Analysis of Backend Server

Investigation and analysis of security and performance of a backend server

Vahid Shirvani
Abstract

Analysis of Backend Server

Vahid Shirvani

MUSES is a European Union funded project that is the result of collaboration between several European companies and universities. The project has the well known client-server architecture. Security and performance are two important factors to consider when developing web server applications. Security is crucial due to the malicious attacks from hackers and performance is essential in order to stay responsive when numerous clients are connected. This thesis project has analyzed the MUSES project from a security and performance perspective in order to detect issues, discuss the consequences and propose or even in some cases apply the solution. The thesis has resulted in MUSES server being improved not only in the mentioned areas but also gaining more stability. Other positive outcome from the project has been the creation of a client program which can be used for benchmarking servers.
# Table of Contents

1. Introduction.................................................................................................................. 7
   1.1. The MUSES........................................................................................................... 7
1.2. The Problem................................................................................................................ 8
1.3. Background............................................................................................................... 9
1.4. Related Work........................................................................................................... 10

2. Methods.......................................................................................................................... 11
   2.1. Security.................................................................................................................. 11
   2.2. Performance.......................................................................................................... 11

3. System Analysis.......................................................................................................... 13
   3.1. Security................................................................................................................ 13
       3.1.1. Injection........................................................................................................ 13
       3.1.2. Authentication............................................................................................ 13
       3.1.3. Authorization.............................................................................................. 14
       3.1.4. Confidentiality............................................................................................ 15
       3.1.5. Updates........................................................................................................ 16
   3.2. Performance.......................................................................................................... 17
       3.2.1. Benchmarking.............................................................................................. 17
       3.2.2. Exclusive access.......................................................................................... 19
       3.2.3. Apache Tomcat............................................................................................ 21
       3.2.4. Database....................................................................................................... 22
       3.2.5. Hardware....................................................................................................... 23

4. Results.......................................................................................................................... 24
   4.1. Security................................................................................................................ 24
   4.2. Performance.......................................................................................................... 24

5. Conclusions and Future Work.................................................................................... 30

6. Bibliography.................................................................................................................... 32
   Appendix A: Apache Tomcat....................................................................................... 35
   Appendix B: Database.................................................................................................. 36
   Appendix C: Benchmarks............................................................................................ 38
1. Introduction

1.1. The MUSES

Sweden Connectivity AB is involved in an open source project called Multiplatform Usable Endpoint Security (MUSES) Project which is funded by European Union. Apart from just Sweden Connectivity, there are six other European companies and universities that are members of the consortium.

One of MUSES' main purposes is to protect company assets. MUSES is a more advanced access control system compared to the built-in access control in today's operating systems. An example of a use case scenario could be a company that has provided one of its employee with a cellphone which has MUSES application installed. If the employee tries to access one of the company's sensitive documents while being connected to an insecure public WiFi then he will be confronted with a security warning. MUSES provides companies with a mechanism to control endpoint devices. What and how assets should be protected are defined by policy rules. If companies want to have more relaxed or restrictive rules, they only need to modify the policy files and not the source code itself. The job of MUSES application is to enforce these policies.

![Illustration 1: MUSES client architecture](1)
Both illustrations 1 and 2 demonstrate the high level view of the entire MUSES system. As we can see MUSES uses the well known client-server architecture. The MUSES client application is able to sense the context through some software sensors and make a decision to whether or not grant access to the asked resource. Context could for example be the currently running applications in the background. The database on the client side is used for storing user actions. Keeping a history of all the actions user has done is useful during the decision process. Bear in mind that the processing power on the client side is limited so some decisions have to be made by the server. The server contains several modules. The Knowledge Refinement System (KRS) is a data mining module that generates new policy rules. When the client is not able to make a local decision on its own then the Continuous Real-Time Event Processor (CRTEP) will do the job of making an online decision. As the name applies Connection Manager is the module that connects the system to the outside world. This module acts as a port which handles all the traffic. For more information, document [1] describing every module in detail is available.

A prototype of the entire system has been created and is available right now on Github. The prototype will evolve and at some point be released as a commercial product. The final product of MUSES will be used by companies. They will install the client software on employee devices in order to protect sensitive assets.

1.2. The Problem

This thesis will analyze the prototype from a both security and performance perspective. The goal is to make sure that the product is reasonably secure and has an acceptable performance before release.

A lot of software created nowadays has some imperfections because companies want to create a prototype as quick as possible. They take shortcuts when possible in order to save time and money. This is completely understandable, but it also comes at a cost. This can lead to prototypes having some security flaws and performance issues. Unless security has been one of the top priorities or part of the major goals from day one the system probably contains some sort of vulnerability.

One of the companies involved in the MUSES project does provide expertise in the security related fields. This has been necessary because MUSES project is aimed at protecting company assets
which is a security functionality. Most of the focus of this company has been in implementing the policy manager inside MUSES project. Therefore this thesis project will look into a different aspect of security. The area that will be explored by this thesis project is how protected the MUSES system itself is and not how good it protects some other assets. It is valuable to have an outsider audit this part of the system due to it being less in focus.

As you have seen in the previous part, the MUSES project contains many different modules which makes it quite large and complex. This thesis will mostly focus on Connection Manager, User Context Event Data Receiver, Database, communication channel and the server container which hosts the entire web application. Deciding to focus on modules that connects the system to the outside world is quite a natural choice. Attackers will have their interactions with these modules so holding a strong first line of defense is important. Beside security, Connection Manager plays an essential role when it comes to performance as well. All the clients will interact directly with this module so it is crucial to not introduce any bottlenecks and keep the natural concurrency and parallelism of a server.

Worth keeping in mind that security and performance sometimes intersect with each other. This could occasionally mean that improving one could lead to worsening the other. Though this is not always the case, under certain circumstances improving either one would enhance the other as well. So to summarize; the task of this thesis project consist of investigating the MUSES system and detect issues, discuss the consequences and propose solutions.

1.3. Background

The programming language picked for development of both the client and the server prototypes have been Java. Apart from the standard library, other third party frameworks have also been used in order boost the development process. The server side is built on Java EE technology, so knowledge from either a course or a relevant litterateur [2] [3] that covers this area is recommended. The wide variety of available frameworks for back-end servers are many, so despite having knowledge from related courses, one might still have to spend some time to get familiar with some of the frameworks used in this particular project.

The methods used for analyzing the system from a security standpoint will be very similar to the methods taught in most of the advanced computer security courses at academic level. Both relevant introductory litterateur [4] as well as advanced level litterateur [5] could be useful resources. Methods can range from finding security flaws by studying the source code but also doing penetration testing while treating the system as a black-box.

The performance has not been one of the main goals during the development of MUSES project. No previous measurements have ever been done, so there are no figures available as a reference point on how well the server performs. This thesis project will aim at producing these missing figures which will be a very useful data contribution. The reader is encouraged to have some prior knowledge in concurrent and parallel programming. There are some great litterature [6] [7] [8] available that cover the best practices for this type of programming style. Worth noting that while some of these techniques are language agnostic and could easily be translated over to other programming languages, some of them might not. As the language of choice for the server has been Java, the techniques used to achieve concurrency and parallelism will be more tailored toward this particular language.
1.4. Related Work

Computer security as a subject is very broad and contains many different aspects. It has evolved faster since the invention of the first personal computer and has become more and more relevant especially after the arrival of the world wide web. Security on the web is a crucial element. Without it, e-commerce and online banking among many other activities would have not been possible. The secure web is possible today thanks to the accumulative effort of numerous scientists over the many years.

As mentioned before this thesis project will consist of studying the MUSES system trying to identify security related weaknesses. This consist of performing a security audit on the system [9] [10]. Once flaws are found, solutions will be suggested. It is important to note that these solutions for security vulnerabilities are invented by others.

As we will go through an entire system searching for flaws, we will probably come a cross some common mistakes. Having some previous familiarity with the common security vulnerabilities and their respective fix could be helpful. These are some of the relevant work by others that might be used in this project.

- The most common weakness among web applications is the vulnerability against SQL injection [11]. When a vulnerability becomes this widespread then naturally it will attract a lot of researches. People will try to invent more efficient ways to prevent such attacks [12] [13] and some try to analyze the available solutions in order to compare them against each other [14].
- One of the ways that hackers will try to enter a system is through pretending to be some legitimate user. Before the hacker can do such thing they usually need to implant some script inside the web application. This security vulnerability is called Cross-site Scripting (CSS/XSS) and fortunately some prevention methods exist [15] [16].
- On the other hand some attacker might try to make a legitimate user perform their action for them. This type of attack is known as Cross-Site Request Forgery (CSRF). The most preferable way to make a web application immune to this type of attack is through Synchronizer Token Pattern [17].
- Authentication is one of the key elements of security. Proper way of accomplishing authentication by credentials is through using hashing methods [18]. The server can authenticate itself to clients through public key infrastructure [19].

One way to improve the performance of a web server is to upgrade the hardware. There has been a paper published on a related subject [20]. The author analyzes the performance of a web application on different hardware configurations. Although this is not exactly what is going to be done in this thesis project however there are some other great general advices that are worth having in mind during the execution of this project. This thesis project will focus on making the server perform better on the existing hardware through optimization.

MUSES project has led to some great research and publications [21] [22] [23] [24] [25] [26] [27]. These are worth referencing in case MUSES has woken the curiosity of some readers that would like to know more about the associated publications.
2. Methods

2.1. Security

The goals for this thesis project will be realistic. Making a piece of software 100% secure is practically impossible and that is not the goal, but a piece of software can be reasonably secure by following the best practices for avoiding the common pitfalls. This is an achievable goal that this thesis project will aim for. Saltzer and Schroeder's 1975 Design Principles [28], and Top 10 Secure Coding Practices [29] will be used as reference for best practices. The reference for common pitfalls will based on Top 25 Most Dangerous Software Errors [11] published by CWE/SANS. These lists are not disjoint and there are no restrictions that the security related practices or pitfalls have to be part of the above lists, so if anything of value gets discovered during the code review process, it will be pointed out regardless. Although this thesis project will mainly focus on pointing out the cases were these principles has been violated, the obedience of these principles will be acknowledged when security precautions have been taken by the MUSES development team. The points mentioned below are the areas that will be examined:

**Injection:** The clients can misuse the server's trust and send malicious data. The source code of the web application shall be reviewed in order to find out if the server does neutralize the incoming data before processing it.

**Authentication:** The server should only accept data from legitimate users, so every client must identify itself before starting the communicate. The implemented authentication mechanism in place will be reviewed in order to determine its security strength. The authentication is usually mutual which means that the server also has to prove itself to the clients. After all no client wants to connect to a server that has been setup by hackers.

**Authorization:** There should be boundaries on what clients are able to access. Even a legitimate client should not have the permission to do everything on the server. The server source code will be examined to see if such access control is in place and how well it is built.

**Confidentiality:** The data sent back and forth between the two parties could contain sensitive information. One of the most popular solutions in such scenario is encryption. Not all encryption algorithms are the same. While some are strong and secure, others could be weak and obsolete. This thesis project will evaluate the choices made for encryption algorithms on the server side.

**Updates:** One of the reasons software programs get updated, is to patch the discovered security vulnerabilities. The server machine which hosts the web application most probably contains other software services as well. Depending on which version of those programs are installed, the potential that they could expose the entire system exist. Thus we shall go through the system and identify what else is running and determine if it introduces some risk or not.

2.2. Performance

MUSES server shall provide service to many simultaneous clients, so it is important that it can
handle several client connections at the same time. The points mentioned below are the areas that
will be examined in order to achieve this goal.

**Benchmarking:** Stress test the server and measure the results. The currently available Android
client is not capable of sending multiple requests because that is not a correct behavior from a
client, so a special client needs to be developed for this specific use case. The special client will
measure the response time and success rate of the all the sent requests. Some other parts worth
monitoring during stress testing could be processor load, memory, bandwidth, etc.

**Exclusive access:** MUSES server is hosted inside Apache Tomcat which is a container. A container
will spawn multiple threads (one for each client/connection if necessary) in order to be able to serve
multiple clients at the same time. Once a program is required to be thread-safe, special care needs to
be taken otherwise the program will run slow or even crash/halt. Access to shared resources
between several threads needs to be handled properly. In some cases shared resources can be the
cause of bottlenecks. According to Amdahl's Law the speed up of a program (in an ideal scenario) is
limited to the sequential fraction of the program.

**Optimizations:** It is important to avoid optimizations that results in small improvements, so during
the code review I will avoid suggesting micro optimizations. On the other hand some optimizations
are not from an economical perspective worth carrying out due to the necessary time and resources
needed. There are sometimes tradeoffs between code that is readable and code that is optimized. If
the performance gain is small then it is not worth spending the time making the code look more
complex. Basically the 80-20 rule from Pareto principle [30] should be followed. If applied here it
would mean that 20% of the code (also known as hot path) is being executed 80% of the time, so
for best outcome hot path should be optimized. If there are several parts that could be optimized, the
priority should be based on performance gain; largest contributor first.

**Frameworks:** Java EE is a specification and not an implementation, so there are a lot of
frameworks available for use. MUSES server does take advantage of some of the frameworks
available in the market. Although it is true that the choice of a framework is subjective, it still could
be discussed and analyzed from a performance perspective. The code base for the used frameworks
will not be reviewed during this thesis project but their configurations will be. In some cases the
default configurations might not be the most optimal one.
3. System Analysis

3.1. Security

3.1.1. Injection

From a security perspective the server should never trust the input data that comes from the clients. The server should always assume that the received data could contain malicious content. Therefore the received data must always be neutralized before propagating further into the system for processing. Neutralizing the input could for example either be escaping special characters in the input string, or completely rejecting the input if it does or does not fulfill certain criteria. For instance reject the input if it contains semicolon, or reject the input if it is anything else other than login/logout.

In the case of MUSES server the input will be a part of the HTTP header and the payload body (POST request). Every time a client wants to send some data to the server, it will pack the data inside a JSON object which is then located inside the payload section. Once the server receives the JSON object it will insert it into the database and also send it to the event processor unit. SQL injections are a very common vulnerability among servers, but MUSES server is immune against them due to use of Hibernate Named Queries. Bear in mind that the safety does not come from the fact that the server is using Hibernate framework, but it is the Named Query that uses parameter binding (also known as prepared or parameterized statements) under the hood that makes it resilient against SQL/HQL injections. Apart from that prepared statements will lead to better performance which is a bonus.

The most common mistake to become prone to OS injection is the use of exec() either from System or Runtime class. A simple search through the project did not yield any results for the use of exec method, so the MUSES server is resilient to OS injection as well because it is not trying to read any input script for execution. It is very important that it remains like this because scripts are executed with system privileges and that could be harmful if it gets replaced with a malicious one.

It is worth mentioning that even if Cross-site Scripting (CSS/XSS) injection would be possible on the MUSES server, it would not have any effect because the client is not web-browser based. XSS requires a client that can parse HTML, Javascript and CSS (Cascading Style Sheets). Same goes for Cross-Site Request Forgery (CSRF) that also requires the client to have a web-browser like behavior.

3.1.2. Authentication

The MUSES client will verify the authenticity of the server through the Public key infrastructure (PKI) [19]. During the establishment of the HTTPS (HTTP over TLS) connection, the server will provide its public key which is signed by a mutual trusted Certificate Authority (CA). The CA used for issuing the certificate for MUSES server is not part of the Root Certificate Authority list which means that the public key is self-signed. This is not a security concern as long as the server's certificate is pinned (hard-coded) into all the clients and the CA's private key is not compromised. Unfortunately the current version of the client fetches (only once at start-up) the certificate from the SD card. The SD card is accessible by the user and applications which mean the certificate could potentially be replaced with another one. This is understandable as it eases the development
process, but this behavior should not be a part of the finalized version used in production as the client should only talk to a predefined server. One solution to this issue could be to import the certificate of the trusted servers into a Keystore and then make the Keystore part of the project source code.

On the other hand, the client will authenticate itself through the means of providing credentials. The authentication mechanism used on the server side is a custom built one. From a security perspective it is better to rely on the container to handle sensitive tasks such as authentication. Both Java EE and Spring Framework provide great support for authentication management. The implementation is also much more simpler as it only requires some minor changes in the deployment descriptor file (a configuration file in XML format). Relaying on the container for authentication would mean less code which translates into less bugs, easier maintenance and simpler design. Furthermore the authentication manager provided in containers benefit from years of testing which is hard to argue against. In MUSES case only the path “/commain” has to be mentions in the configuration file, and after that Apache Tomcat container will make sure that every client is logged in before it can access this resource. This is much simpler than explicitly checking if the user is authenticated in all of the possible scenarios. The code base is large, and if some critical function somewhere among that complexity misses the authentication checking process that would make the server prone to security issues.

Another security flaw that is present in MUSES server is the possibility of unlimited login attempts. This flaw in authentication manager makes the server vulnerable to dictionary and brute-force attacks. Even if the custom built authentication manager would be replaced by one provided in e.g. Spring Security Framework, this weakness could still be present, but the solution for fixing it would be very simple. Either the LockOut Realm mechanism available in Apache Tomcat [31] has to be used (preferred solution) or if that for some reason is not enough then a custom lock out mechanism can be built using the available listeners offered by Spring Security. On every login attempt a counter inside the listener would be increased, once it surpasses the defined threshold then the user would be blocked.

Authentication through credentials is common practice and is widely used across the web but sometimes it is not enough. Some user might choose weak passwords because they are easier to remember while some other users have the habit of using the same password across multiple services. Even if the average password strength would be strong, passwords could still be stolen through phishing and Social Engineering. A solution for such issue could be Two factor authentication which provides another layer of defense. There are numerous methods that can be chosen as the second factor e.g. geographical location, biometrics (fingerprint) or even a simple email confirmation link.

### 3.1.3. Authorization

Defines what the user is able to do which is different from authentication that only verifies the identity (who) of the user. Solving one does not fix the other because they are both orthogonal. Therefore it is very important to get both authentication and authorization right otherwise the service could be compromised for malicious purposes. Unfortunately there are some security concerns regarding how authorization is handled in MUSES server. Let us look at an example, once a client is logged in he/she can start to send events with others name. If MUSES would be deployed in some corporation in future, this could be exploited by the employees. Employee Alice could write the name of another employee Bob in her events before sending them. This would lead to server storing the events in the database under Bob’s name. This is possible because of the presence two issues, A) before accepting the event the server checks if the client is logged in. It is great that
server does such a sanity check, but authentication does not solve authorization. B) server trusts the client with the given credentials included in the received event. The solution to this security flaws is quite simple. The server should discard the credentials inside the event, and query the authentication manager for the currently logged in client identity and then store the event inside database with that name. This way all the events coming from Alice will be registered in the database with the name Alice. It is worth mentioning that both the authentication and authorization issues claimed above have been proven to exist with a tool called SimulateManyHttpClients that was developed during the thesis project for performance measurement purposes. The SimulateManyHttpClients tool was able to do many login attempts as well as sending events with semi-random content.

3.1.4. Confidentiality

The connection between the MUSES server and the clients are encrypted using HTTPS protocol. This makes the communication channel safe from eavesdropping among other things which is important because of the data sensitivity. Encrypting the communication channel is a step in right direction, but there are still other parts of the system that needs to be worked on in order to achieve a reasonable secure server. Defense in depth is a concept of using multilayer security throughout the system, so in the event of a vulnerability exploitation a fall back on another layer of security will protect the confidential data from being exposed. Just like a thief that will search the house after a break in for valuable goods, a hacker will also search the target machine for valuable data or information once he has succeeded in breaking in. Therefore is defense in depth important because it works like barriers that can slow down or even stop the hacker once he is inside. The credentials for all clients are stored inside a database on the server side. These are of course necessary for authentication of the clients during login. The current security concern regarding these credentials is all the passwords which are stored in plain text format. Imagine if this table containing the credentials would be leaked. That would allow anyone to login as a legitimate client. If some clients use the same password across several services, then those services could also be exploited. The solution to such issue is simple. Instead of storing the password itself, the hash of the password should be stored. Hash is used over encryption because there is no need for reversibility and also the length of the output is fixed. In order to avoid Rainbow attacks, password should be combined with a salt before hashing. To protect server from Replay attacks a nonce should be often used in requests.

The MUSES server does not offer a mechanism for the clients to use in order to change their passwords. Clients might have to go through the hopes of contacting the administrator for such a small thing as changing their passwords. This is contrary to the Principle of Psychological Acceptability which states that the protection mechanism involved should be easy to use otherwise people will skip it. The tomcat-users.xml file found in Tomcats installation folder contains administrator credentials. Anyone with a read access to this file can take control of the Apache Tomcat container. For example a hacker could host a malicious web application as a Backdoor. Therefore should the read and write access permissions on this file be restricted to only Tomcat app itself. This might make the task of adding or removing administrators more difficult especially if it happens more often, so a better solution would be to move the credentials to the database entirely by using a JDBCRealm. Once properly configured the server.xml file permissions shall be changed to Tomcat only to avoid all possible tinkering. The authentication mechanism of the MUSES server GUI (which was developed by me during the Hackathon event in Spain) actually uses JDBCRealm. See MusesSRM on Github.
or appendices for example on how this could be done.

Looking at the source code of the MUSES server reveals that the Spring Security framework has been implemented but not fully utilized yet, that is, it has not replaced the custom authentication manager yet. It is great that the project will take advantage of a more mature solution, but there are some concerns about how it is configured at the moment. The spring-security.xml which is used as a configuration file contains hard-coded credentials. Hard-coding the credentials make the process of adding and removing users troublesome due to the necessary compilation and deployment after modifying the user list. Apart from that, the password is written in plain text and viewable by anyone. The solution is to move the credential list to the database and hash all the passwords. Spring Security does offer support for this feature [32] which makes the implementation quite simple. Even if the project would have been closed source and not viewable by any outsider, hard-coding the credentials into the source code would still be wrong. Security through obscurity is a bad idea, and in this case extracting credentials from binary blobs are not impossible.

MUSES server stores its data inside a database. The credentials that the server uses for establishing the database connection are hard-coded into the source code (hibernate.cfg.xml and persistence.xml) which makes them viewable by anyone. There are several ways this issue could be solved. One way would be to let Hibernate fetch the credentials using a JNDI look-up. In this case the credentials would be stored inside the Tomcat's server.xml which should only be readable by Tomcat alone.

3.1.5. Updates

There is one severe security concern inside the connection manager of MUSES server. It is described in more detail in the exclusive access section below because it also relates to performance issues. A short summary is that some variables in the servlet that should not be shared globally across all incoming requests are indeed global. The consequence could be that the response for different clients could get mixed up.

As we know, new security vulnerabilities are constantly being discovered in all sorts of software now days. Software makers usually release patches and updates in order to cover these. So there is a valid incentive from a security stand point to always stay on the latest version of whatever program you are running. On the other hand one might argue that the latest versions could contain some stability issue due to the added new features that have not been properly tested. So either way there is a compromise that an organization has to make. Some of the software that are installed on the public server machine hosting MUSES server are listed below. The number of vulnerabilities that each version contains has been obtained by querying cvedetails.com which is a well known public security vulnerability database.

- Operating system installed on the server machine is Ubuntu 12.04 LTS. There are 104 security vulnerabilities found for this version as of writing this report. Bear in mind that this is the Long Term Support variant of Ubuntu so security patches are available.
- The current Linux kernel version on the server is 3.5.0 which contains 204 vulnerabilities.
- The installed servlet container is Apache Tomcat with version 7.0.26. This version contains 11 security vulnerabilities.
- The version of Java Development Kit (JDK) installed on the server is 1.6.0_27. This version contains 15 vulnerabilities where more than half of them are critical.
- Apache Maven is the build automation tool used in MUSES server. The installed version of Maven on the public server is 3.0.4. Interestingly that is the only version of Maven that has ever had a vulnerability.
• The relational database management system (RDBMS) installed on the server is MySQL 5.5.32. This version contains 42 security vulnerabilities.
• Jenkins provides continuous integration services for software development. Current installed version is 1.570 which contains 8 security vulnerabilities.

There are certainly more programs installed on the server machine. There is no point in going through each and every one to discuss how many vulnerabilities each have. The point was to demonstrate how important it is that all the programs on the server have to be updated to at least a version that does not contain any critical vulnerability. Just to put it into perspective, some of those vulnerabilities let an attacker completely take over the system and gain administrator privileges. For example once an attacker has managed to get into the internal LAN network then he has surpassed the first line of defense which is the routers Network Address Translation (NAT) which act as a firewall. At this point he will be able to get access to the continuous integration tool called Jenkins which contains all the source code for all the projects. Therefore it is not a good idea to host a public MUSES server on the internal network of the organization. Apart from installed programs, MUSES server code base has some dependencies to other third party frameworks. List of all the dependencies can be found in the pom.xml file in the root directory of the project source code. Although less probable even these frameworks that the project depends on could contain security vulnerabilities. The best advice would be to fetch and build using the most recent version of the framework.

3.2. Performance

3.2.1. Benchmarking

It is important to know how the server would react once many clients are connected. Testing this with the current available Android client would be challenging as it either would require many hardware devices or some changes to the client code base. Changing the code base for a client that has not been developed for such a purpose in first place could be difficult plus the client contains many other components that are irrelevant for benchmarking (unnecessary complexity). Even in that scenario other issues like for example wireless connection could distort the results. One solution was to study the communication protocol between the client and the server and then to develop a completely new client based on that. The new client would of course not contain unnecessary components in order to save time. The proposed client got developed for desktop platform under the codename SimulateManyHttpClients. It is worth mentioning that the client application was at first developed using an implementation of HTTP client available in Java's standard library. After some unexpected results and troubleshooting, it turned out that the current lightweight implemented framework is simply not designed for sending thousands of requests per second. So the decision was made to rewrite the client application using the Apache's HTTP client that takes advantage of pool connections. One of the main differences was that Apache's concrete implementation was meant to be used in heavy scenarios like stress testing. Here is a simple use case scenario:

C:\Users\vahid\svn\trunk\source\SimulateManyHttpClients\target>smhc
3605.0  100.0  0.0

In this case the client has connected to the default predefined server and measured the average response time, response success rate and login success rate respectively. Here are the available options that can be explicitly set:
C:\Users\vahid\svn\trunk\source\SimulateManyHttpClients\target>smhc -h
usage: smhc
    -b  use random credentials for login (simulate brute-force)
    -c <arg>  number of clients
    -h  help menu
    -m <arg>  choose mode: 1 is urlconnection, 2 is urlconnection with
               thread pool and 3 is apache client with threadpool
    -n <arg>  number of attempts each client will do
    -p <arg>  the type of payload for http post requests. Options are login,
               logout and sync
    -r <arg>  maximum delay between each attempt in milliseconds
    -s  preserve the cookie in order to have a session
    -t <arg>  the value for the connection-type in the header. Options are
               connect, data and poll
    -u <arg>  complete url of the server (without / at the end)
    -v  enable verbose output

As you can see the client can change behavior just by defining some options on execution. For example number of clients that should be connected to the server can be set with -c option. The developed SimulateManyHttpClients is a multithreaded program, so each client is executed in a separate thread for higher throughput. Different combinations can be used to replicate different scenarios which can be useful when monitoring the server. The reported time, starts just before the request has been sent and ends just after a response has been received. Multiple HTTP POST requests are sent (defined by -n option) for a more accurate average time. Apart from the number of request, the time interval that all the requests are in-between can also be set before running. For example let us say that we want to send 100 requests from the SimulateManyHttpClients to the server. Should all the requests be packed closely that is one after another, or should they be distributed across 10 seconds? The time frame can be specified with -r option. If the -r option is not present then all the generated requests will be sent one after another. The MUSES server is actually deployed in a server machine which is available around the clock. It is mainly used by developers of the project for testing purposes. Using that shared server for benchmarking was not an option because it required stress testing which potentially could make the server unresponsive for others. The solution of course was to deploy the MUSES server on a different machine, dedicated only for measurements. For more accurate results the SimulateManyHttpClients and the MUSES server had to run on separate machines. This way the two applications would not have to compete for the same hardware resources. The hardware and software specifications of the client and server machines for testing were:

<table>
<thead>
<tr>
<th>Machine</th>
<th>Server</th>
<th>Client</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS</td>
<td>Ubuntu 12.04 in VirtualBox</td>
<td>Microsoft Windows 8 64 bit</td>
</tr>
<tr>
<td>CPU</td>
<td>Intel Core i5-4210U</td>
<td>Intel Core i5-3210M</td>
</tr>
<tr>
<td>RAM</td>
<td>2GB for VirtualBox</td>
<td>6GB</td>
</tr>
<tr>
<td>Link</td>
<td>1Gbps Ethernet</td>
<td>1Gbps Ethernet</td>
</tr>
<tr>
<td>Java</td>
<td>1.8.0_40</td>
<td>1.7.0_21</td>
</tr>
<tr>
<td>Container</td>
<td>Apache Tomcat 7.0.26</td>
<td>-</td>
</tr>
<tr>
<td>MySQL</td>
<td>5.5.41</td>
<td>-</td>
</tr>
</tbody>
</table>

Hardware and software specifications
As specified above the two devices were connected via 1Gbps Ethernet cable (through a network switch) which never seemed to be causing any bottlenecks during measurements. On average only 1% of the available bandwidth were used which is far from total capacity. Also worth mentioning is that the results from client pinging the server were less than 1ms which is negligible. So this means that we can ignore the contribution of the communication link to the response time. However in reality this is not the case. As a reference the result of me sitting in Stockholm and pinging a server in Stockholm (kth.se) and Gothenburg (chalmers.se) were respectively 7 and 23 milliseconds. So such latency should be accounted for in reality if the distance between the server and the clients are 10 to roughly 400 kilometers. It only takes light 1.3 milliseconds to travel 400 kilometers in vacuum, so the reasons behind the additional delay are factors like non straight lines, equipment like repeaters and switches in the middle, medium of the channel not being ideal vacuum and etc.

All the benchmarks have one thing in common. They all measure the behavior of the server as the number of clients gets increased. This is achieved by executing the SimulateManyHttpClients many times inside a loop and increasing the number of clients for each iteration. Instead of running the SimulateManyHttpClients many times manually with different parameters a script was used to automate this process.

In order to be on the safe side, the virtual machine that ran the server was restarted after each benchmark session. This was done to avoid any effect that one benchmarking could have on the next measuring session.

Some of the benchmarking results are presented as graphs in result chapter. The rest of the graphs can be found in the appendices for the readers who are interested.

### 3.2.2. Exclusive access

One instance of a servlet can be accessed by many threads. This is also true for JavaServer Pages (JSP) as they are technically servlets as well. Multithreading is how Apache Tomcat container achieves parallelism. This is important because you need to make sure that the servlet stays thread-safe. Violating thread-safety will make the server unstable and could potentially lead to data corruption and crash. MUSES server contains one servlet called ComMainServlet.java. The code snippet bellow is a section from that servlet.

```java
public class ComMainServlet extends HttpServlet {

    private static final long serialVersionUID = 1L;
    private static Logger logger = Logger.getLogger(ComMainServlet.class.getName());
    private Helper helper;
    private SessionHandler sessionHandler;
    private ConnectionManager connectionManager;
    private String dataAttachedInCurrentRequest;
    private String dataToSendBackInResponse="";
    private static final String DATA = "data";
    private static final int INTERVAL_TO_WAIT = 5;
    private static final long SLEEP_INTERVAL = 1000;
    private static final String MUSES_TAG = "MUSES_TAG";
}
```

As you can see these are all instance and class variables of the servlet class. Some of these members do not violate the thread-safety rule because they at least fulfill one of these criteria:

- A final primitive type e.g. final int, final long
- A final reference type which is immutable e.g. final String

The criteria simply mean that the member will be constant and read only. The rest of the members
could potentially be harmful if special care has not been taken. Special care in this case means exclusive access to the member during read or write. One way to achieve exclusive access is through synchronization blocks or methods. Bear in mind though that exclusive access although necessary for correct functioning, it will slow down the server in terms of performance. The reason exclusive access slows down the server is that it takes away the parallelism and forces sequential execution. Let us look at some of these members:

- The Logger class from the Log4j utility is actually thread-safe [33] so it will not cause any stability issues but it will slow down the server. It slows down by only allowing one thread to log at a time. Apart from that, logging involves a lot of I/O access (writing to disk) and that is also a slow process on its own. The proper solution for this member would be to only do the logging in debug mode (development) and not on release mode (production). One Boolean flag could be used to enable or disable the logging in the entire system.
- The Helper class can be seen as a utility class for extracting the cookie from the HTTP request instance. The Helper class is not thread-safe but that should not matter because the helper should not be a member in first place. A quick fix would be to remove the internal state of helper and make all the methods static. Bear in mind that making helper thread-safe and keeping it as a member is not an option either because helper it not supposed to be a global shared instance between all the clients (Application Scope).
- The sessionHandler internally has a set which contains all the cookies of all clients. The name is misleading because it tries to be a cookie manager on the application level among other things. It is great that the process of adding and removing cookies are handled in a thread-safe manner but the problem is that sessionHandler is not really necessary. Containers like Apache Tomcat already have far better cookie managers so sessionHandler is doing redundant work and slowing down the system.
- The ConnectionManager class contains a queue. The queue in turn contains data that should be sent back to clients. The process of adding and removing elements from the queue is not thread-safe which could lead to data race. The quick fix would be to change the concrete implementation of the queue to one that is thread-safe. The better solution would be to not have a queue for the responses at all. This would require a minor architecture change to the custom developed client and server protocol. Instead of clients pulling the calculated results from the server, the server would push it down to clients when it is ready.
- The dataToSendBackInResponse is a quite self-explanatory. The reads and writes to this reference is not properly synchronized across the application, but that is not even the real issue. Every response should be unique and belongs to one particular request. For this reason the dataToSendBackInResponse should be kept in Request Scope [34]. Having it as an instance variable will making it a global variable just like keeping it in Application Scope. Keeping this member could lead to every client potentially receiving the same response which is a security concern. Once removed as a member it should explicitly be passed to every method that requires it. The same applies to dataAttachedInCurrentRequest.

So to summarize the solution, instance members should be avoided for performance reasons (unless it is read-only) and if it is really necessary then proper synchronization is mandatory for mutable ones.

So we have pointed out and discussed in detail some of the thread-safety related issues in the main servlet. The consequences of a non thread-safe servlet can be quite severe for example it could lead to data corruption under load. So after a meeting on the subject the decision was made to focus on fixing these serious issues. This was an obvious and logical decision to make. The correct
functioning of the server is simply more important that its performance. There is no value in making a wrong server, faster. Hence the MUSES server code base was forked and all the fixes related to the thread-safety were applied to the forked version. Worth mentioning that the scope of the fixes was much broader and not just limited to the mentioned servlet, many other source files had to be modified as they also were not thread-safe. The goal, the whole time was to fix and make the server thread-safe without changing its behavior. The communication interface had to stay the same in order to keep compatibility with the existing clients.

Another modification to the code base that led to performance improvement was the use of fixed size thread pools. The logic for handling incoming events was written in such a way that it created one thread per incoming event to carry out the job. The problem with such approach is that too many incoming events will spawn too many threads. The overhead of creating, destroying and scheduling all those threads will just slow down the system. Hence the code was modified to use a fixed size thread pool. Thread pools can be seen as workers and the incoming events can be seen as work units that will be submitted to the pool for execution.

3.2.3. Apache Tomcat

All the requests/responses in HTTP 1.1 are sent over one TCP connection. The connection is said to be persistent unless it is explicitly declared not to be. The default HTTP connector in Tomcat 6 and 7 is of type Blocking I/O (BIO). BIO will create one thread per connection which means that if 100 clients are connected to the server then 100 threads have been spawned to handle them (assuming keepalive timeout has not been reached). Creating threads and scheduling them do not come free. The overhead must pay off otherwise it is a waste of resource if the connections are not being used effectively. One solution to this has been Non-blocking I/O (NIO) which has a pool of threads that handles all the client connections. The thread will return to the pool (releasing resources) after response has been sent (not the end of stream, but the end of the current content in the stream) instead of waiting for the next request/response. This model is cheaper and uses less threads/resources to achieve the same results. Therefore it scales better which leads to the ability to serve more overall connections on the server [35]. Worth mentioning that the number of concurrent connections that a thread can handle depends on the size of the memory available for heap section.

After inspecting the current configurations used on the MUSES server, it appeared that the default BIO configurations was used. The only modification that has to be done to change this configuration is to change the protocol attribute for the connector tag inside the server.xml file which can be found in Tomcat’s installation folder. See appendices for example.

Apache Tomcat is configured to use very little resources by default. These configurations could be easily modified to let the container allocate more memory and processor cores which would lead to better performance results. The instructions on how to modify such configuration files are provided in the appendices.

As you know the connection between MUSES sever and the clients are encrypted, mainly due to confidentiality and integrity. The encryption of data does not come free without any network and processing overhead. In some scenarios there could be a tradeoff between security and performance that has to be made. Fortunately if the right encryption algorithm is combined with the right hardware then both can be achieved without any compromise. The list of available cipher suites directly depends on which version of java is installed on the machine. Obviously newer version of Java have support for more cipher suites. The currently used cipher suite on MUSES server is SSL_RSA_WITH_RC4_128_SHA. Let us look at it in more detail:

- SSL: Java version 6 has only support for SSLv3 and TLSv1. If MUSES wants to support the
more secure options like TLSv1.2 then it has to upgrade to at least Java version 7 or above.

- **RSA:** Is the asymmetric encryption algorithm that is used for the verification of the certificate and exchanging keys. Bear in mind that this is a one-time cost, so the performance inefficiency of the RSA is not relevant. Later requests will reuse this handshake unless it is explicitly defined not to. Also worth mentioning that the data traffic will not be encrypted by this algorithm.

- **RC4_128:** This is the symmetric algorithm that will be used for encrypting the messages passed between the server and the client. RC4 is a low cost algorithm in terms of necessary processing power, but unfortunately it is considered to be breakable from a security standpoint [36]. Bear in mind that it would still require a fair amount of effort from an attacker to break it, so it highly depends on how much valuable the data is relative to the effort. If the hardware of the server machine contains a decent modern CPU e.g. Intel Core i5 or i7 then RC4 should without a hesitation be exchanged in favor of AES because of the available built in hardware acceleration. AES is more secure and also less taxing on supported hardware.

- **SHA:** Is the hashing algorithm used for checking the integrity of the received message. This is necessary because the server needs to make sure that the content has not been tampered with. Both weaker and stronger variants exist, but nonetheless SHA is a good enough choice. Intel's future CPU architecture (codename Skylake) will have support in form of SHA extension. That is worth considering during hardware upgrades. Meanwhile a little more performance could potentially be squeezed by using MD5. Bear in mind that TLS uses a special version of it called HMAC which is acceptable from a cryptographic perspective.

RC4 is not usually recommended to use on the server side nowadays but it is fast and immune against the BEAST attack which might have been the reason MUSES project picked it in early days. A cipher suite which is safe today might be vulnerable tomorrow, so the choice should be revisited every now and then during security checkups. One way to find out which algorithm is a good fit for the current hardware is to run openssl speed in the terminal for some benchmarking. Just as a reference Amazon.com uses TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256. Galois/Counter Mod (GCM) has been widely adopted because of its performance and efficiency but bear in mind that it is only available in Java 8 and above. See appendices for example on how the cipher suite can be changed on the server side. When choosing a proper cipher suite, one should also take the client into account. What options the client supports and also which one of those give the best security and performance results. It turns out that Elliptic Curve (EC) and GCM is only available in Android 5.0 Lollipop and above. Hardware support by dedicated cryptography instructions for both AES and SHA has been added to the instruction set since ARMv8-A. Samsung Galaxy Note 4 was the first device with ARMv8-A processor architecture. It contained Cortex-A57 and Cortex-A53 cores in a big.LITTLE configuration.

### 3.2.4. Database

The database plays a central role in MUSES server. The surrounding modules are designed in such a way that they will constantly query the database for both retrieval as well as storing the data. For example there is a constant stream of events coming in from all the connected users. These events will all be pushed into the database for later analysis. Depending on many different factors there is a possibility that the database communication channels could be saturated. This is not an uncommon problem among business that deals with such a heavy database usage. Although some suggestions for solving such a problem will be discussed here but keep in mind that some organizations hire a
database expert for the sole purpose of solving such a task, so more can definitely be done in this area for claiming additional performance.

Databases store their records on disk. Disks are great because of their large storage capacity relative to the cost. However the disadvantage is the slow access. One way to increase the performance is to cache the frequently used data in memory. Hibernate is the implementation that have been used in MUSES as Object-relational mapping (ORM) solution. Hibernate does have a first-level cache that is used by default. However the first-level cache is limited to the session scope. If a session has a short life span (which it does in MUSES server) so will the cache. Hibernate does offer a second-level cache for usage but it is turned off by default. The second-level cache is shared between different sessions, and it is associated with the session factory scope, so it has a longer life span. As long as the MUSES server is ran on a single machine there should not be any problems regarding enabling the second-level cache but if the server would be deployed on distributed systems then cache coherency is something of concern. This is due to the fact that some nodes keep their objects in their local cache which is not synced with the global database at all times. In order to solve this Hibernate does offer configurations on object basis, so certain objects that require consistency can explicitly be marked. The appendices include example of Plain Old Java Object (POJO) that has cache enabled.

Java Persistence API (JPA) is a specification for management of rational data in Java. It is simply an abstraction on top on Hibernate. It is a great solution for making the code base independent of a single concrete implementation. The second-level cache is not an exclusive feature of the Hibernate implementation, but it is also available in JPA. Although JPA is present in MUSES server, the project is not completely decoupled from Hibernate due to usage of Session and SessionFactory instead of EntityManager and EntityManagerFactory. The appendices also include an example of a POJO that has cache enabled with JPA.

Another way to improve the performance of the database is to take advantage of the pool connections. The MUSES server already is taking advantage of this mechanism which is great but some minor configurations had still to be done. To be more specific the implementation used is called c3p0 which is one the most mature implementations available.

### 3.2.5. Hardware

Apart from the changes in software, hardware also plays a large role in performance. Hardware upgrades are always an available option for performance increase. In some cases it could be even more cost effective to take this path due to rapid technological progress (Moore's law). Once different hardware configurations become available then it could be interesting to analyze the applications weak scaling property. When we executed our benchmarks, the underlying hardware resource was fixed while problem size grew, this is known as strong scaling property. In weak scaling, the hardware resources would be upgraded with the same rate as the problem itself.

At the moment, MUSES server is ran on one single machine. This is completely fine in development environment, but once deployed in production then vertical scaling might not be sufficient, so additional work might be necessary in order to achieve horizontal scaling. Instead of investing in data centers a cheaper option would be to deploy the MUSES server inside server farms like for instance Amazon AWS or Microsoft Azure. The benefit of such services is their ability to scale on demand, and you would only pay for how much resources you consume.
4. Results

The purpose of this system analysis has been to search for security and performance issues, discussing the consequences and propose solutions for the found problems. The solution to solve all the security and performance issues have been presented either in report, appendix or executed as written code.

The System Analysis chapter has been long and detailed. This chapter will instead focus on results that have been the outcome of this analysis.

4.1. Security

The most common security vulnerability among servers is the vulnerability against injections. So the decision to investigate MUSES server for such weakness was natural. The result of this inspection turned out to be the immunity of MUSES server against SQL/HQL injections due to the use of prepared statements. Other injections like XSS and CSRF did not apply to MUSES project because of the client which lacked web-browser based behavior.

The authentication of server-to-client was based on PKI through certificate pinning. The minor issue was the fact that the certificate was kept on the SD-card of the cellphone which introduced some security concerns. The authentication of the client-to-server was credential based with the lack of salt, nonce, challenge token and a lock out mechanism. Apart from that it was recommended to use the built-in web application security of Java EE in favor of the custom made one. Some attempts have been made to integrate Spring Security framework which is great but it is not completely done yet which leaves more to be desired.

During the process of code audit a more MUSES specific security issue was discovered that was authorization related. The lack of proper authorization allowed clients to insert data into database under the name of other clients.

There were also some hard-coded credentials inside different modules which understandably make the development process easier but obviously go against the confidentiality principle.

Lastly we went through some of the programs installed on the server machine to raise awareness to the number of vulnerabilities that they contain. Although this might seem obvious at first, one should realize that this is the most probable path an attacker will take in order to exploit the system hence the importance.

4.2. Performance

As we have mentioned before, the benchmarking of the server has been done using a custom client application that was developed during the thesis project solely for the purposes of benchmarking. After investigating some of the results the decision were made to rewrite the client application using another framework that was more suitable for heavy usage scenarios like stress testing.

We have looked into the thread-safety of the server in the exclusive access section of the System Analysis chapter. Examples of some existing problems were pointed out and the seriousness was discussed. All the fixes were applied to a forked version of the MUSES server. The motivation for these fixes was first and foremost stability (correct functioning without data corruption) and then performance. Some performance improvements were done by using a fixed sized thread pool instead of spawning many threads.

The default configuration of containers and frameworks may not be the most convenient from a
performance angle, thus they got examined and better options were proposed. One of the options was related to encryption of the communication channel. The suggested change in the cipher suite not only made performance improvements but also made the communication channel more secure. MUSES server makes regular calls to database, so exploring that area and trying to improve its performance was natural. The server already took advantage of prepared statements and pool connections which is great but did not utilize the option for second level caching. So the motivation and instructions for appliance were given.

All the results from the benchmarking sessions have been represented below. The measurements have been done on both the original server as well as the new modified version so that the any improvement in response time could be spotted. Bear in mind though that the original MUSES server code does suffer from thread safety issues. This means that the responses sent back to clients from the server could contain wrong and corrupted data.

Response time as a function of clients

![Graph showing response time as a function of clients](image)

*Illustration 3: Throwing away the cookie so no session is preserved*

Illustration 3 demonstrates the response time of the server to connect attempts. The parameters were set so that each client would try to connect 11 times without keeping the cookie for a session. That means that each request will lead to container create a new session. The connect_before curve is the results of running the benchmark on the original MUSES server code base which does not contain any modifications. As we can see from the resulting curve the response time grows with an exponential complexity which is not great. Worth mentioning that the web application did not respond anymore when 95 clients were sending their requests because the container had already crashed at that point. On contrary the modified version of the server code managed to keep a 5 millisecond response time up to 170 clients. After that the response time started to slightly increase. The source code has been modified in several places but the most probable contributor to this improvement is likely to be the disabling of the session handling in the web application. The container already has the best session handler so doing it in the application as well is redundant job.
Illustration 4: Keeping the cookie in order to preserve a session

Illustration 4 also demonstrates the response time of the server to connect attempts. The parameters were set so that each client would try to connect 11 times. The first connect request is different from the remaining 10 because that request will make the container create a new session. After that the gained cookie will be sent in every request in order to preserve a session. Bear in mind though that each client will have its own session. The purpose of this benchmark compared to previous one is to measure how fast the server responds to requests that already have a session. The average value is noisy at the beginning but becomes more stable later on as the number of requests increase. Larger sample size always leads to more accurate results. We can clearly see that the changes inside the server code have led to improvements in the response time. While the original servers response time grows, the modified version keeps a constant time up to 200 clients. The reason the response time slightly decreases is most probably due the Just-In-Time (JIT) compiler warming up. When some part of the program is executed several times then JIT compiler will detect the hot path and optimize that section which will result in faster execution. Bear in mind that JIT exists on both client and server because they are both developed using Java technology.
Response time as a function of clients

How will the response time grows relative to the increasing number of clients

Illustration 5: Synchronization requests

The synchronization request is a special MUSES protocol request. The client simply asks the server for the latest policies available. The server will then send a JSON object containing some policy rules. The client used for benchmarking will not take any special action with the received JSON data. The client must authenticate itself by logging in before starting to request such synchronization otherwise the server will complain. The time for login request is excluded from the measurements. The cookie must obviously be kept and send in every request in order to keep an authenticated session.

The 5rd illustration reveals that the modified version of the server works as expected while the original server grows rapidly in two stages and then becomes completely unresponsive. The next illustration tells us more about the cause of such behavior.
Success rate as a function of clients

How will the success rate relative to the increasing number of clients

Illustration 6: Original server’s success rate on sync requests

Every response received from the server should contain the HTTP response code 200 (OK) which means that the execution went correctly without any errors. The special client used for benchmarking has been developed in such a way that it expects such response otherwise it gets registered as a failed request. As we can see in the illustration 6, the success rate of the requests sent to the original server falls rapidly which is an indication of internal errors. This could for example be due to some faulty logic throwing runtime exceptions inside the web application. The success rate of the modified version of the server was not included in the graph because it was 100% all the time regardless of number of clients which is how it should be.
Response time as a function of clients

How will the response time grow relative to the increasing number of clients

*Illustration 7: Poll requests*

The clients can retrieve data from the server every now and then by polling with a certain frequency. This is the opposite of a pushing mechanism where the server sends the processed data to the clients when it is ready. Just like the synchronization requests the clients must be logged in before attempting to do any polling. The result shown in illustration 7 is similar to the connect request graph. The original server has a linear complexity while the modified version stays at a constant rate.

Now that we have seen some real figures on how the server performs let put it into the perspective. The modified version of the server has kept a response time in millisecond scale which counts as a very responsive server for such application. We would still be below the acceptable limit even if we added the latency caused from the real-world communication link explained in the previous chapter. Users can certainly tolerate latencies up to a few seconds before having a feeling of slow and sluggish experience. Millisecond latency are only important in applications that have *hard real-time* requirements like for instance first person multiplayer video games. MUSES counts toward applications with *soft real-time* requirements. Missing a deadline every now and then does not cause any disaster contrary to the ABS system of a car which is expected to always react on time.
5. Conclusions and Future Work

The goal of this thesis project was to analyze the MUSES server code base and make it more secure and perform faster. Judging from the job done and the gained results, MUSES server has definitely gone from a slower and less secure state to a more improved one. However this does not mean that the work in this area is completely done. There is always room for improvements. The standards for a prototype that will be evolved and eventually released as a product are quite high as there are some responsibilities involved. The time spent optimizing for improvements must pay off. Time spent versus optimization may not be beneficial in case it has reached the point of diminishing returns.

It is worth mentioning that the execution has had other positive outcomes that were not part of the initial main goals. Stability which is a very important factor has been one the side effects. Another great side effect was the fact that the client program created for benchmarking has been used in another internal project at Sweden Connectivity.

The time dedicated to this thesis project has been limited to one semester. Some of the ideas could not be explored due to the shortage of time but are still worth mentioning as future work projects. Below is a summary of some of these ideas.

- Heed compiler warnings is one of the code practices that leads to better security. Although SonarSource is used in Jenkins, developers are encouraged to also use the -Xlint flag during compilation.
- Fail-safe Defaults also known as Default Deny is another important security principle. Not only should the server logic fulfill this principle but also every defined MUSES policy.
- Every program and user on the server should run with least privilege necessary. For example the MUSES server web application has root privileges to the database. This makes the development process easier but is definitely not essential.
- Letting some White Hat hackers that are not aware of the internals of the MUSES server (Black box) hack into the server their own way. The information gained from such process could be very useful.
- An Intrusion detection system like for example Snort could also be very useful security addition. Such software combined with proper rules could detect and prevent attacks.
- There might be some design patterns that would be beneficial to the MUSES server from a performance perspective. For example patterns like Actors, Fork/Join (Java SE 7) or Streams (Java SE 8) can be very beneficial for the sections that do concurrency and parallel processing of the data.
- The client program that was developed for benchmarking reasons has turned out to be useful even in other areas. The client can be more utilized by expanding its feature set. For example the client can be used in the testing phase in order to see if a server build is functioning correctly as expected.
- Programs like ApacheBench or Apache JMeter are designed for load tests and performance measurements. The challenge in using such tools could be the fact that MUSES has its own custom developed protocol on top of HTTP for communication so using these tools might require some proper altering.
- Database normalization and query tuning by an expert could lead to some performance improvements.
- The client is designed to retrieve data from server every now and then through polling. Alternatives like pushing the data from server to client are worth exploring as they have less
overhead. Examples of such technologies are the WebSocket standard which could be used in favor of HTTP.

- As suggested in sub-chapter 3.2.5, some data on weak scaling property of the web application on different hardware configurations could be of interest.
6. Bibliography


Appendix A: Apache Tomcat

The server.xml file can be found in CATALINA_HOME which is Tomcat's installation folder.

In order to use the database as a realm add the Realm tag just bellow the already existing one.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Server port="8005" shutdown="SHUTDOWN">
...
<Service name="Catalina">
...
<Engine defaultHost="localhost" name="Catalina">
  <Realm className="org.apache.catalina.realm.JDBCRealm"
    driverName="org.gjt.mm.mysql.Driver"
    connectionURL="jdbc:mysql://localhost/tomcat" connectionName="tomcat"
    connectionPassword="tomcat123" userTable="admins"
    userNameCol="username" userCredCol="password" userRoleTable="roles"
    rolenameCol="rolename" digest="md5"/>
</Engine>
...
</Service>
</Server>
```

To change the configuration of the HTTPS connection, change the cipher suite. The cipher suite is an attribute of the connector tag in server.xml.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Server port="8005" shutdown="SHUTDOWN">
...
<Service name="Catalina">
  <Connector ... ciphers="SSL_RSA_WITH_RC4_128_SHA, SSL_RSA_WITH_RC4_128_MD5"
    .../>
...
</Service>
</Server>
```

In order to enable NIO change the protocol attribute in server.xml. It should look similar to this after modification:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Server port="8005" shutdown="SHUTDOWN">
...
<Service name="Catalina">
  <Connector connectionTimeout="20000" port="8080"
...
</Service>
</Server>
```

In order to increase the amount of memory used by Tomcat and letting it take better advantage of multicore processors change the value JAVA_OPTS in /etc/default/tomcat7. It should look similar to this after modification:

```
JAVA_OPTS="-Djava.awt.headless=true -Xms128m -Xmx384m -XX:+UseConcMarkSweepGC -XX:+CMSIncrementalMode"
```
Appendix B: Database

This is how SimpleEvents.hbm.xml would look like with a cache tag:

```xml
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN" "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<!-- Generated 27-Nov-2014 11:36:48 by Hibernate Tools 3.4.0.CR1 -->
<hibernate-mapping>
  <class name="SimpleEvents" table="simple_events" catalog="muses">
    <cache usage="read-write" />
    <id name="eventId" type="java.lang.Long">
      <column name="event_id" />
      <generator class="identity" />
    </id>
    <many-to-one name="users" class="Users" fetch="select">
      <column name="user_id" not-null="true"/>
      <comment>FK to table USERS(user_id)</comment>
    </many-to-one>
  </class>
</hibernate-mapping>

The Strategy used in the above example has been read-write but keep in mind that different persistence classes might require different strategies. Refer to Hibernate documentation for more information on strategy properties. Apart from the strategies, the cache provider also has to be defined in the Hibernate configuration file hibernate.cfg.xml. Bear in mind that different option are available.

```xml
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD 3.0//EN" "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <property name="hibernate.dialect">org.hibernate дialect.MySQLDialect</property>
    <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
    <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/muses</property>
    <property name="hibernate.connection.username">test</property>
    <property name="hibernate.connection.password">test</property>
    ...
    <property name="hibernate.cache.provider_class">
      org.hibernate.cache.EhCacheProvider
    </property>
    <!-- mapping -->
    ...
    <mapping class="eu.musesproject.server.entity.SimpleEvents" />
  </session-factory>
</hibernate-configuration>
```

In order to make an object eligible to caching in JPA we need to use the @Cacheable annotation. An example when @Cacheable has been added to SimpleEvents.java:

```java
package eu.musesproject.server.entity;
```
import java.io.Serializable;
import javax.persistence.*;
import java.sql.Time;
import java.util.Date;
import java.util.List;

/**
 * The persistent class for the simple_events database table.
 */
@Entity
@Table(name="simple_events")
@NamedQuery(name="SimpleEvents.findAll", query="SELECT s FROM SimpleEvents s")
@Cacheable(true)
public class SimpleEvents implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    @Column(name="event_id", unique=true, nullable=false)
    private String eventId;
    ...
}

Apart from the annotation above we need to specify a global setting as well. This will be done inside the persistence.xml file.

<persistance version="1.0" encoding="UTF-8"?>
  <persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence" http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
    <persistence-unit name="server">
      <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
      <class>eu.musesprojec.server.entity.SimpleEvents</class>
      ...
      <shared-cache-mode>ENABLE_SELECTIVE</shared-cache-mode>
      <properties>
        <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/muses" />
        <property name="javax.persistence.jdbc.user" value="muses" />
        <property name="javax.persistence.jdbc.password" value="muses11" />
        <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver" />
      </properties>
    </persistence-unit>
  </persistence>

The ENABLE_SELECTIVE will permit caching only for objects that have @Cacheable. If you on the other hand would like to skip the @Cacheable and make all objects cacheable in one go use ALL. Other options are also available, refer to Java EE documentation for more information.
Appendix C: Benchmarks

The MUSES client and server use HTTPS for sending JSON objects to each other. The content of the JSON object is based on a custom protocol that the MUSES team have created. The type of the request can be chosen by setting a value in the POST request header. The value for example can be connect, poll, data (used for login or local_decision) or etc. Apart from the connect, synchronization and poll benchmark that we saw in the result chapter of report some other measurements have also been done. These are the results from those benchmarks:

Response time as a function of clients

![Response time graph]

**Illustration 8: Login attempts**

Illustration 8 demonstrates the response time of the server to login attempts. Even though repeated login attempts are a security concern, it could still be a great reference for performance analysis. The parameters were set so that each client would try to login 11 times although only the last 10 will be apart of the measurement. We can see in the far right side that it almost takes 350 milliseconds for the original server to respond to 10x150=1500 login requests. According to the results, the login after the optimizations look better on average than the results from the original login measurements.

Bear in mind though that the authentication mechanism in the server will eventually change. Right now the web application itself is doing the job of authentication but the future plans indicate that this task will be left to container to handle. So this benchmark will be less relevant in the future when Spring Security is fully implemented and utilized.

38
When the client can not make a decision on its own then it will ask the server by sending a decision request. During the measurements when the SimulateManyHttpClients sent the decision requests, some HTTP 500 Internal Server Errors where sent back as a response code which led to poor successful response rate. However this was not the case for modified version of the server code. Worth mentioning that the response time of 16 clients were so long that it had to be interrupted. The MUSES server is programmed in such a way that it gives itself 5 seconds to prepare the response. That is the reason average time for the fixed server code stays above 5000 milliseconds. This threshold value can easily be tuned inside the code if necessary so it is rather a design decision than a performance issue. Illustration 10 shows the success rate graph of the original server.
Success rate as a function of clients

How will the success rate be relative to the increasing number of clients

*Illustration 10: Success rate of the original server while getting decision requests*