UNIVERSITY OF TARTU
Faculty of Mathematics and Computer Science
Institute of Computer Science
Distributed Systems Group
Information Technology

Martti Vasar

A Framework for Verifying Scalability and Performance of Cloud Based Web Applications

Master thesis (30 ECTS)

Supervisor: Dr. Satish Narayana Srirama

Author: .... "....." May 2012
Supervisor: .... "....." May 2012
Head of the chair: .... "....." 2012

TARTU 2012

Abstract

Network usage and bandwidth speeds have increased massively and vast majority of people are using Internet on daily bases. This has increased CPU utilization on servers meaning that sites with large visits are using hundreds of computers to accommodate increasing traffic rates to the services. Making plans for hardware ordering to upgrade old servers or to add new servers is not a straightforward process and has to be carefully considered. There is a need to predict traffic rate for future usage. Buying too many servers can mean revenue loss and buying too few servers can result in losing clients. To overcome this problem, it is wise to consider moving services into virtual cloud and make server provisioning as an automatic step. One of the popular cloud service providers, Amazon is giving possibility to use large amounts of computing power for running servers in virtual environment with single click. They are providing services to provision as many servers as needed to run, depending how loaded the servers are and whatever we need to do, to add new servers or to remove existing ones. This will eliminate problems associated with ordering new hardware. Adding new servers is an automatic process and will follow the demand, like adding more servers for peak hours and removing unnecessary servers at night or when the traffic is low. Customer pays only for the used resources on the cloud. This thesis focuses on setting up a testbed for the cloud that will run web application, which will be scaled horizontally (by replicating already running servers) and will use the benchmark tool for stressing out the web application, by simulating huge number of concurrent requests and proper load-balancing mechanisms. This study gives us a proper picture how servers in the cloud are scaled and whole process remains transparent for the end user, as it sees the web application as one server. In conclusion, the framework is helpful in analyzing the performance of cloud based applications, in several of our research activities.

Acknowledgements

This thesis would not be possible without the encouragement and supervision of Satish Narayana Srirama. I am grateful for his support. I would also like to thank Marlon Dumas giving opportunity to work with Distribute Systems Group, giving the resource and possibilities to work with private and public cloud systems. Satish Narayana Srirama helped a lot for getting started with cloud computing and running services in the cloud. It has been great experience to work together. My special thanks goes to Michele Mazzucco for giving ideas about configuring servers in the cloud and assistance for setting up different provisioning algorithms.

Contents

1 Introduction **1**
1.1 Contributions . . . 3
1.2 Outline . . . 4

2 The State of Art **5**
2.1 Cloudstone . . . 5
2.2 To move or not to move . . . 6
2.3 Wikibench . . . 8
2.4 Web server farm in the cloud . . . 9
2.5 Services that support provisioning and auto scaling properties 9

3 Framework for Verifying Scalability and Performance **11**
3.1 Problem statement . . . 13
3.2 Cloud Computing . . . 13
3.2.1 Cloud computing as a Service . . . 14
3.3 SciCloud project . . . 16
3.4 Eucalyptus . . . 17
3.4.1 Eucalyptus architecture . . . 17
3.5 Amazon Cloud . . . 21
3.5.1 Amazon EC2 and Auto Scaling . . . 21
3.5.2 Amazon pricing . . . 22
3.6 MediaWiki architecture . . . 22
3.6.1 Memcached reducing load on database server . . . 24
3.6.2 LAMP stack . . . 26
3.6.3 Operating system configuration . . . 29
3.7 Study of load balancers . . . 30
3.7.1 Pen . . . 30
3.7.2 Nginx as reverse-proxy server . . . 31
3.7.3 Comparison between nginx and pen . . . 33
3.8 Configuration of the infrastructure . . . 34

3.8.1 Endpoints for SciCloud and Amazon 35
3.8.2 Setting up services and MediaWiki installation 35
3.8.3 Wikipedia article dumps 36
3.9 Framework scaling instances in the cloud 37
3.9.1 Calculating arrival rate 39
3.9.2 Removing servers from the cloud 39
3.10 Framework layout 40
3.10.1 CloudController 41
3.10.2 ServerStatus 42
3.10.3 BenchMark 44
3.10.4 Additional HTTP server 45
3.10.5 Distinguishing different servers by their role 45

4 Measuring Performance of The Cloud 48
4.1 Using performance measuring tools under Ubuntu 48
4.1.1 Apache tool AB 48
4.1.2 Ubuntu sysstat package 49
4.2 Using third party tools 50
4.2.1 RRD database 52
4.2.2 Collectd 52
4.2.3 Cacti 53
4.3 Framework Performance Measuring Tools 56

5 Preliminary Experiments and Evaluation of The Framework 58
5.1 Importance of experiments and stress testing 59
5.2 Identifying most suitable configuration 59
5.2.1 Experiments with `m1.small`, `m1.large` and `c1.medium` instances 61
5.2.2 Experiments configuration 61
5.2.3 Measuring service time 62
5.2.4 Measuring maximum throughput 63
5.2.5 Summary of different instance types performance . . . 64
5.2.6 CPU cycle waiting behind Xen hypervisor 67
5.2.7 Measuring CPU utilization with ramp up 68
5.2.8 Determining concurrent connections 69
5.3 Experiment to establish limits of the service for current configuration 74
5.3.1 Deadlock triggered by MySQL 75
5.3.2 Fixing deadlock 75
5.3.3 Network utilization 76
5.3.4 CPU utilization 76

5.3.5 Replicating MySQL and memcached instance 77
5.3.6 Finding out network bandwidth upper limit 78
5.4 Validating difference between different availability zones in the same region 79
5.5 Conclusion of the configuration optimization 82
5.5.1 MediaWiki configuration and tuning 82
5.5.2 PHP caching 84
5.5.3 MySQL tuning 84

6 Policies and Validation of Scalability 86
6.1 Why is elasticity of the cloud important 86
6.2 Basic configuration for running the experiments 87
6.3 Traces used to test policies 90
6.4 Utilization policies 90
6.4.1 Auto Scale 91
6.4.2 Auto Scale results 93
6.5 Revenue based policies 95
6.5.1 Algorithm for provisioning the servers 95
6.5.2 Arrival rate for the next hour 97
6.5.3 Optimal heuristics 98
6.5.4 Optimal heuristics results 101
6.6 Summary and results 103

7 Conclusions 105

8 Future Work 107
8.1 Benchmark Tool Updates 107
8.2 Server Comparison 108

Raamistik pilvel põhinevate veebirakenduste skaleeruvuse ja jõudluse kontrollimiseks 109

Appendix 117

List of Figures

3.1 A cluster-based web server configuration. 12
3.2 Layout of the Eucalyptus architecture, showing how the services are connected with each other. Router enables access from the outside world to connect with the cloud and running instances [54]. 17
3.3 Flowchart showing how request and responses between different servers are being transferred when the requested page is already cached. Note that MediaWiki still makes request to database to see, whatever the page exists and is not changed. . 23
3.4 Flowchart showing how request and responses between different servers are being transferred when the requested page is not cached. There are several queries and calls to memcached and MySQL server to compile the page. Once ready, the parsed page is stored in the memcached. 24
3.5 Cumulative distribution of response times for `c1.medium` instance with different caching policies. 26
3.6 Displaying experiment configuration in the cloud environment and how the requests are going through. All the servers are located in the cloud, for reducing network latency, they are located in the same availability zone. 40
3.7 Showing interactions between cloud interface and virtual machine running on the cloud through CloudController application. 41
3.8 Common life cycle of instance running in the cloud and reconfigured while the service change state [53]. 43

4.1 Standard output of `iosatat` command. 50
4.2 Standard output of `sar` using parameter to show memory usage. 50
4.3 Standard output of `free` to get information about physical memory usage. 50
4.4 Standard output of `sar` using parameter to show CPU usage. 51

4.5 Standard output of `ifconfig` to gather network interface traffic bandwidth. 51
4.6 Standard output of `uptime` to see information how long the computer has been running and also get Linux load for 1, 5 and 15 minutes. 51
4.7 Standard output of `ps` with various options to gather how many processes are working. Because command itself counts as a single process, these results should be subtracted by 1. . . 51
4.8 Showing Linux load gathered with Cacti and drawn by Cacti using RRDTool. 54
4.9 Fraction of the experiment running 10 servers, showing average CPU usage for Apache server and benchmark response time. For each 5 minutes, there is a spike, because cacti starts to measure metrics of the server. 55
4.10 Showing spikes caused by Cacti, when collecting different performance metrics from the server. 55
4.11 Modifying ServerStatus code and using JavaScript, it is possible to draw graph of the CPU usage that can be viewed when opening browser on port 5555. 57

5.1 Instance `m1.small`, `c1.medium` and `m1.large` service time distribution. Left side shows service time cumulative distribution. 62
5.2 Instance `m1.small` response time and CPU usage for ramp up test. In the case of this instance type server was able for 6 requests at maximum and then the service became unresponsive. Only 40% of the CPU cycle was enabled for user. Most of the CPU cycle, over 55%, was going to steal, meaning that virtualization was dividing available CPU usage to other instances on the same physical machine. 65
5.3 Instance `c1.medium` response time and CPU usage for ramp up test. Server was able for 28 requests at maximum and at the peak the average response time increased to 3 seconds and server started to become oversaturated. 66
5.4 Instance `m1.large` response time and CPU usage for ramp up test. Server was able for 18 requests at maximum and at the peak the average response time increased to 5 seconds and server was heavily overloaded. 66
5.5 Ramp up experiment using `c1.medium` instances with 5 Apache servers showing CPU utilization for different servers, while generating load between 1 request per second to 150 requests per second. 69

5.6 Ramp up experiment showing difference between two Fair configuration (using 6 or 10 concurrent connections per Apache server) and without Fair using `c1.medium` instances with 5 Apache servers. Left side graph shows CPU usage for Apache servers and right side shows 1 minute load. 70
5.7 Ramp up experiment showing difference between two Fair configurations (using 6 or 10 concurrent connections per Apache server) and without Fair using `c1.medium` instances with 5 Apache servers. Left side graph shows average response times for requests and right side shows how many jobs are being processed by Apache servers. 71
5.8 Ramp up experiment showing difference between two Fair configurations (using 6 or 10 concurrent connections per Apache server) and without Fair using `c1.medium` instances with 5 Apache servers. This shows amount of successful requests going through the system. 72
5.9 Concurrent connections experiment using `c1.medium` with 10 Apache servers showing CPU utilization and servers throughput. Service rate shows, how many requests one Apache server is capable of serving. 73
5.10 Concurrent connections experiment using `c1.medium` with 10 Apache servers showing Apache servers 1 minute load and response time. 73
5.11 Ramp up experiment with 20 `c1.medium` Apache servers and generating load up to 550 requests per second, showing CPU usage for different servers. Nginx used Fair module to distribute the traffic (used the least round-robin), therefore there were large CPU usage difference between various Apache servers at the beginning of the experiment. 77
5.12 How to implement two nginx load balancers to increase redundancy and fault tolerance, and in the same time increasing maximum throughput of the service. For MySQL, there should be master-slave configuration, where updates and saves are done to the master and slave is used for reading operations. 80
5.13 Comparisons when running MediaWiki infrastructure with XCache enabled and without XCache on `c1`.medium Amazon instance. These are response times with 30 second resolution. Both configurations used memcached with filled cached. 85

6.1 Left graph shows over provisioning, where the policy is running more servers than needed. Right graph shows situations of under provisioning, running fewer servers than needed. Graph at the top shows how dynamically allocated servers can handle traffic fluctuating in time. 88
6.2 Two week ClarkNet traffic trace with 1 minute intervals showing arrivals per second. The red line defines section of the trace used to generate the load for the service. This section has been scaled up to 600 requests per second in the experiment. 90
6.3 Showing Amazon Auto Scale experiment configuration in the cloud. Servers were kept in the same availability zone to reduce the latency of requests. Autoscaling group has been created, where Auto Scale algorithm can replicate servers when the demand increases and terminate servers from the group, while the demand decreases. 92
6.4 Running servers compared to arrival rate λ, CPU usage and response times over 24 hours using Amazon Auto Scale policy. 94
6.5 Removing servers should be done before full hour, because otherwise there is need to pay for the extra instance hour. . . 96
6.6 Adding servers is done at the full hour mark to make sure that while the servers are added and needed to be removed, there is no need to pay for the extra instance hour. 96
6.7 Revenue function values, when trying to find the optimal maximum for 300 requests per second and 600 requests per second using erlang units to calculate the throughput. 101
6.8 Running servers compared to arrival rate λ, CPU usage and response times over 24 hours using Optimal policy. 102

Chapter 1

Introduction

Applications and services are moved to the cloud to provide elasticity and custom configuration of the servers in a runtime. Cloud computing gives perfect platform to deploy scalable applications, that can be provisioned by the incoming traffic rate or server utilization. This helps customers and service owners to save money, but in the same time have an immense capacity of computing power in his hands, that can be unleashed with clicks, commands or make it as an automatic process.

Applications running in the cloud are flexible, scalable and fault tolerant, if they are properly configured. Framework given by the cloud computing service providers allows us to replicate the same servers already running in the cloud. We can replace faulty servers by requesting new servers from the cloud. There are two common ways to scale service in the cloud: horizontally or vertically. Vertical scaling means that the servers are replaced with more powerful machines while the demand increases (increasing CPU cores, memory, network bandwidth or hard drive speed) or replaced with slower machines, if the demand decreases. Horizontal scaling means that new servers are added along to the already running servers while the demand increases or already running servers are terminated, if the demand decreases. Either way, there is a need for the reasonable decision why to use one or another scaling methods, also have to consider how easy or hard it is to configure such changes. When requesting servers from the cloud pool, it takes some time to become operational. With vertical scaling we have to ensure, that closing one server and activating another server is smooth process, where no delays or rejections occur. It is easier to scale horizontally as there are fewer steps needed to be done when configuring the servers. For example, replacing MySQL with faster machine should be transparent, none of the incoming requests should fail. Using vertical scaling, it means in order to replace existing MySQL server, we have to run a new server, replicate its state and data into new machine, terminate the old server and made configuration in other servers, which use MySQL database and change the IP address with new one.

Why is cloud computing becoming essential for running large applications? One key aspect is that virtual machines are easier to install than their counterparts - physical servers. With physical servers, we need to make an extra cost and decisions how these servers should be managed. Not to mention extra cost on hardware, making disaster plan and having a solid backup plan. Using virtual machines, everything is done by the data center workers and customer has to only use command line or graphical interface to manage servers in the cloud, without having to know network topology and other problems related to installing physical servers. There is no need for capacity planning as you can immediately request from the cloud new servers without need to buy or order them thus waiting for the arrival. The speed and capability to use large amount of CPU power is making cloud computing prestigious and many web pages have already deployed in the cloud.

With virtual machines, we have two states, whatever it is running or it does not. If something is wrong or the virtual machine has crashed, we can easily replace it with a new machine just in a couple of minutes. Failure with a physical machine means, that you have a box wasting the space and not working. You need to find the cause of the problem and sometimes it can be tricky and time consuming operations. There might be faulty memory module or some of the components does not work any more as the hardware constantly wears off and ageing. This means investments to new hardware and making new configuration plans. If the hardware is not on-site, ordering and waiting for the piece to arrive can take days. With virtual cloud capabilities, this is taken care of other people and the system itself is seen from the web as high fault tolerant system, where single failure cannot cause service becoming unavailable.

Virtual computing allows us to produce virtual images with user's applications on it, that are easy to be replicated and scale them as the demand over the time for the service changes. This differs for physical machines, where every machine operating system (OS) has to be installed separately or use tools to duplicate first machine disk image to the other machines. This procedure involves physical contact with the machine, also users has to have knowledge about the topology of their network and configuration of the system. Virtualization adds another abstraction layer, that one virtual system is able to run on any machine, despite of what hardware has been installed. There is no need to worry about driver dependencies and updates to get everything working.

1.1 Contributions

This thesis aims to uses both private and public clouds to configure and set up infrastructure for running web application and monitoring its performance. For private clouds, SciCloud [1, 2] was used. The SciCloud project is used among students and university workers to have start up projects to run different applications and tools on virtual environment with the capabilities to scale the applications. These applications can be later transferred to Amazon for doing larger scale experiments and check, how the system will work in a public cloud environment [1, 2]. Thesis mainly explores the possibilities to configure and run web applications on the cloud. It studies different ways to automatically configure servers on the cloud based of the current traffic flow, this means how and when to add new servers or remove them. The particular software deployed in the cloud was *MediaWiki* application, which is used to run *Wikipedia* project and therefore has good scaling properties. According to 2009 year summary from WikiPedia, their site gets at peak 75 000 request per second on their different sub-projects and sub-wiki pages. The system is running on over 500 severs and is managed by 16 workers [28]. These numbers are out of the scope of this thesis and are hard to simulate, as these experiments would cost immense amount of money, setting up such system for short time periods is not a feasible and there might occur other problems related to network bandwidth, meaning DNS based traffic distribution should be used.

Framework provided by the thesis looks different possibilities for automatically provisioning servers on the cloud depending on the load. One option is to use service time as an indicator, how many servers are going to be needed, another is to use average CPU usage to determine how loaded the servers are and whatever there is need to add or remove servers. This can be later on tested with experiments over one day to see, how server provisioning is working and validating the configuration. The framework will measure performance of the servers running in the cloud with 30 second interval to see possible bottlenecks or errors that can occur while running the experiments. The outcome of this thesis will be the application set, capable for collecting arrival rates for the servers, measuring their performance, dynamically changing load balancer configuration, adding or removing servers depending on the demand and easily deploying infrastructure to the cloud with various predefined scripts. It also includes benchmark tool, that can be used to stress out the load balancer, while the requests are divided between back-end servers, it will stress out overall system including cache and database server.

Various empirical experiments with measured performance metrics help to optimize services in the cloud. Most of the applications' and servers' default configurations are meant for wide audience and does not fit well with high performance service. Experiments help to identify possible bottlenecks in the system to see, which service configuration should be changed to improve the overall throughput. Two well known experiments have been conducted to measure service time and system maximum throughput. First, the measurements can be gathered when doing experiment with low load, generating requests in a way that only one job is in the system. Latter, the measurements can be gathered when doing ramp up experiment meaning that you have a test when, for each time unit there is a increase of load (arrival rate) to the servers and load is increased, until the maximum throughput is found.

1.2 Outline

This work is suitable for people, who are aware of computer systems, Linux commands and have some knowledge in cloud computing and know about script languages. The work is divided into 8 chapters. The 2st chapter focuses on the state of art, summarizing related topics, showing what the others have already done and what issues they have seen. The 3nd chapter focuses on deploying the infrastructure, what tools can be used to achieve it. The 4rd chapter gives set of tools or ways how to measure performance of the servers in the cloud system and how the measuring was done. The 5th chapter focuses on conducting of preliminary experiments to validate the infrastructure testbed and to fetch important performance parameters for optimizing the services in the cloud. It will provide a list of things we have been optimizing to improve the speed of service. The 6th chapter shows some results for running one day experiments with fixed traffic. We are focusing on Amazon Auto Scale and optimal policy to see the difference in these algorithms, also this helps us to validate the cloud configuration. The 7th chapter gives conclusion of the work and what has been accomplished. The 8th chapter will give overview, what has been left out of the framework as there has been no time left to improve these characteristics and should be consider for future work.

Chapter 2

The State of Art

There are several studies to provision and scale virtual machines on the cloud horizontally and/or vertically. There are even several frameworks to make deployment of infrastructure to the cloud easier. The main idea for the frameworks has been to reduce the time it takes to configure cloud instances, that will reduce experiment length and cost. Cloudstone [44] is a framework, that is compatible with Web 2.0 applications and supports application deployment in the cloud helping to reduce time it takes to configure the servers and to run the experiments. Wikibench [37] is being taken as a base for building such framework. There has been already conducted large scale experiments with MediaWiki installation, but this study did not use the cloud and the dynamical server allocation was not their primary idea. MediaWiki application and other related services have been installed on the physical servers. There have been also studies, that discourages moving applications to the cloud giving some key notes why this is not feasible, because in the most of cases it does not help to reduce cost for running of servers [47].

2.1 Cloudstone

Cloudstone is a benchmarking framework built for the clouds [44]. It tries to be compatible with Web 2.0 applications and uses all the benefits that Web 2.0 supports. Compared with Web 1.0 applications, where content was mostly static and generated for the user, it deals with nowadays technologies, where users can share content through blogs, photostreams and tagging. It uses social application Olio for benchmarking purpose, that meets the Web 2.0 application criteria. Web 2.0 applications have higher write/read ratio as content can be added by the web service users, that will put higher workload to the disk and database engine.

Cloudstone supports automatic deployment of system to the cloud. It gives different ways to set up the cloud and is flexible, giving possibility to configure MySQL database in a master-slave configuration. Scripts provided by Cloudstone work well with Amazon EC2. It is possible to deploy, terminate, restart and configure databases, web servers, application servers and load balancers in a deployment environment. It uses Faban for generating the load. Faban supports collection of the performance metrics during the experiments using Linux tools *iostat, mpstat, vmstat, netstat*, etc. It supports configuring memcached into the existing web application, that will allow to store already rendered pages, database query results and user session data in to the memory for faster processing. Initially Cloudstone was using Apache with *mod_proxy* to distribute the load, but later on switched to *nginx* as *mod_proxy* was becoming bottleneck under high loads, supporting only 135 concurrent users. Nginx gives more flexibility in configuration and was capable for much higher throughput, making it ideal tool for the Cloudstone.

This paper focuses on two questions, that have been achieved through experiments: (i) *how many (concurrent) users can be served for a fixed dollar cost per month on Amazon EC2?* and (ii) *how many can be supported without replicating the MySQL database.* Their approach was different compared to other papers and articles as they were using much bigger Amazon EC2 instances. Using `m1.xlarge` (8 EC2 units, 4 virtual cores x 2 EC2 units each) and `c1.xlarge` (20 EC2 units, 8 virtual cores x 2.5 EC2 units each), both cost $0.80 per instance-hour. Some important and interesting findings from the paper are:

1. Discovered, that `c1.xlarge` is capable of serving more MySQL queries per second, than `m1.xlarge`, making the MySQL database CPU-bound.
2. Turning of the logging, throughput increased at least 20% for some cases.
3. Capacity limit was hit by gigabit Ethernet with 1,000 concurrent users.

2.2 To move or not to move

Cloud computing is relatively expensive resource to be used. There has to be well thought-out plan for moving existing infrastructure to the cloud. Dedicated servers with ISP in a longer time span would be more cost efficient than running service on the cloud.

There are different studies that try to compare two different options to run services in terms of cost and usability [44, 47]. Cloud computing gives good platform for developing scalable applications and is cheaper to start than buying separate hardware. Setting up hardware is more time consuming and person must be well aware of the network topology and how to duplicate and replicate service to different servers, compared to the cloud computing, where customer has to only configure one virtual machine image and can replicate the same configuration to different servers. It is possible to build image from your own local image, installing operating system and necessary software and uploading it to the cloud or there is a possibility to use already existing images provided by Amazon or other vendors and install only necessary software. There are also bundled images with different software already installed, you can get machine, where MySQL database and Apache server with PHP are running, but for security reasons it would be better to build image from a scratch as you might not be aware, what has been installed there by others.

Cloud computing gives almost infinitive resource to your hands. There have been many start up projects that have been launched in the Amazon EC2 cloud and have gained popularity through Facebook or other social media. Good advertising will led to massive usage of service and growth of demand. This means that there should be more resources to handle the load. This can be easily achieved on the cloud environment as you can start up instantly new servers. Using your own server park with small amount of hardware facilitating the service, you are not able to respond to the extensive user base growth as buying or ordering new servers are time consuming and they have to be configured with proper software. If those start up projects have not been in the cloud and running service in small server park or even on a single server, they have not had such a huge success, because the service would have been overloaded, and users could not access it and, therefore, losing the interest [50].

Cloud computing gives great platform for deploying scalable applications, but it is not suitable for all cases. It depends of the purpose of service and how widely it should be available for the audience, using the application. For small projects and local web pages, it would not be good solution, as holding up the infrastructure will cost more money, than using virtual HTTP servers. Cloud computing is a perfect platform for developing new infrastructure and services as renting the servers for smaller time periods is not that expensive than using dedicated or virtual servers. Definite plus for cloud computing is, that it gives possibility to rapidly scale your application to see the boundaries of the system. It would be hard to find service providers willing to give large amount of servers for small time period with low cost.

There is also option for using hybrid configuration of dedicated servers and cloud servers to run the application. For longer time span it is cheaper to have your own servers, but sometimes you might need extra resources for short time and, therefore, it is a good idea to request extra servers from the cloud. Good example would be the tax collection web site, where most of the time traffic is low. When the deadline arrives for tax collecting, you might need extra servers to accommodate incoming requests and these can be easily acquired from the cloud.

2.3 Wikibench

The thesis takes novel approach for benchmarking web application using realistic traffic with the realistic infrastructure using real data. Wikibench is benchmarking tool specially built for MediaWiki application, it supports two types of queries while benchmarking the system. It supports reading operations using `HTTP GET` queries and occasionally makes database updates through web interface using `HTTP POST` queries. This will simulate users, that are reading different articles and editors, who are updating, editing or creating new articles. Many of the benchmark tools are using only `GET` request and do not really deal with updating database. Using `POST` queries will really stress out the database as updating entries in the database uses more CPU and hard disk than generating simple read requests. The performance of a database depends on the database structure, how complex, how many keys and entries it consists.

Wikibench consists of a set of tools meant for benchmarking MediaWiki application with the real traffic. They have developed load generator, that can be started in clusters and uses Wikipedia traces [1] for requesting pages. These traces will show when the request was made and the tools tries to generate the request with same frequency. Using real traces, means that the database has to be filled with real data, because otherwise we might get content not found errors. Thesis uses Wikipedia database dumps [2] to have realistic environment for stressing out the system.

This work will compare other benchmarking tools, that can be used for stressing out the system. Thesis brings out some major missing features and explains why the new developed benchmarking tool is necessary. The major problem is that the most of traffic generators are making artificial traffic that cannot be compared with the real life situations. The problem is that generated traffic is uniformly distributed and does not fluctuate so much in

[1]http://www.wikibench.eu/?page_id=60
[2]http://dumps.wikimedia.org/backup-index.html

time and have fixed amount of simultaneous connections. Wikibench includes Wikijector tools used to generate the load. It has programmed to have two roles, it can act as a controller or a worker. Controller's task is to hold eye on the workers, collect response times from workers and coordinate workers jobs. Workers are first sitting in an idle state and waiting start command from the controller. Controller will also define, where to get the traces and how long to run the experiment.

Because of using traces, it is hard to control how many requests the load generator will make. Wikipedia is mostly stable system and most of the requests have been already cached. Every time a new experiment is started, the cache is empty. When using traces and Wikijector, it is hard to fill the cache as the servers are able to serve 3 to 4 times fewer requests compared when the information is coming from the cache. It is mentioned in the thesis, that at the beginning of experiments the response to the request will take more time as the cache will not be filled and in the later stage, the response time will be reduced drastically. This thesis uses mixed requests meaning that the traces were containing page requests to static files (images, style sheet files, logos, javascript) and dynamic pages (server needs to generate page with PHP). Difference in response time between those two requests is from 2 milliseconds to one second while the cache is not filled.

2.4 Web server farm in the cloud

Good overview about Amazon EC2 cloud performance gives paper [50] written by Huan Liu and Sewook Wee. They give good performance comparison between different instances provided by the Amazon. It also gives comparison with other cloud providers to give vague idea, how well or bad one service provider will do in this field. Most of instances are able to transmit 800 Mbps traffic (input and output combined) according to their tests. This will give rough idea, how network bound we can be when carrying out experiments.

2.5 Services that support provisioning and auto scaling properties

There are many service providers giving possibility to scale cloud on demand and is made as an automatic process. Service providers are asking additional fee for auto scaling and provisioning services and generally are using Amazon EC2 cloud to deploy the infrastructure. Scalr [51] supports multiple cloud systems, has different payment options and is open source, meaning that any one can download the code and start using Scalr as free (if there is no need for on-line support). Scalr is a hosted service and can be accessed from the homepage. If you have downloaded the source, you can set it up to run on your server.

Scalr helps to manage public and private cloud infrastructure like Amazon EC2, Rackspace, Cloudstack, Nimbula and Eucalyptus. Scalr has web interface for configuring and setting up the cloud. User has to add his credientals to access the cloud. It is possible to generate different roles (web server, database, load balancer) and mark which instances should be used. For example, you can use larger instance for database and smaller instances for web servers. Load is balanced between web servers using nginx.

The next tool for managing cloud is RightScale [52]. It is not open source, meaning that you have to pay for the service. RightScale supports also multiple clouds. Compared to Scalr, RightScale supports live monitoring of the instances and performance metrics are gathered with collectd. User of this service must be aware, that data collection uses traffic outside of the cloud, meaning you have to pay extra money for the traffic used to collect the performance metrics.

Both managing tools are largely customizable and can use scripts to modify, install new services and change configuration files in the instance on the boot up. They help to reduce the complexity for starting scalable application on the cloud. They are supporting different scaling configuration, making possible to scale web servers and databases independently. Both of the services are supporting back up system. User can activate periodical backups to S3 or other storage devices. This will be useful for making database dumps. Both of the services can save time for system administrator, as most of the configuration can be done through graphical interface. Misbehaving or unreachable servers are terminated and replaced by new servers making the service transparent for the end user and minimizing the downtime. There is also possibility to configure database in master slave configuration to reduce the load on the main database unit and supporting more concurrent users.

Chapter 3

Framework for Verifying Scalability and Performance

This chapter will give overview of various tools and technologies used to set up framework for conducting experiments on the cloud using MediaWiki application. MediaWiki application is used as it has proven already good scalability and is capable of handling large amount of requests. Simplified MediaWiki service installation was used as file cache and other services were adding complexity to already large system. With cache load-balancer like Varnish or Squid (used by Wikipedia) support much larger volume of traffic, meaning network bandwidth limitations might arise with single load balancer.

A cluster-based web server (figure 3.1) consist of a front-end node (usually load balancer) and various number of back-end servers. The front-end (in larger configurations there might be more than one load balancer) will distribute user requests between back-end servers. Front-end server has the biggest impact and should have fast computer with large bandwidth capacity. Back-end servers can be commodity servers as the failure of these servers does not affect overall performance.

Current framework configuration contains one database server (MySQL), one cache server (memcached), several load generators (depending how many web servers are running and what load they need to generate), one load balancer (nginx, handling and directing request to the back end servers) and several web application servers (Apache2 with PHP5). As mentioned early, some of the components were left out like file caching and proxy systems (Varnish or Squid) to reduce complexity of setting up the infrastructure and reducing servers needed to run the experiments. It will also make replication of the experiments harder, as for each experiment, there must be knowledge, how the requests have been cached. Despite of the fact, that framework

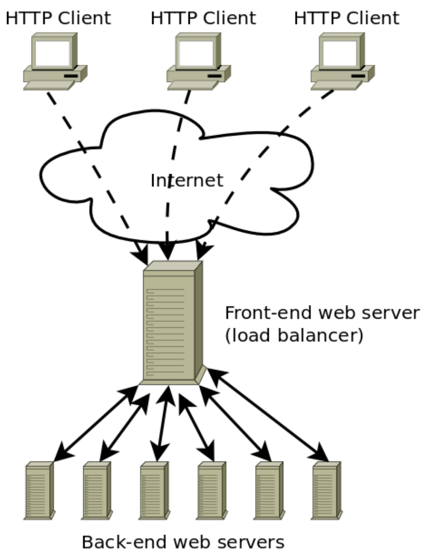


Figure 3.1: A cluster-based web server configuration.

could use Wikijector [37] for generating load, the framework uses the simplified tool for benchmarking purpose that makes only view requests (loading only dynamical PHP page requests), that is capable of reading trace curve and making requests based on the URL list provided to the program. This helped to use smaller fraction of the Wikipedia dumps, making development, framework deployment and conducting the experiments faster. This thesis is interested in scaling only back-end Apache servers and does not use cloud scaling properties to add or remove database servers. This configuration adds more complexity, as the database has to be configured in master and slave mode, where master will deal with insertion and updates requests, and slave will deal with read requests. Reason to dump Wikijector was, that it was resulting slower response times from requests as the code for making requests and reading responses to see the status of the requests (was it failure or success) and length of the response. Therefore, to process this information, it was taking longer time, resulting in larger response times measured by the program.

Step by step instructions how to configure, set up virtual machine instance on the cloud and run experiments are available on the DVD. This thesis may contain some of commands that are used to run various tools, but to get full picture and reassemble experiments done here, reader has to go through instructions, that are stored on the DVD.

3.1 Problem statement

Conducting large scale experiments is a time consuming task. Each experiment in the cloud needs to be done carefully, keeping track of various parameters and remembering which tasks have been done. Because of many factors, starting such experiments are not error-free and mistakes might happen. It would be bad to discover the mistake at the end of the experiment as this will cost extra money and takes more time to complete the task.

Current framework is finding solution how to deploy scientific infrastructure in the cloud, that would be easy to use and can replicate experiments. This thesis tries to find solution, how to deploy services in the cloud and hold configuration in one place. This solution should make it easier to run large scale experiments in the cloud; this involves starting necessary services in different servers, depending on the role of the server and monitoring their performance. This framework will help to reduce time it takes to run the experiments and in the same time collect performance metrics from other servers. At the end of the each experiment, results are gathered. These results can be downloaded into the local computer to analyse collected data.

3.2 Cloud Computing

What is cloud computing? There is not a single definition for this term as there is already saturated list of different names given by leading experts in this field [13]. I. Foster, Y. Zhao, I. Raicu and S. Lu propose definition as following [14]:

> *A large-scale distributed computing paradigm that is driven by economies of scale, in which a pool of abstracted, virtualized, dynamically-scalable, managed computing power, storage, platforms, and services are delivered on demand to external customers over the Internet.*

The article claimed, that there is a rapid increase of interest in cloud computing, especially to own one, this has been affected by the following factors: i) rapid decrease in hardware cost, but meanwhile increase in computing power and storage capacity; ii) the exponentially growing data size in scientific instrumentation and simulation; and iii) simulation and software development in public clouds costs more money then carrying out in private cloud [14]. Cloud computing is utility computing model, where customers, using the service, are paying based on the usage and pricing given by the service provider. Virtual machines are running on physical machines, that consume energy and other resources. Customer needs to pay fees to support the cost it takes to run the service, to the employees taking care of machines and replace broken down hardware. Providers can charge user by the hours, they have been using virtual machines, by the traffic it has passed and by the storage cost.

There are three groups of services provided by different cloud service providers: i) IaaS - Infrastructure as a Service; ii) Saas - Software as a Service; and iii) PaaS - Platform as a Service. Amazon EC2 provides the IaaS service and this thesis focus on building framework for Infrastructure as a Service. It gives cloud computing customer free hands to have their own operating system with needed software. This flexibility gives free hands on modifying the virtual machine and giving infinite possibilities to configure the machine. One of most important factors in cloud computing is that it can scale up and down dynamically depending on application resource needs.

3.2.1 Cloud computing as a Service

There are several service providers giving possibility to use cloud computing to solve computationally intensive tasks and giving almost infinitive amount of resources to use. There are several abstraction layers of services provided by these providers: Software as a Service (SaaS), Platform as a Service (PaaS) and Infrastructure as a Service (IaaS).

Software as a Service

Software as a service is a software delivery model in which software and its associated data are hosted centrally, typically in the cloud and are accessed by users using a thin client. SaaS has become a common delivery model for most business applications. These applications mostly can be used by paying small fee. In 2010, SaaS sales reached to 10 billion dollars [41].

SaaS software can be used by large amount of customers with little resources making it possible to run it at a low price. These services are generally priced by a subscription fees, most commonly using a monthly fee or an annual fee. Because of large user base, some of the providers can offer applications with freemium model. This means, that some of the services are provided without a charge, but it has limited functionality or scope, and fees are charged for extended functionality or larger scope. Some other SaaS providers like Google can provide applications as a free for users, because the revenue is being derived from alternate sources such as advertising. Advertising is one common way to provide underlying software as a free for the end users. Typical SaaS applications provided by Google are Gmail (an e-mail application) and Picasa (a picture application).

Platform as a Service

This category of cloud computing provides a computing platform and a solution stack as a service. In the classic layered model of cloud computing, the PaaS layer lies between the SaaS and the IaaS layer [42]. It offers facilitation and environment to deploy applications without the cost and complexity of buying and managing the underlying hardware. There is a great variety of combinations of services to support the application development life-cycle. Some of the providers give services, where user can select any programming language, any database and any operating system giving free hands on development.

Popular platforms of PaaS are Google App Engine that allows developing and hosting web applications in Google-managed datacenters. These applications are easily scalable. Google App Engine is free up to a certain level of consumed resources. Additional fees are acquired for extra storage, bandwidth or instance hours required by the application.

Infrastructure as a Service

IaaS model cloud provider gives users virtual machines, that can use any operating system they like and install infrastructure from scratch with necessary software. Popular IaaS provider is Amazon EC2. Our experiments, infrastructure building and validation is done in IaaS cloud computing model. IaaS bills users by the instance hours used for each virtual machine, bandwidth usage, storage usage and other services, when they are activated. There are plenty of other services that can be accompanied with existing infrastructure. User can use Elastic Load Balancing service with Auto Scaling, all the necessary services are there to support scalability of application on the cloud. Some of the services are provided by free, e.g. gathering CPU, memory and network usage is provided without no additional charge, but these values are collected with 5 minute intervals. If customers wants to measure the performance frequently or use custom metrics, he has to pay for the extra service. To advertise the service and attract new customers, Amazon EC2 gives for new users for one year each month limited amount of free instance hours.

3.3 SciCloud project

The main goal of the scientific computing cloud (SciCloud) project [1, 2] is to study the scope of establishing private clouds at universities. With these clouds, students and researchers can efficiently use the already existing resources of university computer networks, in solving computationally intensive scientific, mathematical, and academic problems [2]. Previously such problems were targeted by batch-oriented models of the GRID computing domain. The current project gives more freedom and interactive approach for computing intensive analysis and problems, giving possibilities to use large class of applications and work flow. This project also gives better collaboration among different study and interest groups of universities and possibility to test pilot and innovative projects.

The project mainly focus on the development of a framework, establishing models and methods; auto scaling, data and state management, and interoperability of the private clouds. When such clouds have been developed, it is possible to lease them to external use for governmental institutes or firms to invest in divers studies like seismic analysis, drug discovery and large scale data analyses for commercial firms to improve their productivity and revenue.

SciCloud is built and deployed on already existing cluster, using Eucalyptus software. Eucalyptus is one of the most widely deployed worlds leading open-source cloud computing platform for on-premise (private) Infrastructure as a Service (IaaS) clouds [3]. IaaS systems give users the ability to run and control entire virtual machine instances and replicate same configuration through out the cloud on different physical machines. SciCloud has custom built images from Fedora, Ubuntu and other linux distributions ready to use by others. These can be used as a base images and everyone with access to the cloud could easily make their own private image with the software needed to run the experiments. While custom images are built, it is simple to replicate them and conduct experiments.

Eucalyptus is compatible with Amazon EC2 [19] cloud and gives opportunity to develop and make preliminary tests on the private cloud to reduce the cost. Private cloud is a great help for the researchers and academic communities as the initial expenses of the experiments can be reduced significantly. Private cloud experiments and infrastructure deployment can be later on transfer to Amazon EC2 cloud to check, if the application and work flow will work with other cloud environments. SciCloud resources are limited by the means of physical hosts and available cores, it does not allow to request immense amount of virtual machine, but this is possible with Amazon EC2.

3.4 Eucalyptus

Eucalyptus is open-source cloud system able to run on different Linux distributions and works with several virtualization technologies. Eucalyptus is framework that uses computational and storage infrastructure available mostly for academic users and can be used for innovative and experimental studies. It is used in private clouds as IaaS (Infrastructure as a Service) system and is compatible with Amazon EC2 cloud. Eucalyptus uses existing infrastructure to create scalable and secure web services layer that abstracts compute, network and storage from user to offer simple IaaS on the fly. It takes full advantage today's modern infrastructure virtualization software that is capable of scaling up and down services depending on application workloads [3]. Compatibility with Amazon EC2 gives possibility to reduce development costs of the experiments and analyses, conducting them first on the private cloud.

3.4.1 Eucalyptus architecture

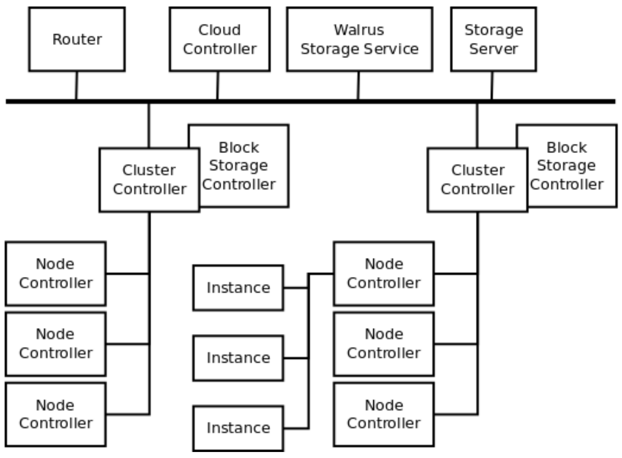


Figure 3.2: Layout of the Eucalyptus architecture, showing how the services are connected with each other. Router enables access from the outside world to connect with the cloud and running instances [54].

The architecture of the Eucalyptus is simple and flexible with a hierarchical design. It uses SOAP (Simple Object Access Protocol) service that is emulation of Amazon EC2 service, where users are allowed to start, control and terminate virtual machines on the cloud. Layout of the architecture is shown on figure 3.2. The architecture consist of four main parts: [38]

1. **Node Controller** controls the execution, inspection and termination of VM instances on the host where it runs.

2. **Cluster Controller** gathers information and schedules VM execution on specific node controllers, as well as manages virtual instance network.

3. **Storage Controller (Walrus)** is a put/get storage service that implements Amazon's S3 [35] interface, providing a mechanism for storing and accessing virtual machine images and user data.

4. **Cloud Controller** is the entry-point into the cloud for users and administrators. It queries node managers for information about resources, makes high-level scheduling decisions, and implements them by making requests to cluster controller.

Node Controller

Node Controller is located on every node that is designated to run VM instances. It controls execution of virtual machines (running, adding and removing them) and makes queries to discover the node's physical resources and gather state of the running VM instances. Node Controller is controlled by Cluster Controller. Node Controller executes commands sent by Cluster Controller, addition to this, it will give feedback of the current health and performance of the node to notify Cluster Controller of any problems that might arise.

Node Controller broadcasts to Cluster Controller characteristic of physical machine - number of cores, total amount of memory and free memory, total amount of disk space and available disk space. Information collected can be propagated and collected for Cluster Controller in response to the commands `describeResource` and `describeInstance`. Cluster Controller can start and stop instances giving `runInstance` and `terminateInstance` commands to the Node Controller. Only authorized person can execute these commands, e.g. only the owner of the instance or administrator can trigger termination command. While executing command for running new instance, Cloud Controller has to wait confirmation from Node Controller about resource availability.

To start the instance, Node Controller makes local copy of the image file (containing kernel, root file system and ramdisk image) over network from remote image repository. These images are packed, copying them over network and unpacking them takes time and therefore there are couple of minute delays while the virtual image starts running and is accessible. It also depends how loaded the network and physical host is and how much power the virtual machine has. Hypervisor (e.g. XEN) is instructed to boot the instance. To stop the instance, the Node Controller instructs hypervisor to terminate VM, close network endpoint (each instance gets unique IP address from the IP pool) and clean up files associated with the image. Amazon EC2 provides private IP address and public IP address for each instance with corresponding DNS host names.

All the Node Controllers should have access to the outside world or at least have access to inner network to allow users to connect with requested instances.

Cluster Controller

Cluster Controller has network connectivity with Node Controller and Cloud Controller. Many of the functions are similar to Node Controller, but are meant for running on every Node Controller the Cluster Controller is connected. For example commands `runInstances`, `terminateInstances`, `describeResources` and `describeInstances` gives possibility to execute them on several Node Controllers. Cluster Controller has three main functions: (i) schedule incoming requests for starting instance (check availability of the resources), (ii) control the instance virtual network overlay and (iii) gather information from a set of Node Controllers assigned to the Cluster Controller.

Cluster Controller takes care of finding most suitable node to run virtual machines. When new requests to start instances has been made, Cluster Controller checks from set of Node Controllers available resources using `describeResource` to find out most suitable candidate for running requested image.

Storage Controller (Walrus)

Walrus is a data storage service that leverages standard web services technologies (Axis2, Mule) and its interface is compatible with Amazon's Simple Storage Service (S3) [35]. Walrus implements the REST (via HTTP), sometimes termed the "Query" interface as well as the SOAP interfaces that are compatible with S3. Walrus provides two types of functionality [38]:

1. Users that have access to Eucalyptus can use Walrus to stream data in to/out of the cloud as well as from instances that they have started on nodes.

2. In addition, Walrus acts as a storage service for VM images. Root file system as well as kernel and ram disk used to instantiate VMs on nodes can be uploaded to Walrus and accessed from nodes.

VM images are stored and managed by Walrus. VM images are packaged, encrypted and uploaded by standard EC2 tools provided by Amazon. These tools compress images, encrypt them using user credentials, and split them into multiple parts and are accompanied by image description file, called the manifest (in XML format). Node Controller downloads requested images from Walrus before instantiating it on a node. If authentication between Node Controller and Walrus is successful, Walrus will decrypt the image and send it over the network to Node Controller.

Cloud Controller

Eucalyptus underlying virtualized resources are exposed and managed by the Cloud Controller. The Cloud Controller is a collection of web-services which are best grouped by their roles into three categories: [38]

1. **Resource Services** perform system-wide arbitration of resource allocations, let users manipulate properties of the virtual machines and networks, and monitor both system components and virtual resources.

2. **Data Services** govern persistent user and system data and provide for a configurable user environment for formulating resource allocation request properties.

3. **Interface Services** present user-visible interfaces, handling authentication and protocol translation, and expose system management tools providing.

In addition to the programmatic interfaces (SOAP and REST services), the Interface tier also offers a Web interface for cloud users and administrators to ease the use of the cloud system. Using a Web browser, it is possible for new users to sign up for cloud access, download the cryptographic credentials needed for the programmatic interface and query the system. The administrator can additionally manage user accounts, inspect the availability of system components and make modifications to the resources.

It also gives collection of interface web service entry points for user requests using a variety of interface specification (e.g. EC2's SOAP and URL query, S3's SOAP and REST). Users can make requests using either the EC2 SOAP or EC2 URL query protocols. This has allowed wide variety of tools which comply with the EC2 and S3 interfaces to work without modifications.

3.5 Amazon Cloud

Amazon is public cloud provider who provides on-demand computational and storage resources to the clients. It can be used to run scalable applications and cost of running these applications depends on the storage, compute and transfer resources it will consume. Different execution plans on the same applications may results in different costs. Framework built by this thesis is using Amazon EC2 resources to allow automatic scaling of the service, thus trying to minimize the cost for running the service. Using MediaWiki application, it is possible to implement realistic benchmark with real data and traffic. Scaling the servers in the cloud should be transparent to the end user and in ideal case, no failure should happen while servers are provisioned.

3.5.1 Amazon EC2 and Auto Scaling

Amazon EC2 (Elastic Compute Cloud) is a web service that provides resizeable compute capacity in the cloud. It is designed to make web-scale computing easier for developers [19]. It provides great variety tools to manage the cloud automatically, manually or Amazon customers can make their own provisioning tools accompanied with Amazon tools to take control over the cloud. With Elastic Load Balancing [34] and Auto Scaling [9], giving possibility to use scaling properties of the cloud to easily distribute requests using Elastic Load Balancing service between varying amount of back-end servers, that are allocated by Auto Scaling service to the cloud. With these services, it is easy to turn any application in the cloud scalable and this will be transparent for the end user, as they are not aware, that servers are removed or added to the cloud. Elastic Load Balancing gives possibility to distribute load between different regions and availability zones depending where the requests were made to make sure, that the traffic between servers and clients have low latency.

This thesis will look properties of Amazon Auto Scale and how to set up the service in the cloud. Amazon Auto Scale works using different thresholds assigned by the user to manage servers in the cloud. This thesis will look, when using CPU based provisioning, how the servers are added and removed through provisioning process and what parameters should be used. For using Auto Scaling, user needs to install additional command line tools [12] to define Auto Scaling group and policies used to measure the performance of the instances in the group and take provisioning decision based on that. Amazon Auto Scale is free of charge, user has to pay for the instance hours for the instances running in the scaling group, that were added by the Amazon Auto Scale.

3.5.2 Amazon pricing

With Amazon cloud you have to pay for what you use. Amazon gives great variety of different servers with different characteristics. They are all charged with different prices. The time of writing this thesis, regions US East (Virgina) and US West (Oregon) [20] are the cheapest one where to run the servers. Amazon charges for running the instances by the time they are engaged. The time is measured from the instance boot (it takes some time until the instance pending state is changed to running and for pending state Amazon does not charge money) to the instance termination. Charge is taken by calculating the full hour the instance was used, meaning that partial hours are charged as a full hour. This should be taken into consideration when closing the instances. For example, if you have large list of different instances started in different times, it would be wise to select those instances out from the list, which are closest to the full hour to avoid paying for the computational power that was not fully used.

3.6 MediaWiki architecture

MediaWiki application in general uses different technologies to run the system. These technologies are not mandatory to use for a single server configuration with small amount of traffic, but for site, that has large amount of users, it is highly recommended. MediaWiki can be easily installed on top of LAMP [15] stack (or WAMP - for windows, XAMPP - supports most of the operating systems) without need for making any additional changes to the system. MediaWiki application is written in PHP, but there are some speed concerns, as the PHP is interpreted language, meaning each time web page is visited, the code to render the page has to be interpreted again, making rendering the pages CPU intensive task. MediaWiki is using regular expression to convert text in MediaWiki format to HTML every time a user visits the page. There are different applications and daemon programs to improve performance of the MediaWiki application.

Performance gain through caching. The main idea for improving the performance is to try to cache the data and the code, this will reduce the amount of time required to run for each request and therefore minimizing CPU usage. MediaWiki supports several data caching methods, e.g. saving into files, storing visited pages in the database or storing pages in the memory with help of the memcached daemon. This section outlines some basic tools to run and improve the performance of the MediaWiki. For single server configuration, it is recommended to use file caching, as it is easiest to set up and probably only way, when using virtual Apache server provider, you might not have permissions to access shell to install additional programs and start them [4]. File caching will use local or mounted file system (mounting extra space from the cloud using Walrus or S3 service) to store already compiled pages on the hard disk that can be easily fetched and sent to users, who revisit the page. MediaWiki stores its content in MediaWiki mark-up language in the database and this needs to be formatted into HTML. It means that each request to the database, the application needs to go through the content and format given text to appropriate HTML format. This process will uses regular expressions and therefore for larger contents will use large amount of CPU on the Apache server. To improve the performance of the server, it is important to use caching, as it will eliminate the need to parse the content into HTML again.

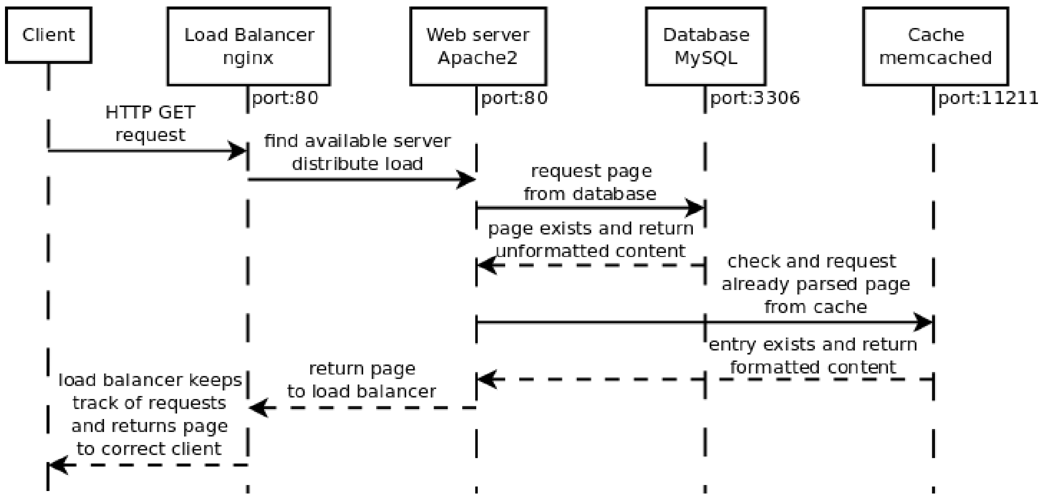

Client
Load Balancer nginx
Web server Apache2
Database MySQL
Cache memcached
port:80
port:80
port:3306
port:11211
HTTP GET request
find available server distribute load
request page from database
page exists and return unformatted content
check and request already parsed page from cache
entry exists and return formatted content
return page to load balancer
load balancer keeps track of requests and returns page to correct client

Figure 3.3: Flowchart showing how request and responses between different servers are being transferred when the requested page is already cached. Note that MediaWiki still makes request to database to see, whatever the page exists and is not changed.

Figures 3.3 and 3.4 shows typical flow through the system between different servers to show, how the request is parsed and completed. Even though for already cached pages, still request to MySQL is generated to see whatever the page still exist in the system and is not changed. Second query goes into memcached, that gets already cached page, this page is compiled from previous user visiting the page and therefore fetching these types of pages are faster as there is no need to parsing it again (formatting structured text to HTML, this will also include checking whatever links are active or dead, including pictures, formulas and other objects from the page). Figure 3.4 demonstrates how multiple requests going through the system to memcached and MySQL database server to fetch additional information that is needed for parsing the content. First, each entry is asked from the memcached, if it does not exist in the memcached, it is queried from the MySQL. If the MySQL query is successful (link/page/object exists in the database), it is stored in memcached, so the later queries should be faster. Because of multiply queries made by Apache server, it would be good, if memcached, MySQL and Apache servers are located in the same availability zone to reduce network latency.

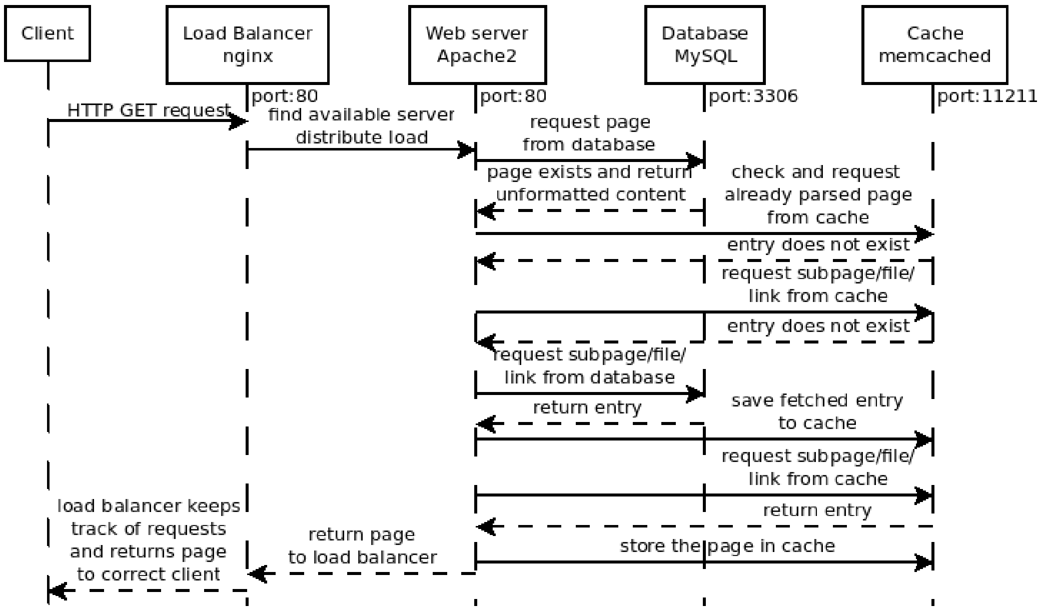

Client
Load Balancer nginx
Web server Apache2
Database MySQL
Cache memcached
port:80
port:80
port:3306
port:11211
HTTP GET request
find available server distribute load
request page from database
page exists and return unformatted content
check and request already parsed page from cache
entry does not exist
request subpage/file/ link from cache
entry does not exist
request subpage/file/ link from database
return entry
save fetched entry to cache
request subpage/file/ link from cache
return entry
store the page in cache
return page to load balancer
load balancer keeps track of requests and returns page to correct client

Figure 3.4: Flowchart showing how request and responses between different servers are being transferred when the requested page is not cached. There are several queries and calls to memcached and MySQL server to compile the page. Once ready, the parsed page is stored in the memcached.

3.6.1 Memcached reducing load on database server

Memcached is key-value based cached storage database program running as daemon on the server, that holds all of the stored information in the memory. It is powerful and fast program able to retrieve entries from the database in $O(1)$ time and currently, maximum entry size is limited to 1 MB. Wikipedia has pages exceeding this limit, but it is solved by packing content with `gzcompress`. This gives more flexibility in storing larger cached pages in the memory [5]. Memcached is not a persistent storage engine, meaning that when restarting the system, the cache becomes empty. This will increase response time of the requests and to stabilize the system, the cache needs to be filled again to improve the performance of the service.

Memcached in the MediaWiki application environment is deployed as a separate server. All the back-end servers can connect with central memcached server to fetch pages and add new content into the memory. If using several memcached servers, system administrator has to be careful, when adding or removing memcached servers, as the cache becomes invalid, making the service slower (response times increases, as the content of the pages has to be parsed again) until the cache is filled again.

Three experiments have been conducted to see the difference, how much memcached and XCache can improve service speed. First experiment did not use neither of the caching capabilities. The content was generated for each page visited by PHP again and the PHP code was not cached in the memory. Second experiment uses XCache to cache PHP code into memory and the third uses both caching option, where memcached was filled with the requests. For the experiments, Amazon EC2 `c1.medium` instance was used (see later chapters, why `c1.medium` instance was selected). Test set consisted of random pages requested from the system, but for each test, the URI addresses requested stayed the same to have fair comparison between experiments. The instance used for the testing gained slower CPU E5506 from the cloud, which has clock speed 2.13 GHz (this is slower compared to E5510, which has 2.33 GHz clock speed). Without any caching the average response time for one hour experiment was 469.6 ms. Enabling the XCache for PHP code caching, the average response time improved to 335.7 ms. It improved the speed of the service, but not su much as enabling memcached and the requests were fetched from the cache. The average response time dropped down to 79.2 ms. Using both caching options, the service speed increased at least 6 times. Figure 3.5 shows response times distribution for all the experiments. Parsing the content by PHP is using more CPU power, showing clearly, that regular expressions takes more time when converting the content to HTML and validating the links. With constant load, one request using memcached and XCache for `c1.medium` Apache server used on average 1.5% of CPU and without memcached, Apache server used on average 7.4% of CPU. When disabling both caching options, the average CPU usage increased to 11.2%.

MediaWiki configuration file `LocalSettings.php` allows to change cache validation time limit. This will define the time it takes to mark entry in the cache as invalid, meaning the requested page has to be parsed again

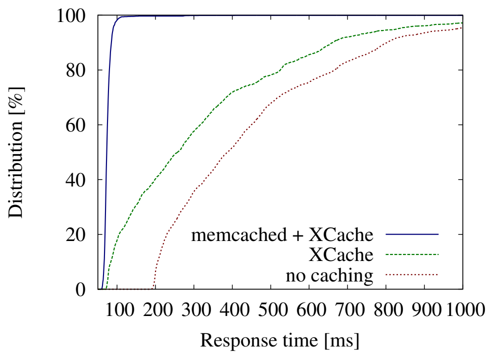


Figure 3.5: Cumulative distribution of response times for `c1.medium` instance with different caching policies.

and converted to HTML. Changing the configuration file will invalidate the whole cache and it is not accepted behaviour, when adding new servers and changing IP addresses for memcached and MySQL in the configuration file, as it will affect response times. To overcome this, variable `$wgCacheEpoch` has to be changed to 1 in order to disable the invalidation of the cache, when changing configuration file.

Documentation from MediaWiki gives instructions to start and activate memcached on the MediaWiki [24]. Configuration file variable `$wgParser CacheExpireTime` shows for how many seconds the parsed content stored in the memcached is considered valid. If this time has been passed, fetching information from memcached will return false value and old value is cleared from cache. New value is therefore requested from MySQL database (including converting the content to HTML) and added to the memcached memory. Memcached does not itself go through values stored in its memory, they are invalidated if someone tries to fetch old data. To get most of the system, it has to be made sure, that memcached is properly installed and running. On default, MediaWiki timeout for memcached server is set to 10 seconds, if the memcached server is not reachable and pages from MediaWiki application takes more than 10 seconds to respond, then there is a problem with memcached installation and configuration.

3.6.2 LAMP stack

LAMP refers to Linux, Apache, MySql and PHP installation. There are several variations for having web-application server, but this thesis is looking for LAMP configuration to run MediaWiki application on the back-end servers. Requests from nginx reverse proxy are divided between these LAMP stacks. The back-end servers are using PHP version 3.5.3, Apache with version 2.2.17 and MySQL server with version 5.1.54. MySQL is disabled for Apache back-end servers and is used as a separate server.

Apache configuration

Apache is open-source software working as a server in default configuration on port 80 and serving files from the file system. Default Apache configuration is meant for serving static files and is not suitable with serving dynamic content as the service time and resource utilization differs. Serving static pages does not use much resources compared when the page is generated and render by the help of PHP. To improve the service throughput and reducing risk for overloading the servers, the configuration file has to change for reducing maximal number of allowed users and to reduce timeout period for requests. This helps to prevent overloading the Apache server with excessive requests and also reduce memory footprint. Memory can be real bottleneck, if the server starts to run out of free memory, the operating system starts swapping the memory. Swapping means that operating system starts to write old memory chunks to the hard disk, making fetching information from memory significantly slower, also adding items into memory is going to be slower and thus increasing back log.

Following configuration is valid for `c1.medium` server on Amazon cloud. It has been observed, that with following configuration (using memcached and XCache) on average, one `c1.medium` server was able to handle 26-34 request per second depending, which CPU was assigned to the virtual machine, state of the virtual machine host, network utilization and wear of the system. Reducing `MaxClients` for each Apache to 10, helped to make service more stable under heavy load. The default option was set to 125, which is relatively large number as the server is only capable of serving 4× fewer requests and the requests should not be in the system at the same time. Reducing `timeout` to 10 seconds and disabling `KeepAlive` to free up sockets immediately after the content has sent to the client also improved service throughput and making overloading the system harder. When using default configuration, it can be noticed that, while generating more requests the server was capable of serving, the throughput started to decrease, showing that for each new request a new process was made. Because not using correct number of limited maximum clients, the system had more jobs inside than it was able to serve and affecting other jobs, making the response time longer, resulting in one minute system load going over 20 units. Using 2 core machine, meaning that when 1 minute load is going over 2, the job queue is longer than the server is able to serve. It has been problematic, as with higher loads the server tends to crash.

PHP configuration

PHP is most widely used scripting language for serving dynamic content over web. The language is simple to use, there is no need to define variables and each variable type can be changed on the fly, that is not possible for many other languages. PHP is interpreted language, meaning that for every request the written code is parsed again by the interpreter. This method is slower compared to compiled languages, but gives more freedom for programming. There are several applications and utilities out that helps to cache PHP code, so it is not going to be parsed every time new request is made. This can lead to 2 to 5 times faster code execution depending how complex the code is and how many different source files are included. For complex and large PHP files caching the code can help a lot as the server does not need to read the files from the hard disk every time a new request is made, helping to reduce response time. While decreasing response time, the server is able to serve more requests. But it is not always good to use PHP opcode cacher, as going through small codes can increase overall time.

MySQL configuration

Default MySQL configuration is modest and is meant to run along with the other applications in the same server. Infrastructure configuration provided by the current thesis needs to run MySQL database on a separate server for MediaWiki, meaning that MySQL can use more resources and to improve the throughput, some of the parameters should be increased. Increasing `max_connections` allows more connections to be handled by the database. With higher loads, the old connections changed to idle state and were not given to the new connections, limiting the amount of new connections made to the database. Reducing TCP packet timeout for Ubuntu allowed to kill such connections. If MediaWiki cannot make connection to the database, it will return database error to the end user. While generating at least 300 request per seconds, some of the connections were changed to the idle state and MySQL ran out of free connections, blocking new incoming requests and making service unreachable.

The other configuration part concerns where to store the database files. The default option will use image file system to store the data, but it is limited in size and has to be considered, while starting to upload data from Wikipedia dumps to MySQL database. The second option is to use S3 (or Walrus) storage space, but this means that the extra storage has to be mounted to

MySQL instance. MySQL configuration variable `datadir` has to be changed and pointed to the mount point. If starting MySQL instance later, when the image is already bundled, it will not be able to start the MySQL service, because data directory is missing. It is needed to the first mount S3 storage with instance and then start MySQL service. It makes things more complex for configuring and setting up the infrastructure, as it has to make sure, whatever the attached volume is correctly working and is mounted to correct folder.

To mount Amazon S3 storage with instance, it is needed to run `ec2-attach -volume`. When using newly created volume, it is need to format the volume in order to use it. To see, if attached volume command has worked, it can be used command `sudo fdisk -l` on the MySQL instance and check, whatever the device is in the list, should be the last one. Mostly it can be `/dev/sdd`, but may vary depending on the image. Using command `sudo mkfs -t ext4 /dev/sdd`, it is possible to format the volume with ext4 file system. After this, it is needed to make mount point, this can be folder in the file system. To create new directory, use command `sudo mkdir /var/data` and for mounting attached volume, use command `sudo mount /dev/sdd /var/data`. Now the `/var/data` is containing mounted volume and it is possible to write files there. System administrator can copy MySQL data into newly created volume, change the configuration file `datadir` location and make sure, that mysql user has rights to write into this volume. If everything is done, it is possible to start MySQL service, it should start instantly. If it takes time or freezes, it can mean, that there is something wrong with the configuration. Rechecking permissions and whatever the image is mounted or not can solve the issue.

3.6.3 Operating system configuration

There are some limitations in Ubuntu operating system, that might also affect service quality and speed. Most important is to increase open file descriptors [17] for nginx, memcached and MySQL servers as opened sockets are also counted as file descriptors and under heavy load these servers might run out of free sockets and eventually starting to block incoming connections making the service unresponsive. Because using same image for all the servers, then these changes are system wide and affecting all the servers running in the cloud by the framework.

3.7 Study of load balancers

There are different load balancer servers that work as a reverse-proxy servers and are capable of redirecting traffic to the back end servers. Each of them has different characteristics and maximum throughput, depending how it was built and which language was used. This thesis will look two load balancers. First is a Pen and the other is nginx, they are written in C language, meaning that they should have good performance and high throughput.

3.7.1 Pen

Pen is a simple reverse-proxy server, written in C language [39]. It is easy to set up and make it running. It uses least round-robin algorithm for distributing incoming requests. It keeps track of the clients and tries to send them to the same server. It ensures that when using PHP sessions, user state is remembered. This helps to stay user logged in and the state of the user can be saved in the same server, meaning there is no need for centralized session (but when removing the server from the cloud, the state has to be transferred into existing server). If lookup table is full (this is table, where pen is keeping track of the clients) it tries to remove oldest entries from the table and use most idle server to direct the traffic. It is possible to remove lookup method and only direct traffic to most idle server (which has least connections). For benchmarking purposes, it would be best to remove IP lookup table as load generators have fixed IP addresses and the load is therefore not evenly distributed or distributed only to small amount of servers, leaving other servers into the idle state. Note that nginx also provides possibility to use lookup table (it is called hash table) to redirect the traffic to the back-end servers, depending on the client IP.

Pen high availability and redundancy

Pen can ensure high availability by checking individual servers, whatever they are up and running or are down and maintained. Pen supports active-passive failure server configuration when running multiple pen servers increasing service redundancy, using VRRP (Virtual Router Redundancy Protocol) to decide which of the Pen servers is active. *The Virtual Router Redundancy Protocol (VRRP) is a computer networking protocol that provides for automatic assignment of available Internet Protocol (IP) routers to participating hosts* [40]. This thesis only focuses for single load balancer configuration and does not use active-passive failure configuration for load balancers.

Running pen from command line

Pen starts from command line and can be completely configured there. It is also possible to define configuration file. When trying to reconfigure the servers, it is needed to kill the process first and start it again.

```
$ sudo pen -r -t 10 -S 2 -p /var/run/pen.pid
   80 server-1:80 server-2:80
```

Above command will start pen load balancer, that accepts connections on port 80 and redirects traffic to the servers `server-1:80` and `server-2:80`. It has connection timeout set to 10 seconds with parameter `-t` and ignores IP lookup for the users with parameter `-r`. Parameter `-p` is used for storing process ID for later usage and restarting the service. Starting services under port 1024 in Linux, it is needed to have super user privileges. Following command will shut down the pen. This can be easily written into shell script to support running it as a service.

```
$ sudo kill -9 ‘cat /var/run/pen.pid‘
```

3.7.2 Nginx as reverse-proxy server

Nginx is reverse proxy application running on the server as daemon written in the C language by Russian programmer Igor Sysoev [31]. It uses network OSI layer 7 (Application layer) [32, 50] to processes and forward the requests. This means that every request is entering to the operating system and is not forwarded by the network card, thus meaning each request to nginx (working as a reverse proxy), is taking computer CPU time to process the requests to the back-end servers. Nginx uses native operating system functions to poll the existing asynchronous connections, making it possible to serve large amount of request with small CPU usage. It uses Unix socket layers, which also decreases CPU overhead as it passes messages between client and back-end server directly through network card, without need to communicate with operating system.

On `c1.medium`, one nginx server is capable of handling over 10 000 requests per second (but with this traffic rate, it is possible to run into network bandwidth problem). Wikipedia is getting traffic at 75 000 requests per second at the peak time [28]. To be capable of handling such load it means that bigger machines should be used or servers should be separated between different regions to distribute load evenly and reducing traffic for one front-end reverse proxy server. Testing at these scales is difficult task and this thesis does not involve experiments with same amount of traffic as Wikipedia is getting along. Nginx has been already installed in 19,121,348 (11.28% of all servers) active servers based on Netcraft Web based survey on October 2011 [6], showing that large amount of system administrators are already trusting nginx as a front-end servers.

Nginx load balancing algorithm

The default distribution model for nginx server is in the round-robin fashion [7]. This model is one of the oldest and each system with capabilities of distributing requests supports this model by default. Round-robin method uses a list of servers and an indicator to determine, in which server the last request was sent. For every new request, the indicator value is increased and next server is picked from the back-end server pool. If the value exceeds servers count in the list, it starts from the beginning. This is not effective method for distributing load, where request times vary, especially if the servers are under high load or the requests are not in equal size in terms of CPU time they take. Wikipedia content varies in size and complexity, there are some pages that are just redirects and others, which are full of information, meaning that they will need more CPU processor time to parse the content. While distributing requests in round-robin fashion, we might direct large page request to one server and redirects to other server, meaning that the first server might not be able to serve the requests in reasonable time, but the latter server is sitting in the idle state, the resources are not distributed equally. Variation for back-end servers CPU usage from 50% to 90% has been observed, while conducting experiments with default round-robin and using different size of page requests. Unequal distribution can overload the servers with high CPU utilization and some of the requests are dropped, thus not resulting in best performance, that could be achieved from this configuration.

Nginx supporting least round-robin

To make distribution of the requests on the reverse proxy much better, it is possible to use add-on module fair [8]. Fair module for nginx does not use queuing on the proxy side. The main idea for using fair is to drop excessive requests. It has option to define how many concurrent request one server can have. If there has no room left to send the request to back-end servers (i.e. all the servers are filled with the maximum amount of concurrent request and there are no idle servers), the request is dropped. This behaviour of dropping excessive requests helps to avoid overloading back-end servers. Even if they are working at full power, they still manage to maintain good response time and are capable of serving the requests. There is a thin line between maximum throughput and overloading the server. This must be carefully observed, but in the best case scenario, the maximum amount of concurrent connections should be smaller, as running servers over 90% CPU usage is not normal.

Nginx modules

Nginx has different modules [1] and addons [2]. With `HttpStubStatusModule` it is possible to gather information about active connections and how many visits there have been done. This helps to gather how many requests per time unit there have been and this can be used for provisioning correct amount of servers using various policies. This module is deployed as a separate page on the server and can be accessed with simple `HTTP GET` request and user can define the page URI (Uniform Resource Identifier), where to access it. Unfortunately the statistics given by nginx is not as good as compared to HAProxy. HAProxy, is giving various information, including many connections to the back end servers have been done, how much bytes have been transferred and maximum queue for each server. To get similar information, it is needed for the back-end Apache servers to use module `mod_status` [3]. It will give information amount of connections generated and can be logged to identify where and how the requests are distributed. Keeping track of this information can help to find possible bottlenecks in the service, are the requests forward to the back-end servers and are the requests distributed by load balancer equally.

3.7.3 Comparison between nginx and pen

Both of the load balancer with different configurations have been tested to identify best suitable load balancer. Pen is simpler to start the service as it can be done without any configuration file through the command line. Nginx needs some additional tuning, manual reading to get configuration right and how to use upstream module to set up reverse-proxy. Nginx has better support for reloading the configuration on fly and has better documentation. For both cases, the documentation of load balancers are incomplete

[1]http://wiki.nginx.org/Modules
[2]http://wiki.nginx.org/3rdPartyModules
[3]http://httpd.apache.org/docs/2.0/mod/mod_status.html

and much of the configuration examples have to be looked from different Internet communities and forums. Nginx has good support for Russian and has complete documentation in Russian, but lacks of English help.

Nginx is capable of serving more requests and has better overall performance compared to Pen. This is why the thesis will use nginx over Pen to distribute the load. It was easier to read total requests from nginx, as it was already provided by `HttpStubStatusModule` to show active connections and how many requests have been made. This task was more complicated for Pen, as there was a need to run shell script to receive the output of the statistics. Pen authors are strongly advising to use cronjob for this task. But even with this output it does not tell you how many requests have been generated and how the requests have been divided. It only shows active connections, list of last clients and how much traffic (bandwidth) has been gone through and only possible indicator of arrival rates would be to use bandwidth and average size for each request.

Nginx has plenty of third party libraries, but to use these, you have to compile the nginx from source code and add library to the source. Some of the libraries are only compatible with old builds of nginx as they have not been updated by the authors and therefore cannot be used with latest nginx versions. Some older nginx versions have problems with stability and user has to be careful when running older version software as there might be security holes and bugs. It is strongly advised to read mailing lists and other resources to see, if there are any known major problems. This thesis uses library called fair and in order to add it to nginx, it was needed to use older version of nginx to get it working. For that purpose nginx version 0.8.32 was used and no strange behaviour for different experiments was observed. Some earlier versions had problems with reloading the configuration as they could not terminate old daemons and the service became stuck and unresponsive. It had to be killed manually and restart the service.

3.8 Configuration of the infrastructure

For keeping things simple, framework build by this thesis uses single image and all the services are installed there. They are disabled at start up. The idea is to run and start only necessary services depending which role the server has. By default, additional HTTP server at port 8080 was started, that was showing content from `/mnt` folder. This mount point had plenty of room to write and content saved there is not bundled to the new image (it needs to have writing permissions, otherwise only privileged user can access it). This is good location to temporarily store private keys and bundle the new image together. The image is packed and has to be uploaded to the cloud in order to get the new image working and ready to launch new instances. This helps to protect leaking the private key to other users, especially when the image has marked as public, extra care has to be taken. HTTP server running on port 8080 allows to easily download content from the cloud without the need to use `scp` (secure copy tool under Linux). Cloud user has to make sure, that additional ports are added into security group, otherwise they cannot be accessed outside the cloud.

3.8.1 Endpoints for SciCloud and Amazon

Endpoints show where the SOAP service is accessible. From there, it is possible to feed in commands to manage the cloud (kill instances, run new instances, query list of already running instances). SciCloud is accessible from `https://katel42.hpc.ut.ee:8443` and Amazon EC2 `us-east` region is accessible from `https://ec2.us-east-1.amazonaws.com`. For each region, Amazon EC2 has different endpoints. The excellent tool to get overview of running instances, add or remove instances is Mozilla Firefox add-on Elasticfox [4]. In order to get it working, user needs to configure endpoint location and enter keys, that were retrieved from the cloud administrator or downloaded from Amazon Web Service console. It is also possible to use `euca` or `ec2` command line tools to run and terminate instances (look `euca-run-instances` and `euca-terminate-instances`). Commands starting with `euca` are meant for Eucalyptus cloud and `ec2` commands are meant for Amazon EC2 cloud. Elasticfox is compatible with Amazon EC2 cloud, but user can use Amazon Web Service to log in to the web interface and manage keys, images and instances there.

3.8.2 Setting up services and MediaWiki installation

In order to configure the infrastructure in the cloud, at the beginning user has to use the base image, where to build its infrastructure. This thesis uses 11.04 Ubuntu with 32bit architecture for Amazon EC2, it will support running `m1.small` and `c1.medium` instances. Using aptitude tool (`apt-get install packages`), following packages were installed: apache2, apache2-mpm-prefork, mysql-server, php5, php5-mysql, php5-xcache, memcached, sysstat, openjdk-6-jre, ec2-api-tools.

Apache and php5 are used to serve the dynamic content using MediaWiki application. XCache will improve PHP script execution times as the code

[4] http://aws.amazon.com/developertools/609?_encoding=UTF8

is cached and do not have to read it from the hard disk, MySQL is used to run database and the content of MediaWiki is stored there, memcached will enable caching support for MediaWiki, sysstat gives tools to measure performance of the instance and openjdk is used to run Java jar files. EC2 api tools add command line scripts that are necessary for framework produced by this thesis. These are used to configure the cloud, add and remove instances. Some of the packages might already be installed, depending which image was used to run the instance.

MediaWiki was downloaded [5] (version 1.16.5, released in May 2011) and unpacked to folder `/var/www`. This is default location for Apache home directory and files from there are shown to the users who visit the web page. MediaWiki has simple installation interface and user is automatically redirected from front page `localhost/mediawiki/` to the installation page. To get installation process to be smooth, it is recommended to use single instances to install all the necessary services and install MediaWiki to local MySQL database. Make sure that MySQL service is running when installing the MediaWiki. It will create necessary database structure to store the Wikipedia content. Script for uploading the data from Wikipedia dumps to database do not support using prefix for table names, meaning that when installing MediaWiki, prefix form should be left empty. When installing the MediaWiki, user can use localhost as a host for MySQL and memcached services. This should later be changed to the correct instance IP addresses, where the services are deployed, but this is automatically done by the framework. Using MySQL and memcached service to install the application, it is possible later to check whether both are running correctly and configuration works. This is important step, because if later on bundling the image together and finding out, the service does not work correctly, there is need to fix the errors and problems. If later on requesting the new bundled image, for every instance, these fixes have to be done again, meaning it would be better to fix the image, bundle it again and upload to the cloud to retrieve new ID of the image that can be launched with fixed configuration.

3.8.3 Wikipedia article dumps

If the installation of MediaWiki is done, it is time to download the articles of Wikipedia. Wikipedia makes the database dumps in regular bases to ensure data availability for mirror sites or for users who want to run it locally. When using S3 storage to store the database data, make sure that it is working correctly and it is mounted into correct place, MySQL is correctly

[5] http://www.mediawiki.org/wiki/Download

configured and MediaWiki tables have been created to the database.

In the current study the infrastructure configuration uses file system given by the image to store the data for MySQL database. Amazon EC2 gives 10 GB disk space for `c1.medium` instance, it was enough to initialize database with two Wikipedia article dumps of January 15, 2011 [6] [7], consisting of 166,977 articles. This corresponds to 2.8GB of file system space. The operational dataset, however, is composed of about 1000 randomly selected articles across the database. These restrictions have taken into account to avoid having large time spending on filling the cache. Filling the cache was necessary as the experiments conducted by this thesis were interested in performance of stable system with high cache hit ratio. Using fully filled cached for MediaWiki and using Wikipedia articles dumps, it is possible to achive 97% memcached hit ratio and 75% of queries are cached by MySQL.

These dumps are in XML format and in order to upload them in to the database, they have to be converted to SQL queries. There is Java program from WikiBench called (available on DVD) `mwdumper.jar` [8], that can be used to convert XML to SQL, pipe the output of the dumps into database using MySQL. Following is example how to upload dumps into database.

```
$ java -jar mwdumper.jar --format=sql:1.5
   enwiki-20110115-pages-articles1.xml.bz2 |
   mysql -u root -p wikidb --password=root
```

To check, if it is working, open up browser and visit MediaWiki web page and click on "Random article". If new page is fetched and it is different than "Main Page", MediaWiki has been successfully installed. Framework build by this thesis takes care of the rest, configuring correct IPs for MySQL and memcached host, making it working in multi-server configuration, configuring back-end servers and adding them into load balancer server list.

3.9 Framework scaling instances in the cloud

Prerequisite tools are needed in order to add, remove and retrieve information about instances from command line. This study uses Amazon EC2

[6]http://download.wikimedia.org/enwiki/20110115/ enwiki-20110115-pages-articles1.xml.bz2 (192 MB)

[7]http://download.wikimedia.org/enwiki/20110115/ enwiki-20110115-pages-articles2.xml.bz2 (263.3 MB)

[8]http://math.ut.ee/∼wazz/mwdumper.jar

instance `ami-e358958a` (Ubuntu 11.04 32 bit, kernel 2.6.38) [9], it has repository for these tools and can be installed with ease using aptitude tool [30]. SciCloud is working on Eucalyptus cloud platform and it has `euca` tools to manage the cloud. Amazon in the other hand uses `ec2` tools to have access with the cloud. The commands are same as Eucalyptus is built to be compatible with Amazon cloud interface, only the first part of command differs. To install Amazon tools on Ubuntu, you have to install package `ec2-api-tools` [26] and have to make sure, that multiverse is enabled [27].

Using commands `euca-run-instances` or `ec2-run-instances` respectively for starting the new instances. Using `man` page [29] under Linux, it will give information how to use the commands and which parameters can be used. With parameter `-n` user can define how many servers are going to be started upon a request. These tools also need a private key and certificate key to access Amazon EC2 service. Instances requested from the Amazon EC2 cloud starts relatively fast, it takes average 2 minutes to get `c1`.medium instance up and running, depending of the speed of the instance. Smaller instance `m1.small` is slower to get running and takes more time compared to `c1.medium`, because it has limited CPU power and the IO performance is moderate.

Example of how to start instance on Amazon EC2

```
$ ec2-run-instances ami-e358958a -n 2 -key keypair -K user_priv_key
   -C user_cert_key --instance-type c1.medium
   --region us-east-1 --availability-zone us-east-1c
```

This command starts virtual machines image (image code `ami-e358958a`), parameter `-n` defines how many servers are requested, `-K` is private key (can get from AWS console) and `-C` is certificate key. Parameter `--instance-type` defines, which machines are required, it is also possible to define in which region the instance is started (default is `us-east`). Parameter `-key` will define key pair, that is used to allow SSH access with the instance. If the request is successful, this command will output list of requested instance identifiers. These can be used for tracking purpose, do check whatever requested instances are up and running, or still waiting in pending state. The logic for adding and removing servers is implemented by the framework and can be found, when looking CloudController source code.

[9]http://thecloudmarket.com/image/ami-e358958a–ebs-ubuntu-images-ubuntu-natty-11-04-i386-server-20111003

3.9.1 Calculating arrival rate

Nginx uses `HttpStubStatusModule` module, that will provide information about how many requests has been made to the nginx. Knowing time and arrivals from the first measurement and from the last measurement, it is possible to calculate arrival rate for that time period in request per second using following formula

$$\lambda = \frac{h_2 - h_1}{t_2 - t_1} \tag{3.1}$$

where t notates time in seconds, h notates arrivals to the system, index 1 means first measurement and index 2 means latter measurement. With this information, it is possible to calculate how many servers are going to be needed, keeping in the mind the service time and maximum throughput of one server. Requesting additional servers can be achieved running `ec2-run-instances`. If this command is successful, it will return list containing unique ID for each requested server and this can be used later to determine, whatever the requested server has started and is running, is still pending or in rare conditions have failed. When requesting a new server, it takes some time to copy the image to physical host and start the virtual machine. For programming logic, it is possible to use separate thread to watch status of the requested instances and not affecting the main loop. If the instance is becoming available, it needs to run automatic script to change the IP addresses in the MediaWiki configuration file and start Apache server. Addition to this, new servers have to be added to the list of upstream servers in nginx configuration and reload nginx. Using command nginx and sending signal with parameter `-s` to execute following command `nginx -s reload` will reload already running nginx configuration. This will spawn new processes and sends signal to the old processes. If there are many jobs in the system, nginx old processes waits until all the requests are served and then will kill itself. This allows transparent reloading of the load balancer, without visitor of the web page knowing, that service are restarted, making it possible to update and reload nginx configuration, without losing any connections.

3.9.2 Removing servers from the cloud

Removing the servers is easier than adding them, as there are only two steps necessary to do. First step is to remove servers that are going to put off from nginx upstream configuration and reload nginx. This makes sure that we do not lose any connection in the process of shutting down Apache server virtual machine. With command `ec2-terminate-instances instance-id` we can terminate already running instance, it will closed down quickly. Going through the steps for adding, keeping eye on already running instances and removing servers, it will give idea, how auto scaling in the cloud can accomplished. For further information, how server adding and removing processes looks like and how the servers are monitored, you have to look at the CloudController code.

3.10 Framework layout

Framework built by this study supports private (SciCloud running on Eucalyptus) and public (Amazon EC2) cloud. It is capable of starting necessary services and terminating unneeded services depending on the server role. Output of this study is the development of three main programs that will give complete overview of the cloud and have possibility to configure the cloud on fly. The framework is still in experimental state and should not be used to run in the real applications.

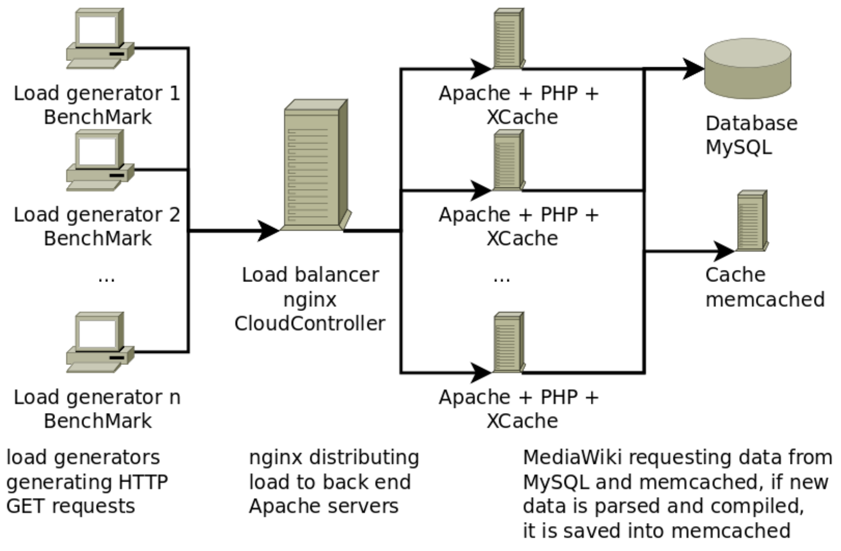


Figure 3.6: Displaying experiment configuration in the cloud environment and how the requests are going through. All the servers are located in the cloud, for reducing network latency, they are located in the same availability zone.

Figure 3.6 shows experiment configuration and how the serves are connected with each other. This figure shows how requests generated by the load generator are passed through different servers. All the servers are running ServerStatus for measuring the performance metrics.

3.10.1 CloudController

CloudController is a Java program that takes care of configuring infrastructure. It will run predefined commands using Secure Shell (SSH) depending on the role marked for the configuration on each server. It is capable of provisioning servers. Using `euca` or `ec2` tools, it can add or remove servers. These changes are synchronized with nginx load balancer to immediately direct traffic to the new instances and remove servers from the `upstream` list to avoid bad responses. Between adding and removing servers, none of the requests in the system nor upcoming requests are lost. It gathers arrival rate from nginx `HTTPStubStatusModule`, this can be used by policy algorithms to determine amount of servers needed. It will also gather statistics and performance metrics from running instances, connecting with each instance ServerStatus program and fetching collected data using TCP (Transmission Control Protocol) connection. All the gathered statistics include time measured from the beginning of experiments and outputs are written to files, it will help easily to compare results and parameters throughout the experiment.

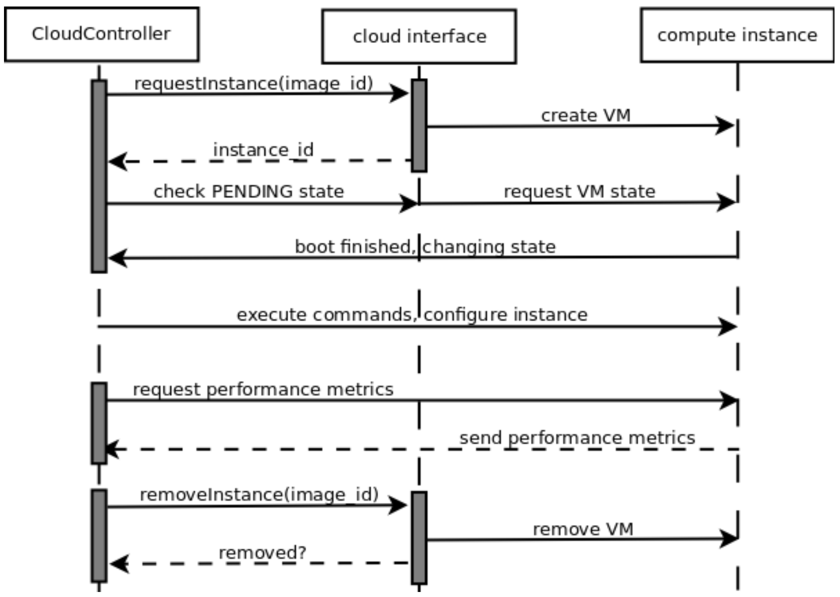


Figure 3.7: Showing interactions between cloud interface and virtual machine running on the cloud through CloudController application.

Figure 3.7 shows common interactions made by CloudController to control and monitor the status of the instances. Its main purpose is to gather statistics from running servers throughout the experiments to see how system acts with varying load, giving vague idea how the requests are distributed and how loaded the other servers are. There are two ways for provision of servers in the cloud using CloudController. First way to measure arrivals rate and run algorithm to determine, how many servers are needed to cope with such load. Other option is to measure average CPU usage what is gathered from back-end Apache servers and can define threshold, when to add servers (e.g. average CPU exceeding 70%) or remove servers (e.g. average CPU drops below 40%).

Adding or removing servers means additional configuration on the nginx configuration, changing server list in the nginx `upstream` module. For adding servers, CloudController will constantly request server state from cloud interface, to see whatever requested servers have been booted and changed its state for running. If the server changes state from pending to running, CloudController will send predefined commands to the instance to configure MediaWiki application with correct IP addresses to the memcached and MySQL servers. When the configuration has finished, it will add the new server to nginx `upstream` list and reload nginx. This process for `c1.medium` takes on average 3 minutes.

Commands sent to the instances are defined in the files and can be changed, depending whatever it needs to install additional software or start additional services. It is possible to download files from public internet site, to get additional programs or configuration files, or copy them from nginx instance using `scp`. The framework is not using any DNS service or static IP service from Amazon EC2 to define memcached and MySQL locations in the cloud. This decision was made to reduce the cost of the experiments and support dynamical allocation, without need to rely on other services. Without using DNS service, all the traffic generated during the experiment will stay inside cloud, meaning that no additional payment for bandwidth is necessary as the traffic is not redirected out of the cloud.

3.10.2 ServerStatus

ServerStatus is a Java program, that is executed at the beginning of the virtual machine operating system boot. To start it from the boot, execution command to the start up script `/etc/rc.local` has to be added. This will be executed, when the virtual machine operating system has booted up and running. It works as a server and listens incoming TCP network packets on port 5555. Every 15 seconds, it gathers performance measures from the cloud and stores it in the memory. It does not output anything to the console nor write any statistics to hard drive to minimize resource usage on the server not to affect other service performances. CloudController will collect data

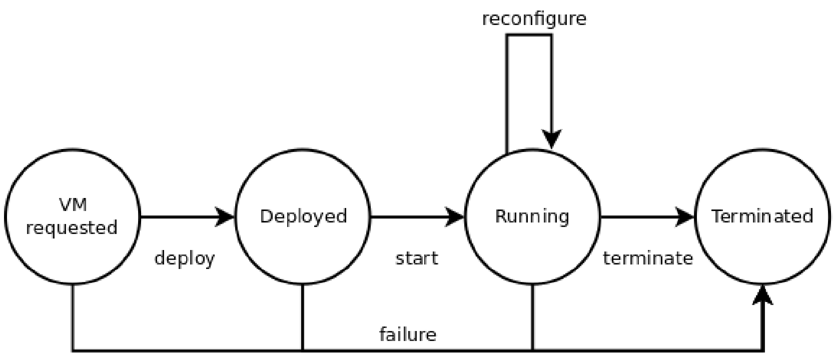


Figure 3.8: Common life cycle of instance running in the cloud and reconfigured while the service change state [53].

gathered by ServerStatus and will save the results on the nginx instance.

There are different ways to connect with ServerStatus for gathering the statistics. One way is to connect through port 5555 and send messages in plain text mode (possible to use telnet) or generate HTTP requests, making it possible to use web browser to see gathered performance metrics. It will gather data about current CPU usage, memory usage, hard disk usage, network usage, Linux load, how many running processes there are and how many requests are made to the Apache server. ServerStatus includes full set of different system metrics, that help to measure performance of the server. It is easy to combine the results, as the ClountController gathers them into the one place. These parameters can be used by CloudController making provisioning decisions, do we need to add new servers or remove existing servers, looking Apache back-end servers average CPU usage.

Possibility to access and see statistics with the web browser gives quick overview of the current state of servers. These statistics can be also looked from the ClountController output to check, if everything works as expected. CloudController provides information about how the memcached is filled and how many requests to the memcached gets a hit (the requested page is fetched from the cache). Collecting data and showing them in real time helps to notice problems early (e.g. one of the Apache servers is not reachable by nginx load balancer) and restart the experiment. It still needs active interaction by the user to keep eye on the server and does not include any kind of warning systems, for example sending e-mail to the user, while one of the server has become unresponsive or the cache hit ratio is low.

There are some limitations with ServerStatus, but is an excellent tool for giving feedback about the performance of the server. Its sole purpose is to track server usage, it does not contain any warning system and, therefore, it is not suitable for live monitoring servers in real life environments. It would be better do use Nagios and Ganglia combination while monitoring large number of servers, as there is active support and user base behind it.

Following is an example command how to access ServerStatus statistics from Ubuntu command line using bash.

```
$ echo "STATS" | /bin/netcat -q 2 127.0.0.1 5555
```

3.10.3 BenchMark

BenchMark is a Java program, that is stressing out load balancer and Apache servers. It uses trace curves to follow the amount of requests needed and uses URL list to make random page requests. It will output all the response times with the response code for later analysis. The traffic curve can be scaled depending of the user needs, it is possible to set maximum and minimum arrival values (requests per second). User can also define memcached IP to keep track of cache hit ratio to make sure that everything is going as expected. With getting cache hit ratio and server response times, it will give adequate information whenever the servers are ready for running the full-scale experiment or not. Each experiments conducted by this study use filled cache as it represents stable system. Looking cache hit ratio, it helped to determine, when it is the right time to start the experiment.

BenchMark is using Java sockets for generating the load. The requests are simple and only first line of the response is parsed by the program to gather information whenever the request was successful or not, other lines are discarded, buffer is cleaned and socket closed to make sure that servers do not wait until all the content has been loaded. All responses with code 200 are marked as successful (page redirects from URL lists were removed). MediaWiki uses error codes to define request state and gives information about current state of the service. For example, when the MySQL is not accessible or is overloaded, it will return 503 error and if pages does not exist, it will return 404 error. There are also nginx error 502, that is given by Fair module. This is returned, when the Fair module determines, that there are no more free sockets left to connect with the back-end servers (Fair is used to limit concurrent connections). Different error codes give an idea, where or why the error occurred. Sometimes it is needed to visit MediaWiki web page to see exact error code as the BenchMark tool does not download requested files.

There are two options for generating the requests. First one is to equally distribute the requests into one second, but with small load, it does not present real traffic characteristics as the arrival rate varies. The other option is to use randomly distributed requests that give better characteristics with the real world applications. Making ramp up test or measuring service time it would be better to use equally distributed requests as there is not so much fluctuation in the traffic and is easier to interpret the data (to see maximum throughput of one server). Under heavy load and using large amount of back-end servers, it does not differ, if the requests are equally distributed or randomly distributed for one second as the load balancer takes care of distributing these requests. If the load balancer is configured correctly it is not possible to overload back-end servers, while most of the requests have been done in the first part of the second, back-end servers should work well with such load as they have enough power to cope with rapid increase and decrease in the load curve. Randomly distributed traffic varies between seconds and do not guarantee the load defined by the user. Calculating random arrival times can also affect amount of load that program is able to generate and therefore using equally distributed load, the load generator is able to send more requests to the server.

3.10.4 Additional HTTP server

For each instance, an extra HTTP server is running on the port 8080 and home directory set to `/mnt` to be able to download experiment results from the cloud. Folder `/mnt` uses external space, it does not use image file system, and therefore has more room. By default, Amazon EC2 built in file system in the image has 3 to 10GB total virtual disk space depending which instances are used. When running large scale experiments, this room can quickly run out and therefore it has been decisioned to use `/mnt` to store temporarily the results. It is also possible to use `scp` program to copy the results from cloud using private key to access the instance. It has to make sure, that folder `/mnt` has set proper permissions, so the other programs can access and write there.

3.10.5 Distinguishing different servers by their role

MediaWiki framework is using different servers and roles to run the service. There are three options how to implement configuring the cloud. For example, MediaWiki configuration file needs correct IP addresses for memcachend and MySQL servers, these modifications have to be done for all the Apache back-end servers. In order to distinguish different servers, it is needed to assign tags or keep records of the running instances. One option is to use Amazon AWS console to tag running instances. These tags can be filtered out by using REST services for EC2 to query the requested servers. This option requires that application takes care of configuration of different servers, has private and certificate key copied to the cloud to access the EC2 services and could request list of running servers. Because of security reasons, it is not a good idea to store your private and certificate key on the virtual image and also using key hashes for running ec2-describe-instances as they might be grabbed by third persons or if the image is made public and user does not remember, that the key is stored there. This allows access to your key and thus others are able to run the instances and access cloud behalf of the key owner. When using AWS console to tag running instances, user must copy its keys to the cloud to make sure that he can use Amazon EC2 cloud commands to gather information about running instances and filter out correct roles for each instance.

Important caveat. When configuring IP addresses for MediaWiki configuration, it is important to use private IP address as it will reduce the latency and also no additional data transfer fees are taken as the traffic is inside the cloud. When using public IP address, the traffic is redirected to the outside of the cloud, thus increasing the latency and having to pay for the traffic. Short experiment generating 15000 (4 request per second for a hour) requests can use 1 GB traffic between load balancer and back-end servers. 1 GB traffic will cost 5 to 12 cents depending how much traffic has been already processed in the US East region.

Next option is to define roles based on private IP addresses and configuration is done solely relying on the predefined server list. This is used by CloudController. In this way, CloudController is aware of servers running in the cloud and which roles they have. Connecting with instances are done through SSH (Secure Shell) using private key to access them. With SSH, it is possible to execute commands and copy new files in to the server. When setting up the cloud for the experiment, one instance needs to have CloudController already copied or bundled with the image (make sure the private key is copied, not stored on the image). CloudController support two types of configuration, one is when user defines which servers to use and which roles to assign, other option would use CloudController automatic function to retrieve list of running servers from the cloud and assign the roles automatically and make the configuration. CloudController uses predefined commands in the text files to configure the cloud instances (copying, removing files, changing permissions, executing applications, etc), these instances can be changed by the user, depending what he or she wants to achieve. When necessary software is copied and services started, user needs to connect to worker instance and execute

BenchMark application to stress out the load balancer. Some parts of the deployment has been still left to be done by the user, as some specific task may needed to be done before starting full scale experiment (e.g. filling the cache).

Example of secure shell, how to access instance in the cloud and execute command.

```
$ ssh -o StrictHostKeyChecking=no -i private.key ubuntu@host 'ls'
```

SCP (Secure Copying Protocol) can be used to copy files from local machines to remote machine. It uses same parameters as Secure Shell. In addition, user has to define which file or files he wants to copy and where they are copied. It is also possible to change file name.

```
$ scp -o StrictHostKeyChecking=no -i private.key /home/ubuntu/local.txt
   ubuntu@host:/home/ubuntu/
```

If user is interested in making experiments with empty MySQL cache, he has to call out MySQL query `RESET QUERY CACHE`. It will clear the cache of recent queries. For clearing cache for memcached, one option is to restart the service, next option is to send command `flush_all` through TCP to memcached (see shell command at the bottom) or modifying MediaWiki configuration file for invalidating cache (if `$wgCacheEpoch` is not modified).

```
$ echo "flush_all" | /bin/netcat -q 2 127.0.0.1 11211
```

Chapter 4

Measuring Performance of The Cloud

This chapter gives overview of statistics gathering utilities, that can be used to measure performance of the servers running in the cloud. Monitoring server performance metrics gives good overview how the servers are working in the cloud and it will help to solve bottlenecks and other issues, that can occur while stress testing the system.

4.1 Using performance measuring tools under Ubuntu

Ubuntu was used as a primary operating system, that has many useful tools installed by default. These tools are capable of measuring performance of the machine and monitoring the current state. Ubuntu repository gives good set of applications and programs, that have built in functionalities to measure and monitor performance of the server. They can be easily installed using aptitude tool.

Ubuntu is storing some of the CPU, network, IO and memory metrics in the `/proc` virtual directory, that can be used to read about the current state of the server. It is possible to read server metrics, without having to install extra tools from repository.

4.1.1 Apache tool AB

Important caveat. Benchmarking tools are meant for generating high load for servers they are tested and, therefore, stressing out the system to get idea of its limits. These tools should not be used for public services and web pages, as it is more-or-less a tool for a distributed denial-of-service attack (DDoS). One might look carefully, which systems it starts to test and make sure that it does not affect other running services on that system. It is the best to do these benchmarks in closed systems.

To check performance of the cloud and measure performance metrics, we need to stress test the system. AB (Apache Benchmark) [18] is utility that comes along with Apache installation. AB is simple tool to measure throughput of the web server using different parameters. This tool supports multi-threaded requests, giving possibility to define how many requests should be generated and how many of the requests should run in parallel. Because of its robustness and simplicity, it does not support any flexibility defining custom load curve or having varying traffic. It is good to measure maximum throughput of the server or getting service time of the system. It should be tested on the same network (availability zone) or otherwise bandwidth limit and latency could affect the results. Using Ubuntu manual `man ab`, it gives following description for this tool:

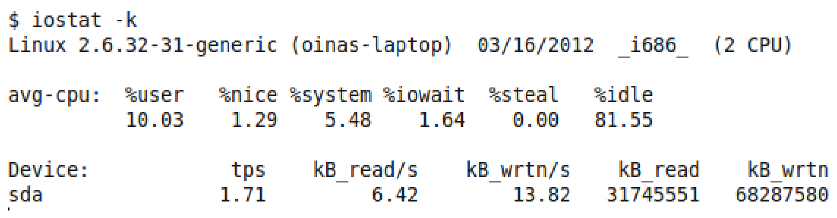
> *ab is a tool for benchmarking your Apache Hypertext Transfer Protocol (HTTP) server. It is designed to give you an impression of how your current Apache installation performs. This especially shows you how many requests per second your Apache installation is capable of serving.*

4.1.2 Ubuntu sysstat package

Sysstat package can be installed using command line tool aptitude: `sudo apt-get install sysstat`. This package installs various shell tools that can be used to collect system performance metrics. Using cron job, it is possible for certain intervals collect the data into file and monitor system performance. Cron only allows running its jobs with minimum in 1 minute intervals. Following is an example how to access cron and add job in to the list.

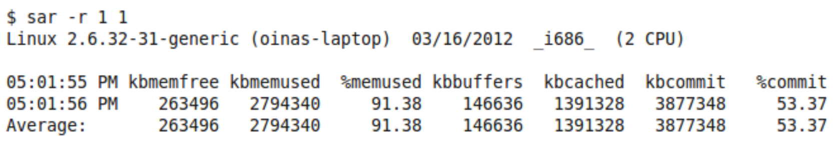
```
$ crontab -e
    # m h dom mon dow command
    */1 * * * * sar -u 1 1 >> /tmp/io_usage.txt
```

With `sysstat`, important tools for gathering statistics are `sar` and `iostat`. With `sar`, the user can get overview of computer performance as it logs all the necessary performance measurements in 10 minute intervals (when the



```
$ iostat -k
Linux 2.6.32-31-generic (oinas-laptop)  03/16/2012  _i686_  (2 CPU)

avg-cpu:  %user   %nice %system %iowait  %steal   %idle
          10.03    1.29    5.48    1.64    0.00   81.55

Device:            tps    kB_read/s    kB_wrtn/s    kB_read    kB_wrtn
sda               1.71         6.42        13.82   31745551   68287580
```

Figure 4.1: Standard output of `iosatat` command.



```
$ sar -r 1 1
Linux 2.6.32-31-generic (oinas-laptop)  03/16/2012  _i686_  (2 CPU)

05:01:55 PM kbmemfree kbmemused  %memused kbbuffers  kbcached  kbcommit   %commit
05:01:56 PM    263496   2794340     91.38    146636   1391328   3877348     53.37
Average:       263496   2794340     91.38    146636   1391328   3877348     53.37
```

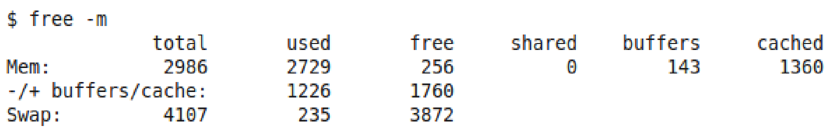
Figure 4.2: Standard output of `sar` using parameter to show memory usage.

user installs it first time, he/she must enable logging in the configuration file `/etc/sysstat/sysstat`). It is also possible to call out `sar` directly from command line to gain information about current state of the system. Command `sar 1 10` will output for each second ten times current average CPU usage. Command `iostat -k` is useful, when trying to measure information about disk operations and speeds. Parameter `-k` will force to show disk speed and kilobytes, by default, operations are used for the counters.

Using combined shell script of previous tools, it is possible to gather system performance statistics to monitor the usage and health for different servers. This gathering process can be built into web service, that can be accessed by other servers or use `ssh` to run the commands remotely.

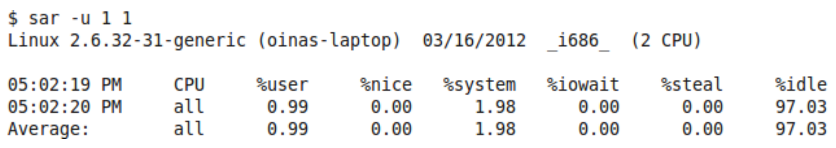
4.2 Using third party tools

Here is a list of some of the popular third party tools, that can be used for measuring server performance in the cloud.



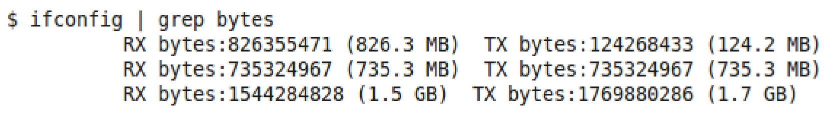
```
$ free -m
             total       used       free     shared    buffers     cached
Mem:          2986       2729        256          0        143       1360
-/+ buffers/cache:       1226       1760
Swap:         4107        235       3872
```

Figure 4.3: Standard output of `free` to get information about physical memory usage.



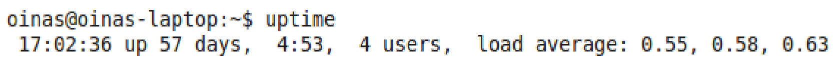
```
$ sar -u 1 1
Linux 2.6.32-31-generic (oinas-laptop)  03/16/2012  _i686_  (2 CPU)

05:02:19 PM     CPU     %user     %nice   %system   %iowait    %steal     %idle
05:02:20 PM     all      0.99      0.00      1.98      0.00      0.00     97.03
Average:        all      0.99      0.00      1.98      0.00      0.00     97.03
```

Figure 4.4: Standard output of `sar` using parameter to show CPU usage.



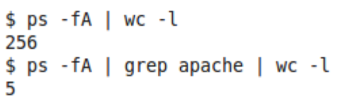
```
$ ifconfig | grep bytes
          RX bytes:826355471 (826.3 MB)  TX bytes:124268433 (124.2 MB)
          RX bytes:735324967 (735.3 MB)  TX bytes:735324967 (735.3 MB)
          RX bytes:1544284828 (1.5 GB)  TX bytes:1769880286 (1.7 GB)
```

Figure 4.5: Standard output of `ifconfig` to gather network interface traffic bandwidth.



```
oinas@oinas-laptop:~$ uptime
 17:02:36 up 57 days,  4:53,  4 users,  load average: 0.55, 0.58, 0.63
```

Figure 4.6: Standard output of `uptime` to see information how long the computer has been running and also get Linux load for 1, 5 and 15 minutes.



```
$ ps -fA | wc -l
256
$ ps -fA | grep apache | wc -l
5
```

Figure 4.7: Standard output of `ps` with various options to gather how many processes are working. Because command itself counts as a single process, these results should be subtracted by 1.

4.2.1 RRD database

Next tools collectd and cacti are using RRD (Round-Robin database) to store its data. Round-Robin database is a flat file structure. The RRD works as a circular array, that has no end or start point. Because of this structure, the RRD files are always in fixed size. It contains information how the file is structured and what variables it is containing at the header section of the file. The other part of file will contain actual data. These data are divided into the separate sections, supporting different time spans and density of data (i.e. measurements for each seconds, weeks and months). This will be useful when drawing the graphs for longer time span. RRDTool is used for drawing the graphs from RRD files. Using different resolutions and time spans, it will automatically select which data to use for plotting purpose. With bigger time spans, it will use measurements averaged from small units to larger units (from 10 second intervals to day, from day intervals to month etc). This will reduce time it takes to draw graphs, when trying to plot CPU usage for one year, but also the file size is fixed and smaller. Storing CPU usage for one year with 10 second interval would not be feasible. Using different resolutions and arrays in RRD file, it is possible to achieve data gathering for larger time spans than 1 week.

4.2.2 Collectd

Collectd is tool written in C++ and it can be used to measure different statistics in the server. It has built-in plug-ins that eases gathering statistics from different servers to one central server, it is possible to define one central collectd server, that will gather other collectd servers statistics or use multicast support. Main server for collectd will create for all the other servers different folders in the file system, where RRD files are created and data fetched from the client servers added. Multicast will be good to use in the cloud environment, where instance IP addresses are changing while conducting new experiments, thus reducing effort to configure IP for client servers.

By default, `collectd` gathers performance metrics at interval 10 seconds. This can be modified to reduce the traffic, if using server-client configuration. For local collection, it uses minimal amount of CPU to process requests as it have highly optimized code written in C++ language and does not impact the system performance, while measuring different metrics. With default configuration, one RRD file takes up to 0.1 megabytes in the file system and all the collected data with various RRD files can take up to 30 megabytes. Plugin `rrdtool` will write out gathered metrics into RRD files. This should be activated on collectd server. Using collectd as server-client configuration, it is needed to use built-in plug-in `network`, this will gather statistics from different servers into one server. One option would be to copy RRD files from the servers, but because of large size, it is not feasible to copy them at regular intervals, and `network` plug-in should be used when we want to draw the real time graphics.

Collectd, in comparison to Ganglia (one of the most popular tools for visualizing server performance in clusters), is not a visualization tool, it does not have built-in function to show the graphs of the server performances. RRDtool created by Tobias Oetiker [33] has functionality to draw the graphs. There are some examples added to the collectd package, that uses RRDtool to draw graphs using PHP or CGI, introducing possibilities that collectd with RRDtool can provide.

Amazon EC2 cloud gives great variety of servers to be used and the architecture might be different (32bit or 64bit). If the architecture stays the same for all servers, then there is no problem collecting and drawing the data using collectd. If some servers are using 64bit architecture (e.g. database server, needing more CPU power or memory) and some are using 32bit architecture, to merge these graphs or draw them from another machine, the files have to be converted. With `rrddump` it is possible to convert RRD files into XML (Extensible Markup Language). They are at least 10× bigger, so it would be wise to pack the results before downloading to local or another machine. Converting should be done in the same machine (otherwise rrdump is not able to read values from the file) or with the same architecture. If the files are converted, it is possible to download them to another machine, that uses different architecture. Program `rrdrestore` is used to convert XML files back to RRD format. This should be done, where the graph drawing logic is done. Otherwise trying to draw different architecture RRD graphs will throw an error.

4.2.3 Cacti

Cacti is an application bundle that consist of web application and system tools to gather various data and visualize it. It has administration interface, where user can set up new graphs or download new ones from Cacti homepage and export them. Public interface gives information about metrics gathered from servers. The system itself is using RRD files to store the data. It is possible to add as many servers into the list as needed and role for each server can be assigned. Role will reduce user input, as they contain templates, what graphs and what metrics to gather. There is no need to gather MySQL statistics for memcached server.

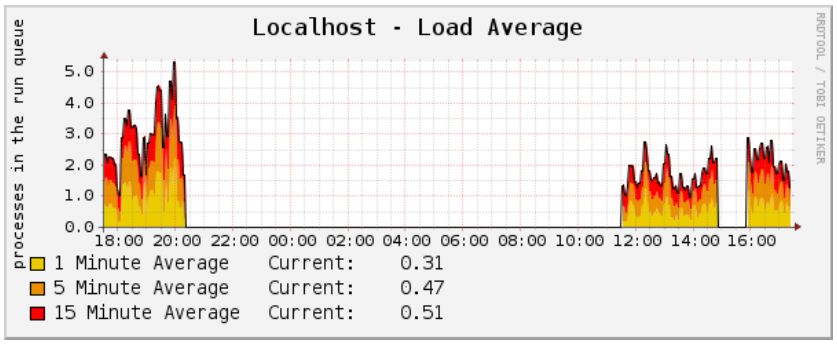


Figure 4.8: Showing Linux load gathered with Cacti and drawn by Cacti using RRDTool.

During writing this thesis, Cacti was tested to measure the performance of servers in the cloud, but unfortunately gathering the data was CPU intensive and there were several spikes in CPU usage. Cacti default gathering interval is set to 5 minutes, which should give enough precision for longer time and should hold network traffic as minimum as possible when transferring data across different servers. Even though the gathering is done in 5 minute intervals, it can generate spikes in CPU usage, while gathering parameters. There was at least 20% of CPU increase, when gathering various information (CPU, memory, network traffic, disk usage) from the server. For gathering, Cacti uses PHP script, that has added to cronjob list to fetch necessary data. While gathering metrics for 10 graphs, it took around 10 seconds for single server to process the data. Increasing the measuring points to 80, the time it takes to collect the statistics increased at least 60 to 100 seconds. Cacti gathering process was highly affecting CPU usage and affecting performance and speed of the service.

Cacti homepage suggested using of Spine (Cactid, written in C language) for polling the performance metrics instead of the default PHP. Spine supports multiple threads and should gather statistics much quicker way and with less overhead. Even though going over to Spine, there were still spikes in the performance graphs. The CPU usage decreased compared to the PHP poller, but still the spikes existed and affected service performance.

Cacti is not using caching when drawing graphs, meaning that when browsing between different web pages, the graphs have to be drawn again by the RRDtool impacting the server performance, where the graphs are collected. This means, that the service has to run in a separate server from the framework or it should be looked only at the end of the experiment.

Figure 4.9 shows clearly, that for every 5 minutes there are spikes, that

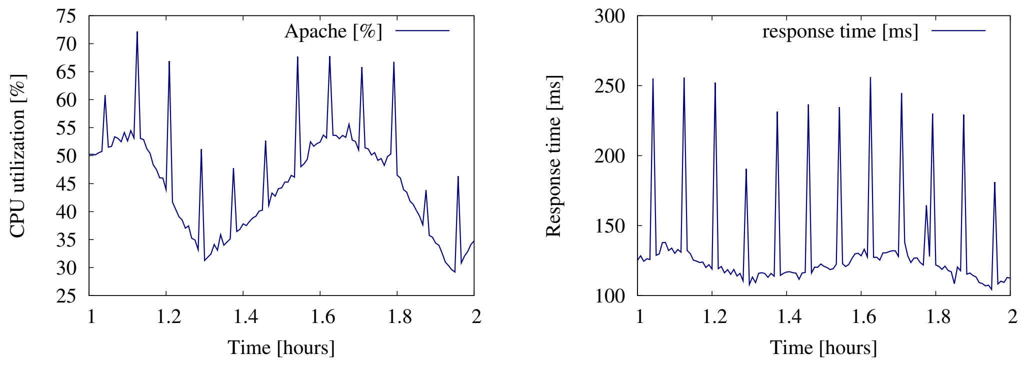


Figure 4.9: Fraction of the experiment running 10 servers, showing average CPU usage for Apache server and benchmark response time. For each 5 minutes, there is a spike, because cacti starts to measure metrics of the server.

are caused by Cacti, trying to measure the different performance metrics. The gathering process seemed to work at least 30 seconds and affected the service performance. For each spike, the response time increased at least 100ms. Figure 4.10 shows 1 and 5 minute load for Ubuntu and indicating, that too much resource is used while collecting data. This data collection did not resulted in lost traffic for that hour, but still can be considered as a performance issue.

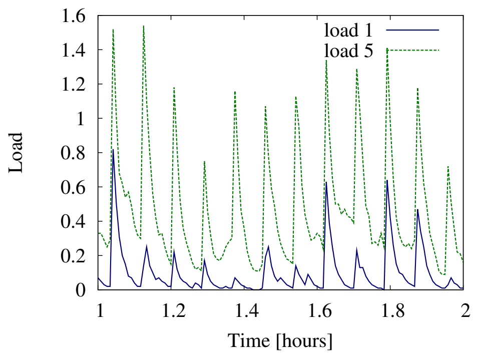


Figure 4.10: Showing spikes caused by Cacti, when collecting different performance metrics from the server.

4.3 Framework Performance Measuring Tools

While trying different tools, there was not a single good tool for the framework to collect performance metrics from the cloud. Idea was to have simple tool for giving current performance measurements through HTTP request. Collectd is easy to install, configure and get it running, but usage of RRD files to store the data complicated the use, as there is need to use RRDtool to fetch measured values and synchronize the results with response times collected by BenchMark. Synchronizing different files together was becoming complex when using dynamical allocation policies as the timestamps were different for each file for each server. The process was not straightforward and, therefore, collectd was not used in the framework. Collectd speed and performance was good and did not affect other services running in the server, even if it was using small interval and frequently collecting the data. Cacti used also RRD files to store the data. However, its performance was not satisfactory even while collecting data with 5 minute intervals.

One option is to use `sar` utility, gathering statistics with 10 minute intervals (or use cron job to collect data into one file). Tool `sar` is giving metrics for different parameters and the previously measured values can be retrieved again. But for this option, the time interval was too large (1 hour experiment means, that you have only 6 measuring points) and still some synchronization has to be done at the end of experiment to align the values with BenchMark results.

Solution for gathering the metrics was to have for each virtual machine a web service, that will run `sar` and `iostat` commands and parse the output to gather performance metrics. These metrics are kept in the memory, no output or files are written. Only way to access these stored metrics is to connect through network with the service and it will return information about gathered values. There is a central program running on nginx server, that is aware of running virtual machines in the cloud and connects for certain time periods with each server and retrieves values collected by the web service. This solution helped to collect the performance metrics of all servers into one location using timestamps. These collected values can be easily compared with timestamps generated by the BenchMark utility and no further synchronization has to be done at the end of the experiment to put the results together. ServerStatus program has been built, that will take care of collecting and parsing performance metrics from instance for each 15 seconds and listens incoming connection. Logic for collecting the statistics from ServerStatus is done in CloudController that maintains list of running servers and knows, which servers have to be visited to gather the statistics. For each gathering phase, CloudController shows in its output CPU, memory and network usage for each server, making it easy to keep an eye on the experiment.

One option was to run `sar` and `iostat` commands through ssh (Secure Shell), but under heavy load handshake, secure connection channel establishment and authentication will take significant amount of time and might fail to collect the data, making it unusable at higher loads.

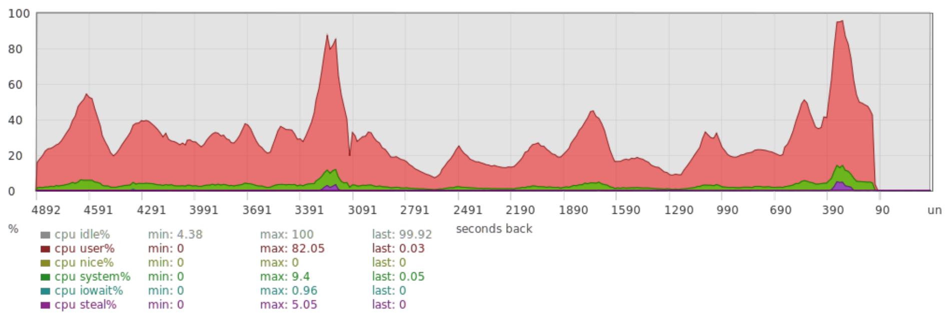


Figure 4.11: Modifying ServerStatus code and using JavaScript, it is possible to draw graph of the CPU usage that can be viewed when opening browser on port 5555.

Chapter 5

Preliminary Experiments and Evaluation of The Framework

This chapter overviews the experiments conducted to learn MediaWiki characteristics. This gives overview how well the framework scales and the maximum throughput of the servers. Experiments help to understand the system behaviour under high load, whether it would be possible to optimize the service or one of its components and what would be the most suitable configuration to run the services in the cloud. Putting servers under high load will help to determine maximum throughput of each server. This value can be used for scaling purpose to know, when it is the right time to add servers and when the servers should be removed from the cloud.

Conducting experiments will help to find possible bottlenecks in the system. As the service configuration is complex, there is no single failure point and without knowing the characteristics of application, there is no understanding where the slowdowns can occur. This chapter will give experiments for different configuration, where the configuration files of services are changed and different Amazon EC2 instances are used. If not told otherwise, experiments running the MediaWiki application have been done with `c1.medium` Amazon EC2 servers. Some of the servers can be downgraded, e.g. memcached, as during the experiments it is not possible to overload the memcached, but to have comparison between different services and how much resources they use, same server types are used for all the servers.

Experiments conducted here to measure service performance were only querying dynamical pages (PHP files). Static files (CSS, pictures, JavaScript etc) were left out, as the service time for these two requests is too different to load balance in the same manner between back-end servers. There is a need to define another `upstream` load balancing group for the static pages on different servers. Another issue with the static pages is, that the servers are capable of handling more requests, meaning, that there has to be deployed more load generators to reach the increased load and using more servers means, that cost of conducting the experiments will definitely increase.

5.1 Importance of experiments and stress testing

Experiments for stress testing the system have been a part of a QoS (Quality of Service) for a long time. It has important role to determine system flaws before it is going to the public and the system can be considered as a final product. It gives performance measures and throughput limits for each service, giving better vision in making capacity planning. Poor QoS can lead to frustrated customers, which tends to lead to the lost business opportunities [46]. QoS includes measuring time it takes to process single response and measure server overall throughputs. Increase in response times means, that the jobs are staying longer in the system, eventually leading into backlogging and making system unresponsive. Longer response times will keep off potential customers, as they will not want to waste their time for waiting when the new pages will be rendered. Without making stress testing, it might lead to following two outcomes [45]:

1. Service will fail at worst possible time, generating lots of frustrating customers or losing important transactions.

2. While bottlenecks happen in the system under heavy load, system administrator might not be aware, where or why they happen and finding the problem solution a lot harder.

5.2 Identifying most suitable configuration

Amazon gives great number of different server types, that can be started in the Amazon EC2 cloud. They vary in CPU speed, CPU threads, memory amount, IO/Network performance and architecture (32bit or 64bit). For example `m1.small` and `c1.medium` uses 32bit architecture and, therefore, when building image for the Amazon, it is not possible to run the same image on servers that are using 64bit machines. In order to achieve it, it is necessary to re-bundle the same image for 64bit machines.

This study was looking for the best server type to be used from Amazon EC2 cloud, when running MediaWiki application in the cloud. Experiments with instances `m1.small`, `m1.large` and `c1.medium` were conducted and performance measured. Servers `m1.small` and `c1.medium` shares the same amount of physical memory (1.7 GB), but the latter one has 5 EC2 units of CPU power with 2 virtual cores and `m1.small` has only 1 EC2 unit of CPU power with single core. 1 EC2 unit is approximately equal to Pentium XEON 1.2GHz processor [10]. Instance `m1.large` has 4 EC2 units of CPU power with 2 virtual cores and has more memory (7.5 GB) than previous instances. Previous tests in the SciCloud have shown that Apache servers running MediaWiki application is CPU bound and, therefore, most suitable configuration would be to use servers that have more CPU power. Reading characteristics for each server type, the most suitable server for running the MediaWiki should be `c1.medium`. It has been proven by Cloudstone paper [44] that when using High CPU instances (e.g. `c1.medium`), the performance is better compared with Standard instances, as the most of the applications are CPU bound, meaning that giving more CPU power will significantly increase the throughput. Liu and Wee [50] demonstrated that `m1.small` instance was more CPU-bound, whereas `c1.medium` instance was usually bounded by a network bandwidth (800 Mbps).

CPU speed in Amazon EC2 cloud depends on two factors: (i) how much resources virtualization gives for the instance and (ii) physical host CPU speed and memory bandwidth. Information about CPU given for the instance can be looked from Ubuntu under virtual file `/proc/cpuinfo`. Instance `c1.medium` was mostly using CPU E5410 @ 2.33GHz (6MB cache) in `us-east` region. There were occasionally servers working with lower frequency, e.g. E5506 @ 2.13GHz (4MB cache). Looking the results of experiments, there was clear difference that servers with smaller clock speed had maximum throughput smaller. While E5410 was capable of serving 30+ requests per second at maximum (with all the configuration optimizations), E5506 was little bit slower and was able to serve only 26+ requests per second. As the request complexity for different pages differs and performance loss of virtualization [57], it is not possible to give the exact results, but the estimates of average. Following command can be used to look CPU related information, when using Ubuntu operating system.

```
$ cat /proc/cpuinfo
```

5.2.1 Experiments with `m1.small`, `m1.large` and `c1.medium` instances

For selecting the best instance from the list, it would be good to perform the comparison between the different instance types. These comparisons have been performed based on the ratio between instance hour price and how many requests one instance is capable of serving as a maximum with one hour. Amazon EC2 pricing policy takes charges for full hour, making it reasonable smallest time unit to compare the results. Running instance for one hour the `m1.small` instance costs 0.085$, `c1.medium` instance costs 0.17$ and `m1.large` instance costs 0.34$ in Amazon us-east region [20].

Two types of experiments were conducted for different instances to measure (i) service time (making one request for each second) and (ii) maximum throughput (increasing load in certain time period). All the experiments performed here have necessary optimization already done, meaning that memcached is running and filled with pages and PHP is using XCache to improve speed of showing dynamical pages. Service time was calculated by taking average response time for one hour experiment, where for each second, one request was made. This would use minimal load for the server and theoretical value of maximum throughput can be calculated. Notice that two core machines are able to serve twice as much. If having service time 100 ms, it means that for one second one core machine is able to serve 10 requests and two core machines are able to serve 20 requests. Second experiment was conducted to measure the maximum throughput of the servers. This was achieved using incremental ramp up test, where for certain time period, load was increased by one unit (request per second). This will give enough information to calculate the price for single request, using estimated value how many requests one server is able to process in one hour and what is cost of server. Most suitable instance should have lowest cost for one request.

5.2.2 Experiments configuration

The configuration consisted of one Apache server, one nginx server, one memcached server, one MySQL server and one load generator instance. Instances `m1.small`, `c1.medium` or `m1.large` used as Apache depending of the purpose of the test. Other servers were selected `c1.medium` to make sure, that the other services are not slowing down the Apache instance and the comparison would be fair. It was important to run the experiments with the same requested instances (nginx, MySQL, memcached) and they were located in the same availability zone to reduce latency. Running with the same instances is vital as it might significantly affect end results. For example, if requesting

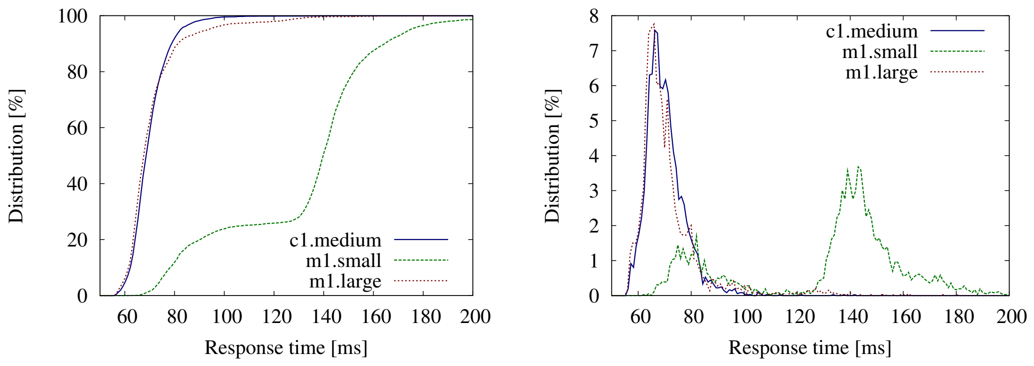


Figure 5.1: Instance `m1.small`, `c1.medium` and `m1.large` service time distribution. Left side shows service time cumulative distribution.

new servers and for some reasons memcached or MySQL server is working with lower speed (host system wear, different CPU), it will not give us comparable values as the service speed is affected by other parameters. Load generator used the same URI list for request to remove randomness of one experiment getting slightly smaller page requests than others, meaning that the pages are served faster and the pages were requested in the same order. Throughout the experiment, memcached hit ratio was over 97%, meaning that the most of the pages were fetched from the cache. This will represent stable system, as for typical web server running for a long time, the cache should be mostly filled and response times should not vary. It is easier to determine service time, as the variation is smaller. Purpose of the ramp up test was to check maximum throughput and how it works, when overloading the system. To achieve this, maximum connections were increased to 30 for nginx and Apache.

5.2.3 Measuring service time

First test was to measure the service time for each instance, to know how much time it takes to process one job by the server. Note that instances `c1.medium` and `m1.large` use two cores, meaning that two jobs can be processed concurrently. Service time was determined with the experiment, where for each second one request was generated. Average service time can be used also for different algorithms to calculate how many servers are going to be needed to successfully respond for every request entering into the system.

Table 5.1 shows the results for the service times between different instance types. Using mean or median value, it is possible to calculate theoretical maximum throughput of the server. This value can be validated with the next

Percentile	`m1.small` (ms)	`c1.medium` (ms)	`m1.large` (ms)
min	62	57	56
25%	108	66	65
50%	140	70	68
75%	149	74	74
90%	165	80	82
95%	175	83	94
99%	210	96	131
max	459	278	367

Table 5.1: Service time distribution for `m1.small` (mean 132 ms), `c1.medium` (mean 71 ms) and `m1.large` (mean 71 ms).

experiment, where maximum throughput is measured. Instance `m1.small` is able to serve half of the requests faster than 140 ms. Using formula 5.1, theoretically `m1.small` is capable of handling 7 requests per second.

$$\mu = \frac{1}{service\_time} \times CPU_{cores} \tag{5.1}$$

With high CPU instance (`c1.medium`), half of the requests with in a 70 ms, meaning that for one second, one core can process 14.29 requests per second and two cores are capable of serving 28 requests per second. In the same way, instance `m1.large` is capable of handling half of the requests with in a 68 ms and because of two cores, should be capable of serving 29 requests per second.

Instance `m1.large` response time distribution is similar to `c1.medium`. This can be related, that they both use 2 cores to process the request. The difference comes with maximum throughput, where `m1.large` has 4 EC2 units of CPU power (`c1.medium` has 5 EC2 units), meaning that with the same physical core, it has fewer CPU cycles to use. Next experiment showed that at least 26% of the CPU went to the steal, but for `c1.medium`, the number was around 6%.

5.2.4 Measuring maximum throughput

Second experiment was to test overall throughput of the system, making ramp up test, where load was increased in every two minutes by 2 requests per second. All the experiments consist also network latency, but it is relatively small and mostly it was less than 1 millisecond and, therefore, it is not consider as an important factor to be interested, while comparing different server types.

In this experiment, the load was increasing from 1 request per second to 30 requests per second with in 1 hour. Load generator requested same pages in order to have fair comparison. Theoretical throughputs have been calculated from the previous experiments.

Single Apache server, running `c1.medium`, had the largest throughput compared to the others servers. It was capable of serving 28 (see figure 5.3) requests per second. Response times during the ramp up experiment were stable and did not fluctuate, some of the requests were larger, meaning that for some time periods there were more jobs in the system than the server was capable of handling, this is why there were several pikes in the response time. Another problem with this experiment was, that when starting experiment with Apache from the cold start (instance was just booted), there weren't enough workers started by the Apache. Center of the experiment, the response time was slightly increased as the workers were not able to serve the incoming requests. When Apache was spawning new workers, the response time dropped and everything worked again as intended. Observing this behaviour through experiment, the experiment was restarted. Before re-running, Apache server was spammed with large amount of request, to make sure that there were enough workers by Apache to process the requests to remove the strange waves and increases in the response times.

Even though `c1.medium` and `m1.large` use the same amount of cores and the service time experiment shows the similar results, the latter instance has been capable of only serving 18 requests per second at maximum (see figure 5.4). It was observed, that for this instance there was a large CPU steal usage (over 25%), meaning that the Apache server was not capable of fully utilizing CPU. The situation with `m1.small` was even worse, as more CPU usage (over 55%) was blocked by virtualization and the system was capable of serving only 6 requests per second (see figure 5.2).

5.2.5 Summary of different instance types performance

The results of different comparisons as well as the selection of the best option is presented in table 5.2. We were seeking instance that is capable of serving one request at the smallest possible price maximizing throughput over the cost.

In the ramp up experiment, the load was increased for every 2 minutes and statistics were gather for every 30 seconds, thus having 4 measured values for each load. From the figures 5.2, 5.3 and 5.4, the response time average is taken for every minute and showing minimum and maximum values for

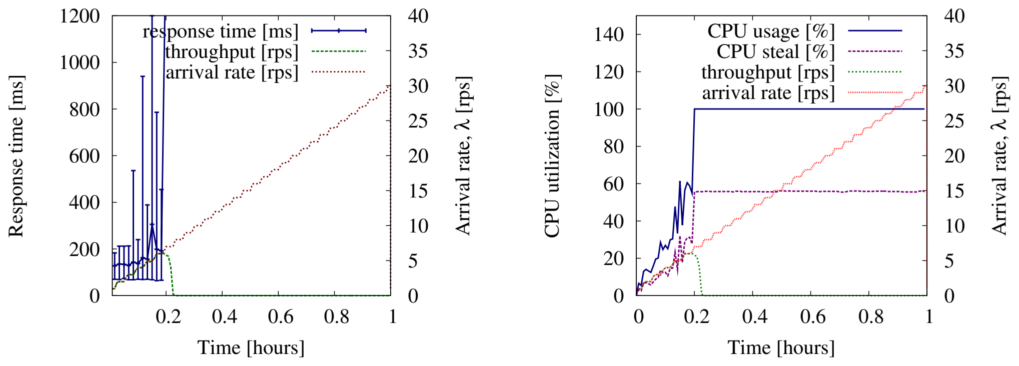


Figure 5.2: Instance `m1.small` response time and CPU usage for ramp up test. In the case of this instance type server was able for 6 requests at maximum and then the service became unresponsive. Only 40% of the CPU cycle was enabled for user. Most of the CPU cycle, over 55%, was going to steal, meaning that virtualization was dividing available CPU usage to other instances on the same physical machine.

that minute. These graphs show the performance for each instance to get understanding, how they work under the high load. Overloading `m1.large` server shows clearly, that when exceeding the threshold of the maximum throughput, the service degrades and actual throughput decreases. Using configuration for all the instances to allow 30 maximum clients, there were more jobs coming into system and overloading the system. This will explain why for `m1.large` the throughput will decrease, while the arrival rate will increase. This can be easily fixed reducing the maximum number of clients in Apache configuration or using connection limit for load balancer.

It was interesting to see that for `m1.large` instance, serving one request costs more money than for `m1.small`. Customers of Amazon EC2 pay extra money for the extra memory (7.5 GB of memory) while using `m1.large` instance. However, this amount of memory is not necessary for running MediaWiki on Apache, as it will be never fully used. This is paying more money for the resources that actually will not be used. This instance would be better to use for memcached or MySQL as there is more room in the memory to cache the requests and objects. The best option for running MediaWiki application with the framework was to use `c1.medium` instance (1.7 GB of memory). Memory consumption was small and it was cheaper to use compared to the other instance types, as one request was significantly less expensive.

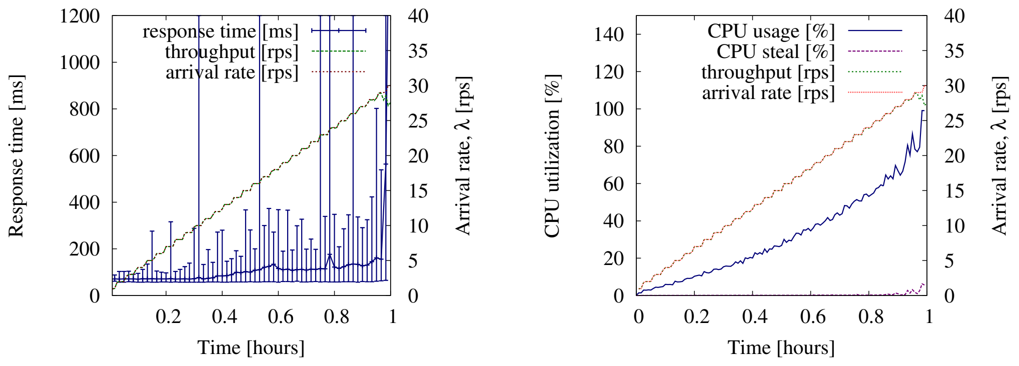


Figure 5.3: Instance `c1.medium` response time and CPU usage for ramp up test. Server was able for 28 requests at maximum and at the peak the average response time increased to 3 seconds and server started to become oversaturated.

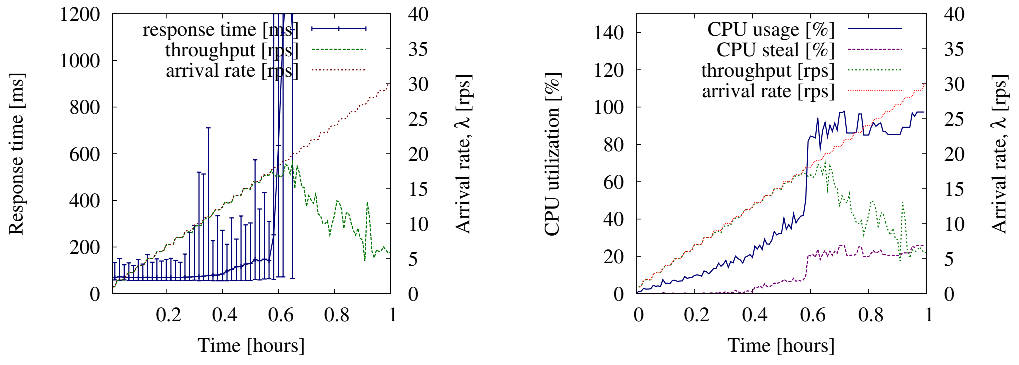


Figure 5.4: Instance `m1.large` response time and CPU usage for ramp up test. Server was able for 18 requests at maximum and at the peak the average response time increased to 5 seconds and server was heavily overloaded.

measurement	m1.small	m1.large	c1.medium
min. response time	62 ms	56 ms	57 ms
avg. response time	132 ms	71 ms	71 ms
max. response time	459 ms	367 ms	278 ms
max. throughput	6 rps	18 rps	28 rps
max. CPU steal	56.04 %	25.14 %	6.23 %
cost per hour	0.085 $	0.34 $	0.17 $
price per request	3.935×10^{-6} $	5.247×10^{-6} $	1.687×10^{-6} $

Table 5.2: Comparison between different instance types, showing their capabilities, cost and how much single request will cost (this is calculated using estimated value how many request one instance is capable of serving in one hour, dividing the total request count with the instance price for one hour). Response time statistics are taken from service time experiment.

5.2.6 CPU cycle waiting behind Xen hypervisor

Figure 5.1 shows how response time distribution differs for `m1.small` instance, having one portion of the requests served faster than the other portion. This can be caused by, because some of pages are relatively short and do not need excessive CPU usage to process the content, but for others it is necessary to process more objects through, meaning that operating system uses more CPU while processing these requests. But because of virtualization and having limited amount of CPU power given, the Xen hypervisor is blocking CPU usage for virtual image, meaning that for processing request the application has to wait until CPU usage is allowed again, making the response time longer. Therefore, we selected out randomly two requests to compare the page complexity and time it takes to serve it. First selected page (*Strontianite*) was relatively short and second page (*Summary_of_Decameron_tales*) was larger and contained more items and links. Processing larger page was using more CPU and because `m1.small` instance is allowed to use 44% of the CPU for one core in the virtual environment, the CPU usage is being blocked away by virtualization, increasing response time for larger pages at least 2×.

There is a reference [49], that instances, which have large CPU steal usage, is not suitable for real time application (like web server) where under the heavy load all the CPU power is needed. It is better to run web applications at a constant speed without need to worry about virtualization taking away CPU cycles. Results for `m1.small` are pretty bad, as it is really not capable of serving adequate number of requests. At the peak it was able to serve 6 requests per second that was too few taking into account today's traffic and bandwidths. It will be easy to overload the server with excessive incoming requests, making it harder for dynamically adding new servers depending of the load, as there is a need to run extra spare servers or add servers in pairs (while the load increases, at least 2 servers should be added).

During the experiments, it was observed, that when Apache was spawning new processes, there were some peaks in CPU usage affecting the performance of the server. If `MaxClients` was set to high and server reaches maximum throughput, it was no longer capable of serving excessive request, but would generate many new processes. This moment, the maximum throughput decreases, as more CPU will be used by Apache to manage the PHP workers. This can be also noticed while looking Apache process count and CPU usage. There seems to be a common trend.

5.2.7 Measuring CPU utilization with ramp up

To see how CPU utilization changes over time for different servers, another ramp up test with `c1.medium` instances was used (see figure 5.5). Next experiment shows results for ramp up stress test, where for 1 hour, load from 1 request per second to 150 requests per second was generated in the increasing manner. The configuration consisted of 5 Apache, 1 nginx, 1 MySQL and 1 memcached servers. Using Fair module, at the beginning of the experiment, most of the requests were going to the first two servers, as the other three were staying in idle state. At 25 requests per second, all the 5 servers were getting requests. This behaviour happened, because with the small loads, fewer servers were capable of serving request faster than they were coming in, meaning that the server became again into the idle state and, therefore, the next request is served by the same server. This configuration was using Fair module with nginx using least round-robin algorithm to distribute the load.

At the middle of the experiment, the jobs were evenly distributed and the CPU utilization stayed the same between Apache servers. Further increase in traffic will disrupt the evenness, this shows that some of the instances are faster and capable of serving request with fewer CPU utilization. Through different experiments and tests conducted by writing this thesis, there were some server running as `c1.medium` acting unexpectedly and capable of serving only 20 requests per second with 100% of CPU utilization, others have been able to serve sometimes almost 34 requests per second. For servers performing only 20 requests per second, reboot, service restarts and other things

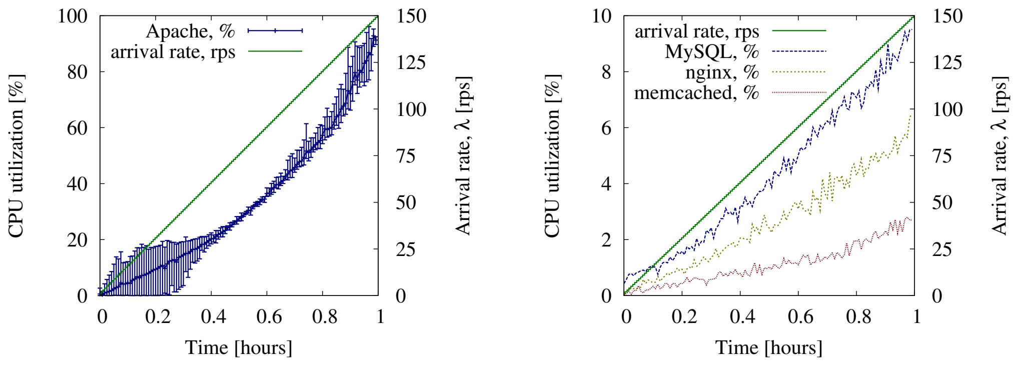


Figure 5.5: Ramp up experiment using `c1.medium` instances with 5 Apache servers showing CPU utilization for different servers, while generating load between 1 request per second to 150 requests per second.

have been tried to do, but none of them have worked, meaning that virtualization does not work well [57] or the physical host has been overloaded.

Even though the portion of servers working not as expected (`c1.medium` instance serving only 20 requests per second) has been relatively low, but still it needs some attention, while deploying and running them in a real life application. One option is to measure performance for each server while they become available, indicating any performance issues before they are added to the load balancer server list. If there is any performance issue, this can be easily killed and a new one requested. Other option is to see, if load balancer supports weighted traffic distribution, then it is possible to set small weights for low performance servers compared to the other servers.

5.2.8 Determining concurrent connections

Load balancer nginx used in the experiments has been compiled and built using Fair module that allows determining maximum amount of concurrent connections each back end servers allowed to have. This helps to prevent overloading Apache servers, as load balancer is dropping excessive connections. It uses least round-robin algorithm to distribute traffic between back-end servers. This means, that nginx tries to find from server pool the most idle server and pass the request there. In this way, nginx is populating jobs evenly and under heavy load, load balancer is not overloading the servers with excessive connections. With low arrival rate and using more servers than needed, the requests are not equally divided, as the first servers are only loaded. The reason is simple, because with low load, the first servers are capable of serving the request faster than they enter into the system making them again for nginx as a idle server, where next request will be sent.

Determining difference between default round-robin and least round-robin

To see difference between default and least round-robin distribution model, a small experiment with 5 back-end servers has been conducted to have picture, how requests are distributed with varying traffic for one hour, using ramp up test. Ramp up test gives large variation in the traffic, that helps to explain, what happens when the load is small, what happens if the servers are moderately loaded and what happens under the high load. Highest arrival rate is selected by the maximum throughput the servers should be capable of serving, using $5 \times 30 = 150$ requests per second, as this should be the maximum possible throughput in perfect situation.

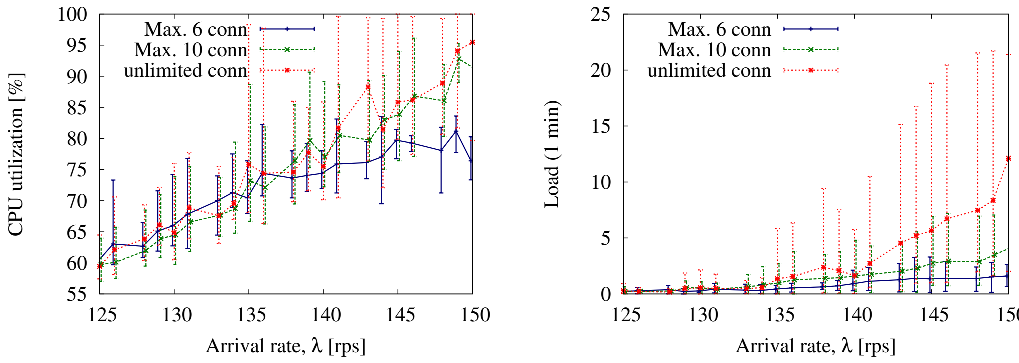


Figure 5.6: Ramp up experiment showing difference between two Fair configuration (using 6 or 10 concurrent connections per Apache server) and without Fair using `c1.medium` instances with 5 Apache servers. Left side graph shows CPU usage for Apache servers and right side shows 1 minute load.

Figure 5.6 shows the difference between using Fair module for nginx with different configurations and also without Fair activated on the nginx side. While reducing concurrent connections to 6, the experiment was only able to stress the back-end Apache servers to use 75% of CPU cycle, also the 1 minute load stayed low and reasonable, indicating that there were no excessive jobs waiting for the execution. While not using Fair module, the system gets easily oversaturated and excessive jobs will enter into the system. There is a large variation for different Apache servers in terms of CPU usage and 1 minute load, showing that the requests are not evenly distributed, congesting some of the servers, but the others are still able to serve the content, but the traffic is routed in wrong way and therefore decreasing the potential throughput. With Fair, the traffic is tried to be routed to the servers, which have least concurrent connections, making sure that busy servers do not get excessive requests and trying to serve by least busy servers. Using cached data, the difference between least round-robin and default round-robin was not large. Using empty memcached, the distribution was much better with the least round-robin algorithm.

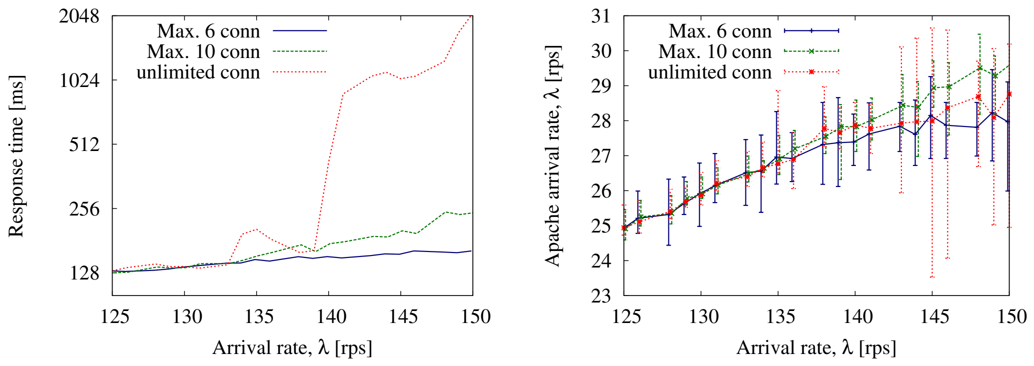


Figure 5.7: Ramp up experiment showing difference between two Fair configurations (using 6 or 10 concurrent connections per Apache server) and without Fair using `c1.medium` instances with 5 Apache servers. Left side graph shows average response times for requests and right side shows how many jobs are being processed by Apache servers.

Figure 5.7 shows difference in response times and how the requests have been served between different Apache servers. At the higher loads, using default round-robin and not dropping the excessive connections, some of the servers have to serve only 24 requests per second, while others have to serve over 30 requests per second pushing some of the servers to the limits. It expresses also in the response times, seeing at least 2 seconds average response time (for servers and clients, socket timeout was limited to 10 seconds, actually the timeout is much higher in real systems).

It is logical to see that when using fewer connection, the response time is also smaller as the load balancer is not saturating the service. Using 6 concurrent connections, the average response time stayed around 160 ms. Increasing concurrent connections to 10, it was 245 ms, one and a half times larger. It is not possible to say, that 6 concurrent connections is the best option, as it depends on the configuration and also in the management decisions. At the peak traffic, 150 requests per second, back-end servers were able to serve 147 requests per second using 10 concurrent connections for each back-end server, while with 6, the throughput was 139 requests per second. Losing 8 requests per second, improves response time 85 ms.

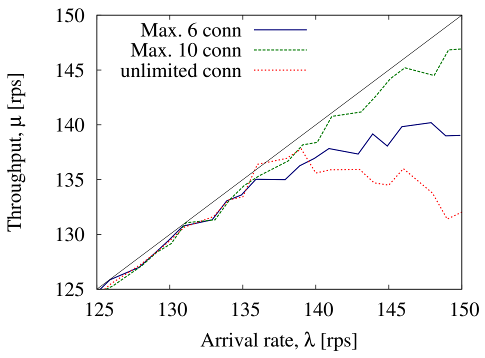


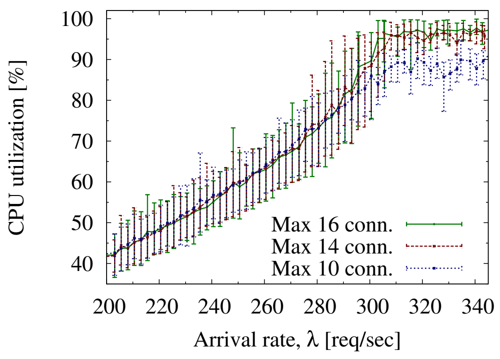
Figure 5.8: Ramp up experiment showing difference between two Fair configurations (using 6 or 10 concurrent connections per Apache server) and without Fair using `c1.medium` instances with 5 Apache servers. This shows amount of successful requests going through the system.

Experiments

Several experiments were conducted to see how concurrent connections can affect overall system performance by measuring the overall response time, CPU utilization and traffic lost. From the experiment it looks, that 10 concurrent connections for `c1.medium` (that is capable of serving 30 requests per second) is the limit where the system is already saturated. Increasing this number will increase response time drastically and affect service badly.

The results showed that the perfect configuration would have to decrease the concurrent connections to 6 as the system is more stable and the average response time does not vary so much. Using fewer connections the service maximum throughput decreases, as fewer jobs are allowed into the system. Using only 2 concurrent connections per server and 5 Apache servers, Apache servers utilize on average 50% CPU, showing clearly that using small amount of concurrent connections, large number of jobs will be rejected, decreasing significantly the maximum throughput of service.

Figure 5.9 shows that even if the throughput stays almost the same, the jobs are executed with much less CPU utilization. What happens is that using larger amount of concurrent connections, extra jobs are entering in to



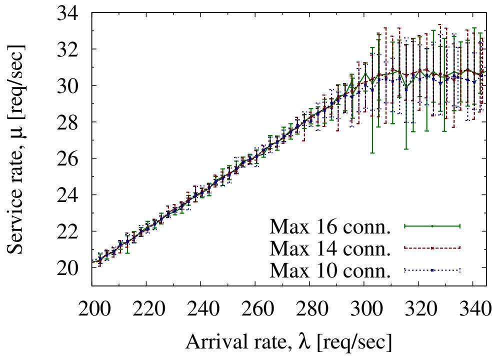
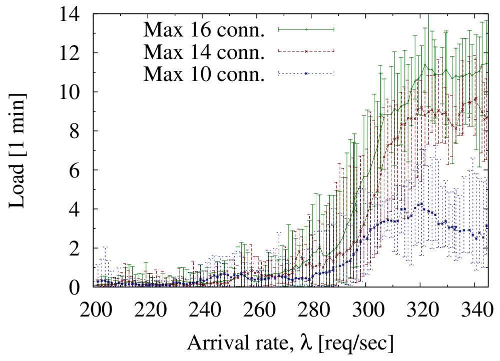


Figure 5.9: Concurrent connections experiment using `c1.medium` with 10 Apache servers showing CPU utilization and servers throughput. Service rate shows, how many requests one Apache server is capable of serving.

the system and congesting the server, while using smaller number, those extra requests are dropped to maintain good throughput with better response time and smaller CPU utilization.



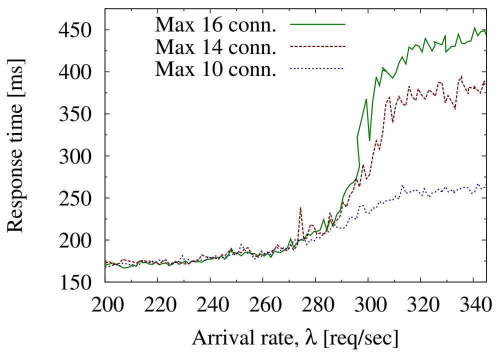


Figure 5.10: Concurrent connections experiment using `c1.medium` with 10 Apache servers showing Apache servers 1 minute load and response time.

Figure 5.10 shows that when using fewer concurrent connections, it is possible to reduce response time as fewer jobs enter into the system. When using 10 concurrent connections the load balancer slightly overloads the back-end servers. The response time has increased and the serves 1 minute average load is over 2, meaning that there are more jobs in the system than the server is capable of handling (job queue is larger than the server capable of processing them). Instance `c1.medium` is having two cores, meaning that one minute load should stay below 2.

Observations and findings

Even though, using fewer concurrent connections seems to be the better choice, the latter experiments use still 10 concurrent connections. This has been selected, because it allows much higher throughput and shows how service under heavy load actually works. It is getting better with the larger amount of back-end servers (the saturation point is shifted further away, making response time faster). Using fewer concurrent connections does not work well, when having small amount of back-end servers running as the nginx is not capable of distributing incoming request to the free sockets (they are filled) and they are dropped. Experiments shows, that when using 5 back-end Apache servers and 6 concurrent connections, 3 requests per second per server have been lost on average and average CPU usage is 75%, meaning that there is some unused room for serving the requests. Selection of the right number how many concurrent connections to use depends, how the services is going to be distributed for the end users and are the clients willing to wait more time or lose interest quickly and quit using the service. The intention of thesis was to run servers in optimal way, meaning that amount of requests dropped had to be minimum and meantime, to hold load high for all the servers. With 20 Apache servers and using 10 concurrent connections, it was possible for some cases to serve 600 requests at the peak (30 requests per second per server) and CPU utilization was varying for Apache instances from 90% to 98%.

5.3 Experiment to establish limits of the service for current configuration

This thesis set the border to run at maximum 20 back-end Apache servers, as it is capable of handling large scale traffic and overall experiments cost is not high. This experiment should give feedback if the MediaWiki is configured correctly and is capable of handling the large amount of traffic. This will also show how many MySQL, memcached or other instances are needed to support traffic for 20 back-end Apache servers. This amount of traffic is generated by using three load generators, it also reduces CPU usage for the load generators resulting in precise measured response times as they are not affected by excessive CPU usage. This test tries to increase utilization for other services in the MediaWiki architecture in order to establish the limits for other servers.

Instance	Sending	Receiving	Utilization
1 nginx	313.17	319.31	79%
1 memcached	37.57	3.12	5%
1 MySQL	121.32	25.83	18%
1 Worker	3.34	102.76	13%
1 Apache	15.78	9.46	3%

Table 5.3: Showing network traffic bandwidth for different servers using `c1.medium` instance, while generating load of 600 requests per second (all the values are shown as megabits per second).

5.3.1 Deadlock triggered by MySQL

First re-runs of this experiment were failures, as the MySQL server triggered deadlock error. This occurred every time and the deadlock error seems to happen when generating HTTP request to nginx load balancer at rate of 400 requests per second. Following error was thrown by MediaWiki application:

```
    1213:  Deadlock found when trying to get lock;
try restarting transaction
```

5.3.2 Fixing deadlock

This problem took some time and struggling to get rid of it. Finding glues from various mailing lists and going through MediaWiki manual there was an option disabling the page counter. Page counter is used to keep track of how many visits have been done for one page for statistics purpose. While generating large amount of requests, MySQL database was not able to cope with all the update queries sended by MediaWiki application and triggered deadlock error while trying to update values in the same time. Its implementation for counting the pages has been badly implemented and should be reconsidered. It should probably use memcahed for temporary count in the case of some visits and push them later into the database. Luckily, this option can be easily turned off from `LocalSettings.php` configuration file, modifying variable `$wgDisableCounter` to `false`. This solved the problem and deadlock did not occurred anymore.

Instance	min	avg	max
1 nginx	25.46%	27.3%	29.18%
1 memcached	8.54%	9.48%	11.65%
1 MySQL	54.28%	57.9%	64.32%
20 Apache	92.2%	94.06%	99.36%

Table 5.4: Showing average CPU utilization for different servers (`c1.medium`), while generating load of 600 requests per second for 5 minutes.

5.3.3 Network utilization

Table 5.3 shows network usage for different server types under the high load. The values presented here are taken maximum for 30 second time span to see the maximum network usage for each instance to give idea of the limitations for running the service on the cloud with following configuration. Only instance, that can be congested with network traffic is nginx as it is sending and receiving pages from back-end servers to clients doubling the amount of traffic. Other studies have been shown, that 800 Mbps is the limit of traffic for single instance, meaning that nginx network is highly utilized, but still has some room for serving 600 requests per second without a problem. The limit for this configuration was hit with 800 requests per second and that was expected result.

5.3.4 CPU utilization

Framework built for this thesis measured performance and statistics metrics from the cloud during the experiments. Table 5.4 shows instance CPU utilization using 20 Apache servers, while generating 600 requests per second to see how the jobs entering into the system are divided by different resources. Please note, that these values were collected, while using only one memcached and one MySQL server. Apache servers had very large CPU utilization, but still were able to serve all the 600 incoming requests (some of the requests were blocked by nginx, 3 rps). Some of the servers were slower, because they were using different CPU, where clock speed was slower and cache size was smaller of the CPU. The difference can be looked from the table 5.4, where faster servers were using only 92.2% of the CPU and slower servers were overloaded and using 99.36% of the CPU.

Figure 5.11 shows CPU usage and variation with different arrival rates, when using 20 back-end Apache servers. This experiment used ramp up test with increasing load for one hour. Fair module distributed requests to only small portion of the servers with low arrival rate, thus the figure showing larger variation between Apache CPU usage between the servers for the first part of the experiment.

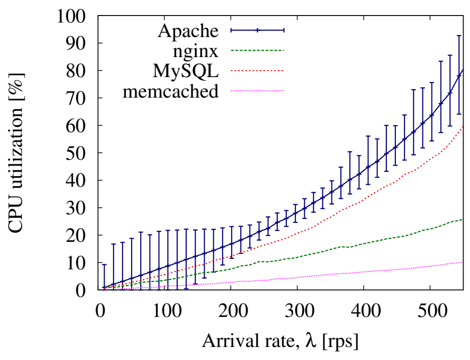


Figure 5.11: Ramp up experiment with 20 `c1.medium` Apache servers and generating load up to 550 requests per second, showing CPU usage for different servers. Nginx used Fair module to distribute the traffic (used the least round-robin), therefore there were large CPU usage difference between various Apache servers at the beginning of the experiment.

5.3.5 Replicating MySQL and memcached instance

In this thesis, different configuration options were looked, whether it would be possible to decrease the response time and to increase the overall throughput. Adding another memcached and/or MySQL instance should theoretically improve the service speed, as the MediaWiki database and cache requests should take smaller time as they are processed by more servers, resulting in lower response times. When adding a new server, it reduced the jobs in one system by half and, therefore, meaning that each job can be served faster and probably reducing response time. Memcached CPU utilization for different experiments was small, when nginx was getting 600 requests per second traffic, it was mostly under 10% and adding a new memcached instance did not really decreased response time as the amount of the jobs was already relatively low in the memcached.

On the other hand, MySQL had quite a big CPU utilization, staying around 60%. Adding replicated MySQL server, it was possible to reduce MySQL CPU utilization to 20% for both MySQL instances. Even tough MySQL load was significantly decreased, the response time dropped only

25 ms (from 492 milliseconds to 467 milliseconds, note that the Apaches were already little bit saturated, as the average response time was relatively large). Because the change in response time was small, this would indicate that the Apache servers were not really waiting behind MySQL database nor memcached server to serve the content and were more CPU bound by its processing speed, how fast they could parse the PHP file and rendered the page.

5.3.6 Finding out network bandwidth upper limit

Previous experiments have shown, that 600 requests per second with 20 servers is not a problem to be served. This rate was possible by tuning some parameters in MediaWiki application configuration and changing some variables for the operating system to allow more file descriptors to be opened. With default configuration (out of the box), the limits were hit quickly, as the first problem was with MySQL database, as it was becoming oversaturated with update requests. Other problem was that memcached hit ratio was low and pages were not stored in the list. Tuning and changing parameters from various places, the maximum throughput for one server in Amazon EC2 was achieved. Network usage demonstrated that nginx had already very large network utilization, meaning that when adding new Apache servers and increasing the load, it was possible to hit the limit. Next experiment was ramp up test, generating load from 600 requests per second to 900 requests per second for one hour to see the limits of other services or the nginx instance. 10 extra back-end servers were added, ending up with 30 Apache servers, to make sure, that the connections were not dropped because Apache was overloaded.

When generating 700 requests per second, experiment showed, that MySQL was largely overloaded (while using 1 MySQL instance) and jobs in the system were taking larger amount of time to be processed. As the jobs were staying longer in the system, nginx quickly ran out of 10 concurrent connection limit for each server, as the Apaches were not able to serve the content in reasonable time. Looking CPU statistics gathered from the back-end servers showed, that Apache was only moderately utilized, using only 50% of the CPU, this can mean that the Apache servers were waiting behind other services. Even adding new Apache servers did not help to solve this problem, as the CPU usage was still low and nginx was dropping incoming requests. MySQL was the bottleneck, as the Apache servers had to wait behind database queries. MySQL was using 95% of CPU and 1 minute load was over 10, while generating 700 requests per second.

Network bandwidth limit was still not reached for nginx. To improve the service for much higher arrival rate, there is need to upgrade MySQL server with larger instance or replicate the server, so the read operations are divided by the two database servers. Looking results of the experiments showed, that nginx and memcached servers CPU usage was still low, indicating that they were not CPU bound. Using command `iftop` (tool to see network transmission speed) while the generated load was around 700 requests per second, it was showing as a peak network bandwidth (incoming and outgoing requests) 763Mbit/second. This is close to the maximum value given for `c1.medium` instance 800Mbit/second, meaning that nginx was already close to hit the limit.

Next experiment was trying to push the limits and used 2 MySQL servers to reduce query time in the MySQL database as the CPU usage was reduced and the servers had to cope with incoming requests, in same time reducing Apache response times. Experiment showed that the bandwidth upper limit is between 700 and 800 requests per second, depending which requests were sent and how much traffic other Amazon EC2 customers instances were using. While hitting the bandwidth limit, Apache average CPU (using 30 servers) usage was around 64%, for both MySQL server it was around 30%, nginx was using 57% of the CPU and memcached 13%. Increasing amount of concurrent connections did not improve service throughput, as it seemed like the connections were left opened and nginx lacked the network capacity to pass the requests and responses. At peak the traffic rate was at least 792Mbit/second traffic (combined incoming and outgoing traffic) for the nginx instance and looking other parameters (CPU usage was normal for all the servers), it could be concluded that nginx with `c1.medium` was not able to go over 700 requests per second. If there is need to serve more requests, additional load balancer or server with larger bandwidth to cope with the traffic is needed. With extra nginx it is possible to use DNS entries to distribute the load between different nginx servers or use Amazon Elastic Cloud to distribute the traffic between those two instances (look figure 5.12 for one of the possible solutions).

5.4 Validating difference between different availability zones in the same region

Currently region `us-east-1` gives 6 different availability zones to run the instances. With in the same availability zone, the network latency should be minimal. Next experiment is to identify problems that might happen, when servers are scattered in different availability zones. One might think,

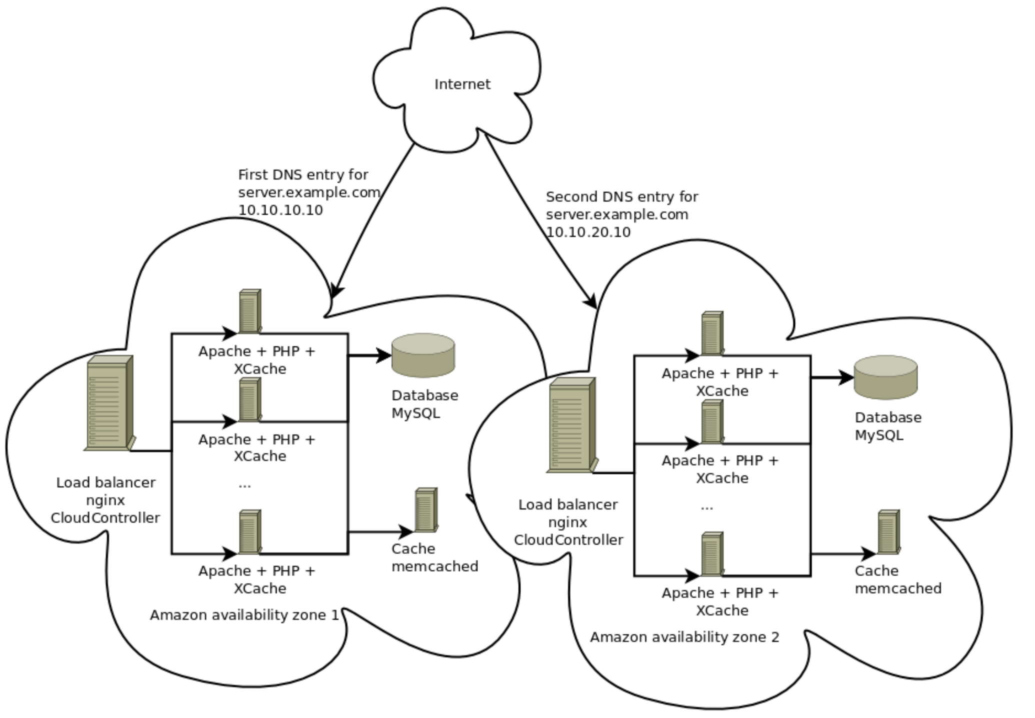

Internet
First DNS entry for server.example.com 10.10.10.10
Second DNS entry for server.example.com 10.10.20.10
Apache + PHP + XCache
Apache + PHP + XCache
...
Apache + PHP + XCache
Database MySQL
Cache memcached
Load balancer nginx CloudController
Amazon availability zone 1
Apache + PHP + XCache
Apache + PHP + XCache
...
Apache + PHP + XCache
Database MySQL
Cache memcached
Load balancer nginx CloudController
Amazon availability zone 2

Figure 5.12: How to implement two nginx load balancers to increase redundancy and fault tolerance, and in the same time increasing maximum throughput of the service. For MySQL, there should be master-slave configuration, where updates and saves are done to the master and slave is used for reading operations.

that the response time increases as the network latency affects each request made out of the availability zone. This can affect requests to memcached and MySQL servers, as the MediaWiki is doing lots of queries to cache and database server, meaning each request will increase delay in the response time and can affect throughput of the server.

Table 5.5 shows average response times for 10 ping requests to different availability zones. Zone `us-east-1c` seems to have slowest connection while connecting to `us-east-1b` and `us-east-1e`. It would be advised not to use these combinations of zones, especially when deploying memcached and MySQL servers into the different zones from Apache server. This definitely increases latency of the response time.

Running Apache server in `us-east-1b` and all the other servers in `us-east-1c` zone, Apache was able to serve 25 requests per second as a maximum, over that, the average response time was going over 1 second and started losing

zones	east-1b	east-1c	east-1d	east-1e
east-1b	0.322 ms	1.678 ms	0.967 ms	0.671 ms
east-1c	1.630 ms	0.332 ms	0.941 ms	1.654 ms
east-1d	0.886 ms	0.978 ms	0.327 ms	0.802 ms
east-1e	0.673 ms	1.928 ms	0.814 ms	0.321 ms

Table 5.5: Pivot table showing average response times for different availability zones for ping command in us-east region. Availability zone `us-east-1a` was not accessible when this thesis was written and, therefore, left out from the table.

of jobs. It is clear, that running servers in different zones, it will increase the response time as requesting content from memcached and MySQL will take more time. Running service time experiment, where MySQL and memcached were in the other availability zone it was showing 125 ms as an average for `c1.medium` instance compared to running all the services in the same zone, the average response time was 70 ms, indicating that multiple requests take more time to be processed through network channels. For this experiment ping command between MySQL and Apache server was resulting on average 1.7 ms latency.

5.5 Conclusion of the configuration optimization

First experiment demonstrated very bad results in terms of performance and receiving large response times, thus request per second was relatively lowwe than expected. Various improvements and configuration tuning have been done to improve the performance and throughput of the services. Process to optimize the configuration has been long and slow, as there is not know-how for setting up such system in the cloud environment.

There are many variables that can affect the service performance. Changing smaller instances `m1.small` with faster instance `c1.medium` helped significantly to increase the throughput. Tuning parameters of Apache and PHP for Amazon EC2 `m1.small` instance did not really improve the service speed and the servers were easily crashing under heavy load.

Tuning of different services (MySQL, MediaWiki, Apache, PHP, nginx) eventually paid off. For example, `c1.medium` without any tuning and not using memcached was only capable of serving 8 requests per second at full steam. Adding additional services and changing configuration files, it was possible to raise the maximum throughput to 28 requests per second. Later experiments showed, that depending which CPU was assigned to the instance, the maximum throughput varies between 26 to 34 requests per second. Linux TCP parameters and open file descriptors have been changed to allow more connection and kill connection staying too long in the idle state.

5.5.1 MediaWiki configuration and tuning

MediaWiki gives system administrator a lot of different options to tune and configure the application. It gives great variety of choices for caching: (i) memcached, (ii) store in MySQL and (iii) store in the file system. Enabling memcached improved the speed of service significantly. Caching will store already rendered page, making retrieving the same page much faster than for the first request. For small MediaWiki configurations (containing one server), it was advised to use file system caching as it was the easiest to set up. Memcached is the best solution for large MediaWiki installations, where there are tens to hundreds servers actively requesting cached information from centralized location. Value retrieval from the memcached is fast as all the content is stored in the physical memory and is limited to the network speed.

Using default MySQL configuration, deadlock was triggered with the heavy load when trying to get lock for update query by MySQL system making any further queries impossible, unless the services or transactions were restarted. It was caused by page counter, that had statistical purpose to log how many view to the page were made. While generating at least 400 requests per second, some of the requests might have been for the same entry or there were too many updates, that MySQL was not able to keep track with, resulting in the blocked content. This option can be turned of from the configuration file setting variable $wgDisableCounter to the true, disabling the page counter updates.

Another problem that occurred during essential experiments was that the cache hit ratio was staying under 30%, meaning that requesting same pages had to be processed again by PHP to generate the page, because they were not stored in the cache. Cause of this problem was that memcached had limits for storing key-value pairs in the cache, the value had to not exceed over 1 megabyte. To ensure that the content was not large to store it in the cache, the content was packed by MediaWiki to reduce the size significantly, but the problem was trigged, because variable `$wgMaxMsgCacheEntrySize` by default was small, 10 kilobytes. This means that packed content, exceeding 10 kilobytes were not stored in the cache and, therefore, resulting in small cache ratio. The problem was solved by increasing the cache entry to 750 kilobytes to allow larger pages to be stored in the cached. This helped to increase the memcached hit ratio to 97%, improving the response time.

MediaWiki has built in a way, where configuration file changes will invalidate the cache immediately and every time configuration changes have been done, the cache has to be filled again. In order to overcome this problem, administrator can overwrite value in the configuration file for field `$wgCacheEpoch` to number 1, this will ensure that even if making changes to the configuration file, cache is not cleared. Field `$wgCacheEpoch` is having a value, storing previous configuration file date. If the configuration file date will changes, the MediaWiki application will take care of invalidating the cache entries in the memory. This is not a proper mechanism, while adding new servers, as the configuration file has to be changed to add memcached and MySQL server IP addresses. While doing it, the cache is invalidated, increasing average response time and increasing load for back-end servers for every time a new server is added to the server list. Parameter `$wgParserCacheExpireTime` defines how much time it takes to expire entry from the cache. To make sure that during the experiments, the cache is not going to be invalidated, the value is set to 25 * 3600 (seconds are used to define the cache length), making the cache valid at least for one day and one hour.

5.5.2 PHP caching

PHP code can be cached using opt code, this will reduce disk reads as it holds pre-interpreted PHP code in the memory and each time a new request is made, it will not have to interpret the code again. Also this reduces memory usage of the PHP scripts. For opt code caching, XCache [25] was used, available from Ubuntu repository as a package named php5-xcache. XCache uses validation check to ensure, that the files have not been changed, when they are interpreted. If there are changes in files, cache is invalidated and the new source is interpreted to execute code with the latest version. Experiments showed that when XCache opt code cacher was enabled, there were at least 3 times faster response times. Without opt code cache, one Amazon EC2 `c1.medium` server was able to server 8 requests per second. Figure 5.13 shows the difference between both configurations. When enabling XCache on PHP it dropped average service time from 200 ms to 70 ms. This configuration was already using memcached showing that not only caching with memcached helped to improve the service time, but other services needed to be also tuned to maximize the throughput.

5.5.3 MySQL tuning

For MySQL `max_connections` was increased from 100 to 500 to ensure enough connections are available, when service was under heavy load. Socket `wait_timeout` value was reduced from 28800 (it is 8 hours) to 180 seconds. 8 hours is long time to have sockets in the idle state waiting for new connections, eating up available resources. Another variable changed in the configuration was `connect_timeout` to make sure long queries or unresponsive server connections were killed and released for other connections.

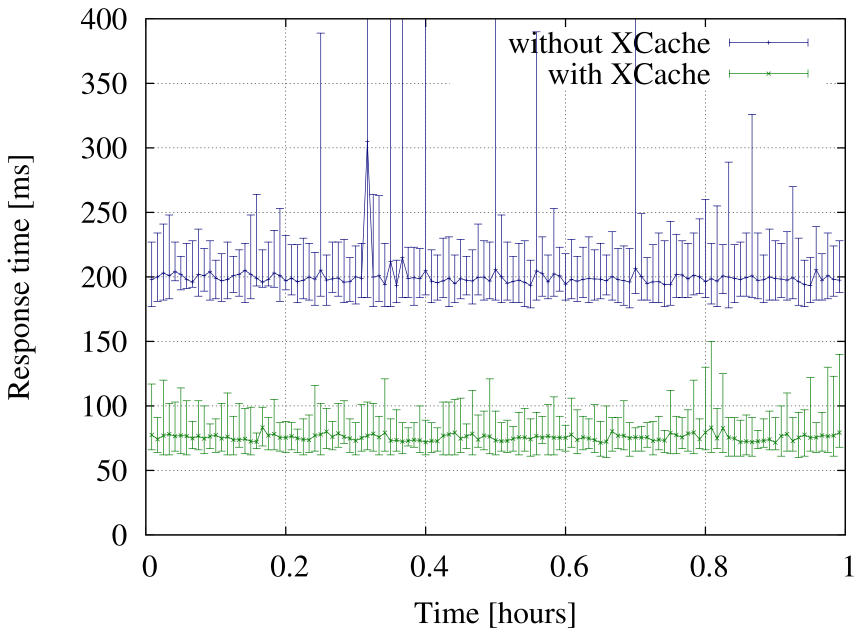


Figure 5.13: Comparisons when running MediaWiki infrastructure with XCache enabled and without XCache on `c1`.medium Amazon instance. These are response times with 30 second resolution. Both configurations used memcached with filled cached.

Chapter 6

Policies and Validation of Scalability

This section describes two policies used in running large scale experiments for 24 hours to validate the framework. First policy uses the revenue function to calculate the optimal amount of servers, whereas, the other policy uses Auto Scale service provided by Amazon to add and remove servers from the cloud. Both of the policies are limited to 20 servers as maximum to give better comparison. For comparing the policies, important parameters are monitored with 30 seconds interval for every instance running on the cloud. At the end of each experiment, CPU utilization, server hours, jobs lost, average response time and revenue generated have been calculated.

6.1 Why is elasticity of the cloud important

Cloud computing allows users to add and remove servers on fly depending on their current needs, making possible to regulate servers count in the cloud by the number of incoming requests. This helps service owner to save money in terms of running smaller number of servers at night and serving all the clients in the peak hours. This will save money, as there is no need to run the extra servers needed at the peak hours for whole time, reducing cost significantly.

Example of elasticity. Let's consider that owner of a service has at peak 600 requests per second to his homepage and have cloud configuration same as defined in the Table 6.1. Each server is capable of serving at maximum 30 requests per second, this means at the peak there should be at least 20 servers to serve all the incoming requests. At midnight the service is visited at interval 100 requests per second. Roughly at average, if the traffic curve is constantly increasing and decreasing, the service equals to 350 requests per second for a whole day. If the provisioning is done by the incoming traffic, it will use $\frac{350}{30} \times 24 = 280$ server hours, thus having to pay 47.60$ for the resource used for a one day. For comparison, using fixed amount of servers running, provisioning servers count for the peak of 600 requests per second, meaning that $\frac{600}{30} \times 24 = 480$ server hours are being used for a whole day and having to pay 81.60$ for the resources. This is 1.7× larger than using the dynamical amount of servers, thus showing that with correctly provisioning it will save money for the service provider and therefore can increase net revenue.

This configuration can be used by PaaS or SaaS providers, who are trying to minimize the cost to run the services, but in the same time trying to be able to serve every client using their service, maximizing the profit. PaaS and SaaS provider are charging customers based on monthly usage or how many requests have been done, in the same time IaaS provider Amazon EC2 is charging PaaS or SaaS provider by instance hours. Using dynamically allocated servers, running servers at minimal costs at the night hours and serving all the customers at the peak hours, it is possible to take most out of the service provided by maximizing profit.

Figure 6.1 shows three different ways how to provision servers. If the arrival rate is exceeding the throughput, it is under provisioning. If there are more servers than needed, then it is over provisioning and paying more money to run the servers. Ideal case would change servers depending of the arrival rate. Optimal allocation tries to find the best configuration to meet these demands. To improve optimal allocation, one needs to predict the traffic or trend of the traffic, as traffic fluctuates over the time, it is hard to provision servers for the next hours without knowledge, how the traffic changes. These experiment does not include traffic prediction and therefore with increasing traffic, there is a slight loss in the incoming requests, depending how large is the increase.

6.2 Basic configuration for running the experiments

For using optimal policy and for comparing revenue for different experiments, it is needed to describe some basic parameters. The configuration for experiments is shown below in table 6.1. Service time for each request

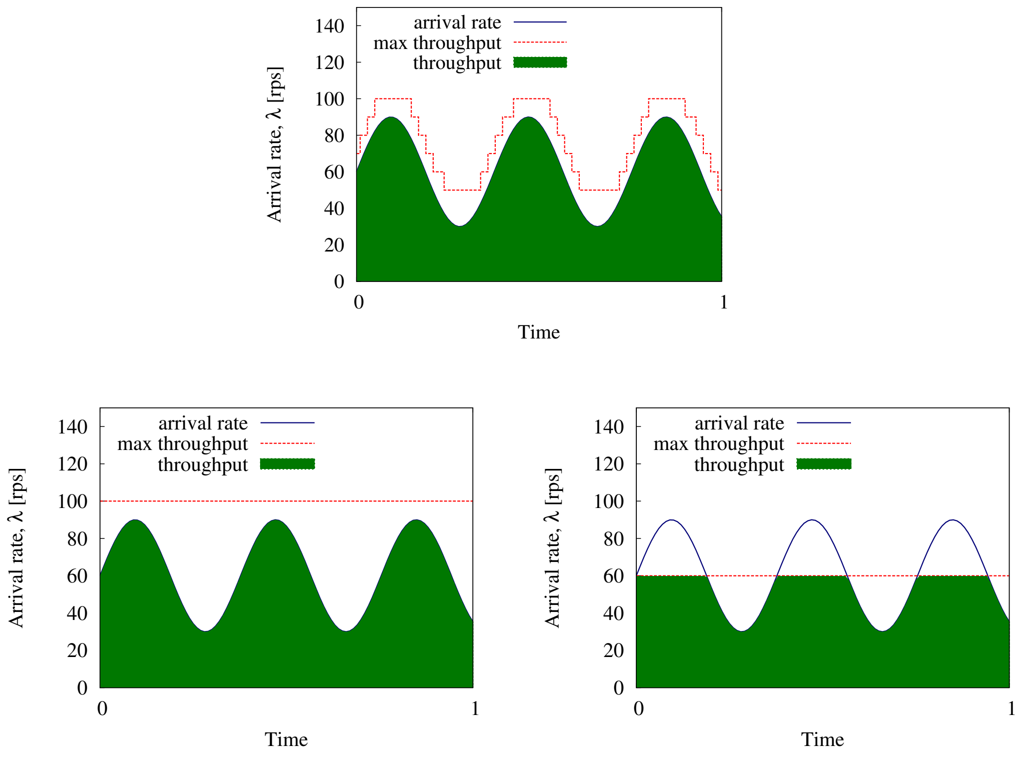


Figure 6.1: Left graph shows over provisioning, where the policy is running more servers than needed. Right graph shows situations of under provisioning, running fewer servers than needed. Graph at the top shows how dynamically allocated servers can handle traffic fluctuating in time.

is set to 70 milliseconds, this is time it takes to serve 50% of the fastest requests. Amazon EC2 cloud charges clients based on full hour for the instance they are using, there needs to be time limits when the servers can be added and removed. To be safe side, for each full hour, servers are added and for each full hour before 5 minutes, servers are removed. This will ensure, that no server, what is removed, is charged for an extra full hour, because the instance termination takes more time than regularly. This type of server adding and removing cannot be managed when using proactive policies like Amazon Auto Scale and this approach is used only for the optimal policy.

Apache server is using the multi-core configuration, two jobs can be concurrently executed in the server, meaning that we can serve 2 requests with 70 milliseconds (one request can be counted as 35 milliseconds). That explains, how the service rate can be 28 requests per second, as it determines how many 35 milliseconds jobs server could serve during one second. Charge per job is selected through multiple simulations in a manner, that with peak load, there are at least 20 servers involved.

Using at maximum 20 servers, the maximum generated load for the experiments was set to 600 requests per second. This gives the feedback how the servers work under the saturation point. This amount of requests should ensure, that load balancer is not hitting the network bandwidth limitation (800 Mbps). From previous experiments, the network overall utilization was around 79%. The limit of 800 Mbps has been tested and verified by Huan Liu and Sewook Wee [50]. Otherwise exceeding this limitation means, that there is need to use DNS load balancer or Amazon tools to distribute the load.

Parameter	**Description**	**Value**
μ	Service rate	28.571 jobs/sec
c	Charge per job	0.0017¢
d	Cost per server	17¢/hour
n	Running servers	1–20

Table 6.1: Parameters used for revenue based policy.

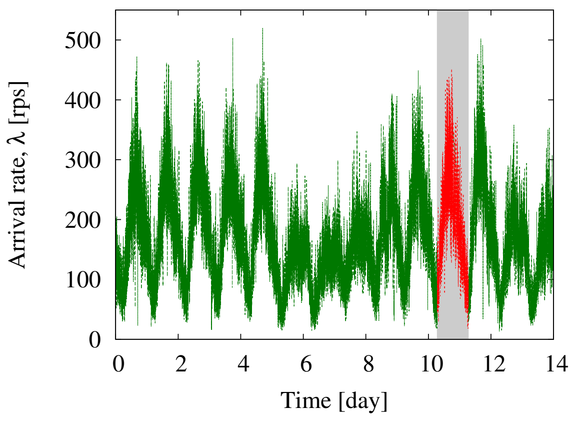


Figure 6.2: Two week ClarkNet traffic trace with 1 minute intervals showing arrivals per second. The red line defines section of the trace used to generate the load for the service. This section has been scaled up to 600 requests per second in the experiment.

6.3 Traces used to test policies

For testing the framework and how well service scales in the cloud, ClarkNet traces were used, that were available from The Internet Traffic Archive [1][2] web page. It contains two sets of ClarkNet traces with overall of two weeks of traffic from WWW service provider. The traces were with one second precision, but these were changed to 1 minute intervals. For certain time periods, there were sudden drops in arrival rate (from 200 requests per minute to 5 requests per minute), the spikes were replaced with similar values from previous minute. Figure 6.4 shows section of trace marked in a red color (one day), that was used to run the experiments. The trace was scaled between 35 requests to 600 requests per second for better fitting into the experiment configuration.

6.4 Utilization policies

Amazon Cloud gives great set of tools to manage cloud configuration on fly. One of the practical tools is auto scaling option based on variety of parameters that user can set. Because the current configuration has bottlenecks mostly on Apache side, especially with CPU usage, it would be wise

[1]http://ita.ee.lbl.gov/

[2]ftp://ita.ee.lbl.gov/traces/clarknet_access_log_Sep4.gz

to monitor CPU Utilization on Apache instances and manage server count based on average CPU usage over the certain time.

6.4.1 Auto Scale

Auto Scale allows scaling of Amazon EC2 capacity up or down automatically according to conditions defined [9]. This ensures, that enough servers are running during demand spikes to maintain performance and servers are released, when demand drops to minimize the cost. This service is available with Amazon CloudWatch and no additional charge is taken beyond the network transfer and API call fees. Auto Scaling is particularly well-suited for services that experience variability in hourly or daily usage, and job size and time it takes to be processed is not fixed.

Configuring and setting up Auto Scale is simple, user needs the additional tools that are available on Amazon web page [3], have a valid Amazon account and has configured instance, used for scaling. If each server has its own database, no additional configuration is needed. It is up to user if they want to use Load Balancing provided by Amazon or implement their own load balancing mechanism. Using Amazon Elastic Load Balancer, it provides the single IP, where the back-end servers are available and accessible, additional fee is charged by amount of traffic it processes and how many hours it has been working. For each hour, user has to pay 0.8¢to 2.5¢for each GB, price depends how much data has been already processed [20]. Compared with `m1.micro` spot instance for balancing load between back-end servers, it is still cheaper to use Amazon Elastic Load Balancer.

It is more complicated, when user wants to implement their own load balancing or use some other program for load balancing, but still use Amazon Auto Scale to scale the servers in the autoscale group. It needs to track, when new instances are started or already running instances have been terminated. One option is to use Amazon command line or API tools to retrieve list of active servers using command *ec2-describe-instances* and keep track of which servers have been terminated and which have been added. The other option is to build a script on the Auto Scale instance that sends the heart beat to load balancer at certain intervals. Load balancer has to listen incoming packets and if it is coming from a new IP, it means that a new instance has been started and if it does not receive the heart beat from already known instance for certain time period, it can be removed from list as the server is probably removed by Auto Scale.

In order to replicate benchmarks and compare Amazon Auto Scale with

[3]http://aws.amazon.com/developertools/2535

optimal policy, nginx was used as the load balancer that had 10 concurrent connections set to each back-end servers to avoid overloading the system. Going through Amazon manuals, there was no option to change maximum amount of concurrent connections with the back-end servers, while using Amazon Elastic Load Balancer. Script for keeping eye on the cloud was deployed on the nginx instance. Script was responsible of keeping track of the active back-end servers allocated by Amazon Auto Scale in the scaling group and knows IP addresses for load balancer (nginx), cache (memcached) and database (MySQL). This allowed easier configuration, when the new instances were added by the Auto Scale algorithm into the cloud. The program main loop queried running instances from the cloud, configured the new instances MediaWiki configuration with correct memcached and MySQL IP, and made modifications in nginx server list, adding the new servers. In the same manner, terminated servers were removed by the program from the nginx server list and the service was restarted.

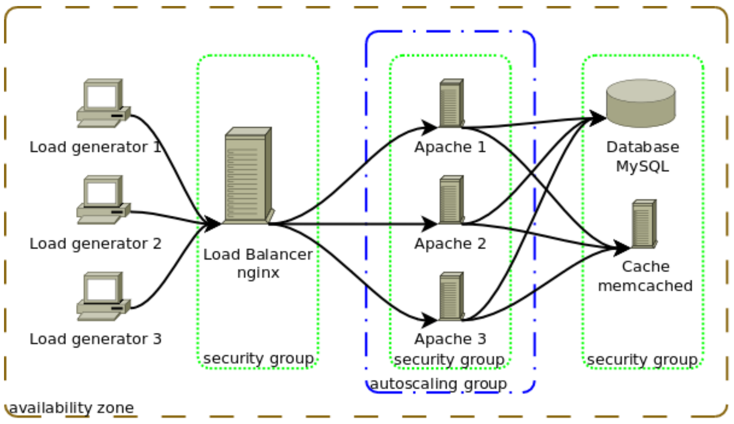


Figure 6.3: Showing Amazon Auto Scale experiment configuration in the cloud. Servers were kept in the same availability zone to reduce the latency of requests. Autoscaling group has been created, where Auto Scale algorithm can replicate servers when the demand increases and terminate servers from the group, while the demand decreases.

Figure 6.3 shows configuration in the Amazon cloud. Auto scale group was created with Amazon tools helping for Amazon to keep track of servers, which were added, when the demand increased. In that way, Amazon knows, which servers can be removed, if the demand decreases, as it takes from the autoscale server pool existing server and terminates it. Auto Scale configuration requires, which instance (instance ID) has to be run and in which availability zone they has to be deployed. Auto Scale was configured with following parameters: 1 server at minimum, 20 servers at maximum, upper CPU utilization threshold 70%, lower CPU utilization threshold 60%, the breach time 15 minutes, upper breach increase as 1 and lower breach decrease as 1. CPU utilization is taken based on average among all of the instances running in the autoscale group. Breach time defines how much time certain threshold has to be in order to add or remove servers. For current experiment configuration, when average CPU utilization stays over 70% for 15 minutes, it starts adding servers one by one and if it drops below 60%, it removes servers in the same way as it adds.

The scaled traces from ClarkNet were starting with over 100 requests per second, meaning that 1 back-end instance is not capable of serving the request. As the arrival rate for the first hour was much larger than for the last hour, it was not possible to set minimum amount of servers higher, as it might affect the results. Amazon Auto Scale has to be warmed up with constant arrival rate of 100 requests per second in order to add necessary amount of servers. If this was achieved, the experiment was started. The results are shown on the figure 6.4.

Auto Scale removing servers. During the experiment, Auto Scale algorithm removed instance from the running server pool that was not closest to the full hour, sometimes removing instances that was 30 minutes away from the hour. Considering that Amazon charges for full hour, this is good for increasing their profit as many users use this service and they are not able to use the server provided by auto scale in full extent.

6.4.2 Auto Scale results

Experiment using Auto Scale was using large *CPU utilization* upper bound, meaning that new servers were added too late, thus losing the traffic. Characteristics of this approach are that it will try to increase rapidly more server, when the certain threshold has been reached. Even if one server has been already added and the load is still high, it tries to add a new server. There is an option called *cool-down* when using command *as-update-auto-scaling-group* [12] that will freeze Auto Scale for the certain time period when it has made a new decision to avoid excessive adding or removal of servers. Parameter *cool-down* was set to 2 minutes as it would help to gain quickly more servers, when the load increases more than the system capabilities available for serving the load. Probably a small cool down time would be bad, when the traffic is fluctuating more, as it might remove and add rapidly servers, making cost of the servers much higher, as they are removed too quickly from the cloud and are not used to full extent and the partial

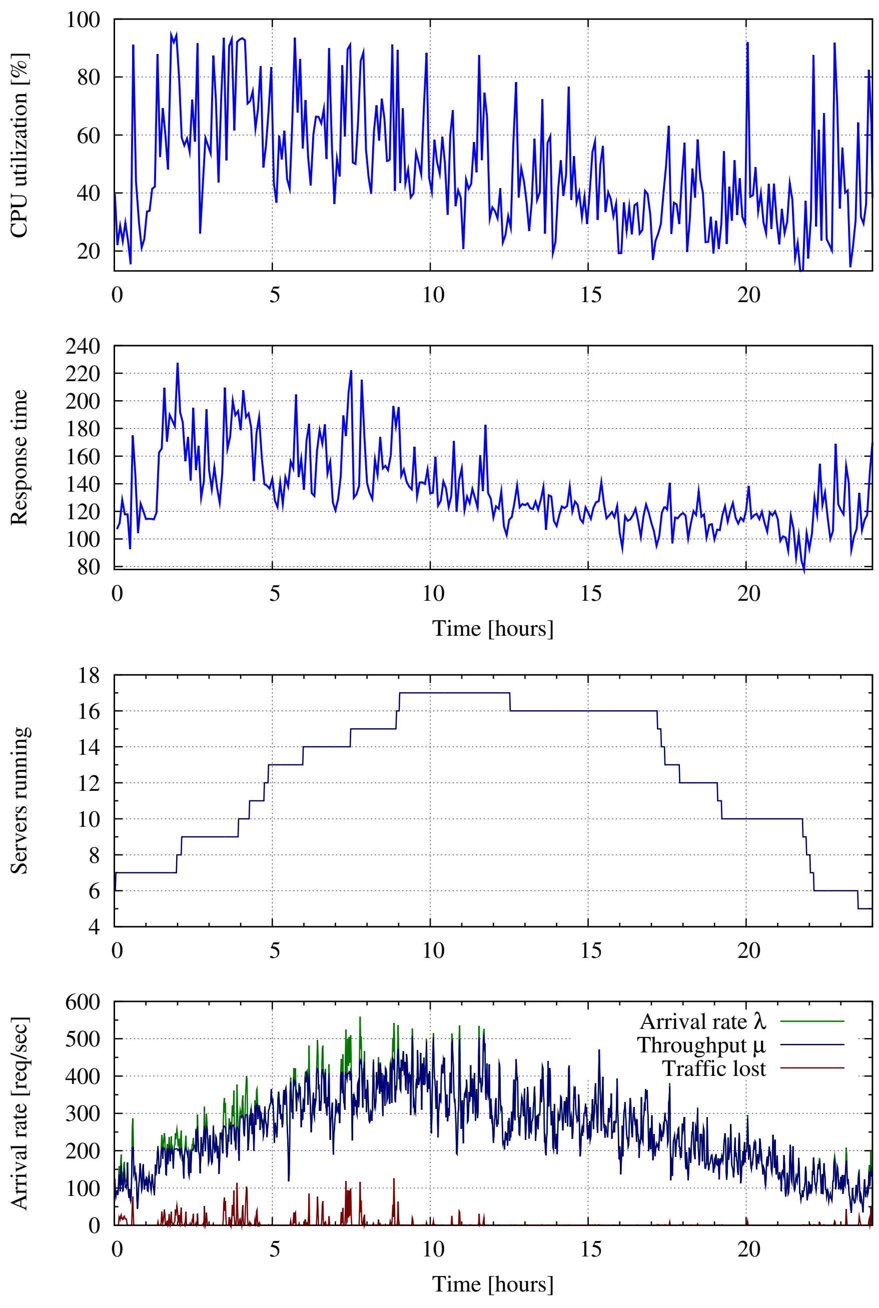


Figure 6.4: Running servers compared to arrival rate λ, CPU usage and response times over 24 hours using Amazon Auto Scale policy.

hours are charged as full hours.

The second part of the experiment gave better results as the traffic was decreasing and all the requests went successfully through. Auto Scale did descent job determining when to add or remove servers. There has some room to improve the algorithm by changing some of parameters, thus it would cope also with rising trend (one option would to decrease upper CPU threshold to 60 %). One side-effect of the Auto Scale is that it tries to remove sometimes servers to early not letting them to run proper amount of time reasonable for the Amazon Cloud service user.

6.5 Revenue based policies

Revenue based policies are calculating amount of servers depending how much revenue one request is generating, maximum throughput of the server, server charge for an hour and arrival rate for the next time interval. Every processed request by a server generates a fixed revenue. This revenue might come from advertisements or from sales. It is not straightforward to set revenue, as there is no clear linkage between how much one request can increase overall income. Using these parameters, the revenue based policies tries to find the best trade off adding or removing servers, while maximizing the profit. It needs to be careful when setting up parameters as all the parameters can largely affect policy decisions how many servers are needed. Setting income for each request larger, the servers are most likely to run in an idle state. Decreasing the income will decrease the amount of running servers, which will decrease the maximum throughput, increases amount of jobs in the system and increasing system utilization.

6.5.1 Algorithm for provisioning the servers

To use optimal policy for provisioning servers, there needs to be algorithm that takes care of process of using policy to calculate the amount of servers, provisioning servers in the cloud and updating load balancer with the new server list. Algorithm works as follows:

1. 5 minutes before the full hour measures arrival rate for the last hour (weighted average does not improve the net profit for policy based algorithms and therefore is left out). Arrival rate is taken from nginx HTTPStatusModule that is running on address `localhost/server-status`

2. 5 minutes before the full hour, run policy algorithm to determine the servers count using average arrival rate calculated from the previous

step

3. Removing servers from the cloud, if the algorithm calculated fewer servers than currently running and re-configures the load balancer
4. If algorithm calculated more servers than running, wait for 5 minutes and then request new servers from the cloud server pool
5. For each 30 seconds, check if requested servers have changed state from pending to running. If yes, configure the MediaWiki servers with correct MySQL and memcached IP addresses, add new servers into the load balancer list and reload nginx
6. If all requested servers are configured and added into the load balancer server list, halt previous step

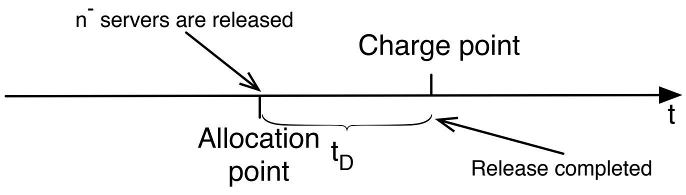


Figure 6.5: Removing servers should be done before full hour, because otherwise there is need to pay for the extra instance hour.

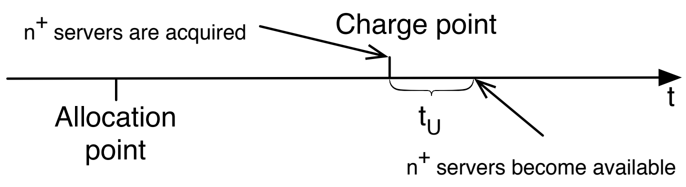


Figure 6.6: Adding servers is done at the full hour mark to make sure that while the servers are added and needed to be removed, there is no need to pay for the extra instance hour.

Figures 6.5 and 6.6 show how the servers are added and removed from the cloud. Removing step occurs first and algorithm, calculating the amount of servers is triggered at this point. If there is no need to remove any servers, algorithm will wait until full hour, when the adding servers function is called out. For adding servers, calculated amount of servers can be taken from the previous step, as there is no need to calculate it again. Adding and removing servers in a way described by the algorithm ensures that there is no need to pay for extra instance hour for the resources that cannot be use (i.e. instance has been working for 1 hour and 1 minute and is terminated, meaning that the instance is charged for two hours).

6.5.2 Arrival rate for the next hour

There are several options how to feed in average arrival rate for algorithms determining needed servers count for the next hour. Simplest way is to take average arrival rate for the previous measured hour and use this as an input. The downside of this approach is that the average arrival rate can increase or decrease for the next hour, because of the characteristics of the traffic fluctuate over the time. To improve giving better arrival rate for the algorithm, it is possible to use weighted average, where the last measured units have more weight while calculating the average, meaning that it should give slight trend, whatever the arrival rate is going up or down. Because servers are charged based on full hours, the arrival rates should be measured more than once per hour to improve the accuracy of the trend. For example, if the load starts to increase at the end of the hour, probably it has to be increased at the beginning of the next hour, making the given average arrival rate more precise.

Experiments showed, that when using 20 servers, having 600 requests per second at peaks and using ClarkNet traces, there was not a big difference between using the regular average and weighted average (calculating arrival rate with 5 minute intervals), as the algorithms were resulting the same amount of servers needed for the next hour.

Previous hour traffic gives a good presumption, what the traffic might be for the next hour. In the case of best scenario, it will stay at the same level. If the arrival rate increases, then amount of servers have been underprovisioned. It depends how optimistic the parameters are when using different algorithms. If letting too much room for the traffic and the arrival rate drops for the next hour, then the servers have been over provisioned. This means that there are more servers running than needed and we have to pay more for running the service.

Second option is to predict the load for the next hour, based on the training set or using previous arrival rates for each hour. Visits to the sites are mostly deterministic and can be predicted. You might say, that at the day time, there are more visits than at the night, when everybody is sleeping. It also depends, who are the visitors. In the site, visited from various locations of the world, the traffic between night and day time does not differ, fluctuating only little. To increase the prediction, the predictor function has to take into account what weekday it is, since at the weekends the traffic volume tends to be smaller.

Even for both options, the decision should be taken carefully when to add or remove servers. It is rather easy to do if one takes an average hour during a day, but it is hard to predict, when looking at much smaller time scale. Traffic fluctuates largely when looked arrival rates per second. Service owner must ensure, that these spikes can be handled by the configuration provided and service is reachable for everyone, who tries to connect with it.

6.5.3 Optimal heuristics

Second experiment was using the optimal heuristics to determine the servers needed to run the service in the cloud. It uses several parameters to maximize the outcome, using hill climbing method. Tuning parameters for optimal heuristics is tricky, as it can significantly change the end result. It needs to take into account, that service time used by optimal heuristics has to be divided by number of cores, as the jobs can executed in parallel. Service time, cost of server and throughput are fixed values and the policy mainly depends on how much income one request gives (or charge Saas or PaaS user pays for each request).

Formulation

Let us give a mathematical formulation for optimal heuristics. Consider arrival rate λ over time period one hour. Algorithm goal is to find the best optimal using hill-climbing method. It is unimodal function [56], meaning that there is only one single maximum. This policy tries to maximize parameter net revenue. The idea is to find the best option where enough jobs are going through the system, but still is capable producing maximum amount of money. It needs to find maximum for a function 6.1

$$r_n = \mu \times c - n \times d \tag{6.1}$$

where μ is the maximum throughput for one hour for current configuration calculated from arrival rate (using predictive arrival rate or last hour arrival rate) using equation 6.6, c shows income per request, n is the incremental value that shows how many servers there should be, d is charge for each server for an hour and r is the net revenue, that is tried to be maximized. These values are all presented in table 6.1. To find optimal solution, Erlang B and erlang unit is used to calculate maximum throughput for each n, while knowing the arrival rate to the system. For each n, hill-climbing method is used to find out local maximum. Server amount is increased until the revenue function starts to decrease, indicating that function has reached to its maximum.

Erlang B gives probability traffic going through the system and is used by telephone companies to calculate, how many agents are needed to be able to receive all the calls made by the customers. It uses service time, arrival rate and amount of servers to show how many requests can be processed. Using Erlang B in the optimal policy calculations, it is possible to calculate maximum theoretically throughput of the configuration, thus making possible to find maximum of equation 6.1 (e.g. we cannot serve 60 requests per second, while using only 1 server with maximum throughput of 28 requests per second).

The formula provides the GoS (grade of service) which is the probability Pb that a new request arriving will be dropped by the load balancer since, all the back-end servers will be busy [55].

$$P_b = B(E, m) = \frac{\frac{E^m}{m!}}{\sum_{i=0}^{m} \frac{E^i}{i!}} \tag{6.2}$$

$$E = \lambda \times h \tag{6.3}$$

Using equation 6.3, we can calculate the offered traffic rate in erlangs, using λ as an arrival rate and h as a service time (hold time). Because of using two core machine for the back-end servers, the service time is two times smaller as the server is capable of handling two requests concurrently ($70ms/2cores = 35ms$). Putting E from formula 6.3 to formula 6.2 and using m as an amount of servers, we can calculate the blocking probability. Value P_b will be used, when calculating revenue for formula 6.1, calculating λ from equation 6.6, thus getting the maximum throughput.

For finding optimal configuration, hill-climbing method is used in equations 6.1 and 6.6 to find best local maximum. This approach iterates through the list of possible server configurations from 1 to ∞ (see equation 6.4) and stops only when finding that the next iteration has got a smaller value than current iteration, meaning that the current iteration has got a maximum value. This function will return n, that represents amount of servers needed to run the service.

$$r_{best} = max(r_1, r_2, ..., r_\infty) \tag{6.4}$$

Running several experiments and tests, Erlang B calculated throughput is not the same as the throughput achieved by the real environment. Erlang

B assumes that for each agent (server) is capable of handling only one call (request). This does not match with computer networks, where computer is capable of handling multiple requests in the same time. Blocking value calculated by Erlang B is much higher than in the real situation. Load balancer uses as maximum 10 concurrent connections for each back-end server and excessive requests are dropped. This is M/M/n/n queue model and does not fit well with Erlang B.

Second approach is to use server count and traffic rate in erlangs calculated by equation 6.3, this will give system utilization that is similar to the Linux 1 minute load. Equation 6.5 shows the way to calculate the blocking probability, that is similar to the results of experiment. If the value is under 0, it means no job is blocked and the value is assigned to 0, not to allow negative values. Using calculated blocking probability and lambda (see equation 6.6), equation gives how many requests will go through the system.

$$P_b = 1 - \frac{s_{count}}{E} = 1 - \frac{s_{count}}{\lambda \times h}, if P_b < 0 : P_b = 0 \tag{6.5}$$

$$\mu = (1 - P_b) \times \lambda \tag{6.6}$$

In equation 6.5 the parameter s_{count} is amount of servers and other parameters are the same as previously described. Using modified version of erlang formula it shows the similar results as the experiments does, thus making the calculation of revenue more precise. Figure 6.7 shows how the optimal maximum is calculated with revenue function. The revenue is increasing in the first section of the graph, because more jobs are going through the system and each job earns more money than there is a necessity to pay for engagement of additional servers. When the maximum is reached, adding more servers do not improve throughput and the revenue starts to decrease, as we have to pay more for the extra servers, but the income does not increase. Using for arrival rate 600 requests per second and other values are the same as shown in the table 6.1, the optimal maximum is achieved with 21 servers, meaning that at the peak, there needs to be at least 21 servers to accommodate the traffic.

Hill-climbing method. It is an iterative algorithm that starts with an arbitrary solution to a problem, then attempts to find a better solution by incrementally changing a single element of the solution. If the change produces a better solution, an incremental change is made to the new solution, repeating until no further improvements can be found [21].

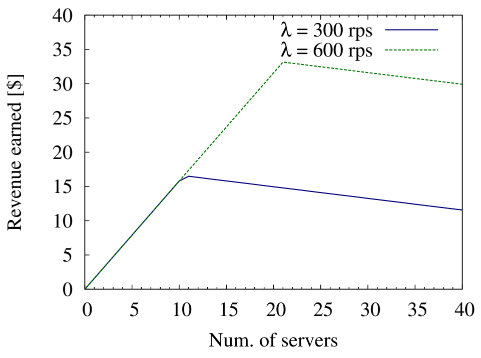


Figure 6.7: Revenue function values, when trying to find the optimal maximum for 300 requests per second and 600 requests per second using erlang units to calculate the throughput.

6.5.4 Optimal heuristics results

For optimal heuristics, service time was set as 35 ms (using 2 core machines, meaning 70 ms service time has to be divided by 2, as it is possible to concurrently process two requests) and Fair module used 10 concurrent connections for each back-end server. Figure 6.8 shows, that while load is increasing, the servers are added later and maximum throughput is smaller than actual arrival rate, thus losing jobs. There should be at least 1 extra server in spare, when calculating amount of servers needed to run the service. The other half of the experiment was with the decreasing traffic closing to the evening and fewer clients were visiting the page. Removing the servers worked well and failure count was minimal.

To improve the optimal policy, it needs to have a trend function that can determine, when the traffic increases and if that happens, an extra server should be added to the already provisioned servers to accommodate the increasing arrival rate.

This configuration seemed to work better than Amazon Auto Scale function, resulting in a fewer jobs lost. This is caused by, that Auto Scale uses large CPU utilization threshold when trying to add the new servers or the breach time was set large. Decreasing one of these values should improve the outcome.

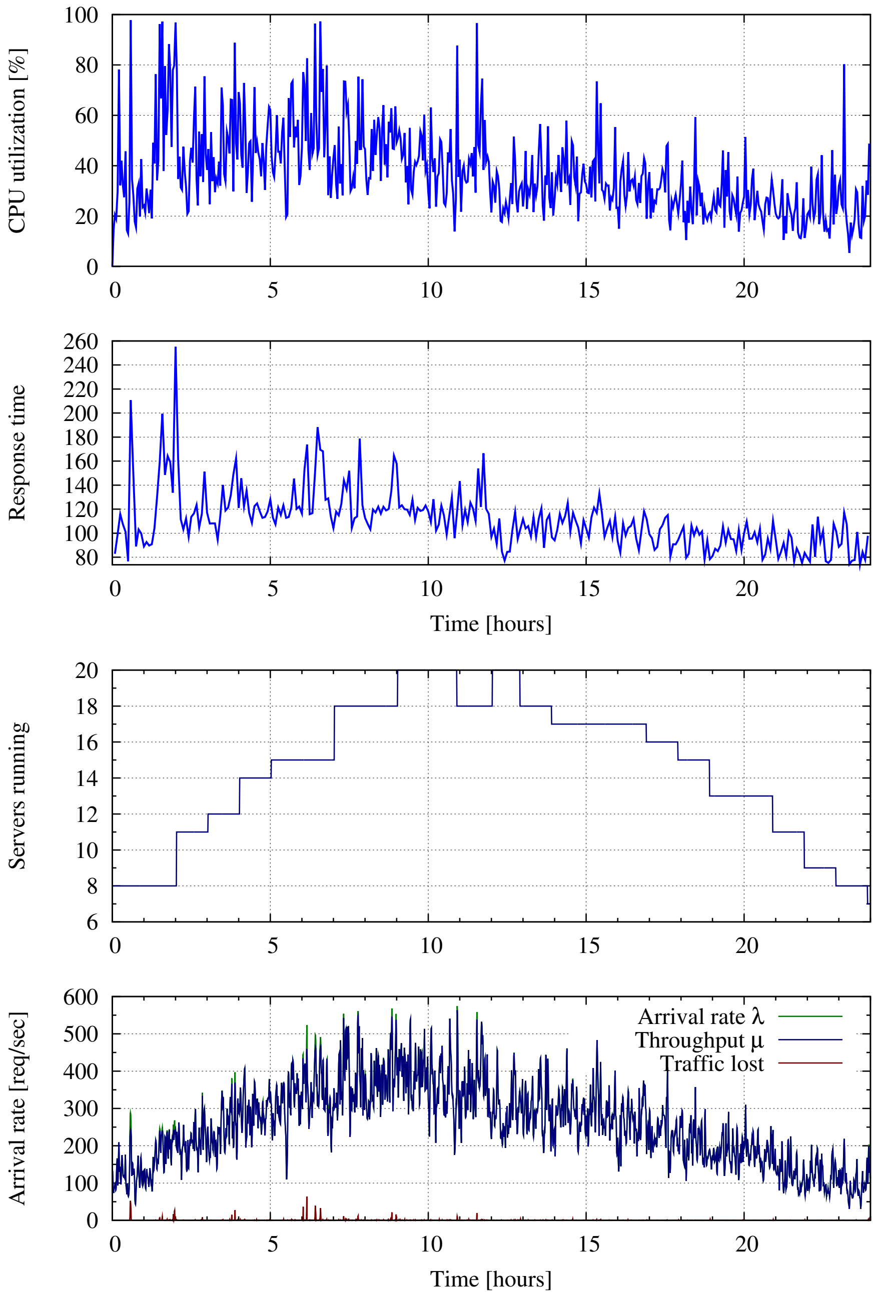


Figure 6.8: Running servers compared to arrival rate λ, CPU usage and response times over 24 hours using Optimal policy.

6.6 Summary and results

Each policy has its own pros and cons and using one of those policies mainly depends on the system, and what the decisions management is taking in terms of what they are willing to pay. Amazon Auto Scale is mostly enough when determining the needed servers count for the system. Current configuration will work fine for traffics, where the load is steadily increasing for each hour and does not have large spikes. Amazon Auto Scale has variety of parameters, that can be tuned, but this mainly relies on the fact, which traffic the service is getting. If there are more spikes and traffic is fluctuating largely, lower CPU threshold for adding the servers should be used. It is possible to define, how many servers are going to be added or removed when certain threshold is exceeded. However, the system owner has to be careful when defining this, as adding too many servers may result in lower CPU usage, meaning that those extra servers are removed in no time. Amazon Auto Scale policy will work fine for different traffics, that varies in response time as the primary unit for taking the decisions for server provisioning is taken from average CPU and therefore making it an easy service that can be set up with small time and little effort, as mostly there is no need to do additional experiments to identify service time.

Revenue based policies are much more complex as system owner must be aware of the servers capabilities and how they are running in the cloud. This is why some preliminary experiments with the servers have to be done, to know the limits for each server. Even though the service time is mostly fixed value if the response time does not vary much and server hour based charge is fixed, there is need to calculate the correct charge (income) value for each request made. There is a trial and error process that involves many simulation and running experiments to see the best configuration. Compared with Amazon Auto Scale, revenue based policies are harder to set up, as it involves mathematical complexity and other factors like response time variation that can affect the results.

measurement	Auto Scale	Optimal policy	always on
min. response time	56 ms	55 ms	53 ms
avg. response time	144.6 ms	116.0 ms	101.8 ms
99% response time	409 ms	321 ms	254 ms
avg. CPU usage	44.62%	37.22%	23.84%
Instance hours	312	351	480
Cost of the servers	53.04$	59.67$	81.60$
Amount of jobs lost	343763	32288	2148
Revenue	318.52$	317.18$	295.76$

Table 6.2: Summary table of Auto Scale and optimal policy algorithms. Experiment conducted lasted 24 hours and the same traffic was used resulting on 22.2 millions of requests. Cost of the servers and revenue are calculated only using Apache servers and one instance hour for `c1.medium` charge is 0.17$.

Chapter 7

Conclusions

This master thesis studies possibilities to develop infrastructure to run MediaWiki application on the cloud, that is capable of scaling MediaWiki replica instances vertically, depending on arrival rate. This infrastructure will support measuring performance of the cloud, making it possible to find possible bottlenecks from the service. This work includes tools how to measure and monitor performance of servers in the cloud.

Amazon EC2 cloud is used to deploy the infrastructure. It allows to run virtual images on the cloud, that are charged by instance-hours used. Amazon users can make their own virtual images with the required software, that can be easily replicated throughout the cloud. MediaWiki application was installed one of those images running Ubuntu operating system. Framework was built to support scaling the cloud and configuring servers for different experiments.

During development phase, different experiments were performed to clarify the best possible solution to run service in the cloud environment, as Amazon EC2 cloud gave users great variability of servers and other services, that could be used. In the case of some servers and applications there was a need for configuration tuning to improve performance and throughput of the system, as they did not work well as out-of-the-box.

This master thesis studied possibilities to scale servers in the cloud, using Amazon own Auto Scale service, which is free of charge and optimal heuristic for provision of needed servers. Two algorithms were used and compared, to see pros and cons and how to configure algorithms to support scaling. Amazon EC2 API tools were used to support managing the cloud. All the instances were constantly measured and statistics gathered to get information about CPU, memory and network usage. Validating scaling properties and algorithms, large scale experiment for 1 day was conducted for both of algorithms where at least 22 million requests were generated. Different experiments demonstrated that when calculating the cost function to hold server up in the cloud would pay less money if using the dynamical allocation of servers than having the fixed number of servers running.

Chapter 8

Future Work

This section describes some important topics not covered within this thesis. This topics have to be considered as the crucial ones and they should be studied in future in order to improve the work flow of experiments or policies.

8.1 Benchmark Tool Updates

Wikijector, which came with Wikibench, was taken as an example, while building BenchMark program. BenchMark program supported the predefined arrival rate (λ) for each time period and had URI list to make the traffic for the front-end server. It made easier to conduct the experiments as the trace logs provided by the Wikijector meant that the requests had to be changed to fit with the data dumped into the MySQL database or whole database had to be uploaded into the database. With Wikijector, it was harder to fill the cache and remove 404 `Not Found` page errors from the trace files, making it time consuming to conduct various experiments.

BenchMark had some certain fall-backs that degrades the performance for generating new requests. It uses fixed size thread pool to hold generated connection. Initializing the thread and starting the thread took time. If setting thread pool count too small, the available threads might run out with large experiments and already running threads were terminated, while still processing response from the web service. Using `c1.medium` instance type, that had 2 cores, was only capable of generating 500 requests per second. The solution would be to use the non-blocking sockets for generating requests to the server. It will allow to generate larger volume of traffic with lower number of load generators and reducing experiment cost. Instance running the load generator CPU usage was tried to hold under 50%, as larger usage might affect response time measurement precision and this made comparing different experiments harder.

8.2 Server Comparison

With cloud based configuration there are thousands of computers that have been bought in separate times have different conditions in terms of wear and have different components. These results in inequality between different servers requested from the cloud server pool. Experiments conducted by this thesis did not take into account the performance of a single server. It is possible to have some script on an instance image, that will be executed at server boot and would measure the performance of the server, e.g. measure CPU speed, how much time it takes to calculate complex formulas, also speed of a virtual hard disk, measure of time and throughput of writing and reading from it. It is possible to build metrics from previously mentioned parameters and this value can be used to compare with the other servers, and the load balancer could use this as a measurement when putting servers weight while reconfiguring server list in the configuration file. This will reduce load on servers that are slower and increase load on servers that are faster, but still maintain a stable performance and not overloading the system.

There were some cases, where one server was working at $\frac{1}{2}$ or even $\frac{1}{3}$ of speed of the fastest server under heavy load, even if they were same instance type. Using above mentioned method to compare all the servers currently running, we could eliminate the problem, but there is still a problem with policies, because they have to consider maximum throughput based on the measured performance.

Raamistik pilvel põhinevate veebirakenduste skaleeruvuse ja jõudluse kontrollimiseks

Magistritöö (30 EAP)

Martti Vasar

Resümee

Antud magistritöö uurib võimalusi, kuidas kasutada veebirakendust MediaWiki, mida kasutatakse Wikipedia rakendamiseks, ja kuidas kasutada antud teenust mitme serveri peal nii, et see oleks kõige optimaalsem ja samas kõik veebikülastajad saaks teenusele ligi mõistliku ajaga. Amazon küsib raha pilves toimivate masinate ajalise kasutamise eest, ümardades pooleldi kasutatud tunnid täistundideks. Antud töö sisaldab vahendeid kuidas mõõta pilves olevate serverite jõudlust ning võimekust ja skaleerida antud veebirakendust.

Amazon EC2 pilvesüsteemis on võimalik kasutajatel koostada virtuaalseid tõmmiseid operatsiooni süsteemidest, mida saab pilves rakendada XEN virtualiseerimise keskkonnas, kui eraldiseisvat serverit. Antud virtuaalse tõmmise peale sai paigaldatud tööks vaja minev keskkond, et koguda andmeid serverite kasutuse kohta ja võimaldada platvormi, mis lubab dünaamiliselt ajas lisada servereid ja eemaldada neid.

Magistritöö uurib Amazon EC2 pilvesüsteemi kasutusvõimalusi, mille hulka kuulub Auto Scale, mis aitab skaleerida pilves kasutatavaid rakendusi horisontaalselt. Amazon pilve kasutatakse antud töös MediaWiki seadistamiseks ja suuremahuliste eksperimentide rakendamiseks. Vajalik on teha palju optimiseerimisi ja seadistamisi, et suurendada teenuse läbilaske võimsust. Antud töö raames loodud raamistik aitab mõõta serverite kasutust, kogudes

andmeid protsessori, mälu ja võrgu kasutamise kohta. See aitab leida süsteemis olevaid kitsaskohti, mis võivad põhjustada süsteemi olulist aeglustumist.

Antud töö raames sai tehtud erinevaid teste, et selgitada välja parim võimalik paigutus ja seadistus. Saavutatud seadistust kontrolliti hiljem 2 suuremahulise eksperimentiga, mis kestis üks päev ja mille käigus tekitati 22 miljonit päringut, leidmaks kuidas raamistik võimaldab teenust pilves skaleerida ülesse päringute arvu tõusmisel ja vähendada servereid, kui päringute arv väheneb. Ühes eksperimendis kasutati optimaalset heuristikat, et selgitada välja optimaalne serverite arv, mida on vaja pilves rakendada. Teine eksperimentidest kasutas Amazon Auto Scale teenust, mis kasutas serverite keskmist protsessori kasutamist, et selgitada välja, kas pilves on vaja servereid lisada või eemaldada. Antud eksperimendid näitavad selgelt, et kasutades dünaamilist arvu servereid, olenevalt päringute arvust, on võimalik teenuse üleval hoidmiseks säästa raha.

Bibliography

[1] *Scientific Computing on the Cloud (SciCloud) - Distributed Systems Group site*;
http://ds.cs.ut.ee/research/scicloud;
last viewed 2. December 2011

[2] Satish Srirama, Oleg Batrashev, Eero Vainikko; *SciCloud: Scientific Computing on the Cloud*; 10th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2010), May 17-20, 2010, pp. 579. IEEE Computer Society.

[3] *Cloud IaaS — Cloud Computing Software from Eucalyptus*;
http://www.eucalyptus.com/products/eee;
last viewed 2. December 2011

[4] *Manual:File cache - MediaWiki*;
http://www.mediawiki.org/wiki/Manual:File_cache;
last viewed 2. December 2011

[5] *memcached - a distributed memory object caching system*;
http://memcached.org/;
last viewed 2. December 2011

[6] *October 2011 Web Server Survy*;
http://news.netcraft.com/archives/2011/10/06/ october-2011-web-server-survey.html;
last viewed 2. December 2011

[7] *HTTPUpstreamModule*;
http://wiki.nginx.org/HttpUpstreamModule#upstream;
last viewed 2. December 2011

[8] Dipankar Sarkar; *Nginx 1 Web Server Implementation Cookbook*; PACKT publishing, 1st Edition, 2011, Pages: 192, ISBN 978-1-849514-96-5.

[9] *Auto Scaling*;
http://aws.amazon.com/autoscaling/;
last viewed 19. January 2012

[10] *Amazon EC2 Instance Types*;
http://aws.amazon.com/ec2/instance-types/;
last viewed 26. January 2012

[11] *Percona - Documentation - The tcprstat User's Manual*;
http://www.percona.com/docs/wiki/tcprstat:start;
last viewed 26. January 2012

[12] *Auto Scaling Command Line Tool : Developer Tools : Amazon Web Services*;
http://aws.amazon.com/developertools/2535;
last viewed 31. January 2012

[13] *Twenty-One Experts Define Cloud Computing*;
http://cloudcomputing.sys-con.com/node/612375/print;
last viewed 09. February 2012

[14] Foster, I.; Yong Zhao; Raicu, I.; Lu, S.; *Cloud Computing and Grid Computing 360-Degree Compared*; Grid Computing Environments Workshop, 2008. GCE '08, Pages: 1-10, 12-16 Nov. 2008

[15] *LAMP - (software bundle) - Wikipedia, the free encyclopedia*;
http://en.wikipedia.org/wiki/LAMP_(software_bundle);
last viewed 09. February 2012

[16] *PHP: Description of core php.ini directives - Manual*;
http://www.php.net/manual/en/ini.core.php#ini.memory-limit;
last viewed 09. February 2012

[17] *Linux Increase The Maximum Number Of Open Files / File Descriptors (FD)*;
http://www.cyberciti.biz/faq/linux-increase-the-maximum-number-of-open-files/;
last viewed 09. February 2012

[18] *ab - Apache HTTP server benchmarking tool - Apache HTTP server*;
http://httpd.apache.org/docs/2.0/programs/ab.html;
last viewed 09. February 2012

[19] *Amazon Elastic Compute Cloud (Amazon EC2)*;
http://aws.amazon.com/ec2/;
last viewed 02. March 2012

[20] *Amazon EC2 Pricing*;
http://aws.amazon.com/ec2/pricing/;
last viewed 09. February 2012

[21] *Hill climbing - Wikipedia, the free encyclopedia*;
http://en.wikipedia.org/wiki/Hill_climbing;
last viewed 14. February 2012

[22] *The CLOUDS Lab: Flagship Projects - Gridbus and Cloudbus*;
http://www.cloudbus.org/cloudsim/;
last viewed 14. February 2012

[23] Rajkumar Buyya, Rajiv Ranjan, Rodrigo N. Calheiros; *Modeling and Simulation of Scalable Cloud Computing Environments and the CloudSim Toolkit: Challenges and Opportunities*; High Performance Computing & Simulation, 2009. HPCS '09. International Conference on, Pages: 1-11, 21-24 June 2009

[24] *[mediawiki] Contents of /trunk/phase3/docs/memcached.txt*;v
http://svn.wikimedia.org/viewvc/mediawiki/trunk/
phase3/docs/memcached.txt?view=markup;
last viewed 25. February 2012

[25] *XCache*; http://xcache.lighttpd.net/;
last viewed 25. February 2012

[26] *EC2StartersGuide - Community Ubuntu Documentation*;
https://help.ubuntu.com/community/EC2StartersGuide;
last viewed 25. February 2012

[27] *Repositories/CommandLine - Community Ubuntu Documentation*;
https://help.ubuntu.com/community/Repositories/CommandLine
#Adding_the_Universe_and_Multiverse_Repositories;
last viewed 25. February 2012

[28] *Wikimedia Technical & Operational Infrastructure - A high level overview of Wikimedia Operations*;
http://upload.wikimedia.org/wikipedia/commons/3/33/Rob_Halsell_-
_Wikimania_2009 _-_Wikimedia_Operations_%26_Technical_Overview.pdf;
last viewed 27. February 2012

[29] *Man Page - Wikipedia, the free encyclopedia*;
http://en.wikipedia.org/wiki/Manpage;
last viewed 27. February 2012

[30] *Apt-Get*;
https://help.ubuntu.com/8.04/serverguide/C/apt-get.html;
last viewed 27. February 2012

[31] *Interview with creator of NGINX Igor Sysoev — Web Hosting Skills*;
http://www.webhostingskills.com/open_source/articles/interview_with_creator_of_nginx_igor_sysoev;
last viewed 28. February 2012

[32] *osi 7 layer model*;
http://www.escotal.com/osilayer.html;
last viewed 28. February 2012

[33] *RRDtool - About RRDtool*;
http://oss.oetiker.ch/rrdtool/;
last viewed 1. March 2012

[34] *Elastic Load Balancing*;
http://aws.amazon.com/elasticloadbalancing/;
last viewed 2. March 2012

[35] *Amazon Simple Storage Service (Amazon S3)*;
http://aws.amazon.com/s3/;
last viewed 9. March 2012

[36] Amazon EC2 Spot Instances;
http://aws.amazon.com/ec2/spot-instances/;
last viewed 2. March 2012

[37] Erik-Jan van Baare; *WikiBench: A distributed, Wikipedia based web application benchmark*; Master thesis, VU University Amsterdam, May 2009.

[38] Nurmi, D., Wolski, R., Grzegorczyk, C., Obertelli, G., Soman, S., Youseff, L., Zagorodnov, D.; *The Eucalyptus Open-source Cloud-computing System*; Cluster Computing and the Grid, 2009. CCGRID '09. 9th IEEE/ACM International Symposium on. 18-21 May 2009. Pages: 124 - 131.

[39] *TimingEvents - haproxy-docs - Timing events - HAProxy Documentation*;
http://siag.nu/pen/;
last viewed 16. March 2012

[40] *Virtual Router Redundancy Protocol - Wikipedia, the free encyclopedia*;
http://en.wikipedia.org/wiki/Virtual_Router_Redundancy_Protocol;
last viewed 16. March 2012

[41] *Software as a service - Wikipedia, the free encyclopedia*;
http://en.wikipedia.org/wiki/Software_as_a_service;
last viewed 16. March 2012

[42] *Platform as a service - Wikipedia, the free encyclopedia*;
http://en.wikipedia.org/wiki/Platform_as_a_service;
last viewed 16. March 2012

[43] *Infrastructure as a service - Wikipedia, the free encyclopedia*;
http://en.wikipedia.org/wiki/Infrastructure_as_a_service;
last viewed 16. March 2012

[44] Will Sobel, Shanti Subramanyam, Akara Sucharitakul, Jimmy Nguyen, Hubert Wong, Arthur Klepchukov, Sheetal Patil, Armando Fox, David Patterson; *Cloudstone: Multi-Platform, Multi-Language Benchmark and Measurement Tools for Web 2.0*; 2008.

[45] Chuck McAuley; *Watch Out for That Hockey Stick: The Need for Network Device Product Evaluations — BreakingPoint*;
http://www.breakingpointsystems.com/community/blog/network-device-product-evaluations/;
last viewed 22. March 2012

[46] Daniel A. Menascé; *Load Testing of Web Sites*;
http://cs.gmu.edu/ menasce/papers/IEEE-IC-LoadTesting-July-2002.pdf;
last viewed 22. March 2012

[47] Byung Chul Tak, Bhuvan Urgaonkar, Anand Sivasubramaniam; *To Move or Not to Move: The Economics of Cloud Computing*; USENIX HotCloud'11 Conference, Portland, USA, June 14–17, 2011..

[48] *What is 'steal time' in my sysstat output? — Racker Hacker*;
http://rackerhacker.com/2008/11/04/what-is-steal-time-in-my-sysstat-output/;
last viewed 22. March 2012

[49] Nicolai Wadstrom; *Why Amazon EC2 CPU Steal Time does not work (well) for Web apps — Nicolai Wadstrom*; http://nicolaiwadstrom.com/blog/2012/01/03/why-amazon-ec2-steal-cpu-does-not-work-for-web-apps/; last viewed 22. March 2012

[50] Huan Liu, Sewook Wee; *Web Server Farm in the Cloud: Performance Evaluation and Dynamic Architecture*; Lecture Notes in Computer Science, 2009, Volume 5931/2009, 369-380, DOI: 10.1007/978-3-642-10665-1_34

[51] *Scalr - Cloud Management Software*; http://scalr.net/; last viewed 5. April 2012

[52] *Cloud Comptuing Management Platform by RightScale*; http://www.rightscale.com/; last viewed 5. April 2012

[53] Rafael Moreno-Vozmediano, Ruben S. Montero, Ignacio M. Llorent; *Elastic management of web server clusters on distributed virtual infrastructure*; Volume 23, Issue 13, Article first published online: 14 FEB 2011;

[54] Mike Kirkwood; *Enterprise Cloud Control: Q&A with Eucalyptus CTO Dr. Rich Wolski*; http://www.readwriteweb.com/cloud/2010/03/eucalyptus-amazon-vmware.php; last viewed 10. April 2012

[55] *Erlang (unit) - Wikipedia, the free encyclopedia*; http://en.wikipedia.org/wiki/Erlang_(unit); last viewed 27. February 2012

[56] *Mathematical Programming Glossary*; http://glossary.computing.society.informs.org/second.php?page=U.html; last viewed 08. May 2012

[57] S. N. Srirama, O. Batrashev, P. Jakovits, E. Vainikko; *Scalability of Parallel Scientific Applications on the Cloud*; Scientific Programming Journal, Special Issue on Science-driven Cloud Computing, 19(2-3):91-105, 2011.
IOS Press. DOI 10.3233/SPR-2011-0320.

Appendix

All the necessary instructions, commands and scripts are located on the DVD.