

Online MBPeT Dashboard - Enabling Performance Testing as a Service in the Cloud

Aaron Pratt

Master of Science Thesis
Supervisors: Dragos Truscan, Tanwir Ahmad
Software Engineering Laboratory
Computer Engineering
Faculty of Science and Engineering
Åbo Akademi University
April 2016

ABSTRACT

In this thesis, tool support is addressed for the combined disciplines of Model-based testing and performance testing. Model-based testing (MBT) utilizes abstract behavioral models to automate test generation, thus decreasing time and cost of test creation. MBT is a functional testing technique, thereby focusing on output, behavior, and functionality. Performance testing, however, is non-functional and is concerned with responsiveness and stability under various load conditions.

MBPeT (Model-Based Performance evaluation Tool) is one such tool which utilizes probabilistic models, representing dynamic real-world user behavior patterns, to generate synthetic workload against a System Under Test and in turn carry out performance analysis based on key performance indicators (KPI). Developed at Åbo Akademi University, the MBPeT tool is currently comprised of a downloadable command-line based tool as well as a graphical user interface. The goal of this thesis project is two-fold: 1) to extend the existing MBPeT tool by deploying it as a web-based application, thereby removing the requirement of local installation, and 2) to design a user interface for this web application which will add new user interaction paradigms to the existing feature set of the tool. All phases of the MBPeT process will be realized via this single web deployment location including probabilistic model creation, test configurations, test session execution against a SUT with real-time monitoring of user configurable metric, and final test report generation and display.

This web application (MBPeT Dashboard) is implemented with the Java programming language on top of the Vaadin framework for rich internet application development. The Vaadin framework handles the complicated web communications processes and front-end technologies, freeing developers to implement the business logic as well as the user interface in pure Java.

A number of experiments are run in a case study environment to validate the functionality of the newly developed Dashboard application as well as the scalability of the solution implemented in handling multiple concurrent users. The results support a successful solution with regards to the functional and performance criteria defined, while improvements and optimizations are suggested to increase both of these factors.

Keywords: model-based testing, performance testing, cloud, rich internet application, Vaadin, tool support

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my supervisor, Adjunct Professor Dragos Truscan. Thank you first for granting me the opportunity to work on this project and for giving me this task. I am very grateful for your support, trust, encouragement, guidance, explanations, and patience. I greatly enjoyed the times of collaboration and discussions. I wish to express my sincere thanks to Tanwir Ahmad for all of the assistance and guidance during the development process. Thank you for your quick responses, positive attitude, and hard work.

Emily, Isabella, and Timothy, you are the sweetest children anyone could ever wish to have. You have shown such understanding during the times when I couldn't stop and play, but you nevertheless poured out your love for me. I will always love you, no matter what, because you are mine and I am yours, and the more I love you, the more beautiful and wonderful you are.

Sara, you have been such a tremendous help and support during this entire process. You were exceptionally encouraging, invested, excited, you sacrificed so much time and energy, you took on more responsibility at home and with the kids, and you did it all with joy and love. You are a joy to come home to and the one that I want to share everything with. I love you and appreciate you immensely. You have been a rock for me so many times for which I am supremely thankful.

I want to thank all of my family, immediate and in-laws, for your ongoing love, support, assistance, and care. It is such a blessing to call you family and friends.

Jesus, I am first and foremost indebted to you. Every good thing I can lay claim to has come straight from you, and you continue to carry me through the hard. I am eternally grateful for your grace. You are *the* rock in my life, and even if every other good thing faded away, you would still be my treasure and the one thing that I ultimately need.

CONTENTS

Abstract	I
Acknowledgments	II
Contents	III
Abbreviations and Terms	V
List of Figures	VII
1. Introduction	1
1.1 Scenario Introduction and Background	1
1.2 Project Objective.....	5
1.3 Thesis Structure	5
2. Background	7
2.1 Software Testing	7
2.1.1 What is it...or isn't?	7
2.1.2 Why is it important?.....	8
2.1.3 How is it done?.....	9
2.1.4 Performance Testing.....	11
2.1.5 Model-Based Testing	13
2.1.6 Model-Based Performance Testing	15
2.2 Cloud & cloud services	17
2.2.1 Advantages of Cloud Solutions	18
2.2.2 Disadvantages of Cloud Solutions	19
2.2.3 Cloud Service Models.....	20
2.3 Rich Internet Applications.....	22
2.4 Vaadin framework.....	23
3. Overview of the MBPeT Tool	26
3.1 Tool Architecture.....	26
3.1.1 Master Node.....	27
3.1.2 Slave Node	28
3.2 Tool Operational Process	29
3.2.1 End-User Interaction.....	30
3.2.2 Load Generation Process.....	31
3.3 Probabilistic Timed Automata Models	32

4. Project Requirements	36
4.1 Project Statement.....	36
4.2 Functional requirements	37
4.3 Non-Functional requirements	38
5. Implementation	40
5.1 Approach.....	40
5.2 Development Environment	41
5.3 Deployment Environment.....	42
5.4 Development Process	42
5.4.1 Technologies Used	42
5.4.2 Application Structure	43
5.4.3 MBPeT Connection and Combined Architecture	51
5.4.4 System Inputs and Outputs	53
5.4.5 Additional Components and Contributions	57
6. Case Study	64
6.1 Case Study: YAAS.....	64
6.2 Test Architecture and Tooling	65
6.3 Experiment Scenario.....	66
6.4 Experiment 1	68
6.5 Experiment 2	72
6.6 Experiment 3	74
7. Evaluation.....	77
7.1 Case Study Analysis.....	77
7.1.1 CPU	77
7.1.2 Memory	79
7.1.3 Optimizations.....	81
7.2 Implementation Lessons Learned	82
8. Conclusion.....	85
Bibliography.....	87

ABBREVIATIONS AND TERMS

AJAX	Asynchronous JavaScript and XML
CLI	Command-line Interface
DOM	Document Object Model
Flex	Fast Lexical Analyser
GUI	Graphical User Interface
GWT	Google Web Toolkit
IaaS	Infrastructure as a Service
IP	Internet Protocol
IT	Information Technology
JavaScript	An interpreted or scripting language for the web
JDBC	Java Database Connectivity
JPA	Java Persistence API
JSON	JavaScript Object Notation
JSP	JavaServer Pages
JVM	Java Virtual Machine
KPI	key performance indicators
LAN	local area network
MBPeT	Model-Based Performance evaluation Tool
MBT	Model Based Testing
MVC	Model-View-Controller
MVP	Model-View-Presenter
NIST	National Institute of Standards and Technology
PaaS	Platform as a Service
POJO	Plain Old Java Object
PTA	Probabilistic Timed Automata
RAM	Random-access Memory

RIA	Rich Internet Application
SaaS	Software as a Service
SSH	Secure Shell
SSHD	Solid State Hybrid Drive
SUT	System Under Test
TCP	Transmission Control Protocol
TRT	Target Response Time
TS	Test Suite
UDP	User Datagram Protocol
UI	User Interface
VU	Virtual User(s)
WAMP	Windows, Apache, MySQL, PHP
XML	EXtensible Markup Language

LIST OF FIGURES

Figure 1: Relative cost of error correction [10]	9
Figure 2: Testing Strategies [9]	10
Figure 3: V-model of software development and testing.....	11
Figure 4: Performance Testing Core Activities [12]	13
Figure 5: MBT Test Process (testing tools shown in bold boxes) [8]	16
Figure 6: Cloud Architecture of Service Model Layers [14].....	20
Figure 7: The MVC Model in a WebApp Implementation [9]	23
Figure 8: Vaadin Framework and Runtime Architecture [23]	25
Figure 9: Distributed Architecture of MBPeT tool [7]	26
Figure 10: Master Node [7].....	28
Figure 11: Slave Node [7]	29
Figure 12: PTA model [4].....	34
Figure 13: PTA model for YAAS application [4, 7].....	34
Figure 14: WebApp + MBPeT Connection	41
Figure 15: External libraries referenced by the Web-app project	43
Figure 16: Navigation and Views (plus Registration Window)	46
Figure 17: Ace Editor Add-on Component	48
Figure 18: Diagram Builder View + Corresponding DOT Model	49
Figure 19: Master Terminal Window during Test Session	52
Figure 20: Model Input in Dashboard app	55
Figure 21: Settings Input in Dashboard app.....	55
Figure 22: Adapter Input in Dashboard app	56
Figure 23: I/O for MBPeT Web Application.....	56
Figure 24: Final Test Report Output from Dashboard app	57
Figure 25: Optional test configurations window	59
Figure 26: Monitoring Tab during test session, with aggregated and individual response times highlighted.....	60
Figure 27: Web-App / MBPeT shared directories.....	61
Figure 28: Non-bidder User Model	64
Figure 29: Passive User Model.....	65
Figure 30: Test Architecture	65
Figure 31: Combined Performance on Host Server. Experiment 1a	69
Figure 32: VisualVM-Tomcat CPU and Memory charts. Experiment 1a	69
Figure 33: Reduced page size and load time with gzip compression	70
Figure 34: VisualVM-Tomcat CPU, Memory, and Threads charts. Experiment 1b	71
Figure 35: Combined Performance on Host Server. Experiment 1b	72
Figure 36: VisualVM-Tomcat CPU, Memory, and Threads charts. Experiment 2	73

Figure 37: Combined Performance on Host Server. Experiment 2	74
Figure 38: VisualVM-Tomcat CPU, Memory, and Threads charts. Experiment 3	75
Figure 39: Combined Performance on Host Server. Experiment 3	76
Figure 40: CPU Utilization by Experiment	78
Figure 41: Projected CPU usage by web-app alone and combined load at given user count.....	79
Figure 42: Memory Consumption by Experiment	79
Figure 43: Project linear memory consumption by web-app alone and combined load at given user count.....	80

Chapter 1

1. INTRODUCTION

A core principle in the vast category of software engineering is that of software testing. As demands for new systems, new functionality, and higher quality rise, the importance of software testing increases in order to accomplish these goals. The practice of testing software systems is not new and spans all aspects of system development, from start to delivery (beyond simply completing the development tasks). With many abstraction layers and various techniques to test each one, the time and cost of properly testing can skyrocket.

Model-based testing (MBT) is a functional testing technique which utilizes abstract models to automate test generation. These probabilistic models resemble actual end-user behavior. Although model-based testing is not new, it remains however under-utilized in the industry even though it offers significant advantages over other testing techniques. While functional testing concentrates on system output, behavior, and functionality, *performance testing* is a discipline concerned not with the functionality of a system but rather with the effectiveness, responsiveness, and scalability of the system.

The underlying work which this thesis is founded upon is the development process of a *rich internet application* (RIA) as a Web-based GUI (graphical user interface) to an existing tool. This tool, MBPeT (Model-Based Performance evaluation Tool), is a model-based performance evaluation tool that was developed at Åbo Akademi University in Turku, Finland. This tool seeks to combine the advantages of MBT and probabilistic models to the field of performance testing of web applications. Further explanations of this tool are offered below in Section 1.2 Project Objective.

The aim of this thesis is not only to document the problem, chosen solution, and implementation of this engineering project, but also to provide a study into the technology behind the MBPeT tool. Finally, an evaluation of the performance requirements of the newly developed rich internet application (RIA) and the framework chosen to accomplish this task will be given. The web application development will henceforth be referred to by its title, the MBPeT Dashboard.

1.1 Scenario Introduction and Background

Technology is at the forefront of the fast-changing world as we know it today. Many periods of past history were defined by dominant trends such as the Industrial Revolution of the 18th and 19th centuries. The technological revolution

is nothing new, but the world still remains in the thick of this society altering trend. Software plays a critical role in the ever increasing innovations of new devices and applications which both business and personal lives depend upon. In both business and personal life, costumers increasingly demand that these devices and tools not only function as promised, but also that these devices function consistently, fast, and reliably. This means that the software must be of high quality, reliable, fault tolerant, scalable, efficient, etc. For example, a smartphone must be able to not only perform the various tasks of the operating system and applications, but it must perform these tasks consistently, almost instantaneously, and without excess drain on the battery so that it will last a whole day. These requirements go beyond pure functionality and into performance and quality assurance. Thus, a key to success in today's industry and market is high end performance and quality, and that better than the competitors'. Though proper design and implementation is important, software testing is the dominant method used to achieve these high levels of system performance and quality. In order to achieve these high standards, the large amount of time required for testing becomes very expensive. This can result in various outcomes. For example, testing processes are minimized, leading to lower quality assurance of a product. Alternatively a larger budget is dedicated to proper testing leading to more expensive products; or a compromise of the two might be reached.

Since high quality and performance are the key characteristics required for success in this discussion and software testing is the primary method of achieving or proving these goals, then a working definition must be attained. Various philosophies and ideas exist about what exactly software testing is. *Software testing*, according to William Hetzel, is “*the process of establishing confidence that a program or system does what it is supposed to*” [1]. Glenford Meyers wrote that “*Testing is the process of executing a program or system with the intent of finding errors*” [2]. The common fallacy that many beginners take when approaching testing is the assumption that the end goal of testing is to demonstrate the absence of any errors, or to show that a program performs all of its intended functions with 100 per cent perfection. Research has proven, however, that a more negative approach is far more effective. E.W. Dijkstra pointed out that “*program testing can be used to show the presence of bugs, but never to show their absence*” [3]. Based on these definitions, rather than trying to prove that no errors exist, software testing aims at finding as many bugs as possible in order to correct the faults and avoid failures, thereby checking that the system meets the pre-defined requirements and satisfies certain agreed upon level of quality. Therefore, rather than proving that no errors exist, the job of testing is to prove that errors *do* exist, to reveal any defects that are present.

Even after the goals of testing are defined, the methods by which these criteria are met vary in approach and practice. The general process of discovering the existing defects (or faults) is by executing a test or series of tests (known as a *test suite* - TS) against the particular *system under test* (SUT). Depending on which abstraction layer is under consideration, various types of testing can be executed. A software test typically compares the actual output of a program (measured by the executed test) against the expected correct output for any specific known input. A simple example would be that if a system is supposed to double any input value, then the expected output of inputting the number 2 would be the number 4.

Functional testing is one type of testing and is defined by [4] as “looking for defects in the software that are related to the system’s functionality”. These defects could include for example wrong behavior, erroneous output, missing functionality, etc. The key question in functional testing is “does the system fulfil the functional requirements”? Back to the previous simple example, does

$$2 \times 2 = 4 ?$$

However, not all requirements or potential defects are related to functionality. Some of these other aspects of a software system might include performance, usability, reliability, scalability, and security. *Non-functional testing* describes these types of testing where the focus is not aimed only at “does the system perform the functional requirements”, but the tests aim to show if the system can still perform these requirements under various scenarios. In other words, what conditions might alter the behavior of the SUT? Is the system secure? If not, a security breach could very well end up in a broken system that is no longer to fulfil its functionality (security testing). Will the system handle not only 1 or 5 users, but 100 or 1,000 or more users? Is the system intuitive or easy to use or can the outputs be misinterpreted (usability testing)?

Performance testing is of particular concern for this thesis work. This non-functional type of testing is concerned with performance-related concerns, such as how a system functions in regards to responsiveness and stability under various load conditions. It may not be enough for a system to operate normally with only 20 users but to result in undesired behavior with 100 users. A system may be designed or implemented in such a way that a heavy workload (too many concurrent users, too low system resources, etc.) causes slow response or even system failure. Therefore, when running performance tests, the area of concern moves from whether the output is correct or not to such concerns as how many actions can be processed per cycle (throughput), or how long do certain actions take between input and output (response time). These performance testing

paradigms include load testing, stress testing, scalability testing, and capacity testing.

The areas of concern for the system behind this work include response times, throughput, scalability, reliability, success/error rate, and resource utilization. These coincide with interest areas of traditional performance testing. Performance testing does have drawbacks however. First, like many testing disciplines, performance testing requires significant manual effort. When tests must be manually created, problems arise when the system inevitably changes throughout the development life-cycle. When the code changes the test scripts must also be altered. This is not only difficult, but it also leads to a following common problem which is that testing is typically time consuming and therefore extremely costly.

Model-based testing (MBT) is somewhat of a newcomer, compared to longstanding testing techniques, and it offers possible solutions to the pitfalls of conventional functional testing approaches. Since the main drawback of traditional testing is the tedious, manual test creation process which requires significant time and cost, model-based testing seeks to solve this by raising the abstraction layer. Rather than writing dozens or hundreds of test cases manually, this is accomplished by the test engineer—IT (Information Technology) professional responsible for one or more technical test activities such as designing test input, producing test case values, running test scripts, analysing results, and reporting to managers [5]—who writes abstract models that represent expected behavior of the SUT. The testing tool can then automatically generate test cases from that model and then execute them against the system under test. Since this process removes the most time consuming task of test creation, MBT can greatly reduce the overall testing time and cost.

Although research and academics have pushed for MBT, it remains less utilized in the professional fields than do more traditional testing practices. Although MBT is a functional testing paradigm, some effort has been given to prove that the philosophy of the model-based approach can be adopted and implemented in a performance testing scenario [6]. Åbo Akademi University is one such university that has invested research and development in the field of model-based testing practices. Specifically, the MBPeT tool has been developed in the Software Engineering Laboratory as one such tool in order to assist the performance testing process. MBPeT is a performance testing tool essentially concerned with two things: load generation and system monitoring. MBPeT first focuses on creating abstract, probabilistic models which simulate *virtual user* (VU) types. These user models have an element of randomness to them, in contrast to static test scripts, which more accurately simulate various types of users, e.g. passive, normal, aggressive user etc. Second, using these models, the

system generates a synthetic workload and executes that against the SUT. By extending MBT principles and focusing on performance specific aspects, in contrast to functional aspects, MBPeT monitors system performance under the duration of the test session [7]. In particular, various *key performance indicators* (KPIs) are monitored during the test session which can include response times, throughput, memory and CPU usage, etc.

1.2 Project Objective

Previously, MBPeT has been available only as a stand-alone, downloadable application which can be run on a Windows or Linux based machine. The tool is currently executable via either a command-line interface (CLI) or GUI. However, this execution is limited to local installations. These limitations are the basis for this thesis work. The two primary tasks are twofold. First, to develop a new web-based *user interface* (UI) for the tool that allows remote execution as a service via web technologies in a user friendly environment. This should accommodate the existing input and design methods while also introducing new user functionality for the model design process, such as drag-and-drop model building (see Section 5.4.2.3 Add-ons). Second, by working together with the existing development team of MBPeT, the tool should be centralized and accessible without any end-user installation required. Scalability should also be considered in the new deployment solution in order to accommodate multiple users on a single execution environment. This should be accomplished by deploying the new front-end UI, as well as the MBPeT tool itself, to a web server which is accessible via the internet.

The purpose of this thesis is to document the theory, design, implementation, and results of this development project. The pre-existing components include the working MBPeT system as documented by [7]. The aforementioned team will make all changes to the MBPeT tool that are necessary to accomplish the tasks of this project including allowing connectivity by the new UI, and restructuring any functionality in order to accommodate this new interface with its new deployment solution. The responsibility of this work is to design and implement the user interface, enable connectivity and interaction with MBPeT, deployment to the web server for production use, and finally evaluate the techniques used as well as certain performance requirements.

1.3 Thesis Structure

Chapter 2 will provide the reader with the theoretical background on the problem area and specific disciplines that form the foundation of this work and the existing MBPeT tool. This will give a more complete understanding of what

problem is being solved by this form of software testing, and this tool in particular. Also, this chapter will lay the technological foundation for the development process used in this work. Chapter 3 continues with an overview of the existing MBPeT tool. This summary will focus on two aspects, 1) summarize the *tool's architecture*, and 2) describe the operational process of MBPeT in what steps are carried out through the life-cycle of a test session. By explaining not only what the tool does, but also how its various components are related and function together, we will provide a foundation for understanding the testing concept in use. Likewise this will give a glimpse into the design decisions made for the Dashboard application. Chapter 4 will summarize the problem being solved as well as present the requirements (functional and performance) set on this new system. Chapter 5 details the design and development stages of the actual engineering tasks that were carried out to accomplish the project goals. Chapter 6 presents a case study scenario in which several experiments will be carried out to prove the effectiveness of the solution implemented. In Chapter 7, the results will be analyzed and evaluated to show how well the goals of the thesis project were fulfilled. Finally, Chapter 8 provides a brief conclusion and a summary of the final results.

Chapter 2

2. BACKGROUND

This chapter is a discussion of the most relevant theoretical topics related to this work. There are in essence two categories of topics that are relevant to this thesis. The first category is the discipline of software engineering known as software testing. This is the realm that the MBPeT tool functions in, as has been previously introduced in some detail. In addition to testing, the subject of the cloud and cloud services are discussed in this section, as the MBPeT tool targets the testing of web-based applications and services, in contrast to local OS applications. The second category discussed focuses on software engineering specifically related to internet technologies. This section provides the theoretical and technical background for the software development project which this thesis is primarily concerned with. This project work was introduced in Section 4.1 Project Statement as the development of a rich internet application interface for the existing MBPeT tool. These background subject matters include web applications, web development processes and technologies, and specific frameworks for performing such tasks.

2.1 Software Testing

2.1.1 What is it...or isn't?

The job of a software developer is to build a system that performs the intended functions correctly. It is a common mistake to bring this ideology over into the realm of software testing and then make the false definition that:

“Testing is the process of demonstrating that errors are not present.”

Or even to say that:

“The purpose of testing is to show that a program performs its intended functions correctly.”

Even though these seem accurate at first value, [2] duly points out that these definitions are actually upside-down.

Just as creating a software system is supposed to add functionality and value, so testing itself should add value to the system. A test that shows no existing errors has added no value to the overall project. However, by uncovering errors in the system, thereby allowing them to be corrected, testing has now provided

opportunity to raise the level of reliability and over quality of the software. As upside-down as this can appear, finding and removing errors lead to greater quality, not proving that no errors exist [2]. So if the assumption of a test engineer is that errors exist and he must find them, his job task and definition would more appropriately be:

“Testing is the process of executing a program with the intent of finding errors.”

Similarly, testing is defined [8] by as:

“An activity performed for evaluating product quality, and for improving it, by identifying defects and problems.”

So there is a sort of paradox to properly understanding testing. This paradox is that a failure to find a failure is a failure, not a success. Or as a common adage states, “If it ain’t broke, you’re not trying hard enough”. The goal is not to find no faults and thereby “prove” quality, but rather to find as many faults as possible, in order to fix them and improve quality.

2.1.2 Why is it important?

If the practical goal of testing is to find errors, then the motivational goal is to guarantee a certain level of quality. Since most systems are built for customers, either corporate or consumer, then the project is a business endeavour. As such, any project should follow a budget. Testing can often account for more time and effort than any other aspect of a software project [9]. Figures vary, but it is not uncommon to allocate 40-60% of a total project’s effort on testing alone [5]. Not only is it important and costly, but when testing is postponed, the resulting cost of finding and correcting errors increases exponentially (find more on test techniques and phases in Section 2.1.3 How is it done?). Figure 1 nedan depicts the growth rate of cost as a project moves further towards completion.

Testing, then, is important not only to guarantee a certain quality level but also is a critical element to the cost of a software project. The later testing is performed, coupled with the lack of discovering existing errors, together introduce exponential growth in overall cost.

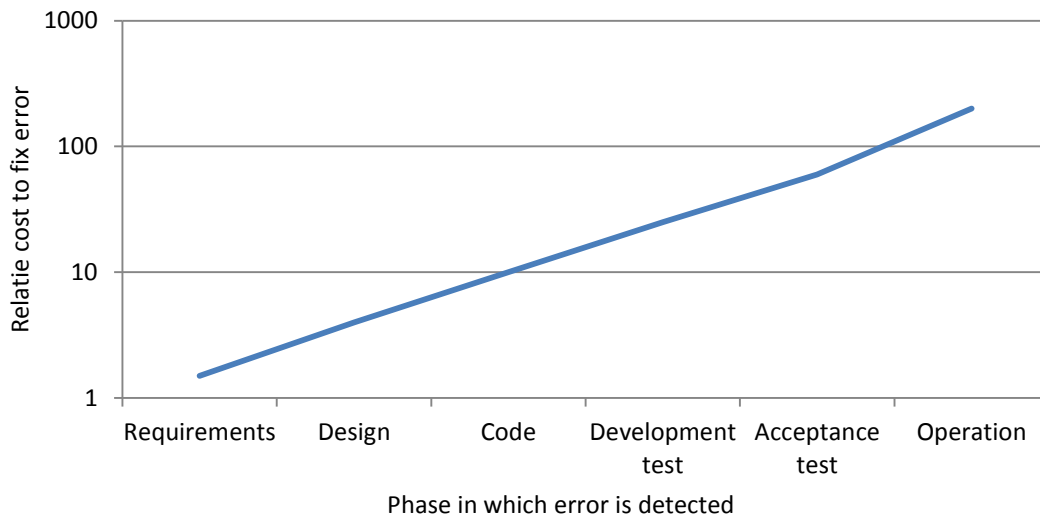


Figure 1: Relative cost of error correction [10]

2.1.3 How is it done?

Before looking at test techniques, it is helpful to define what these “errors” are that a test engineer is looking for. The term “bug” is widely used to generically refer to a problem in a piece of software. While there is historic significance to the origin of this term, it is nonetheless misleading. [5] argues that “a bug” should be rather termed an *error* because it honestly places the blame squarely on the programmer who wrote the code, rather than shifting the blame to some mythical “bug”.

Errors can originate from numerous sources. A far from exhaustive list would include things such as:

- Hardware malfunction
- Function errors
- Requirements errors
- Design errors
- UI errors
- Logic errors, e.g. calculation errors
- Program structure errors

Because of the various sources of possible origins of errors, there have developed many approaches to testing that could be categorized as *techniques* and *strategies*.

Some testing techniques are classified depending on what criterion is used to measure the adequacy, or coverage, of a set of test cases (commonly known as *test suite*). Coverage-based testing, fault-based testing, and error-based testing are three such techniques. Other techniques are classified based on the amount of source information available to the test. Black-box and white-box testing are two such techniques. Black-box refers to “deriving tests from *external* descriptions of the software, including specification, requirements, and design”; while white-box refers to “deriving tests from the source code *internals* of the software, specifically including branches, individual conditions, and statements” [5]. The former focuses on specifications without explicit knowledge of the underlying code, while the latter focuses on structural implementations of the code itself. [9] [11]

These various strategies exist because each one focuses on a specific aspect of the overall process or system. Testing often begins at the component level (code level) and works progressively outward toward the integration of the entire system, or through the delivery and acceptance process. Figure 2 portrays this outward move of the testing cycle, beginning at the innermost level concentrating on the unit level (method, components, class, etc.) and moving outward to integrating the entire system as a whole.

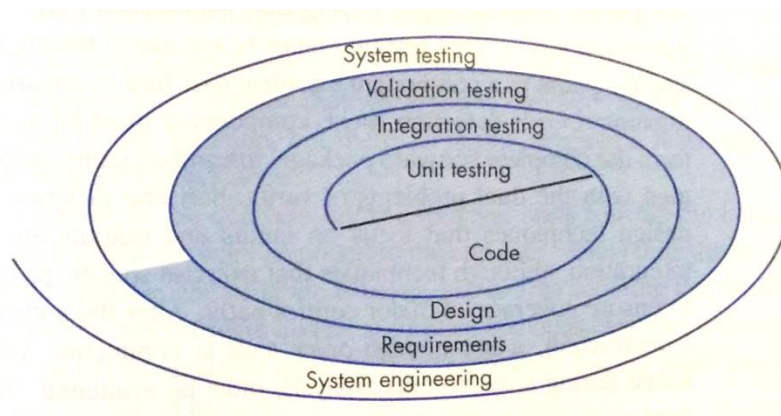


Figure 2: Testing Strategies [9]

Another common way to depict the various stages where the different testing strategies and techniques are implemented as the well-known V-model that connects the software development processes with their corresponding testing procedures.

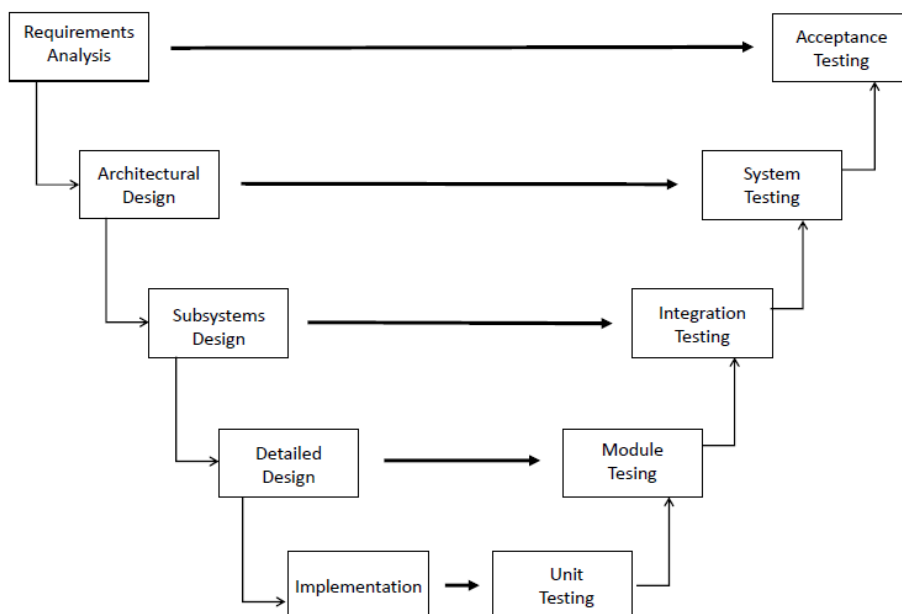


Figure 3: V-model of software development and testing

Another breakdown of a system can be done based on functional vs. non-functional requirements. This was already introduced in the Chapter 1 as the difference between the system’s actual functionality (output, behavior, etc.) and secondary properties like performance, security, scalability, etc. Since these distinctions in requirements exist, there are therefore distinctions in testing processes that fit each requirement tactic. The testing procedures shown in Figure 2 and Figure 3 are mainly applicable to functional type requirements because they focus on the question, “does the system fulfil its specified functionality”? Security, usability, and performance are separate issues, and can meet their requirements (or not meet them) regardless of if the functional requirements are fulfilled, and vice versa. For example, a system might do everything it is required, while performing well or poorly under a certain load. Of course load, security, scalability etc. can cause breaks in functionality, but the two entities are still separate concerns.

Because MBPeT is a performance testing tool, attention is now turned specifically to non-functional requirements related to performance.

2.1.4 Performance Testing

Chapter 1 gave a short summary of two software testing categories, namely functional testing and performance testing. The former dealing with a system’s *actual* functionality and output etc. while the latter focuses on the system

performance while it carries out these functional requirements. Performance testing is defined by [12] as:

“Performance testing is a type of testing intended to determine the speed, effectiveness, responsiveness, reliability, throughput, interoperability, and scalability of a system and/or application under a given workload.”

It is possible to apply performance testing to numerous targets such as software applications, system resources, targeted application components, databases, network bandwidth, and more. Not only does this provide information about the SUT itself, but also provides insight into potential bottlenecks and points of failure which could originate in the actual software, the computer hardware, or the network. [12] Because numerous specific aspects fall under the larger category of performance testing, they each become a testing practice of their own together with their own objectives and techniques. A few of the most common performance testing paradigms are load testing (how does the system—e.g. CPU, memory—perform under certain loads), stress testing (how much load can the system handle before a failure), and scalability testing (how does system performance change under various / increasing loads). The importance of these test criteria becomes immediately clear. Though a system might operate according to the functional requirements, these performance requirements can result in anything from low bandwidth to bottlenecks to system failure, thereby nullifying the functional requirements.

The core activities in performance testing create a multi-faceted process. A test environment must be chosen that appropriately models real-life expected conditions. If load or stress is tested only up to 70 per cent of what will be the actual system’s usage, then these tests will not yield accurate and helpful results. The acceptance criteria must then be defined. This might include e.g. what is an acceptable response time, what is the minimum throughput allowed, how much of the resources should be allowed to be utilized at any given time, or what should be the maximum wait time for processing requests. Based on these criteria the tests should be planned and designed and the environment should be set up to match the requirements. The test plan must now be recorded and the tests can be run. The final step is to analyze and report the results to the test manager or supervisor in charge. This process is depicted in Figure 4. [12]

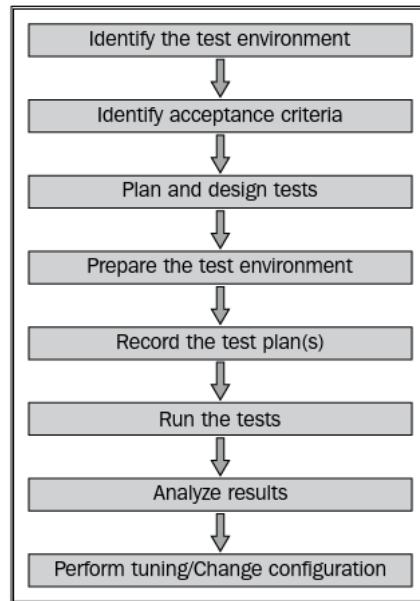


Figure 4: Performance Testing Core Activities [12]

This can easily become a large process demanding much time and resources. These result in the high cost so often associated with testing which is the problem that MBT tries to solve. Any part of this multi-stage process that can be automated and sped up can result in significant time and cost savings.

The discussion related to Figure 4 above referred to the amount of load or stress placed on a system during testing. The concept of load is key in performance testing, as the goal is to discover how the system performs with not only 1 active user, but more importantly with multiple concurrent users. Supporting concurrent users relates to the scalability of the system. In performance testing, the function controlling the target number of concurrent users to test against the SUT is known the *ramp* function. Sometimes called ramp-up rate, this function controls the rate at which the total number of concurrent users is increased. One common way of defining this ramp function is by an array of pair values referred to as milestones [7]. For example, a milestone pair could be listed as (120, 30) which states that at 120 seconds into the test session the total concurrent user count should reach 30 users. An expanded example of a ramp function could look as follows:

```
ramp_list = [ (0,0), (120,30), (300, 100), (450, 200), (600, 600), (750, 40) ]
```

2.1.5 Model-Based Testing

Model-based testing, although still comparatively new, has become a bit of a buzzword as it has gained traction in recent years. At least 4 main approaches have emerged for how to implement MBT.

1. Generation of test input data from a domain model
2. Generation of test cases from an environment model
3. Generation of test cases from a behavior model
4. Generation of test scripts from abstract tests

A thorough discussion of the differences and key characteristics of each approach can be found in [8]. The point of interest here is that this helps build a definition of MBT as “*the automation of the design of black-box tests.*” The similarity is that in black-box testing, without having full knowledge of the inner workings of the system, tests must be designed that check for fulfilled specifications of the *behavior* of the SUT. The difference between MBT and black-box testing lies in that rather than manually writing test cases based on documentation, MBT test engineers create *models* of the expected behavior of the SUT. Then tests can be *generated* automatically from those models.

Based on the definition and comparison to black-box testing, the impact of MBT on the test creation or recording process presented in Figure 4 (step 5) starts to become clear. The planning and design phase stays relatively the same, but the test creation time is greatly reduced due to automation. In addition to test design MBT, and specifically the MBPeT tool, assist and automate the environment setup, test recording, test execution, and even analysis stages.

One critique to this might be that traditional testing processes do offer some automation to speed up the test process, while sticking to longer established practices. In addition to a completely manual testing process, other *classic* testing processes include capture/replay, script-based testing, and keyword-driven automated testing [8]. Although these techniques do offer a certain amount of automation and automatic test generation, they still require arguably significant more manual effort. In capture/replay testing for example, time and cost is reduced by capturing an initial test session against the SUT, which in turn allows those interactions to be automatically replayed later in a subsequent test run. The drawback with this approach is that the initial test case must still be manually designed and implemented. Additionally, capture/replay lacks a level of abstraction and also is restrictively sensitive to changes in the SUT which both result in possible frequent breaks when trying to replay old tests. MBT solves this by its greater abstraction due to the nature of using behavioural models at a broader scope of the system.

2.1.6 Model-Based Performance Testing

In creating the MBPeT tool, its authors are combining the two disciplines of performance and model-based testing into one new category. This new discipline is being referred to as Model-Based Performance Testing. In this approach, the benefits from MBT of models which automate test generation is combined with the goals of performance testing in analysing performance metrics like throughput, response times, and other KPIs.

In model-based testing, abstract behavioural models server to generate test cases automatically. This saves significant time over manual test creation. Model-based performance testing and the MBPeT tool build on this principle. PTA (Probabilistic Timed Automata) models play a significant role in the MBPeT tool. Probabilistic timed automata can be defined as a set of *locations* with connected *transitions* which are selected based on probabilities (for a further discussion on PTAs see Section 3.3 Probabilistic Timed Automata Models). In addition to generating a test from an abstract model, a variety of test cases can be generated still based off of that one abstract model, simply by using different selection criteria [8]. An example of this at work is that the probabilistic timed automata allow different selections to be made at each/or many of the different location nodes depending on the probabilities linked to those actions. This demonstrates how a single model can result in different paths being taken and therefore different selection criteria being used.

In MBT, models lead to tests and test suites. Model-based performance testing extends this to define that models are used for load generation for performance testing purposed [6]. This is accomplished by the PTA models. These models represent user behavior against the SUT and are used by the MBPeT tool to generate synthetic workload against the system under test [7]. During a test session, the MBPeT tool follows a ramp function included in the test configurations to generate a precise number of virtual users against the system under test at any given moment. Based on this operation procedure, it can be stated that model-based performance testing, and MBPeT, utilize models leading to load generation.

Another MBT principle which is utilized in model-based performance testing is related to the test generation process. In particular, the relative timing between the generation of a test case and the actual execution of a test leads to a distinction between on-line and off-line testing. The prior has the advantage of reacting in real-time with the actual outputs of the SUT, therefore allowing the test generator to directly see which paths are selected. The latter, however, has a clear separation between test generation and execution. In particular, the tests must be generated before they are run, in contrast to combining the steps. With an offline approach, test cases are actually run on the model itself [13]. In order

to run a test session against the SUT, an adapter is required in order to apply the recorded tests to the SUT interface. Because MBPeT operates from an offline approach, this motivates the needs for the tool to have as inputs not only the models and test configurations, but also an adapter as well.

Finally, just as a set of activities specific to performance testing were defined in Section 2.1.4 Performance Testing, it is now possible to redefine the paradigm specific to MBPeT. These stages can now be broken down to the following 5 steps [8] and are displayed in Figure 5.

1. Model the SUT and/or its environment
2. Generate abstract tests from the model
3. Concretize the abstract tests to make them executable
4. Execute the tests in the SUT
5. Analyze the results

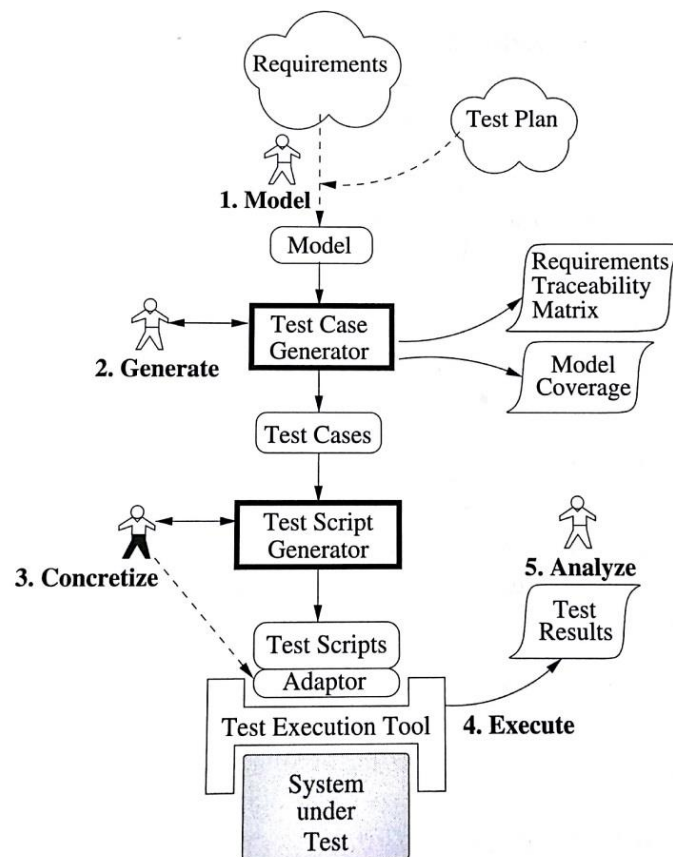


Figure 5: MBT Test Process (testing tools shown in bold boxes) [8]

2.2 Cloud & cloud services

The term “the cloud” is a widely used term by now. However, within the grand scope of the history of computing, *cloud computing* is still a relatively new term. Even outside the IT industry “the cloud” is often spoken of but perhaps vaguely defined as being “up there in the sky somewhere”. One definition of cloud computing states [14]:

“Cloud computing is on-demand access to virtualized IT resources that are housed outside of your own data center, shared by others, simple to use, paid for via subscription, and accessed over the Web.”

Though this definition provides a general sense, it misses a few realities and limits the possibilities. For example, cloud environments might be privately deployed rather than open over the public internet. Hence a more fundamental definition is helpful.

A more official definition is given by NIST (National Institute of Standards and Technology) to be [15]:

“A model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This cloud model promotes availability and is composed of five essential characteristics, three delivery models, and four deployment models.”

Cloud computing in itself is a large subject and an in depth analysis is not necessary here. The primary aspect that is relevant is the emphasis that cloud computing offers *hosted services* over a network connection. These services may vary from hardware, platforms, or services [16]. Services are offered to perform various task and these services are not deployed on local machines as in traditional IT environments, rather are accessible via the internet or private LAN (local area network).

The services offered over the cloud span wide ranges of free and paid services such as web-based email services (e.g. Google’s Gmail®, Microsoft’s Hotmail®), search engines (e.g. Google, Microsoft Bing™), social networking sites (e.g. Facebook, Twitter, YouTube, LinkedIn), business management services, building and hosting applications, storage space services (e.g. Amazon Web Services-AWS) [15].

2.2.1 Advantages of Cloud Solutions

The essential characteristics and advantages of cloud computing include [15]:

Access to resources

Perhaps the greatest advantage is the access of powerful processing given to multiple remote machines. Single, or clustered, cloud machines might offer greater computational ability, faster speed, and larger storage capacity than would be possible to make available for numerous personal computers. Because a cloud provider specializes in providing resources, they often can provide a company with more options than might be possible for a company to provide solely for itself at its own cost.

Mobility

Almost world-wide access to resources are made possible with web-based services. This would not be possible in most local deployment scenarios.

Scalability

Cloud services are typically structured in a “pay-as-you-use” or a monthly subscription model. This allows great flexibility with the ability to easily, and temporarily, increase or decrease the features and functionality. This might be storage space changes, memory or CPU allocation, etc.

Data security

The fact that data centers specialise in cloud computing provides them with advantages that companies have a harder time in replicating. This was seen in hardware and resource allocation and scaling. Security is another area where they must specialise as it is a key component to the service they offer. Because a data center will already have maximum security concerns built in to their plan, it is often cheaper and easier to implement that it would be for every company to handle it themselves.

Maintenance and support

Cloud suppliers usually handle maintenance and upgrades of their own systems. This means that updates, upgrades, and backups are included in the subscription fee and do not require special customer input and assistance.

Cost savings (efficiency)

Due to the subscription and “pay as you use” model, the cost savings of using a cloud computing provider are often substantial. Rather than providing all necessary aspects in-house, cloud computing centers offer all of the hardware, network resources, support, maintenance, security, and IT infrastructure for the same flat rate. All of these individual costs would otherwise have to be maintained by the company itself and would be a very large investment.

Environmentally friendly

It is a debated topic whether or not large data centers are more environmentally friendly than a much larger number of smaller in-house solutions. It is argued that data centers are better suited because they can run their servers at about 80 per cent capacity and handle all of their clients. Meanwhile every individual solution would run servers at a much lower percentage of as low as 5. However, there is only a slight difference in energy consumption of a system running at 80 as opposed to 5 per cent capacity. So the system in higher use is more productive. Also less overall systems would be running.

Data centers also have the advantage of being able to design and run the entire center in a more environmentally friendly way than individual in-house solutions can.

2.2.2 Disadvantages of Cloud Solutions

Internet connection reliability

The necessity for round the clock access to the internet can become a problem in the case of loss of connection or slow connection areas. The more business-critical these services are, the more of a negative impact this can have. However, as internet connectivity continues to improve in speed and coverage, this possibility becomes less likely.

Dependence on the supplier

Anytime relying on an outside supplier is a factor in a business there is risk involved. This work previously stated the supplier's many contributions as advantages to cloud computing over in-house solutions. This can of course become a negative factor if the supplier fails to fulfil their promises. Proper risk assessment should be evaluated with a cloud computing provider just as would be done of any third party supplier.

Data protection and security concerns

Security and protection concerns fall under the same scenario as supplier dependence. It was previously stated that cloud centers can often offer greater and reliable security than in-house solutions can. Again, this could be the opposite if the supplier has poor security practices in place. Anytime data is entrusted to outside parties, utmost concern and assessment must be used to ensure its reliability.

2.2.3 Cloud Service Models

The emphasis thus far has been on cloud computing as offering services. The architecture of cloud computing is likewise often divided into several layers, or models, that correspond to different types of services offered. These layers move from the ground up and from a sort of “technology stack” [14]. These three major layers are the IaaS (*Infrastructure as a Service*), PaaS (*Platform as a Service*), and SaaS (*Software as a Service*) layers. Each layer builds progressively on top of the previous as the focus of each is aimed at a specific customer segment. The variations focus on what services are being offered, from the underlying infrastructure alone towards more developed solutions and eventually software systems. These architectural layers can be visualized in Figure 6.

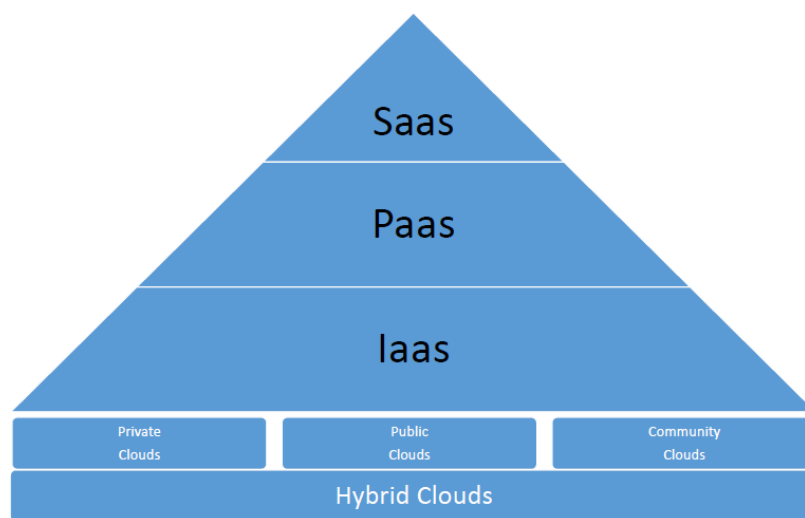


Figure 6: Cloud Architecture of Service Model Layers [14]

Infrastructure as a Service (IaaS)

The ground layer of this architecture is functionally exactly that. It is the foundational infrastructure offered to users focusing on providing resources. These resources include storage, network resources, and computational power [15]. Users typically have control over the system to install whatever OS, applications, and services of their choice on the cloud machine [4]. As the name implies then, the infrastructure is provided with which users can setup whatever environment they need.

Platform as a Service (PaaS)

While the IaaS offers the underlying resources, the PaaS layer provides a platform where users can create, deploy, and run applications using a *specific set* of programming languages, tools, libraries, frameworks supported by the cloud provider [4]. While this environment might be able to be built by the user on an IaaS layer, here in PaaS the cloud supplier builds the system, applications, tools,

libraries, languages etc. as a package that is ready for use by the user. This implies an extra layer of service offered to the user (less workload for the user) at the cost of less control by the user over the OS, libraries, and applications.

Software as a Service (SaaS)

The top layer is the most restricted layer and refers to providing cloud software applications to customers [15]. At this layer, the cloud supplier offers the most developed solutions of ready applications to the user. However, the user has the least amount of control over the system, but can only utilize the specific existing applications provided. This service is the most finely tuned to specific needs and is the quickest solution, whereas IaaS offers the most flexibility for the user to build a system to do anything according to their needs.

Where They Live

A cloud environment as was just explained can be deployed in a few different ways. Some organizations might have specific needs for strategic, operational, or security reasons which require a more tightly restricted cloud setting. This is referred to as a *private cloud*. A private cloud can be defined as “a fully functional cloud that is owned, operated, and presumably restricted to a particular organization” [14]. Many clouds, including most of the first, were *public clouds*. These are openly available to the public and often, though not always, focus on application level services. Examples of this would be many of Google’s services like search or mail as well as Amazon’s online marketplace. The third type of cloud is a *community cloud* which is essentially a reorganized public cloud by competing/cooperating businesses in a specific market, such as financial services. This is defined by [14] as a sort of shopping mall for cloud services co-located to help all parties achieve a critical mass for customers interested in that niche. Finally, a *hybrid cloud* is simply a combination of some or all of the previous mentioned environments. The primary benefit of this over a single cloud type is robustness [14].

Testing as a Service (TaaS)

Another type of service that fits into the realm of cloud computing is *Testing as a Service* (TaaS). TaaS or Software Testing as a Service is an outsourcing model where testing responsibilities are outsourced to a third party rather than done completely in-house. As such TaaS is sometimes referred to as “on-demand” testing. TaaS shares many similarities with SaaS, as it often functions as third party testing application and support services. Testing models that fit TaaS include automated regression testing, performance testing, security testing, and monitoring/testing of cloud-based applications.

Oracle Testing as a Service (TaaS) is a prime example of one such cloud-based platform providing testing services [17]. The features of this solution include

automated application testing, test script *execution* against the SUT, application *monitoring* and diagnostic data analysis. They claim that this TaaS solution significantly reduces testing time and cost, but without a loss of quality. Many of these criteria share great similarities to the MBPeT tool this work described. MBPeT seeks to automate the test process by its use of PTA models to generate virtual users. These models are then executed as tests against the SUT while live system monitoring, as well as final test reports, offer data analysis for the test designer.

Therefore, MBPeT has at least two key features that make it suit well within the scope of cloud computing. One is the previous description of TaaS as part of the cloud computing, comparing to Oracle's TaaS platform. Additionally, the distributed architecture of MBPeT's master and slave nodes (see Section 3.1 Tool Architecture) make deployment of it well-suited to a cloud deployment environment.

2.3 Rich Internet Applications

As web based services grew from static web pages to integrate more desktop like interactivity, dynamic web applications arose. Traditionally these were HTML documents rendered by the server upon user requests and interaction. The server did all the heavy lifting (page compilation) and the role of the client was to merely intercept user input and send and receive server requests/responses [18]. As the demand for greater complexity and functionalities increased, growing nearer to that of traditional desktop applications, the "classic" (HTML-based) web applications reached their limits with regard to these higher demands [19]. This is particularly true with higher levels of interaction, computation, and multimedia support. [20]

The advancement from traditional web architectures to Rich Internet Applications (RIAs) offers a solution to these limitations. An RIA accomplishes this in part by combining to strength of the Web distribution model with the strength and power of desktop application-like interactivity [20]. In contrast to the heavy server load in traditional web applications, in an RIA the client is assigned part of the data and computation workload. Provided new data does not need to be exchanged, this allows the end user to perform various and even complex interaction tasks without the necessity of a server round-trip which takes more time and load on the server. However, in the event that a server round-trip is necessary, the server still does not have to completely regenerate the entire page along with all its data and multimedia content. Rather, the client can selectively retrieve only that specific data that needs to be changed, then

update its own internal status, and finally redisplay the modified page via the web browser. [19]

With the solutions being addressed and the emphasis on greater interactivity, usability, and data communication, [21] defines RIAs as follows:

“[RIAs] are a variant of Web-based systems providing sophisticated interfaces for representing complex processes and data, minimizing client-server data transfers and moving the interaction and presentation layers from the server to the client. Typically, a RIA is loaded by the client along with some initial data; then, it manages data rendering and event processing, communicating with the server when the user requires further information or must submit data.”

In addition to the client-server infrastructure, another typical architectural element that desktop design brought over to web applications is the concept of application tiers. This three tiered structured divides an application into three segments known as *Presentation* (displays the user interface allowing user interaction on the front-end), *Business Logic* (controls the application logic and functionality), and *Data* (stores and retrieves data in the back-end) [22]. This structure is most commonly known as the *Model-View-Controller* (MVC) model and is commonly used in most WebApp frameworks today.

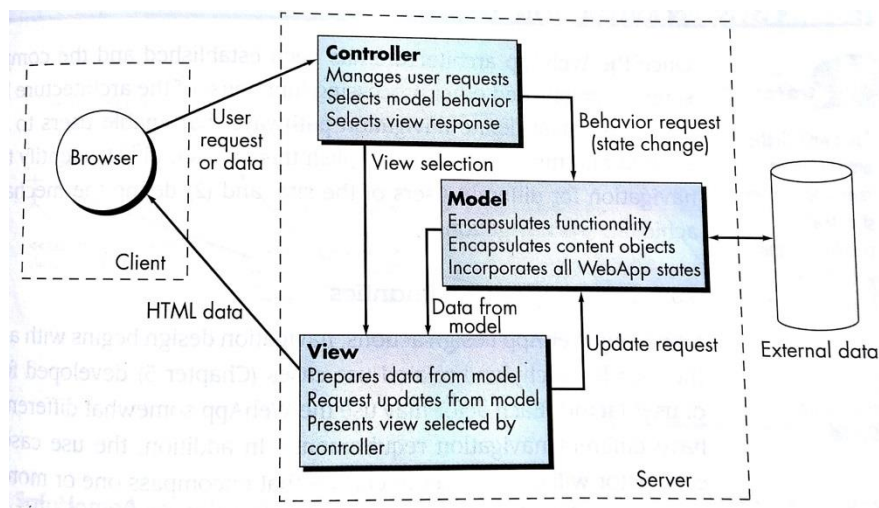


Figure 7: The MVC Model in a WebApp Implementation [9]

2.4 Vaadin framework

Traditional WebApp development often requires multiple technologies and languages, called “full-stack”, in order to achieve the end product. This includes front-end, or client-side, technologies for presentation and interactivity like HTML, CSS, JavaScript, and DOM (Document Object Model), as well as back-

end, or server-side, technologies such as SQL, Java EE, Servlet API, and JSP (JavaServer Pages). This is where frameworks like Vaadin come into play.

Vaadin Framework, henceforth referred to as Vaadin, is “*a Java web application development framework that is designed to make creation and maintenance of high quality web-based user interfaces easy*” [23]. Vaadin actually supports two different programming models, both a server-side and a client-side, but this focus will be on the more powerful one, the *server-side model*. This server-side framework handles the UI in the client browser as well as the AJAX (Asynchronous JavaScript and XML) communications between the client and server. This allows the developer to focus on the actual business logic of the application rather than spending enormous amounts of time on all the intricate details of a web application. The developer is not required to directly handle AJAX, HTML, CSS, or JavaScript; rather the entire system can be developed in pure Java. This presents the main goal of Vaadin, which besides being a UI framework for building beautiful looking user interfaces, the driving force is to increase *productivity*, i.e. speed and efficiency of software development processes. [23]

The server-side architecture model consists of a server-side framework and a client-side engine which is depicted in Figure 8 nedan. The client-side engine runs in the user’s web browser and handles rendering the UI to the client (Built-in, Add-on, and Custom Widgets) and delivering communication back to the server. The server-side framework is made of a *server-side API*, *client-side API* for interacting with the client engine, *UI components and widgets*, *themes*, and a *data model*. Although no custom theming and CSS is required, it is still a supported feature to customize the application look-and-feel through the use of theme resources. In this server-side model, the UI components are rendered in the aforementioned client engine, however the components have a server-side implementation as well which assists in data binding. The data model allows for binding the server-side UI components directly to the underlying data through the use of the business logic, Java beans, and Persistence framework (e.g. Hibernate or EclipseLink).

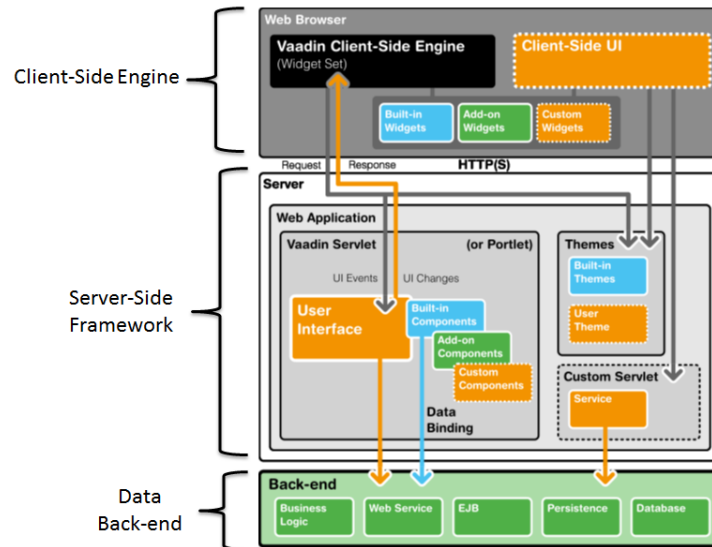


Figure 8: Vaadin Framework and Runtime Architecture [23]

A number of advantages that Vaadin brings to developers are confirmed by [22] to be:

- No need for “full stack”, as the coding is solely in Java. The only thing to know beside Java is Vaadin's own API. This means:
 - The UI code is fully object-oriented
 - There's no spaghetti JavaScript to maintain
 - It is executed on the server side
- The IDE's full power is in our hands with refactoring and code completion.
- No plugin to install on the client's browser, ensuring all users that browse our application will be able to use it as-is.
- Vaadin uses GWT (Google Web Toolkit) under the hood, so it supports all browsers that the version of GWT also supports, i.e. GWT handles the differences.
- Vaadin conforms to standards such as HTML and CSS, making the technology future proof. For example, many applications created with Vaadin run seamlessly on mobile devices although they were not initially designed to do so.
- Vaadin Framework is currently distributed under the Apache License version 2.0, which makes it free to use as an Open Source framework.

Chapter 3

3. OVERVIEW OF THE MBPeT TOOL

This chapter will investigate two aspects of the MBPeT tool itself. This deeper understanding of how the tool is designed, and how its use is intended, will provide background insight into design decisions of the MBPeT Dashboard and what the logic is behind the flow of operation.

3.1 Tool Architecture

The MBPeT tool's architecture will be explored in this section. Only a summary is needed, rather than a low level detailed look, as the official technical document for MBPeT has its own chapter on architectural design [7].

Section 1.2 Project Objective described MBPeT as a standalone application, downloadable to any local Windows or Linux (currently Ubuntu or Fedora) machine. The reality is slightly more complex, as it does not have a centralized architecture but rather was built with a distributed architecture. In a centralized architecture, the system may be built on *Object-Oriented* (OO) principles or not, but regardless the system itself is deployed as a single coherent unit. A distributed architecture however *decomposes* the system into unique parts of lower complexity that cooperate together typically via a communication channel functioning towards a single united goal [11]. MBPeT is separated into two unique pieces referred to as *nodes*: a master node and slave node(s). This chapter will describe both nodes in detail below. This distributed architecture of the entire MBPeT tool, which is designed to fit a cloud environment, is depicted in Figure 9.

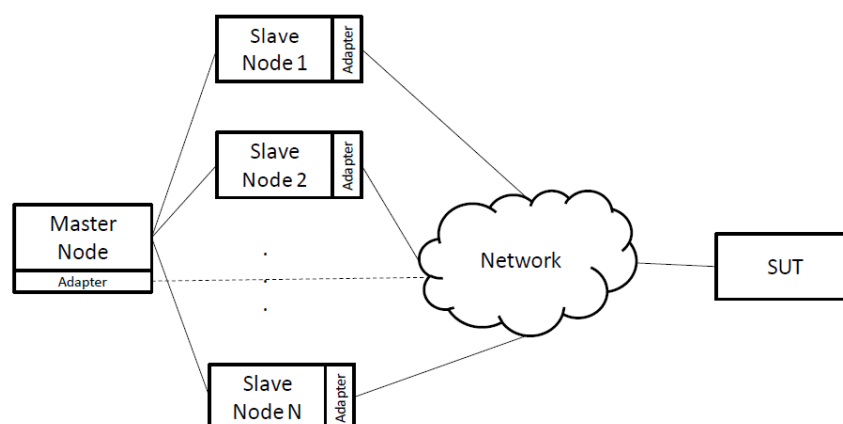


Figure 9: Distributed Architecture of MBPeT tool [7]

MBPeT was built using the Python [24] programming language. Python is a popular programming language used for web programming, GUI development, software development, and system administration to name a few. Additionally, a few third party libraries were used for needed functionality such as system monitoring, graph plotting, DOT model processing, and SSH (secure shell) communication. [7]

3.1.1 Master Node

The master node itself is divided into four separate modules. These modules are as follows:

- Core
- Model Validator
- Slave Controller
- Test Report Creator

The master is the only node end users interact with directly. Therefore, just as the master is a sort of control center for the whole test session, the *Core* module is the control center for all master processes. The core handles initiation, communication, and process flow between all other modules. The second module is the *Model Validator*. It is responsible for analysing the VU profiles (models) that were given as input to check them for syntactic errors. Because these models are the basis for this model-based testing procedure, it is critical to have well-formed models that follow the syntax and validation rules of the system, otherwise the test could yield inaccurate results. The *Slave Controller* is the specific module of the master which actually handles the interfacing with the slave nodes which was previously noted. The final module, the *Test Report Creator*, is charged with, as the name implies, generating the final test report after all slave nodes have completed the test cycle. [7]

The benefits of this separation of tasks are greater scalability and parallelism. Scalability is achieved by keeping the master process lightweight, while parallelism is achieved as the separate modules can each run on different processor cores [7]. This combination allows for greater performance and scalability of the entire MBPeT tool. Figure 10 nedan shows a diagram illustration of the master node.

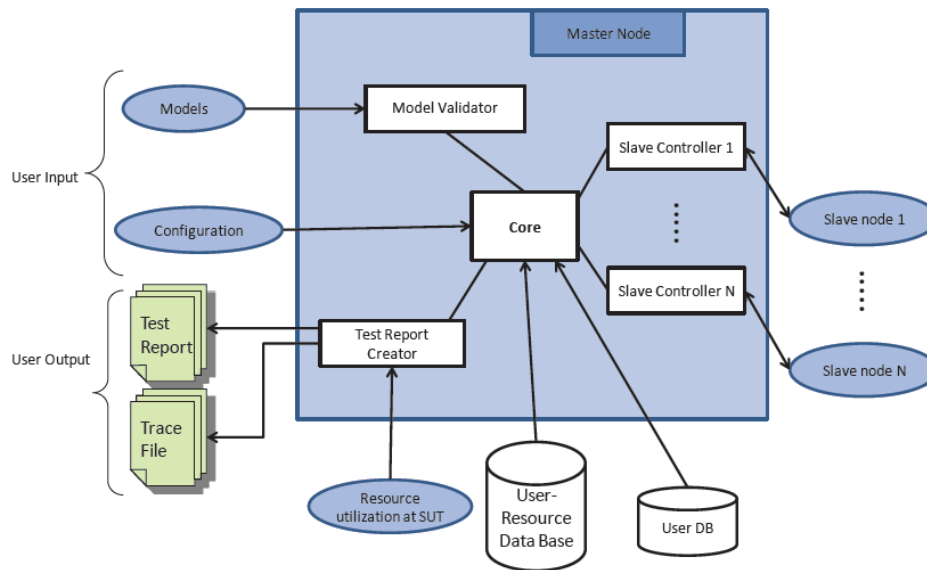


Figure 10: Master Node [7]

3.1.2 Slave Node

Similar to the Master node, the slaves have a module-based architecture for the division of tasks. These modules are:

- Load Initiator
- Model Parser
- Load Generator
- Resource Monitor
- Reporter

Just as the master node is responsible to initialize the test at the test suite level (over the entire system), each unique slave entity has a *Load Initiator* that prepares all the test configurations and files it received from the master core in preparation for running the test. At this point, the master has already validated the models, but the slave's *Model Parser* parses the model into a local structure for use in the test case. The *Load Generator* module uses the specifications from the settings file (test configuration) to generator the virtual users and create the actual load for the test. The test duration, number of users, and user-think-time (time in between user actions) are some of the main test parameters that the Load Generator utilizes at this step. [7]

When running MBPeT, a user can decide how many slaves to run during the test. This is in essence setting the maximum number of slaves that are available for the test session. At the beginning of test, all selected slaves are initialized, but only the first begins load generation. The slave's *Resource Monitor* module is a separate thread in the slave that periodically checks the system conditions. If

the test parameters call for an increase in virtual users but the slave has become saturated, it keeps a constant user count and informs the master that it is saturated. From there the next slave can begin load generation to continue the test session according to the settings provided. Finally, a *Reporter* module sends report files back to the master node when the test has completed. The master node uses these report files to compile the single and final test report document [7]. See Figure 11 nedan for a diagram of the architecture of a slave node.

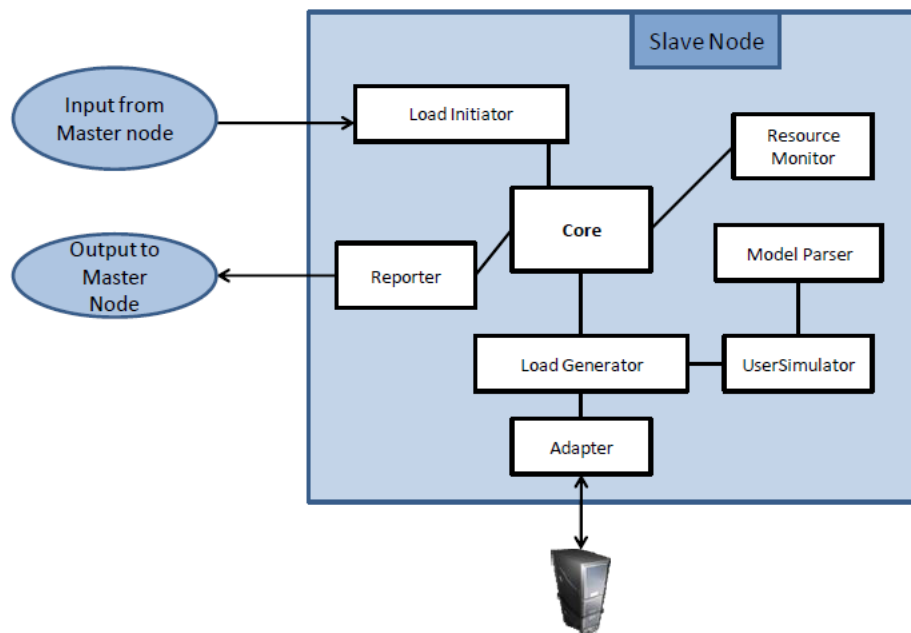


Figure 11: Slave Node [7]

3.2 Tool Operational Process

Now that the architecture of the system and its two primary components (master and slave) are clear, a brief overview will be given of both 1) how user interact with and use MBPeT, and 2) the actual load generation and test execution procedure.

End users communicate only ever with the master node, never a slave directly. The master node is responsible then to receive all relevant test parameters and then act as the centralized control center for the duration of the test session. Upon validating the input parameters, the master communicates with and controls all remote slave nodes. The master initializes each slave by providing it with the relevant information for the SUT as well as the test configurations it received from the user. The master then hands over the work load to the slave, meanwhile becoming responsible to receive live system monitoring info about the test. After test completion, the master generates a final test report for the whole test session.

3.2.1 End-User Interaction

The usage of the MBPeT tool can be viewed from two perspectives. First, it can be viewed from the actual interaction of the end user with the system. In terms of user interaction, the process of running MBPeT is quite short and non-complex. The process could be broken down into three steps:

1. Prepare Configuration Settings
2. Launch Master Node
3. Launch Slave Node(s)

The first step of preparing the files and settings for the test session is the most time consuming as this is where the end user not only makes decisions on test settings (e.g. test duration, total VU count, mean user-think-time, ramp value, etc.) but more importantly must create the user profiles that serve as the models for the test session. The *settings* are contained in a Python files names *settings.py*, and contain all the test parameters. The *models directory* contains the model files for each unique user profile. And the *test adapter (adapter.py)* defines the actions the user profiles carry out.

Section 3.3 Probabilistic Timed Automata Models gives more detailed information on the models themselves including how they are built and on what basis these dynamic users function and make decisions.

The original MBPeT system is primarily a command-line based application. This makes the latter two user interactions very brief and simple. There are various arguments that allow specific settings to be enabled. These arguments can be found with the help of the OS's native help argument in the command-line interface. First, the master process is started with a command as follows (all commands are shown in Linux format):

```
./mbpet_cli test_project
```

To specify the number of slaves that should be targeted, the desired number would be added as the first argument as follows:

```
./mbpet_cli test_project 4
```

By default the master process initializes slave nodes and then waits for user confirmation before beginning the test sequence. The following `-s` command ignores this requirement and starts load generation automatically, while the `-b` and IP (internet protocol) address point the master to a listening UDP (user datagram protocol) client for extra monitoring information to be displayed.

```
./mbpet_cli test_project 4 -b localhost:9999 -s
```

Finally, the TCP (transmission control protocol) port can be specified with the `-p` argument. This argument overrides the default TCP port, of 6000, which the master and slave nodes utilize for communication procedures during the extent of the test session.

```
./mbpet_cli test_project 4 -p 6001 -b localhost:9999 -s
```

A slave node is started with a very simple command with one argument, the IP address of the master node. For example:

```
./mbpet_slave 127.0.0.1
```

Additional optional arguments can be given to the slave. One particular such argument that was crucial during the Dashboard implementation phase was the TCP port assignment argument to match that of the running master.

```
./mbpet_slave 127.0.0.1 -p 6001
```

Beyond these execution commands, there is no further input required from the end user. The master node and the optional UDP client display live updates at a one second interval. The master also displays any error messages and finally completion messages after successful test session. In summary, the end user only interacts with the master node, and the system in general, by initialization and then test retrieval after the entire session is complete.

3.2.2 Load Generation Process

Second, the system level perspective of the program itself it can be examined. Only a very brief explanation of this system level process is needed since the official documentation [7] describes the process in more detail. The entire session life of running MBPeT against a SUT can be summarized by three phases 1) Test Setup, 2) Load Generation, and 3) Test Reporting.

Test Setup

In respect to the MBPeT system, a test session begins when an end user launches the application, together with the test configurations. The Core module of the master node receives these input parameters (user models, settings.py, adapter.py, number of slaves) and then runs the aforementioned *Model Validator*. Now the master can focus on listening for slaves to connect with over

socket connections. The process of connecting and initializing slaves is handled by the *Slave Controller* module of the master node. When all slaves are initialized with the test configurations, the master can start the *Resource Monitor* for the SUT and proceed to step 2, *Load Generation*.

Load Generation

A simplified explanation of the *Load Generation* cycle could state that:

The load generation on an idle slave begins. So long as the slave is not saturated, it continues running the test while sending monitoring data back to the master node. If the slave becomes saturated, it fixes the virtual user level at the current state and notifies the master who in turn can start up the next slave to continue the process. This whole process continues until the test comes to completion and the final stage can begin.

Due to the architectural design examined in Section 3.1 Tool Architecture, a dedicated module within the slave and master handle all the various processes involved in this cycle. The master communicates with the slave via the *Slave Controller*. The slave can handle all the tasks during the load generation cycle because of the dedicated module to handle load generation, while another module handles resource monitoring etc.

Test Reporting

The final step begins when the test duration is reached and the *Reporter* module of the slave(s) send the final test data and results back to the master. The master *Slave Controller* then terminates all connected slaves and terminates the *Resource Monitor*. Finally, when all connections are closed and processes have terminated, the Test Report Creator can take over and generate the final test report. After this the master terminates itself and the test session is complete.

A detailed sequence diagram of this entire process can be found on page 16 of the official MBPeT technical report [7].

3.3 Probabilistic Timed Automata Models

Since the focus of this work is based off of model-based testing, and MBPeT is being presented as a Model Based Performance Testing tool, the final aspect of MBPeT worth describing in detail here are the virtual user models which are utilized. These models therefore, are where MBPeT distinguishes itself from other testing tools that might operate on other principles besides MBT.

It has been thus far mentioned that the models represent a set of virtual users and are used to generate a probabilistic, synthetic workload. To be more precise, abstract models serve as a set of Probabilistic Timed Automata (PTA) which model the VU profiles needed by MBPeT to run against the SUT [6]. PTAs are defined by [25] as “*a modelling formalism for systems that exhibit probabilistic, nondeterministic and real-time characteristics*”. PTAs are defined in terms of their properties as a set of *locations* with connected *transitions*. Further, a probabilistic transition consists of 1) a source location, 2) a time constraint or *invariant condition*, 3) a *probability*, and 4) an *action* [26]. This quadruple provides a PTA with its ability to make dynamic decisions based on probability rather than static, never changing actions which always result in the same outcome. The time constraint represents the time an end user might take to think before their next action. Because any single location can have multiple destinations, the probability represents the chance of one transition being chosen over another. The foundational principles and behavior of PTAs therefore make them highly suitable for dynamically representing real-life user behavior, and as such makes them suited to MBT purposes [4].

It is crucial to design the PTAs (with the combinations of actions, clock time, and probability) in such a way as to yield a generated load from model users as closely to that of real-world users. The motivation behind this being that if the VU models fail to properly depict actual users of a SUT, then the test results will be inaccurate towards actual system load and performance [27]. If PTA models cannot yield reliable conclusions and make accurate performance predictions, then the results will naturally be inconclusive and give false assurances of expected performance.

PTA fall under the category of formal methods and can be defined in terms of traditional mathematical definitions [25, 26]. Figure 12 displays a very simple PTA model diagram with a set of locations, shown as circles, connected by a set of edges. The label values along each edge represent the probability of that edge being selected, the wait clock time before firing that edge, and the action to be executed upon firing the event. The circles (locations) are commonly labelled in numerical ascension as in this diagram, and the double circle is a typical format used to denote the exit location. Figure 13 shows a larger example of a PTA model used by [7] in their case study of an auction site web application.

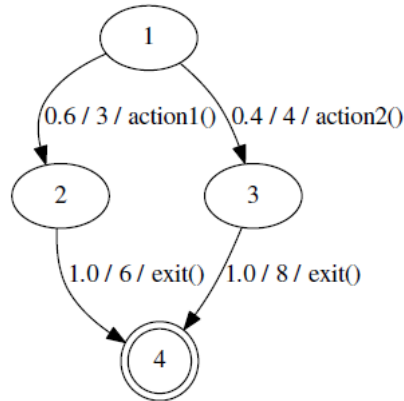


Figure 12: PTA model [4]

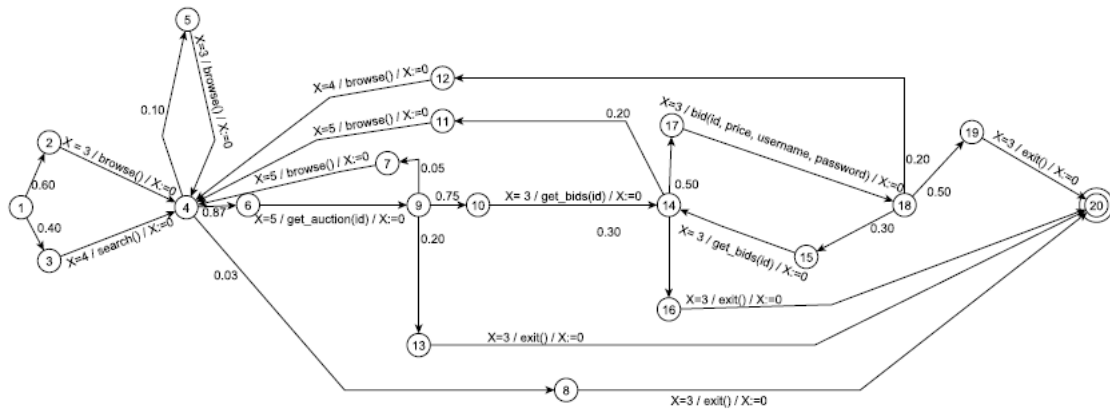


Figure 13: PTA model for YAAS application [4, 7]

PTAs can also be modelled in code format. In fact, it is often this code definition of a PTA that is used by a charting tool to generate a graph such as was presented in Figure 12 and Figure 13. The programming language known as the DOT Language is one such graph description language which is not limited to, but can be used for defining PTAs. The syntax for this language is fairly straightforward. A barebones requirement for defining a PTA model with DOT requires three things, 1) define the graph type and name, 2) define the states (nodes or locations), and 3) define the edge connections optionally with the probability, time, and action related to it [28]. The following code snippet shows an example DOT file of a PTA model:

```

digraph example_user_behavior {
  1
  2
  3
  4 [shape = "doublecircle"]

  1 -> 2 [label = "0.6 / 3 / action1()"];
  1 -> 3 [label = "0.4 / 4 / action2()"];
  2 -> 4 [label = "1.0 / 6 / exit()"];
  3 -> 4 [label = "1.0 / 8 / exit()"];
}

```

A number of third party programs and libraries exist that allow rendering DOT files into graphical depictions like Figure 12 and Figure 13. To name only a few popular such options, *Graphviz* [29] defined the DOT language and has a popular graph visualization software that supports numerous graph languages, *Canviz* [30] is a JavaScript xdot rendering library for drawing Graphviz graphs to a web browser canvas, and VizierFX [31] is a Flex (Fast Lexical Analyser) library for drawing network graphs.

MBPeT accepts a valid DOT file (*.gv* or *.dot*) as input for all PTA models affiliated with the SUT. It uses the pydot library as an interface between the DOT models and the Python code of the tool itself. Pydot is a “*full interface to create, handle, modify, and process graphs in Graphviz’s DOT language*” [32]. MBPeT uses pydot specifically in the aforementioned *Model Parser* and *Model Validator* modules to parse the abstract DOT models that will be used to generate the VUs [7].

Chapter 4

4. PROJECT REQUIREMENTS

4.1 Project Statement

As the previous chapter introduced, the underlying work of this thesis is a software development project. This work will result in a new software application by extending an existing solution with new features, functionalities, and connectivity. The software project can be defined as such:

A web application should be designed and implemented which will provide (1) a *graphical user interface* allowing all functionalities of the MBPeT tool (model and test parameter creation; test execution with real-time monitoring; and reporting) to be carried out in an intuitive, user-friendly manner, and (2) centralize the deployment of MBPeT's master component to a web server (in contrast to its previous local system-level deployment) where a scalable number of users can simultaneously utilize the system in parallel.

The importance and reach of this project has two potential levels. The immediate relevance affects the Software Engineering Lab at Abo Akademi where the previous research and development has taken place, and where the most immediate use of this software will occur. The goal as such is to simplify the tool's usage and make it more user-friendly. The team in the lab will profit the most in their own use of the MBPeT system.

The second potential level of influence this project can have is in the academic and industry level at large. As was briefly introduced in the previous chapter, model-based testing is newer and less widely used in its application area, despite the fact that academic circles have preached its many benefits and solutions to problems of cost and complexity. If MBT can be a viable solution for academic and professional circles alike, then more proof cases and better tooling can help to advance this. Therefore, time and cost saving for one, and tool support for second, is where the relevance and impact of this work can play a role. Adequate and promising tools that would result in real savings for companies can boost interest and usage of MBT testing in replacement of slower and more costly solutions.

A large number of both open-source and commercial performance testing tools do already exist. A complete list would be too long for the scope of this work, but the following are a few of the most popular tools. WebLOAD is a commercial load

and stress testing platform that validates systems through the use of virtual users, similarly in principle to MBT but still not strictly model-based. WebLOAD utilizes action recording to more quickly duplicate and replicate user profiles [33]. Apache's JMeter is a popular open-source that performs load/performance testing as well as functional testing aimed at Java applications. Though it is powerful and has a wide range of applications beyond its original target of web applications, test creation is manually handled [34]. LoadRunner by Hewlett Packard is a tool that allows test engineers to monitor not only performance but also system and infrastructure information relevant to performance criteria. LoadRunner emphasizes its ability to enable fast load testing to minimize time and cost [35]. A final tool by HP is httpperf that gives the ability to generate various workloads and measures web server performance. Its focus is not to measure only a single benchmark but to measure multiple benchmarks across micro and macro-level layers [36].

One obvious takeaway from even this short list is that companies clearly see the need to streamline the testing process. The route chosen to accomplish this varies. Some choose to stick with a certain testing framework, which might historically be manual and time consuming efforts, while innovating tools that speed up the process. The point of model-based testing is to change the testing paradigm at the root, rather than trying to automate manual-based techniques. MBPeT and this project aim to provide one powerful tool based on these proven theories.

4.2 Functional requirements

The functional requirements for the Dashboard application were defined at the outset of the project by the project owner. The requirements can be summarized under 4 categories. These categories include User accounts, Test configuration, Test Execution, and Reporting. The complete set of functional requirements is listed below:

1. User Accounts

- **FR1.** A user should be able to register and log in
- **FR2.** Multiple concurrent users should be supported

2. Test Configuration

- Model profiles
 - **FR3.** User creates (draws)/edits model on the webpage and saves
 - *Alternative* - user uploads a DOT file which is displayed graphically
- Ramp Function
 - **FR4.** User can edit the ramp function on the page

- *Alternative* - a predefined ramp function can be "textually" specified as a list of tuples, where the first number is the relative time of the experiment and the second is number of concurrent users. E.g. [(0,0), (10,30), (30,30)]
- Settings
 - **FR5.** User should be able to specify test session parameters:
 - Test duration
 - Mean user think time
 - IP address of the SUT
- Test adapter
 - **FR6.** Write Python or XML code on the webpage
 - *Alternative* - upload file

3. Execution

- **FR7.** Each master process runs independently for each user
- **FR8.** There can be several masters running in parallel on the same node (computer/server). At the moment we do not consider distributing masters to several computers.
- **FR9.** The GUI should show the dynamic ramp and based on user selection:
 - Response time
 - Error rate
 - Throughput (actions per second)
- **FR10.** KPIs (Response Time, Bandwidth, Throughput) updated live.
- **FR11.** Clicking/hovering over a curve in the graph should show coordinates
- **FR12.** Execution log must be shown, including:
 - Possible compilation errors (e.g., wrong parameters, problems in the adapter, invalid model format etc.)

4. Reporting

- **FR13.** Test report should be saved and available online after execution
- **FR14.** Test configuration should be saved for each experiment
- **FR15.** An old experiment can be rerun and a new test session is created
- **FR16.** The execution log should be saved and available online after execution

4.3 Non-Functional requirements

A number of non-functional requirements can be identified for the Dashboard application. These requirements can be grouped according to performance, security, and usability factors.

1. Performance

- **NFR1.** System resource (namely CPU and Memory) utilization should be devoid of bottlenecks, deadlocks, or memory leaks

- **NFR2.** System resource utilization should grow at a linear, expected rate (corresponding to load and user count)

2. Security

- **NFR3.** Application functionality should be limited to registered users
- **NFR4.** Authentication should be verified via login credentials

3. Usability a.k.a. look-and-feel

- **NFR4.** UI layout should be clean—according to customer approval
- **NFR5.** System use should be intuitive and easy to learn
- **NFR6.** Various elements of test configurations (models, settings, adapters) should be grouped together forming an intuitive workflow – proceeding towards test execution

Chapter 5

5. IMPLEMENTATION

This chapter serves as a documentation of the practical steps taken to realize the final solution of the MBPeT Dashboard. The purpose of this chapter is to not merely document the work completed and the engineering principles employed, but more importantly to introduce the contributions brought about by this engineering project. These various contributions are applicable and useful for the future development of MBPeT, for the Vaadin community at large, and for the individual expansion of the author's understanding and skill set towards future projects.

The tools, technologies, and environments used will be documented followed by an overview of the broad structure of the Dashboard. Various specific components, both conceptually speaking and specific classes) will be explained including the navigation process, add-on component development, connection and communication with the MBPeT tool, and finally various additional components which served a critical role in completing the implementation.

5.1 Approach

At the start of the development project, the task was given to create a Web Application that would provide a new UI as well as a single online access point from where to run the MBPeT tool. The scenario was described to be a sort of "black-box" access to the MBPeT master node (as described in Section 3.1 Tool Architecture). This description indicated that limited knowledge, or more accurately limited access, into the inner-workings of MBPeT would be given or required to accomplish this web interface. Figure 14 demonstrates the basic connection and reach of the Dashboard application to the existing MBPeT architecture.

Other requirements for new functionality that this Dashboard web application should offer included a drag-and-drop VU model builder, charts for both live system monitoring and drag-and-drop editing, and online code editing.

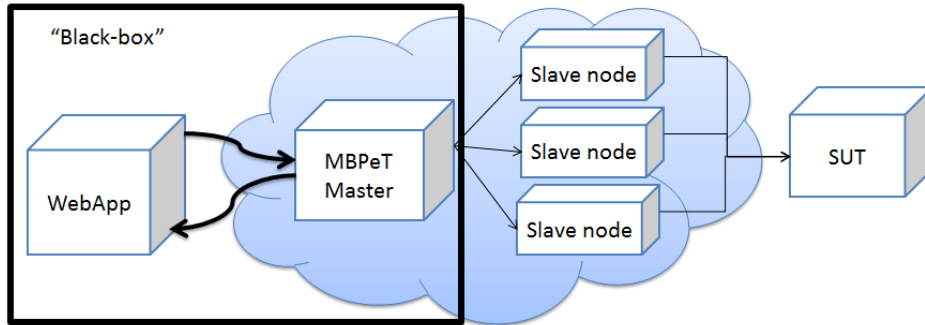


Figure 14: WebApp + MBPeT Connection

5.2 Development Environment

The development phases differed in the environment utilized in each. During development, a personal laptop running Windows 8.1 was chosen as that OS was the most familiar to the author, in contrast to Linux or Mac OS X. This 64-bit version ran on a 4th gen Intel Core i7 dual-core processor at 2.00 GHz with 8 GB of RAM and a hybrid SSHD (Solid State Hybrid Drive).

The popular Eclipse IDE for Java EE Developers was chosen as the development environment. In addition to manually downloading and installing the zip-file, Vaadin supports plugins for Eclipse, NetBeans, and IntelliJ for easy integration with their framework. At the time of this project Vaadin was at version 7.5.5. Dependency managers such as Maven and Apache Ivy are suggested, though optional, support tools for managing external jars and dependencies of any given project. Ivy was pre-installed and utilized in part for this project. However, due to the specific add-ons, tools, and libraries needed, many of these dependencies were manually included. Apache Tomcat is a lightweight, open source Java server that has long been the go-to for Vaadin projects. Tomcat easily integrates with Eclipse and serves the application to the browser. Tomcat 7 was utilized for both the development and deployment environment. Finally, WampServer (Windows, Apache, MySQL, PHP) provided the MySQL database backend for the Dashboard during development.

Running the MBPeT tool during development was done in two ways. As Section 3.1 Tool Architecture previously stated, the master node runs on Windows or Unix while the slave nodes only support Unix (currently Ubuntu or Fedora). Therefore, the master node was run locally via the command line on the Windows machine, whereas the slave node(s) ran inside a VirtualBox installation, from the same Windows machine, running Ubuntu 14.

5.3 Deployment Environment

The final deployment environment of the Dashboard application was hosted on a Linux web server owned by Åbo Akademi. The host machine features 8-cores of Intel i7 architecture CPU, 16 GB of RAM, a 7200 rpm hard drive and Fedora 16 as the host operating system. Due to the familiarity with Tomcat, it was used also for final deployment on the Linux server to host the Dashboard application. The MySQL database was brought over from the development machine and copied to the already running MySQL service running on the server, providing the backend persistence.

The same server machine that hosted the Dashboard application is also responsible to host and run the MBPeT system, both master and slave nodes. Having all systems run locally on the same machine simplified the “black-box” connectivity process between the Dashboard and the MBPeT system itself. Since MBPeT, in addition to its previous GUI, was a command line tool, having both systems share the same logical drive and directory structure allowed the web application to run both master and slave nodes via executing shell commands (processes) directly on the Linux system. Because this required little to no alterations to the existing MBPeT system, it served as a logical and efficient choice.

5.4 Development Process

5.4.1 Technologies Used

In addition to the above mentioned tools and platforms, numerous other technologies were implemented to realize the Dashboard application. Although one of the main marketing points of Vaadin is that the framework frees the developer from handling all the complicated processes of traditional web development and rather writing just plain Java code, this does not mean, however, that one cannot extend a Vaadin project in various ways. In addition to the recent feature of building apps with declarative UI's in HTML, developers can always access the underlying CSS files and customize any part of the application theme. Although the new Valo theme was chosen for this UI as it matched early design mockups, CSS was utilized to customize specific components and layouts.

Other significant libraries and technologies used include: JPA (Java Persistence API) to provide the object-relationship mapping for the database, JDBC (Java Database Connectivity) as the driver for MySQL Connector/J, Hibernate Validator for automatic validation of model fields (in forms etc.), Vaadin's

BeanValidation library for further POJO (Plain Old Java Object) validation, JSON (JavaScript Object Notation) as the data exchange syntax between master node and Dashboard during test monitoring, UDP and Datagram Sockets as the transmission protocol of the aforementioned data exchanges, and GitHub was chosen as the versioning repository. Figure 15 displays the lib folder of the project containing all external libraries and jars.

JavaScript was additionally required for implementing necessary add-ons, which are discussed in detail below in Section 5.4.2.3 Add-ons.

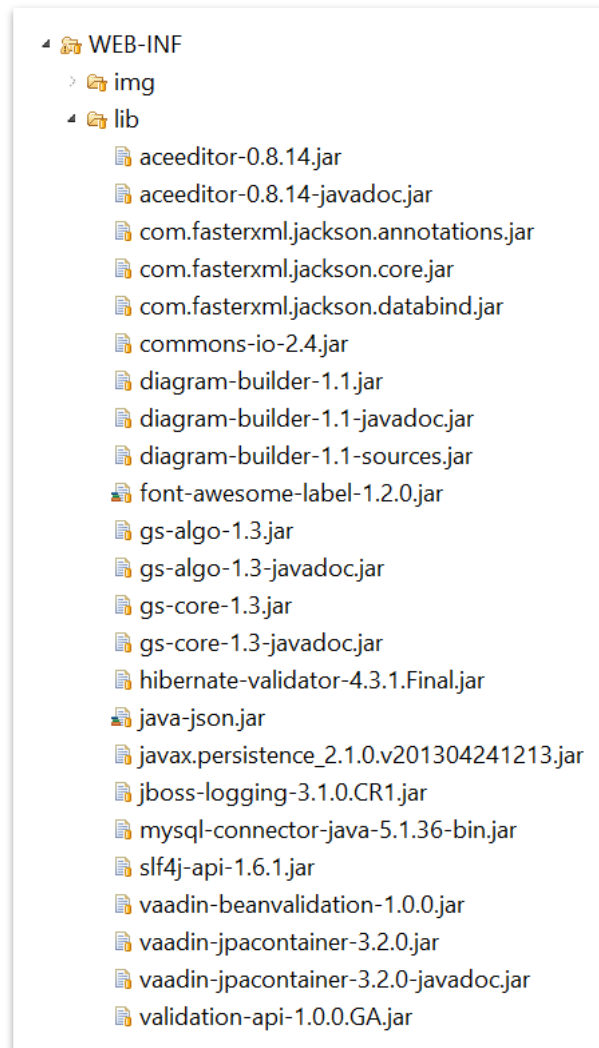


Figure 15: External libraries referenced by the Web-app project

5.4.2 Application Structure

Vaadin not only supports, but stresses following e.g. the well-known MVC architectural paradigm for structuring software projects. To be more precise, although Vaadin previously recommended MVC, it now advocates the similar MVP (*Model-View-Presenter*) pattern. This pattern essentially adds an interface

between the view and the controller, thereby better isolating the view from direct interaction with the models themselves. The purpose of this is to allow easier unit testing of the presenter and model components [23]. This project, however, chose to follow the more traditional MVC pattern due to better familiarity for efficiency sake.

The following sections will provide a brief explanation of some key components of the actual code structure.

5.4.2.1 Domain Models

The primary domain models needed for this application included *Users*, *SUT's*, and *TestSessions*. This already provides a simple understanding of the workflow of the application: one first creates a *user* account, when logged in to the system a user can create as many target *SUT's* (a target system for running test against) as desired, and for each target SUT, numerous individual *TestSessions* can be created and run. Below is a code snippet of the *TestSession* model (without constructors and getters/setters) showing the fields with the JPA and Hibernate annotations.

```
@Entity
public class TestSession {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int id;

    @NotNull
    @Size(min = 1, max = 40)
    private String title;

    @ManyToOne
    @JoinColumn(name = "parentsut", referencedColumnName = "ID")
    private SUT parentsut;

    @OneToMany(mappedBy = "parentsession")
    private List<Model> models;

    @OneToOne(mappedBy = "ownersession")
    private Parameters parameters;

    @OneToOne(mappedBy = "ownersession")
    private Adapter adapter;

    @OneToOne(mappedBy = "ownersession")
    private AdapterXML adapterxml;

    ...
}
```

A `TestSession` encapsulates three components needed by MBPeT, a test *Adapter* (Python or XML), test *parameters*, and the PTA *models* (not to be confused with the Java POJO model). Separate domain models represent each of these three components, as the code snippet above demonstrates.

The annotations for the fields were used by JPA to automatically create database tables for all models, thus automating and speeding up development tasks. Vaadin contains, as a free add-on rather than part of the core itself, a `JPAContainer` that makes it possible to easily bind UI components to the database via JPA.

5.4.2.2 Navigation and Views

Navigation in Vaadin application can be handled by various methods depending on the application structure in question. Assuming, as in this case, that at least two to three separate views (screens) exist in the application, the best way to handle navigation is the *Navigator* component built in to Vaadin. For this project it was sufficient with only two views which the Navigator switched between, the *LoginView* and the *MainView* of the dashboard (a *RegistrationView* could be a third, but in this case was handled in a popup window instead). The Navigator not only handles the actually navigation to the view requested, but also automatically sets the URI fragment in the browser address bar. This function is useful for development (passing information such as “id” from one view to another) as well as for the end user by clearly displaying where in the application they currently reside.

Further navigating between individual SUT’s and `TestSessions` are handled indirectly by the Navigator. Selecting an item from the left hand menu triggers an event navigating to the `MainView`. Whenever a `View` in Vaadin is navigated to it calls its mandatory `enter()` method. This method then scans the URI fragment to determine which SUT or `TestSession` “screen” is being requested, and proceeds to load that content to the main layout section to the right. In this way, once logged in, a user never navigates away from the `MainView` itself, but still employs the navigator’s functionality to switch content within that view (or screen). The code snippet below displays part of the `MainView`’s `enter()` method which provides the sub-content navigation. Figure 16 depicts the primary application navigation.

```

public void enter(ViewChangeEvent event) {
    ...
    if (event.getParameters().equals("landingPage")
        || event.getParameters() == null {
        contentLayout.setContent(new LandingPage(tree));
    } else if (event.getParameters().contains("sut=")) {
        // navigate to SUT "home page"
        contentLayout.setContent(new CaseViewer(
            event.getParameters(), tree));
    } else {
        // navigate to Session "page"
        contentLayout.setContent(new SessionViewer(
            event.getParameters(), tree));
    }
}
}

```

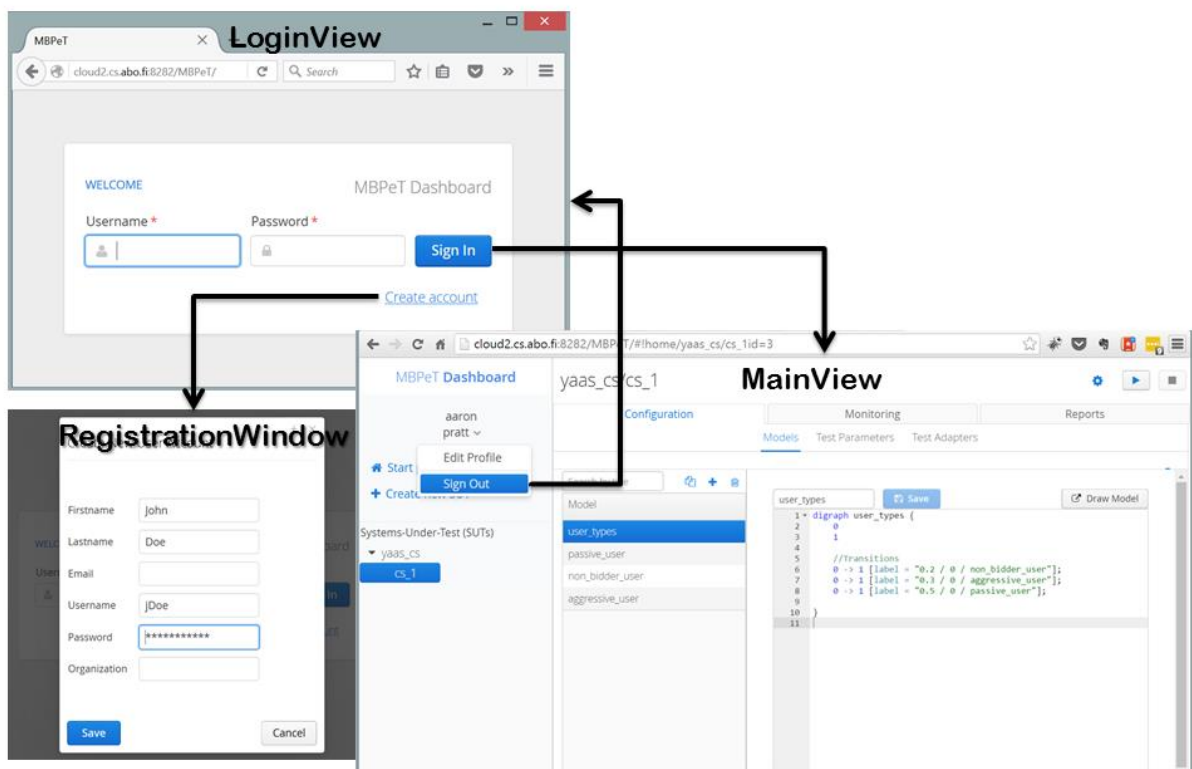


Figure 16: Navigation and Views (plus Registration Window)

5.4.2.3 Add-ons

Vaadin is primarily a UI framework and therefore focused first towards user interface components. Much of the benefit that Vaadin provides goes beyond the UI itself and into the fact that Vaadin handles all the client-server interaction and traditional web communications. This allows its users to quickly and efficiently develop their applications. Although the scope of UI components provided by Vaadin core is vast, there remains room to grow. Therefore not only is Vaadin itself constantly updating and adding new features to its core, but also the Vaadin Directory exists as a marketplace for add-on components. These

components can be developed by Vaadin officially or any third party developer who desires to contribute. Vaadin supports JavaScript integration by allowing developers to write their own JavaScript components, or take existing ones, and port them over into a Vaadin component. This allows the component to function as any other Java Object with constructors, getters/setters, methods, and even state information (if the component is stateful).

The UI specific requirements for the MBPeT Dashboard included several functionalities that do not exist “out-of-the-box” in Vaadin core. Therefore it was necessary to use an existing add-on from the directory, extend an existing add-on, and finally to create a new add-on from scratch. These three components are described below.

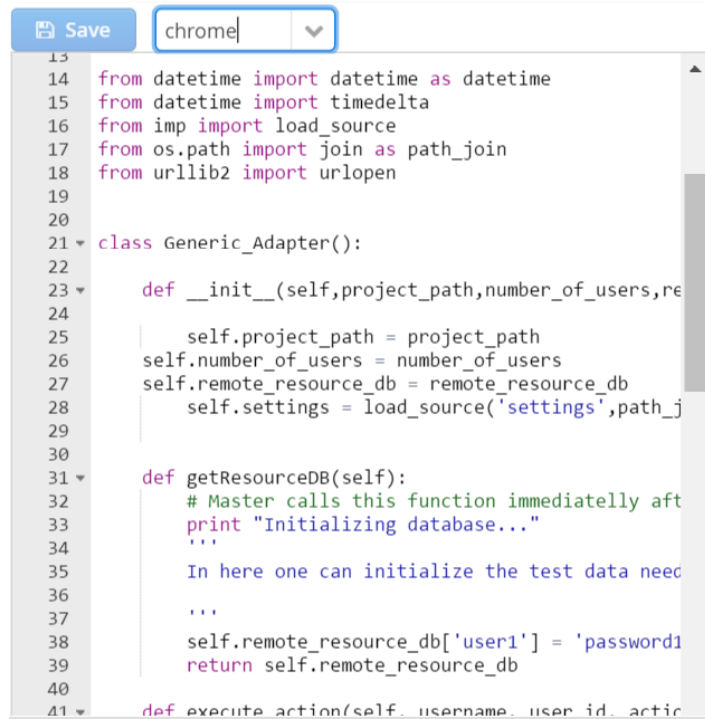
Recall from Figure 8: Vaadin Framework and Runtime Architecture which depicted 3 different widget types which have both a client side and a server side implementation. All three add-on types (Built-in, Add-on, and Custom) are utilized in the Dashboard application. Built-in components were sufficient for the majority of the application requirements. The AceEditor and DiagramBuilder described below represent Add-on components utilized. And finally FlotChart represents the Custom component implemented for this project.

Ace Editor

The first need which was not possible with Vaadin core components was the ability to write code (e.g. Python, DOT, or XML) from inside the Dashboard and save it as a file to disk. Vaadin does support several textual input field types including a rich text editor. Though it would be possible to write code in such a field and save it as any desired file type, language specific syntax highlighting and formatting are not supported by default.

The AceEditor add-on component, however, already exists in the directory [37] and is a code editor written in JavaScript which is easily embeddable in web pages. It supports a wide variety of language syntaxes, line wrapping, code folding, indentation, search and replace, and more features.

This component only required adding to the project’s `lib` directory with the other dependencies and then a recompile of the project widgetset (adding any client-side components requires the widgetset to be recompiled-which is handled easily by the Vaadin Eclipse plugin). Figure 17 displays the AceEditor component being used for writing the Test Adapter for a TestSession while using the editor color style of “chrome”.



```
13
14 from datetime import datetime as datetime
15 from datetime import timedelta
16 from imp import load_source
17 from os.path import join as path_join
18 from urllib2 import urlopen
19
20
21 class Generic_Adapter():
22
23     def __init__(self,project_path,number_of_users,remote_resource_db):
24
25         self.project_path = project_path
26         self.number_of_users = number_of_users
27         self.remote_resource_db = remote_resource_db
28         self.settings = load_source('settings',path_join(project_path,'settings.py'))
29
30
31     def getResourceDB(self):
32         # Master calls this function immediatelly after the class is created
33         print "Initializing database..."
34         '''
35         In here one can initialize the test data needed for the application
36         '''
37
38         self.remote_resource_db['user1'] = 'password1'
39         return self.remote_resource_db
40
41     def execute_action(self,username,user_id,action):
```

Figure 17: Ace Editor Add-on Component

Diagram Builder

The second requirement that was not possible with core Vaadin components was that in addition to writing DOT code, Dashboard users should be able to create and edit the PTA models in a drag-and-drop manner. This graphical component functions similar to e.g. creating such charts as workflows, UML diagrams, or even building graphs in Microsoft PowerPoint.

Another third party add-on already existed that matched these requirements quite closely, named the Diagram Builder [38]. The original component is built with JavaScript by ALLOYUI [39]. The use case of the Diagram Builder is quite simple. Nodes can be added to the chart and dragged to the desired position, and connectors can be drawn between any two nodes to form a link or flow. This follows the same pattern as that of DOT model illustrations as depicted in Figure 12 and Figure 13.

A few customizations were required to fit the Diagram Builder to the MBPeT use case. This included removing some of the default node types which were unnecessary (joins, forks, conditions, and tasks). Second, the default name of any new node or connector was the node type, followed by a random string of numbers, e.g. state1783 or connector1893. The DOT models required that the nodes be numerically named in ascending order, starting from either 0 or 1, depending on the model type. Finally, a custom controller needed to be written to

parse back-and-forth between the output of the Diagram Builder and the DOT syntax required by the PTA models. Figure 18 shows a DOT model that has been generated into the corresponding chart on the Diagram Builder.

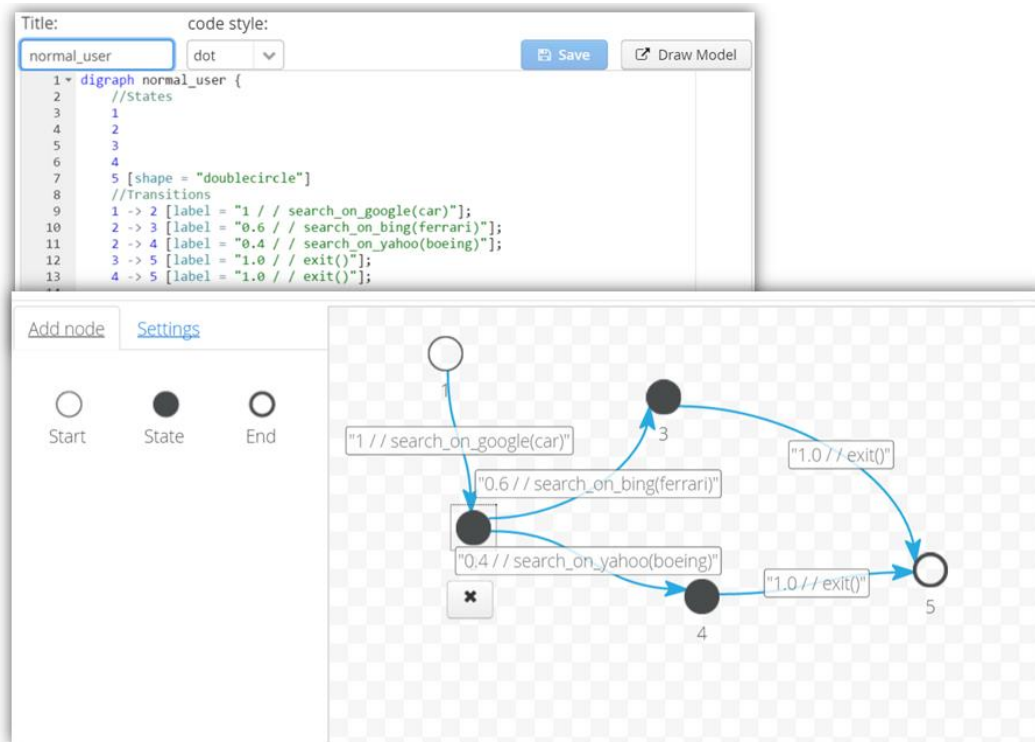


Figure 18: Diagram Builder View + Corresponding DOT Model

FlotCharts

The final requirement of the UI which is not supported in core Vaadin components are charts. Though Vaadin does not have a chart component built-in, the official Vaadin charts, although powerful and highly customizable, come in the form of an expensive commercial license. Because it was necessary that all libraries and tools used in this project be open source, another chart solution was needed. The solution chosen was to build a custom add-on component using the Flot charting library. Flot is a JavaScript plotting library using jQuery which is fairly lightweight with simple setup and usage [40]. It should be noted that Flot has been used by several sources as a reference example for how to build a custom add-on component in Vaadin [41] [42]. Although these sources laid the groundwork for how to create a Vaadin component from a JavaScript library like Flot, these tutorials only gave the simplest implementation and did not provide full API support for all of Flot's options and functionalities. Neither does there currently exist a ready-made add-on component available in the Vaadin Directory.

Due to these factors, considerable effort was needed to implement all of the charting functionalities and options set by the design and functional requirements for the Dashboard application. Among these further functions not implemented by any tutorial were:

- mouse click events for data points
- mouse hover events over data points
- drag-and-drop capability of data points
- real-time updates

It is beyond the scope of this document to write a tutorial on how to implement a JavaScript library into a Vaadin component, however a brief summary of the tasks and process taken is documented below to support the effort needed to meet the functional requirements of the project.

Flot Implementation Details

Vaadin's architecture is based on storing a server-side state [43], and a typical UI component has both a client-side and a server-side implementation. Because a chart component is a UI component, a client-side widgetset is required for user interaction. However, the state of the component is still stored on the server-side. A JavaScript component, like Flot, has a client-side rendering implementation already, but to be a proper Vaadin component it still requires a server-side implementation to maintain its state information.

The first file needed is a state class (`FlotChartState`) where the data is stored server-side and made available to JavaScript. This class can be considered as a shared state communication portal or a transport box between the Java server-side code and the JavaScript client-side code. Next, the actual Vaadin server-side component is created (`FlotChart`) by extending `AbstractJavaScriptComponent` and referencing any necessary JavaScript libraries via the `@JavaScript` annotation. A simple case requires only jQuery, Flot library itself, and the Flot connector. Because of the extended functionalities needed, extra plugins and libraries were also required:

```
@JavaScript({"js/jquery.min.js",
            "js/jquery.flot.js",
            "js/flot_connector.js",
            "js/jquery.flot.crosshair.min.js",
            "js/jquery.flot.mouse.js",
            "js/jquery.flot.JUMlib.js",
            "js/jquery.flot.resize.js"
            })
```

Data is sent to the chart as a JSON such as:

```
String data = "[[0,0],[5,5],[10,5],[12,8],[15,12]]";
chart.setData(data);
```

This data is now stored in the server side Java code of the newly created Vaadin component, however the data needs to be sent to the JavaScript chart to update the actual UI component which is powered by the reference JS library. For this a `flot_connector.js` file is created as the client-side connector. In its simplest form, the code required to plot the chart in this connector is as follows:

```
window.com_apratt_flotcharts_FlotChart = function() {  
  
    var element = $(this.getElement());  
    var state = this.getState();  
  
    this.onStateChange = function() {  
        $.plot(element, state.data, state.options);  
    }  
}
```

In order to support the extra functionalities, both all three files of `FlotChart.java`, `FlotChartState.java`, and `flot_connector.js` require extra implementations (for `datadrop` events, `plotclick` events, `plthover` events, and `real-time updates`) as well as interfaces need to be created for each function.

A final note should be mentioned regarding the JSON library needed for Flot in Vaadin. At the time of building the custom component using Vaadin version 7.3, the specific JSON implementation in Vaadin core was `org.json`. However, after v.7.4 that was changed to `elemental.json`. Because creating the Flot component was handled early on the development process, after completion it was left waiting for inclusion into the Dashboard project till a later time. Since time had elapsed and the new working draft of the Dashboard application was using Vaadin 7.5+, this introduced incompatible code which required both switching to the new `elemental.json` jar as well as updating some Java code to use the correct JSON types and methods of that library. See the following pages for reference to this issue [44] [45].

5.4.3 MBPeT Connection and Combined Architecture

The process of executing a test session using the MBPeT system was detailed in Section 3.2.1 End-User Interaction. Additionally, Section 5.3 Deployment Environment ovan briefly introduced the decision to deploy the MBPeT Dashboard alongside the master/slave MBPeT nodes on the same server. This solution required minimal alterations to the MBPeT system itself in order to make the connection to the Dashboard. Running all systems on the same file-system also made the actual execution process of MBPeT from the Dashboard a natural extension of the existing command-line process.

Executing the master and slave nodes is achieved using Java's `ProcessBuilder` class. The attributes of this class, among others, include the *command* to be executed, a configurable *working directory*, and standard *output and error streams*. Because every user of the Dashboard application has their own working directory with their own unique copy of the MBPeT master node, the system must first point the execution command to the correct file directory. The out and error streams are captured using a `StreamGobbler` thread. In the case of the master node, this output is displayed in a terminal window during a test session to provide system feedback information to the end user (see Figure 19 nedan). The command attribute can be a list of strings representing the target program/script and any valid arguments. In this case the command run on the Fedora Linux server was:

```
command = "./mbpet_cli test_project/ 1 -p 6001 -b localhost:9999 -s";
ProcessBuilder pb = new ProcessBuilder("/bin/bash", "-c", command);
```

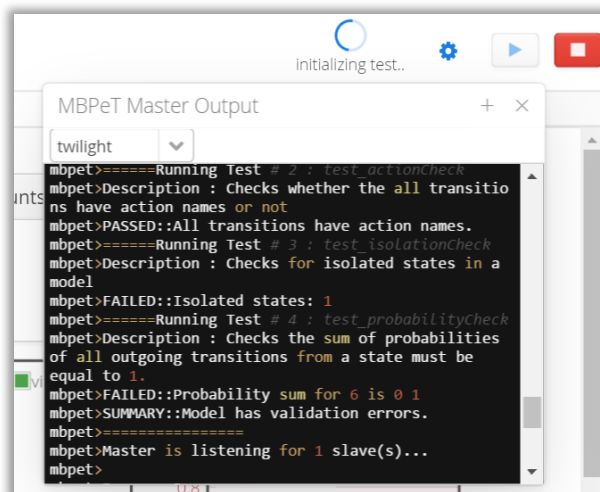


Figure 19: Master Terminal Window during Test Session

From an architectural viewpoint, although a single target platform was chosen for this proof of concept, several other deployment solutions were discussed. In regards to the three components of the online system being built (web-app, master, slave), the combined architecture chosen and being described here could be visualized as:



The advantages of this solution allowed for minimal extra effort in deploying the Dashboard and connecting it to the MBPeT system. This solution could, however present a potential risk of resource overutilization. Namely, is the CPU and memory consumption on the server going to prove detrimental when running all

three nodes on the same server instance? One alternative which still allows for a close coupling of the web-app and master node would be to deploy only the slave node(s) remotely on a separate server (e.g. Amazon EC2) as depicted below. This is still possible under the present solution because instead of the Dashboard executing the slave process directly on the host system, it could easily execute a custom script which could in turn trigger a slave process on a remote host.



Naturally, a further extension of this principle could be to also trigger the master via a script rather than executing it locally. With the current solution, this is still technically feasible. However, the primary drawback to such a solution is that an alternative method of retrieving the master's terminal output would need to be implemented. Running a local master enables simple capture of its output, which is displayed for Dashboard users in the terminal-style window during a test session. Deploying a remote master would require additional transmitting of the master's output back to the Dashboard application, if that functionality was still desired. It is recommended to have this output as it provides end users with confirmation and up to the second monitoring of the master node's status. This final solution of all three nodes running on their own host system would look as follows:



In summary, the *limitations* of this combined deployment architecture are likely to revolve around resource allocation. As all three nodes (web-app, master, and slave) require both CPU and memory, how scalable this solution is remains to be seen. The question of scalability will be discussed later on and is dependent on a large number of factors and the optimization techniques implemented. The *extendibility* of this solution could prove a useful principle for future projects. In addition to the above described relocation of master and slave nodes to remote systems, in theory this solution could be extended to other services and tools as well, particularly any such tool that can be externally launched with input parameters and requires minimal further user interaction.

5.4.4 System Inputs and Outputs

A word should be said regarding the inputs and outputs of the Dashboard application. This does not refer to the architecture of the system or which technologies are used at each level (e.g. back-end, server-side, client-side

technologies), as these have been described already in this chapter. This section is concerned only with the end-user interactions via the UI and the outputs generated by the system as a whole and presented via the browser interface.

Naturally the first I/O's worth mentioning include standard user account operations such as creating a user, signing in, and editing user profile. We are primarily concerned here with I/O's relating to MBPeT and what value is added by this Dashboard web application. In short, the web-app's two *input* categories include: test configurations and test execution data, while the *output* includes: test execution monitoring data (textual and charts) and test reports.

Following the requirements of an MBPeT based test, the necessary inputs for a test session against a target SUT must include models, test parameters, and test adapters. From a UI perspective, an input model is accepted via a DOT file typed in an embedded text editor or by the aforementioned DiagramBuilder component. The settings (test parameters) can be input manually via text editor or a form, and the adaptors (Python and XML) are also text files entered via an embedded editor component. Figure 20, Figure 21, and Figure 22 nedan demonstrate entering these three input types in the Dashboard application.

During the execution of a test session the output of MBPeT (master node terminal output and data updates via JSON and UDP packets) is, from the Dashboard perspective, provided as inputs to the Dashboard application. These inputs are in turn processed to provide, as output, real-time system monitoring throughout the test cycle. These outputs are visible to the user on the MonitoringTab in the form of textual information (e.g. "virtual users:9" or "total sent bandwidth:20MB") and charts (e.g. user ramp, aggregated response times, individual action response times). Figure 23 depicts the Dashboard application in terms of its inputs and outputs.

The last outputs the Dashboard application provides are the final test report files which are generated by the master node of MBPeT. Therefore these test reports can be seen from one perspective as inputs (because they are not generated and processed by internal functions like the monitoring data) as well as outputs (because they are outputted to the end-user of the Dashboard). Figure 24 displays a screenshot of the Report Tab viewing a final test report generated automatically by the MBPeT tool.

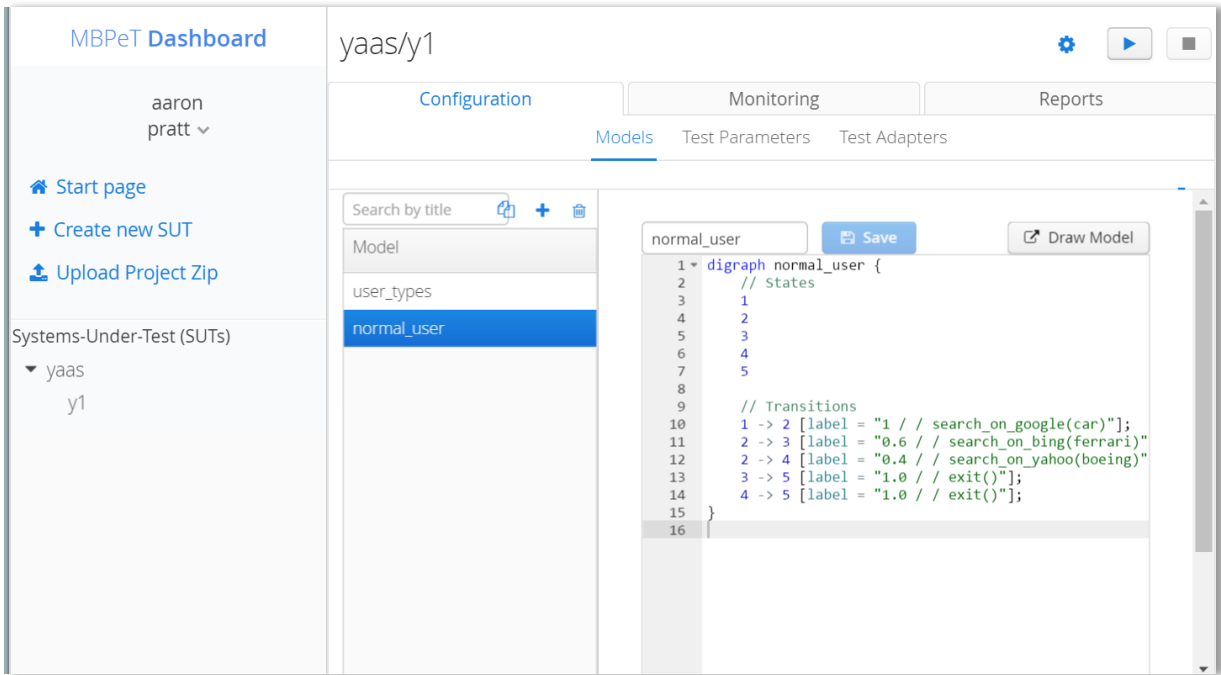


Figure 20: Model Input in Dashboard app

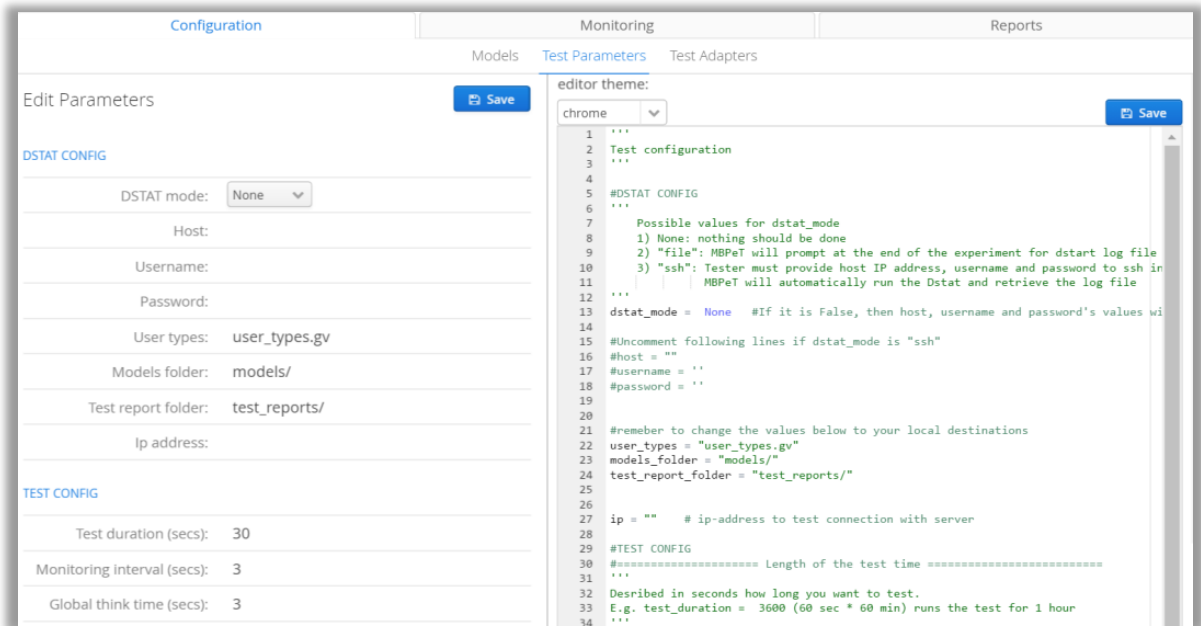


Figure 21: Settings Input in Dashboard app


```

1 from datetime import datetime as datetime
2 from datetime import timedelta
3 from imp import load_source
4 from os.path import join as path_join
5 from urllib2 import urlopen
6
7
8 class Generic_Adapter():
9
10     def __init__(self, project_path, number_of_users, remote_resource_db = {}):
11
12         self.project_path = project_path
13         self.number_of_users = number_of_users
14         self.remote_resource_db = remote_resource_db
15         self.settings = load_source('settings', path_join(self.project_path, 'settings.py'))
16
17
18     def getResourceDB(self):
19         # Master calls this function immediately after __init__
20         print "Initializing database..."
21         '''
22         In here one can initialize the test data needed for the test
23         '''
24
25         self.remote_resource_db['user1'] = 'password1' # Example
26         return self.remote_resource_db
27
28     def execute_action(self, username, user_id, action, parameters):

```

Figure 22: Adapter Input in Dashboard app

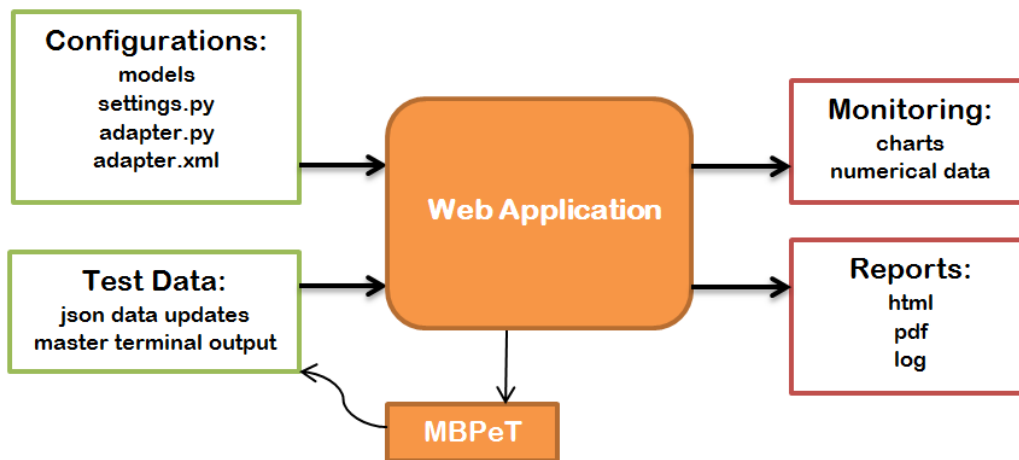


Figure 23: I/O for MBPeT Web Application

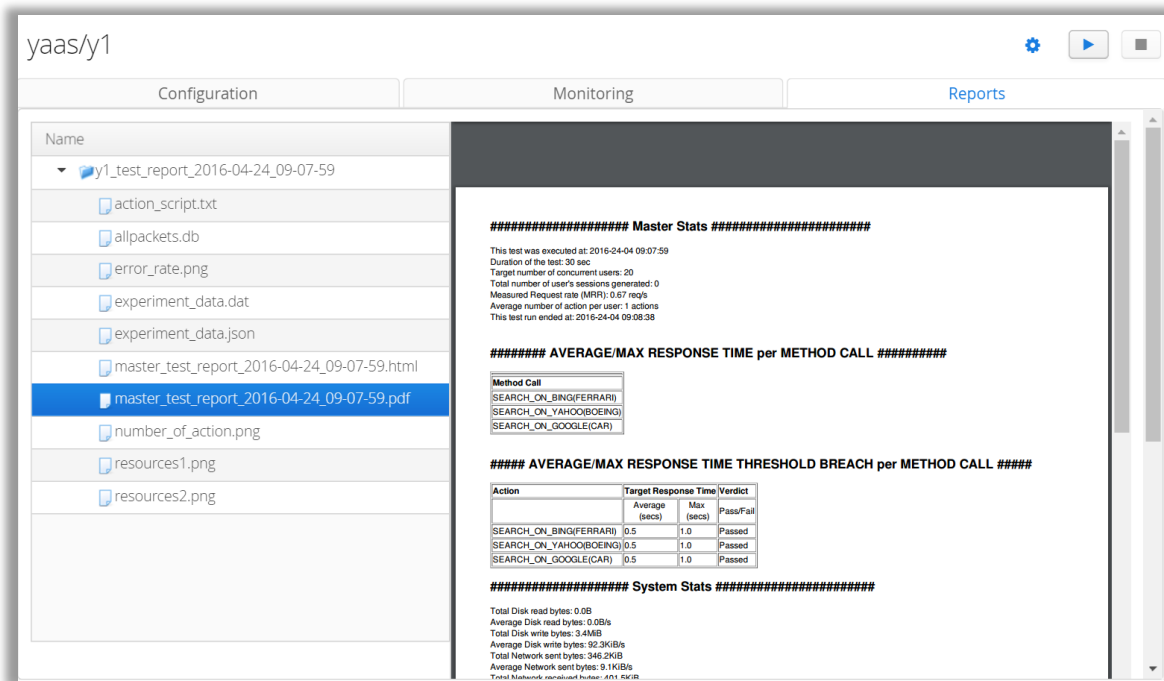


Figure 24: Final Test Report Output from Dashboard app

5.4.5 Additional Components and Contributions

A number of additional features of the Dashboard application should be mentioned which are either critical to the overall functionality of the application or crate added value to the original MBPeT system which did not previously exist. This section will briefly name and detail a few such contributions.

UDP Listener Client

In order to gather up to the second monitoring data back from the MBPeT system during a test execution phase, the Dashboard needs to have a communication channel with the master node. Without a communication channel the web-app would naturally be in the dark and receive no monitoring data (e.g. active slaves, no VUs, throughput, error rate, success rate, bandwidth, action response times etc.). It was decided the simplest method to achieve the necessary data transfer for the current requirement set was to utilize the connectionless UDP protocol rather than a constant connection oriented TCP solution.

The process is straightforward. Before initializing the master node, the `UDPThreadWorker` class first creates a `DatagramSocket` which is automatically bound to any available port. The reason this happens first is so that the system can then inform the master node what `[IP address : UDP port]` pair to send data updates over. The UDP client then runs in a loop until either the user interrupts the test or the test is complete. In this loop, a `DatagramPacket` first receives any

packets received over the port, after which the JSON is extracted and parsed by the `JsonProcessor` utility (see below) and the pertinent data is sent to its appropriate destination.

This class is aptly named a “thread” worker for two important reasons. First, because this process runs for the entirety of the test duration, be it 30 seconds or 2 hours, it cannot run in the main application thread as no other processes could efficiently be concurrently run. Second, running a separate thread, together with the enablement of Push capability in Vaadin, the data which is parsed is able to be pushed to the monitoring tab and the UI components updated.

JSON Processor

UDP was chosen as the transmission protocol to communicate data updates to the Dashboard, and the syntax for these update was decided to be JSON. Although XML would be another obvious choice, JSON was ultimately considered to be more lightweight, more readable, and is easy to parse with Java APIs. The following is an example JSON object sent over UDP:

```
{
  "timestamp": 1447420385.642112,
  "summary": {
    "net_send_total": 2611.0,
    "throughput": 2,
    "net_recv_total": 7967.0,
    "resp_avg": 1.2429685000000001
  },
  "values": {
    "resp": [
      {"action": "search_on_yahoo",
        "timestamp": 1447420384.800802,
        "tag": "resp", "val": 0.772451},
      {"action": "search_on_google",
        "timestamp": 1447420384.978765,
        "tag": "resp", "val": 0.367306}
    ],
    "net_recv": [
      { "timestamp": 1447421894.533734,
        "tag": "net_recv", "val": 34449}],
    "net_send": [
      { "timestamp": 1447421894.533734,
        "tag": "net_send", "val": 7030}],
    "error": []
  },
  "target_user": 9,
  "slave_name": "Slave 1"
}
```

The `JsonProcessor` receives as input the JSON extracted from the UDP packet, parses the JSON, and then returns as output back to the UDP client the various data objects to be pushed back to the UI in their appropriate locations. For example, the slave name is reported to the Slaves Panel to indicate which slave

nodes are currently active; the bandwidth, user count, success, and error are reported to their relative panels; virtual user count (ramp) chart is updated; and the response times are calculated and displayed in the both the aggregated response panel for all actions, aggregated response chart of all actions combined, and the individual response time charts.

Individual Action Monitoring

These monitoring functionalities just described provide a significant contribution to the Dashboard application in contrast to the existing command-line or GUI application because previously only minimal data updates were provided such as the ramp and aggregated response time. These monitoring settings are configurable at the start of the test session via the cog wheel next to the start button in the top right corner of the Session View (see Figure 25 nedan). A sub-window opens providing three contexts of optional settings. The first is a twin column selector listing all individual actions related to this test session. These actions correspond to the transitions between nodes in the DOT models (see Figure 12 and Figure 13) and are signified in the `settings.py` as *target response times* (TRT). Any selected actions have their data then displayed in two ways: 1) the aggregated response time of that action is displayed in a textual panel, and 2) a chart is generated and updated every second with the most recent response time of that particular action. Figure 26 nedan shows the panel and charts displaying the average and individual response times respectively.

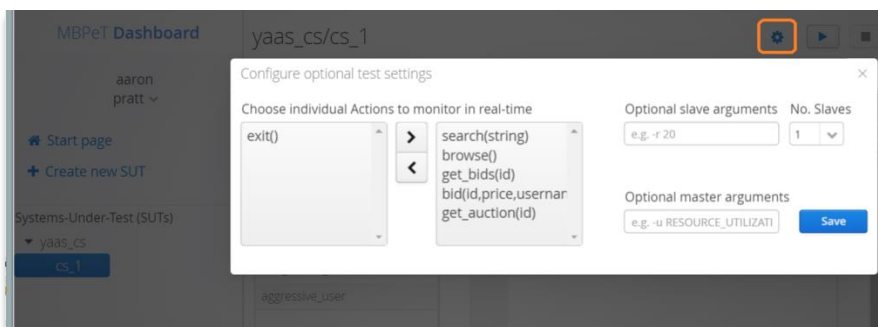


Figure 25: Optional test configurations window

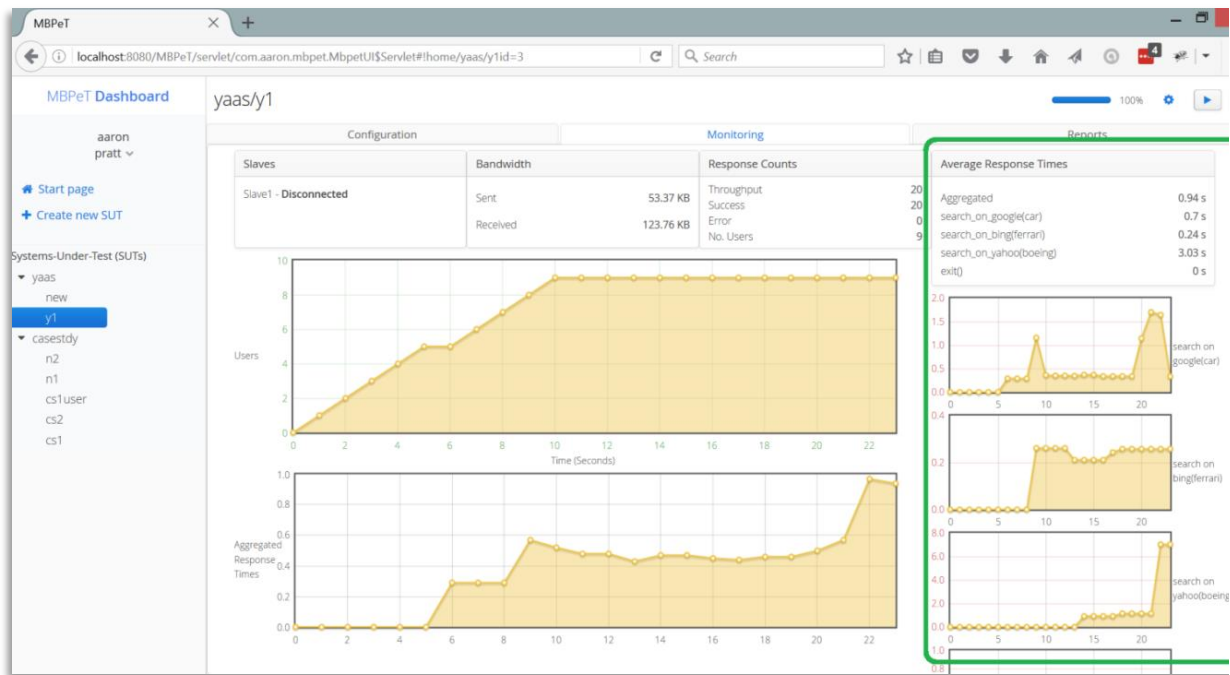


Figure 26: Monitoring Tab during test session, with aggregated and individual response times highlighted.

PDF Reports

The MBPeT tool automatically generates a final test report at the successful completion of a test session. This report includes individual chart images, an action script file, a log file, and an HTML master report. One limitation of the architectural deployment scenario already explained (see 5.4.3 MBPeT Connection and Combined Architecture), is that the location on disk where MBPeT and the Dashboard application host the project files is not located within a web server directory which can serve files over the internet. This meant that the HTML master report file, which links to the image files (using standard html ``) failed to dynamically load the images in a web view from inside the Vaadin application. As a workaround to this, the `wkhtmltopdf` command-line tool was utilized to generate a pdf version of the master test report. This tool is executed directly on the Linux OS using the ProcessBuilder technique previously explained. The command executed looks similar to the following:

```
String command = "wkhtmltopdf " +
    html.getAbsolutePath() + " " +
    destinationFolder + "/" +
    FilenameUtils.removeExtension(html.getName()) + ".pdf";
```

Concurrent Web-App / MBPeT Users

One of the purposes for deploying a web application, in contrast to a desktop application, is the ability to have multiple concurrent users from a “single installation”. Instead of every user downloading and installing on their local machine the MBPeT tool; it should be available for use by those same users via a single URL. In order to accomplish this, the way MBPeT itself functions presents three specific hurdles to overcome.

The first hurdle is in regards to the MBPeT master directory itself. Launching the `mbpet_cli` file initializes the master node and begins the testing procedure. Even though one could execute that same physical file but give two different port numbers as parameters, it would not be sufficient to run concurrent users from the MBPeT perspective. The entire master node directory structure must be unique for each test session running on a single computer. The solution chosen was to have a master copy of the master node reside in the shared base directory of the Dashboard and MBPeT (see `nedan`). Whenever a new Dashboard user account is created, the entire master directory is copied inside that user’s private home directory (inside the ‘users’ directory below). This ensures that each user will be running their own unique instance of MBPeT, and therefore not interfering with test results of other concurrent users.

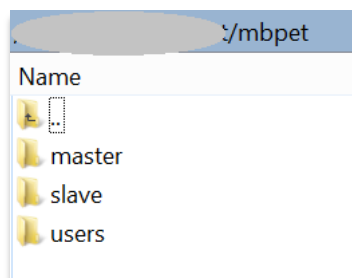


Figure 27: Web-App / MBPeT shared directories

The second obstacle to accomplishing concurrency in this respect is to ensure that any simultaneous MBPeT test sessions being conducted on the web server via the Dashboard application utilize a unique TCP port. Recall that the master and slave nodes are separate entities and can be deployed even on completely separate machines (e.g. master on the Åbo Akademi web server and slave(s) on Amazon). Throughout the test duration the master and slave node(s) communicate via TCP connection. That port is by default set to 6000, but can be set using the argument `-p [port number]`. Because all Dashboard users are running on the same physical system, concurrent users cannot utilize the same TCP port for master/slave communication. The test may not fail or crash, but the effect is that both, e.g. Joe and Jane receive faulty results as they both receive

traffic from each other's test session. To solve this, the `MasterUtils` class chooses, at random, an available port and uses that as the `-p` argument when executing the process. The built-in constructor argument: `new ServerSocket(0)` accomplishes this random assignment.

The final issue to handle for executing an MBPeT test session from the Dashboard application is a repeat issue explained above in the UDP Client. That is that fact that, like the UDP client, the master and slave nodes are simultaneously running during the entire test duration. Therefore, both master and slave processes are handled in separate threads, apart from the main thread of the Vaadin application. The code snippet below shows an excerpt of the thread responsible to start the master process and also shows how to update the UI in a thread safe manner using `Push`.

```
public void startMaster(...) {
    new Thread(new Runnable() {
        @Override
        public void run() {
            ...
            //update UI thread-safely
            UI.getCurrent().access(new Runnable() {
                public void run() {
                    masterTerminalWindow.insertDataToEditor(
                        new StringBuilder("mbpet>" + command));
                }
            });
        }
    }).start();
}
```

Deployment Optimizations for Scalability

A necessary step that should not be neglected when deploying a web application involves properly configuring the deployment environment and optimizing the hosting setup. Optimizing a Vaadin application differs very little from any other web application and [46] and [47] provide a good place to start. The following steps were taken to provide an acceptable level of optimization.

Enable gzip compression. Web applications transfer data over http request. Textual content such as HTML, CSS, JavaScript, and JSON in particular can be significantly compressed to save bandwidth.

Configure the JVM memory parameters. One solution to running out of memory is to add more memory. The amount of memory given to the Java Virtual Machine (JVM) of course depends on the host system and the resources available. The server hosting the Dashboard application was configured with the settings: `-Xms3g -Xmx6g -Xss512k -server` where `-Xms` and `-Xmx` represent the

minimum and maximum heap size respectively and the `-Xss` can be used to reduce the stack size of threads.

Optimize the Vaadin Widgetset. The client-side engine, or widgetset, is often the largest single transfer by a Vaadin application [47]. To its credit, Vaadin core contains many UI components and having all of these in the widgetset requires them to be transferred to the client-side engine. By optimizing the widgetset, one can limit the components loaded to only those actually used by your given application, thus reducing space consumption.

Configure Apache for high concurrent mode. The `server.xml` file of apache tomcat can be altered such as setting a max thread count and max number of concurrent connections:

```
<Connector port="8282" protocol="HTTP/1.1"
  connectionTimeout="20000" redirectPort="8443"

  maxThreads="4096"
  acceptCount="2048 "
  maxConnections="2048 " />
```

Tune the JVM's garbage collection. Rather than having the JVM wait till almost the entire heap is utilized, the garbage collection technique and settings can be altered to run more frequently or in shorter periods. Garbage first (G1) mode is one suitable option for heap sizes larger than 4 GB and is set with one simple argument:

```
-XX:+UseG1GC
```


Chapter 6

6. CASE STUDY

6.1 Case Study: YAAS

The target SUT chosen for the case study is the YAAS (*yet another auction site*) web application and RESTful web service. This was chosen as it was the same system used for the proof of concept case study in the MBPeT tool official documentation [7]. The auction site provides varied use case scenarios as it supports services and functionalities allowed for both non-registered and registered users only. These services include browsing, searching auctions, retrieving auctions, and bidding. YAAS was developed in Python on top of the Django web-framework.

Three different virtual user types are defined by the PTA models: passive, aggressive, and non-bidder users—based on the amount of interaction with the system-under-test’s database. Each model defines a unique user profile that exhibits different use cases and patterns of activity against the SUT.

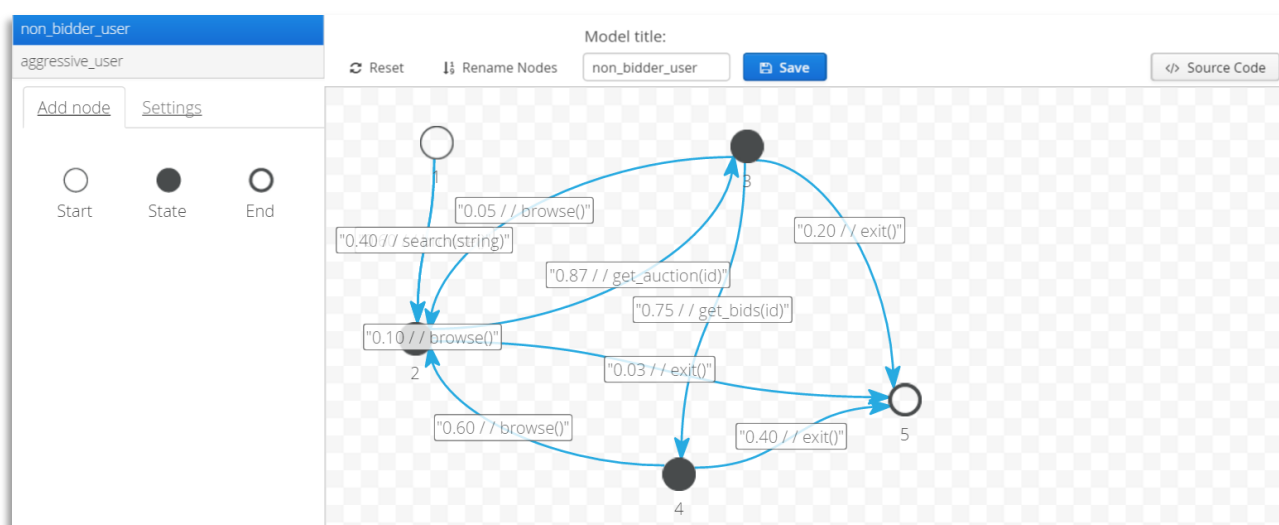


Figure 28: Non-bidder User Model

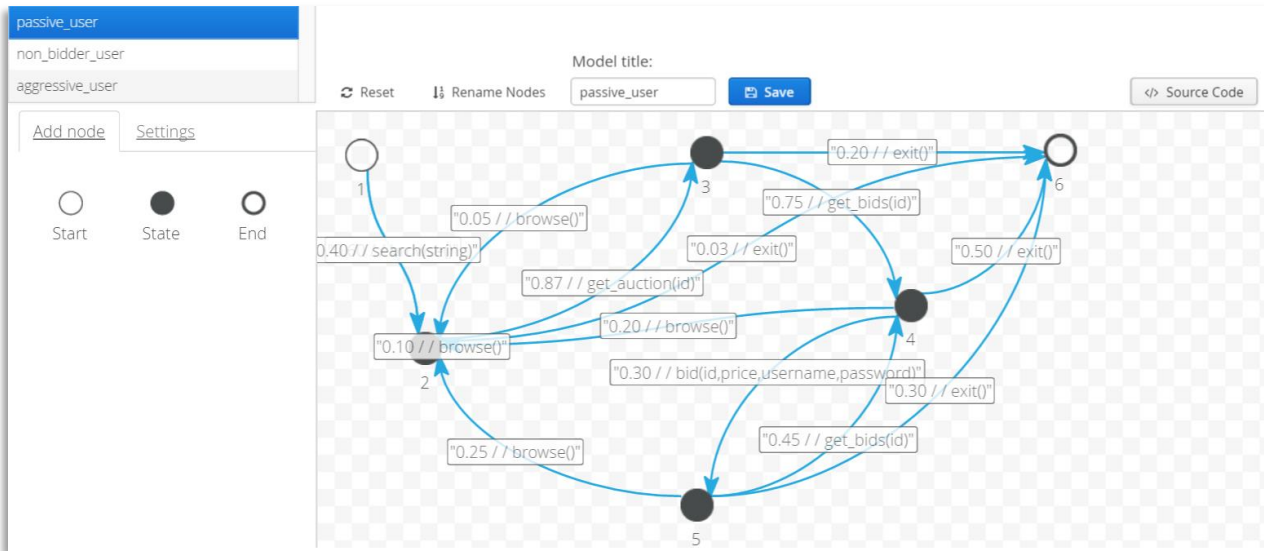


Figure 29: Passive User Model

6.2 Test Architecture and Tooling

Architecture

It has already been established that the web-app, master, and slave(s) are all deployed to the cloud2 server featuring an 8-core CPU, 16 GB of memory, 1 Gb Ethernet, 7200 rpm hard drive, and Fedora 16 operating system. The SUT is deployed on a similar server with the same hardware configurations also residing on the Åbo Akademi network.

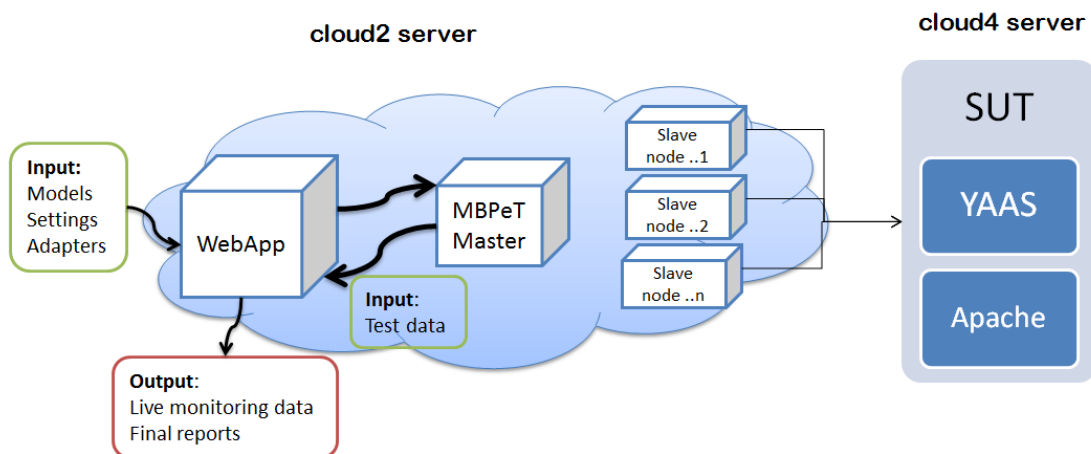


Figure 30: Test Architecture

Tooling

Two separate tools recorded test data during all test sessions. The first tool was the Java VisualVM monitoring and lightweight profiling tool [48]. This tool provides finer tuned monitoring information of Java-based applications running on a JVM (Java Virtual Machine). After installation and minimal setup, VisualVM was able to provide detailed performance analysis of the Tomcat server in real-time. This data is filterable to show only classes, threads, variables etc. of the Dashboard application itself, as well as the Tomcat server as a whole. Of particular interest was the monitoring of the CPU and memory (heap) in order to trace any memory leaks. This first set of data will provide performance evaluation for the Dashboard as a standalone unit.

The second tool was the Linux `top` command to directly monitor the Java application process running on the deployment web server. A specific set of arguments and configurations allowed us to monitor those processes initiated by Java on the system. Initially, the Tomcat server and the subsequent Dashboard application are the only Java processes being utilized for this project. However, because the Java ProcessBuilder APIs were used to execute the master and slave nodes, the `top` output registers these processes as sub-processes of Java. Therefore the master and slave CPU and memory load is included in the output of `top`. This is confirmed by two ways. Firstly, the `htop` tool [49] with its built-in tree sorting function confirmed that both `mbpet_cli` (master) and `mbpet_slave` were running as sub-processes of Java. Second, the more limited standard `top` command lists no MBPeT processes at all when they are initiated from the Dashboard. Together with the `htop` results, this confirmed that `top` actually measures all three systems (`we-app` + `master` + `slave`) in on output. This second monitoring data will therefore provide performance evaluation of all three systems together as a unit, as was discussed in Section 5.4.3 MBPeT Connection and Combined Architecture.

Only a small number of concurrent users were required in order to give an initial evaluation of the system performance. Therefore it was not necessary to employ an automated stress testing tool such as Gatling or JMeter in order to simulate users.

6.3 Experiment Scenario

The goal of the case study was to provide a proof of concept for the solution pursued in this Dashboard application. Specific criteria were twofold: 1) to demonstrate that the Dashboard successfully fulfils the functional requirements and completes test sessions against a SUT with MBPeT, and 2) to evaluate the scalability of the Dashboard application on a small scale. CPU and memory

usage of the host server was defined as the metrics to be monitored in order to measure the effective scalability of the solution. Of particular focus was the monitoring of system memory during MBPeT sessions *greater than* ten minutes in duration as well as multiple *concurrent users*. Both of these two factors are considered to be potential bottlenecks regarding memory usage where scalability is concerned.

The expected results are that all user operations from the Dashboard will require significantly less resources with regard to CPU and memory utilization than running/monitoring an MBPeT test session. This hypothesis is based on the implementation details of monitoring such a test session. Specifically, the processing and memory load is expected to increase substantially since the UDPThreadWorker will be receiving data at 1 second intervals, performing calculations and data parsing, and updating the UI thread-safely via push.

The use case for a single Dashboard user included: log in, create or edit the test configurations (models, parameters, and adapters), run a 15 minute *duration* test session against the YAAS application, retrieve and view the report files generated, and finally log out of the system.

In addition to the duration, two additional test settings worth mentioning are the *Ramp List* and the action *TargetResponseTimes*. The Ramp value for all tests was set to [(0, 0), (120, 20)] which denotes that MBPeT will scale the total number of virtual users up to 20 VUs by 120 seconds (2 minutes) into the test session. After that point, MBPeT will constantly apply a load of 20 concurrent VUs, according to the model profiles, until the end of the test session. The specific actions (TRT), as defined in the models, are listed in the settings.py file as such:

```
TargetResponseTime = {
    'search(string)': {'average': 3.0, 'max': 6.0},
    'browse()': {'average': 4.0, 'max': 8.0},
    'get_bids(id)': {'average': 3.0, 'max': 6.0},
    'bid(id,price,username,password)': {'average': 5.0, 'max': 10.0},
    'get_auction(id)': {'average': 2.0, 'max': 4.0},
    'exit()': {'average': 1.0, 'max': 1.0}}
```

Finally, as previously discussed in the tooling section above, the first and primary focus of the evaluation is the performance of the Dashboard as a standalone unit. This data will be monitored by VisualVM. Additionally, the `top` output will provide the same data metrics but including the load of the MBPeT master and slave. This combined resource load will assist in evaluating the scalability of the current combined deployment scenario. By this evaluation, it

should be possible to determine if a remote deployment of the slave nodes is urgent, or if the current solution is scalable.

6.4 Experiment 1

The goal for this experiment was to set a baseline measurement of the load the Dashboard application generates on the system with just a single user. By this we can monitor the variation between a single user and a multi (concurrent) user scenario.

The deployment optimizations in Section 5.4.5 Additional Components and Contributions presented the final settings chosen after this first round of tests which provided an opportunity to make needed adjustments. Therefore the only optimizations made before this experiment were: optimizing the Vaadin Widgetset, configuring Apache for high concurrent mode, and setting the JVM memory parameters to

```
-Xms2g -Xmx2g -Xss512k -server
```

Experiment 1a

This first experiment executed with acceptable resource response until the MBPeT test session exceeded 10 minutes in duration (recall the duration for all tests were set to 15 minutes). At this point the heap memory consumption of Tomcat significantly increased as did the CPU usage, as confirmed by both the `top` system monitor and the VisualVM Tomcat specific monitoring tool (see Figure 31 and Figure 32). This combination of events dead-locked the Dashboard application and the MBPeT test session failed to proceed to completion.

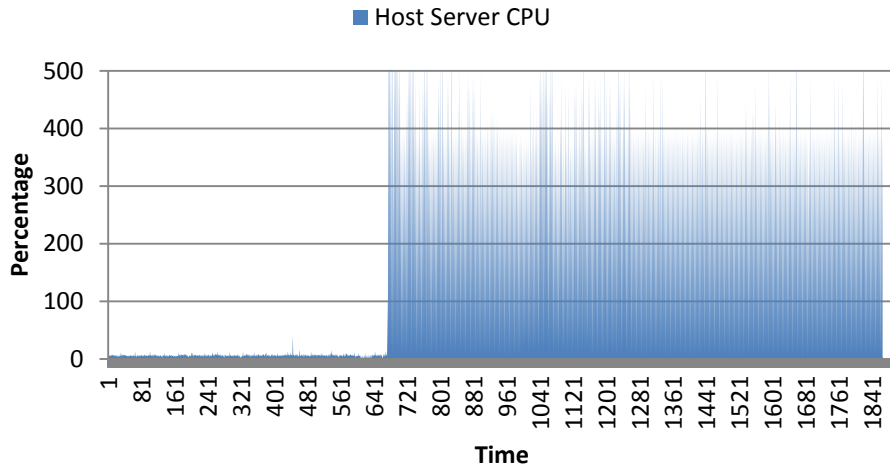


Figure 31: Combined Performance on Host Server. Experiment 1a

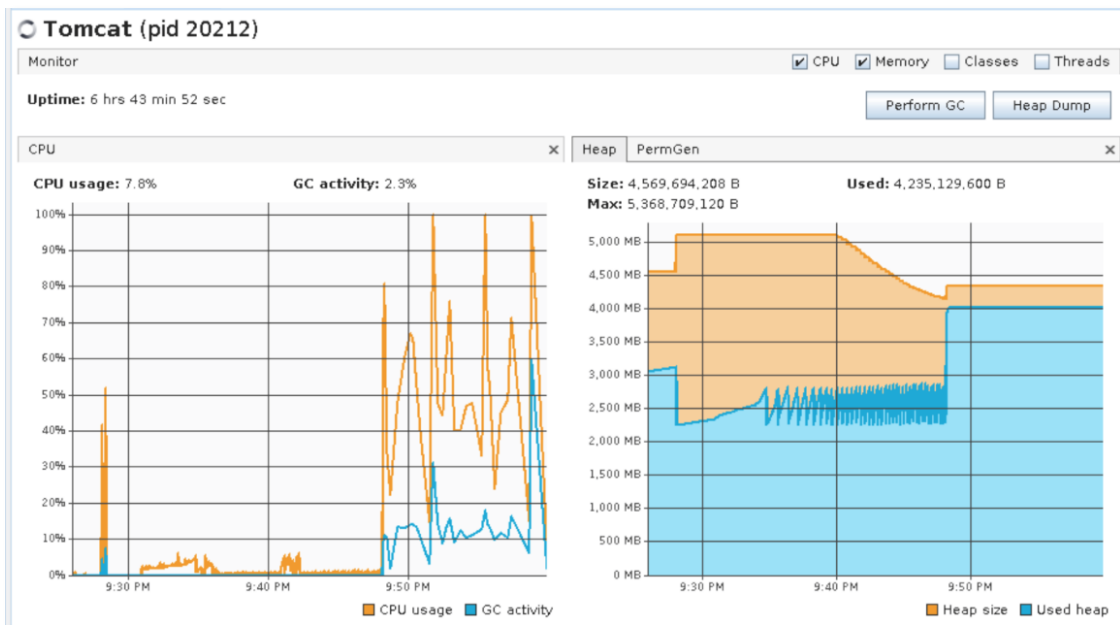


Figure 32: VisualVM-Tomcat CPU and Memory charts. Experiment 1a

The result of this first experiment revealed two flaws which demanded immediate correction. First, after investigation into memory usage records provided by VisualVM, the length of the Master Terminal Window on the MonitoringTab was discovered to be growing too large for a test of this lengthy duration. The original implementation added all new master terminal output to the AceEditor display and never reset the editor. This choice was to provide end-users with the ability to access the entire master output for debugging or reporting purposes. However, the growth of the editor past 3000+ lines caused both a strain on the AceEditor component as well as adverse performance from

using `String` concatenation to add the 1 second interval led updates. To counteract this first issue, a `StringBuilder` was substituted in place of the `String` data type and the editor implementation was altered to clear and reset the content after roughly every 100 lines.

The second short-coming which lead to the system failure was a lack of sufficient optimizations at various levels of the Tomcat deployment. First, since one of the solutions to running out of memory is ironically, to add more memory, the JVM memory settings were increased to `-Xms3g -Xmx6g` allocating up to 6 GB of memory to Tomcat. Second, gzip compression was enabled with the following setting (`compression="on" compressionMinSize="2048"`) which reduced the page size and page load time from 2.3KB and 270ms to 965B and 150ms respectively (see Figure 33). Third, the default garbage collection mechanism of Tomcat was overridden using argument `-XX:+UseG1GC` to use instead the G1 garbage processing tactic.

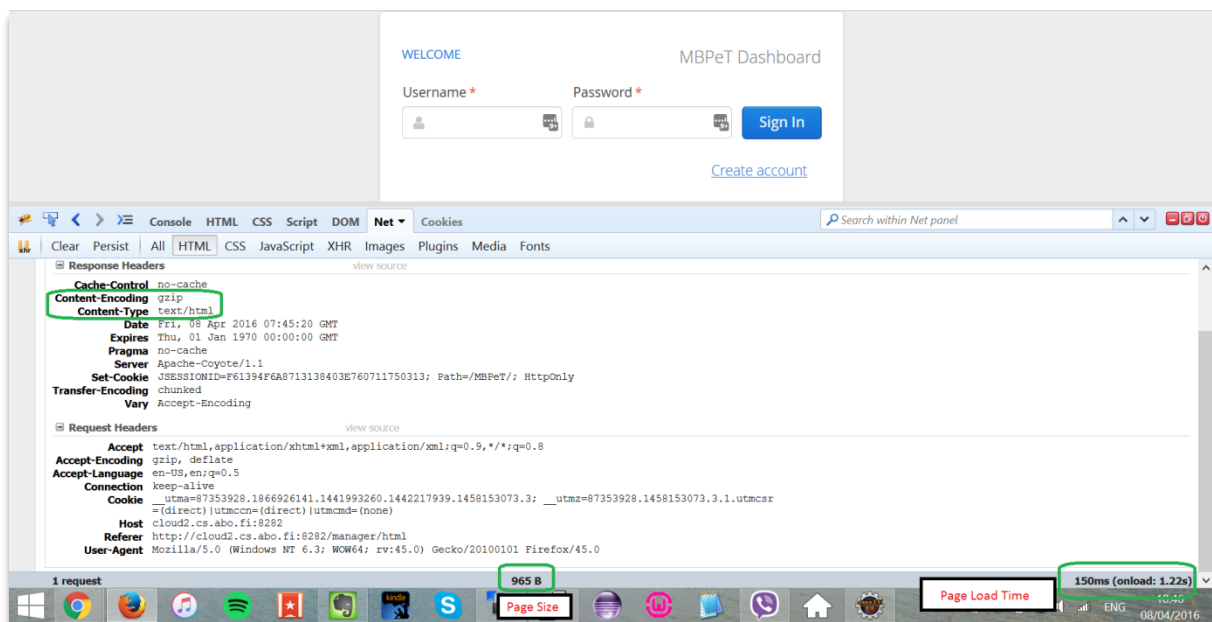


Figure 33: Reduced page size and load time with gzip compression

Experiment 1b

After making these adjustments, the first experiment was run again a second time with significantly better results. The experiment ran through the use case successfully with no dead-locks, and MBPeT successfully completed the 15 minute test against YAAS.

The CPU usage of the Tomcat server averaged close to 1% with a maximum peak of 15%. The results show that the memory leak has been fixed as a consistent pattern of memory consumption was recorded (see Figure 34).

At least two conclusions can be deduced from the heap memory record as displayed in Figure 34. First, the pattern displayed indicates that all end-user functions of the Dashboard, excluding running and monitoring an MBPeT test session, utilize significantly less memory than the test monitoring process. This deduction can be made by examining the memory chart. With the exception of a clear 12-15 minute increase in rapid memory usage, the heap memory grows at a slow and steady rate both before and after the 15 minute test monitoring process. Second, focusing on the jagged-edge section of the memory chart, where utilization increased substantially, confirms that the monitoring process of receiving and parsing data and then updating the UI does in fact produce a greater load on the system. However, the memory consumption at this stage still displays a stable and expected flow. Although the memory is used up at a much faster rate, as it should given the implementation, both the customized JVM garbage collection appears to be functioning more effectively than in experiment 1a and also the stable increase and decrease in memory, without the indication of a memory leak, show that the actual resource load follows a pattern expected by the implementation and no apparent memory leaks or bottlenecks exist.

Lastly, the Threads chart at the beneath CPU and memory displays the correlation between increased users and activity and memory consumption. Simple observation shows that the time period of the thread activity increase correlates to the same time period of increased memory consumption.

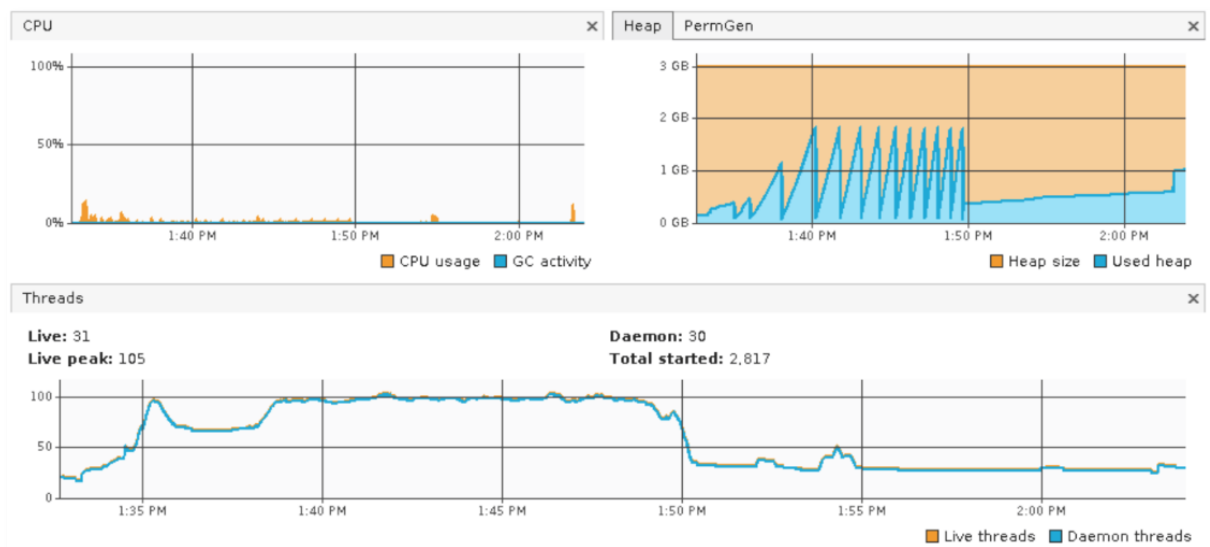


Figure 34: VisualVM-Tomcat CPU, Memory, and Threads charts. Experiment 1b

The `top` results, containing the combined data of all three systems (web-app + master + slave), differed only slightly from the singular results above. Whereas

the singular CPU usage averages around 1%, the combined data averaged at 5.3% throughout the entire experiment with minimal peaks in CPU load.

While the memory utilization of Tomcat peaked at just below 2 GB before garbage collection, the combined memory performance (see below) peaked just over 2100 MB at a calculated average of 980 MB. Both the Tomcat specific as well as the combined architecture chart acknowledges that the memory consumption rises and falls at a fast rate depending on the process being executed.

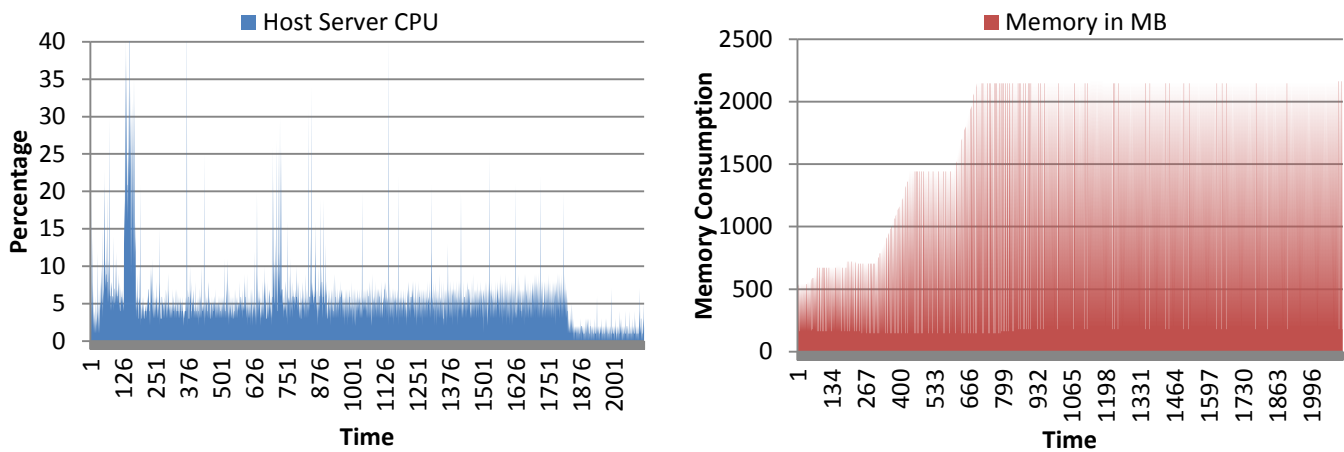


Figure 35: Combined Performance on Host Server. Experiment 1b

6.5 Experiment 2

The goal for this experiment was twofold, 1) to prove that the Dashboard application supports concurrency in both the test configuration phase and in the test execution/monitoring phase, and 2) to measure the performance and compare it experiment 1 in order to evaluate the scalability at a small scale. For this experiment, 5 concurrent users ran simultaneously through the same workflow as Experiment 1 and successfully completed the 15 minute MBPeT test against the YAAS application.

The Tomcat and Dashboard specific monitoring results yielded highly similar outcomes to those of Experiment 1. The CPU consumption increased almost negligibly from 1% on average to 2.7% with a highest peak of 13%.

The heap memory followed the same pattern as in Experiment 1 with non-test functions requiring small amount of memory and then rapid memory utilization during the test monitoring process. The peak memory limit before garbage collection remained right at 2 GB, far from the max heap limit which was set to 6

GB. The only notable difference between the first two tests is that the rate at which the heap memory grows has increased. This is naturally expected due to the increase from 1 to 5 users.

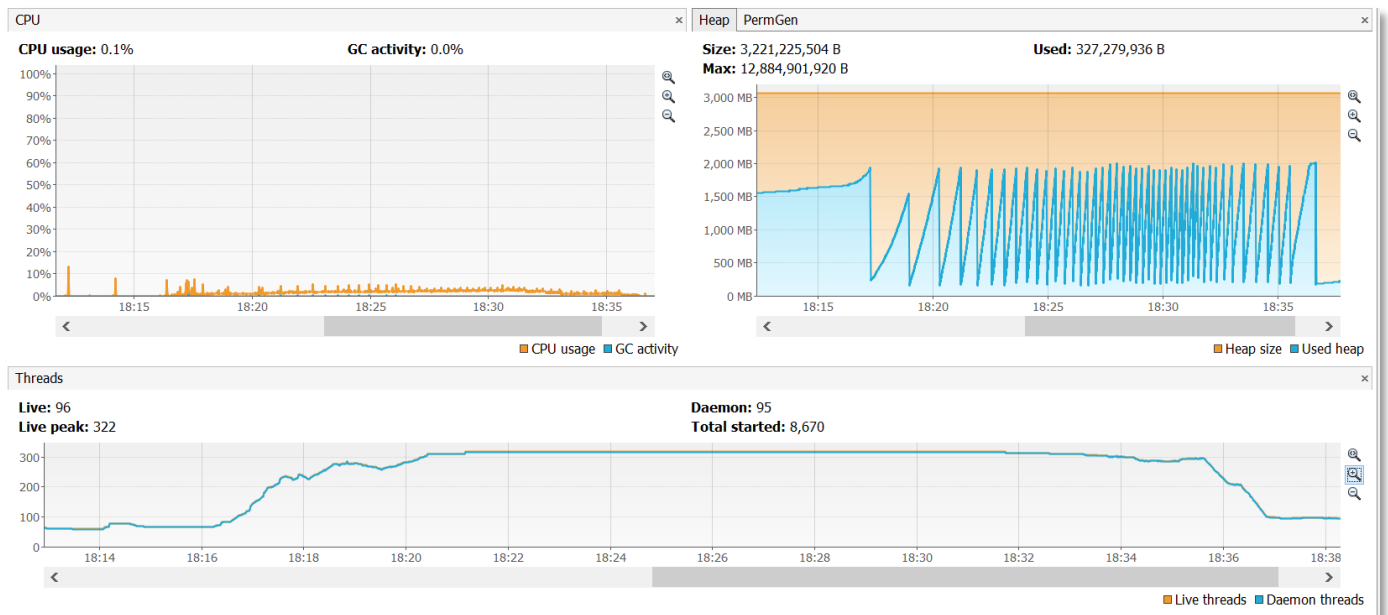


Figure 36: VisualVM-Tomcat CPU, Memory, and Threads charts. Experiment 2

The `top` data yielded a similar change from those of experiment 1 as did the Tomcat specific results. CPU usage in Experiment 1 measured an average of 5.3% and rose to 8.7% in Experiment 2. This average value rose only slightly despite the fact that the scattered CPU load did increase overall for a larger time period compared to the fairly flat results in the first experiment.

The memory average increased from 980 MB to only 1240 MB with the increase in users. The peak memory level increased only from 2100 MB to 2359 MB. As the Tomcat heap memory still maxed out at 2 GB, the combined memory usage is still less than 400 MB more.

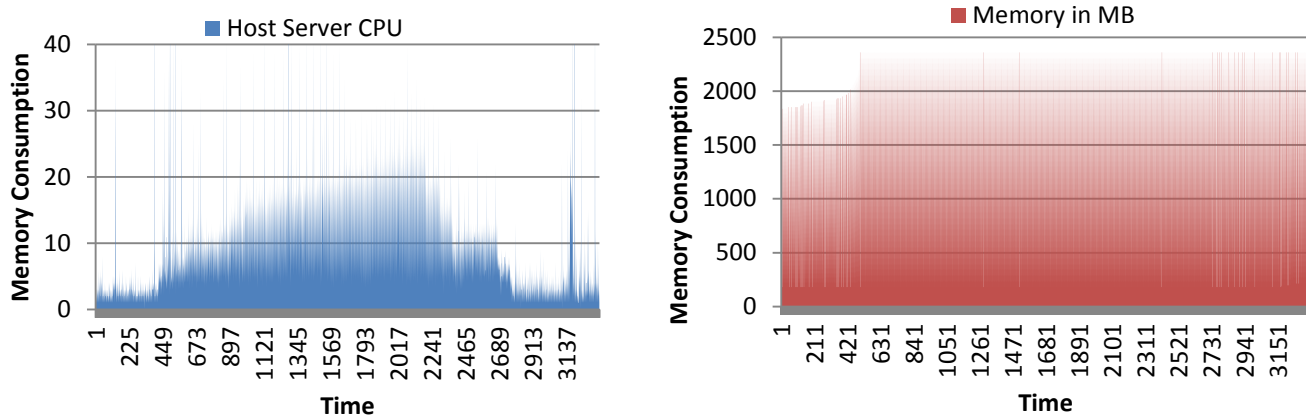


Figure 37: Combined Performance on Host Server. Experiment 2

6.6 Experiment 3

The final experiment continues with the same effort as Experiment 2, seeking to stress the system slightly more to observe the scalability. The total concurrent user count to be run in this final experiment was set to 10 users running the same workflow including a 15 minute duration test session.

The Tomcat activity resulted as expected based on the previous experiments. The system responded holding the same type of resource pattern with consistent growth in the same amount as with the previous user increase. The CPU usage remained at an average of 4.5% usage with a maximum peak of 12.7%.

The memory heap again followed the same consumption pattern seen in the previous experiments, with a slow increase rate for configuration tasks, a rapid increase in heap usage and decrease due to garbage collection during the monitoring phase, and finally the peak levels remaining close to 2 GB, well below the maximum value we defined at 6 GB. During a five to ten minute space of time the rate of memory consumption is noticeably faster, with no visible gap between rise and fall lines. The maximum memory before garbage collection level has now risen to 2.39 GB compared to the previous maximum of 2.0 GB. The max levels and garbage collection remain consistent, however, and give no cause for concern of memory leaks.

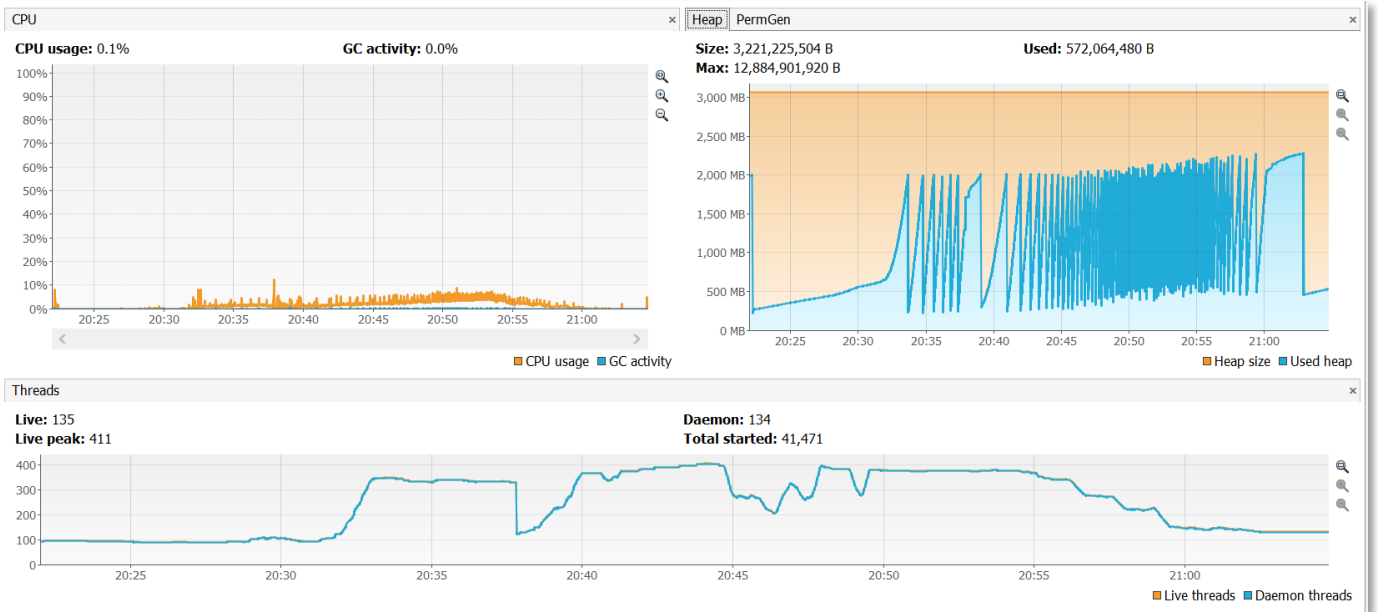


Figure 38: VisualVM-Tomcat CPU, Memory, and Threads charts. Experiment 3

Finally, the combined processes results from `top` revealed similar growth from the previous experiment. The average CPU percentage increased by nearly identical quantity as it did in the previous experiment, this time from 8.7% to 12% CPU activity. While this growth continued upward, the peak CPU levels remained steady and the graphs (Figure 37 and Figure 39) show a similar activity pattern.

The overall memory consumption remained consistent with only minor growth. The overall average memory utilization increased only from 1240 MB to 1371 MB. The maximum single amount of memory consumed increased to 2670 MB with an increase of nearly exactly 300 MB compared to a 400 MB growth in the previous experiment.

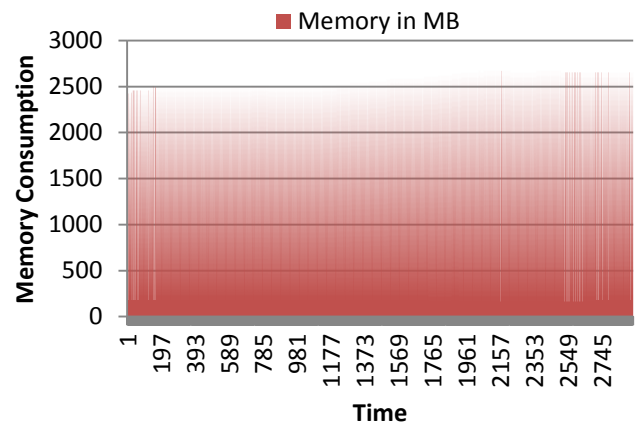
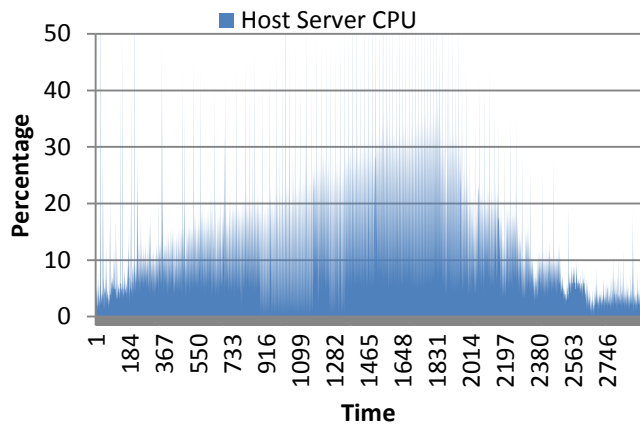


Figure 39: Combined Performance on Host Server. Experiment 3

Chapter 7

7. EVALUATION

The evaluation and discussion will be divided into two sections to be dealt with separately. The first section will summarize the results of the case study experiments and make some observations and analysis. The second section will be a discussion of the implementation process itself. More specifically, this section will identify implementation decisions which were successful, implementation decisions which proved to be problematic and could be improved, and lessons learned.

7.1 Case Study Analysis

The previous chapter provided initial summaries of the experiments and made some simple comparisons and analysis of the changes in resource consumption throughout the tests. This chapter will be a more in-depth analysis of the data received in order to make an accurate summation of the performance of both the Dashboard application alone, and the combined deployment scenario. The results will be analysed and some conclusions drawn, and finally some solutions to any problems will be given.

Table 1 summarizes the CPU and memory data gathered during the experiments. The data is distinguished by those entries derived from VisualVM (taking only Tomcat into account) and those entries gathered from the `top` output monitoring all three systems combined (web-app + master + slave).

Experiment / no. Users	Ave. CPU% - Tomcat	Ave. CPU% - Combined	CPU% Increase - Tomcat	CPU% Increase - Combined	Ave. Memory% - Combined	Ave. Memory MB - Combined	Ave. Memory Increase	Peak Memory - Combined	Peak Memory - Tomcat	Peak Memory Increase - Tomcat
1 / 1	1%	5.3%	-	-	5.9%	980MB	-	2100MB	1993MB	-
2 / 5	2.7%	8.7%	1.7%	3.4%	7.6%	1238MB	258MB	2359MB	2120MB	127MB
3 / 10	4.5%	12.2%	1.8%	3.5%	8.4%	1371MB	133MB	2670MB	2399MB	279MB

Table 1: Benchmarks from VisualVM: monitoring Tomcat, and top: monitoring cloud2

7.1.1 CPU

Observing both the figures in the previous chapter (displaying the CPU usage of Tomcat) and Table 1 ovan together shows that the processing power required to run a concurrent user count numbering in the dozens should not be a significant load on the system. Based on the three experiments conducted, a clearly linear growth rate occurred as is depicted in Figure 40 nedan. The assumption must be

then that a similar upward trend would continue as the concurrency level increases. If that would be the case, the current configuration could handle 140 concurrent users until the average CPU load grows over 50%, a threshold often recommended to not push beyond. Depending on the target level of course, this number is rather low when compared to one Vaadin study which claimed their server configuration should be able to handle 3000 users [46]. This disparity is likely due to the nature of and functional difference between the two applications, namely the real-time monitoring. Careful observation of the VisualVM CPU charts reveals that apart from the time periods where real-time monitoring is taking place, the normal CRUD type operations of the Dashboard application for even 10 users hardly registers on the chart, averaging around 0.4%. We estimate that at that rate, the current configuration would require only 8.3% CPU load to run the same 140 users. At a linear growth rate following that metric, our system would handle over 8000 concurrent users before reaching an average CPU load of 50%.

The second set of figures regarding CPU reveals the added load of running both master and slave nodes on the same physical server as the Dashboard application. Like the Tomcat specific results, the data revealed an almost perfectly linear growth rate, simply at a greater load on the host server due to the added master and slave processes (see Figure 40). If this linear growth pattern holds, the cloud2 server would operate at 50% CPU capacity at only 65 concurrent users, almost exactly half the amount when not taking MBPeT into consideration. Figure 41 shows the combined load of both metrics stacked together to show how many users can be handled at a given CPU load.

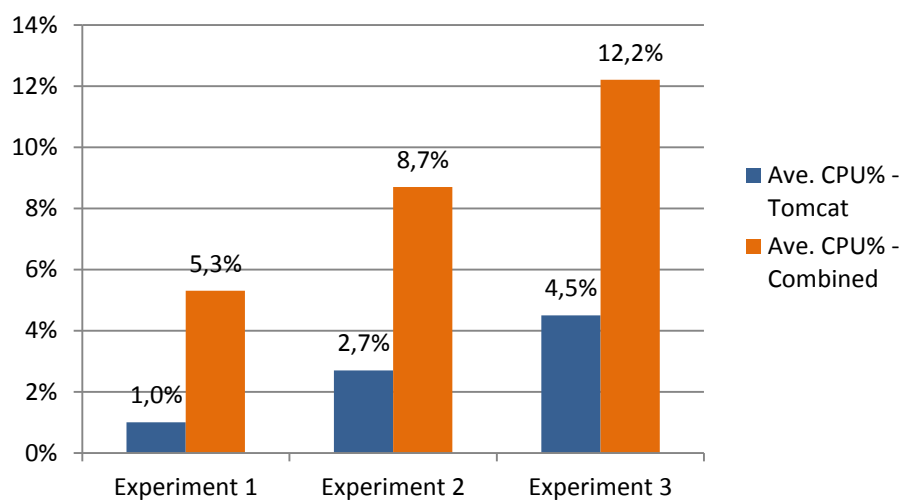


Figure 40: CPU Utilization by Experiment

CPU Utilization Growth Projection

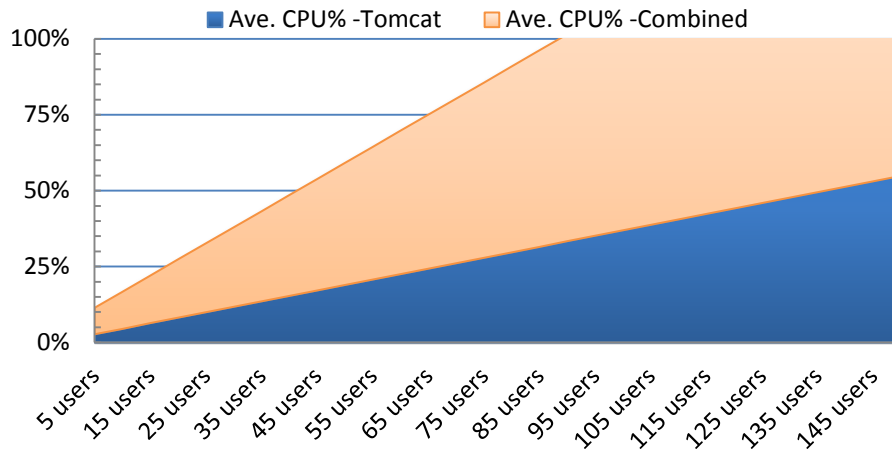


Figure 41: Projected CPU usage by web-app alone and combined load at given user count

7.1.2 Memory

We will again first consider the Dashboard results alone, and then consider the total memory of all three systems. The immediate significant load regarding memory is the heap size of Tomcat (see Figure 42). As with CPU, the average value is perhaps the more accurate indicator of how much memory is actually needed. However, because the heap size of Tomcat allows up to the maximum allotted capacity to be utilized, and the system can only allocate what memory it actually has at its disposal, the following calculations are based on the peak values gathered.

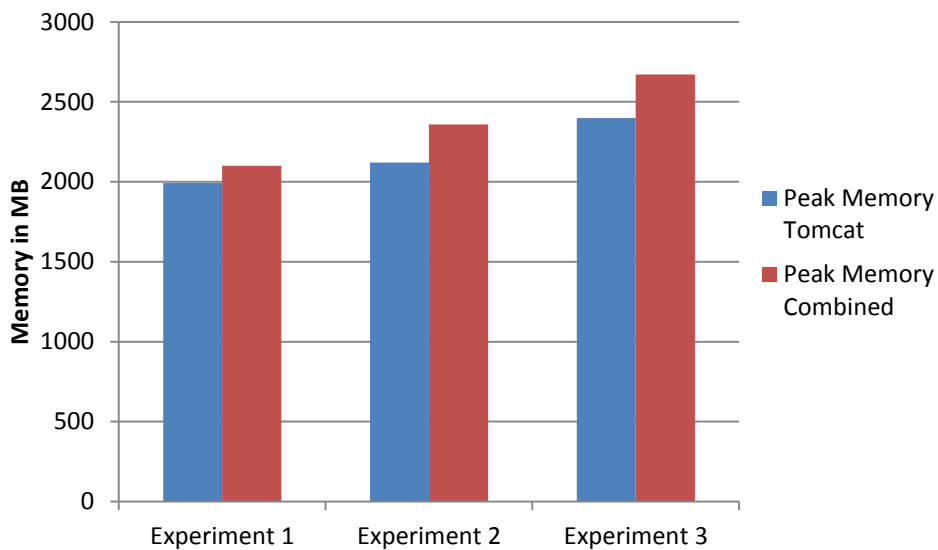


Figure 42: Memory Consumption by Experiment

The results of the three experiments we ran show the Tomcat memory hardly rising over 2 GB, which is well beneath the maximum allotted amount of 6 GB. This seemingly indicates that the user count could be enlarged considerably. However, if we draw out the existing growth rate across the three experiments we ran, the maximum heap size of 6 GB would be reached by 75 concurrent users, assuming they were all running simultaneous lengthy tests similar to our scenario. Keep in mind that the cloud2 machine has 16 GB of RAM at its disposal, so more memory could of course be allocated. In that scenario, by the time the system reached the 140 user mark, which is the maximum allowed by the CPU, the Tomcat peak level would be around 9.6 GB and the combined total peak memory would reach around 10.7 GB. With a total of 16 GB of system memory this is a reasonable assumption.

It is not guaranteed, however, that the memory usage would continue in a linear growth pattern. This is due to the nature of the garbage collector which allows available memory to be allocated and used by Tomcat within the bounds of limits set in our optimization procedure. Following the rules of the particular protocol enabled, the garbage collector will free up as much memory as it can in repeated cycles, as the previous VisualVM charts demonstrated. This implies that more users should be able to utilize the allotted memory capacity as long as the garbage collector and other memory optimizations are effective. Even though this is true by itself, it does not stand to reason that the garbage collector alone can support an infinite number of users. There are simply more factors than this alone, such as session size, length of sessions before time-out, etc.

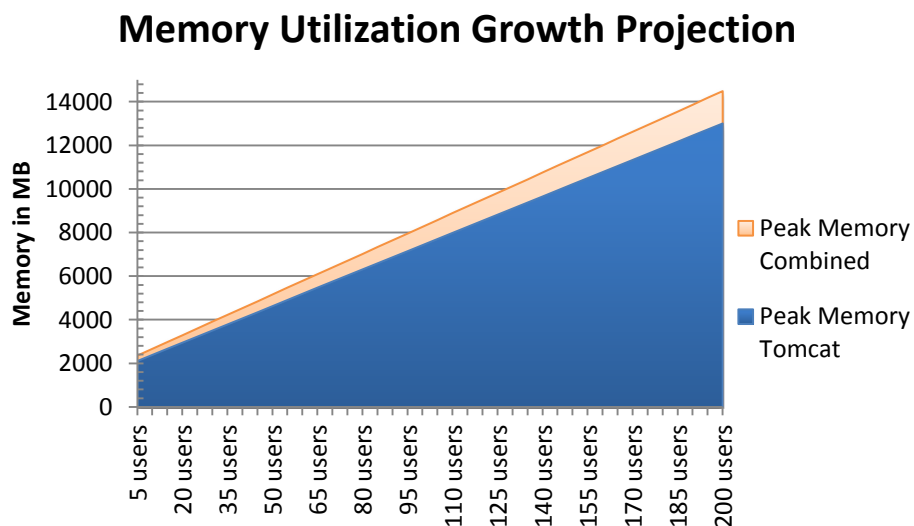


Figure 43: Project linear memory consumption by web-app alone and combined load at given user count

7.1.3 Optimizations

So far, we have proved that our system is functional and that it supports concurrent users. The data indicates that the scalability, however, is somewhat limited, depending of course on the desired target user count. Both the CPU and the memory indicate potential problem areas where bottlenecks will occur. A few solutions exist which could drastically increase the scalability of the Dashboard application. Since processor and memory are separate concerns, we will offer solutions for each situation.

CPU

The first task that likely requires the least amount of change would be to analyse in more depth where performance bottlenecks are arising from. This is possible still with tools like VisualVM as we have been using, but other tools also exist, like XRebel or JProfiler. Profiling can help locate exactly what parts of the application, or Tomcat itself, are using up the CPU. This can imply that the database or the ORM is putting more load on the CPU than the actual business logic. By locating the exact processor intensive areas, proper optimization and adjustments can be made. Another factor to check is the garbage collection strategy. Various garbage collection strategies can be tested in order to discover which alternative functions most optimally in a given configuration and system.

Finally, if still more room to scale up is required, not much else can be done before choosing a different deployment strategy. In our scenario, this presents a few different options. Without redeploying MBPeT to another server, clustering can be implemented using a load balancer and numerous application servers such as Tomcat or Wildfly. Although this is eventually the only alternative to ensure the ability to scale even further, given our current three-tiered deployment scenario, it would make more sense to start by redeploying MBPeT to another server rather than running all three systems on one machine. The first step, and easiest to implement given the current implementation, would be to deploy all slave nodes to a remote server. The easiest way to execute a slave with the current implementation would be to write a custom script which the Dashboard executes, in place of executing the slave directly. This script could then handle the remote initialization of slave nodes. When this solution proves not to provide enough room for growth, deploying the master node remotely could be the next step. For this to work, a new method of connecting to the master would need to be implemented, as the current solution relies on capturing the terminal output of the master which is only possible when residing on the same OS.

Memory

In order to optimize the memory requirements for scalability, the solutions vary in the amount of intrusiveness or expense. One simple place to start is the Vaadin session. A shorter session timeout or shorter keep-alive time can be used to allow sessions to time out faster. This would apply to idle sessions or those users who failed to log out. Given the nature of running lengthy test sessions with MBPeT, it might be that long session times are unavoidable in our use case. We have already optimized the widgetset, but still more can be done here. Even with loading only those widgets used in our application, it is still possible to apply lazy loading so that these components are loaded into the session only when navigated to the view in which they are held.

Lazy loading can be implemented on containers also, such as the JPAContainer utilized for this project. The Vaadin Directory contains additional lightweight containers which also support lazy loading. This tactic can prove especially useful because as the size of the database grows, and if all tables are loaded into containers in the business logic, this becomes an obvious bottleneck and accounts for a huge portion of the session size.

If still more memory is required, the last two options are to allocate more memory available to the system, or to employ clustering, as was mentioned regarding CPU above.

7.2 Implementation Lessons Learned

A few brief comments should be made in summarizing a number of specific topics discussed in this thesis and employed during the implementation of the project.

For the author of the development project, this was not the first project undertaken using the Vaadin framework. Past project work with Vaadin has left mainly positive experience, so much so that it was the obvious choice for what platform to use to realize this project. Building the UI with Vaadin is effective and powerful and quite intricate interfaces can be built in a short period of time.

Despite the fact that it markets itself as a UI framework, it would have been a great benefit to have a more powerful and efficient chart component as a part of the core component set. A few different 3rd party add-ons exist in the directory, but most are not nearly as polished for a production site as the rest of the core components are. Vaadin's official chart library is a rather expensive add-on. Although they are powerful and polished, the criteria for this project required only open source tools and non-commercial licenses.

One lesson learned too late in the process to change is that JPAContainer is not recommended to be used any longer. It is deceptive because there are more in-depth tutorials for JPAContainer than for many other components and add-ons. This is due to the fact that the intention had been to market this component as a commercial add-on to help fund the core development. This explains the larger number of tutorials and articles related to it. The article [50] describes in more detail the reasoning behind the intended use of JPAContainer, why it did not take off as intended and fundamental reasons why it is no longer recommended over other alternatives. Although JPAContainer has some helpful features, there are ultimately simpler solutions available now in either Vaadin's core or the directory. Many of these options are more efficient as well and are better suited to e.g. scalability concerns and memory overhead issues. For these reasons, in connection with the previous section analysing the scalability results of our experiments, we would recommend restructuring the container implementation decisions around another suitable alternative

We discussed in Section 5.4.2.3 Add-ons the switch made from `org.json` to `elemental.json` in the Vaadin core. In the scope of this project, this latter library was found to consist of a less robust API in comparison to the former JSON library. The result was that implementing the same functionality was more complicated and required slightly more lines of code to achieve the same results as with the original library. It appears almost as if the API is not yet fully developed and would require further development.

Non-Functional Requirements Evaluation

The performance analysis and functional requirements have been under primary focus during the case study and evaluation. However, a short evaluation should be given for the other previously mentioned non-functional requirements of the MBPeT Dashboard application (see Section 4.3 Non-Functional requirements).

Security

The security concerns created two requirements. The first introduced the ability to create a user account for the Dashboard, while the second required that access to all pages (Vaadin calls views) other than the login page be restricted to authenticated users only. Figure 16 in Section 5.4.2.2 Navigation and Views displays the LoginView and RegistrationWindow which fulfill this requirement. Only login credentials that match a registered user's username and coinciding password stored in the database are able to successfully authenticate and log in to the Dashboard application and access the main content.

Usability

The look-and-feel and usability concerns were validated by the stakeholders through an iterative process. This step-by-step process of obtaining feedback on

design and implementation decisions made possible the repeated improvement of the user interface layout and grouping of content, and ultimately yielded a solution approved by the stakeholders. Vaadin's API for building user interfaces makes it possible to quickly refactor UI designs. For example, many components have a certain set of interchangeable APIs, making it possible to seamlessly exchange one component type for another. Thanks to Vaadin's emphasis on enabling rapid development, design changes were possible to implement in relative short amounts of time.

Chapter 8

8. CONCLUSION

The premise of this thesis was the development of a rich internet application to realize a scalable web solution to the existing standalone MBPeT tool. The primary objective behind deploying MBPeT via a web application was to remove the necessity of local installation by making the service openly available via the internet, to achieve a scalable solution supporting multiple concurrent users from the single point of installation in the cloud. A secondary requirement was to implement a new and full-featured user interface beyond the capabilities of the existing command-line and GUI interface. This UI should be user-friendly, intuitive, and contain more advanced graphical capabilities compared to the existing solution such as editable charts and drag-and-drop PTA model building.

We first began by explaining the theory behind software testing as a whole, and further into the specific realm of performance testing through the *model-based* approach. Because the specific context of the MBPeT tool is related to web application testing, a further discussion of cloud services followed, as did a brief explanation of RIAs and the Vaadin framework. Vaadin is a Java web application development framework that is designed to enable rapid development of high quality rich internet applications.

Additionally, a concise description of the architecture and functional procedure of the MBPeT tool provided greater insight into the functionality and design of the tool. This insight exposed motivations behind design decisions made for the MBPeT Dashboard application we developed. The design and development process was documented with specific emphasis on the main contributions specific to this project. These primary contributions included the creation of an add-on component based on the JavaScript Flot charting library, the realisation of a web deployment solution for executing MBPeT test sessions remotely, the addition of new test report files in pdf format, and a wider array of user configurable, real-time monitoring of metrics such as user defined KPI values corresponding to the each action defined by the PTA models.

Three experiments were conducted to evaluate the fulfilment of functional requirements as well as the system's effective scalability. The former criterion was confirmed by the successful completion of all phases of the workflow of the evaluation tests. The MBPeT Dashboard application also successfully completed all experiments under the specified target loads prescribed by the supervisor. Although the experiments were completed successfully, we analysed the results and found that the current implementation has potential bottlenecks in regards

to CPU and memory consumption which are likely to limit the extent to which the system will scale in the number of concurrent users supported. We identified the causes of these limitations residing in either the design decisions of the Dashboard application and also the current combined deployment strategy in which all three relevant systems (web-app, master, and slave) are deployed on the same host server. We defined what optimizations have already been implemented to increase performance and scalability, and we proposed the implementation of additional tactics specific to optimizing both CPU and memory consumption. Due to the nature of high resource demand of real-time monitoring of lengthy test sessions, a higher load is to be expected as compared to simpler CRUD style applications. Although a few select changes are recommended to the Dashboard application solution (such as replacing the JPAContainers), the recommended next step alterations include additional standard optimization tactics, but more importantly migrating the deployment of MBPeT slave nodes and eventually master nodes to a remote server.

BIBLIOGRAPHY

- [1] W. Hetzel, Program Test Methods, Prentice Hall, 1973.
- [2] G. J. Myers, C. Sandler and T. Badgett, The Art of Software Testing, 3rd ed., Wiley, 1979.
- [3] E. W. Dijkstra, “Notes on Structured Programming,” April 1970. [Online]. Available: <https://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF>. [Accessed 4 February 2016].
- [4] Abbors, Fredrik; Åbo Akademi University; Faculty of Science and Engineering, “Model-Based Testing of Software Systems : Functionality and Performance,” *TUCS Dissertations*, no. 202, 2015.
- [5] P. Ammann and J. Offutt, Introduction to Software Testing, New York, NY.: Cambridge University Press, 2008.
- [6] F. Abbors, T. Ahmad, D. Truscan and I. Porres, “Model-Based Performance Testing of Web Services Using Probabilistic Timed Automata,” in *Proceedings of the 2013 10th International Conference on Web Information Systems and Technologies*, 2013.
- [7] T. Ahmad, F. Abbors, D. Truscan and I. Porres, “Model-Based Performance Testing Using the MBPeT Tool,” Turku Centre for Computer Science, Turku, 2013.
- [8] M. Utting and B. Legeard, Practical Model-Based Testing, San Francisco, CA: Elsevier Inc., 2007.
- [9] R. Pressman, Software Engineering. A Practitioner's Approach, New York: McGraw-Hill, 2010.
- [10] B. B. Boehm, Software Engineering Economics, Englewood Cliffs, NJ.: Prentice Hall, 1981.
- [11] H. v. Vliet, Software Engineering Principles and Practice, West Sussex, England: John Wiley & Sons, Ltd, 2008.
- [12] B. Erinle, Performance Testing with JMeter 2.9, Birmingham, UK: Packt Publishing, 2013.

- [13] M. Utting, A. Pretschner and B. Legeard, “A Taxonomy of Model-Based Testing,” *Software Testing, Verification & Reliability*, vol. 22, no. 5, pp. 297-312, 2012.
- [14] E. A. Marks and B. Loano, *Executive's Guide to Cloud Computing*, Hoboken, New Jersey: John Wiley & Sons, Inc., 2010.
- [15] BCS The Chartered Institute for IT, *Cloud Computing : Moving IT out of the Office*, British Informatics Society, 2012.
- [16] L. M. Vaquero, L. Rodero-Merino, J. Caceres and M. Lindner, “A Break in the Clouds: Towards a Cloud Definition,” *ACM SIGCOMM Computer Communication Review*, vol. 39, no. 1, 2009.
- [17] Oracle, “Oracle Testing As A Service,” [Online]. Available: <http://www.oracle.com/technetwork/oem/cloud-mgmt/ds-oracletesting-as-a-service-1905796.pdf>. [Accessed 15 February 2016].
- [18] S. Comai and G. T. Carughi, “A Behavioral Model for Rich Internet Applications,” *Lecture Notes in Computer Science*, vol. 4607, pp. 364 - 369, 2007.
- [19] M. Linaje, J. C. Preciado and F. Sánchez-Figueroa, “A Method for Model Based Design of Rich Internet Application Interactive User Interfaces,” in *Proceedings of the 7th International Conference on Web Engineering*, C0mo, Italy, 2007.
- [20] A. Bozzon, S. Comai, P. Fraternali and G. Toffetti Carughi, “Conceptual Modeling and Code Generation for Rich Internet Applications,” in *Proceedings of the 6th International Conference on Web Engineering*, New York, NY, 2006.
- [21] G. Toffetti, S. Comai, A. Bozzon and P. Fraternali, “Modeling Distributed Events in Data-Intensive Rich Internet Applications,” in *Proc. 7th Int. Conf. on Web Information Systems Engineering*, -, 2007.
- [22] N. Fränkel, *Learning Vaadin 7*, Packt Publishing: Birmingham, UK, 2014.
- [23] M. Grönroos, *Book of Vaadin*, 7 ed., Turku Finland: Vaadin Ltd., 2015.
- [24] Python Software Foundation, “Python,” Python Software Foundation, [Online]. Available: <http://www.python.org/>. [Accessed 8 February 2016].

- [25] G. Norman, D. Parker and J. Sproston, "Model Checking for Probabilistic Timed Automata," *Formal Methods in System Design*, vol. 43, no. 2, pp. 164-190, 2013.
- [26] M. Kwiatkowska, G. Norman, D. Parker and J. Sproston, "Performance Analysis of Probabilistic Timed Automata Using Digital Clocks. Formal Methods in System Design," 2006.
- [27] J. Shaw, "Web application performance testing - a case study of an on-line learning application," *BT Technology Journal*, vol. 18, no. 2, pp. 79-86, 2000.
- [28] graphviz, "The DOT Language," graphviz, [Online]. Available: <http://www.graphviz.org/doc/info/lang.html>. [Accessed 9 February 2016].
- [29] Graphviz, "Graphviz - Graph Visualization Software," Graphviz, [Online]. Available: <http://www.graphviz.org/>. [Accessed 9 February 2016].
- [30] Google, "Google Code Archive," Google, [Online]. Available: <https://code.google.com/archive/p/canviz/>. [Accessed 9 February 2016].
- [31] markandrewgoetz.com, "Google Code Archive," [Online]. Available: <https://code.google.com/archive/p/vizierfx/>. [Accessed 9 February 2016].
- [32] Python Software Foundation, "Python," Python Software Foundation, [Online]. Available: <https://pypi.python.org/pypi/pydot>. [Accessed 9 February 2016].
- [33] RadView, "Load Testing and Website Performance Testing Tools - RadView," RadView, [Online]. Available: <http://www.radview.com/about-webload/features/test-creation/>. [Accessed 5 February 2016].
- [34] The Apache Software Foundation., "Apache JMeter," [Online]. Available: <https://jmeter.apache.org/>. [Accessed 5 February 2016].
- [35] Hewlett Packard Enterprise, "HP LoadRunner," [Online]. Available: <http://www8.hp.com/us/en/software-solutions/loadrunner-load-testing/>. [Accessed 5 February 2016].
- [36] Hewlett Packard Enterprise, "httperf homepage," [Online]. Available: <http://www.labs.hp.com/research/linux/httperf/>. [Accessed 5 February 2016].
- [37] Vaadin Ltd., "Vaadin Directory," [Online]. Available: <https://vaadin.com/directory#!addon/aceeditor>. [Accessed 17 March 2016].

- [38] Vaadin Ltd., “Vaadin Directory,” [Online]. Available: <https://vaadin.com/directory#!addon/diagram-builder>. [Accessed 17 March 2016].
- [39] ALLOYUI, “AlloyUI Robust UI Tools,” [Online]. Available: <http://alloyui.com/tutorials/diagram-builder/>. [Accessed 17 March 2016].
- [40] IOLA and Ole Laursen, “Flot,” [Online]. Available: <http://www.flotcharts.org/>. [Accessed 7 April 2016].
- [41] Vaadin Ltd., “Vaadin wiki,” Vaadin Ltd., [Online]. Available: <https://vaadin.com/wiki/-/wiki/Main/Integrating+a+JavaScript+component>. [Accessed 7 April 2016].
- [42] J. Holan and K. Ondrej, Vaadin 7 Cookbook, Birmingham: PACKT Publishing Ltd., 2013.
- [43] Vaadin Ltd., “Vaadin Forum,” Vaadin Ltd., [Online]. Available: <https://vaadin.com/forum#!thread/296531>. [Accessed 7 April 2016].
- [44] Vaadin Ltd., “Vaadin - forum: Javascript component stopped working,” Vaadin Ltd., [Online]. Available: <https://vaadin.com/forum#!thread/11441026>. [Accessed 7 April 2016].
- [45] Vaadin Ltd., “Vaadin - Release Notes for Vaadin Framework 7.4.0,” Vaadin Ltd., [Online]. Available: <https://vaadin.com/download/release/7.4/7.4.0/release-notes.html>. [Accessed 7 April 2016].
- [46] Vaadin Ltd., “Vaadin - Scalable Web Applications Introduction,” Vaadin Ltd., [Online]. Available: https://vaadin.com/wiki?p_p_id=36&p_r_p_185834411_title=Scalable+web+applications&p_r_p_185834411_nodeName=vaadin.com+wiki. [Accessed 7 April 2016].
- [47] Vaadin Ltd., “Vaadin - Optimizing Hosting Setup,” Vaadin Ltd., [Online]. Available: <https://vaadin.com/blog/-/blogs/optimizing-hosting-setup>. [Accessed 7 April 2016].
- [48] “VisualVM,” Java.net, [Online]. Available: <https://visualvm.java.net/>. [Accessed 6 April 2016].
- [49] H. Muhammad, “HTOP,” [Online]. Available: <http://hisham.hm/htop/>.

[Accessed 22 April 2016].

- [50] Vaadin Ltd., “Vaadin blog - 3 pro tips for Vaadin developers,” Vaadin Ltd., [Online]. Available: <https://vaadin.com/web/matti/blog/-/blogs/3-pro-tips-for-vaadin-developers>. [Accessed 11 April 2016].