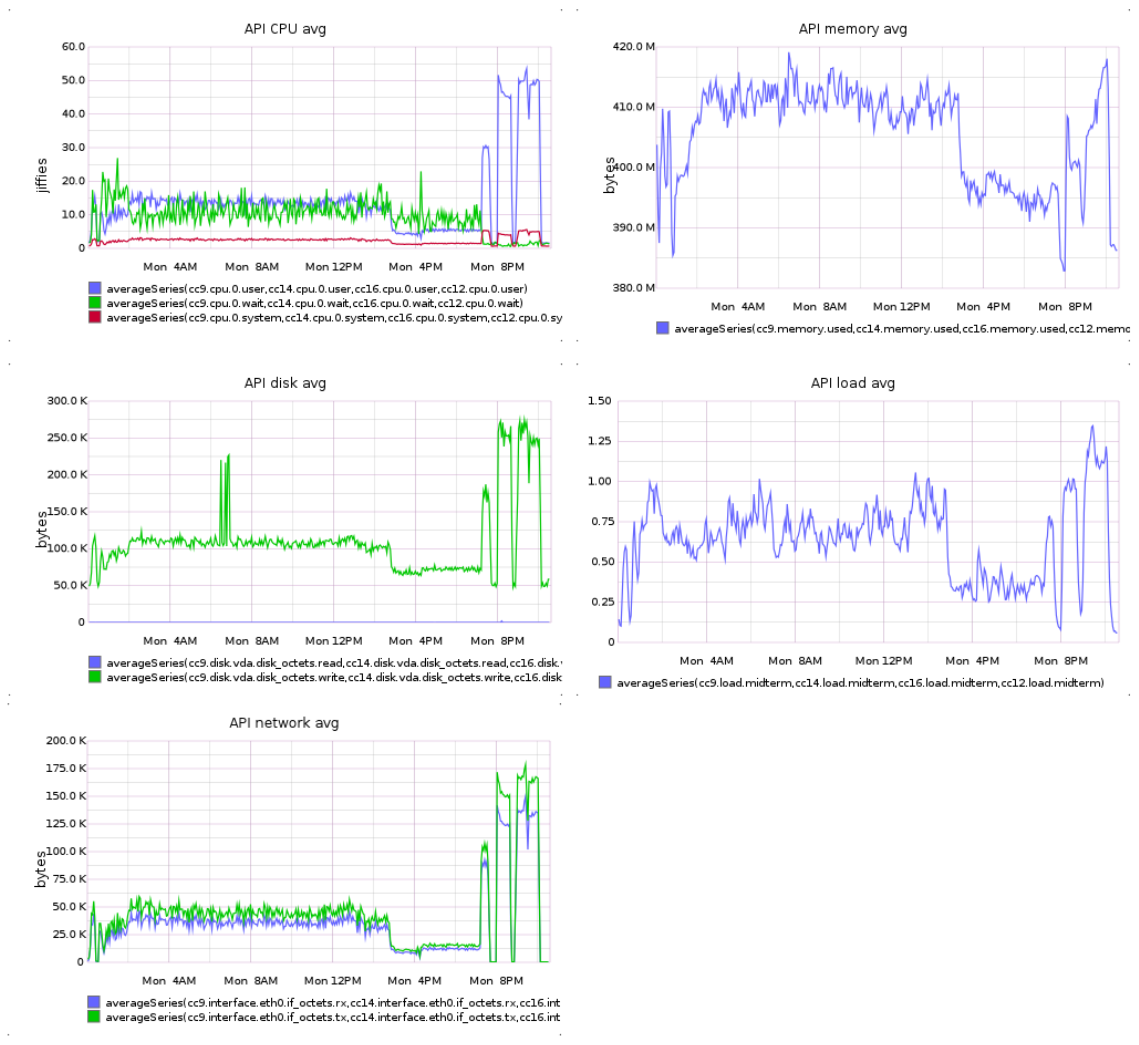


Figure 7.16: Hardware API — third run

Store was no longer suffering many small writes to disk, CPU wait times stepped down to reasonable values and this dramatically increased performance. The first test run hinted strongly at the API making inefficient use of the Operational Store and CPU wait times became a problem when datapoint insertion increased, further hinting at this problem. We strongly encourage this point be addressed in future implementations of the platform.

At 19:45 the workers stopped processing data unexpectedly. *Postmortem* analysis did not unveil what caused this sudden stop. Worker logs were incomplete and did not reveal enough information about what occurred, further suggesting better logging capabilities should be considered. We suspect an implementation error. Figure 7.20 shows hardware utilization of the workers. Their only active period was from 14:47 to 19:12, as evidenced the increase in CPU user time along with all other metrics. Still, it is clear that during this time period the node was under utilized.

Figure 7.21 shows the progression of outstanding tasks in the platform. Tasks were only inserted at 13:30, when the first datasets finished being uploaded. As discussed above regarding

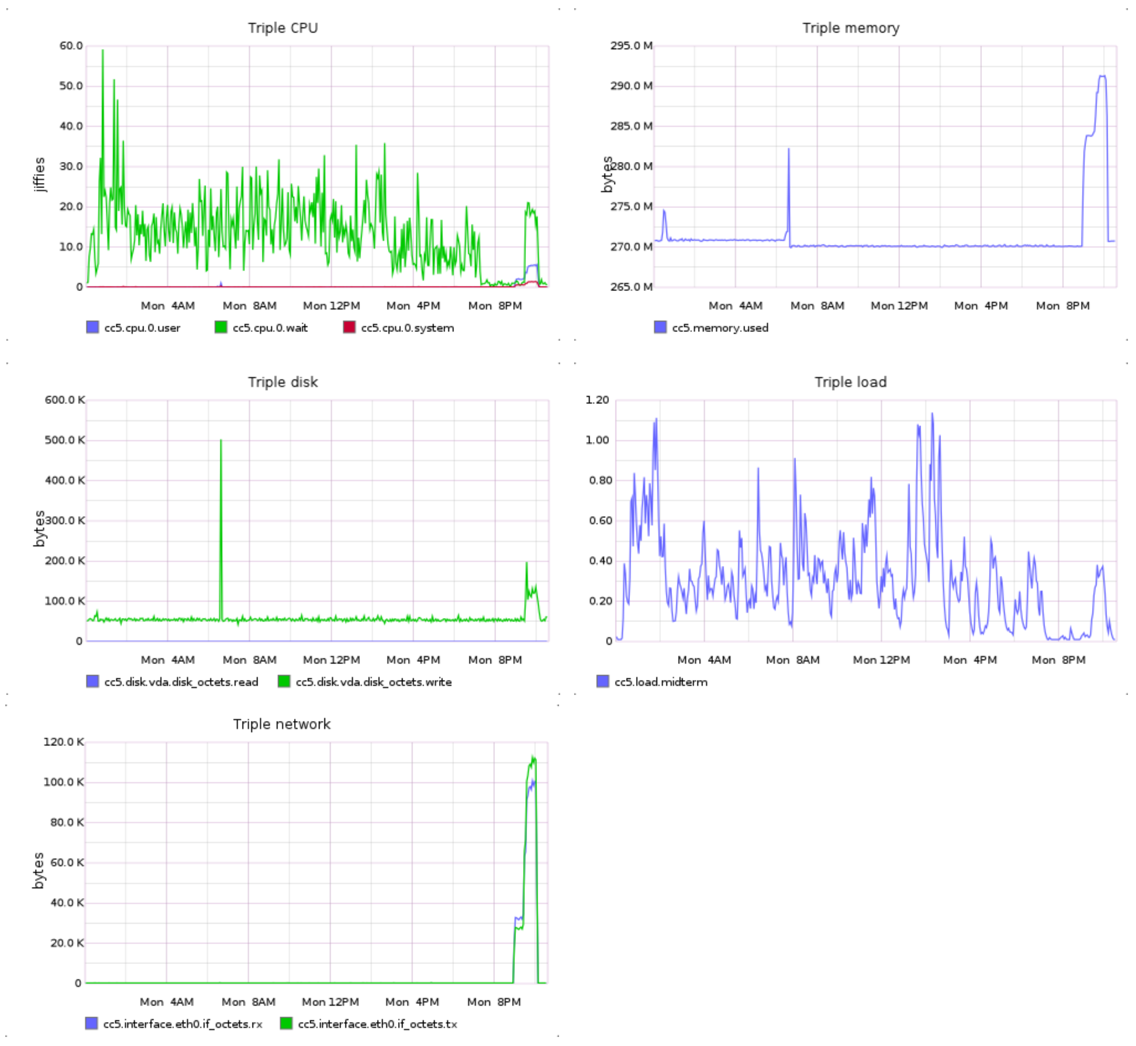


Figure 7.17: Hardware Triple Store — third run

datapoint insertion rate, the workers took an inordinate amount of time to complete the first procedures which were Aggregations. The second procedures, Calculators, took significantly less time to complete, due to increased rate of insertion. The graph ends at 3 outstanding tasks, due to what we suspect were implementation errors in the workers. Unfortunately, this graph is not as informative as in previous test runs because it does not show whether the workers could keep up with task scheduling.

Nevertheless, at 20:00 we started the Researcher load profile, discussed above. The API CPU user time increased as did the Operational Store's, consistent with previous results. The Time Series Store component became essentially idle, as expected. Request throughput during this time is rather underwhelming, as we would expect greater read capacity from the current configuration of the platform. We cannot attribute this phenomenon to any cause with confidence, as increasing the number of Researcher instances did not increase the throughput. Note that this behavior makes use of both the Operational Store and the Time Series Store to retrieve fragments of datasets. We

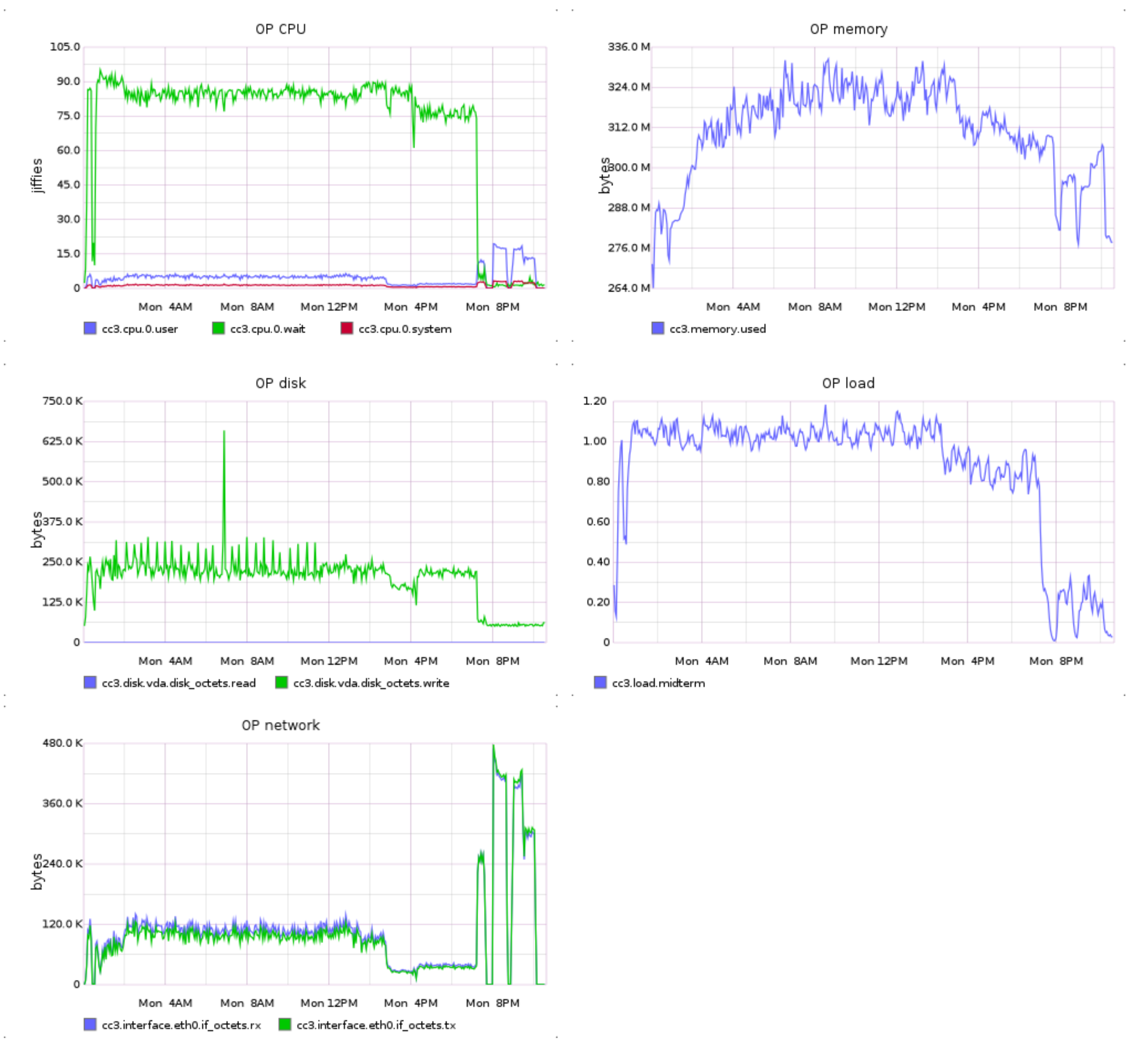


Figure 7.18: Hardware Operational Store — third run

suspect the way the API manages connections to the Operational Store is to blame. In order to further test this behavior we would configure a slave node to the Operational Store and redo the run with this load profile. This is a topic for future experimentation.

At 20:54 we stopped the Researcher profile and started the Tripler profile. The first plateau in request throughput corresponds to the erroneous implementation of the Tripler profile and shows the read capacity of the store. The second plateau corresponds to the correct implementation and shows the intended read/write behavior. Expectedly, read/write mixed load incurs additional overhead and the throughput drops. As we can see from Figure 7.17, CPU wait and user times increase during this period to manageable values. Still, request throughput is underwhelmingly low. Since the strategy the API uses for managing connections to the Triple Store is the same as with the Operational Store, the same comments above are applicable in this case.

The general magnitude of the run, as defined by the third set of metrics in Section 6.3, is the following: 2293620 datapoints, 9 tasks, of which 3 never finished, 68 datasets and 7581 triples.

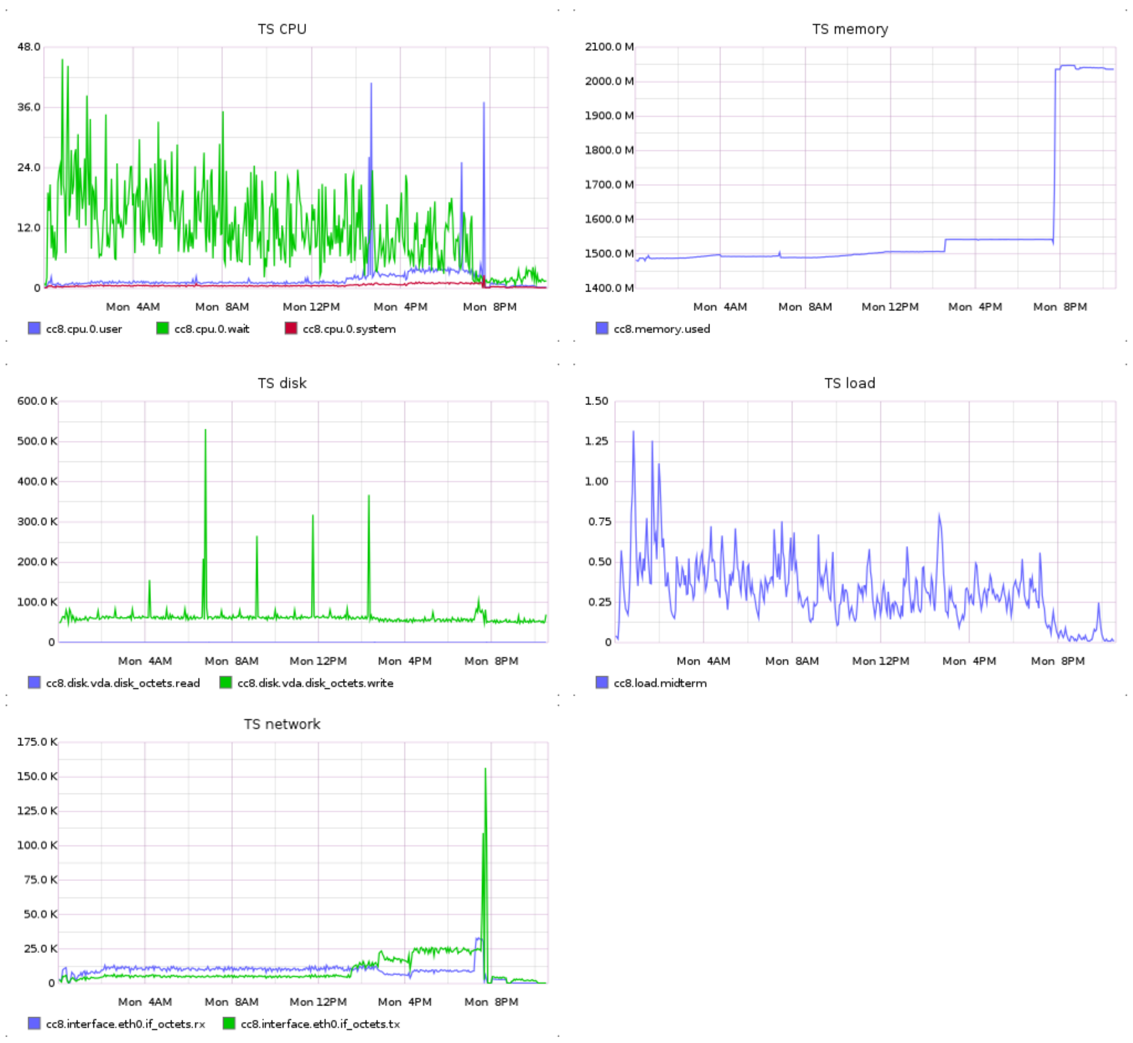


Figure 7.19: Hardware Time Series Store — third run

The relative small amount of triples stems from the fact that fewer resources were present that could be inserted as triples. The way the Triple behavior discovers resources causes this.

7.4.1 Retrospective

This test run shows results for highly specific load profiles instead of the mixed and generic load profiles used in the previous runs. This further informs us on what bottlenecks exist in some specific components of the platform. Implementation bugs surfaced in the Workers, along with other implementation details that, while not bugs *per se*, substantially reduce the performance of the platform. These constitute the main points one should address in improving the performance of the platform. During the second and third load profiles the hardware seems underutilized. This hints that more clients could consume services from the platform and still maintain quality of service. Disk contention quickly became a destabilizing factor in this test run, which stemmed

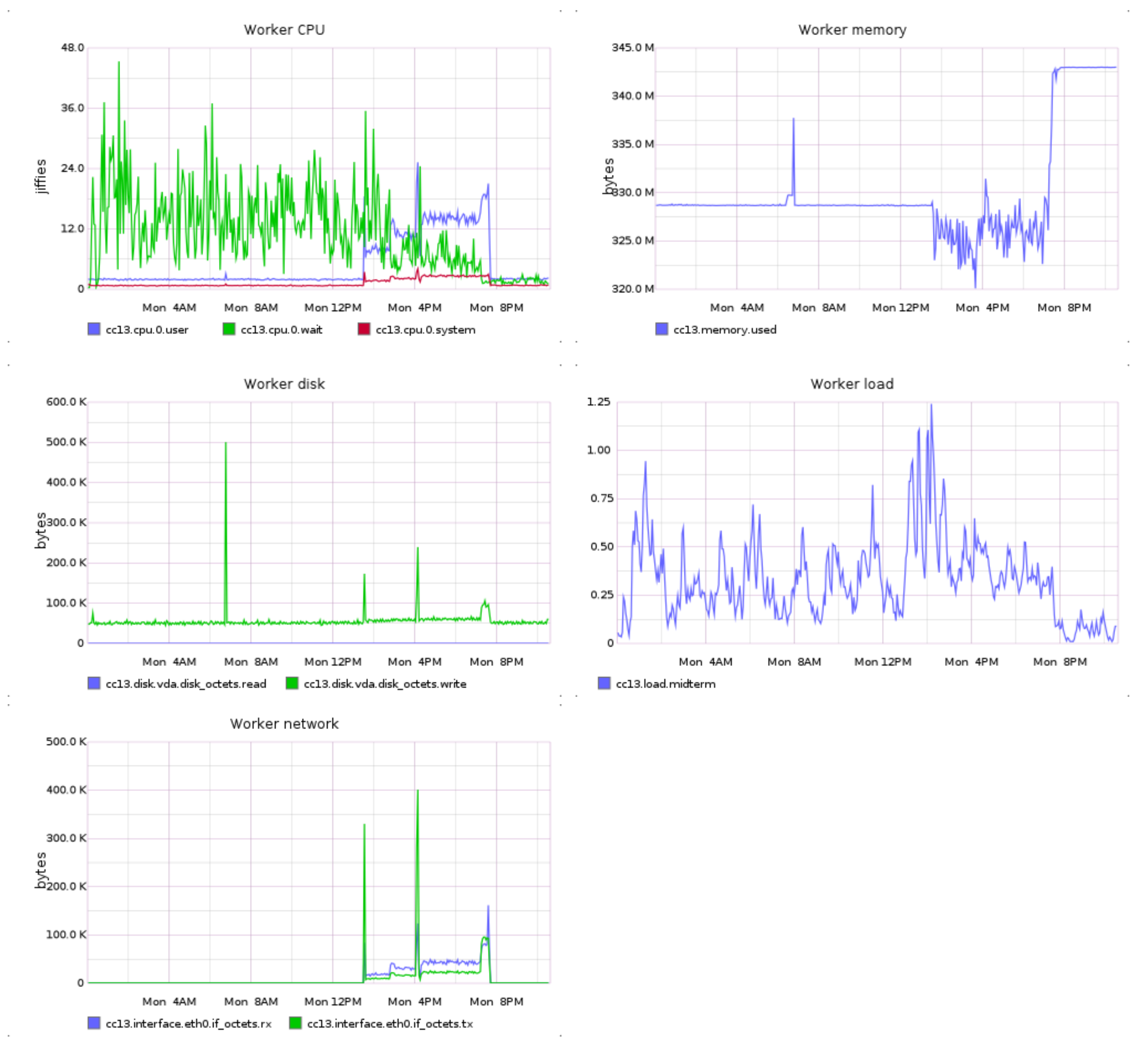


Figure 7.20: Hardware workers — third run

from an implementation detail that overwhelmed the write capacity of the disk drive and affected all other nodes.

7.5 Lessons learned

During our test runs we assessed the ability of the platform to scale its capacity and withstand different load profiles. The first conclusion we can draw is that localized optimizations have a much greater impact on performance than other strategies for increasing performance. While this was not the main purpose of this work, it is nonetheless an important aspect of distributed systems. Seemingly small optimizations can have an enormous influence in a system with complex interactions, such as this one.

The second conclusion is more relevant to our work. We found evidence that the architecture is sound and allows for scalability. We can clearly see this in the transition from the first test run

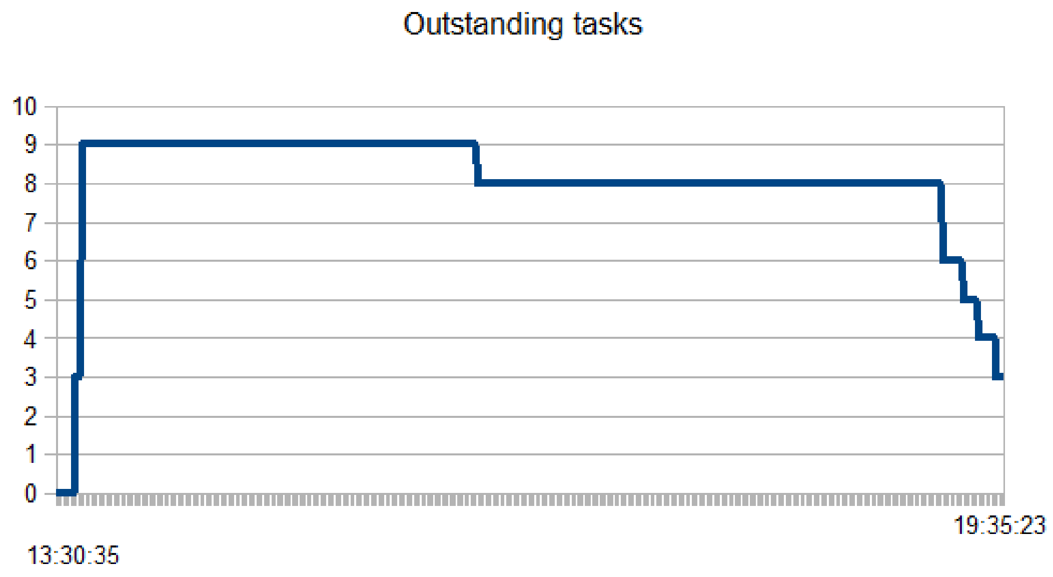


Figure 7.21: Outstanding tasks — third run

to the second when we introduced two more nodes. The performance, as measured by request throughput, increased approximately two-fold. Unfortunately,we could not test further scalability of the platform because disk contention skewed results, mainly when the Operational Store was constantly making small synchronous writes to disk. We would not attribute this behavior to the Operational Store before the third test run when the load profile of the Workers differed slightly causing a different codepath on the API to be followed. This reveals that the platform is much more performant than what one would suspect just by seeing the graphs on the first and second test runs. This phenomenon can be briefly seen in the third run where datapoint insertion increases three-fold. One detail that we did not take into account was batch inserting of datapoints. Datapoint insertion is, in our opinion, the main feature of the platform: field agents and Workers insert datapoints, raw and processed, respectively. The current implementation of the API allows for batch insertion of datapoints, but we explicitly did not implement clients (or Workers) that take advantage of this feature. This simplifies measurement because one request on the API corresponds to one datapoint. However, we suspect that batching insertions could improve throughput as fewer HTTP roundtrips would be required and the Cassandra driver provides functionality for efficiently performing many concurrent operations. Evidence exists that strongly suggest this hypothesis[15].

Another important finding was an approximation of the optimal ratio of nodes. It relates performance of the platform with the number of nodes grouped by the component they implement. As we stated before, the suggested hardware in order to sustain approximately 40 requests per second of mixed load is 3 API nodes and 1 node for each other component. All components show they are at near full capacity, with the exception of the Time Series Store which was over-provisioned

[15] http://datastax.github.io/python-driver/performance.html

and the Triple Store, which was not over-provisioned, but could handle more load. As the third test run showed, 1 Time Series Store node is enough to handle that particular load profile. This raises the question that perhaps the Time Series Store is not needed and moving the *Datapoint* resource to the Operational Store may be feasible and desirable, reducing the complexity of the whole platform. The amount of data we introduced throughout our test runs was decidedly not enough to warrant the creation of the Time Series Store, but in that case the limitations on the Operational Store would also apply to the management of the *Datapoints*. Namely, they would be limited to the highest disk capacity of the nodes in the Operational Store, write throughput would not scale as well, as per the master/slave architecture, and the more flexible table schema of Cassandra, compared to PostgreSQL, would increase the complexity of managing the dimensions associated with a datapoint/dataset. The write throughput consideration is of most concern because, as we saw throughout the test runs, overusing the Operational Store's disk will negatively impact performance system-wide.

As mentioned before, some of results we gathered were skewed and may not be particularly accurate. The most flagrant factor was disk contention on the physical host. This was mainly caused by an implementation detail, but nevertheless impeded our ability to further test the scalability of the platform. In this case, the main bottleneck to scalability was the disk drive, not the architecture itself. We could make a case that implementation details also prevented us to exploit horizontal scalability to its full potential. As we can see during the second test run, scaling the number of Time Series Store nodes did not increase performance and, in fact, reduced quality of service, as measured by datapoint insertion latency. The second implementation detail that impeded performance were the writes on the Operational Store reported on the third test run. Implementing a solution to the excessive writes on the Operational Store would be very important, as the write capacity of that store cannot be horizontally scaled using the current architecture.

We also measured the time the host took to configure a node for a particular component. The virtual machines were already running, but idling, waiting for configuration directives from the orchestration component. Again, disk contention skewed configuration times, artificially increasing them to the order of 20 minutes per node. We cannot provide concrete timings for configuration of idle nodes as a baseline because this data was not recorded, but 20 minutes per node is higher than what one would expect in order to provide fast response to increasing demand of services[16]. This is an artifact from our testing environment and we would not see it production settings where nodes do not share physical hosts, which is ideal. Should these timings be measured and found to be lacking in response time, pre-configuring disk images with templates of nodes and deploying those disk images as appropriate would improve the ability of the platform to react to sudden increase in demand. The downside of this strategy would be the need to manage the images and recreating them when configurations change, further complicated by the need to prevent configuration drift between the new nodes joining the system and the nodes already running. If the orchestration component cannot perform this synchronization effectively, a *naïve* approach would

[16]Lacking these measurements yet having performed this ourselves, we still claim that configuration takes significantly less time when the physical host is idle.

be to progressively decommission stale nodes and deploying the new disk images. Perhaps a better approach would be to run the platform by underutilizing hardware in order to provide a safety buffer for spikes in demand and commission nodes from base disk images instead, as is currently done.

Regarding the quantity of data, we could not generate enough data in order to measure "big data" behavior from the platform. This would have been desirable, since the main goal of the platform is managing extreme amounts of data. The combination of poor throughput and limited time led us to not exploring bigger data.

The final drawback to our testing was not performing a baseline test in which every component ran on a single node. This would provide important perspective on how the platform would scale from one node to several. While developing we deployed all nodes to a single, less powerful physical machine. The objective was not to perform load tests but to determine correct functionality of all the components. Still, these results would not be comparable to the ones we collected, as the physical hardware of both machines was significantly different. This topic is likely a good candidate for future studies, as it provides insight onto how segregating the components to different nodes affect performance. In our work, we assumed it would have a positive impact on performance, but cannot validate this claim.

In summary, we found that the current architecture is not the main detractor to performance, but instead the implementation details have a much greater impact on performance. This is interesting in that we validate an important point in our work, that the architecture is sound; it does not on its own impede performance. Misuse of the architecture is the main drawback to performance and we found plenty of evidence for this in all three test runs. In fact, anecdotal evidence elsewhere also suggests that this sort of optimizations can have a sizable impact on performance[17][18]. As such, we would advise future implementors of the platform to focus on localized optimizations first and then, when these no longer bear significant fruit, horizontally scale the platform by increasing nodes on the appropriate components.

[17]https://www.facebook.com/notes/facebook-engineering/hiphop-for-php-more-optimizations-for-efficient-servers/10150121348198920

[18]The relevant passage is: "Because of the sheer size of the executable, instruction cache misses become a significant factor in Web server performance.".

Chapter 8

Conclusion

This is the final chapter of this dissertation. In it, we conclude our work and reflect on what we have done and learned. In the last section we propose what future work we find most relevant for further studies.

8.1 Final remarks

In the beginning we have set out to create a scalable distributed system that could store, process and manage vast amounts of research data. This work is relevant to the scientific process in that it expedites it through automation of various tasks. This automation also causes the process to be less prone to human error, possibly increasing the quality of research. As the "Fourth Paradigm" [HTT09] posits, human researchers enter the process in later stages after data is processed via these kinds of data management platforms. Manually interpreting the amount of data generated by sensors on the field would be prohibitively expensive and slow. Researchers from various fields of study will find this platform suitable to conduct their work, with the simple restriction that the data they collect and investigate is in the form of a time series.

They will be able to manage their data and other less structured research artifacts, collectively known as "Research Objects" [DR13]. Simple management of these Research Objects is achieved implicitly via the data model we implemented. More complex and explicit management of the Research Objects comes in the form of a semantic web of connections among the Research Objects. This semantic web is achieved using an ontology to relate Research Objects to one another. It allows for fast searching of Research Objects, following ever more complex parameters so as to find the most relevant data for whichever purpose [BHBl09]. This, of course, assumes that data is open for inspection and usage by all participants of the platform. Opening up data for everyone to review, discuss and otherwise use is important for scientific progress as it allows for verification of previous results as well as providing previous data from where to launch new investigations.

This platform was designed to work as if it were a cloud service. It provides its services through an API that may be consumed by an application that implements the user interface and business logic of a particular field of study. Its main features are raw data storage and processing, using the map-reduce [DG04] paradigm. The platform will also be able to store and process the ever increasing amounts of data collected autonomously in the field, which one would colloquially refer to as “Big Data”. We instead handled a large amount of small datasets, or “long tail data”. Its mechanism to increase storage and processing capacity is by increasing the amount of hardware in the platform. Various hardware nodes play specific roles which allows the platform to accommodate an arbitrarily large number of nodes, increasing performance at each step. The platform also features a semi-automated sub-system for orchestrating and managing node configuration. It instructs nodes in what applications they should have installed and synchronizes the configuration of the applications so that the platform becomes more reliable.

During this work we tested the platform by introducing load in it. This load was artificial in nature, but did approximate production conditions. The most important finding of all was that the architecture we envisioned and implemented is sound and amenable for scalability. Further, we proposed an approximation of an optimal ratio for role distribution among the nodes. These claims are supported by the data we collected during our load tests. Throughout load testing we also came across various details that are relevant to the study of scalable systems, but were not the focus of our work. We found that seemingly small optimizations in some components can have an enormous impact on performance. This is easily accounted for by the fact that the behavior of a distributed system is comprised by the interactions of the various components and not simply the sum of the individual components.

The most important limiting factor in our work was the hardware available. We used a single physical machine and partitioned it into smaller virtual machines. The drawback is that the disk drive, incidentally the slowest component in the physical architecture of a computer, was shared among all the virtual machines. This means that if one node misbehaves and abuses IO capacity it will affect the other nodes. This is precisely what happened in our tests and constitutes increasingly “loud” noise in our data that prevented us from continuing our experiments with confidence.

In conclusion, we believe this work was a success in that the architecture we proposed is indeed effective and the resulting platform allows for faster and more reliable scientific progress.

8.2 Future work

Throughout this document we have hinted at various opportunities for improving the work or for taking different directions. In this last section we present what the most prominent opportunities are.

The platform as it is now is slightly incomplete. While we identified a passable implementation for the BLOB Store, it was not an optimal solution, as it probably could not provide enough storage capacity for its intended uses. Combined with the inefficient use of storage space, this component can certainly benefit for more focused study. We have previously suggested that distributed file systems could be an answer, but other data partitioning and load balancing strategies should be explored.

The load profiles and data we used during our experiments were artificial in nature. This means that the results we gathered may not approximate production usage, although we have attempted to mimic those scenarios. In future studies we suggest that load profiles be gathered from real usage and the behaviors updated or otherwise reworked. The data we used was not generated from any real source, be it from the WindScanner.eu project or others. We instead generated mock data which could conceivably have been collected in the field. It is simple, however, and may not be representative of real data in terms of complexity. The same can be said bout the procedures applied to the data. These are in fact real procedures that a researcher may want to apply to their data, in particular the 20-second average. The other procedures that convert between units of measurement probably do not warrant a procedure of their own in the platform, but are nonetheless valid. They illustrate a pattern we expect to see in production: that procedures are CPU bound. We suggest that future studies should be carried out in closer proximity to specialists of a given field, so as to gather real data and implement real procedures.

During the test runs we alluded to (and fixed) some inefficiencies with various parts of the code we implemented. We have subsequently observed that these fixes had a greater impact on performance than expected. We propose that future implementations review their code and closely monitor the interactions between the components. Monitoring the logs of the various applications in use proves useful to diagnose erroneous behavior. In our implementation log inspection was done manually which quickly becomes morose and tedious. In addition, the applications we implemented logged information *ad hoc* and sometimes no useful information could be extracted. We suggest that a centralized logging facility be implemented with all applications routing their logs to it. The applications implemented by the maintainers of the platform, namely the API and Workers, should be reviewed and a logging structure be put into place, keeping relevance.

With the current implementation, any resource that the API accepts for creation is done with one resource per HTTP response. We propose that performance can be increased if batch insertions by the clients are fully supported. In particular, inserting datapoints using this strategy is a good idea for further investigation.

In our work we proposed a way for the various applications to discover services and balance the load among the nodes. Whereas the most common pattern for this is having a centralized load balancer and application router, we instead decentralized the load balancers and routers. We implemented them in every node that consumes remote services: for example, the API nodes consume from the Operational, Triple and Time Series Stores, as well as from the Job Servers. In this case every API node runs an instance of the particular router software that provides access to all those services, with their configurations being synchronized by Fabric. While this is a novel approach we did not encounter anywhere else, there are other ways to solve service discovery. One example is using DNS SRV records[1]. This strategy is reported to be effective[2]. It uses familiar, solid technology and it trades the need to manually synchronize the configurations of the routers for some additional complexity on the platform, maintaining load balancing while still not having

[1]`http://tools.ietf.org/html/rfc2782`

[2]`http://labs.spotify.com/2013/02/25/in-praise-of-boring-technology/`

a SPOF. Also, the clients will need little or no modification in order to support this alternative. Reportedly, other solutions exist[3]. We suggest future studies should identify and experiment with other service discovery strategies, so as to ascertain their merits and drawbacks.

A change we have hinted at before is absorbing the Job Server node type into another node type. As they are implemented now, the node barely uses its assigned resources. This would reduce the complexity of the platform and ease service discovery. We suggest this component be integrated into either the Workers or the API, although we give preference to the Workers. In this way, assuming Gearman is used, the API would contact a Worker and assign it a job, essentially becoming a load balanced, asynchronous Remote Procedure Call (RPC). Configurations on the Worker would be simpler (because they now consume from *localhost*), with the difference that they should now schedule mapped jobs to the API, which then re-assigns them elsewhere, instead of directly scheduling them on the job server which now resides in *localhost* and the only node consuming from that particular job server is itself. If the job servers reside in the API nodes the Workers would need to pull jobs, instead of being pushed as in the previous case. Depending on the service discovery strategy in use jobs might become enqueued for an unspecified amount of time, should no Worker pull jobs from that particular API node ever or otherwise in an unacceptable time frame. Contrast with the above option in that jobs are immediately queued and their completion time known or estimable (barring erroneous conditions).

In the same spirit as this last suggestion, we propose that Gearman be substituted with Celery and RabbitMQ, as initially contemplated. This change would void the need for our implementation of job synchronization and provide a more powerful method of describing and implementing workflows. It would certainly increase platform stability and features at virtually no cost, other than porting the code. The initial cause of our change to Gearman was that we were misusing either the Cassandra driver or the Celery process model. We decided, for our work, that the process model should be replaced; Cassandra access was possible, albeit with unacceptable quality of service. Additionally, replacing Celery was feasible, whereas replacing Cassandra, given we had already decided on the data model and implemented the API, was not an option according to our time frames. However, while developing the Worker code, we found that we should write to Cassandra using the API, since that particular codepath was already implemented there. The code on the Workers to read from Cassandra was implemented before that and read directly from it, bypassing the API (which, again, already had that code implemented). We then strongly propose that the Worker code be modified to take advantage of the code already implemented on the API, in particular read access to Cassandra. This represents an opportunity for some considerable changes. The Workers would no longer have a dependency on the Cassandra driver. This solves our problem for implementing Celery, dramatically increasing our ability to schedule complex workflows and, importantly, void the need for our synchronization features which were the source of much of the implementation errors in our work. With the current service discovery and load balancing strategy, it would also entirely eliminate the need for the Workers to have a local instance of HAProxy for contacting the Time Series Store. Instead, our workaround for contacting the API

[3] `http://jasonwilder.com/blog/2014/02/04/service-discovery-in-the-cloud/`

with nginx would become the *de facto* solution, since the Workers only need to load balance the API. Contacting RabbitMQ would either be made with a hardcoded configuration (as it is now; if Celery is amenable to multiple brokers) or implicitly if RabbitMQ resides in *localhost*.

The final and promptly identified improvement we suggest is implementing autoscale features. This means that the platform should be able to introspect its own internal state and performance and commission or decommission computing resources as demand for its services fluctuates. We concede that this is not trivial to implement and substantial effort must be made towards it. Nevertheless, continuously automating the orchestration of the platform will further increase its reliability and reduce its operational costs. The first steps would be to refactor the configuration scripts to not being interactive as this requires operators to supervise the process of adding nodes. Then, when a node is added, further automation is possible by automatically updating the relevant configurations in other nodes. The final step would be for the software to introspect the performance and demand of the platform's resources and, when certain thresholds are reached, it would take action and commission or decommission a node of the appropriate type. These thresholds have been hinted at throughout our load tests, but we provided no specific numbers in order to guide the heuristics behind the process besides the approximation of the optimal ratio. We suggest that, for now, the process of adding or removing a node from the network be initiated by the operator, but the process of configuring the new node and updating the existing nodes should be automatic. Using Fabric in this scenario becomes particularly morose because this features needs to be implemented, whereas with other more sophisticated solutions this process of updating the configurations is more autonomous. While Fabric is not at all unsuitable, we suggest that alternatives to it be studied and implemented and their merits assessed.

All these suggestions and improvements need to be made collectively and not individually. Implementing individual improvements might not always be possible as sometimes they depend on other improvements or the ones that not have any dependencies may not integrate as well with other parts of the system, resulting in less synergy among the components. As we previously remarked, in distributed system it is the interactions of the components that is important; the whole of the system is much more intricate than the sum of its parts.

References

[Ber08] Francine Berman. Got data?: a guide to data preservation in the information age. *Communications of the ACM, 51(12):50–56*, June 2008.

[BHBl09] Christian Bizer, Tom Heath, and Tim Berners-lee. Linked Data – The Story So Far. *International Journal on Semantic Web and Information Systems*, 5(3 (Special Issue on Linked Data)):1–22, 2009.

[BK12] Florian Bauer and Martin Kaltenböck. *Linked Open Data: The Essentials – A Quick Start Guide for Decision Makers*. The Semantic Web Company; Renewable Energy and Energy Efficiency Partnership, 2012.

[Boh13] Shannon Bohle. What is E-science and How Should it be Managed? – Scientific and Medical Libraries, 2013. http://www.scilogs.com/scientific_and_medical_libraries/what-is-e-science-and-how-should-it-be-managed/, last accessed on 2014-01-24.

[Bor11] Christine L. Borgman. The Conundrum of Sharing Research Data. *Journal of the American Society for Information Science and Technology*, pages 1–40, June 2011.

[CDG+08] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[Coh04] Mike Cohn. *User stories applied: For agile software development*. Addison-Wesley Professional, 2004.

[CR06] Tony Casson and Patrick S. Ryan. Open Source Adoption in the Public Sector, and Their Relationship to Microsoft's Market Dominance. *Standards Edge: Unifier or Divider, Sherrie Bolin, ed., Sheridan Books*, pages 87–99, 2006.

[De 09] David De Roure. e-Research. Replacing the Paper: the six Rs of the e-Research Record, 2009. http://blog.openwetware.org/deroure/?p=56, last accessed on 2014-01-29.

[DG04] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplied Data Processing on Large Clusters. In *Proc of 6th Symposium on Operating Systems Design and Implementation*, pages 137–149, 2004.

[DHJ+07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 205–220. ACM, 2007.

[DR13] David De Roure. Towards computational research objects. In *Proceedings of the 1st International Workshop on Digital Preservation of Research Methods and Artefacts*, DPRMA '13, pages 16–19, New York, NY, USA, 2013. ACM.

[FGJ$^+$09] Armando Fox, Rean Griffith, A Joseph, R Katz, A Konwinski, G Lee, D Patterson, A Rabkin, and I Stoica. Above the clouds: A Berkeley view of cloud computing. *Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS*, 28:13, 2009.

[GGL03] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. *ACM SIGOPS Operating Systems Review*, 37(5):29, 2003.

[GLPR14] Filipe Gomes, João Correia Lopes, José Laginha Palma, and Luís Frölen Ribeiro. WindS@UP: The e-Science Platform for WindScanner.eu. *Journal of Physics: Conference Series.*, 524(The Science of Making Torque from Wind 2014 (TORQUE 2014) 18–20 June 2014, Copenhagen, Denmark), 2014.

[Gol11] Yaron Goland. Stuff Yaron Finds Interesting, 2011. http://www.goland.org/average_percentile_services/, last accessed on 2014-05-21.

[Gri13] Ilya Grigorik. High Performance Browser Networking, 2013. http://chimera.labs.oreilly.com/books/1230000000545/ch10.html, last accessed on 2014-05-12.

[HAMS08] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. OLTP through the looking glass, and what we found there. *Proceedings of the 2008 ACM SIGMOD international conference on Management of data - SIGMOD'08*, page 981, 2008.

[HKM09] Adrian Holovaty and Jacob Kaplan-Moss. The Django Book, 2009. http://www.djangobook.com/en/2.0/chapter12.html, last accessed on 2014-05-12.

[HP11] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[HT02] Tony Hey and Anne E Trefethen. The UK e-science core programme and the grid. *Future Generation Computer Systems*, 18(8):1017–1031, 2002.

[HTT09] Tony Hey, Stewart Tansley, and Kristin Tolle. Jim Gray on eScience: a transformed scientific method. In Tony Hey, Stewart Tansley, and Kristin Tolle, editors, *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Research, Redmond, 2009.

[KR14] James Keene and Jonathan Reichental. Open Data – City of Palo Alto, 2014. http://data.cityofpaloalto.org/, last accessed on 2014-02-09.

[LG13] João Correia Lopes and Filipe Gomes. Establishment of methodologies for data quality assurance and exchange. Technical report, WindScanner.eu project deliverable D5.5, October 2013.

[LS10] Feng-Tse Lin and Teng-San Shih. Cloud computing: the emerging computing technology. *ICIC Express Letters*, 1(1):33–38, 2010.

[M$^+$98] Gordon E Moore et al. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85, 1998.

[MMSW07] M. Michael, J.E. Moreira, D. Shiloach, and R.W. Wisniewski. Scale-up x Scale-out: A Case Study using Nutch/Lucene. *2007 IEEE International Parallel and Distributed Processing Symposium*, 2007.

[MSS$^+$12] Torben Mikkelsen, Søren Siggaard Knudsen, Mikael Sjöholm, Nikolas Angelou, and Anders Tegtmeier. WindScanner.eu - a new Remote Sensing Research Infrastructure for On- and Offshore Wind Energy. In *International Conference on Wind Energy: Materials, Engineering, and Policies (WEMEP-2012, Hyderabad, India, 22-23 November)*, 2012.

[PDF07] Heather A. Piwowar, Roger S. Day, and Douglas B. Fridsma. Sharing detailed research data is associated with increased citation rate. *PLoS ONE*, 2(3):e308, 03 2007.

[Sta13] Richard Stallman. Why Open Source Misses the Point of Free Software – GNU Project – Free Software Foundation, 2013. http://www.gnu.org/philosophy/open-source-misses-the-point.html, last accessed on 2014-02-07.

[Str10] Christof Strauch. NoSQL Databases. 2010. unpublished, http://home.aubg.bg/students/ENL100/Cloud%20Computing/Research%20Paper/nosqldbs.pdf, last accessed on 2014-02-05.

[Wil10] Dominic Williams. HBase vs Cassandra: why we moved, 2010. http://ria101.wordpress.com/2010/02/24/hbase-vs-cassandra-why-we-moved/, last accessed on 2014-02-06.

REFERENCES

Appendix A

Application usage

Examples of usage of the API may be consulted in this appendix. In the first section we present the administration application which manipulates data in the Operational and Triple Stores. The second section illustrates the interface that documents the API and enables its users to interact directly with it. The last section mentions an *addendum* to the API which could not, at this time, be illustrated with the application that generates the documentation.

A.1 Administration interface

In this section we present the most representative figures regarding the administration interface.

Figure A.1 shows the dashboard of the administration application. It is automatically generated by Django and is intended for use by the maintainers of the platform. It provides direct access to the tables defined in the Operational and Triple Stores, grouped by the *apps* they belong to. The figures we present throughout this section are the most representative of this interface.

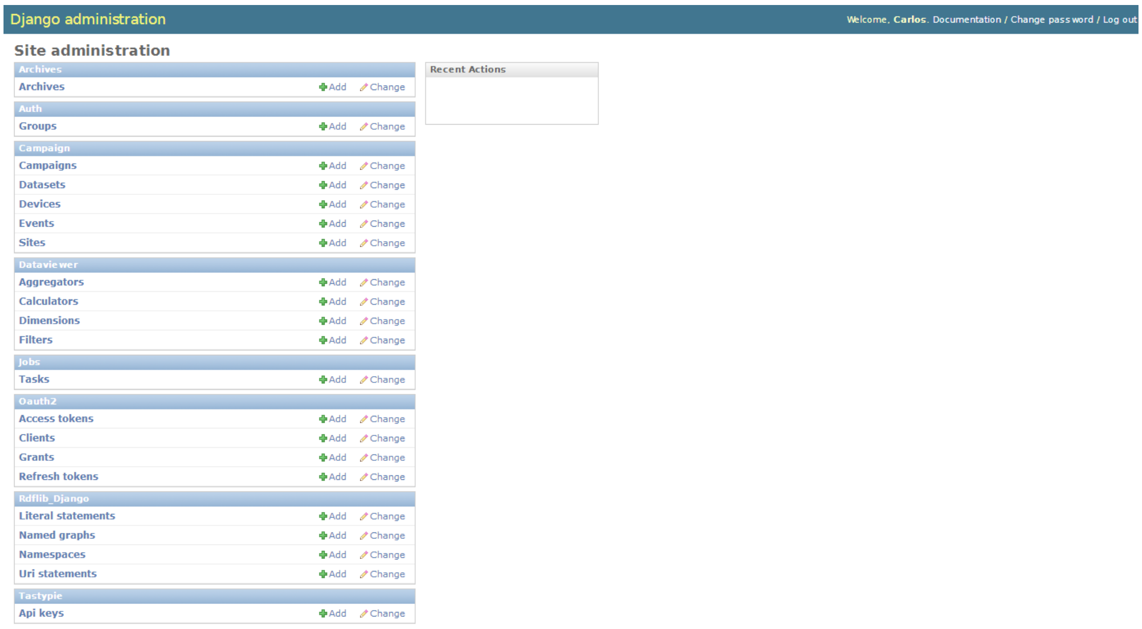
Figure A.2 shows the list of existent Campaigns. Note the button on the top-right corner which allows the administrator to create new instances of Campaigns.

Figure A.3 shows the list of defined Dimensions.

Figure A.4 shows the list of Datasets. Some datasets are named as “raw” or “derived” data, but this is only to increase understandability of the role of the dataset. Intentionally, there is no attribute that marks a dataset as “derived”; that task is left to the semantic connections.

Figure A.5 shows the web form that allows the administrator to edit the information regarding a specific dataset. Note that the fields of the form coincide with the columns of the table that define the Dataset.

Figure A.6 shows the list of tasks the clients have scheduled on the platform. The “task” field denotes which procedure should be applied and which procedure is its ancestor. For example, the last task is */api/v1/aggregator/1/ - root task* which means that the procedure is */api/v1/aggregator/1/* and that it is the root task; there is no task before it. The task immediately above is procedure */api/v1/calculator/1/* and has the parent */api/v1/aggregator/1/*, which is the one directly below. The fourth task is the last task in that pipeline, */api/v1/calculator/2/* and its parent is the previously


Django administration

Welcome, Carlos. Documentation / Change password / Log out

Site administration

Archives		
Archives	Add	Change

Auth		
Groups	Add	Change

Campaign		
Campaigns	Add	Change
Datasets	Add	Change
Devices	Add	Change
Events	Add	Change
Sites	Add	Change

Dataviewer		
Aggregators	Add	Change
Calculators	Add	Change
Dimensions	Add	Change
Filters	Add	Change

Jobs		
Tasks	Add	Change

Oauth2		
Access tokens	Add	Change
Clients	Add	Change
Grants	Add	Change
Refresh tokens	Add	Change

Rdflib_Django		
Literal statements	Add	Change
Named graphs	Add	Change
Namespaces	Add	Change
Uri statements	Add	Change

Tastypie		
Api keys	Add	Change

Recent Actions

Figure A.1: Administration dashboard

mentioned procedure */api/v1/aggregator/1/*. The first three tasks comprise a different instance of the same sequence of instructions applied to a different set of datasets.

Figure A.7 shows the list of triples and its respective context that are present in the Triple Store. Note that the "predicate" is a link to another resource in the Operational Store that does not represent a useful predicate. The rationale for inserting useless predicates has been explained elsewhere.

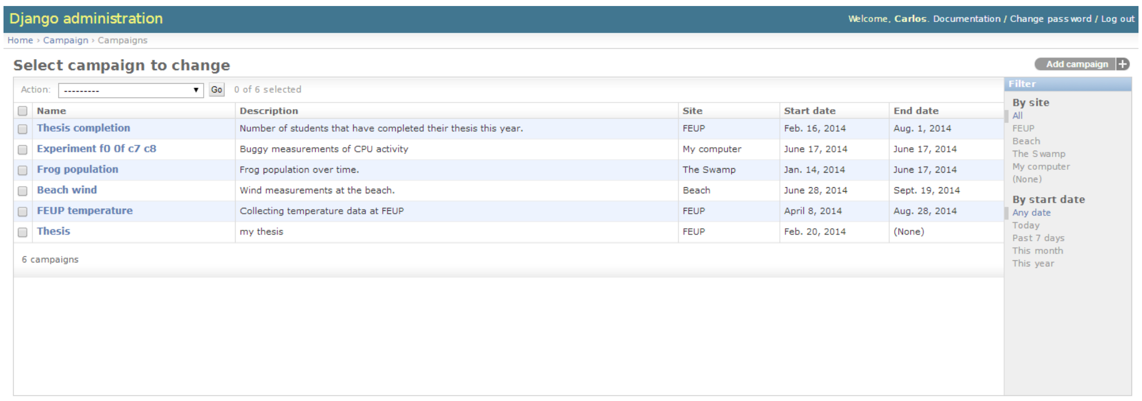
A.2 API interactive interface

In this section we present the most representative figures regarding the API interactive interface. We used django-tastypie-swagger[1] to automate the generation of the documentation. Using this interface and consulting the documentation for the API is simply a matter of installing a mock platform with just an API node. For instructions on how to do this, refer to Annex B.

Figure A.8 shows the dashboard of the API web interface. All the endpoints are visible and two of them are expanded. The expansion only reveals the HTTP methods that the API support on those particular endpoints/resources. In this case, Datapoints may not be retrieved by requesting via GET on the */datapoint* endpoint.

Figure A.9 illustrates the expansion of the POST action on the Campaign resource. It presents to the user the documentation of this particular endpoint: a small description and the attributes and data types that make up the resource. The user may also fill in the form with relevant information and send it to be processed by the API. In this case we filled the input field with information

[1] https://github.com/concentricsky/django-tastypie-swagger



Select campaign to change

0 of 6 selected

Name	Description	Site	Start date	End date
Thesis completion	Number of students that have completed their thesis this year.	FEUP	Feb. 16, 2014	Aug. 1, 2014
Experiment f0 0f c7 c8	Buggy measurements of CPU activity	My computer	June 17, 2014	June 17, 2014
Frog population	Frog population over time.	The Swamp	Jan. 14, 2014	June 17, 2014
Beach wind	Wind measurements at the beach.	Beach	June 28, 2014	Sept. 19, 2014
FEUP temperature	Collecting temperature data at FEUP	FEUP	April 8, 2014	Aug. 28, 2014
Thesis	my thesis	FEUP	Feb. 20, 2014	(None)

6 campaigns

Figure A.2: Campaign list

regarding a hypothetical Campaign and submitted it. The "site" key in the JSON string is the canonical URI for the Site resource to which this campaign should be associated.

Figure A.10 shows the result of submitting the POST request in Figure A.9. Note that in the headers of the response is returned the canonical URI for the resource that was just created. It is in the header "Location".

Figure A.11 shows the result of performing a GET request relative to the dataset of id 1. Again, the description of the resource is given above and the response may be seen below. For brevity, the whole list of its datapoints is partially omitted. Note that the resource shows its list of Dimensions and related Site (related through Campaign) are shown with their canonical URI for easier inspection of those resources.

A.3 Addendum

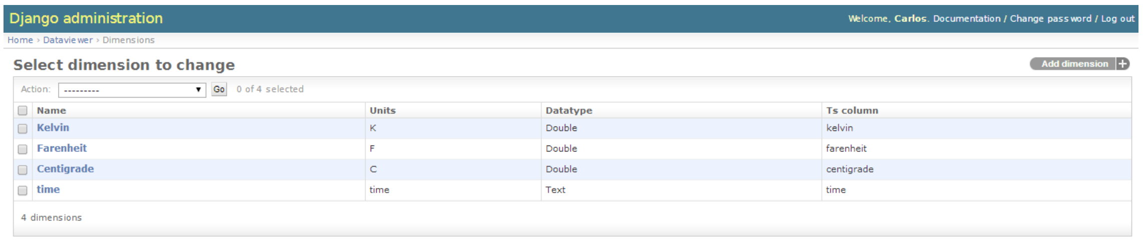
In Section A.2 we presented some parts of the API and its documentation. We introduced a feature to limit the range of datapoints each dataset should return. This feature could not be integrated cleanly with the documentation tool we used and we must make reference to it.

The */api/v1/dataset/{id}* endpoint accepts two optional URL parameters: "upper_time" and "lower_time". They bound the range of the datapoints returned with "upper_time" being the upper (most recent) time and the "lower_time" being the lower (most ancient) time.

An example of usage is the following:

```
wget http://myplatform.com/api/v1/dataset/1/?format=json&
        upper_time=2014-06-17%2017:03:12.669632&
        lower_time=2014-06-17%2017:03:02.669632
```

The response is in every respect similar to any other request on this resource with the exception that the list of datapoints returned is only the one in the range of "upper_time" and "lower_time", both inclusive. Note that the "%20" characters represent URL encoded spaces. For clarity, we have shown the URI using line breaks, but in actual usage no line breaks should be used.


Django administration
Welcome, Carlos. Documentation / Change password / Log out
Home › Dataviewer › Dimensions
Add dimension
Action: --------- Go

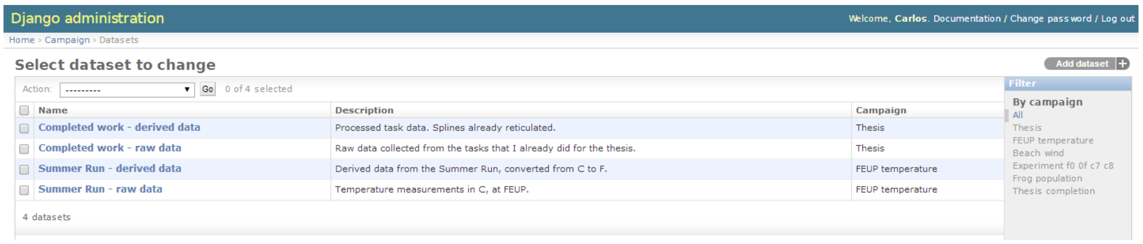
Select dimension to change

0 of 4 selected

Name	Units	Datatype	Ts column
Kelvin	K	Double	kelvin
Farenheit	F	Double	farenheit
Centigrade	C	Double	centigrade
time	time	Text	time

4 dimensions

Figure A.3: Dimension list



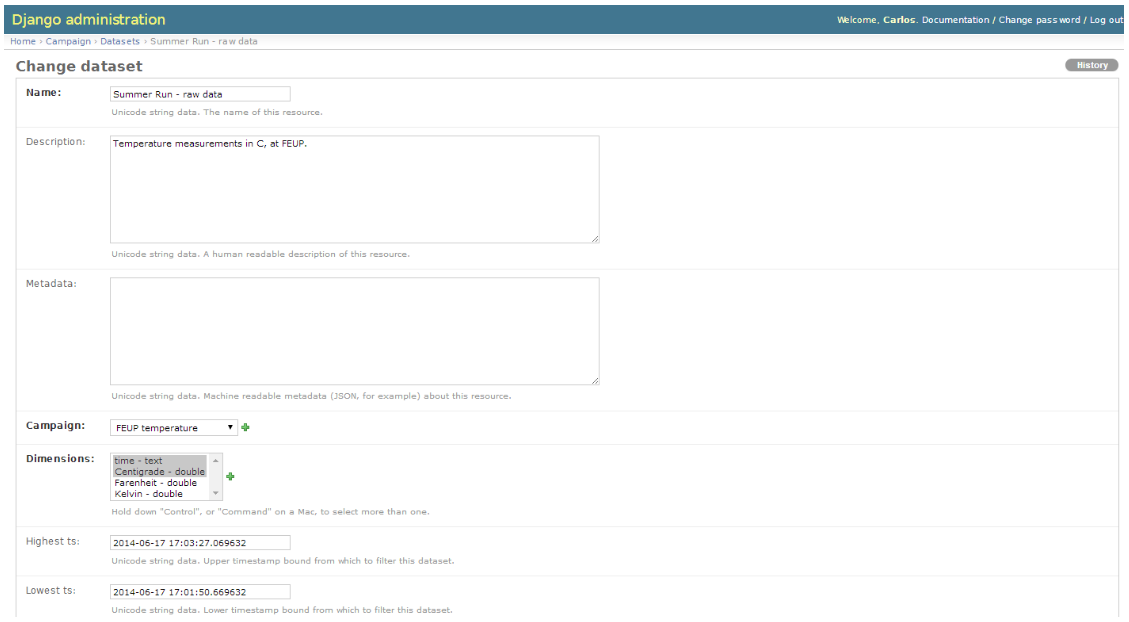
Select dataset to change

0 of 4 selected

Name	Description	Campaign
Completed work - derived data	Processed task data. Splines already reticulated.	Thesis
Completed work - raw data	Raw data collected from the tasks that I already did for the thesis.	Thesis
Summer Run - derived data	Derived data from the Summer Run, converted from C to F.	FEUP temperature
Summer Run - raw data	Temperature measurements in C, at FEUP.	FEUP temperature

4 datasets

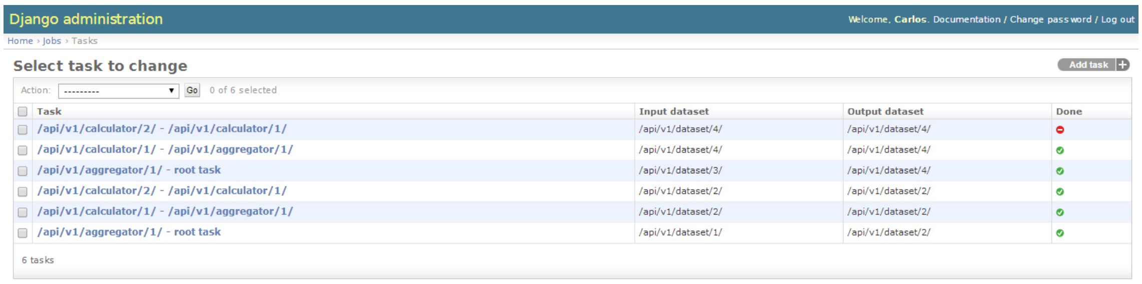
Figure A.4: Dataset list


Django administration
Welcome, Carlos. Documentation / Change password / Log out
Home › Campaign › Datasets › Summer Run - raw data
History

Change dataset

Name:	Summer Run - raw data Unicode string data. The name of this resource.
Description:	Temperature measurements in C, at FEUP. Unicode string data. A human readable description of this resource.
Metadata:	Unicode string data. Machine readable metadata (JSON, for example) about this resource.
Campaign:	FEUP temperature
Dimensions:	time - text Centigrade - double Farenheit - double Kelvin - double Hold down "Control", or "Command" on a Mac, to select more than one.
Highest ts:	2014-06-17 17:03:27.069632 Unicode string data. Upper timestamp bound from which to filter this dataset.
Lowest ts:	2014-06-17 17:01:50.669632 Unicode string data. Lower timestamp bound from which to filter this dataset.

Figure A.5: Dataset detail



Django administration

Welcome, **Carlos**. Documentation / Change password / Log out

Home › Jobs › Tasks

Select task to change

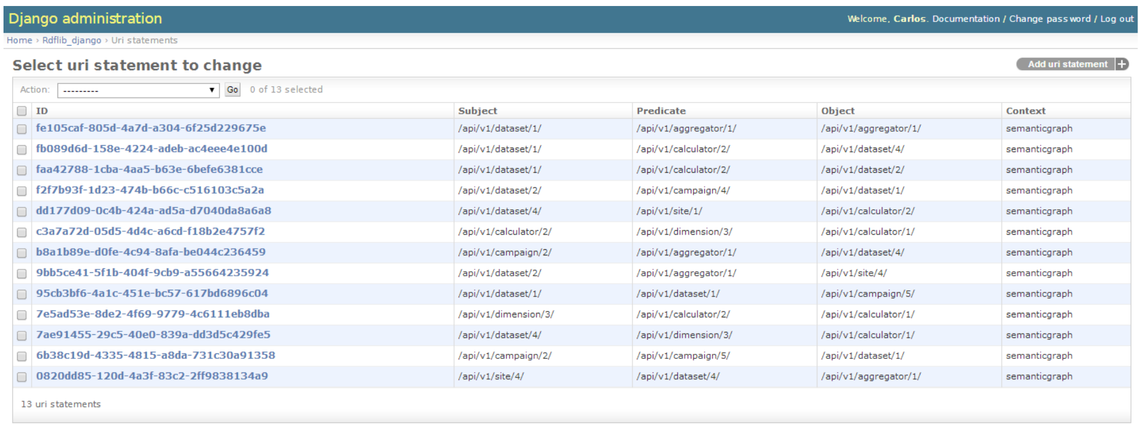
Add task +

Action: --------- Go 0 of 6 selected

Task	Input dataset	Output dataset	Done
/api/v1/calculator/2/ - /api/v1/calculator/1/	/api/v1/dataset/4/	/api/v1/dataset/4/	⛔
/api/v1/calculator/1/ - /api/v1/aggregator/1/	/api/v1/dataset/4/	/api/v1/dataset/4/	✓
/api/v1/aggregator/1/ - root task	/api/v1/dataset/3/	/api/v1/dataset/4/	✓
/api/v1/calculator/2/ - /api/v1/calculator/1/	/api/v1/dataset/2/	/api/v1/dataset/2/	✓
/api/v1/calculator/1/ - /api/v1/aggregator/1/	/api/v1/dataset/2/	/api/v1/dataset/2/	✓
/api/v1/aggregator/1/ - root task	/api/v1/dataset/1/	/api/v1/dataset/2/	✓

6 tasks

Figure A.6: Task list



Django administration

Welcome, **Carlos**. Documentation / Change password / Log out

Home › Rdflib_django › Uri statements

Select uri statement to change

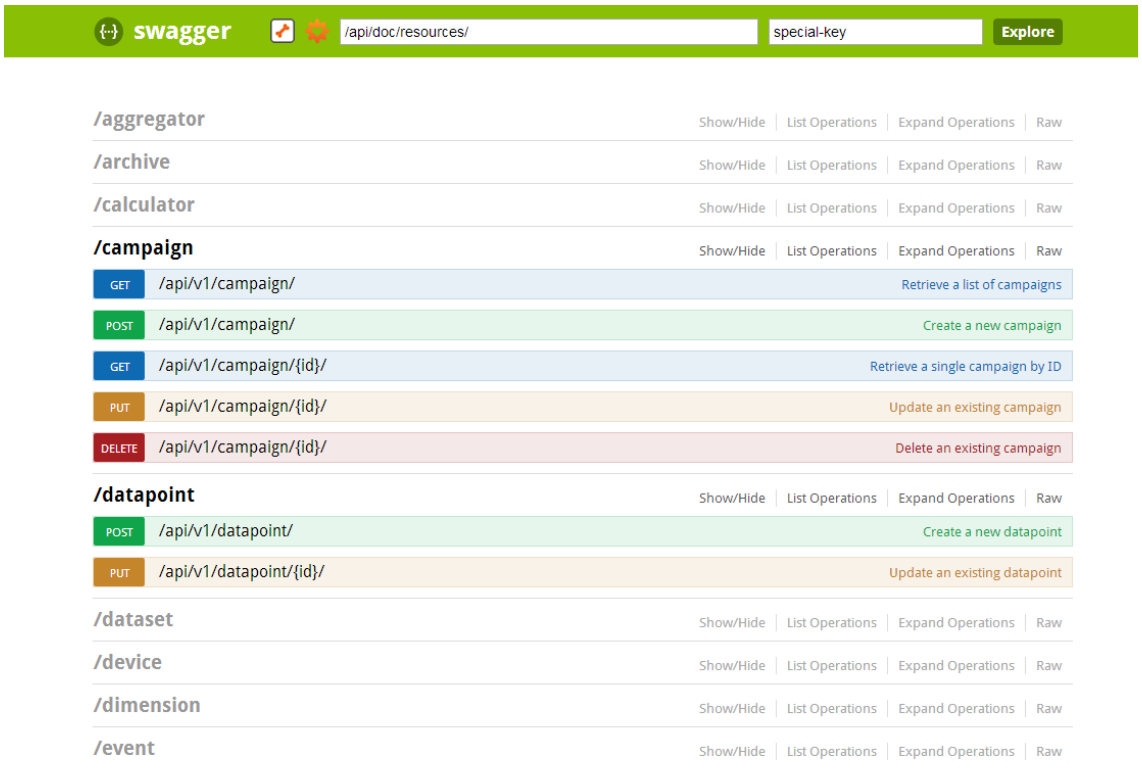
Add uri statement +

Action: --------- Go 0 of 13 selected

ID	Subject	Predicate	Object	Context
fe105caf-805d-4a7d-a304-6f25d229675e	/api/v1/dataset/1/	/api/v1/aggregator/1/	/api/v1/aggregator/1/	semanticgraph
fb089d6d-158e-4224-adeb-ac4eee4e100d	/api/v1/dataset/1/	/api/v1/calculator/2/	/api/v1/dataset/4/	semanticgraph
faa42788-1cba-4aa5-b63e-6befe6381cce	/api/v1/dataset/1/	/api/v1/calculator/2/	/api/v1/dataset/2/	semanticgraph
f2f7b93f-1d23-474b-b66c-c516103c5a2a	/api/v1/dataset/2/	/api/v1/campaign/4/	/api/v1/dataset/1/	semanticgraph
dd177d09-0c4b-424a-ad5a-d7040da8a6a8	/api/v1/dataset/4/	/api/v1/site/1/	/api/v1/calculator/2/	semanticgraph
c3a7a72d-05d5-4d4c-a6cd-f18b2e4757f2	/api/v1/calculator/2/	/api/v1/dimension/3/	/api/v1/calculator/1/	semanticgraph
b8a1b89e-d0fe-4c94-8afa-be044c236459	/api/v1/campaign/2/	/api/v1/aggregator/1/	/api/v1/dataset/4/	semanticgraph
9bb5ce41-5f1b-404f-9cb9-a55664235924	/api/v1/dataset/2/	/api/v1/aggregator/1/	/api/v1/site/4/	semanticgraph
95cb3bf6-4a1c-451e-bc57-617bd6896c04	/api/v1/dataset/1/	/api/v1/dataset/1/	/api/v1/campaign/5/	semanticgraph
7e5ad53e-8de2-4f69-9779-4c6111eb8dba	/api/v1/dimension/3/	/api/v1/calculator/2/	/api/v1/calculator/1/	semanticgraph
7ae91455-29c5-40e0-839a-dd3d5c429fe5	/api/v1/dataset/4/	/api/v1/dimension/3/	/api/v1/calculator/1/	semanticgraph
6b38c19d-4335-4815-a8da-731c30a91358	/api/v1/campaign/2/	/api/v1/campaign/5/	/api/v1/dataset/1/	semanticgraph
0820dd85-120d-4a3f-83c2-2ff9838134a9	/api/v1/site/4/	/api/v1/dataset/4/	/api/v1/aggregator/1/	semanticgraph

13 uri statements

Figure A.7: Triple list



/aggregator Show/Hide | List Operations | Expand Operations | Raw

/archive Show/Hide | List Operations | Expand Operations | Raw

/calculator Show/Hide | List Operations | Expand Operations | Raw

/campaign Show/Hide | List Operations | Expand Operations | Raw

GET	/api/v1/campaign/	Retrieve a list of campaigns
POST	/api/v1/campaign/	Create a new campaign
GET	/api/v1/campaign/{id}/	Retrieve a single campaign by ID
PUT	/api/v1/campaign/{id}/	Update an existing campaign
DELETE	/api/v1/campaign/{id}/	Delete an existing campaign

/datapoint Show/Hide | List Operations | Expand Operations | Raw

POST	/api/v1/datapoint/	Create a new datapoint
PUT	/api/v1/datapoint/{id}/	Update an existing datapoint

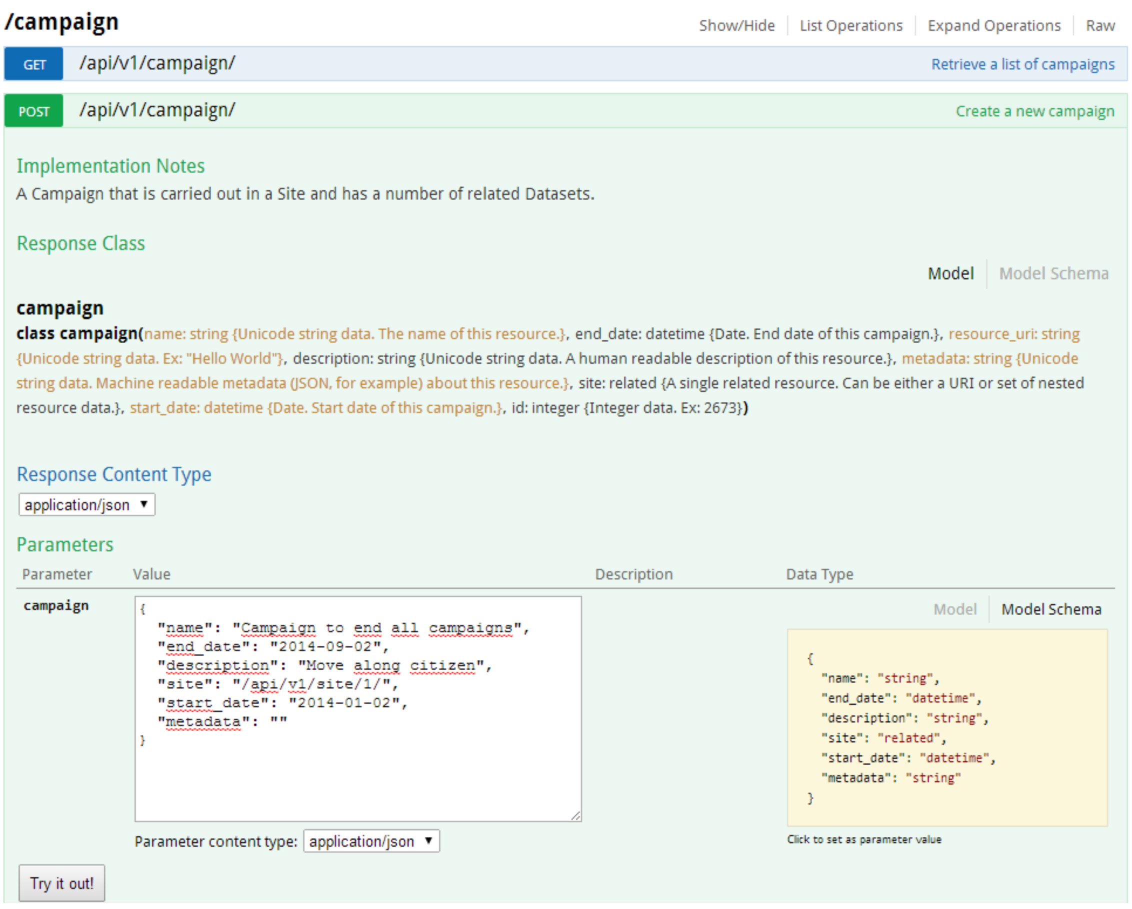
/dataset Show/Hide | List Operations | Expand Operations | Raw

/device Show/Hide | List Operations | Expand Operations | Raw

/dimension Show/Hide | List Operations | Expand Operations | Raw

/event Show/Hide | List Operations | Expand Operations | Raw

Figure A.8: API dashboard



/campaign

Show/Hide | List Operations | Expand Operations | Raw

GET /api/v1/campaign/ Retrieve a list of campaigns

POST /api/v1/campaign/ Create a new campaign

Implementation Notes

A Campaign that is carried out in a Site and has a number of related Datasets.

Response Class

Model | Model Schema

campaign

class campaign(name: string {Unicode string data. The name of this resource.}, end_date: datetime {Date. End date of this campaign.}, resource_uri: string {Unicode string data. Ex: "Hello World"}, description: string {Unicode string data. A human readable description of this resource.}, metadata: string {Unicode string data. Machine readable metadata (JSON, for example) about this resource.}, site: related {A single related resource. Can be either a URI or set of nested resource data.}, start_date: datetime {Date. Start date of this campaign.}, id: integer {Integer data. Ex: 2673}**)**

Response Content Type

application/json ▼

Parameters

Parameter	Value	Description	Data Type
campaign	see below		see below

Value:

```
{
  "name": "Campaign to end all campaigns",
  "end_date": "2014-09-02",
  "description": "Move along citizen",
  "site": "/api/v1/site/1/",
  "start_date": "2014-01-02",
  "metadata": ""
}
```

Parameter content type: application/json ▼

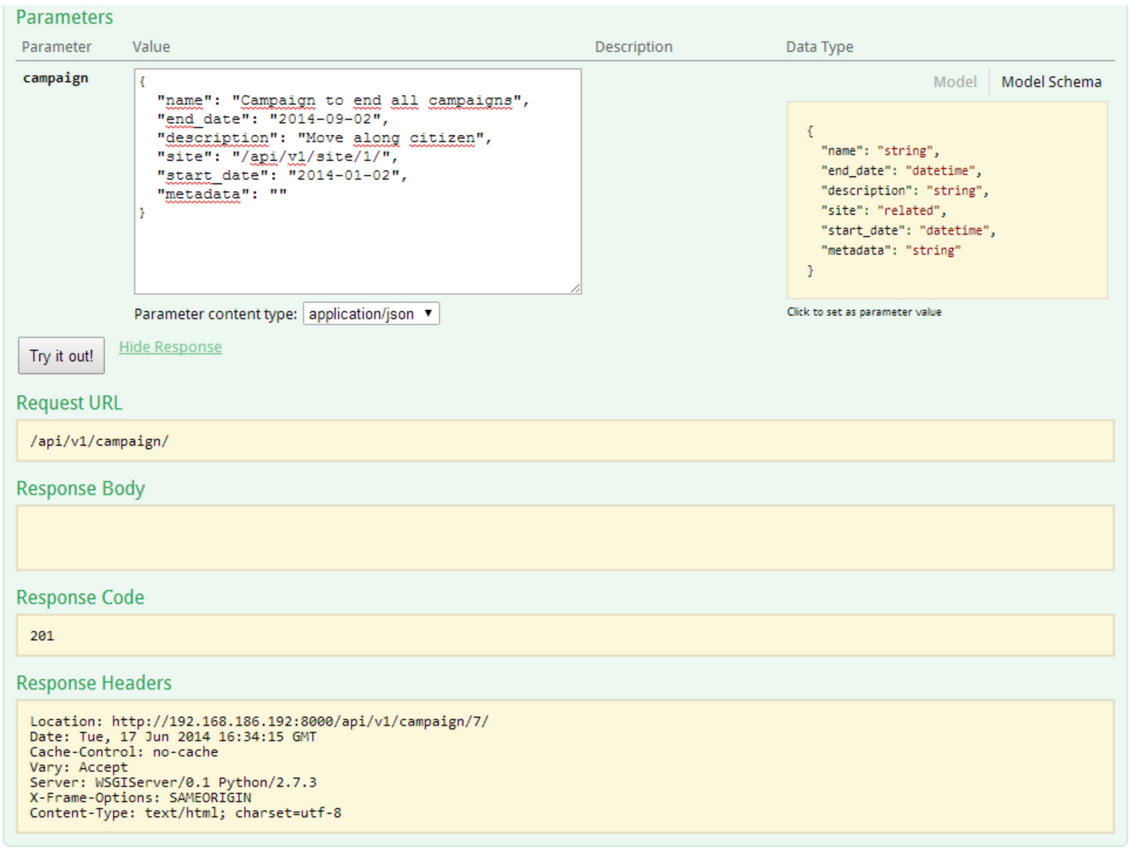
Data Type: Model | Model Schema

```
{
  "name": "string",
  "end_date": "datetime",
  "description": "string",
  "site": "related",
  "start_date": "datetime",
  "metadata": "string"
}
```

Click to set as parameter value

Try it out!

Figure A.9: Campaign POST request



campaign

```
{
  "name": "Campaign to end all campaigns",
  "end_date": "2014-09-02",
  "description": "Move along citizen",
  "site": "/api/v1/site/1/",
  "start_date": "2014-01-02",
  "metadata": ""
}
```

```
{
  "name": "string",
  "end_date": "datetime",
  "description": "string",
  "site": "related",
  "start_date": "datetime",
  "metadata": "string"
}
```

Parameter content type: application/json

Request URL

```
/api/v1/campaign/
```

Response Body

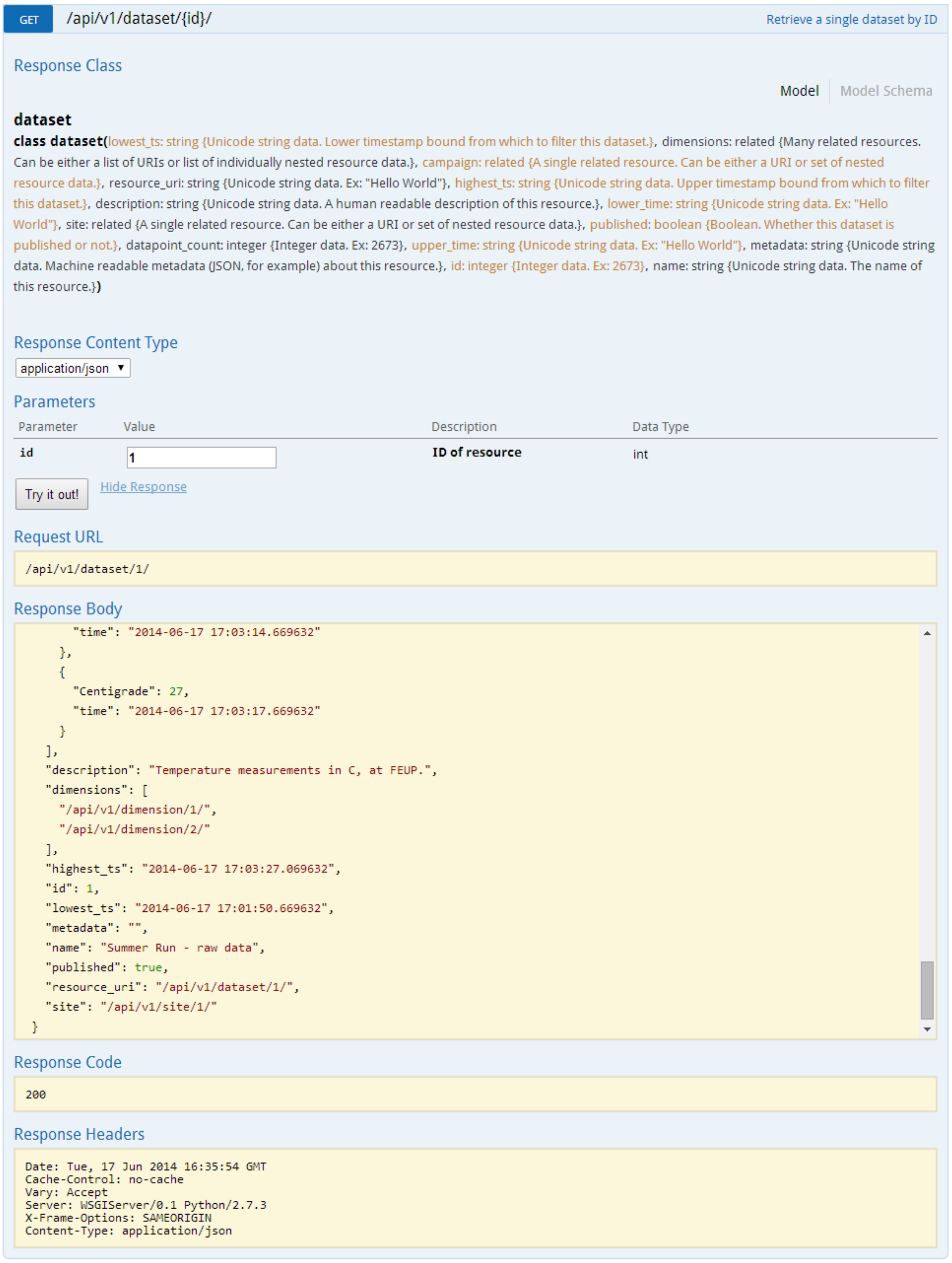
Response Code

```
201
```

Response Headers

```
Location: http://192.168.186.192:8000/api/v1/campaign/7/
Date: Tue, 17 Jun 2014 16:34:15 GMT
Cache-Control: no-cache
Vary: Accept
Server: WSGIServer/0.1 Python/2.7.3
X-Frame-Options: SAMEORIGIN
Content-Type: text/html; charset=utf-8
```

Figure A.10: Campaign POST response



GET /api/v1/dataset/{id}/ Retrieve a single dataset by ID

Response Class

Model | Model Schema

dataset

class dataset(lowest_ts: string {Unicode string data. Lower timestamp bound from which to filter this dataset.}, dimensions: related {Many related resources. Can be either a list of URIs or list of individually nested resource data.}, campaign: related {A single related resource. Can be either a URI or set of nested resource data.}, resource_uri: string {Unicode string data. Ex: "Hello World"}, highest_ts: string {Unicode string data. Upper timestamp bound from which to filter this dataset.}, description: string {Unicode string data. A human readable description of this resource.}, lower_time: string {Unicode string data. Ex: "Hello World"}, site: related {A single related resource. Can be either a URI or set of nested resource data.}, published: boolean {Boolean. Whether this dataset is published or not.}, datapoint_count: integer {Integer data. Ex: 2673}, upper_time: string {Unicode string data. Ex: "Hello World"}, metadata: string {Unicode string data. Machine readable metadata (JSON, for example) about this resource.}, id: integer {Integer data. Ex: 2673}, name: string {Unicode string data. The name of this resource.}**)**

Response Content Type

application/json

Parameters

Parameter	Value	Description	Data Type
id	1	ID of resource	int

Try it out! Hide Response

Request URL

```
/api/v1/dataset/1/
```

Response Body

```
        "time": "2014-06-17 17:03:14.669632"
      },
      {
        "Centigrade": 27,
        "time": "2014-06-17 17:03:17.669632"
      }
    ],
    "description": "Temperature measurements in C, at FEUP.",
    "dimensions": [
      "/api/v1/dimension/1/",
      "/api/v1/dimension/2/"
    ],
    "highest_ts": "2014-06-17 17:03:27.069632",
    "id": 1,
    "lowest_ts": "2014-06-17 17:01:50.669632",
    "metadata": "",
    "name": "Summer Run - raw data",
    "published": true,
    "resource_uri": "/api/v1/dataset/1/",
    "site": "/api/v1/site/1/"
  }
```

Response Code

```
200
```

Response Headers

```
Date: Tue, 17 Jun 2014 16:35:54 GMT
Cache-Control: no-cache
Vary: Accept
Server: WSGIServer/0.1 Python/2.7.3
X-Frame-Options: SAMEORIGIN
Content-Type: application/json
```

Figure A.11: Dataset detail

Appendix B

Node deployment

In this appendix we provide a step by step account of how we initialized the platform. At the end of this process, the reader will have the same configuration we had when we started the first test run. The code repositories for the custom code we developed may be found elsewhere.

B.1 Deploying the initial platform

We started with all virtual machines already commissioned and running. They were running the base Ubuntu 12.04 disk image. We assume the user that we use to login and run remote commands is named "ec2-user" Any time the code snippets have something inside "<>", it means the user must supply whichever value is identified. For example, if the code reads

```
fab -H ec2-user@<ip_of_metric_node> metric
```

and the IP of the metrics node is "192.168.1.123", then the user should write

```
fab -H ec2-user@192.168.1.123 metric
```

Some installations are interactive and require operator intervention. Others are not; operators should supervise the whole process. If the machines are configured to use key pairs for logins, they will be automatically used.

We chose the first machine to be the *command* node. We cloned the repository *fabrica* to the machine and ran the following commands on the shell:

```
sudo apt-get update
sudo apt-get install python-virtualenv
sudo pip install fabric
```

We chose the second server as our metrics gatherer. In order to install it we must first edit the *inventory.py* file and modify the key "metric_ip" to the appropriate IP of the metric machine. After that we run the following command while working in the folder of that file.

```
fab -H ec2-user@<ip_of_metric_node> metric
```

After the node is configured, the Graphite web interface will be available on that IP on port 8000. If, after configuring other nodes and refreshing the Graphite page, you do not see their stats and hardware utilization, you must login to the offending node and manually restart the *collectd* daemon.

```
sudo service collectd restart
```

This is a known issue that manifested intermittently and we could not determine its cause.

The next node we configured was the Operational Store master node. We changed the appropriate key in *inventory.py* and ran the following command:

```
fab -H ec2-user@<ip_of_opmaster_node> opstore
```

The next node was the Triple Store master, in which we followed a similar process. Changing *inventory.py* and running:

```
fab -H ec2-user@<ip_of_triplemaster_node> triplestore
```

After that we configured the Job server. Notice that in *inventory.py* the value to the relevant key is a list of IPs. If only one node of this type exists, then the value should be a list of a single element.

```
fab -H ec2-user@<ip_of_jobserver_node> jobserver
```

The configuration will terminate with an error and suggest that the user consult *syslog*. This is a known issue and is expected; everything was installed successfully.

The next node we installed was Time Series Store node. For this kind of node we have to choose one or more "seed" nodes. The first node is a good candidate for a seed node. We changed the "seed_ips" key to the relevant IP and also the "cassadra_nodes" key. We then ran the following command:

```
fab -H ec2-user@<ip_of_tsstore_node> tsstore
```

The next node was an API node. We inserted the API in the *inventory.py* and ran the command:

```
fab -H ec2-user@<ip_of_api_node> api
```

The operator will want to create a superuser when prompted to. The administration web interface of the API may be accessed on that IP, on port 80. The credentials to login are the ones previously defined when configuring the node.

The final node was a Worker node. The platform does not need to keep track of Worker IPs in *inventory.py*. We ran:

```
fab -H ec2-user@<ip_of_worker> workers
```

Because we had to change from HAProxy to nginx on the Worker nodes after we had the configuration scripts in place, we did not have the opportunity to fix them in a timely manner. Therefore, we manually configured nginx. We first installed it, then we changed the configuration, then we restarted it.

```
sudo apt-get install nginx
<edit default configuration>
sudo service nginx restart
```

The following snippet shows the relevant configuration changes:

```
#in http section
#the name api may be changed to anything you like
upstream api
{
  server <ip_of_one_api_node>;
  server <ip_of_another_api_node_if_it_exists>;
  server <ip_of_yet_another>;
}

#in server section

location /
{
#you will have to use the same name here as you used above
  proxy_pass http://api;
}
```

As the final step, the administrator must login to the administration section of the API mentioned above and must insert one Dimension object into the database. The important fields are *name* and *ts_column*. They must both be “time”; all other attributes may be freely chosen. The administrator must also insert a NamedGraph object in the database. It has a single attribute, *name*, which may be freely chosen. The platform is fully usable at this point.

In order to introduce load we must configure the clients in any number of other machines. The clients are not managed by the configuration scripts and must be manually deployed. We logged in to the client machines and performed the following steps:

```
sudo apt-get install python-virtualenv
cd ~
mkdir ws_client
git clone <url to ws_client repository>
virtualenv env #env here can be any name you like
source env/bin/activate
pip install requests
cd ws_client
```

At this point both the client and the platform are ready to operate. In order to bootstrap the process, we must run the *init.py* script. We invoke it like so, after the previous configuration steps:

```
python init.py
```

Only then can we start running behaviors. The behaviors are invoked like so:

```
python researcher.py #for the Researcher behavior
python tripler.py #for the Tripler behavior
python provider.py # for the Data Provider behavior
```

Note they are all configured to run indefinitely until killed. In order to tweak the size of the datasets the Data Provider will insert, we must change the attribute *length_multiplier*. The default is 0.1 days which correspond to 3000 datapoints per dataset, which correspond to over 9000 seconds worth of data. On a final comment about the clients, the Researcher and Tripler share code in an extremely hacky way. We apologize.

B.2 Scaling the components

In this section we explain how we configured and added more nodes to the platform.

The basic premise is that when we configure a new node we must first insert it into the *inventory.py* and then run the command:

```
fab -H ec2-user@<ip_of_target_node> <role_of_target_node>
```

After that, we must update the configuration of all other nodes that track the IPs of the freshly configured node. We do it like this:

```
fab -H ec2-user@<ip_of_old_node_that_needs_update>
        <current_role_of_node>:update=True
```

So, for example, when we are configuring a new Time Series Store node on IP 192.168.1.123, we have two Workers on IPs 192.168.1.124 and 192.168.1.125 and one API node on 192.168.1.126 we would do the following:

```
<insert 192.168.1.123 into the inventory>
fab -H ec2-user@192.168.1.123 tsstore
fab -H ec2-user@192.168.1.124 worker:update=True
fab -H ec2-user@192.168.1.125 worker:update=True
fab -H ec2-user@192.168.1.126 api:update=True
```

We only update the Workers and the API because they are the only node types that need to track the IPs of the Time Series Store's nodes.

When configuring a slave node to either the Operational Store or Triple Store we do the following:

```
fab -H ec2-user@<ip_of_slave_node>
        <opstore or triplestore, depending on what you want>:slave=True
```

The operator will be prompted for a password in order to start replication. The password is hardcoded and is “repl”. This behavior should be changed as soon as possible. The operator must be careful to write the password right the first time or the machine will stay in an almost unrecoverable state for PostgreSQL and should be recycled.