Designing a load balancing algorithm for compilation servers

Daniel Sunde
Master’s Thesis Spring 2015
Designing a load balancing algorithm for compilation servers

Daniel Sunde

18th May 2015
Abstract

This thesis presents a prototype of a dynamic load balancing algorithm designed for compilation servers. The algorithm takes a user ID and finds the optimal compilation server for the user based on the historical CPU and RAM load of the user. It includes a monitoring tool for gathering user and server resource usage at a process level. The data is stored in a historical database and the historical data is used to predict load on servers. A compilation server system is monitored over time and the results are analysed to find potential bottlenecks in the system. The data gathered by the monitoring script is used to make a custom tailored load balancing algorithm. This paper presents the reasoning behind the design of the monitoring tool and the load balancing algorithm prototype.
Acknowledgement

I would like to express my gratitude to my external supervisors Jon Skarpeteig and Paul Beskow for their constant feedback, comments and engagement during this master thesis. They introduced me to the topic and supported me on the way. Their knowledge and constructive feedback was a great asset throughout the thesis.

Furthermore I would like to thank my internal supervisors Hårek Haugerud and Anis Yazidi for good feedback in meetings and for their teaching during my studies.

Last but not least, I would like to thank my friends, family and loving girlfriend for keeping me cheerful and smiling through the process. Having friends who can proof read and give comments on my paper is of great value and I am lucky to be surrounded by so many talented people.
## Contents

1 Introduction  
  1.1 Motivation  
  1.2 Problem statement  
  1.3 Thesis structure  

2 Background  
  2.1 Load balancer  
    2.1.1 Network and application load balancing  
    2.1.2 Hardware and software load balancing  
    2.1.3 Algorithms  
    2.1.4 Dynamic load balancing of SSH sessions  
  2.2 Monitoring  
    2.2.1 Unix processes  
    2.2.2 Which parameters to monitor  
    2.2.3 Psutil monitoring tool  
    2.2.4 Prediction  
  2.3 RAM cache  
    2.3.1 RSS  
    2.3.2 VMS  
    2.3.3 Cache  

3 Experimental design  
  3.1 Existing system and design  
    3.1.1 Hardware  
    3.1.2 Software  
    3.1.3 User perspective  
    3.1.4 Problems with current design  
  3.2 How to solve current problems
3.2.1 Challenges ........................................... 22
3.3 Experimental design .................................. 23
3.4 Development environment setup ................. 25
3.5 Monitoring ............................................. 27
  3.5.1 Python psutil .................................. 28
  3.5.2 User monitoring ................................. 28
  3.5.3 Server monitoring ......................... 32
  3.5.4 Database storage .............................. 33
3.6 Cached memory ..................................... 37
  3.6.1 How long does files stay in cache? .......... 37
  3.6.2 When can we assume that the cached files have been removed from memory? .... 38
  3.6.3 How much improvement does cache have on performance? ................. 38
  3.6.4 Cache tests .................................... 39
4 Results and analysis ..................................... 41
  4.1 Monitoring script ................................... 41
    4.1.1 User monitoring ................................. 42
    4.1.2 Server monitoring .............................. 42
  4.2 Database ............................................ 42
  4.3 System resources analysis ......................... 43
    4.3.1 CPU ........................................... 43
    4.3.2 CPU results ................................ 47
    4.3.3 RAM .......................................... 48
    4.3.4 RAM Results .................................. 50
    4.3.5 DiskIO ...................................... 50
    4.3.6 DiskIO results ............................... 51
  4.4 Results from cache tests ............................ 52
    4.4.1 Performance improvement .................. 52
    4.4.2 How much load on the system is required to clear cache .............. 53
    4.4.3 Cache results ................................ 54
  4.5 Designing the load balancing algorithm ............ 55
    4.5.1 General Design .............................. 55
    4.5.2 Prediction table .............................. 56
    4.5.3 Available servers ....................... 58
  4.6 Algorithm prototype ............................... 59
List of Figures

3.1 Hardware Design ........................................ 17
3.2 Existing Design Simplified ................................. 19
3.3 New Design Simplified ................................. 24
3.4 Monitoring Design ........................................ 25
3.5 Development Environment ................................. 26
3.6 Database Design ........................................ 34

4.1 One week of server load ................................. 44
4.2 One week of user load ................................. 44
4.3 One work day of CPU load ................................. 45
4.4 One day RAM ........................................ 49
4.5 RAM Available One Day ........................................ 49
4.6 Prediction table of analysed data ................................. 57
4.7 Flow chart for the prototype ........................................ 60
List of Tables

3.1  ESX Hypervisor Hardware .................................................. 16
3.2  SAN Hardware ................................................................. 16
3.3  VM Hardware ................................................................. 17
3.4  Development environment hardware .................................... 27
3.5  Cron job schedule ............................................................ 27
3.6  Compilation time without cache .......................................... 39
3.7  Compilation time with cache .............................................. 39
3.8  Clear cache test ............................................................... 40
4.1  CPU usage for jobs .......................................................... 46
4.2  Cached file test ............................................................... 52
Chapter 1

Introduction

The use of computers and software has been growing exponentially over the last few years and so has the development process. Computers are becoming more advanced and developers are working in large teams to create modern software. The development process has changed vastly and in a large development environment, having the right tools and infrastructure is essential to make the development process more efficient.

One part of making the development process more efficient is by having dedicated compilation servers that can compile software for the developers. Since the compilation of development projects can be resource costly, optimizing the resource usage is important for an efficient system. In a web based scenario (HTTP), a System Administrator can install a load balancer to distribute the load over multiple servers to equally distribute the resource use on each server. In the case of multiple developers doing different projects and compiling different parts of the project, the resource requirements can vary quietly drastically, and the use of a light weight load balancer might not be as efficient as needed.

By monitoring the build servers and the users resource usage on the servers, it should be possible to notice certain patterns for each user and define their resource requirements. Using historical data for the users, it can be possible to increase the efficiency of the load balancer by using more complex algorithms.
1.1 Motivation

This thesis was performed in a company which works with networking solutions, telephony, VoIP, Video Conferencing and related services. The thesis was offered by the Development Operations team in the R&D department. One of their daily routines is managing the compilation servers used by the developers. In its current state a few developers are assigned to a dedicated compilation server each in which they can compile their project. This has its advantages and disadvantages. The advantage of having a dedicated compilation server is that the users can have their projects cached in the memory for faster compiling and their home directory directly connected to their server. By assigning developers to dedicated compilation servers, the developer operations team can observe the servers and make sure that the servers doesn’t get overloaded, and the load is evenly distributed. The downfalls of this setup is that if one compilation server goes down for maintenance or due to failures, the developers assigned to the specific server won’t be able to compile their projects for the duration of the downtime. The requirements for the developers vary and while some of them compile their projects daily, other users only makes changes on a weekly or monthly basis. On some servers with high activity the users will be fighting for system resources, while on other servers there might be lots of available resources.

By making all the servers available to the developers we can avoid some of the disadvantages, but have to overcome some of the challenges that it implies. By installing a load balancer that distribute the developers over all the compilation servers, we can avoid the problems with downtime that occurred with dedicated servers. Developers can be assigned to the build server with the most available resources to optimize resource usage. However, it is not that straight forward since moving a developer to a different compilation server has some cost in performance. When moving a developer, the home directory with the project files need to be moved to the new server, and all previous work stored in cache needs to be read from the disk. The resource requirements for each developer also varies based on the size of the project, how often they build their project and how much of the project they want to compile. Predicting the resource requirements can be difficult since the only data we have is which user is trying to con-
nect, and not how much work they want done or the size of their project.

Using monitoring and historical data it can be possible to calculate the usual resource requirements for a particular user, and make the load balancing decision based off the known data. Applying an algorithm that uses the history of the user, to direct him to a sufficient server, can bypass the negative effects of a load balanced system. The challenge will be to find the optimal parameters that the algorithm should use to make the optimal decision. Constantly monitoring available resources on the compilation servers, the resource usage of the developers and using the historical data to load balance will be the main task for this thesis.

1.2 Problem statement

In this research a new dynamic load balancing algorithm prototype is designed to distribute developers to compilation servers. The algorithm will receive user SSH requests and will find the optimal compilation server for the user in regards to system load. Since the algorithm only receives SSH users with no additional information, the historical resource usage of the user will be the deciding factor for which server it will be assigned to. User resource usage will be monitored in addition to the available server resources. The main resources to be monitored are CPU, RAM, Disk IO and data cached in RAM.

_Design a load balancing algorithm for compilation servers which uses historical data to make optimized decisions._

The main research question that will be the focus of this project are:

- How to monitor user resource usage?
- Which resources are important to monitor for compilation servers?
- What are the limiting factors in the compilation server resources?
- How important is RAM cache for a compilation job?
• How to monitor RAM cache and utilize it?

• Can a more efficient load balancing algorithm be designed based on the gathered data?

1.3 Thesis structure

The rest of the paper is organized as follows. Chapter 2 will look at related work on dynamic load balancing, monitoring system resources and how RAM works. Chapter 3 covers the existing system, changes that has to be made to the system, designing a monitoring tool, how to store data and how to test RAM cache. Chapter 4 presents the results from the monitoring and analyses the data. Using the analysed data, a prototype is designed for the load balancing algorithm. Chapter 5 is a discussion on the results and possible improvements to the algorithm. Finally in Chapter 6 the conclusion is presented.
Chapter 2

Background

The background section covers how a common application load balancer work and what technologies it applies to load balance data. This is important to determine which functions we can use from a load balancer and the functionality it lacks for this specific project. It also covers how to monitor system resources at a process level. To be able to verify if RAM cache is applicable for this project, it is important to understand how RAM works, which will be covered in this section.

2.1 Load balancer

A load balancer is used to distribute load across multiple servers and helps utilize multiple computer resources. The most used load balancers are software based, which is installed on a server that listens to traffic on a specific port[1]. The software forwards the traffic to one of the back-end servers gaining the benefit of having a hidden server structure, which increases security. The back-end server handles the job and returns the results to the load balancer, which then replies to the request from the user. With multiple back-end servers which all can reply to a incoming request, a load balancer offers high availability.

There are multiple reasons for using a load balancer:

**Redundancy:** Making sure a service is always available. Using a load balancer makes sure that the system is always available even if a server should go down for maintenance, random error or server fault. By having
duplicate servers, a system is more reliable and acts as a fail-safe or backup when a problem occurs.

**Distributed workload:** When one server can’t handle the requests or the load from users, having multiple servers to share the load is required to ensure reliability and having enough resources. A load balancer distributes work across multiple servers based on different algorithms.

**Optimize resource usage:** The more advanced algorithms can send workload to the server with the most available resources, or the server with the fewest users. Using the more sophisticated algorithms, the load balancer can observe the back-end server load and assign work based on different factors such as least response time, number of active connections and the current load.

### 2.1.1 Network and application load balancing

A load balancer can use two different methods of deciding the status of a back-end server. Using Network Load Balancing will not work in this project since it is based only on the network layer and uses IP addresses and destination port to make its decisions. It will send ICMP ping requests to the back-end server and will consider them to be available as long as it gets a response or by using a three-way TCP handshake. The other method is based on the application layer and it will take into consideration the content type, custom headers and the application behavior. Using application load balancing, the method of checking availability of back-end servers is by observing the service and its availability.

The difference between them is important because we require the service on the back-end server. A server can be responsive to ICMP ping, but the service on the server might not be running. When the load balancer is passing requests to a back-end application on servers, it is required to use an application load balancer to ensure that the service handling the requests is available[5].
2.1.2 Hardware and software load balancing

There are many options available when trying to chose a load balancing solution. It is based of requirements, features, cost and the complexity. Software based solutions exists in open source form and more complex software that can be bought. There are also hardware based solutions with more complexity, higher performance but at a higher cost.

**Hardware load balancing:** is known as a hardware load-balancing device (HLD) and is a a physical router unit which directs incoming requests to back-end servers. It uses a Application Specific Integrated Circuit (ASIC) chip that acts as a processor that can perform tasks much faster and at higher efficiency. The system works in the same ways as a software based load balancer, but can perform tasks more efficiently and provides some extra security. There are some drawbacks to this solution such as the cost of equipment. New tasks may require new ASIC chips and developing code for such a system is more difficult[5].

**Software load balancing:** offers very good solutions at a much lower cost. There are many open-source load balancers to chose from that have become quiet advanced. Some of them are specifically designed as reverse proxy’s and the intended use is load balancing web servers[3]. They offer some different algorithms to distribute the load, but might lack some of the more advanced algorithms. They operate by having a IP list of the back-end servers and monitors the status of the service availability on the servers. The difference from the hardware solution is that it uses more system resources and overhead to make decisions which makes them less effective than a hardware solution. The reduced cost can be quiet significant and it can be easier to develop customizations to the load balancer.

One of the earliest ways of doing software load balancing was by using DNS. Round Robin DNS was implemented by associating multiple IP addresses with a domain name. When a user requested the IP for that domain name it would get a list of IP’s in return or the DNS would rotate which IP to return. This was a very simple way of doing load balancing and had some few drawbacks such as the DNS returning an IP of a server that might not be responding. DNS did not balance load very well between
the servers. This has become a standard implementation of DNS\cite{2}.

2.1.3 Algorithms

The most important part for this project is the load balancing algorithms and if they are advanced enough to be used. This section covers the different algorithms load balancers use and how they work and are described in detail in a paper by Deshmukh\cite{5}.

Static algorithms

Static algorithms are the simpler version with less overhead and faster decision making. They work very well in an environment where the work load of incoming requests are similar. They can make decisions based of the current load of the system or an average load on the system. Static algorithms are not well suited for a system that varies in work load during the day. Some of the available algorithms used by load balancers for static balancing are:

Random Scheduling: is the simplest way to balance load. It randomly picks a server from it’s back-end server list and forwards the traffic. It is very simple to implement, but since it chose a server by random, the work load wont be balanced among the servers. One server might get overloaded while others are not being utilized.

Round-Robin Scheduling: is a better way of distributing load than the random scheduler. While the random scheduler chooses a random server, round-robin traverses through the available back-end server list and sends one requests to each of them. The algorithm will distribute the work load evenly between all the back-end servers. This works well where all the incoming work is about the same size and all the back-end servers are homogeneous, but it will not optimize the utilization of the back-end servers if they have different hardware specifications.

Weighted Round-Robin Scheduling: builds on the round-robin algorithm, but fixes the issue round-robin had with different back-end server specifications. A back-end server with doubled hardware capacity can be assigned
a higher weight and will be assigned more requests than the other servers. It does not consider the processing time of each job so if the incoming traffic varies in work size, the system won’t do a good job of balancing the work load over the back-end servers.

**Dynamic algorithms**

Dynamic algorithms is an improvement over the static algorithms where it monitors the system and the current load on the servers to make better decisions of where to send the request. It optimizes the utilization of the back-end server hardware, but at a higher cost of overhead and work at the load balancer.

**Central Queue Algorithm:** keeps a job queue on the load balancer and has a central load manager. When a new job arrives to the load balancer it is put in the job queue under the principal FIFO. If there are available servers in the central load manager the job is assigned to the server. If all servers are busy, the job will be buffered in the queue and wait for an available server. When a server’s processor load falls under a certain threshold it will be added to the central load manager and check the job queue if there are any jobs waiting. If there are jobs waiting, the job will be assigned to the now available server.

**Dynamic Round Robin:** Is very similar to Weighted Round Robin, but instead of the static weights, the servers are monitored and the weight is adjusted. This is a real-time server performance analysis that changes the weights of the servers based on parameters such as number of connections or the fastest response time. These two parameters can be combined to an observed parameter that combines the two. It can also use the predictive method which analyses the trend of the servers to see if they most likely will have much incoming work or if they are declining and are getting close to more available resources. This is an advanced way of load balancing and is not available in the simple and free to use load balancers.
2.1.4 Dynamic load balancing of SSH sessions

Load Balancers are generally designed for the HTTP protocol and applications that receive a homogeneous load. The HTTP protocol which generates uniform load across the connections is not required to be persistent since a HTTP connection usually has a short time to live. When load balancing SSH we get huge differences in the individual load of each connection and the session can last from a few seconds to several hours or more. NASA has developed an SSH load balancer called Ballast \cite{7} which is very similar to this project. It monitors each individual SSH users system load and stores it in a historical database. When a user logs on to the system, the load balancer will check the historical data for the SSH user and look at its user pattern before it decides where to place the connection.

Ballast uses agents on all the back-end servers to gather user resource usage and to monitor available resources. Each user of the system has a Ballast client installed on their local machine which provides the load balancer with SSH information and provides a transparent connection. The load balancer itself stores user resource history, available back-end server resources and uses a specially designed algorithm to make the balancing decisions. In addition it tries to find the most important system resource for the user. If a user is known for using much CPU, the algorithm will value that resource over memory and other resources.

The Ballast Load Balancer covers almost all the aspects of this project, but there are a few differences. Ballast is able to be transparent by using software on all the clients that are using the load balanced system. With over 400 client machines with personal configuration and no easy access, making them all install additional software is not optimal. In addition, the Ballast is designed for a general system and not for a specific use case like this project which is for load balancing compilation servers. Since load balancing users to compilation servers might be highly reliant on cache in the memory, this parameter needs to be specifically designed.
2.2 Monitoring

To be able to monitor system resources we have to understand where this information is available in a Unix system and how we can extract it. There are many parameters to monitor in a system and we have to make decisions on which parameters to monitor. The decision making is based on related work and is covered in this section. Lastly, we cover some tools that can help us in the extraction process.

2.2.1 Unix processes

A unix system uses a pseudo-filesystem mounted under /proc which acts as a read-only interface to the kernel data structure\[10\]. This interface allows access to process information for all running processes in the Unix system. In a Unix system, every process is assigned a process identification number (PID) and under the /proc file system these PIDs are listed as integers and acts like sub directories. Under each PID sub directory we can find all the information about the process in a human readable form which can be utilized to monitor each process individually. Under the location /proc/<PID>/stat, most information is displayed by the process including PID, PPID, CPU, RAM and Disk IO. This is the same information that the tool ps uses.

2.2.2 Which parameters to monitor

This section is based on input from the personnel who have been working with the compilation servers on a daily basis and their observations, It also looks at related work and what has been done before.

Based on the input from the employees who have been working with the system, the most vital parameters that are expected bottle necks are CPU, RAM, Disk IO and Net. They also expect RAM cache to be an important parameter in a compilation server system due to some of the jobs being very disk heavy and spending time reading from disk instead of RAM can be time consuming. The most important resource varies from which user who is compiling their project on the servers. This means that different users have different important resources that they want prioritised.
Related work [7] shows that their main focus is on CPU and RAM. In this paper they ended up designing the algorithm to profile users under three different categories: CPU, RAM and a combination of CPU and RAM. The paper says that in a Unix environment there are no easy way to monitor a process in regards to network traffic since there are no process specific network information in the /proc file system.

2.2.3 Psutil monitoring tool

While the `ps` command and `/proc` can give us all the information about the processes needed for monitoring, having a tool to make the monitoring more efficient can be a good idea. There are a few ways of doing monitoring, but due to inexperience with many of them the decision falls on using a Python script. There exists a library for python called `psutil` which is specifically designed for retrieving information from running processes [11]. It is created for system monitoring and profiling, and can be a good tool to use in combination with python.

Psutil is cross-platform meaning that we can develop a tool that works for multiple operating systems. In our case scenario, we only have unix based servers, but with psutil we have the option to make it a dynamic tool to implement on a variety of systems. The tool has been used in other research where the main goal has to be monitoring system resources and profiling [4], [9]. Psutil combines multiple python libraries and gather the information from `/proc`. It includes methods for collecting all the parameters we are looking for. Using this tool will save development time and will help gather data in an organized and efficient way.

2.2.4 Prediction

By monitoring system resource usage for individual users we can make predictions on how much load that generate on a system [6]. By using this method we can use the historical monitored resource usage to place the user on a server which have enough available resources to cover the predicted load generated by the user.
2.3 RAM cache

Random-access memory (RAM) is a storage layer between the hard drives and the CPU. It reads files from disk to give the CPU easy access to these files when they are needed for computation. For a compilation job, the first step is to read the related files to the RAM before the computation starts. When the computation and compilation process is complete, the new files generated by the compilation is written to disk and is usually the final process of a compilation job. The RAM utilizes different mechanics for how long to store different files and the priority for storage. It uses a paging system and we will look closer on how the cached pages work.

2.3.1 RSS

Resident Set Size (RSS) is how much memory a process currently have in the RAM. It represents the amount of data the current process is actively working on. When the process needs other files or are done with the current files, the data is swapped out and new data is read to the RAM. If the compilation job requires the files that were swapped out, it can easily swap them back in to continue work since it creates mappings to the files that are important for the process.

2.3.2 VMS

Virtual Memory Size (VMS) is how much virtual memory the process has in total. It includes all types of memory such as files that are in the RAM and files that have been swapped out. The VMS includes shared libraries (i.e., files in RAM that are used by multiple processes). VMS can be used to show how much data the process is or has been working on even tho it is currently not in the RAM.
2.3.3 Cache

When a process runs in a Unix system it will read the required data into the RAM. While the process is working on the data, the system keeps the data in the RSS of the RAM to make sure that the working files are easily accessible to the CPU. When the process is done, the files are not kept in the RSS, but that doesn’t necessarily mean that they are trashed from the RAM. Recently used data is kept in the RAM cache in case the process requires the files again. This means if a process is run again before the cache is dropped, the process can use the cached data and avoids reading the files from disk again. This can speed up processes, but the files have a risk of being trashed from the cache when the process is inactive for too long. The way it works is pages in the RAM will have a pressure that increases over time. When the data has been recently used the pressure is set to 0. The system periodically runs through all the cached pages and for each run it will increase the pressure on the page. If a new process is started and is reading data from disk to RAM, the page with the highest pressure will be purged from RAM to make room for the new files. This means that if a process runs with low intervals, it will reset the pressure of its cached files every time it run thus making it more likely that it maintains its data in the RAM cache. If a process is idle for too long and other processes are reading data, there is a high chance that the cache will be purged since the pressure has increased over time.
Chapter 3

Experimental design

In this chapter the current state of the system is explained to better understand the problem. The current state of the system can affect the experimental design, and some modifications to the system needs to be applied for the experimental design to work. The chapter covers how we will monitor and gather data to help in the design of the algorithm. Cache can be hard to monitor, so a test plan is presented to find if cache can be important to our solution. The current state of the system and the new design will be covered in this chapter together with the monitoring tool and a test suite for RAM cache.

3.1 Existing system and design

The compilation servers are virtual machines running in VMware. There are 145 active users on the system divided over 12 virtual compilation servers. The system uses NFS to mount storage on the virtual machines and each developer has a home directory mounted on one of the servers. This means that it is beneficial for a developer to use the server that has its home directory mounted, but a developer can use a different server at a higher cost due to the system having to communicate with the server that has the file directory mounted. This indicates that a developer is free to use a different server than the one with the home directory mounted in case of high load on his dedicated server, but should the dedicated server go down for maintenance, the home directory for all developers assigned to that server will be unavailable. This is an issue that could be solved using a load bal-
ancer and iSCSI covered later in this chapter.

### 3.1.1 Hardware

The hardware consists of 4 VMware ESX machines where each hypervisor is running 3 virtual machines each. The system is homogeneous so each virtual machine has the same hardware available. The system is not over provisioned meaning that the virtual servers has allocated a portion of the available physical processors, but not all of them. Each ESX host consists of the following hardware:

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Specification</th>
<th>Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>2x E5 v3 2699</td>
<td>72 cores</td>
</tr>
<tr>
<td>RAM</td>
<td>2133 MHz</td>
<td>384 GB</td>
</tr>
</tbody>
</table>

Table 3.1: ESX Hypervisor Hardware

The ESX hosts are connected to a EMC VNX5600 SAN through 4 fibre channel links. The SAN consists of a tired storage solution with a mixture of SSD and HDD. The tired storage is divided into three tiers. The first tier (T0) is the RAM on the server. The second tier (T1) consists of 10x200GB SSDs and the final tier (T2) is a combination of SSD and HDD where the mixture is 10% SSD and 90% HDD. The SAN is set up to first access the SSD storage for faster writing and the SSD acts as a tier 1 layer between the RAM and the HDD. This makes the HDD act as a long time storage with files that are rarely used, and the SSD contains active files and does most of the job.

<table>
<thead>
<tr>
<th>TIER</th>
<th>Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>SSD 10x200GB</td>
</tr>
<tr>
<td>T2</td>
<td>SSD 10% + HDD</td>
</tr>
</tbody>
</table>

Table 3.2: SAN Hardware

Each ESX host is running 3 virtual machines and each virtual machine is assigned the following amount of hardware:
<table>
<thead>
<tr>
<th>Hardware</th>
<th>Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>vCPU</td>
<td>22 cores</td>
</tr>
<tr>
<td>RAM</td>
<td>100 GB</td>
</tr>
<tr>
<td>Disk</td>
<td>2 TB</td>
</tr>
<tr>
<td>Network</td>
<td>10 GBit</td>
</tr>
</tbody>
</table>

Table 3.3: VM Hardware

The virtual machines sums up to use 66 cores and 300 GB RAM from each hypervisor which means that there 6 cores and 84 GB ram available for the background system. This is intentional to avoid over provisioning the hypervisors. There won’t be any changes to the hardware and all changes required to implement a load balancing solution will be software only. The complete hardware system can be seen in 3.1 Hardware Design.

![Diagram of Hardware Design]

Figure 3.1: Hardware Design

3.1.2 Software

For the software side of each virtual machine it is important that they are kept up to date in terms of compilation software and that all servers offers the same functionality in case of a user switching to a different server. Each virtual server is set up with the same specifications and the most important software is:
• Operating system: CentOS 7 64-bit

• Active Directory Authentication and authorization

• NFS mounted home directory /HOME/USER (/etc/auto.home)

• Toolchain

**Active Directory** is the authentication and authorization containing a list of all users that have access to the system and what permission they have on the system.

**NFS** is installed on the servers to allow access to the storage solution. Today’s system uses a static file called `/etc/auto.home` which holds information for which server should mount each different home directory. Each user is assigned to a specific server and that server contains its home directory. The users are allowed to use whichever server they like, but it is beneficial for the developer to use the server that has the home directory mounted locally. Using a different server means that the developers cached file metadata won’t be available and all read and write operations has to be sent to the server containing the developers home directory. This is an issue that causes extra overhead and slows down the general system due to network latency. Another issue with having the users assigned to a specific server is that in case of a system failure or shutdown, the home directory mounted on the failed server won’t be mounted, and the users on that specific server does not get access to their server. The final issue by having a static mount file is the maintenance when a new user needs to be added to the system, an old user has to be removed or when trying to find a balance between users to maximise system performance.

**Toolchain** is a set of tools that contains software for building products on the server. This generally concerns the different compilation tools for different programming languages. The tool makes sure that all servers have the same software available and that everything is updated to the latest version.
3.1.3 User perspective

From a developers point of view the work is done on a local machine and create a SSH connection to the virtual machine containing your home directory (see 3.2 Existing Design). The project the developer is currently working on is stored on the SAN and is accessed through the virtual machine that has mounted the home directory. Should the server with the home directory fail, the developer will not only be unable to compile the project, but also lose access to the files he is working on. Having multiple developers without access to their files will result in a major cost for the company since they are not able to produce any work during the failure.

![Design.png](image)

Figure 3.2: Existing Design Simplified

The developers are spread across the compilation servers based on the static `/etc/auto.home` file. The load on the servers are manually balanced,
and the servers vary from having just one user to having multiple. This is intentional since different users produce different load and have different requirements. At some points there are servers with no load at all while others may be running at full capacity with developers fighting over the available resources. This is hard to balance in the current system since there are many factors that affects the system and how much workload that is produced. Some of these factors can be developers that come and go due to holidays, moved to different project, leave of absent and new employees. When nearing the end of a certain project there might be high spikes in compilation while at a start of a new project there might not be any load at all. This will affect the servers differently and the result will be some servers being overloaded while others have all resources available.

3.1.4 Problems with current design

This section tries to summarize all the problems the current system design has so it will be clear what the experimental solution is trying to solve.

The main problems are:

- **Problem 1**: Utilization of available resources and not being able to balance the workload
- **Problem 2**: The system is not adaptive to change in user patterns and tackling the change in number of users
- **Problem 3**: Downtime becomes an issue for all developers connected to the failing system
- **Problem 4**: Lack of user history

**Problem 1**: Since there are different user patterns and different stages of a development process, the system can have servers running at full capacity while others are standing idle.

**Problem 2**: Multiple users have different routines and there are changes in staff. The use of a static file causes issues since it has to be manually maintained when a new user comes along or a user leaves. Different users
generates different load on the system and it can be hard to manually balance the system and it won’t be adaptive.

**Problem 3:** Having a server failure or taking it down for maintenance causes developers being unable to access their project and they will sit idle during the period of downtime. This can be temporary fixed by editing the static file `/etc/auto.home` and mount their home directory to a different server, but generates extra work for the administrator. This is not a good solution since it might cause extra workload on a server that is already full and more users has to share the limited resources.

**Problem 4:** Having to manually balance the current system is an issue since the system is lacking in user monitoring. Placing a new user in the system can be challenging since the administrator doesn’t know anything about the user or the varying load of the users currently on the system.

### 3.2 How to solve current problems

Implementing a dynamic load balancer into the current system solution will give a major benefit in all the problem areas:

**Problem 1:** To solve the issue with varying work load on the compilation servers, a load balancer can monitor the compilation servers current load and forward users to the server with the most available resources. This helps balancing out the workload over all the compilation servers and ensures that users are receiving their required hardware resources instead of waiting on other processes to finish.

**Problem 2:** By using a load balancer and making dynamic decisions it is possible to move users around and the changes in the number of users on each server will be adaptive. This needs to be combined with changing the current NFS solution to remove the static home directory. By replacing NFS with iSCSI we can automatically mount home directories to the server a user is assigned to, and the home directory can be available on all the servers. This can cause an issue with RAM cache which needs to be solved. By dynamically balancing users across the available compilation servers,
the workload will be spread across all the servers and user will be more likely to get the hardware they require. The only case that is limiting then is how much hardware that is available, but if all hardware is used across all servers, the hardware or number of servers can be expanded which is easier to do.

**Problem 3:** Service outage can be avoided by implementing a load balancer. If a server goes down for maintenance it will simply be taken out of the load balancing pool and no users will be forwarded to it. The system will then automatically forward users to the available servers and users will have their home directories mounted with iSCSI thus removing the downtime for users.

**Problem 4:** By designing a script that monitors the user processes on the system and store the data in a historical database it is possible to gather knowledge on the user patterns and their system resource load. This information can be used to decide which server the user is placed on to help balancing out the overall workload on the servers.

### 3.2.1 Challenges

Implementing a load balancer and iSCSI can solve the major issues the current system has, but it also inflicts some issues on its own that has to be overcome:

- Load balancing algorithms are not generally designed for placing users on compilation servers
- Moving users around in the system has a cost due to files having to be read from disk instead of RAM
- The load balancer needs information about the users to make good decisions
- Different users have different resource requirements
- Historical data has to be gathered to help the load balancer in making decisions
Since the load balancer only receives an SSH request, the load balancing algorithm has to make decisions based on the historical resource usage of the users and the servers. This information can be gathered by implementing a monitoring script for the servers. The monitoring needs to gather data on the server load and the system resource consumption of the different users. The data can be stored in a database and be used by the load balancing algorithm to make decisions.

When data has been gathered, it can be analyzed to find which of the system resources is the limiting factor for compilation. While CPU, RAM and DiskIO is the suspected limiting resources, the analysis of historical data can show accurately which resource is the limiting factor. The limiting factor can be used to improve the load balancing algorithms decision making.

It is suspected that cached files and metadata has significant influence on the performance of compilations. With dedicated servers it was ensured that users were using the same servers and keeping their cache. In a load balancing scenario, the users can be moved around to different servers which means that their cached data is lost. Cache can increase the performance of a compilation job since it indicates less Disk I/O. To decide the impact of cache, testing and analyzing has to be done to determine how much it affects the performance of a system. If the tests show that cache has substantial value, it should be implemented into the load balancing algorithm.

3.3 Experimental design

By implementing a load balancer into the existing design, users will send SSH requests to the load balancer instead of their dedicated server. They will no longer have a dedicated server since the static home directory solution will be replaced with a dynamic solution. By having a dynamic storage solution, users will be able to use any compile server since their home directory will be mounted when they log into a server. The SSH requests is handled by the load balancer and the algorithm makes decisions based on the historical data for the user. When the most suitable server is found, the
request is forwarded to the selected compilation server (Figure: 3.3 New Design Simplified).

To get an overview of the solution we are trying to implement we will cover the intended design and what the goal is with this design. The main goal is to design a load balancing algorithm for forwarding developers to compilation servers and avoid congestion in the system resources. To be able to decide the important factors that the algorithm needs, the compilation servers has to be monitored and data needs to be gathered to find the most important factors for the algorithm. Another factor can be the difference in generated load by the different users so trying to categorize users for their most important resource and average load on a system can be a possible factor. To get the required data to design an algorithm, users resource usage on the compile servers will be monitored. The servers needs
to be monitored so the algorithm is aware of available resources on each system and be able to find if a system is overloaded or can take more users. The data that is collected needs to be monitored over time and this can be achieved by storing the collected data in a database. The user monitoring and server monitoring will be done on each compilation server. The data collected will be pushed to a database that is located on the same server as the algorithm will be running on. The intended design for data collecting and monitoring is described in Figure 3.4 Monitoring Design.

![Figure 3.4: Monitoring Design](image)

### 3.4 Development environment setup

Since the compilation system described is active and in a production state, we have to set up a development environment for our tests and tools to not interfere with the activities on the system. The development environment will be designed like our planned solution in a virtual environment using VMWare. The main purpose of the development environment is to develop a monitoring tool that can be applied to the production environment. The monitoring tool needs to gather system resource data from the compilation
servers for both users and the server utilization. The data will be pushed to a database which will reside on the same server as the load balancing algorithm.

There is no need to go full scale and have a replica of the production environment. The development environment will consist of 5 servers with the following roles:

- One server for the database and load balancing algorithm prototype
- One server to act as an external storage to mimic the planned solution
- Three compilation servers with varying load for testing the monitoring tool

Figure 3.5: Development Environment

The development environment does not require users to be moved between systems and an NFS storage solution is used instead of iSCSI. The reasoning being a simpler setup and that the solution covers the requirements needed to develop the monitoring tool.
Table 3.4: Development environment hardware

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>vCPU</td>
<td>8 cores</td>
</tr>
<tr>
<td>RAM</td>
<td>8GB</td>
</tr>
<tr>
<td>Disk</td>
<td>50GB</td>
</tr>
<tr>
<td>Network</td>
<td>10GBit</td>
</tr>
</tbody>
</table>

The **Load balancer and database** will have a MySQL database installed which will receive all the data from the compilation servers.

**Compilation servers** will have three users on each that are building Linux kernels at different intervals to vary the load on each machine and to simulate compilation processes. The building of the kernel jobs will be added to cron jobs to automate the process. The kernels that are being built will be stored in the NFS storage to simulate how the production environment fetches data.

Table 3.5: Cron job schedule

<table>
<thead>
<tr>
<th>Server</th>
<th>User</th>
<th>Run time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server 1</td>
<td>All</td>
<td>Build every hour</td>
</tr>
<tr>
<td>Server 2</td>
<td>All</td>
<td>Build every 30 min</td>
</tr>
<tr>
<td>Server 3</td>
<td>All</td>
<td>Build every 10 min</td>
</tr>
</tbody>
</table>

**NFS storage** is a storage device for the three users and holds the data they are using to compile. This machine is set up as a NFS server and the compilation servers have the client installed to connect to this device.

### 3.5 Monitoring

The first step to creating the load balancing algorithm is to understand the server load and user load on the system. By monitoring the system and resource usage and analyzing the data, the deciding factors for the load balancing algorithm can be found to make a tailored algorithm for the system at hand.
3.5.1 Python psutil

*Psutil* is a python cross-platform process and system utility that will be used to monitor user processes. It iterates over all running processes in the system and can gather information about each individual process like UID, PID, CPU, RAM and DISK. It can also manage processes by setting priority, stopping or starting the process and manage the state of the process. It’s a robust tool, but for our purposes, this utility will be used to only gather information.

3.5.2 User monitoring

From observing the production system and from related research, the hardware resources to monitor for a user that are the possible bottlenecks are CPU, RAM and Disk IO. For each user on the system, all processes the user runs needs to be monitored and gathered over time to make a suitable test set of data. Other parameters to look at will be the run time and what operation the user is running to be able to make predictions of the user patterns and average generated load.

A Linux environment have easy access to the information required, but has to be put together by a script to gather the desired data. The script is created in Python and uses the library Psutil to gather process information. The script will be run every second on each compilation servers and the script pushes the output to a database. Only processes run by users are monitored and the parent process is used as identifiers while child processes are gathered recursively and organized under the parent process and under the user.

**Script functionality:** The script will get all running processes on the system. The processes will be filtered by User ID where only users with an UID over 1000 will be monitored. This will exclude all system operations and only look at human users. Since parent processes spawn child processes which generates load on the system, the child processes needs to be monitored. The processes will be filtered under parent processes while child processes will recursively be filtered under the parent process. Psutil is then used to gather all the required data for each process and the data is
pushed to a database.

CPU is stored as a percentage of the total available CPU of the system. In a multi-core system the number of cores on the system needs to be found so that the total CPU for all the processes can be divided by the number of cores to get the percentage of the total system. Since all systems are homogeneous, this percentage will be the same on all servers. In a Linux environment the processes information in /etc/proc does not give CPU in percentage, but how many "ticks" the process did at a certain time in the CPU. Process percentage use needs to be calculated by comparing two timestamps and the CPU usage between the time.

\[
\frac{\Delta \text{ProcCPU} + \Delta \text{SystemCPU}}{\Delta \text{Time}} \times 100 \times \text{NumberOfCPU} \quad (3.1)
\]

In the equation the $\Delta$ is two measurements at two different time periods. The User CPU is how much time the process used in the CPU. The System CPU is how much time spent in the system while executing tasks on behalf of the process. These values are added together since they are both load generated by the same process. The total is divided by the time difference between the measurements and the multiplied by 100 to get percentage. If a process has multiple threads it can get a percentage above 100 since it can use multiple cores in the processor. The number of CPU cores are multiplied with the percentage to get the total use of the system.

Since we need a difference over time to calculate CPU, the measurements needs to be stored for each run of our script and then be calculated at the next run. This can be achieved with os.times(), but Psutil has a built-in function called psutil.proc.cpu_percent() and does the calculation for us using the equation and storing the process object for the total run time. The database should contain the total use of the system given in a percentage where the maximum is a 100%. Since the system is homogeneous and all servers consists of 8 cores, the CPU percentage given by psutil.proc.cpu_percent() will be divided by the number of CPU cores and then pushed to the database.

RAM can be difficult to monitor on a process level, but the goal is to find if certain processes or jobs are heavy users of RAM. This means jobs
that works on big data sets and has to read a lot of files into the RAM to be able to compile. Since the system is homogeneous, the percentage usage will be the same on all systems. To give an indicator on how much RAM a process is running, the percentage usage of the process will be stored in the database along with the RSS (Resident Set Size) and VMS (Virtual Memory Size). The RSS and VMS for a process can be seen by using the command `ps` and it is also available with the `psutil` tool with the command `process.memory_info()`. The percentage use is calculated by dividing the RSS for a process by the total system memory.

\[
\frac{\text{ProcRSS}}{\text{TotalRSS}} \times 100 \tag{3.2}
\]

In the `psutil` library there is a method available called `process.memory_percent()` which does the calculations for the individual processes. The RSS will be a varying number which shows how much memory the process has active in the RAM. The VMS shows how much virtual memory the process uses. This is memory which includes all code, data, shared libraries and pages that has been swapped out. There is no planned use for this parameter, but it can be good to include some extra data in special cases where it might have a factor on the performance.

The total amount of RAM used by a process (percentage, RSS and VMS) are monitored for each process and children of the processes are grouped under the main process. The data is then pushed to the database.

**Disk IO** represents how much data a process has to read or write to the hard drive during a compilation. If the compilation job is working on a large data set, the system will spend time reading the necessary files from disk to RAM which can be time consuming on large data sets. During this period of time, the CPU is not working on the actual compilation, but waiting for the files to be ready for access. In these cases the disk operation might be the parameter that slows down the system and not the CPU. By monitoring these parameters we can observe if this is a common problem and if it has to be taking into account when designing the load balancing algorithm. If some jobs are heavy Disk IO processes and does not consume a lot of CPU, the algorithm can be designed to places these sort of jobs to-
gether with heavy CPU jobs since they are using different resources on the system and will not affect each other as much as two heavy CPU processes would.

To monitor the disk in and out operations, *psutil* has a built in method called `process.get_io_counters()` which gathers the number of read/write operations for a process and read/write given in bytes. The method returns the total amount of bytes read or written at the given point. This means that the final data point of the job run will hold the total bytes the process used for the entire run. The other values can be interesting to see at what points in the run the process stops to read or write data to disk, but the most important value will be the total to see if the specific compile job is a heavy user of disk read/write operations.

The monitoring scripts works the same way as for CPU and RAM. Meaning that the compile process will gather all children processes and group them together to get the total amount of the job run. The data is then pushed to the database.

**Processes** that are monitored for each user will give a total overview of the resource usage of a user. The most important processes are the ones that are performing compilation of a project, but all operations a user executes on a compilation server will generate load meaning that to observe total load generated by a user, all processes will be monitored. Since the processes and all generated load will be stored into a database, the processes needs a unique identifier to avoid collisions. The process identifier (PID) can’t be used alone since PID’s are reused in a system and with multiple servers there can be multiple servers using the same PID at the same time. Every process that runs has a start time parameter which can be used to get a unique identifier. To store the data points in the database the system will categorise the processes under the PID in combination with the start time since both of these parameters are constant. However, using just the start time and the PID will not create a unique identifier since in the worst case scenario, multiple servers can start processes at the same time and it can be the same PID since the processes are on different servers. The final piece of the unique identifier will be the user identifier (UID) in combination with PID and start time. Since a user can only be on one server at a time, this
combination will guarantee that the processes are unique in the database and that all the data points can be categorised under the same identifier.

3.5.3 Server monitoring

The server monitoring script is created in a similar way as the user process monitoring but can be simplified quite a bit. When monitoring the compilation servers we want to gather the total load on the system and we don’t have to filter on specific processes or gather child processes recursively. *Psutil* is used for the server monitoring in a *python* script. The *psutil* library has specific methods for observing the system resource usage.

The server monitoring script is executed every minute and stores the data under a unique identifier which will be the server name. The script stores the parameters: server name, CPU usage in percentage, RAM used in percentage, RAM available, RAM cached, disk read operations and disk write operations.

**CPU** is monitored on each server to find the current load on the system and to find if the server is available to handle more load. Using the *psutil* library in *python* we have a method called *psutil.cpu_count()* and *psutil.cpu_percent()* which can be used to find the current load and can be used to get load over time since we are observing the system over time. The cpu_percent() method returns a percentage of CPU used on the server. Since the system has multiple cores the percentage can be above 100% since each 100% indicates a core running at max capacity. On our servers we have a total of 22 cores which means that the maximum capacity will be 2200%. The method cpu_count returns the number of cores and by dividing the cpu percentage by the number of cores we get a number in percentage with the maximum of 100% where a 100% is full load on all cores. Since the system is homogeneous, this calculation will work for all servers and by making the maximum capacity to be 100% it will be easier to compare it with the numbers we get from the user monitoring.

**RAM** is monitored on each server to see the general load on each server and to find if this has any impact on how we want to design our load
balancing algorithm. *Psutil* offers a few more parameters when gathering information about the RAM from the entire system with the method `psutil.virtual_memory()`. While monitoring processes the only information available was RSS, VMS and percentage use of the entire system. When monitoring the RAM of the entire system we have access to total physical memory, available memory, percentage usage and a few more. The most notable here is that we can get how much of the memory is being used as cache. It is suspected that for compilation jobs with big data sets, the cache can be important to improve the compilation speed since the jobs does not have to do as many read/write operations if the files are already available in cache.

In the server script we will be storing the percentage usage of memory, the available memory and the amount cached. These parameters are pushed to the database along with the other parameters from CPU and DiskIO.

**DiskIO** can be monitored on a system level to see the total of read/write operations given in count and bytes since last restart of the server. It’s not all that use full to monitor, but measurements can be compared to find at which times the system is doing a lot of disk operations. *Psutil* offers a method called `psutil.disk_io_counters()` and we will utilize the read bytes and write bytes given by the system. This is the total amount of bytes read/write since the last restart and will only increase over time. To get any useful data out of this we can calculate the difference between measurements.

### 3.5.4 Database storage

A MySQL database will be created to store all the data gathered by the user monitoring scripts and the server monitoring scripts running on all the different compilation servers. On the server side there will be one INSERT statement each minute for each server. The user monitoring script will get one measurement each second per process for each server which ends up being a lot of INSERT statements during peek hours (i.e., daytime office hours). The database design can be seen in Figure: 3.6 Database Design.
The database is divided into 5 different tables. The tables are USER, JOB, jSAMPLE, SERVER and sSAMPLE. This is done to module the structure so we can split the information we get from the monitoring scripts into the appropriate tables. All tables has an auto-increment column called ID which makes sure that every entry in the system is unique. The database will hold raw data so most of the calculations and analysis has to be done after there is some data in the system to analyse.

The USER table is used to store all the different users running on the entire system. These are mostly developers who are using the compilation servers and when a new user enters the system they will be added to this table. The table consists of three columns: UID, NAME and SERVER. UID is the user identification number in the Unix system. These are unique for each user. The NAME column holds the user name to make it human read-
able. The last column is SERVER which holds the last used server that the
user was logged on to. The server column is added in case the results show
that cache has a significant impact on the system. If cache turns out to be
an important parameter, the last used server might be the best choice for
this user to enable the use of stored data in RAM.

The JOB table is used to store each parent process run by a user. This
table holds each unique process run by a user and every data measurement
for the process and its child processes are stored connected to this identifier
table. It consists of the columns PID, UID, START_TIME, CMD_NAME,
COMMAND and SERVER. The PID is the process identification number
given by the Unix system. These are identifiers in the system to keep track
of running processes. The identifier is reused by the system, this means
that this value won’t be unique since we have multiple servers and that
the value is reused. The UID is the user identification number which is
unique for each user. This is a foreign key connected to both the USER
table and the jSAMPLE table. It is used to keep track of which user is
running the process and is part of the unique foreign key connected with
jSAMPLE. START_TIME is used to group processes in the jSAMPLE table
to this identifier table. This column holds a timestamp for when the process
started which is set by the Unix system. This value in combination with
UID and PID, and is used to create the unique identifier in the JOB table.
CMD_NAME is the name of the command executed in the system to start
this process. This is gathered to find the different operations a user executes
in the system and to find how much system resources the different jobs re-
quires. The COMMAND column holds the full command executed on the
system and not just the name of the program run. While CMD_NAME only
holds the program name (i.e., make, git, python), the COMMAND column
holds all the parameters sent with the program (i.e., git commit -a -m "initial
commit"). The usage for this column is to locate specific commands that are
generating a lot of load on a system. This can be specific scenarios where
users are trying to build a project and trying to force the system to use more
CPU cores than available. A command like that can slow down perform-
ance and having the full command to find problems like this can be useful
to improve performance. The SERVER column contains the server name
that this process ran on and is also a foreign key to the SERVER table.
The **jSAMPLE** table stores all the data points and measurements gathered by the user monitoring script. This will be where most of the data is stored. While the JOB table only gets one entry for each command run on the system, this table holds one measurement for each second while the job ran on the system. This will be the most important table to analyse when analysing the results later on. It consists of the columns PID, UID, START_TIME, RUN_TIME, CPU, RAM, RAM_RSS, RAM_VMS, DISK_IN and DISK_OUT. The PID, UID and START_TIME are there as a combination to create a unique identifier to connect the samples to the correct job in the JOB table. RUN_TIME is the time-stamp when the measurement was inserted and can be used to calculate the total run time of the process. CPU holds how much CPU in percentage the process is using on the system. RAM holds the value of how much RAM in percentage the process is using in the system. RAM_RSS is the resident set size in the RAM. RAM_VMS is the virtual memory size. DISK_IN is how much data was read from disk given in bytes. DISK_OUT is how much data was written to disk given in bytes.

The **SERVER** table is just a small table holding the available servers where processes are run and is used to connect USER, JOB and sSAMPLE together. It only has one column named NAME which stores the name of the servers.

The **sSAMPLE** table will contain server measurements and will get one new INSERT each minute for each server that is being monitored. It consists of the columns NAME, TIMESTAMP, CPU, RAM, RAM_AVAILABLE, RAM_CACHED, DISK_IN and DISK_OUT. NAME holds the name of the server that is being monitored. TIMESTAMP is at what time the measurement was entered into the table and can be used to find high and low peaks on the server performance. CPU is how much CPU is used on the server given in percentage. RAM is how much RAM that is used on the server given in percentage. RAM_AVAILABLE is the available RAM on the server given in megabytes. This is calculated with the formula \( \text{free} + \text{buffers} + \text{cached} \) where free is memory not being used at all, buffers are cache for file system metadata and cached is cached files that have a low priority in the RAM. RAM_CACHED will be the largest portion of the RAM since the system stores data in RAM until it is overwritten. The amount is stored in
megabytes and can be used to see how much of the system RAM is used for cache and if there are changes. DISK_IN is the amount of data read from disk since the last system restart. DISK_OUT is the amount of data written to disk since last system restart.

3.6 Cached memory

With the data gathered by the monitoring scripts it is possible to analyse CPU, RAM and DiskIO to find which of the system resources has the biggest impact on our load balancing decisions. The last parameter we want to check which can have an impact on the algorithm is if RAM cache can improve system performance.

If a process uses a lot of resources to read files before the compilation, the files will be read from disk to RAM and stay in the cache until some other process comes and needs the space. There are three important issues we want to figure out with these tests:

1. How long does files stay in cache?
2. When can we assume that the cached files have been removed from memory?
3. How much improvement does cache have on performance?

3.6.1 How long does files stay in cache?

According to Linux documentation ([8]), files are stored in the RAM until a new process comes and requests memory. This means if only one user uses a server and runs a compilation that loads a lot of files, these files will stay in the RAM until the server restarts or the user runs a different process that requires the space in the memory. If the user only runs the same compilation job all the time, the files will keep staying in the RAM until the user runs enough other processes so that the cached files need to be flushed or if the files are changed in the system. There are a lot of information on how cache works in RAM and it is dependent on page size, age and location in the RAM. Cache has an aging factor meaning that the system will clear the
oldest cached files first. Another factor is the page size which means if the system is looking for a particular size it needs to store the new files, it will try and free up space in the RAM that fits that size. This means that it’s not always easy to figure out if the files are still in the cache, so we have to set some boundaries for when it can be a deciding factor in the load balancing decision or if it should be ignored.

3.6.2 When can we assume that the cached files have been removed from memory?

To answer this question we need to perform some tests on the compilation servers and try and find a pattern for when it happens. If no processes has been run on the system since the last run we can assume that all files are cached in RAM. If it has been a few days since the user was logged on to the system and other users has been using the system, we can assume that most of the cache has been replaced and it wont be a benefit anymore. If it has been a few hours since the user was logged on to the system and some other processes has been run in-between, we have to run some test to see how much load a system can handle in-between before we assume that cache has been replaced.

3.6.3 How much improvement does cache have on performance?

This is the most important question and we need to run some tests to figure it out. If cached files only improve performance by a few seconds, it might not be that important. But if cache can improve performance for certain processes by a significant amount, it becomes much more valuable. This performance increase has to be put up against the value of how important the other parameters are for the user (i.e., we can’t put a user that requires 90% CPU on an already full server just to get the benefits of cache since then the CPU overload will slow down all processes on the server). To calculate the performance impact cache can have on a compilation job that requires a lot of DiskIO, some tests has to be done.
3.6.4 Cache tests

There are two tests we want to do on RAM cache:

1. Performance improvement

2. How much load on the system is required to clear cache

**Performance improvement** will be measured by running a few different compilation jobs that requires a varying amount of data. One test will be when no files are cached for the compilation jobs, and the other will check if it's faster to run the same job with all the data already in cache. The runs will then be compared to see the difference in time required to find the improvement in performance.

<table>
<thead>
<tr>
<th>Test type</th>
<th>Size of file</th>
<th>Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small</td>
<td>200MB</td>
<td>No</td>
</tr>
<tr>
<td>Medium</td>
<td>1GB</td>
<td>No</td>
</tr>
<tr>
<td>Medium</td>
<td>2GB</td>
<td>No</td>
</tr>
<tr>
<td>Large</td>
<td>10GB</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 3.6: Compilation time without cache

<table>
<thead>
<tr>
<th>Test type</th>
<th>Size of file</th>
<th>Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small</td>
<td>200MB</td>
<td>Yes</td>
</tr>
<tr>
<td>Medium</td>
<td>1GB</td>
<td>Yes</td>
</tr>
<tr>
<td>Medium</td>
<td>2GB</td>
<td>Yes</td>
</tr>
<tr>
<td>Large</td>
<td>10GB</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 3.7: Compilation time with cache

**How much load on the system is required to clear cache.** To find the threshold for when we can assume that cache is replaced, we will run the same compilation jobs again with cache, but in between runs we will run other processes. In the start we will only run a single process between runs and then re-run the compilation job to see if there is a difference in time. We will continue this process and adding more and more work between runs until we see a loss in the compilation run time and keep going until
the compilation takes as long as it did without any cache.

<table>
<thead>
<tr>
<th>Test type</th>
<th>Number of other processes</th>
<th>Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compilation time</td>
<td>0</td>
<td>No</td>
</tr>
<tr>
<td>Compilation time</td>
<td>0</td>
<td>Yes</td>
</tr>
<tr>
<td>Compilation time</td>
<td>1</td>
<td>?</td>
</tr>
<tr>
<td>Compilation time</td>
<td>2</td>
<td>?</td>
</tr>
<tr>
<td>Compilation time</td>
<td>#</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 3.8: Clear cache test
Chapter 4

Results and analysis

In this chapter the results from our tests and monitoring will be analysed to find the bottlenecks and the most valuable system resources for the compilation servers we are working with. The results are then used to design the algorithm which is covered in this chapter.

4.1 Monitoring script

The development of the monitoring scripts was the most time consuming process. The scripts needed to gather a lot of data for many processes in a short amount of time without consuming too much of the system resources. There were several scripts developed to try and monitor all the resources on the system, but the final design ended on a combination of psutil and user defined methods to improve performance.

When the script was finalized and thoroughly tested in the development environment it was deployed on two of the compilation servers in the production environment. Here it ran for 4 weeks to gather a large enough data set to work on. During this time period the monitoring script collected data from 15 different users and collected on average 1.5 million data points each day. This gave a working set of data with 41 million data points in total for user analyzing and 75 000 data points for server analyzing.
4.1.1 User monitoring

The user monitoring script was using too much system resources because of psutil when gathering data. The goal of the script was to be light weight to not consume too much of the system resources used by compilation jobs. The script had to be profiled to find the bottleneck of the script and it showed that the method `psutil.get_children(recursive=True)` was slowing down the script. The method would traverse through all processes on the system each time it was trying to find a child process that belonged to the parent process. By filtering processes run by users and group them together in our own method it was possible to significantly improve the performance of the script. Instead of looking through all processes on the system to find child processes, the script now only needed to run through the processes that where executed by users. The script went from using 6-7% CPU down to 1% CPU.

The script was pushed to the production environment when most of the bugs in the script had been fixed in the development environment. The script created can be found in Appendix A.

4.1.2 Server monitoring

The server monitoring script was easier to create and a lot more lightweight since it runs once each minute and only pushes one row to the database for each run. Both the user monitoring and server monitoring scripts were added to `/etc/init.d` to assure that the scripts were running in the background of the server.

The script created can be found in Appendix A.

4.2 Database

With 41 million raw data rows in the database, an efficient way of selecting the data and working on it had to be found. A python script working on 1000 rows at a time was used with different selective statements to find the interesting usable data to analyse. With so much data gathered and a large
data set, it would be most sufficient to only calculate results from the data once and store a simplified result in a new table in the database which the load balancing algorithm will use. When new data is gathered it will be analyzed and the value of the results will then be calculated into the new tables values. By doing this we can have easy access to the important data in a short run-time and be able to have a trigger in the database to remove already calculated data from jSAMPLE and sSAMPLE when they exceed one month in age.

For reading data from the database to be used in the analysis, a script was used that was constantly altered to fit our needs depending on what section of the data that was to be analysed. The script generated user profiles, graphs with pyplot and csv files to be used for graphs. The graphs and values found in the following sections are gathered by this script. The script can be found in Appendix F but it’s a dynamic script that is being altered for user specific use.

4.3 System resources analysis

In this section we look at the data from the database to find the bottlenecks in the system and user patterns to help in the design of the algorithm. The first step is to verify the credibility of the data and to check consistency. Comparing the server data and the user data will be done to check if they comply each other.

4.3.1 CPU

The first step is to check if the CPU data is as expected when looking at the server. Then we take a look at the total CPU usage for the users and see if they comply with the server data.

To check the server CPU data we take a selection from a week and plot it in a graph to see the trends. Here we expect the CPU load to be high during office hours and low during nighttime and weekends.
As seen from the graph 4.1 there is much activities during the daytime in week days and not much activity during the night nor in weekends. The spikes during the nighttime are caused by scheduled maintenance on the system. We want to compare this graph collected from a server to the load generated by users which we have monitored to make sure that they match.

Figure 4.1: One week of server load

Figure 4.2: One week of user load
By comparing the graph for server monitoring 4.1 with the graph from user monitoring 4.2, we find that the general trending of the graph appears to be the same, meaning that our data collection is correct. However, there are some discrepancies in the graph which are caused by two things. The first being that we monitor user data every second and used a larger data set to generate the user graph while only having one data point each minute for the server graph. The other issue is that the server monitoring script includes all processes running on the system and not just processes started by a user as the user monitoring script does. This causes the server graph to be more accurate on the total server load since it includes all the system processes in the monitoring.

Even tho the graphs are not exactly the same, as expected due to the scripts monitoring different processes, the general trends and high peaks of CPU load on the system are displayed at the same points in the graph. We can therefore conclude that both are an accurate representation of the CPU load on the server and we can use this data to find what is happening on the system during the congestion points (i.e., when CPU load is 100%). It is easier to see the congestion points when looking at a smaller time frame, so we narrow it down to just looking at the CPU load on a server for an average work day (8 AM to 5 PM).

![CPU for One Day](image)

Figure 4.3: One work day of CPU load
From the graph 4.3 we can see that the congestion in the system appears before and after the lunch break and in the morning for this particular day. At a few points the CPU is running high and hits the maximum capacity. Looking closer at the data we have collected we can see that during these time frames there are multiple users trying to compile jobs and the processes are being slowed down because they have to share the system with other users.

Now that we have located some congestion points and observed that there are multiple users trying to compile that causes the issue, we want to see how much of an impact this has on each users run time and how much they are being slowed down by having to share limited CPU resources. By selecting a test set of jobs with varying run time we can see how much it affects different users. We select a few of the users who are known for running shorter compilation jobs and some who are running compilation jobs over a longer period of time. Then we take a look at how much time the different compilation jobs take under a perfect scenario (i.e., having all system resources all to themselves) and the CPU utilization. We then compare the run time in a perfect scenario to the run time of a shared scenario where they have to share the system resources. It is also interesting to observe how much CPU they are using during a shared scenario and compare it to the CPU utilization in the perfect scenario. This can tell us how much resources they are allocated during a shared scenario compared to the perfect scenario.

<table>
<thead>
<tr>
<th>Job Name</th>
<th>AvgCPU(Perfect)</th>
<th>Time(Perfect)</th>
<th>AvgCPU(shared)</th>
<th>Time(shared)</th>
</tr>
</thead>
<tbody>
<tr>
<td>buildall</td>
<td>38%</td>
<td>20m</td>
<td>30%</td>
<td>27m</td>
</tr>
<tr>
<td>make</td>
<td>40%</td>
<td>13m</td>
<td>27%</td>
<td>20m</td>
</tr>
<tr>
<td>build.py</td>
<td>93%</td>
<td>16m</td>
<td>32%</td>
<td>50m</td>
</tr>
</tbody>
</table>

Table 4.1: CPU usage for jobs

In the table 4.1 we can see a few of the jobs that run on the system. In the perfect scenario the user has asked for how many CPU cores they want to use for the process. Asking for more cores can speed up the process, but it can slow down other users or you can come in conflict with other user requests. If users ask for more than the there are available cores on the sys-
tem they will try to use all CPU resources on the server. If they are using the system for themselves they will get all of it and increase the compilation speed, but during a work day it is not likely that they will be alone on the system.

The table compares a typical compilation where the user has assigned the required resources and one scenario where they are sharing. The perfect scenario numbers don’t change much, but from the observations it is clear that the compilation is slowed down when they are sharing resources with other users and the system is running at maximum capacity. Shorter running jobs who wants to use 20% of the system CPU is slowed down by a few minutes when they have to share resources. Higher demanding jobs who wants to use about half of the system CPU are slowed down by 15 - 30 minutes depending on how much files are being worked on. Jobs that wants to use the entire systems CPU will have a significant loss in compilation time and the extra time is decided by how much they have to share and how much work load they are doing.

In the table we see a very large job called build.py where the build included 32GB of files and the user asked for 32 CPU cores. The user made a mistake here and asked for more cores than available (maximum is 22 cores), but the compilation job was executed out of work hours so the user was assigned all the systems CPU and the project compiled in 16 minutes. The same project has been compiled during office hours and had to share resources with the other users on the system. In that case the compilation time used 50 minutes and it also slowed down all the other users on the system.

4.3.2 CPU results

From our CPU data we find that processes are being slowed down when the system is running at full capacity which happens quite often. For the small jobs it might not seem important that a process uses a minute longer to compile or 5. But when these jobs are compiled multiple times during a day, it accumulates to half hours or an hour. This happens for all users on the system and if each developer has to wait multiple hours a week, the
lost time in compilation adds up and have a very significant impact.

This becomes the most important resource we have to balance in our algorithm. By monitoring the users we can find an average CPU usage or certain patterns in how they use the system. Using this information we can place users together on the servers so the total amount is kept below the maximum capacity so that compilation jobs are not slowed down by server limitations. It can be difficult to find a good method for profiling CPU usage for each user since the CPU load generated by a user is not constant. Users stay logged in on a system the entire day and the CPU load they generate changes during the day from not using anything to running a heavy CPU compilation job which generates a lot of stress on the system.

4.3.3 RAM

From our monitored data of the system we want to find if RAM is a potential bottleneck in the system. While some users compile projects that are CPU heavy, others might compile projects that uses a lot of files and generates heavy load on the RAM with read and write operations. By looking at our data we want to find how much RAM a typical job uses and if there are any users that are heavy users. Another aspect is if this has any impact on the compilation time.

The figure [4.4] shows the changes in the RAM for a typical day. The data kept in RAM which is in use is on average around 10GB but varies from 7GB to 21GB where just a few jobs have a heavy RAM usage during a day. While most jobs reads somewhere between 200MB to 2GB of data while compiling, there are just a few jobs which exceeds this. The rest of the RAM is used for cache which is also considered to be available RAM since cache is purged when other processes require the space. We want to look at the common jobs which uses around 2GB of RAM to see the impact it has on the compilation time. The most important issue we want to check here is if the heaviest RAM users are being slowed down.

The time it takes to read files into the RAM can’t be changed since it depends on the speed of the hard drive and the network transfer speed. The
only way to improve this speed is on consequent runs of the compilation job where the job can read the files from the RAM cache instead of reading from disk. This means that the only issue we can have with RAM is if projects exceeds the amount of RAM available on the servers or if the combination of users exceeds the limit. We take a look at the available RAM for a day to see where the limit is.
In the graph 4.5 we see how much of the RAM that is currently available for use during a day. The available RAM consists mostly of the RAM cache which is considered available for use. Since the RAM actively in use is around 10GB for a day, the rest of almost 90GB is considered available for use. This means that there has to be processes running on the system simultaneously which require 90GB of RAM combined before there is a slow down in the system. There has been no occurrences of this happening in the system. The heaviest RAM job observed in the system used a total of 33GB at one point in the compilation. The system has enough RAM to cover this rare case and slow downs would only occur if there were multiple users running similar RAM heavy jobs at the same time. There is no good way of finding how much this would slow down a process. The job that used 33GB was also using 95% CPU so if there would be two users running on the system asking for the same amount, CPU would be a bottleneck too.

4.3.4 RAM Results

From the monitored data it seems like the system is well provisioned with RAM in regards to the compilation jobs that are currently running on the system. The only problems is if multiple heavy RAM users are working on the same server. In regards to the load balancing algorithm we want to profile users into two different aspects where the first is CPU and the second is RAM. We want to avoid having multiple RAM heavy users on the same server which can be done quite simply in the algorithm. There also exists one last profile which is the case where a user is a heavy user of both CPU and RAM. In these cases we have to create the algorithm in a way so that heavy RAM users are not on the same server and the server has enough available CPU to support them. Since the heavy RAM compilation jobs only happens on rare occasions, the priority will be that the server covers the CPU requirement for the user if there is no other possible way.

4.3.5 DiskIO

The final resource we have been monitoring is the disk read and write operations. When a process start it will read all the required data into RAM from the hard drive located on a NFS server across the network. When the
compilation is finished the generated files from the compilation job is then written to the hard drive in the same manner, it is important to know that write operations are slower than read operations. A compilation job takes programming code and converts it into binary files so for our case there will be more write operations than read operations. Because the servers have such a high amount of RAM which caches so much data, the amount of data required to be read from disk is actually quite low. Only files that has been changed since the last compilation is read into the RAM. If the RAM cache should be cleared because of heavy load on the system, there would be more read operations.

The point of monitoring the disk read and write operations was to figure out if it could be a potential bottleneck and causing longer compilation times. After looking at the monitored data and due to such high amount of RAM in the system, the read operations are already optimized since most of the files are cached in RAM. To put some numbers to this and get it in a perspective we can look at the heaviest RAM job that has been executed in the system during the monitoring. The process used 33GB of the available RAM, but it only read 1GB from the disk. This means 32GB of data was already cached in the RAM in such a heavy job. For jobs of this size RAM cache becomes quite important. Write operations cant be improved with a load balancing algorithm. The only option we have here is to use RAM cache as much as possible to avoid large read and write operations.

### 4.3.6 DiskIO results

Looking at the disk read and write operations it shows that this resource is not a congestion in the compilation time. Even if it was a congestion there is no actual way of improving this resource with the load balancing algorithm since if a process requires to run read or write operations it doesn’t have any other options. The only way we can improve it is by making sure that the compilation jobs has to read as few files as possible from disk and use the RAM cache. This means that the algorithm will try and place users on the last used server as long as the CPU requirements are covered. This is because reading files from disk for a typical compilation job (i.e., 1GB of data) takes a few seconds while having loss in CPU performance prolongs
the compilation time by minutes.

4.4 Results from cache tests

To decide if RAM cache will be an important part of our algorithm we have to figure out how much of an impact cache has on the different compilation jobs. By running our planned tests we get some results which we can analyse to find the impact it has on a compilation job.

4.4.1 Performance improvement

<table>
<thead>
<tr>
<th>Size</th>
<th>Write to disk</th>
<th>Read no cache</th>
<th>Read with cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>200MB</td>
<td>3s</td>
<td>0.5s</td>
<td>0.1s</td>
</tr>
<tr>
<td>1GB</td>
<td>12s</td>
<td>6s</td>
<td>0.5s</td>
</tr>
<tr>
<td>2GB</td>
<td>26s</td>
<td>13s</td>
<td>1s</td>
</tr>
<tr>
<td>7GB</td>
<td>1m21s</td>
<td>40s</td>
<td>3s</td>
</tr>
</tbody>
</table>

Table 4.2: Cached file test

The table[4.2] show the results from our tests. It shows how much time a compilation job spends reading files from the system with and without cache. Since the data is read and written to a NFS server, there is transmission over network included in these tests. While performing the tests, cached files on both the compilation server and the NFS had to be cleared from the cache.

For the smallest read of 200MB we see that utilizing cache has an increase of 5 times. For medium sized job of 1GB the time spent reading files is increased by 12 times. The medium test of 2GB, which represents the largest reads from our monitored data, shows an increase in read by 13 times faster. For the largest job showed here of a 7GB file read, it increases the speed by 13 times.

This is a very significant increase in time spent reading files, but when comparing the disk read time to how long the job is running for, it becomes quite insignificant. An example would be a job that read 2GB of data from
the disk. When reading from cache the process can run 12 seconds faster than when having to read from the NFS storage. If the process itself was running for 30 seconds and we could cut it down to 15 seconds by using the cache, this could have some impact. Specially if the process is running quite often since the time adds up. However, this is not the case for this particular case. The processes that reads up to 2 GB of data lasted for almost an hour in total run time. Improving the run time by 15 seconds when the total time is almost one hour is not a significant amount.

If we look at the job that had 33GB of files in the RAM there would be some different results. If the process read all the data from disk it would take approximately 3m30s by our tests. The process read 32GB of the data directly from RAM since it utilized cache and only 1GB from the disk which we calculated to take 6s. Here the RAM cache increased the run speed of the process by over 3 minutes. The total run time of the process was 16 minutes when using the RAM cache and without it would have been 19 minutes. 3 minutes improvement might not seem as much, but if there can be a small increase in the run time for all the processes in the system by utilizing the RAM cache, it starts to add up and is a significant improvement to all the servers.

4.4.2 How much load on the system is required to clear cache

To answer this question we have to look at how the RAM cache is maintained. The answer is dependant on the size of the file that was stored in the cache and the load on the system. Since cached pages in RAM stay in cache until they are the oldest files the entire cache has to be filled up before the files are purged.

From our observations of the system we see that out of the 100GB of RAM available to all servers, about 80% of this is utilized as cache. For a file of 2GB stored in cache there has to be 78GB of data read to RAM before it is purged from RAM. Since users are not being moved around servers and they use the same server, they get the full use of cache. This means that in the monitored data most compilation jobs don’t have to read data from disk since most of it is already in the RAM. On an average for a week there
is only 8GB of data being read from disk and the rest is already in RAM. This means that in the current state it would take a few weeks to purge a 2GB file from the RAM cache. When users are being load balanced to different servers, the purging will happen a lot faster. On rare occasions there are larger projects built on the servers. The largest observed read 33GB from disk to RAM and using 33% of the available RAM. On these rare occasions about half of the RAM cache is purged which makes a impact on the other users.

By running the tests planned for this section we find that there is no good way of setting a fixed amount of compilation jobs that has to run before the data is purged from cache. The only dependency it has is the amount of data that has been read from disk since the compilation last ran. In other words, we can use the monitoring of servers to decide if our cache is lost since we last ran it. When assigning a user to a server, we can look at the last used server. We check how much data the user read into cache the last run and look at how much data has been read from disk since then. If the data read on the server exceeds 80GB since the last time the user was logged on, we assume that the cache is lost. If its below that threshold, the files might still exist in cache and it can be a viable option to forward the user to that server.

### 4.4.3 Cache results

So what we can conclude from these results which is important for our load balancing algorithm is the following. The algorithm can be designed to set the preferred server to the last used server so that the processes the user is going to run will have the data already in cache. It doesn’t give a major performance boost, but can be helpful over time. But this can never be on the cost of CPU. The user can not be placed on the last used server if that server is already running on maximum capacity in regards to CPU. All processes will have a major performance loss when they have to share CPU, and the time gained from cache is not significant enough to outweigh the importance of CPU.
4.5 Designing the load balancing algorithm

The information from the monitored data shows that there are some favored ways on what resource to design the load balancing algorithm around. The clearest is CPU which decides the compilation time for over 95% of the different jobs running in the system. This means that most users will be balanced with regards to CPU and the most important factor when doing the load balancing is that the server has enough available CPU to cover the users needs. The only other resource we need to keep in mind is RAM cache to achieve the best performance, but never at the cost of CPU.

At this point it might seem like the dynamic round robin algorithm would be a good choice for our case. This algorithm will try and place users on the server with the most available CPU and the server with the fewest users. It includes some predictive methods to take into account the upcoming load on servers. The lacks are that its more likely to move users around which will be a loss in performance in our case since we will have to read data from the hard drive instead of directly from RAM.

4.5.1 General Design

The algorithm should take advantage of the monitored data to make the decisions. At this point we have a rapidly growing database which at some point will run out of space. We don’t require 3 months of data for the algorithm to make the decisions since the users patterns are changing over time and the most resent data is the most accurate for the user resource requirements. By making a script that traverse through the data for the users and generate a predicted load for the user, we can remove old data from the database after it has been analysed and have smaller working sets. For safety measures we keep one month of data in the historical database. A trigger can be used in the database to remove entries that are over one month old.

When deciding where to put a user, the load balancing algorithm has to check the servers and see how much available resources they have. There are a few factors we need to evaluate when looking at the server load. By looking at the last monitored value for each server we get the load on the
system at that moment. We want the load balancing algorithm to be smart and not just look at the current load, but evaluate the load for the last 10 minutes or the last hour so it can make predictions. Another factor will be the amount of users currently on the server and their combined average load.

When the load balancing algorithm knows about the usual load generated by each user and the available load on the servers, it has to profile the users and place them on the server that has their favored system resource. The algorithms first priority is to make sure the user gets a server that has enough available CPU. It will also try and put the user on the last used server if RAM cache is still available.

### 4.5.2 Prediction table

A new table 4.6 in the database is created for holding the calculated average and maximum load generated by a user. A script is implemented and triggered every hour. The script runs through the data in the database which is gathered by the user monitoring script. It calculates the average load the user has generated on the system and the maximum load. Since we want to take advantage of RAM cache the script will find the last used server the user used and note the timestamp for the last run process on the server.

At this point there are two options. If there are no records of the user in the prediction table (i.e., the first run), the script will insert the data into the table in a new row. If the user already exists, the newly gathered data has to be calculated together with the existing data. Since the existing data will be all the historical data combined together, this value should have more weight to it than the new data from the last hour. But the newly gathered data might be a more accurate representation of the users generated load since it represents the new patterns and usage for the user. We have to create an algorithm that merges the numbers together in a way that the prediction table holds the most accurate representation of the typical load of the user.
Combining predicted load

There are a few ways to create this new value and we will present two examples which have their benefits and disadvantages.

**Calculation 1:** will give the historical data and the recent monitored data equal weights and combine them. It can be achieved by adding the values together and dividing the total value by 2 so we get the average. The advantage of this algorithm is that the predicted values for the user adapt faster towards the recently monitored values while still taking into account the historical data. By doing this the prediction table is fast adapting to what is happening in the system, but it could be fast adapting in the wrong direction. As an example, if a user normally generates 40% of CPU load on the system, but for the last hour has only run a short task that generated 10% of CPU load. The calculation would combine these values and set the new predicted load of the user to 25%. This would cause the load balancer to put this user on a server that could handle a load of 25%, but since the user normally uses 40% and there has been a calculation error, the processes running on the server would be slowed down by the CPU load on the system.

**Calculation 2:** is going to give the historical data more weight to avoid problems that would occur in the first calculation. The historical data will be weighted with 95% and the monitored data from the last hour will be
added with a weight of 5%. If we look at the same example from the last calculation, the historical CPU for a user is 40% and in the last hour it is registered that the user has used 10% CPU of the systems resources. Using the method in 4.1 we would get a new value of 39.5%. The calculation has to be adaptive to the value from the resent hour of data monitored. If the newly monitored data is higher then the historical data, 5% of the value from the newly monitored data would be added to the historical value. If the newly monitored data is lower than the historical data, 5% of the value from the newly monitored data will be subtracted from the historical data. Using this method we have a value that is changing towards the monitored data, but at a slower pace to make sure that inconsistencies in user behavior does not affect the system at the same level.

\begin{verbatim}
if monitoredValue > historicalValue:
    newValue = historicalValue + (monitoredValue * 0.05)
else:
    newValue = historicalValue - (monitoredValue * 0.05)
\end{verbatim}

Listing 4.1: Calculation 2

The best option for our algorithm is calculation 2 since it avoids discrepancies in user behavior, but is still adapting towards the changes in user behavior. Weighing newly monitored data with 5% is not necessarily the correct value to assign it and updating the prediction table every hour can be adjusted. These are a part of the starting design for the load balancing algorithm and can be changed to more accurate values based on observations of the algorithms behavior.

\subsection{Available servers}

Since the algorithm will be designed to send a user to the server with the most available CPU, except when the user has RAM cache,a method for finding the server with the most CPU available must be created. The server actual use is available in the server monitor table, but the algorithm should use the user prediction table in addition.

When a user asks the load balancer for a server to be forwarded to, the algorithm should pick the server with the most available CPU resources.
This is achieved by selecting the CPU value for each server from the server monitor table. Since users stay logged in on the servers, the active load on the servers does not represent to potential load on the system. If a user log on during the lunch brake all servers might respond that they have a very low load, but some of them have users already on the system who can become active. To find the server with the least load we combine the monitored server load with the sum of predicted user load for the users active on the server. After combining the server load with the predicted load, the server with the lowest combined value is picked as the server that the user will be forwarded to.

```
for server in serverList:
    activeServerLoad, timestamp = cursor.execute("SELECT CPU, TIMESTAMP FROM sSAMPLE WHERE NAME = %s ORDER BY TIMESTAMP DESC ", server)

    lastHour = timestamp - datetime.timedelta(hours=1)

    userLoad = cursor.execute("SELECT AVG_CPU FROM PREDICTION WHERE LAST_USED_SERVER = %s AND LAST_LOGIN > %s", server, lastHour)

    totalServerLoad = activeServerLoad + sum(userLoad)
```

Listing 4.2: Find CPU server load

The for loop in 4.2 traverses through all the available servers in the load balancing pool and find the active CPU load on the system and adds the value together with the predicted load for all the users active within the last hour on the system. The load balancing algorithm will use the server with the lowest CPU value returned by this code. This is a unique selection for this system and other load balancing algorithms might instead pick the server with the least users and not the server with the lowest potential load.

4.6 Algorithm prototype

When the prediction table is in place and the algorithm has a way of finding the server with the lowest CPU load, a prototype of the algorithm can be created. In this section a prototype for the load balancing algorithm is
designed based on the monitored historical data.

The flow chart 4.7 shows the decision making for the prototype. A user ID is input to the algorithm and the algorithm will find a server for the user. The user can either be put on the server with the lowest load or on the last used server to get the advantage of RAM cache.

Figure 4.7: Flow chart for the prototype
4.6.1 New user

If the user trying to find a server has never been in the system before, the load balancing decision will be to place the user on the server with the lowest CPU load. The CPU load will be calculated by looking at the current server load and the predicted load for the users that are active on the system. When the serve with the lowest load is found, the user is forwarded to the machine. On the next attempt the user logs into the system the user has been added to the system due to the user process monitoring script and data will be available for making the load balancing decision.

4.6.2 Existing users

When an existing user connects to the load balancer and wants a compilation server, the algorithm will look up the user in the prediction table. If the user has been recently active and it’s suspected that the user might have RAM cache available on the last used server, the algorithm will check the RAM for the last used server. The algorithm looks up the timestamp for the user and does some calculations.

The first step is to calculate how much data has been read from disk to the servers RAM since the last login. Two values are fetched from the servers monitoring data. One of them is the diskIN which has been read at the current time and the second is the diskIN the server had at the last login timestamp. By subtracting the diskIN at the last login from the current diskIN value, a difference is created and the value shows how much data that has been read from disk since the last login. Then we compare this value to the amount of cached RAM on the server. If the data read exceeds the available RAM in the system, we assume that the cache for the user has been cleared. If the data read does not exceed the cache, the algorithm looks at the average RAM usage for the user. If the data read from disk and the users RAM usage does not exceed the cache, the algorithm finds that the users cache is still available on the server.

If the algorithm finds that the users cache has been replaced by other data, the algorithm ignores the priority to this server and will find the server in the load balancing pool with the lowest CPU usage.
4.6.3 Cache

If the algorithm finds that the users cache is still available on the last used server, it will check if the server has enough available CPU for the user. The algorithm looks up the servers current CPU load and the predicted load generated by the users active on the system. When looking up the predicted load on the server it gets the predicted CPU load for each user who have been active within the last hour on the server. Since the predicted load generated by the users on the server will include the active load on the server, the algorithm finds which of the values that will be the deciding load. The largest CPU load is picked between the monitored server load and the sum of the predicted user load. If this value in combination with the predicted user CPU load is less than 100 percent, the server is viable for the user and can be used. This means that the user will be forwarded to the last used server it used to keep the cache and the algorithm has made sure that there is enough available CPU in the system. If the calculated CPU load exceeds 100 percent, the server is not viable for the user and the user will be put on the server with the least CPU load.

4.6.4 CPU

If the algorithm finds that the user connecting does not exist in the system, the cache has been replaced or that the last used server with files in cache does not have enough available CPU, the algorithm will put the user on the server with the least amount of CPU load. The server with the least load is found by using the code explained in 4.2. The user will be put on the selected server to spread out the load over all the compilation servers in the load balancing pool.

4.6.5 Prototype usage

The prototype is a starting implementation of the algorithm and not a load balancer. The prototype can take a user ID as a input and will return the name of the server which would be the best option for the user. The user
can then SSH to the server if desired.
Chapter 5

Discussion

During this project, a monitoring tool has been implemented and an algorithm prototype has been designed. This chapter discuss the tools created, their usefulness, improvements and future work.

5.1 Resource monitoring

To design a load balancing algorithm for compilation servers it was important to understand what system resources that are important for a compilation job. The results show that CPU is the most important resource in 95% of the cases and is an important parameter for the design. Since the generated load varies depending on the different users, there is not a correct answer that covers all the cases. The most important resource varies depending on the user, but CPU is a clear favorite.

Since the generated load by users varies and they have different favored resources, the algorithm and system has to be adaptive to cover the need of all the users. This complicates the design, but by having a dynamic and adaptive system it is possible to cover the need for all users.

Based on observations and research, CPU, RAM and DiskIO was monitored for the compilation jobs to find the limiting factor in a system. The results show that CPU and RAM are the deciding factors for compilation jobs which is very reasonable. Network and disk read and write were considered limiting factors. The only scenario where disk or network would
be a limiting factor is if multiple users have to fetch files from the external storage system. Since the algorithm is designed to prioritize RAM cache for the users, the requirement for fetching files externally is lowered.

There are two options for improving disk read and write operations and lowering network traffic. The first option is to improve the hardware. In the first option we are not able to do anything in the algorithm to improve performance. The second option is to avoid moving users between compilation servers so they can use the files already cached in RAM. The second option has been implemented into the algorithm. By prioritizing the last used server for users to utilize files in RAM, the disk IO and network resources have been optimized.

5.2 Monitoring tool

The monitoring tool was designed with compilation servers in mind. Though the monitoring tool was designed for designing an algorithm, the data can be used in more ways.

The user monitoring script gathers information for all the processes a user is running in the system. The script allows for an in-depth view in how a user uses a system. With information on all the commands executed and how much resources these commands consume, the monitored data can be used for more than just a load balancing algorithm:

- An in-depth profiling of each user
- User activity and workload
- All the commands executed by a user
- Finding errors caused by users

The server monitoring script gathers information for all the processes running on the system. This includes both user processes and the system processes giving a total overview of the entire server resource load. The
information was very valuable when designing the algorithm since it displayed how the compilation servers resources were utilized. This information was used to design the algorithm and can give a clear view of the activity on a server. The data was used to create CPU trends for the servers which can be used to find the servers that are overloaded and the servers that can handle more load.

By pushing all the data that is being monitored to a database, a historical view for servers and users is available. Changes in user patterns or server load can be found and graphed for reports or to be used by administrators.

The monitoring tool has great value by itself. Since the historical data is available in a database, graphical tools or web pages can be created and get the historical data from the database. This can be used by system administrators to observe the servers in a GUI and compare system or user workload over time. Since the system is constantly pushing activity to the database, the data can be used to show live trends from servers and users.

5.2.1 Improvements

After working with the tool and analyzing the data, there are changes that could be made to improve the monitoring tool.

There is one more parameter in the system that could have been interesting for profiling the user jobs. When analyzing the compilation jobs we looked at the impact that reading and writing to disk had. The processes has a parameter called \texttt{IOwait}. \texttt{IOwait} happens when a process has to pause the CPU computation to read files from disk or write files to disk when they are not found in RAM. The \texttt{IOwait} parameter would have made the analyzing of processes easier by showing when computation was halted because the process required files from disk. It was possible to analyse the data without this parameter, but the parameter can be added to future work.

The monitoring tool has been specifically designed for homogeneous servers running in a Unix environment. \texttt{Psutil} supports cross-platform
architecture making it possible to make the monitoring scripts usable for all operating systems. All the compilation servers were running the same hardware thus making the percentage for RAM and CPU the same on all the servers. In a heterogeneous environment these values can be calculated differently. RAM can use measurements in gigabytes instead of a percentage. CPU could either allow a value over 100% where each 100 would represent one core or CPU flops could be used to calculate the CPU consumption on the different servers. Making the monitoring tool more dynamic will be future work.

Currently the user monitoring script is running each second and the server monitoring script is running once per minute. The user monitoring script had to be run every second to make sure that all information about the running processes was gathered. When implementing the server monitoring script it was decided that a measurement each minute would be sufficient. The monitored data gave enough information to design an algorithm, but it would have been easier to compare the data from the scripts if the server was monitored each second instead of each minute. This is a simple fix in the user monitoring script.

5.3 Cache in RAM

Results from tests on the RAM showed that compilation time was reduced when the process read files from RAM instead of the external storage system. The reduction in time varied depending on the size of the compilation job. The time saved is significant enough in itself to implement a priority to the last used server in the algorithm.

What makes the RAM cache even more valuable is the fact that it reduces disk IO and network traffic, making sure that neither of these resources becomes a bottleneck. Optimally the users are not moved around at all to keep RAM cache available and keeping disk IO and network traffic to a minimum. If the system is running at full capacity in regards to CPU, it will be better for everyone to move a user to a different server. The main goal for the load balancing algorithm will be to spread all the users load evenly on all compilation servers and enter a static state where users are
not moved at all. The load balancing algorithm will have to run for some time before this state is found and some more advanced methods needs to be implemented into the algorithm to make this happen.

5.3.1 Improvements

The algorithm tries to calculate if there is still RAM cache available for a user on the last used server. This has been achieved with a simple calculation. The calculation assumes that RAM cache is not overwritten before all the other available RAM has been used. This a simple prediction and it might not be accurate because of the way pages are cleared from RAM. Even tho the pages with the highest pressure is removed first from RAM, it is also dependent that these pages match the size of the incoming data. If incoming files require more space in RAM then a certain page allows, it will look for the pages that fit the size and clear them from cache even if other pages have a higher pressure. More research should be conducted on how to predict if the cache is available and be implemented into the algorithm.

5.4 Algorithm prototype

The prototype is utilizing the results from analyzing resources and uses the information to make decisions. It is specifically designed for this compilation server environment with some basic functionality. The algorithm can be applied to any compilation environment and would work for servers who are running other processes too. The algorithm is designed to divide load based on the historical data of users and it will work on any server infrastructure where users are generating load on systems and not just compilation processes.

The prototype has room for improvement to make it more adaptive to changes in the system and making better predictions. There were more methods planned for the algorithm than what has been implemented at this point in time.
5.4.1 Improvements and future work

Profiling users into their favored compilation resource. The results has shown that there are three possible resources which should decide where a user is placed. They are CPU, RAM and the combination of the two. The users who favors CPU should be balanced based on this resource, but keeping the RAM cache is a benefit. RAM users can be placed together with the heavy CPU users since they use different resources. The heavy jobs that require both CPU and RAM might require a server to themselves if their requirements are high enough.

In most cases, CPU and RAM are not mutually exclusive since heavy CPU usage means that the compilation job is working on a lot of files. Heavy RAM jobs on the other hand do occur without heavy CPU usage. There are jobs that reading and writing files and not doing much computation on the files.

By using user profiling the load balancing algorithm could place all the users in advanced based on their profile. The algorithm would then try and spread out the users to make sure that every compilation server gets an equal amount of load. This would be more like a puzzle based algorithm that placed the users that fit together on servers to balance out the load and it would be easier to create an optimal static state.

The prediction method used for calculating the predicted user load is merging the historical data with the recently monitored data. The values has been set to weight the historical data with 95% and the monitored data with 5% when they merge. This was done to avoid that newly monitored data had too much impact on the historical data. In some cases when users are moved to a new project, there might be a big change in a users pattern and the prediction method can be slow to adapt to these changes. This method can be extended to to better handle wast changes in user patterns. This will be done in future work.

The method that finds the server with the lowest CPU load is using the recently monitored value for the server. The method can be extended to use more of the recent historical data to predict if the general server load
is going up or down. This can be achieved by using the last 10 minutes or the last hour of load on the system to predict the server load. The historical data can be used to look at previous days and use the timestamps from the data to match the trend from previous days. If the algorithm knows that load on the system is usually lower during lunch breaks, it can use this historical data to predict if the system load is going up or down. This will be added to the future work.
Chapter 6

Conclusion

This thesis aims to distribute system load evenly between compilation servers using historical data. A monitoring tool was created to monitor servers and users in the compilation server environment at a process level. The monitored data was stored in an historical database where it was analysed. The results from the analysis contributed to creating a load balancing algorithm prototype for the system.

The conduction of this thesis resulted in the following findings:

- Monitoring tool for observing users and server resource usage on multiple compilation servers.
- Explanation of system resources and the important parameters to monitor for a compilation server.
- Analysis of historical data for users and servers to find the limiting factors in regards to compilation tasks.
- The importance and improvements of reading files from RAM for a compilation task instead of reading files from a hard drive.
- Using the historical data and the results from the analysis to design a load balancing algorithm prototype.

The implemented monitoring tool gives unique in-depth information about the users using the servers. By utilizing the information from the historical database, a load balancing algorithm prototype was designed with the goal of improving overall performance for multiple compilation
servers. The monitoring tool can be applied to multiple use cases. The algorithm prototype can be developed further and be implemented in the compilation server environment.
Bibliography


Appendix

The appendix includes 6 scripts that were developed as part of this project. The scripts are used for monitoring, creating a database, analysing the data in the database and a prototype for a load balancing algorithm.

Appendix A: User monitoring script
The user monitoring script is a python script that is designed to run every second on a server. The script monitors user activity on the system and collects data for all the processes the users run. The script pushes the data to a database.

Appendix B: Server monitoring script
The server monitoring script is a python script that is designed to run once every minute on a server. The script monitors the server resources and the load on the system. The script pushes the collected data to a database.

Appendix C: Create database script
The create database script is a SQL script that generates a database schema and 5 tables. The script was used to create the storage database for the user and server monitoring scripts.

Appendix D: Prediction method
The prediction method is a method in a python script. Data is gathered with the user and monitoring scripts and pushed to a database. This method uses the data gathered by the monitoring scripts and creates a summary for the users. The summary is inserted into the prediction table.

Appendix E: Algorithm prototype
The algorithm prototype is a python script that uses some of the concepts
in the thesis to find the optimal server for a user. The prototype is still in development and is making decisions based on the historical and monitored data.

**Appendix F: Dynamic script**

The dynamic script is python code that was used to look at specific data in the database and generate graphs and cvs files. The script was constantly changed to gather different data that was required to answer the questions in the thesis. It was used to look at CPU trends, specific jobs, RAM trends and analyzing system behavior.
Appendix A

User Monitoring Script

```python
import os, datetime, psutil, time, subprocess
import socket
import MySQLdb
import logging

# Connect to database
try:
    db = MySQLdb.connect(host="lb", user="lb", passwd="lb", db="MONITOR")
except Exception, e:
    print 1, repr(e)

# GLOBAL VARIABLES
excludeUser = ['root', 'avahi', 'dbus', 'polkitd', 'rpc', 'rpcuser', 'postfix']
excludeProcess = ['sshd', 'bash']
hostname = socket.gethostname()
cursor = db.cursor()
numOfCPU = psutil.cpu_count()
storeTime = {}

# Convert Bytes to Megabytes
def bytesToMegabytes(n):
    return (n / 1024) / 1024

# Recursive algorithm to find all children of a process
#@profile
def getAllChildren(childList, parentPid):
    completeList = []
    for item in childList:
        ppid = int(item.split(' ')[0])
```

pid = int(item.split(' ')[1])
if ppid == parentPid:
    completeList.append(pid)
    completeList += getAllChildren(childList, pid)
return completeList

# Add pid to psutil process class
def addPID(pid):
    try:
        psutil.Process(pid)
    except psutil.NoSuchProcess:
        pass

# Get all processes run by a user
#@profile
def processList2():
    processList = []
    for proc in psutil.process_iter():
        try:
            real, effective, saved = proc.uids()
        except psutil.NoSuchProcess:
            pass
        if real >= 1000:
            processList.append(proc)
    return processList

def getProcessList():
    processList = []

    pids = [pid for pid in os.listdir('/proc') if pid.isdigit()]

    for pid in pids:
        try:
            filename = '/proc/' + str(pid)
            if os.stat(filename).st_uid >= 1000:
                try:
                    Process(int(pid))
                except psutil.NoSuchProcess:
                    pass
                except OSError:
                    pass
        return processList
# Generates lists of pid, ppid and a combined list
#@profile
def getPidLists(processList):
    pidList, ppidList, allChildren = [], [], []

    for proc in processList:
        try:
            pinfo = proc.as_dict(attrs=['pid', 'name', 'ppid'])
        except psutil.NoSuchProcess:
            pass
        else:
            if pinfo['name'] not in excludeProcess:
                pid = pinfo['pid']
                ppid = pinfo['ppid']
                pidList.append(pid)
                ppidList.append(ppid)
                myStr = str(ppid) + ' ' + str(pid)
                allChildren.append(myStr)

    return pidList, ppidList, allChildren

# Creates a dictionary of key[parent pid] and all its value as children pids
#@profile
def processDict(pidList, ppidList, allChildren):
    parent, childOfParent = [], []
    localDict = {}

    for item in allChildren:
        if int(item.split(' ')[0]) not in pidList:
            parent.append(int(item.split(' ')[1]))
        if int(item.split(' ')[1]) not in ppidList:
            childOfParent.append(item)

    for pid in parent:
        localDict[pid] = getAllChildren(allChildren, pid)

    return localDict

# Returns a dictionary of parent pid and all its children as a process class
#@profile
```python
def getCompleteDict(processDict, processList):
    finalDict = []
    for key, value in processDict.items():
        myList = []
        myProc = ''
        for proc in processList:
            try:
                pinfo = proc.as_dict(attrs=['ppid', 'pid', 'name'])
            except psutil.NoSuchProcess:
                print('Fail:', pinfo)
                pass
            else:
                if key == pinfo['ppid']:
                    myProc = proc
                if pinfo['pid'] in value:
                    myList.append(proc)
            if myProc != '':
                finalDict[myProc] = myList

    return finalDict

# Calculates process system resource usage
#@profile
def calculateResources(processDict):

    # For each main process and all its children
    for proc, children in processDict.items():
        try:
            childrenList = children

            # Get user data and resource data for proc and resource data for children
            timestamp, real, username, name, pid, cmd = getProcData(proc)
            procCPU, procRAM, procss, procvms, procReadIO, procWriteIO = getProcResources(proc, pid)
            childCPU, childRAM, childss, childvms, childReadIO, childWriteIO = getChildData(childrenList)

            # Sum proc and children
            sumCPU = round((procCPU + childCPU)/numOfCPU, 3)
```

sumRAM = round((procRAM + childRAM), 3)

rss = bytesToMegabytes(procRSS + childRSS)
vms = bytesToMegabytes(procVMS + childVMS)

ReadIO = procReadIO + childReadIO
WriteIO = procWriteIO + childWriteIO

# Add the user, job and sample to database if not exists
addUser(real, username)
addJob(pid, real, timestamp, name, cmd)

addSample(pid, real, timestamp, sumCPU, sumRAM, rss, vms,
ReadIO, WriteIO)

db.commit()
except psutil.NoSuchProcess:
    pass

# Remove outdated processes
cleanupStoreTime(processDict)

# @profile
def getProcData(proc):
    try:
        timestamp = datetime.datetime.fromtimestamp(proc.create_time()).strftime("%Y-%m-%d %H:%M:%S")
        real, effective, saved = proc.uids()
        username = proc.username()
        name = proc.name()
        pid = proc.pid
        cmd = ' '.join(proc.cmdline())
    except psutil.NoSuchProcess:
        pass

    return timestamp, real, username, name, pid, cmd

# @profile
def getProcResources(proc, pid):
    try:
        rIO, wIO, rB, wB = proc.get_io_counters()
        rss, vms = proc.memory_info()

        sumCPU = proc.cpu_percent()
        sumRAM = proc.get_memory_percent()
        sumRAMrss = rss
        sumRAMvms = vms
sumReadIO = rIO
sumWriteIO = wIO

except Exception, e:
    print 12, repr(e)
return sumCPU, sumRAM, sumRAMrss, sumRAMvms, sumReadIO, sumWriteIO

# @profile
def getChildData(childrenList):
    sumCPU = 0.0
    sumRAM = 0.0
    sumRAMrss = 0
    sumRAMvms = 0
    sumReadIO = 0
    sumWriteIO = 0
    for child in childrenList:
        if child.is_running():
            try:
                pid = child.pid
                addPID(pid)
                rIO, wIO, rB, wB = child.get_io_counters()
                rss, vms = child.memory_info()
                cpu = child.cpu_percent()
                sumCPU += cpu
                sumRAM += child.get_memory_percent()
                sumRAMrss += rss
                sumRAMvms += vms
                sumReadIO += rIO
                sumWriteIO += wIO
            except Exception, e:
                print 12, repr(e)
    return sumCPU, sumRAM, sumRAMrss, sumRAMvms, sumReadIO, sumWriteIO

# @profile
def cleanupStoreTime(pDict):
    allKeys = []
    for key, value in pDict.iteritems():
        allKeys.append(key)
for key, value in list(storeTime.items()):
    if key not in allKeys:
        del storeTime[key]

# @profile
def addServer(hostname):
    try:
        server = cursor.execute("SELECT NAME FROM SERVER WHERE NAME = %s", hostname)
        except Exception, e:
            print 10, repr(e)

        if server == 0:
            try:
                cursor.execute("INSERT INTO SERVER (NAME) VALUES (%s)", hostname)

                except Exception, e:
                    print 11, repr(e)

# @profile
def addUser(real, username):
    try:
        user = cursor.execute("SELECT UID FROM USER WHERE UID = %s AND NAME = %s", (real, username))

        except Exception, e:
            print 2, repr(e)

        if user == 0:
            try:
                cursor.execute("INSERT INTO USER (UID, NAME, SERVER) VALUES (%s, %s, %s)", (real, username, hostname))

                except Exception, e:
                    print 3, repr(e)

            else:
                try:
                    cursor.execute("UPDATE USER SET SERVER = %s WHERE UID = %s AND NAME = %s", (hostname, real, username))

                    except Exception, e:
                        print 4, repr(e)

# @profile
def addJob(pid, real, timestamp, name, cmd):
try:
    isRunning = cursor.execute("SELECT PID FROM JOB WHERE UID = %s AND PID = %s AND START_TIME = %s", (real, pid, timestamp))
    except Exception, e:
        print 5, repr(e)

if isRunning == 0:
    try:
        cursor.execute("INSERT INTO JOB (PID, UID, START_TIME, CMD_NAME, COMMAND, SERVER) VALUES (%s, %s, %s, %s, %s, %s)", (pid, real, timestamp, name, cmd, hostname))
        except Exception, e:
            print 6, repr(e)

# @profile
def addSample(pid, real, timestamp, CPU, RAM, rss, vms, ReadIO, WriteIO):
    try:
        cursor.execute("INSERT INTO jSAMPLE (PID, UID, START_TIME, CPU, RAM, RSS, RAM_VMS, DISK_IN, DISK_OUT) VALUES (%s, %s, %s, %s, %s, %s, %s, %s, %s)", (pid, real, timestamp, CPU, RAM, rss, vms, ReadIO, WriteIO))
        except Exception, e:
            print 7, repr(e)

logging.basicConfig(level=logging.DEBUG, filename=’/etc/lbmonitor/process.log’)  
if __name__ == '__main__':
    # count = 0
    while (True):
        try:
            addServer(hostname)

            myDict = {}
            processList = processList2()
            pidList, ppidList, childrenList = getPidLists(processList)
            myDict = processDict(pidList, ppidList, childrenList)
            completeDict = getCompleteDict(myDict, processList)

            calculateResources(completeDict)
        except:
logging.exception("Error: ")

time.sleep(1)

# count += 1

pictures/processMonitor.py
import psutil
import socket
import pwd, os, time
import subprocess
import MySQLdb
import logging

try:
    db = MySQLdb.connect(host="lb", user="lb", passwd="lb", db = "MONITOR")
except Exception, e:
    print 1, repr(e)
cursor = db.cursor()
hostname = socket.gethostname()

def bytesToMegabytes(n):
    return (n / 1024) / 1024

def getUsers():
    users = subprocess.check_output("who")
    return set([x.split()[0] for x in users.splitlines()])

def getCPU():
    numOfCPU = psutil.cpu_count()
    usageCPU = psutil.cpu_percent()
    currentUsage = usageCPU / numOfCPU
    return round(usageCPU, 3)

def getRAM():
tot, available, percent, used, free, active, inactive, buff, cache = psutil.virtual_memory()
def getIO():
    read_count, write_count, read_bytes, write_bytes, read_time, write_time = psutil.disk_io_counters()
    return read_bytes, write_bytes

def getNET():
    bytes_sent, bytes_recv, packets_sent, packets_recv, errin, errout, dropin, dropout = psutil.net_io_counters()
    return round(bytes_sent, 9), round(bytes_recv, 9)

def updateServer():
    cpu = getCPU()
    read_count, write_count = getIO()
    ram, available, cache = getRAM()
    # bytes_sent, bytes_recv = getNET()  # print read_bytes
    # print write_bytes

    try:
        server = cursor.execute("SELECT NAME FROM SERVER WHERE NAME = %s", (hostname))
    except Exception, e:
        print 2, repr(e)

    if server == 0:
        try:
            cursor.execute("INSERT INTO SERVER (NAME) VALUES (%s)", (hostname))
        except Exception, e:
            print 3, repr(e)

        try:
            cursor.execute("INSERT INTO sSAMPLE (NAME, CPU, RAM, RAM_AVAILABLE, RAM_CACHED, DISK_IN, DISK_OUT) VALUES (%s,%s,%s,%s,%s,%s,%s)", (hostname, cpu, ram, available, cache, read_count, write_count))
        except Exception, e:
            print 4, repr(e)
        db.commit()  

    #def findProcess():
    #    if new_process_started for user:
# return pid

def monitorCPU(pid):
    # perf cpu for pid

def monitorRAM(pid):
    # perf ram for pid

def monitorIO(pid):
    # perf io for pid

def monitorNet(pid):
    # perf netIO for pid

def sendToServer(IP):
    # transfer data<user, cpu, ram, io, net>

logging.basicConfig(level=logging.DEBUG, filename='/home/dsunde/master/server.log')

if __name__ == '__main__':
    while (True):
        try:
            updateServer()
        except:
            logging.exception("Error: ")
time.sleep(60)
Appendix C

Create Database Script

---

--- Schema MONITOR
---

CREATE SCHEMA IF NOT EXISTS 'MONITOR' DEFAULT CHARACTER SET utf8 COLLATE utf8_general_ci;
USE 'MONITOR';
---

--- Table 'MONITOR'. 'SERVER'
---

CREATE TABLE IF NOT EXISTS 'MONITOR'. 'SERVER' (  
'id' INT NOT NULL AUTO_INCREMENT,  
'NAME' VARCHAR(45) NULL,  
PRIMARY KEY ('id'),  
INDEX 'NAME_IDX' ('NAME' ASC)  
ENGINE = InnoDB;
---

--- Table 'MONITOR'. 'USER'
---

CREATE TABLE IF NOT EXISTS 'MONITOR'. 'USER' (  
'id' INT NOT NULL AUTO_INCREMENT,  
'UID' INT NULL,  
'NAME' VARCHAR(45) NULL,  
'SERVER' VARCHAR(45) NULL,  
PRIMARY KEY ('id'),  
UNIQUE INDEX 'USER_IDX' ('UID ASC', 'NAME ASC'),  
INDEX 'SERVER_FK_IDX' ('SERVER ASC'),  
CONSTRAINT 'USER_SERVER_FK'  
FOREIGN KEY ('SERVER')
---

93
REFERENCES 'MONITOR' . 'SERVER' ('NAME')
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
ENGINE = InnoDB;

---
---
---

CREATE TABLE IF NOT EXISTS 'MONITOR' . 'JOB' ( id INT NOT NULL AUTO_INCREMENT,
    PID INT NULL,
    UID INT NULL,
    START_TIME DATETIME NULL,
    CMD_NAME VARCHAR(45) NULL,
    COMMAND TINYTEXT NULL,
    SERVER VARCHAR(45) NULL,
    PRIMARY KEY (id),
    UNIQUE INDEX 'P_IDX' (PID ASC, UID ASC, START_TIME ASC),
INDEX 'UID_FK_IDX' (UID ASC),
INDEX 'SERVER_FK_IDX' (SERVER ASC),
CONSTRAINT 'JOB_UID_FK'
    FOREIGN KEY (UID)
    REFERENCES 'MONITOR' . 'USER' (UID)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION,
CONSTRAINT 'JOB_SERVER_FK'
    FOREIGN KEY (SERVER)
    REFERENCES 'MONITOR' . 'SERVER' (NAME)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
ENGINE = InnoDB;

---
---
---

CREATE TABLE IF NOT EXISTS 'MONITOR' . 'jSAMPLE' ( id INT NOT NULL AUTO_INCREMENT,
    PID INT NULL,
    UID INT NULL,
    START_TIME DATETIME NULL,
    RUN_TIME TIMESTAMP NULL DEFAULT CURRENT_TIMESTAMP,
    CPU DECIMAL(6,3) NULL,
    RAM DECIMAL(6,3) NULL,
CREATE TABLE IF NOT EXISTS `MONITOR`.`sSAMPLE` (    `id` INT NOT NULL AUTO_INCREMENT,    `NAME` VARCHAR(45) NULL,    `TIMESTAMP` TIMESTAMP NULL DEFAULT CURRENT_TIMESTAMP,    `CPU` DECIMAL(6,3) NULL,    `RAM` DECIMAL(6,3) NULL,    `RAM_AVAILABLE` INT NULL,    `RAM_CACHED` INT NULL,    `DISK_IN` INT NULL,    `DISK_OUT` INT NULL,    PRIMARY KEY (`id`),    INDEX `SERVER_FK_IDX` (`NAME` ASC),    CONSTRAINT `sSAMPLE_SERVER_FK`    FOREIGN KEY (`NAME`)    REFERENCES `MONITOR`.`SERVER` (`NAME`)    ON DELETE NO ACTION    ON UPDATE NO ACTION) ENGINE = InnoDB;
Appendix D

Prediction method

```python
def predictUserLoad(userID):
    # A test to see if the user already exists in the Prediction table
    sqlSelectUser = ("SELECT UID FROM PREDICTION WHERE UID = %s", userID)
    userExists = cursor.execute(sqlSelectUser)

    # Get the monitored data for the user from the user monitor table
    sqlSelect = ("SELECT MAX(CPU), MAX(RAM), AVG(CPU), AVG(RAM), MAX(RUN_TIME) FROM jSAMPLE WHERE UID = %s", userID)
    maxCPU, maxRAM, avgCPU, avgRAM, runTime = cursor.execute(sqlSelect)

    # Get the user name and which server the user was last logged in to
    sqlSelect = ("SELECT NAME, SERVER FROM USER WHERE UID = %s", userID)
    name, server = cursor.execute(sqlSelect)

    # If the user doesn’t exist we insert the new user with the data from the user monitor table
    if userExists == 0:
        sqlInsert = ("INSERT INTO PREDICTION (UID, USER_NAME, LAST_USED_SERVER, LAST_LOGIN, AVG_CPU, MAX_CPU, AVG_RAM, MAX_RAM) VALUES = (%s, %s, %s, %s, %s, %s, %s)", (userID, name, server, runTime, avgCPU, maxCPU, avgRAM, maxRAM))
        cursor.execute(sqlInsert)

    # If the user already exists in the prediction table we combine the existing data with the new monitored data
```
else:
    sqlSelect = ("SELECT AVG_CPU, MAX_CPU, AVG_RAM, MAX_RAM FROM PREDICTION WHERE UID = %s", userID)
    preAvgCPU, preMaxCPU, preAvgRAM, preMaxRAM = cursor.execute(sqlSelect)

    # We want to adjust the value of average CPU for the user in the prediction table by using an algorithm that can adjust the value based on the difference
    difference = avgCPU - preAvgCPU
    newAvgCPU = changeAlgorithm(avgCPU, preAvgCPU, difference)

    # We want to adjust the value of average RAM for the user in the prediction table by using an algorithm that can adjust the value based on the difference
    difference = avgRAM - preAvgRAM
    newAvgRAM = changeAlgorithm(avgRAM, preAvgRAM, difference)

    # We check if the user have heavier CPU jobs running in the system
    if maxCPU > preMaxCPU:
        newMaxCPU = maxCPU
    else:
        newMaxCPU = preMaxCPU

    # Check if the user have heavier RAM jobs running in the system
    if maxRAM > preMaxRAM:
        newMaxRAM = maxRAM
    else:
        newMaxRAM = preMaxRAM

    # Update the prediction table with the new calculated data
    sqlInsert = ("UPDATE PREDICTION SET LAST_USED_SERVER = %s, LAST_LOGIN = %s, AVG_CPU = %s, MAX_CPU = %s, AVG_RAM = %s, MAX_RAM = %s WHERE UID = %s", (server, runTime, newAvgCPU, newMaxCPU, newAvgRAM, newMaxRAM, userID))

    # Clean up the user monitor table
    sqlDelete = ("DELETE FROM jSAMPLE WHERE UID = %s AND RUN_TIME < %s", (userID, runTime))
Appendix E

Algorithm prototype

```python
# Prototype of the load balancing algorithm
import MySQLdb
import sys
import datetime

# Connect to database
try:
    db = MySQLdb.connect(host="lb", user="lb", passwd="lb", db="MONITOR", cursorclass=MySQLdb.cursors.SSCursor)
except Exception, e:
    print 1, repr(e)

# Convert bytes to megabytes
def bytesToMegabytes(n):
    return (n / 1024) / 1024

# Check if the user already exists in the prediction table
def existingUser(userID):
    cursor = db.cursor()
    try:
        userExists = cursor.execute("SELECT USER_NAME FROM PREDICTION WHERE UID = %s", userID)
    except Exception, e:
        print 2, repr(e)
    cursor.close()
    if userExists == 0:
        return False
    else:
        return True
```
def lastUsedServer(userID):
    cursor = db.cursor()
    try:
        lastServer, lastLogin = cursor.execute("SELECT LAST_USED_SERVER, LAST_LOGIN FROM PREDICTION WHERE UID = %s", userID)
    except Exception, e:
        print 2, repr(e)
    cursor.close()
    return lastServer, lastLogin

def serverRAMload(server, timestamp):
    cursor = db.cursor()
    try:
        minDiskIn, maxDiskIn = cursor.execute("SELECT MIN(DISK_IN), MAX(DISK_IN) FROM sSAMPLE WHERE NAME = %s AND TIMESTAMP > %s", (server, timestamp))
    except Exception, e:
        print 2, repr(e)
    cursor.close()

    cursor = db.cursor()
    try:
        cachedRAM = cursor.execute("SELECT RAM_CACHED FROM sSAMPLE WHERE NAME = %s AND DISK_IN > %s", (server, maxDiskIn))
    except Exception, e:
        print 2, repr(e)
    cursor.close()

    diskDifference = maxDiskIn - minDiskIn
    return diskDifference, cachedRAM

def getUserAvgRAM(userID):
    cursor = db.cursor()
    try:
        cursor.execute("SELECT RAM_CACHED FROM sSAMPLE WHERE NAME = %s AND DISK_IN > %s", (server, maxDiskIn))
    except Exception, e:
        print 2, repr(e)
    cursor.close()
avgRAM = cursor.execute("SELECT AVG_RAM FROM PREDICTION WHERE UID = %s", userID)
except Exception, e:
    print 2, repr(e)
cursor.close()
return avgRAM

# Calculate to see if the server might have the user data left in cache
def calculateCache(serverLoad, userLoad, cache):
    serverLoad = bytesToMegabytes(serverLoad)

    # To find how much RAM is still cached we remove the disk read load on the server from the total RAM
    cacheNotReplaced = cache - serverLoad

    # If the user load on RAM is less than what has not been replaced on the server, we can assume that there is still cache available
    if cacheNotReplaced > userLoad:
        return True
    else:
        return False

# Find the active load on the server and the predicted load
def getServerCPU(server):
    # Get the current CPU load on the server
    cursor = db.cursor()
    try:
        serverCPU, timestamp = cursor.execute("SELECT CPU, TIMESTAMP FROM sSAMPLE WHERE NAME = %s ORDER BY TIMESTAMP DESC", server)
    except Exception, e:
        print 2, repr(e)
cursor.close()

    lastHour = timestamp - datetime.timedelta(hours=1)

    # Get the predicted load on the server from the users currently active
    cursor = db.cursor()
    try:
        userCPU = cursor.execute("SELECT AVG_CPU FROM PREDICTION WHERE LAST_USED_SERVER = %s AND LAST_LOGIN > %s", server, lastHour)
    except Exception, e:
# Since active users represents the current load on the system
# we compare which is the largest of them (predicted vs active)
# and chose the largest to challenge the user

if serverCPU > sum(userCPU):
    return serverCPU
else:
    return sum(userCPU)

# Get the predicted CPU load for the user

def getUserCPU(userID):
    cursor = db.cursor()
    try:
        userCPU = cursor.execute("SELECT AVG_CPU FROM PREDICTION WHERE UID = %s", userID)
    except Exception, e:
        print 2, repr(e)
    cursor.close()

    return userCPU

# Find the server with the most available CPU

def leastLoadServer():
    # Find all the servers
    cursor = db.cursor()
    try:
        serverList = cursor.execute("SELECT NAME FROM SERVER")
    except Exception, e:
        print 2, repr(e)
    cursor.close()

    serverDict = {}
    # Traverse through all servers and find their current load
    # combined with the potential load
    for server in serverList:
        activeServerLoad, timestamp = cursor.execute("SELECT CPU, TIMESTAM FROM SAMPLE WHERE NAME = %s ORDER BY TIMESTAMP DESC", server)

        lastHour = timestamp - datetime.timedelta(hours=1)

        userLoad = cursor.execute("SELECT AVG_CPU FROM PREDICTION WHERE LAST_USED_SERVER = %s AND LAST_LOGIN > %s", server,
totalServerLoad = activeServerLoad + sum(userLoad)

serverDict[server] = totalServerLoad

# Print (return) the server with the lowest load
print min(serverDict, key=serverDict.get)

if __name__ == '__main__':
    userID = sys.argv[0]

    userExists = existingUser(userID)

    # If the user is already in the prediction table we want to check if RAM cache is available
    if userExists:
        # Find the RAM load on the server since the last login
        lastServer, lastLogin = lastUsedServer
        loadSinceLast, cachedRAM = serverRAMload(lastServer, lastLogin)

        # Get the avg RAM usage for the user from the prediction table
        avgRAMusage = getUserAvgRAM(userID)

        # See if cache is still available
        cacheAvailable = calculateCache(loadSinceLast, avgRAMusage, cachedRAM)

        # If there is still cache available, we check if the server has enough CPU to support the user
        if cacheAvailable:
            serverCPU = getServerCPU(lastServer)
            userCPU = getUserCPU(userID)
            totalCPU = serverCPU + userCPU

            # If the server with RAM cache has enough CPU the user will be assigned to that server
            if totalCPU < 100:
                print lastServer
            else:
                # If the server is full on CPU load, the server with the least load is selected
                leastLoadServer()
else:
# If the cache has been replaced since last login we move
the user to the server with the most available load
leastLoadServer()
else:
# If the user has never been in the system, we place the user
on the server with the most available capacity
leastLoadServer()
from __future__ import division
import os, datetime, psutil, time, subprocess
import socket
import MySQLdb, MySQLdb.cursors
import logging
import pprint
import math
from decimal import *
import csv
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
from sets import Set
import numpy as np
from itertools import groupby

# Connect to database
try:
    db = MySQLdb.connect(host="localhost", user="lb", passwd="lb", db="MONITOR", cursorclass=MySQLdb.cursors.SSCursor)
except Exception, e:
    print 1, repr(e)

userID = 'espemobe'

def getAllUsers():
    cursor = db.cursor()
    data = []
    try:
        cursor.execute("SELECT NAME FROM USER")
    except Exception, e:
        print 1, repr(e)


except Exception, e:
    print 3, repr(e)

for row in cursor:
    data.append(row)
cursor.close()
return data

def getUserID(userID):
cursor = db.cursor()
    try:
        UID = cursor.execute("SELECT UID FROM USER WHERE NAME = %s", (userID))
    except Exception, e:
        print 2, repr(e)
    row = cursor.fetchone()
cursor.close()
    return row[0]

def getJobs(UID):
cursor = db.cursor()
data = []
    try:
        cursor.execute("SELECT PID, START_TIME, CMD_NAME FROM JOB WHERE UID = %s", (UID))
    except Exception, e:
        print 3, repr(e)

for row in cursor:
    data.append(row)
cursor.close()
    return data#, cursor.rowcount

# for row in cursor.fetchall():
#  PID = row[0]
#  TIME = row[1]
#  getJobData(UID, PID, TIME)

def getJobData(uid):
cursor = db.cursor()
data = []
try:
    cursor.execute("SELECT CPU, RUN_TIME FROM jSAMPLE WHERE UID = %s", (uid))
except Exception, e:
    print 4, repr(e)

for row in cursor:
    data.append(row)
cursor.close()

return data

# try:
#     for row in cursor.fetchall():
#         print row
#         CPU = row[0]
#         total = total + CPU
#     print total, cursor.rowcount
# except Exception, e:
#     print 5, repr(e)

def getServer(UID):
    try:
        server = db.query("SELECT SERVER FROM USER WHERE UID = %s", UID)
    except Exception, e:
        print 5, repr(e)

    return server

def getServerData(server):
    try:
        db.query("SELECT * FROM sSAMPLE WHERE NAME = %s", server)
        server_data = db.store_result()
    except Exception, e:
        print 5, repr(e)

    return server_data

def topFive(data):
    dict1 = dict()
    returnData = []

    for item in data:
        if item[2] in dict1:
            dict1[item[2]].append(item)
dict1[item[2]] = dict1[item[2]] + 1

else:
    dict1[item[2]] = 1

count = 0
for w in sorted(dict1, key=dict1.get, reverse=True):
    if count <= 5:
        combine = str(w) + " " + str(dict1[w])
        returnData.append(combine)
        count += 1
    return returnData

def averageJobCPU(jobData):
    maxCPU = 0
    mySum = 0
    cpuTotal = []
    endTime = jobData[-1][1]
    startTime = jobData[0][1]
    CPUstart = jobData[0][0]

    dataPoints = len(jobData)
    timeDiff = int((endTime - startTime).total_seconds())

    if dataPoints < timeDiff:
        for row in jobData:
            if int((row[1] - startTime).total_seconds()) > 1:
                diff = int((row[1] - startTime).total_seconds()) - 1
                missingCPU = Decimal((CPUstart+row[0]) / 2 * diff) + row[0]
            else:
                missingCPU = row[0]
            cpuTotal.append(missingCPU)
            startTime = row[1]
            CPUstart = row[0]
    if row[0] > maxCPU:
        maxCPU = round(row[0], 3)
    else:
        for row in jobData:
            missingCPU = row[0]
            cpuTotal.append(missingCPU)
            if row[0] > maxCPU:
maxCPU = round(row[0], 3)

try:
    mySum = sum(cpuTotal)/timeDiff
except Exception, e:
    pass

if mySum == None or mySum == 0:
    return 0, timeDiff, maxCPU
else:
    return mySum, timeDiff, maxCPU

def averageTotalCPU(avgCPU):
    return round(sum(avgCPU)/len(avgCPU), 2)

def averageTotalTime(runTime):
    return sum(runTime)/len(runTime)

def timeConverter(sec):
    m, s = divmod(sec, 60)
    h, m = divmod(m, 60)
    return h, m, s

def generateGraphs(userID, user):
    userName = user[0]
    jobNames = []
    data = getJobs(userID)

    for item in data:
        jobNames.append(item[2])

    uniqueJobs = Set(jobNames)
    # uniqueJobs = [‘buildall’]

    for item in uniqueJobs:
        avgCPU = []
        length = 0
        fig, ax = plt.subplots()
        itemData = []
        for row in data:
            rowData = []
            if row[2] == item:
                jobData = getJobData(userID, row[0], row[1])
                avgCPU.append(jobData[2])
                length += 1
                itemData.append(jobData)

        # Data processing
        # Calculate averages, plot graphs, etc.

        # Additional code for generating graphs and data analysis

    return userName, uniqueJobs, avgCPU, timeConverter(length)
CPU, diff, maxCPU = averageJobCPU(jobData)

avgCPU.append(CPU)
for CPU in jobData:
rowData.append(round(CPU[0], 3))

if len(jobData) > length:
    length = len(jobData)
itemData.append(rowData)

x = [sum(e) / len(e) for e in zip(*itemData)]

y = []
count = 0
while count < len(x):
y.append(count)
count += 1

plt.plot(x)
plt.axis([0, len(x), 0, 100])
plt.xlabel('Time in Seconds')
plt.ylabel('% CPU')
plt.title(userName + '-' + item + ' AvgCPU: ' + str(averageTotalCPU(avgCPU)))
plt.grid(True)

directory = 'trends/' + userName + '/'

if not os.path.exists(directory):
os.makedirs(directory)
plt.savefig(directory + item + '.png')

def keyfunc(timestamp, interval = 60):
    # defined a key function.
    # 1. parse the datetime string to datetime object
    # 2. count the time delta (seconds)
    # 3. divided the time delta with interval, which is (6*60) here
    xt = datetime.datetime(2015, 4, 3)
dt = datetime.strptime(timestamp, '%d/%m/%Y %H:%M:%S')
delta_second = int((dt - xt).total_seconds())
normalize_second = (delta_second / interval) * interval
return xt + timedelta(seconds=normalize_second)
logging.basicConfig(level=logging.DEBUG, filename='/home/dsunde/master/dataPoints.log')

if __name__ == '__main__':

d = {}

userName = 'henribak'

# for user in userName:
#   userID = getUserID(user)
#   generateGraphs(userID, user)

userID = getUserID(userName)

data = getJobData(userID)

# data.sort(key=lambda i: i[1]) # sort with timestamp

results = []

print data[1]

for k, g in groupby(data, key=lambda i: keyfunc(i[1])):
    # k would be time interval "03/04/2013 13:30:00", "03/04/2013 13:35:00" ....
    # g would be the level, timestamp pair belong to the interval
    avg_level = sum([x[0] for x in g]) / len(g)
    results.append((k, avg_level))

print results

# data.sort(key=lambda i: i[1])

# startTime = data[0][1]
# nextTime = startTime + datetime.timedelta(0,60)

# for row in data:
#   if startTime <= row[1] <= nextTime:
#       d[nextTime] += row[0]

# print d

# for row in data:
#   if row[1]

# for row in data:
# startTime = row

# if timestamp > last_time and timestamp < timestamp + 1 min:
# sum(CPU)/len(CPU)