UNIVERSITY OF TARTU

FACULTY OF MATHEMATICS AND COMPUTER SCIENCE

Institute of Computer Science

Information Technology

Timur Hassanov

Web Applications Scaling in Amazon Cloud

B.Sc. Thesis (6 ECTS)

Supervisor: Satish Narayana Srirama, Ph.D.

TARTU 2014

Web Applications Scaling in Amazon Cloud

Abstract:

This thesis gives the review of server-side scaling options for Linux-running instances in Amazon Web Services. The application used in terms of the thesis is MediaWiki, which is written in PHP programming language and uses MySQL relational database server.

Keywords:

AWS, Amazon, scaling, auto scaling, EC2, Memcached, PHP, Apache HTTP Server, lighttpd, tsung, LAMP, MediaWiki, OPcache, MySQL, load balancing, ELB

Veebirakenduste skaleerimine Amazon Cloud'is

Lühikokkuvõte:

See bakalaureusetöö annab ülevaate serveripoolsetest skaleerimisvõimalustest Amazon Web Services Linuxi instantsidel. Töö subjektis on valitud MediaWiki, mis on kirjutatud PHP programmeerimiskeeles ning kasutab MySQL relatsioonilist andmebaasi.

Võtmesõnad:

AWS, Amazon, skaleerimine, autoskaleerimine, EC2, Memcached, PHP, Apache HTTP Server, lighttpd, tsung, LAMP, MediaWiki, OPcache, MySQL, koormuse tasakaalustus, ELB

Table of Contents

TABLE OF CONTENTS **3**

LIST OF TABLES **5**

LIST OF FIGURES **5**

INTRODUCTION **6**

2. ENVIRONMENT CONFIGURATION **9**

2.1 SERVER-SIDE SOFTWARE 9

2.1.1 MediaWiki 9

2.1.2 Ubuntu 9

2.1.3 Apache HTTP Server 10

2.1.4 lighttpd 10

2.1.5 PHP 11

2.1.6 OPcache 11

2.1.7 Memcached 11

2.1.8 MySQL 12

2.1.9 Tsung 12

2.2 SERVER CONFIGURATION 12

2.2.1 Web Server Instance 12

2.2.2 Database Server Instance 13

2.2.3 MediaWiki Setup 13

2.2.4 Tsung Setup 14

3. LOAD TESTING IN DIFFERENT ENVIRONMENTS **16**

3.1 SINGLE WEB SERVER INSTANCE 16

3.1.1 Apache HTTP Server vs lighttpd 16

3.1.2 Apache HTTP Server with Different Caching Methods 18

3.2 MULTIPLE WEB SERVER INSTANCES 21

3.2.1 Amazon Elastic Load Balancer 21

3.3 AUTO SCALING 25

3.3.1 Auto Scaling Review 25

3.3.2 The Logic of Auto Scaling 26

3.3.3 Auto Scaling Settings 26

3.3.4 Command Line Tools Set Up *27*
3.3.5 Auto Scaling Setup *28*
3.3.6 Configuration Layout *31*
3.3.7 Test 7. OPcache + full Memcached. Variable interarrival *31*
CONCLUSION **34**
BIBLIOGRAPHY **36**
APPENDIX **39**
SUPPLEMENTARY MATERIAL 39
LICENSE 40

List of Tables

Table 1. Apache HTTP Server vs lighttpd. No caching. Interarrival = 0.3/0.2....... 17

Table 2. Apache HTTP Server vs lighttpd. APC caching. Interarrival = 0.2.......... 18

Table 3. Apache HTTP Server. OPcache + Memcached. Interarrival = 0.2 19

Table 4. Apache HTTP Server. OPcache + Memcached. Interarrival = 0.14 20

Table 5. Test 6 results.. 23

Table 6. Test 7: session mean time.. 32

List of Figures

Figure 1. Test 3: Memcached cache size growth 19

Figure 2. Test 5: CPU load.. 22

Figure 3. Test 5: Mean session time 22

Figure 4. Test 6 results: CPU load 244

Figure 5. Configuration layout 31

Figure 6. Test 7: session mean time 32

Figure 7. Test 7: random pages per second and instances count........................ 33

Introduction

World Wide Web plays significant role in life of every modern human being. People use WWW for entertainment, business, studying, sharing information, etc. On one side there are end users - the ones who use services, on other developers and companies who create these services. According to Qmee (1) more than 570 websites are created in a minute Worldwide. All these websites need to be hosted, that is why not only the number of hosting service providers grows, but also the quality and quantity of services they provide increases.

Cloud hosting and cloud computing have become very popular lately. One of the most well-known and successful cloud computing service providers is Amazon with its Amazon Web Services. Launched in July 2002 (2), it has become the one of the biggest cloud services providers (3) on the market.

Amazon Web Services have user-friendly GUI for most of the available features and well-documented command-line API for advanced features which, makes it easy to use. Today anyone with sufficient knowledge of IT can quickly and easily set up servers and other Amazon Web Services depending on their own needs.

One of the most important features of Amazon Web Services is Auto Scaling. Imagine that company Qwerty runs a small web service with very limited number of users. Qwerty's server is more than capable of handling the load and costs are very low: Micro instance in EU region will cost $0.02 per hour (4). But attention to Qwerty's service grows and Qwerty needs the more powerful server. It is very easy to switch between different types of servers in Amazon Web Services. Then as Qwerty's service grows more one server (with even higher performance, as c1.medium) is not enough anymore, so Qwerty needs to scale. It is also relatively easy to create as many servers on Amazon Web Services as Qwerty wants to and if Qwerty's team has a decent system administrator he will be able to setup them to run together.

Assume that most of the users of Qwerty service are from Estonia. That means the highest activity on servers will be when it is daytime in Estonia. Assume, that Qwerty clients are businesses, so the activity peak will be between 8:00 and 18:00. Company runs several servers that handle the peak load without any

problems, but when working time ends servers don't run on their limits or even close to them. So 14 hours per day and during the whole weekend there is no need for that large amount of servers, but Qwerty still pays for them. The easiest solution would be to start *x* number of the servers at 8:00 and turn them off at 18:00. But what happens if there is a story about Qwerty service in Friday night television programme? A lot of potential customers would want to test the service, but servers can not handle this significant load (Qwerty turned most of them off for the weekend) and won't be able to handle it till Monday morning. So Monday morning not only regular customers will want to interact with the service, but also a large amount of new potential clients; servers fail again - their capacity is just not enough to handle the load.

This is where Amazon Web Services Auto Scaling mechanisms become useful. Auto scaling has to be set up according to some metrics that Amazon Web Services provide. This means, it is possible to set up that if one of the metrics (CPU utilization, latency, memory usage, transaction volumes, error rates, etc.) reaches some value (for example "CPU utilization is more than 70% for the last 3 minutes"), Amazon Web Services will run an extra amount of servers for you automatically, and when the load drops it will terminate excessive resources.

This approach will not only help Qwerty maintain the stability of their service, but also to cut their costs significantly.

One of the goals of current thesis is try to find the good server software configuration (Apache versus lighttpd, OPcache, memcached, etc) to run a specific web application. Another goal is to show how to set up Amazon Auto Scaling mechanisms and prove that they are efficient.

The web application, which will be tested in different environments, is MediaWiki - a free software open source wiki package written in PHP, originally for use on Wikipedia. It is now used by several other projects of the non-profit Wikimedia Foundation and by many other wikis. MediaWiki is one of the most popular applications for running a personal wiki. It is very powerful, therefore very demanding on resources. The scalability of MediaWiki makes it an ideal candidate for the research in terms of this thesis.

The thesis is organized as follows: in chapter 2 we will define the environment configuration for our tests, in chapter 3 we will perform the load tests using different configurations and review the results.

2. Environment Configuration

In the following chapter we will review the server-side software and server configuration for the future tests.

2.1 Server-side Software

In this chapter we will review the different software packages that will be used in terms of this thesis.

2.1.1 MediaWiki

MediaWiki is a free web-based wiki software application. Developed by the Wikimedia Foundation and others, it is used to run all of its projects, including Wikipedia, Wiktionary and Wikinews. Numerous other wikis around the world also use it to power their websites. It is written in the PHP programming language and uses a back-end database. (5)

The main reason why MediaWiki is used in terms of current thesis as the main application is its scalability: because it is used to run one of the highest-traffic sites on the Web, Wikipedia, MediaWiki performance and scalability have been highly optimized (6). Alexa (7) estimates Wikipedia.org to be 6th highest traffic website all over the World, which confirms that the application is strongly scalable not only vertically, but also horizontally. Another reason is the complexity of MediaWiki code, which makes rendering of a page a CPU-intensive task (8). Third reason is that MediaWiki is a widespread application that is used not only by Wikimedia Foundation, but also by such well-known websites as wikiHow, AboutUs.org, Mahalo.com, WikiLeaks and others. Anyone can run their own Wiki using MediaWiki.

MediaWiki 1.22.6 stable release is used in terms of current thesis, it was released on April 24th, 2014.

2.1.2 Ubuntu

Ubuntu is a computer operating system based on the Debian Linux distribution and distributed as free and open source software.

Ubuntu was chosen because Wikipedia and other projects by Wikimedia Foundation run Ubuntu on their servers (9).

Ubuntu 14.04 LTS (Trusty Tahr) Server i386 stable release is used in terms of current thesis, it was released on April 17th, 2014.

2.1.3 Apache HTTP Server

The Apache HTTP Server Project is a collaborative software development effort aimed at creating a robust, commercial-grade, featureful, and freely-available source code implementation of an HTTP (Web) server. (10) The project managed by a group of volunteers located all over the world, using the Internet to communicate, plan and develop the server and its related documentation. Apache HTTP Server project is part of the Apache Software Foundation. In addition, hundreds of users have contributed ideas, code, and documentation to the project. Apache HTTP Server celebrated its 19th birthday as a project on February 2014.

Apache HTTP Server is the most widespread web server in the world. Even though Apache HTTP Server market share is gradually reducing, it remains the biggest player on the market: by the April of 2014 Apache HTTP Server has total estimated share of 52.44%. Apache HTTP Server's closest rival nginx shares about 14.22% of the market. Apache HTTP Server has also the largest share across high loaded websites. Netcraft estimates Apache HTTP Server's share across the million busiest sites to be around 53.44%. nginx shares less than 18% of this market. (11)

Wikipedia itself runs on Apache HTTP Server. (10)

Apache HTTP Server 2.4.9 stable release is used in terms of current thesis, it was released on March 17th, 2014.

2.1.4 lighttpd

lighttpd is an open-source web server, that "is a secure, fast, compliant, and very flexible web-server that has been optimized for high-performance environments. It has a very low memory footprint compared to other web servers and takes care of cpu-load. Its advanced feature-set (FastCGI, CGI, Auth, Output-Compression,

URL-Rewriting and many more) make lighttpd the perfect webserver-software for every server that suffers load problems.” (12)

lighttpd is used by around 0.4% of 10000 most popular websites. (13)

lighttpd was chosen as a comparison to Apache HTTP Server in terms of performance.

lighttpd 1.4.35 stable release is used in terms of current thesis, it was released on March 12, 2014

2.1.5 PHP

PHP is a server-side HTML embedded scripting language, MediaWiki is written in PHP.

PHP 5.5.12 stable release is used in terms of current thesis, it was released on April 30th, 2014.

2.1.6 OPcache

OPcache improves PHP performance by storing precompiled script bytecode in shared memory, thereby removing the need for PHP to load and parse scripts on each request, greatly reducing the amount of time needed to run a script multiple times. (14) MediaWiki supports OPcache.

2.1.7 Memcached

Free & open source, high-performance, distributed memory object caching system, generic in nature, but intended for use in speeding up dynamic web applications by alleviating database load. Memcached is an in-memory key-value store for small chunks of arbitrary data (strings, objects) from results of database calls, API calls, or page rendering. (15)

Memcached was chosen because Wikipedia and other projects by Wikimedia Foundation utilize Memcached for their purposes. (9)

Amazon Web Services include Amazon ElastiCache service, which allows to run Memcached server fairly easily. Amazon ElastiCache is protocol-compliant with Memcached (16) and runs Memcached 1.4.

Small Cache Node (1.3 GB memory, 1 EC2 compute unit, 64-bit platform, moderate I/O capacity) was used in terms of the thesis.

2.1.8 MySQL

MySQL is the world's most popular server open source database. (17)

MySQL was chosen because Wikipedia and other projects by Wikimedia Foundation run MySQL on their database servers. (9)

MySQL 5.6.18 is used in terms of current thesis, it was released on April 11, 2014.

MySQL runs on a dedicated server (Ubuntu 14.04) as MediaWiki manuals advise as a logical step of performance optimization. (18) Database server runs general purpose AWS instance m3.medium.

2.1.9 Tsung

Tsung is an open-source multi-protocol distributed load testing tool. The purpose of Tsung is to simulate users in order to test the scalability and performance of IP based client/server applications. The main reason to select Tsung as a load testing tool over more user-friendly GUI tools as Apache JMeter, is Tsung high performance. (19)

Tsung is used to run load and stress testing of web servers.

Tsung 1.4.2 is used in terms of current thesis, it was released on January 4th, 2012.

tsung_stats.pl script is used to generate reports. tsung_stats.pl script is included into Tsung package.

2.2 Server Configuration

In this chapter we will review the configuration of server side software.

2.2.1 Web Server Instance

Research shows that MediaWiki is quite demanding on web server resources, especially CPU time. Therefore c3.xlarge instance type was selected. According to Amazon (20) c3.xlarge instance types have the following virtual configuration:

- 7.5 GiB of memory
- 14 EC2 Compute Units (4 virtual cores)
- 32-bit or 64-bit platform
- Moderate network performance

The OS selected for the web server instance is Ubuntu Server 14.04 LTS (PV) 64-bit (ami-018c9568).

We decided to perform only horizontal scaling as simplier and more reasonable solution (compared to vertical), especially in conjuction with load balancing. (19)

2.2.2 Database Server Instance

Research shows that MediaWiki is not very demanding to resources of the database server, so one database server instance running on m3.medium type instance should be able to cope with the small load. However, in order to have more consistent results throughout all tests we selected m3.large instance for database server. Further tests will show that current configuration successfully copes with high load from 5 web servers.

2.2.3 MediaWiki Setup

In this chapter we will provide the guides for installing MediaWiki and importing database dump into it.

2.2.3.1 MediaWiki Installation

The installation guide can be found on MediaWiki website. (21) Here is the short review of installation process:

```
>wget http://releases.wikimedia.org/mediawiki/1.22/mediawiki-1.22.6.tar.gz
>gunzip mediawiki-1.22.6.tar.gz
>tar -xvf mediawiki-1.22.6.tar
```

Now we need to set up MediaWiki database. We open http://server-ip/mediawiki-1.22.6/mw-config/index.php. Database setup is very easy and straightforward. We

need to leave the database prefix empty in order to make import of SQL dumps easier.

2.2.3.2 MediaWiki Dumps Import

We need to import a database, so that the tested application would act as a real application with real data. The easiest way to do so is to import a Wikipedia dump from official source. (22) The dump we chose is etwiki, all articles from Estonian Wikipedia at current state. (23) It contains a fairly large amount of articles (around 316000). Significantly larger number of articles, as the whole English Wiki, for example, would mean excessive load on database server. Small amount would mean that the database load is too little.

Dumps are in XML format, so in order to import them they needed to be converted into MySQL queries. We are using mwdumper.jar (comes with WikiBench: http://www.wikibench.eu).

```
>wget http://dumps.wikimedia.org/etwiki/20140427/etwiki-20140427-pages-meta-current.xml.bz2
>java -jar mwdumper.jar --format=sql:1.5 etwiki-20140427-pages-meta-current.xml.bz2 | mysql -u username -p databasename
```

We need to make sure that `databasename` is an existing database and `username` has access to it. Our import was successful: 316149 pages were imported, database size is ≈1041MB. If dump is not imported correctly, truncating the table "page" (contains only one entry) may help. To test if the database was imported we tried to load random page: http://server-ip/mediawiki-1.22.6/index.php/Special:RandomPage which returned page titled "Köie tänav" from Estonian Wikipedia.

2.2.4 Tsung Setup

Another server will be used to run load tests with Tsung. In order to avoid bandwidth bottlenecks and decrease cost of traffic, this server will run in Amazon Cloud, so traffic will be considered to be local. We will run Tsung on same AMI as other instances: ami-018c9568. Tsung is not demanding on resources, so m3.medium instance type should be sufficient.

To install Tsung on Ubuntu we run:

```
>aptitude install tsung
```

Tsung has an option to monitor tested server(s) using different methods: erlang scripts, munin-node, snmp. In order to set up monitoring with erlang scripts, monitored computers need to be accessible through the network and erlang communications must be allowed. SSH needs to be configured to allow connection without password. The same version of Erlang/OTP must be used on all nodes, otherwise it may not work properly. Package erlang-os-mon must be installed on monitored servers.

```
>aptitude install erlang-os-mon
```

The following Tsung configuration XML will be used for the the initial tests:

```
<?xml version="1.0" ?>
<!DOCTYPE tsung SYSTEM "/usr/share/tsung/tsung-1.0.dtd">
<tsung loglevel="notice">
    <clients>
        <client host="localhost" use_controller_vm="true" maxusers="3000"/>
    </clients>
    <servers>
        <server host="main-server" port="80" type="tcp"></server>
    </servers>
    <monitoring>
        <monitor host="main-server" type="erlang" />
    </monitoring>
    <load>
        <arrivalphase phase="1" duration="30" unit="minute">
            <users interarrival="0.3" unit="second"/>
        </arrivalphase>
    </load>
    <sessions>
        <session name="default" probability="100" type="ts_http">
            <request>
                <dyn_variable name="redirect" re="Location: (http://.*)\r"/>
                <http url="index.php/Special:Random" method="GET" ></http>
            </request>
            <request subst="true">
                <http url="%%_redirect%%" method="GET"></http>
            </request>
        </session>
    </sessions>
</tsung>
```

Tsung can be run from multiple clients using Erlang communication, but currently we use only 1 client server. The <monitoring> tags allow us to define the method for monitoring servers. As we decided before it is “erlang”. Current load settings mean, that we have 1 phase of testing which duration is 30 minutes. For example, our website receives approximately 12000 visitors per hour, the interval between visits (interarrival) is 3600/12000 = 0.3 seconds. In order to avoid caching and make the tests more realistic, we will be loading a random page. Tsung will redirect itself to the new location.

3. Load Testing in Different Environments

In this chapter we will review different practices to set up server side software and perform load testing of MediaWiki in different environments.

3.1 Single Web Server Instance

In this chapter we will try different settings and run the tests using one web server instance.

3.1.1 Apache HTTP Server vs lighttpd

The first set of tests will help to determine if there is a significant performance difference between Apache HTTP Server and lighttpd on our environment

3.1.1.1 Test 1. Single instance. No caching

Apache HTTP Server runs using the default configuration on a single instance. Tsung configuration is the same as defined above (30 minutes, interval between user visits is 0.3 seconds or 0.2 seconds). Apache and mysql services are restarted before the test to ensure nothing was cached.

lighttpd with PHP+FastCGI runs using the default configuration on a single instance. Tsung configuration is the same as in previous test except for loaded url. The new value is:

```
<http url="/mediawiki/mediawiki-1.18.0/index.php?title=Special:Random"
method="GET" ></http>
```

lighttpd and mysql services are restarted before the test to ensure nothing was cached.

Access log was turned off for both Apache HTTP Server and lighttpd.

3.1.1.1.1 Results

Web server	Test duration	interarrival (sec)	session mean (msec)	CPU mean (%)	Free memory mean (MB)
Apache	30 min	0.3	962	63.59	6767
lighttpd	30 min	0.3	1170	67.73	7225
Apache	30 min	0.2	19922	93.3	4985
lighttpd	30 min	0.2	41642	91.7	7215

Table 1. Apache HTTP Server vs lighttpd. No caching. Interarrival = 0.3/0.2

As it is seen from the Table 1 Apache HTTP Server and lighttpd cope with current load (interval between users 0.3 sec, ≈12000 random page requests per hour) almost identically: mean session time by Apache HTTP Server was 208 msec lower, its mean memory consumption was 458 MB higher and CPU load 4% lower than results shown by lighttpd. Test shows that Apache HTTP Server performs better in this case. However, the average user most likely will not notice the difference.

Results also show, that both configurations could not cope with higher load (interval between users 0.2 seconds, ≈18000 random page requests per hour). lighttpd was less stable, producing significantly worse results: mean session time by Apache HTTP Server was almost twice lower, its mean memory consumption was 2230 MB higher and CPU load 1.6% higher than results shown by lighttpd.

Results show that both Apache HTTP Server and lighttpd servers with default configurations, without using any caching methods, can serve between 12000 to 18000 requests of random page per hour, which equals from 288000 to 432000 random pages per 24 hours using our server configurations.

3.1.1.2 Test 2. Single instance. OPcache

MediaWiki manual tells that Opcode caches “greatly reduce the amount of time needed to run a script multiple times” (18), that is why we decided to run the next test using the load, that serves without Opcache could not cope with: interval between users is 0.2 seconds.

3.1.1.2.1 Results

Web server	Test duration	interarrival (sec)	session mean (msec)	CPU mean (%)	Free memory mean (MB)
Apache + OPcache	30 min	0.2	625	62.3	7091
lighttpd + OPcache	30 min	0.2	569	56.9	7271

Table 2. Apache HTTP Server vs lighttpd. OPcache. Interarrival = 0.2

Current test results show that using OPcache significantly increases the performance of MediaWiki. It is difficult to compare these results with corresponding load from previous test (because servers could not cope with the load), but it is clearly seen, that the server with the OPcache extension, shows better results than one without it. That proves that there is no reason not to use OPcache extension with MediaWiki. All following tests will be done with the OPcache extension enabled. Results also show that using OPcache for both Apache HTTP Server and lighttpd evens out the performance differences between them, therefore all the following tests will be done using only Apache HTTP Server.

3.1.2 Apache HTTP Server with Different Caching Methods

3.1.2.1 Test 3. Single instance. OPcache + Memcached. Interarrival = 0.2

The following test will be run only using Apache HTTP Server. The goal of the test is to determine whether Memcached increases the performance of MediaWiki.

Previous test results show that using web servers using OPcache managed to cope with a load of 18000 random page requests per hour. MediaWiki manual (24)

tells that Memcached helps to increase the performance, so we decided to run the same test (interarrival = 0.2) and compare the results with Memcached and without.

In current test we start with an empty Memcached cache to check whether the amount of cached data influences the performance or not. In order to have more clear results we run the test for 8 hours.

3.1.2.2 Results

Apache + OPcache + Memcached	0-0.5 h	0.5-1 h	1-2 h	2-3 h	3-4 h	4-5 h	5-6 h	6-7 h	7-8 h
session mean (msec)	631	651	543	481	456	422	400	385	386
CPU mean (%)	66.1	67	61.8	58.6	58	55.3	54	52.4	52.8
Free memory mean (MB)	7059	7054	7059	7069	7066	7078	7076	7071	7071

Table 3. Apache HTTP Server. OPcache + empty Memcached. Interarrival = 0.2

Results show that utilization of OPcache and Memcached noticeably improve the performance of MediaWiki. The performance improves with the increase of cached information by Memcached.

Memcached significantly increases the performance, therefore using Memcached with MediaWiki is highly justified.

Memcached cache was empty before the test. In one hour after starting the test the size of cache was ≈92 MiB. After the 8 hours of test it was ≈383 MB, so we can see that cache growth is not linear.

Current test show that using OPcache and Memcached server can serve more than 5 random pages per second, which is more than 432000 random pages per 24 hours with very good performance.

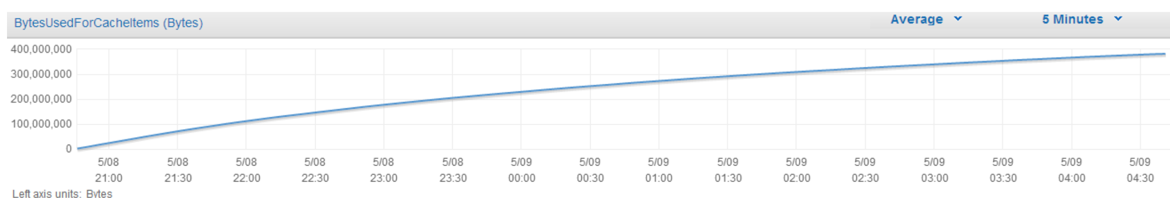

Figure 1. Test 3: Memcached cache size growth

3.1.2.2 Test 4. Single instance. OPcache + Memcached. Interarrival = 0.14

In this test we decided to use the same configuration as before, but decrease the interval between page load to 0.14 seconds. Before the test the Memcache cache size was ≈383 MB.

3.1.2.2.1 Results

Web server	Test duration	interarrival (sec)	session mean (msec)	CPU mean (%)	Free memory mean (MB)
Apache + OPcache + Memcached	60 min	0.14	727	81.9	6985

Table 4. Apache HTTP Server. OPcache + Memcached. Interarrival = 0.14

Current test shows that MediaWiki using OPcache and dedicated Memcached server can serve more than 7 random pages per second, which is more than 617000 pages per 24 hours. Comparison with the results from Test 1 shows that current configuration performs more than 2 times better: it servers more than twice pages more with smaller session mean time.

During the test the size of Memcached cache increased by ≈29 MiB. By the end of the test the size of cache was ≈412 MiB.

3.2 Multiple Web Server Instances

In this chapter we will review different perform load testing of MediaWiki in using several web server instances.

3.2.1 Amazon Elastic Load Balancer

Horizontal scaling consists in transforming a single node setup into a multi-node configuration, in which the load can be distributed using a load balancer. (25)

Elastic Load Balancing automatically distributes incoming application traffic across multiple Amazon EC2 instances. It enables to achieve greater fault tolerance in applications, seamlessly providing the amount of load balancing capacity needed in response to incoming application traffic. (26)

In order to get the best results from the load balancer, it should be managing the set of equivalent instances. If instances do not have the same software versions, application versions or settings, different users may get different versions of the same page. One of the ways to create similar instances is to create an AMI from selected instance (27) and launch the needed number of instances from this AMI.

The next step is creating a load balancer and adding selected instances behind it.

In order to be sure that Memcached cache size is as close to the maximum as possible. in order to get more consistent results, it is reasonable to run the MediaWiki maintenance script for generation of the file cache. This script not only generates the file cache, but also the Memcached cache. After the file cache was built, it was deleted. Total Memcached cache size was ≈708 MiB. The script is run using the following command:

```
>php rebuildFileCache.php 0 overwrite
```

3.2.1.1 Test 5. Two Instances, Single Instance. OPcache + full Memcached. Interarrival = 0.14

For the following test we will create a load balancer and put 2 servers behind it. Theoretically 2 servers should double the performance.

In our case there are 2 instances with identical contents behind the load balancer: the AMI was created from the main server and a new instance of the same type (c3.xlarge) were launched in the same availability zone (us-east-1c).

Test simulates random page request every 0.14 seconds during 1 hour. We start the test with one server behind the load balancer. After approximately 30 minutes, one more instance was added to the load balancer to make sure Load Balancer works properly.

3.2.1.1.1 Results

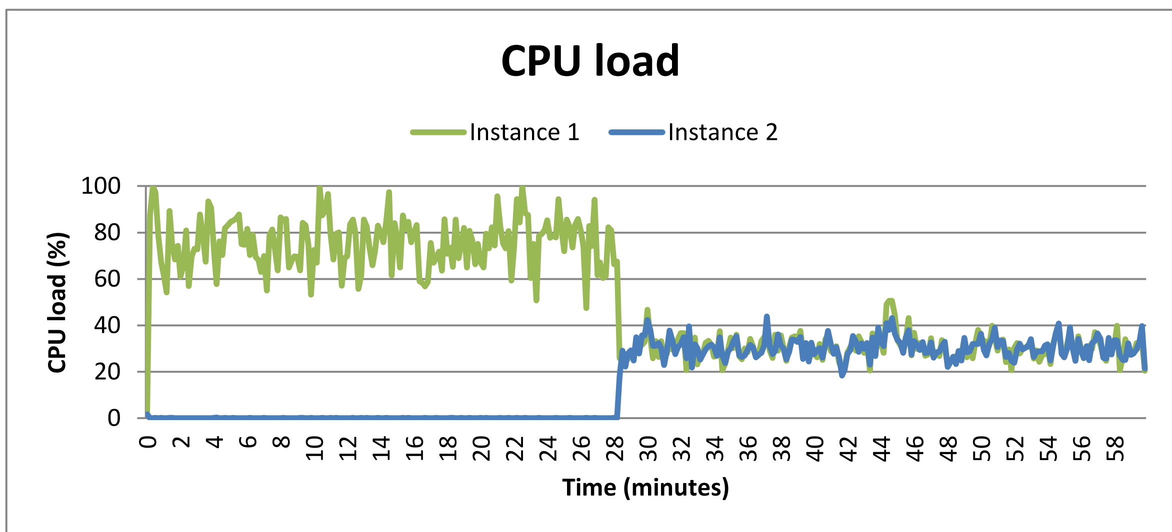

Figure 2. Test 5: CPU load

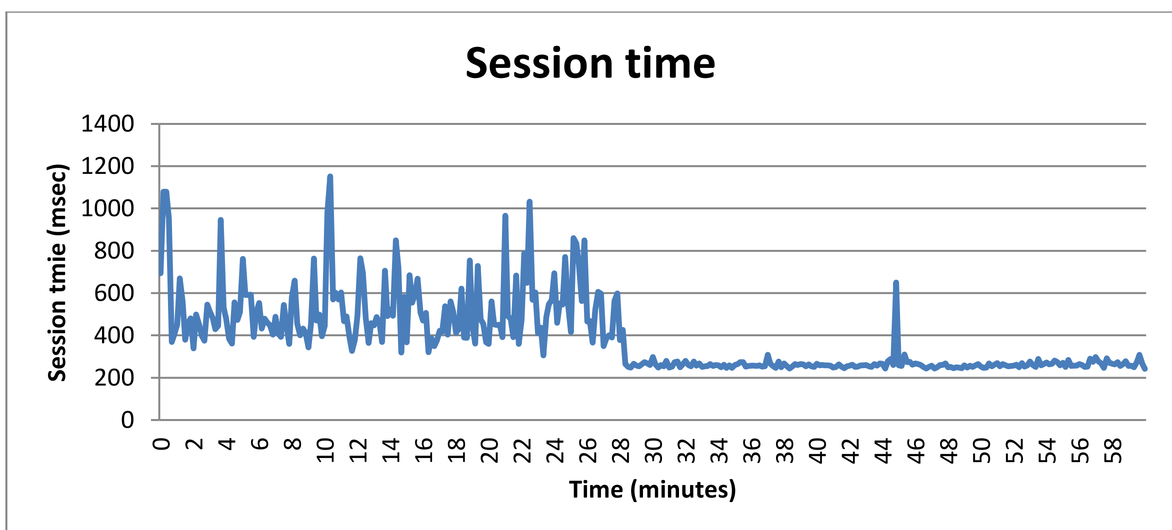

Figure 3. Test 5: Mean session time

Test results show that Elastic Load Balancer is set up properly. During the first phase of the test (1 instance behind the load balancer) average CPU usage of both instances was ≈75% and ≈0% respectively, mean session time was ≈526 msec. During the second phase of the test (2 instances behind the load balancer) average CPU usage of both instances was ≈30% and ≈31% respectively, mean session time was ≈263 msec. Current test shows that current configuration is able to scale almost linear in case of consistent number of requests: two identical instances produce twice less session mean time compared the result with one instance.

3.2.1.2 Test 6. 5 Instances. OPcache + full Memcached. Interarrival = 0.028

In order to check if horizontal scaling is linear and depends on the number of instances behind the load balancer the next test will put 5 equal instances behind the load balancer. All instances were created using the same AMI. All instances are of the same type (c3.xlarge) and launched in the same availability zone (us-east-1c). Test simulates random page request every 0.028 second during 1 hour.

3.2.1.2.1. Results

Web server	Test duration	interarrival (sec)	session mean (msec)	CPU mean (%) (of all instances)	Free memory mean (MB) (of all instances)
(Apache+OPcache)x5 + Memcached	60 min	0.028	338	81	7133

Table 5. Test 6 results

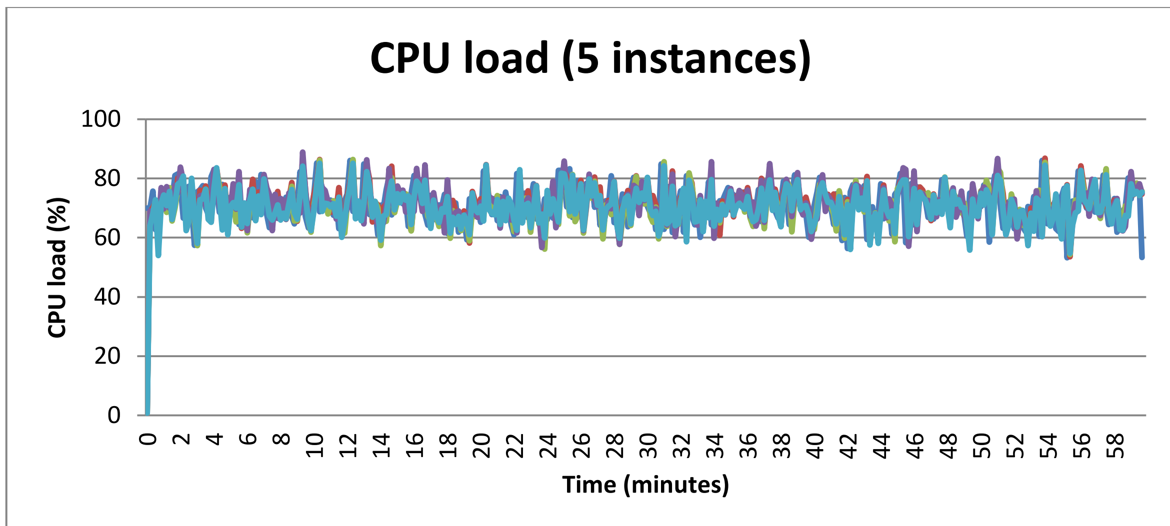

Figure 4. Test 6 results: CPU load

Current test results in comparison to results from the Test 4 show interesting results: we expected session mean time to be very close to the test 3 first phase as we increased amount of instances 5 times and decreased interarrival value 5 times. However, session mean time was 55% smaller. This test shows that distributing load between several instances produces much more consistent and better performance. 5 instances of our configuration, using Memcached, behind load balancer can serve more than 35 random page requests per second, which is more than 3 million random page requests per day.

Figure 4 shows that load balancer distributes the load between instances almost equally: CPU load of all 8 instances were almost the same during the test.

3.3 Auto Scaling

In this chapter we will review the process of setting up Amazon Auto Scaling mechanisms and perform load testing of MediaWiki using Auto Scaling.

3.3.1 Auto Scaling Review

Auto Scaling is a service designed to launch or terminate EC2 instances automatically based on user-defined triggers. Auto Scaling is useful for maintaining a reasonable amount of Amazon EC2 instances that can handle the presented load.

The most important feature of Auto Scaling comes from its name, it responds automatically to changing conditions. We need to specify changes to monitor and how Auto Scaling should respond to those changes. For example, we can setup Auto Scaling to launch 2 additional instances when mean RAM usage exceeds 90 percent during 5 minutes, or to terminate all but one instance for the weekend when traffic is expected to be low.

The following terminology is important when talking about Auto Scaling:

- **Auto Scaling Group.** A set of EC2 instances that represents logical grouping for scaling.
- **Health Check.** Procedures that verify if instance is responding. If not, it may be terminated and new instance will be launched.
- **Launch Configuration.** Represents parameters set up to launch new instances when triggers are fired. For example instance type is a part of Launch Configuration.
- **Alarm**. A CloudWatch alarm is an object that watches over a specified metric. When an alarm changes state it executes action(s) to scale up or down.
- **Policy**. A set of instructions that tells the service how to respond to CloudWatch alarms. We can configure a CloudWatch alarm to send a message to Auto Scaling whenever a specific metric has reached a triggering value. When the alarm sends the message, Auto Scaling executes the associated policy on an Auto Scaling group to scale the group up or down.

- **Trigger**. An object that combines two features: an alarm and a policy. In most cases, we will need two triggers — one trigger for scaling up and another for scaling down.
- **Cooldown**. A period of time after Auto Scaling initiates a scaling activity during which no other scaling activity can take place. A cooldown period allows the effect of a scaling activity to become visible in the metrics that originally triggered the activity. This period is configurable, and gives the system time to perform and adjust to any new scaling activities.

3.3.2 The Logic of Auto Scaling

There are different methods to scale application in AWS. In current thesis we will use the following logic: one instance is run behind the Elastic Load Balancer initially. When defined alarm (will be defined later) changes state - the trigger fires the policy to create a new instance in the current Auto Scaling Group from using the defined Launch Configuration (AMI, type, availability zone, etc) and puts it behind the defined Elastic Load Balancer. When another alarm defines that load has decreased (metrics will be defined later), one instance will be terminated if the number of running instances is higher than 1.

3.3.3 Auto Scaling Settings

One of the issues of autoscaling is to determine CloudWatch alarms. It is impossible to propose an optimal solution that would satisfy all the needs. Different systems have different requirements: if one AWS user is satisfied with his website loading over a second, another would need his website running the same application to be loaded significantly faster. One has more resources to spend on more stable work of his system than the other, so he can run more servers at once. Moreover, even the same application with different database size or different server setup may load much more faster if servers are configured differently (our experiments with OPcache cache turned on and off prove that point).

Also, it is almost never possible to predict the exact amount of the users willing to use one's application at time x: if one's service got a great promotion in media, he

can only predict the possible amount of sudden users, but never can be sure how many instances he will need to successfully cope with that load.

These thoughts lead to a lot of possibilities to experiment with different settings of auto scale setup. Settings used in the following tests will be reviewed further.

3.3.4 Command Line Tools Set Up

In order to set up the Auto Scaling we need to install the Auto Scaling Command Line Tool. (28) This set of scripts is written in Java, so our instance should have Java installed with JAVA_HOME defined. We used openjdk-7-jre package. Tools don't need to be specially installed, just downloaded and unarchived.

```
>wget http://ec2-downloads.s3.amazonaws.com/AutoScaling-2011-01-01.zip
>unzip
```

Next we need to setup the following environment variables (we extracted the archive into /home/developer/aws, bash shell is being used):

```
>export AWS_AUTO_SCALING_HOME=/home/developer/aws/AutoScaling-1.0.61.5
>export PATH=$PATH:$AWS_AUTO_SCALING_HOME/bin
```

In order to authentificate we need to get Access Key ID and Secret Access Key from https://aws-portal.amazon.com/gp/aws/securityCredentials. Then we create a file with the following contents:

```
AWSAccessKeyId=<Write your AWS access ID>
AWSSecretKey=<Write your AWS secret key>
```

Where we define these 2 variables. Next we need to change file permissions and set up one more environment variable.

```
>chmod 600 filename
>export AWS_CREDENTIAL_FILE=filename
```

`filename` should be path to your saved configuration file.

In the next step we need to define the Auto Scaling tools region. The list of regions with endpoints is listed in Amazon Web Services Glossary (29).

```
>export AWS_AUTO_SCALING_URL=https://autoscaling.us-east-1.amazonaws.com
```

To check if Auto Scaling Command Line Tool is installed correctly, try to run `as-version`.

Next we need to install Elastic Load Balancing API Tools for managing Elastic Load Balancers and Amazon CloudWatch Command Line Tool for utilization of CloudWatch and set up the following environment variables:

```
>wget http://ec2-downloads.s3.amazonaws.com/ElasticLoadBalancing.zip
>unzip ElasticLoadBalancing.zip
>wget http://ec2-downloads.s3.amazonaws.com/CloudWatch-2010-08-01.zip
>unzip CloudWatch-2010-08-01.zip
>export EC2_REGION=us-east-1
>export AWS_CLOUDWATCH_HOME=/home/developer/aws/CloudWatch-1.0.20.0
>export AWS_ELB_HOME=/home/developer/aws/ElasticLoadBalancing-1.0.34.0
>export PATH=$PATH:$AWS_CLOUDWATCH_HOME/bin
>export PATH=$PATH:$AWS_ELB_HOME/bin
```

To check if Elastic Load Balancing API Tools and Amazon CloudWatch Command Line Tool are setup run `mon-version` and `elb-version`.

If something is not working, incorrect environment variables may be the problem. One of the ways to setup the environment variables is to manually edit `/etc/environment`. In our case the added lines are:

```
JAVA_HOME=/usr/lib/jvm/java-7-openjdk-amd64
AWS_AUTO_SCALING_HOME= /home/developer/aws/AutoScaling-1.0.61.5
PATH=$PATH:$AWS_AUTO_SCALING_HOME/bin
AWS_CREDENTIAL_FILE=$AWS_AUTO_SCALING_HOME/credential-file-path.template
AWS_AUTO_SCALING_URL= https://autoscaling.us-east-1.amazonaws.com
export EC2_REGION=us-east-1
AWS_CLOUDWATCH_HOME=/home/developer/aws/CloudWatch-1.0.20.0
AWS_ELB_HOME=/home/developer/aws/ElasticLoadBalancing-1.0.34.0
PATH=$PATH:$AWS_CLOUDWATCH_HOME/bin
PATH=$PATH:$AWS_ELB_HOME/bin
```

3.3.5 Auto Scaling Setup

When all reviewed command line tools are set up we can proceed to setting up the autoscaling itself.

The following pre requirements should be met:

- AMI is created from the latest version on the instance that should be replicated and put behind the load balancer in case of triggering of scale up event (in our case AMI id is ami-bc22c0d4).

- Load Balancer is created and the server that has all the command line tools installed and which was used to create the previous AMI is put behind it (in our case LoadBalancer name is defaultLB).

We first create a launch configuration TestLC using our AMI and instance type c3.xlarge (the same as the current server).

```
>as-create-launch-config TestLC --image-id ami-bc22c0d4 --instance-type
c3.xlarge
```

Server responds with a confirmation message.

Then we create the Auto Scaling group TestAutoScalingGroup for our created launch configuration TestLC in us-east-1c availability zone (minimum size of group is 1, maximum is 8) and attach it to our LoadBalancer defaultLB:

```
>as-create-auto-scaling-group TestAutoScalingGroup --launch-configuration
TestLC --availability-zones us-east-1c --min-size 1 --max-size 5 --load-
balancers defaultLB
```

Server responds with a confirmation message.

Next we want to define the scaling policy for scaling up, named TestScaleUpPolicy, for our TestAutoScalingGroup. Policy will add one server, cooldown is 5 minutes.

```
>as-put-scaling-policy TestScaleUpPolicy --auto-scaling-group
TestAutoScalingGroup  --adjustment=1 --type ChangeInCapacity  --cooldown 300
```

Server responds with a confirmation message (in our case `arn:aws:autoscaling:us-east-1:689664445651:scalingPolicy:6e02b29b-5356-42ba-8e0c-53e4c1620fe3:autoScalingGroupName/TestAutoScalingGroup:policyName/TestScaleUpPolicy`). We will need this ARN in for the following step.

In the following step we will create the alarm named TestHighCPUAlarm based on average CPU utilization during 5 minutes. After 5 minutes on average CPU utilization higher than 75% our scaling policy TestScaleUpPolicy from TestAutoScalingGroup will be fired. ARN for --alarm-actions setting is taken from the previous step.

```
>mon-put-metric-alarm TestHighCPUAlarm --comparison-operator
GreaterThanThreshold  --evaluation-periods  1 --metric-name  CPUUtilization  --
namespace  "AWS/EC2"  --period 300  --statistic Average --threshold  75 --
alarm-actions arn:aws:autoscaling:us-east-
1:689664445651:scalingPolicy:6e02b29b-5356-42ba-8e0c-
53e4c1620fe3:autoScalingGroupName/TestAutoScalingGroup:policyName/TestScaleUpPo
licy --dimensions "AutoScalingGroupName=TestAutoScalingGroup"
```

Server responds with a confirmation message.

Now we need to basically repeat last 2 actions and create the scaling policy for scaling down, named TestScaleDownPolicy, for our TestAutoScalingGroup. Policy will terminate one server, cooldown is 5 minutes.

```
>as-put-scaling-policy TestScaleDownPolicy --auto-scaling-group
TestAutoScalingGroup  --adjustment=-1 --type ChangeInCapacity  --cooldown 300
```

Using the received ARN we create the alarm named TestLowCPUAlarm based on average CPU utilization during 5 minutes. After 5 minutes on average CPU utilization lower than 30% our scaling policy TestScaleDownPolicy from TestAutoScalingGroup will be fired. ARN for --alarm-actions setting is taken from the previous step.

```
>mon-put-metric-alarm TestLowCPUAlarm  --comparison-operator  LessThanThreshold
--evaluation-periods  1 --metric-name  CPUUtilization --namespace  "AWS/EC2"  -
-period  300  --statistic Average --threshold  30  --alarm-actions
arn:aws:autoscaling:us-east-1:689664445651:scalingPolicy:7d067b99-36d5-4687-
a933-
cc18d8db4e13:autoScalingGroupName/TestAutoScalingGroup:policyName/TestScaleDown
Policy --dimensions "AutoScalingGroupName=TestAutoScalingGroup"
```

Server responds with a confirmation message.

Now we can check if setup was correct with:

```
as-describe-auto-scaling-groups TestAutoScalingGroup --headers
```

3.3.6 Configuration Layout

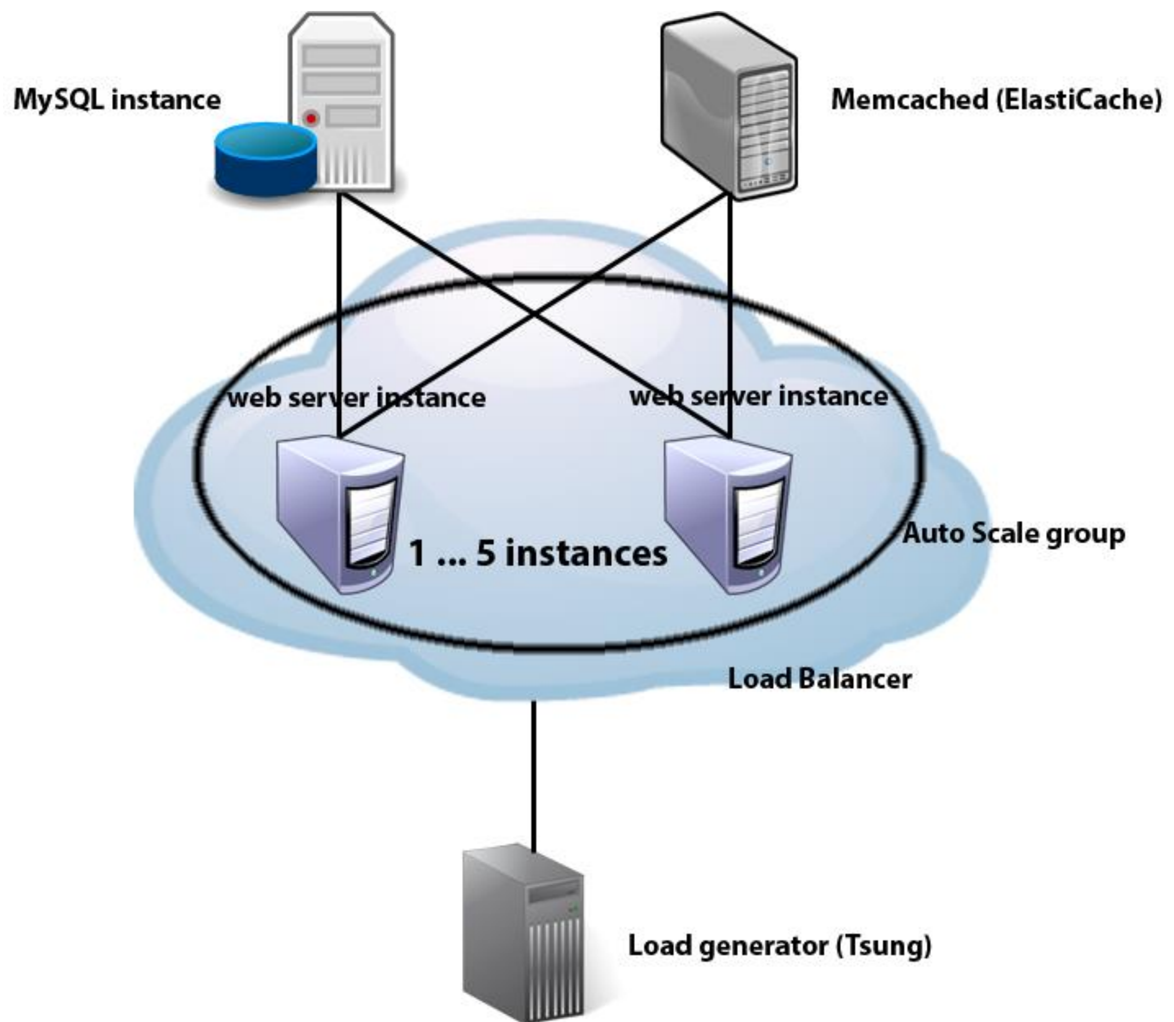

Figure 5. Configuration layout

Current configuration layout shows the relation between all the components of the system: load generator fetches pages from the load balancer. There are 1 to 5 web server instances behind the load balancer that are in auto scale group. Each instance utilizes the same MySQL server instance and Memcached (ElastiCache) node.

3.3.7 Test 7. OPcache + full Memcached. Variable interarrival

In the this test we will try to monitor the behaviour of auto scaling mechanisms. To achieve this we will create a Tsung configuration with several phases of tests. We will start with interarrival = 0.14 and gradually decrease to 0.025 in 12 steps. Steps 1 - 11 will be 7 minutes long each. Step 12 (interarrival = 0.025) will be run for 20

minutes. Steps 13 - 16 will increase interarrival to 0.09604 and each step will be run for 7 minutes. Step 17 (interarrival = 0.14) will be run for 20 minutes. Total test run time is 145 minutes. In current test we emulate sudden increase of page loads on the server, a short period of stability and decrease of attention.

3.3.6.1 Results

Web server	Test duration	interarrival (sec)	session mean (msec)
(Apache+OPcache)x(1...5) + Memcached	145 min	0.14 - 0.025 - 0.14	370

Table 6. Test 7: session mean time

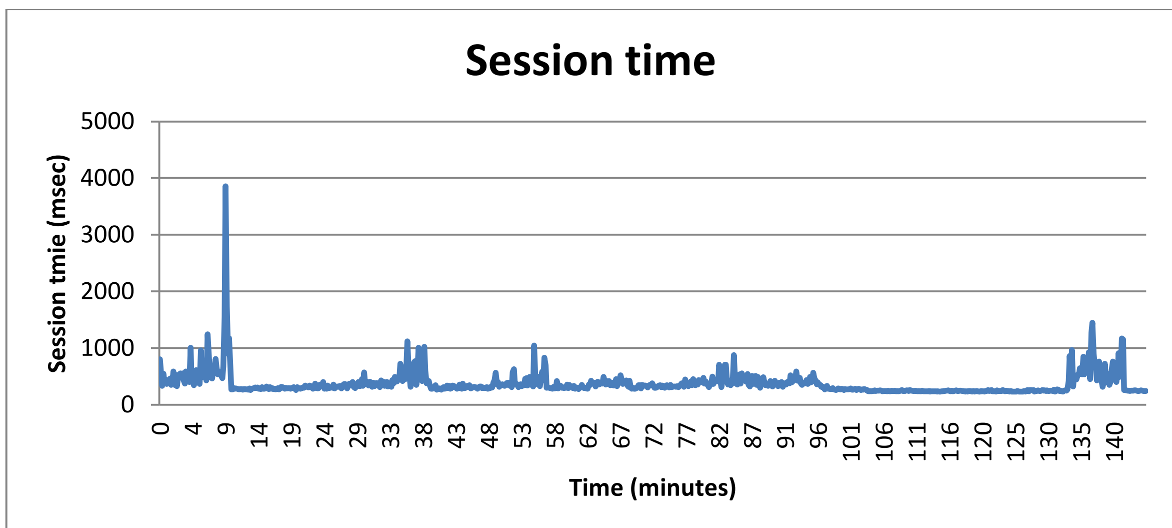

Figure 6. Test 7: session mean time

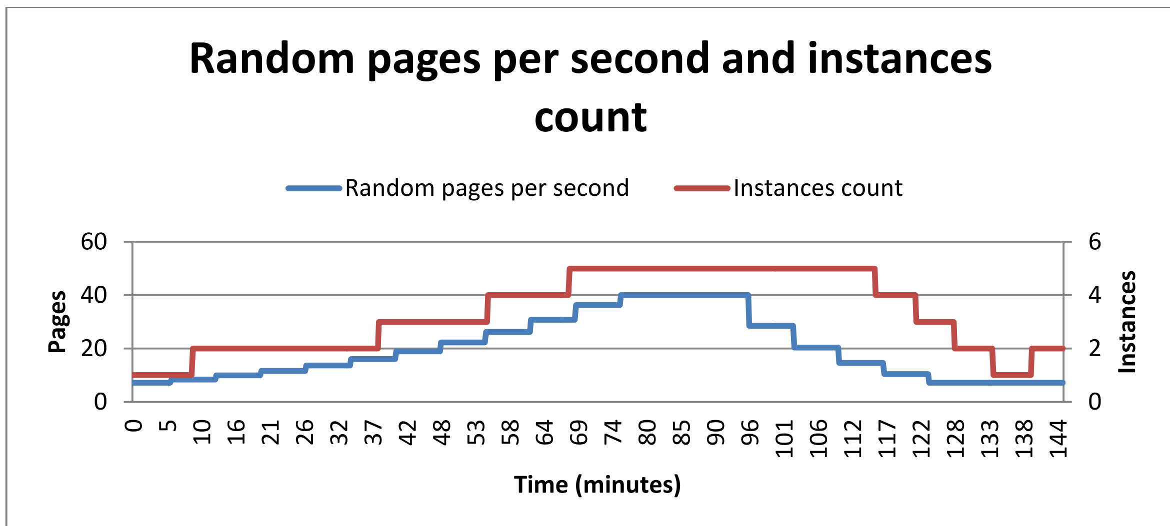

Figure 7. Test 7: random pages per second and instances count

Test results show the effectiveness of auto scaling mechanisms. Session mean time in the test is a fairly good result (370 msec).

In the Figure 6 we can see several peaks that followed the decrease of interarrival variable. As expected, alarms worked and initiated a new instance launch that helped the existing instances to cope with the load. The alarms for scaling down also worked, which is seen on Figure 7.

Test 7 results prove that Amazon auto scaling mechanisms are efficient.

Conclusion

By the end of the thesis both target goals were achieved. We managed to find a good custom web server configuration that can handle the heavy load and we were able to show how to set up Amazon Auto Scaling mechanisms and prove their efficiency.

OPcache turned out to be a very effective and easy to set up tool for decreasing the CPU load by the heavy PHP application. OPcache leveled out the performance difference between Apache HTTP Server and lighttpd, so there is no strong need in experimenting between these two. We selected first one, because it is more popular. Memcached also helps to increase the performance and decrease the load on the web server. Amazon Web Services has a service ElastiCache, which makes the configuration of Memcached instance several clicks step, which also increases its value. We proved that Amazon Elastic Load Balancer evenly distributes the load between several instances, so no special configuration is needed. Our configuration with one MySQL database instance, one ElastiCache Memcached node and 5 web server instances behind the load balancer managed to successfully cope with the load equal to more than 3 million random page requests per 24 hours.

Set up of Auto Scaling mechanisms turned out the be a bit more difficult process, but we managed to review its basics. We also showed that it is efficient and performs well if set up correctly, which can potentially help one to save the good amount of funds.

Information technologies develop all the time: the new stable version of Apache HTTP Server was released during the writing of the thesis, which meant we had to redo all the tests to provide the most recent data. After all the tests were done, one more version of Apache HTTP Server was released again, but it is still not in the official Ubuntu repositories, so we decided it could be the goal of further research alongside with future Amazon Web Services features. Future research can also include the other aspects of scaling in Amazon Cloud: it could be database scaling, usage of Amazon Simple Storage Service for storing files,

utilization of different server side software (e.g. nginx, Cherokee Web Server, mmTurck).

Bibliography

1. What happens online in 60 seconds? [Infographic]. [Online] [Cited: 5 5, 2014.] http://blog.qmee.com/qmee-online-in-60-seconds/.

2. Amazon Media Room: History & Timeline. [Online] [Cited: 5 5, 2014.] http://phx.corporate-ir.net/phoenix.zhtml?c=176060&p=irol-corporateTimeline.

3. CloudTimes Top 100. [Online] [Cited: 5 5, 2014.] http://cloudtimes.org/top100/.

4. Amazon EC2 Pricing. [Online] [Cited: 5 5, 2014.] http://aws.amazon.com/ec2/pricing/.

5. MediaWiki. *Wikipedia.* [Online] [Cited: 5 5, 2014.] http://en.wikipedia.org/wiki/MediaWiki.

6. Wikimedia Grid Report. *Ganglia.* [Online] [Cited: 5 5, 2014.] http://ganglia.wikimedia.org/latest/.

7. Wikipedia.org Site Info. *Alexa.* [Online] [Cited: 5 5, 2014.] http://www.alexa.com/siteinfo/wikipedia.org.

8. **Martti Vasar, Satish Narayana Srirama, Marlon Dumas.** *Framework for Monitoring and Testing Web Application Scalability on the Cloud, Nordic Symposium on Cloud Computing & Internet Technologies (NORDICLOUD 2012), Co-located event at Joint 10th Working IEEE/IFIP Conference on Software Architecture & 6th European Conference on Software Architecture (WICSA/ECSA 2012), August 20-24, 2012, pp. 53-60. ACM.*

9. Wikimedia servers. [Online] [Cited: 5 5, 2014.] http://meta.wikimedia.org/wiki/Wikimedia_servers.

10. About the Apache HTTP Server Project. *The Apache HTTP Server Project.* [Online] [Cited: 5 5, 2014.] http://httpd.apache.org/ABOUT_APACHE.html.

11. Netcraft. *April 2014 Web Server Survey.* [Online] [Cited: 5 5, 2014.] http://news.netcraft.com/archives/2014/04/02/april-2014-web-server-survey.html.

12. Welcome to Lighttpd. *lighty labs.* [Online] [Cited: 5 5, 2014.] http://redmine.lighttpd.net/projects/1/wiki.

13. lighttpd Usage Statistics. [Online] [Cited: 5 5, 2014.] http://trends.builtwith.com/Web-Server/lighttpd.

14. OPcache Introduction. *PHP.* [Online] [Cited: 5 5, 2014.] http://www.php.net/manual/en/intro.opcache.php.

15. What is Memcached? [Online] [Cited: 5 5, 2014.] http://memcached.org/.

16. Amazon ElastiCache. [Online] [Cited: 5 5, 2014.] http://aws.amazon.com/elasticache/.

17. MySQL Editions. *MySQL.* [Online] [Cited: 5 5, 2014.] http://www.mysql.com/products/.

18. MediaWiki. *Manual:Performance tuning.* [Online] [Cited: 5 5, 2014.] http://www.mediawiki.org/wiki/Manual:Performance_tuning.

19. **Satish Srirama, Huber Flores, Martti Vasar.** *"Deliverable D6. 7 Performance testing of cloud applications, Final Release." EU FP7 REMICS, 2013.*

20. EC2 Instance Types. *Amazon.* [Online] [Cited: 5 5, 2014.] http://aws.amazon.com/ec2/instance-types/.

21. MediaWiki. *Manual:Installation guide.* [Online] [Cited: 5 5, 2014.] http://www.mediawiki.org/wiki/Manual:Installation_guide.

22. Database dump progress. *Wikimedia Downloads.* [Online] [Cited: 5 5, 2014.] http://dumps.wikimedia.org/backup-index.html.

23. etwiki dump progress on 20140427. [Online] [Cited: 5 5, 2014.] http://dumps.wikimedia.org/etwiki/20140427/.

24. memcached. *MediaWiki.* [Online] [Cited: 5 5, 2014.] http://www.mediawiki.org/wiki/Memcached.

25. **Antonin Abhervé Andrey Sadovykh, Satish Srirama, Pelle Jakovits, Michael Smialek, Wiktor Nowakowski, Nicolas Ferry, Brice Morin.** *"Deliverable D4. 5 REMICS Migrate Principles and Methods." EU FP7 REMICS, 2013.*

26. Elastic Load Balancing. *Amazon.* [Online] [Cited: 5 5, 2014.] http://aws.amazon.com/elasticloadbalancing/.

27. Video: Create Your Own Customized AMI. *Amazon.* [Online] [Cited: 5 5, 2014.] http://aws.amazon.com/articles/938.

28. Auto Scaling Command Line Tool. *AWS Developer Tools.* [Online] [Cited: 5 5, 2014.] http://aws.amazon.com/developertools/2535.

29. Amazon Web Services Glossary. *AWS Documentation.* [Online] [Cited: 5 5, 2014.] http://docs.amazonwebservices.com/general/latest/gr/rande.html.

Appendix

Supplementary Material

The companion archive attached to this thesis contains the results and Tsung configuration files of all the tests executed during the thesis.

License

Non-exclusive licence to reproduce thesis and make thesis public

I, Timur Hassanov (date of birth: 14.10.1988),

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to:

1.1. reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and

1.2. make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until expiry of the term of validity of the copyright,

of my thesis

Web Applications Scaling in Amazon Cloud,

supervised by Satish Narayana Srirama, Ph.D,

2. I am aware of the fact that the author retains these rights.

3. I certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu, 05.05.2014