6 Performance Testing in the Cloud Using MBPeT

Fredrik Abbors, Tanwir Ahmad, Dragos Truscan, and Ivan Porres
Department of Information Technologies
Åbo Akademi University, Turku, Finland
Email: {fredrik.abbors, tanwir.ahmed, dragos.truscan, ivan.porres}@abo.fi

Abstract–We present a model-based performance testing approach using the MBPeT tool. We use of probabilistic timed automata to model the user profiles and to generate synthetic workload. The MBPeT generates the load in a distributed fashion and applies it in real-time to the system under test, while measuring several key performance indicators, such as response time, throughput, error rate, etc. At the end of the test session, a detailed test report is provided. MBPeT has a distributed architecture and supports load generation distributed over multiple machines. New generation nodes are allocated dynamically during load generation. In this book chapter, we will present the MBPeT tool, its architecture, and demonstrate its applicability with a set of experiments on a case study. We also show that using abstract models for describing the user profiles allows us quickly experiment different load mixes and detect worst case scenarios.

Keywords-Performance testing, model-based testing, MBPeT, cloud.

6.1 Introduction

Software testing is the process of identifying incorrect behavior of a system, also known as revealing defects. Uncovering these defects, typically, consists of running a batch of software tests (*test suite*) against the software itself. In some sense, a second software artefact is built to test the primary one. This is normally referred to as *functional testing*. A software test compares the actual output of the system with the expected output for a particular

known input. If the actual output is the same as the expected output the test passes, otherwise a test fails and a defect is found. Software testing is also the means to assess the quality of a software product. The fewer the defects found during testing, the higher the quality is of that software product. However, not all software defects are related to functionality. Some systems may stop functioning or may prevent other users to access the system simple because the system is under a heavy workload with which it cannot cope. Performance testing is the means of detecting such errors.

Performance testing is the process of determining how a software system performs in terms of responsiveness and stability under a particular workload. The purpose of the workload is that it should match the expected workload (the load that normal users put on the system when using it) as closely as possible. This can be achieved by running a series of tests in parallel, but instead of focusing on the right output the focus is shifted towards measuring non-functional aspects, i.e. the time between input and output (response time) or number of requests processed in a second (throughput).

Traditionally, performance testing has been conducted by running a number of predefined scenarios (or scripts) in parallel. One drawback to this approach is that real users do not behave as static scripts. This can also lead to certain paths in the system being left untested or that certain caching mechanisms in the system kick in due the repetitiveness of the test scripts.

Software testing can be extremely time consuming and costly. In 2005, Caper Jones - chief scientist of Software Productivity Research in Massachusetts - estimated that as much as 60 percent of the software work in the United States was related to detecting and fixing defects [1]. Another drawback is that software testing, as well as performance testing, involves tedious manual work when creating test cases. A software system typically undergoes a lot of changes during its lifetime. Whenever a piece of code is changed, a test has to be updated or created to show that the change did not break any existing functionality or introduce any new defects. This adds more time and cost to testing. In the case of performance testing this implies that one has to be able to benchmark quickly and effectively to check if the performance of the system is affected by the change of the code.

Research effort have be put into solving this dilemma. One of the most promising techniques is Model-Based Testing (MBT). In MBT, the central artefact is a system model. The idea is that the model represents the behavior or the use of the system. Tests are then automatically generated form the model. In MBT the focus has shifted from manually creating tests to maintaining a model that represents the behavior of the system. Due to the fact that tests are automatically generated from a model, MBT copes better with changing requirements and code than traditional testing. Research has

shown that MBT could reduce the total testing costs with 15 percent [8]. MBT has mostly been targeted towards functional testing, however, there exist a few tools that utilizes the power of MBT in the domain of performance testing. In our research we make use of the advantages of MBT in our performance testing approach.

MBPeT is a Python-based tool for performance testing. Load is generated from *probabilistic timed automata* (PTA) models describing the behavior of groups of virtual users. The models are then executed in parallel to get a semi-random workload mix. The abstract PTA models are easy to create and update, facilitating quick iteration cycles. During the load generation phase, the tool also monitors different *key performance indicators* (KPIs) such as response times, throughput, memory, CPU, disk, etc. The MBPeT tool has a distributed architecture where one master node controls several slave node or load generator. This facilitates deployment to a cloud environment. Besides monitoring, the tool also produces a performance test report at the end of the test. The report contains information about the monitored KPIs, such as response times, throughput etc, but also graphs showing how CPU, memory, disk, network utilization varied during a performance test session.

The rest of the report is structured as follows: we briefly enumerate several related works in the following section. Then, is Section 6.3, we briefly describe the load generation process. In Section 6.4, we give an overview of the architecture of the tool. In Section 6.5, we describe how the workload models are created and discuss the probabilistic timed automata formalism. In Section 6.6, we discuss the performance testing process in more detail. In Section 6.7, we present a auction web service case study and a series of experiments using our tool. Finally, in Section 6.8 we present our conclusions and discuss future work.

6.2 Related Work

There exist a plethora of commercial performance testing tools. In the following, we briefly enumerate couple of popular performance testing tools. FABAN is an open source framework for developing and running multi-tier server benchmarks [18]. FABAN has a distributed architecture meaning load can be generated from multiple machines. The tool has three main components: *A harness* - for automating the process of a benchmark run and providing a container for the benchmark driver code, a *Driver framework* - provides an API for people to develop load drivers, and an *Analysis tool* - to provide comprehensive analysis of the data gathers for a test. Load is generated by running multiple scripts in parallel. JMeter [19] is an open source

Java tool for load testing and measuring performance, with the focus on web applications. Jmeter can be set up in a distributed fashion and load is generated from manually created scenarios that are run in parallel. Httperf [6] is a tool for measuring the performance of web servers. Its aim is to facilitate the construction of both micro and macro-level benchmarks. Httperf can be set up to run on multiple machines and load is generated from pre-defined scripts. LoadRunner [7] is a performance testing tool from Hewlett-Packard for examining system behavior and performance. The tool can be run in a distributed fashion and load is generated from pre-recorded scenarios.

Recently several authors have focused on using models for performance analysis and estimation, as well as for load generation. Barna et al., [2] present a model-based testing approach to test the performance of a transactional system. The authors make use of an iterative approach to find the workload stress vectors of a system. An adaptive framework will then drive the system along these stress vectors until a performance stress goal is reached. They use a system model, represented as a two-layered queuing network, and they use analytical techniques to find a workload mix that will saturate a specific system resource. Their approach differs from ours in the sense that they use a model of the system instead of testing against a real implementation of a system.

Other related approaches can be found in [16] and [15]. In the former, the authors have focused on generating valid traces or a synthetic workload for inter-dependent requests typically found in sessions when using web applications. They describe an application model that captures the dependencies for such systems by using Extended Finite State Machines (EFSMs). Combined with a workload model that describes session inter-arrival rates and parameter distributions, their tool *SWAT* outputs valid session traces that are executed using a modified version of *httperf* [12]. The main use of the tool is to perform a sensitivity analysis on the system when different parameters in the workload are changed, e.g., session length, distribution, think time, etc. In the latter, the authors suggest a tool that generates representative user behavior traces from a set of Customer Behavior Model Graphs (CBMG). The CBMG are obtained from execution logs of the system and they use a modified version of the *httperf* utility to generate the traffic from their traces. The methods differ from our approach in the sense they both focus on the trace generation and let other tools take care of generating the load/traffic for the system, while we do on-the-fly load generation from our models.

Denaro [4] proposes an approach for early performance testing of distributed software when the software is built using middleware components technologies, such as J2EE or CORBA. Most of the overall performance of

such a system is determined by the use and configuration of the middleware (e.g. databases). They also note that the coupling between the middleware and the application architecture determines the actual performance. Based on architectural designs of an application the authors can derive application-specific performance tests that can be executed on the early available middleware platform that is used to build the application with. This approach differs from ours in that the authors mainly target distributed systems and testing of the performance of middleware components.

6.3 The Performance Testing Process

In this section we are briefly going to describe the steps of the performance testing process. A more detailed description is given in Section 6.6.

6.3.1 Model Creation

Before we start generation load for the system we first have to create a load profile or a load model that describe the behavior of the users. Since we can not have a model for each individual user we have to create one or several models that represent the behavior for a larger group of users. These models describe how a groups of virtual users (VUs) behave and they are simplified models of how a real users would behave. Section 6.5 gives more details of how the models are constructed. Essentially, we use probabilistic timed automata (PTA) to specify user behavior which describe in an abstract way the sequence of actions a VU can execute against the system and their probabilistic distribution.

6.3.2 Model Validation

Once the models have been created they are checked for consistency and correctness. For instance, we check that the models have a start and end point, that there are no syntactical errors in the models, and that the probabilities and actions have been defined correctly. Once the models have been checked by the MBPeT tool we start generating load for the system under test (SUT).

6.3.3 Test Setup

Before we can actually start generating load we need to set up everything correctly so that the MBPeT can connect to the SUT and generate the appropriate amount of load. To do that one have to fill in a settings file. This file contains e.g., the IP-address of the SUT, what load models to use, how

many parallel virtual users to simulate, ramp up period, and the duration of the performance test. The MBPeT tool needs this information in order to be able to generate the right amount of load.

The tester also needs to implement an adapter for the tool. Every SUT will have its own adapter implementation. The purpose of the adapter is to translate the abstract actions found in the model into concrete actions understandable by the SUT. In case of a web page, a *browse* action would need to be translated into a HTTP *GET* request.

6.3.4 Load Generation

Once everything is set up, load generation begins. The MBPeT tool generates load from the models by starting a new process for every simulated user. Inside that process load is generated by executing the PTA model. For more details please see Section 6.6.2. Please see Section 6.5.2 for more information on PTAs.

6.3.5 Monitoring

During the load testing phase the MBPeT tool monitors the traffic sent on the network to the SUT. The tool monitors the throughput and response time for every action sent to the system. If there is a possibility to connect to the SUT remotely, the MBPeT tool can also monitor the utilization of the CPU, memory, network, disk, etc. This information can be very useful when trying to identify potential bottlenecks in the system. Once the test run is complete and all information is gathered, the tool will create a test report.

6.3.6 Test Reporting

The test report contains information about the parameters monitored during the performance test. It gives statistical values of the mean and max response time for individual actions and displays graphs that show how the repones time varied over time when the load increases. If the tool can be connected remotely to the SUT, the test report will also show how the CPU, memory, and disk was utilized over time when the load was applied to the SUT. Both of these sources of information can be helpful when trying to pin the a potential bottleneck in the system.

6.4 MBPeT Tool Architecture

MBPeT has a distributed architecture. It consists of two types of nodes: a master node and slave nodes. A single master node is responsible of initiating and controlling multiple remote slave nodes, as shown in Figure 6.1. Slave nodes are designed to be identical and generic, in a sense that they do not have prior knowledge of the SUT, its interfaces, or the workload models. That is why for each test session, the master gathers and parses all the required information regarding the SUT and the configuration for each test session and sends that information to all the slave nodes. Once all slaves have been initialized, the master begins the load generation process by starting a single slave while rest of the slaves are idling.

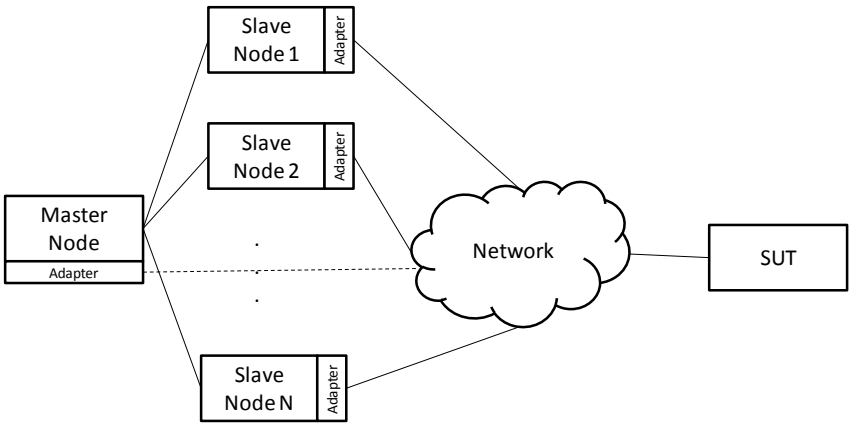

Figure 6.1: Distributed architecture of MBPeT tool

6.4.1 The Master Node

The internal architecture of the master node is shown in Figure 6.2. It contains the following components:

Core Module

The core module of the master node controls the activities of other modules as well as the flow of information among them. It initiates the different modules when their services are required. The core module takes as input the following information and distributes it among all the slave nodes:

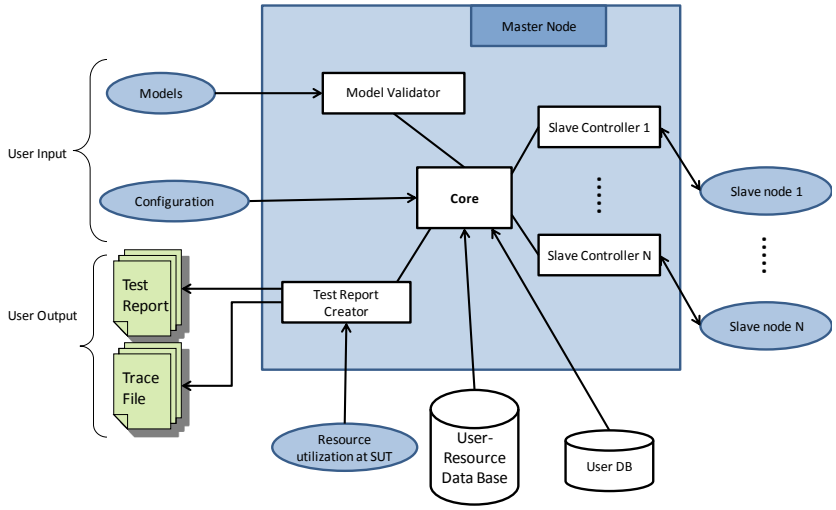

Figure 6.2: Master Node

1. *User Models:* PTA models are employed to mimic the dynamic behavior of the users. Each case-study can have multiple models to represent different types of users. User models are expressed in DOT language [5].

2. *Test Configuration:* It is a collection of different parameters, that are defined in a *Settings* file, which is a case-study specific. A *Settings* file specifies the necessary information about the case-study and this information is later used by the tool to run the experiment. There are some mandatory parameters in the *Settings* file, which have been listed below with the brief description. These parameters can also be provided as command-line arguments to the master node.

   (a) *Test duration*: It defines the duration of a test session in seconds.

   (b) *Number of users*: It specifies the maximum number of concurrent users for a test session.

   (c) *Ramp*: The ramp period is specified for all types of users. It can be defined in two ways. One way is to specify it as a percentage

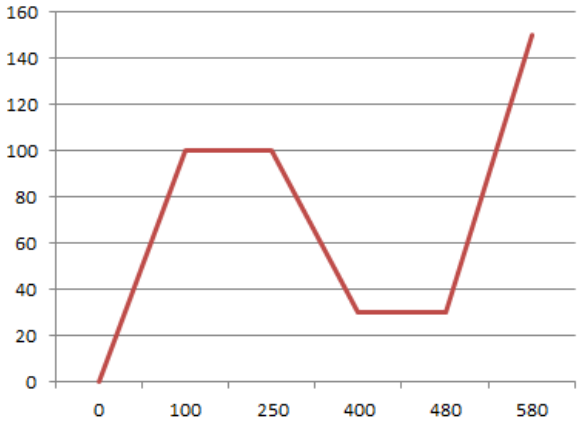

Figure 6.3: Example ramp function

of the total test duration. For example, if the objective of the experiment is to achieve the given number of concurrent users within the 80% of total test duration, then the ramp value would be equal to 0.8. Then, the tool would increase the number of users at a constant rate, in order to achieve the given number of concurrent users within the ramp period.

The ramp period can also be defined as an array of tuples. For instance the ramp function depicted in Figure 6.3, as illustrated in the Listing 6.1. A pair value is referred to as a *milestone*. The first integer in a milestone describes the time duration in seconds since the experiment started and the second integer states the target number of concurrent users at that moment. For example, the fourth milestone in the Listing 6.1, that is (400, 30), indicates that at 400 seconds the number of concurrent users should be 400, and thus starting from the previous milestone (100, 30) the number of concurrent users should drop linearly in the interval 250-400 seconds. Further, a ramp period may consist of several milestones depending upon the experiment design. The benefit of defining the ramp period in this way is that the number of concurrent users could increase and decrease during the test session.

Listing 6.1: Ramp section of Settings file

```
#================ Ramp Period ================
ramp_list = [(0, 0), (100, 100), (250, 100),
(400, 30),(480, 30), (580, 150), ... ]
```

(d) *Monitoring interval*: It specifies how often a slave node should check and report its own local resource utilization level for saturation.

(e) *Resource utilization threshold*: It is a percentage value which defines the upper limit of local resource load at the slave node. A slave node is considered to be saturated if the limit is exceeded.

(f) *Models folder*: A path to a folder which contains all the user models.

(g) *Test report folder*: The tool will save the test report at this given path.

In addition to mandatory parameters, the *Settings* file can contain other parameters, which are related to a particular case-study only. For example, if a SUT is a web server then the IP address of the web server would be an additional parameter in the *Settings* file.

3. *Adapter:* This is a case-study specific module which is used to communicate with SUT. This module translates each action interpreted from the PTA model into a form that is understandable by the SUT, for instance a HTTP request. It also parses the response from the SUT and measures the response time.

4. *Number of Slaves:* This number tells the master node how many slave nodes that are participating in the test session.

Two test databases are used by MBPeT: a user database and a user resource database. The user database contains all the information regarding users such as usernames, passwords or name spaces. In certain cases, the current state of the SUT must be captured, in order to be able to address at load generation time data dependencies between successive requests. As such, the user resource database is used to store references to the resources (e.g. files) available on the SUT for different users. The core module of the master node uses an instance of the test adapter to query the SUT and save that data in the user resource database.

Further, the core module remotely controls the Dstat[1] tool on SUT via SSH protocol. Dstat is a tool that provides detailed information about the system resource utilization in real-time. It logs the system resources utilization information after every specific time interval, one second by default. The delay between each update is specified in the command along with the names of resources to be monitored. This tool creates a log file in which it appends

[1] http://dag.wieers.com/home-made/dstat/

a row of information for each resource column after every update. The log file generated by the Dstat tool is used as basis for generating the test report, including graphs on how SUT's KPIs vary during the test session.

Model Validation Module

The *Model Validator* module validates the load models. It performs different numbers of syntactic checks on all models and generates a report.This report gives error descriptions and the location in model where the error occurred. A model with syntax anomalies could lead to inconclusive results. Therefore it is important to ensure that the all given models are well-formed and no syntax mistakes have been made in implementing the models. Examples of couple of validation rules are:

- Each model should have an initial and a final state
- All transitions have either probabilities or actions
- The sum of probabilities of transitions originating from a location is 1.
- All locations are statically reachable

Slave Controller Module

For each slave node there is an instance of *SlaveController* module in the master node. The purpose of the SlaveController module is to act as a bridge between slave nodes and the core master process and to control the slave nodes until the end of the test. The benefit of this architecture is to keep the master core process light and active, and more scalable. The SlaveController communicates with master core process only in few special cases, so that the core process could perform other tasks instead of communicating with slave nodes. Moreover, it also increases the parallelism in our architecture, all the SlaveControllers and the master's core processes could execute in parallel on different processor cores. Owning to the efficient usage of available resources, the master can perform more tasks in less period of time. A similar approach has been employed at the slave node, where each user is simulated as an independent process for the performance gain.

Test Report Creation Module

This module performs two tasks: Data Aggregation and Report Creation. In the first task, it combines the test results data from all slaves into an internal representation. Further, it retrieves the log file generated by the Dstat tool

from the SUT via Secure File Transfer Protocol (SFTP). The second task of this module is to calculate different statistical indicators and render a test report based on the aggregated data.

6.4.2 The Slave Node

Slave nodes are started with one argument, the IP-address of the master node. The *Core* module opens the socket and connects to the master node at the given IP-address with the default port number. After connecting with the master node successfully, it invokes the Load Initiator module.

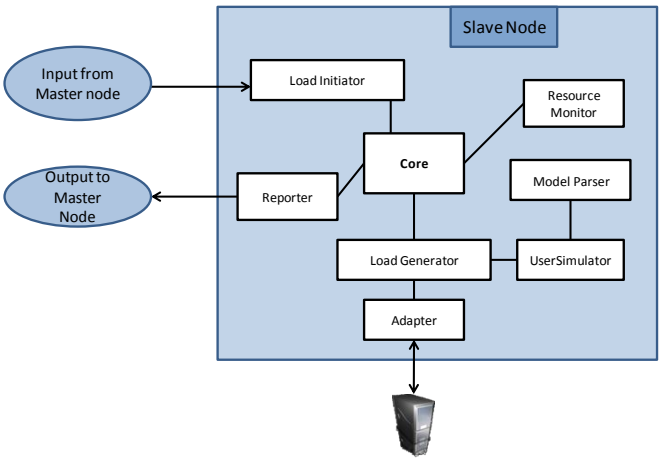

Figure 6.4: Slave Node

Load Initiation Module

The *Load Initiator* module is responsible for initializing the test setup at the slave node as well as storing the case-study and model files in a proper directory structure. It receives all the information from the master node at initialization time.

Model Parser Module

The *Model Parser* module reads the PTA model into an internal structure. It is a helper module that facilitates the UserSimulator module to perform different operations on the PTA model.

Load Generation Module

The purpose of this module is to generate load for the SUT at the desired rate, by creating and maintaining the desired number of concurrent virtual users. It uses the *UserSimulator* module to simulate virtual users where each instance of UserSimulator presents a separate user with unique user ID and session. The UserSimultor utilizes the Model Parser module to get the user's action from the user model and uses the Adapter module to perform the action. Then it waits for a specified period of time (i.e. the user think time) before performing the next action, which is chosen based on the probabilistic distribution.

Resource Monitoring Module

The *Resource Monitor* module runs as a separate thread and wakes up regularly after a specified time period. It performs two tasks every time it wakes up: 1) checks the local resource utilization level and saves the readings, 2) calculates the average of resource utilizations over a certain number of previous consecutive readings. The value obtained from the second task is compared with resource utilization threshold value, defined in the test configuration. If the calculated average is above a set threshold value of 80 percent, then it means that the slave node is about to saturate and the master will be notified. When a slave is getting saturated, its current number of generated users is kept constant, and additional slaves will be delegated to generate the more load.

Reporter Module

All the data that has been gathered during the load generation is dumped into files. The Load Generator creates a separate data file for each user; it means that the total number of simulation data files would be equal to the total number of concurrent users. In order to reduce the communication delay, all these data files are packed into a zip file, and sent to the master at the end of the test session.

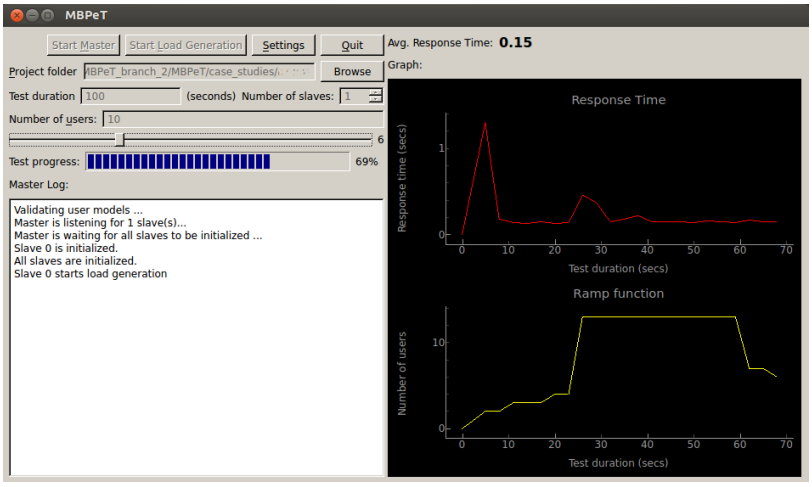

Figure 6.5: Main window of the GUI

6.4.3 Graphical User Interface

The MBPeT tool can be run both in command line and via a graphical user interface (GUI) as shown in Figure 6.5. Feature-wise the GUI is almost identical to the command-line version except for two features:

- The GUI implements the number of users as a slider function. This implies that the number of parallel user can be increased and decreased in real time using the slider, as an alternative to predefining a ramp function at beginning of the test session;
- The average response observed by all slave nodes is plotted in real-time. The response time graphs can be configured to display either one average response time plot for all actions (as currently depicted in Figure 6.5) or one average response time plot for each individual action type.

Additionally, from the GUI, one can specify basically all the test session settings previously described in Section 6.4.1

6.5 Model Creation

In this section we will introduce the load models used for generating load and describe how they are constructed. We will also in theory describe how

load is generated from these models.

6.5.1 Workload Characterization

Traditionally, performance analysis starts first with identifying key performance scenarios, based on the idea that certain scenarios are more frequent than others or certain scenarios impact more on the performance of the system than other scenarios. A performance scenario is a sequence of actions performed by an identified group of users [13]. In some cases, key performance scenarios can consist of only one action, for example "browse", in the case of a web-based system. In the case of Amazon online store, examples of key performance scenarios could be: searching for a product, then adding one or more products into the shopping cart and finally pay for them. In the first example, only one action is sent to the system, namely "browse". In the second example, several actions would have to be sent to the server, e.g. "login", "search","add-to-cart","checkout", etc.

In order to build the workload model, we start by looking and analyzing the requirements and the system specifications, respectively. During this phase we try to get an understanding of how the system is used, what are the different types of users, and what are the key performance scenarios that will impact most on the performance of the system. A user type is characterized by the distribution and the types of actions if performs.

The main sources of information for workload characterization are: Service Level Agreements (SLAs), system specifications and standards, and server execution logs [11]. By studying these sources we identify the inputs of the system with respect to types of transactions (actions), transferred files, file sizes, arrival rates, etc. following the generic guidelines discussed in [3]. In addition, we extract information regarding the KPIs, such as the number of concurrent users the system should support, expected throughput, response times, expected resource utilization demands etc. for different actions under a given load.

We use the following steps in analyzing the workload:

1. Identify the actions that can be executed against the system.
   (a) Analyze what are the required input data and output data for each action. For instance, what is the request type, its parameters, etc.
   (b) Identify dependencies between actions. For example, a user can not execute a logout action before a login action.
2. Identify what classes (types) of users execute each action

3. Identify the most relevant user types.

4. Define the distribution of actions that is performed by each user type.

5. Define an average *think time* per action for each user type.

Table 6.1 shows an example of a user type specification, its actions, action dependencies, and think time ordered in a tabular format. Based on this information we build a *workload model* described as a *probabilistic timed automata* or *PTA*.

Action	Dependency	User Type 1		User Type 2	
		Think time	Frequency	Think Time	Frequency
a_1		t_1	f_1	t_2	f_2
a_2	a_1	t_3	f_3		
a_3	a_1			t_4	f_4
a_4	a_2	t_5	f_5		
a_5	a_4	t_6	f_6	t_7	f_7
a_6	a_3			t_8	f_8

Table 6.1: Example of user types and their actions

6.5.2 Workload Modeling Using PTA

The results of the workload characterization are aggregated in a workload model similar to the one in Figure 6.6, which mimics the real workload under study. One such workload model is created for each identified user type. Basically, the model will depict the sequence of actions a user type can perform and their arrival rate, as a combination of the probability that an action is executed and the think time of the user for that action. In addition, we also identify the user types and their probabilistic distribution. A concrete example will be given in Section 6.7.

All the information that is extracted from the previous phase is aggregated in a workload model which is describes as a probabilistic timed automaton (PTA). A PTA is similar to a state machine in the sense that a PTA consists of a set of locations connected with each other via a set of transitions. However, a PTA also include the notion of time and probabilities. Time is modeled as an invariant clock constraint on transitions and increase at the same rate as real time.

A *probabilistic timed automaton* (PTA) is defined [9] as T = (L,C,*inv*, *Act*, E, δ) where:

- *a set of locations L;*

- *a finite set of clocks C;*
- *an* invariant condition *inv :* $L \rightarrow Z$;
- *a finite set of* actions *Act;*
- *an* action enabledness function $E : L \times Act \rightarrow Z$;
- *a* transition probability function $\delta : (L \times Act) \rightarrow D(2^C \times L)$.

In the above definitions, Z is a set of clock zones. A clock zone is a set of clock values, which is a union of a set of clock regions. Δ is a probabilistic transition function. Informally, the behavior of a probabilistic timed automaton is as follows: In a certain location l, an action a can be chosen when a clock variable reaches its value with a certain probability if the action is enabled in that location l. If the action a is chosen, then the probability of moving to a new location l' is given by δ[l,a](C',l'), where C' is a particular set of clocks to be reset upon firing of the transitions. Figure 6.6 gives an example of a probabilistic timed automata.

The syntax of the automata is as follows: Every transition has an initial location and an end location. Each location is transitively connected from the initial location. The transitions can be labeled with three different values: a probability value, an action, and a clock. The *probability* indicates the chance of that transition being taken. The *action* describes what action to take when the transition is used, and the *clock* indicates how long to wait before firing the transition. Every automaton has an *end location*, depicted with a double circle, that will eventually be reached. It is possible to specify loops in the automaton. It is important to notice that the sum of the probabilities on all outgoing transitions from a given location must be equal to 1. For example, consider location *2* in Figure 6.6: for the PTA to be complete the following must apply: p1 + p2 + p3 = p4 + p5 = 1.

6.6 Performance Testing Process

In this section we describe the performance testing process. Figure 6.7 shows the three steps involved in the process. In the following, we will discuss the three steps in more detail.

6.6.1 Test Setup

Every test run starts with a test setup. In each test setup, there is one master node that carries out the entire test session and generates a report. The

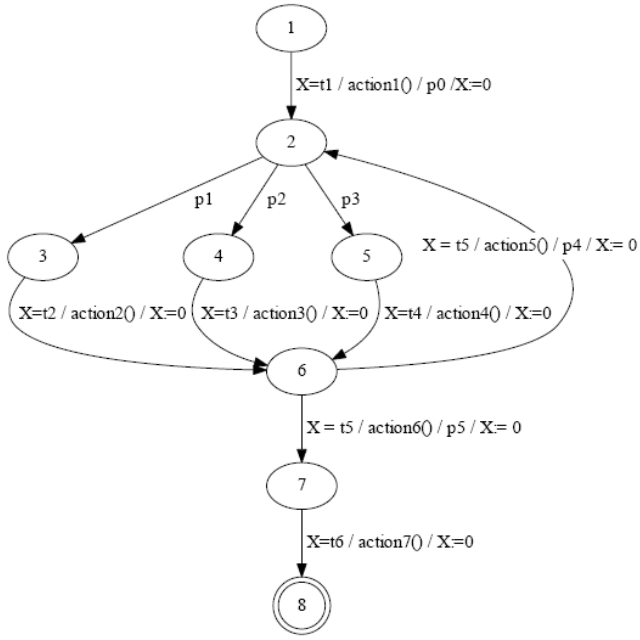

Figure 6.6: Example of a probabilistic timed automaton.

user only interacts with the master node by initializing it with the required parameters (mentioned in the Section 6.4.1) and getting the test report at the end of the test run. The parameter given to the master is the project folder. This folder contains all the files needed for load generation, such as the adapter code, the settings file (if command line mode is used) and other user specific files.

The adapter file and the settings file are the most important. The adapter files explains how the abstract actions found in the load models are translated to concrete actions. The settings file contain information about the test session, such as the location of the load models, IP-address to the SUT, the ramp function, test duration, etc. The same information can also be set from the GUI via the *Settings* button, see Figure 6.8. In here, the user is required to enter the same information as given in the settings file. Additionally, the path to the adapter file and the load models have to be given.

As one may notice in Figure 6.8, the user has the option of defining an average *think time* for the models and its *standard deviation*. If these options are used, the individual think time specified in the models for each action

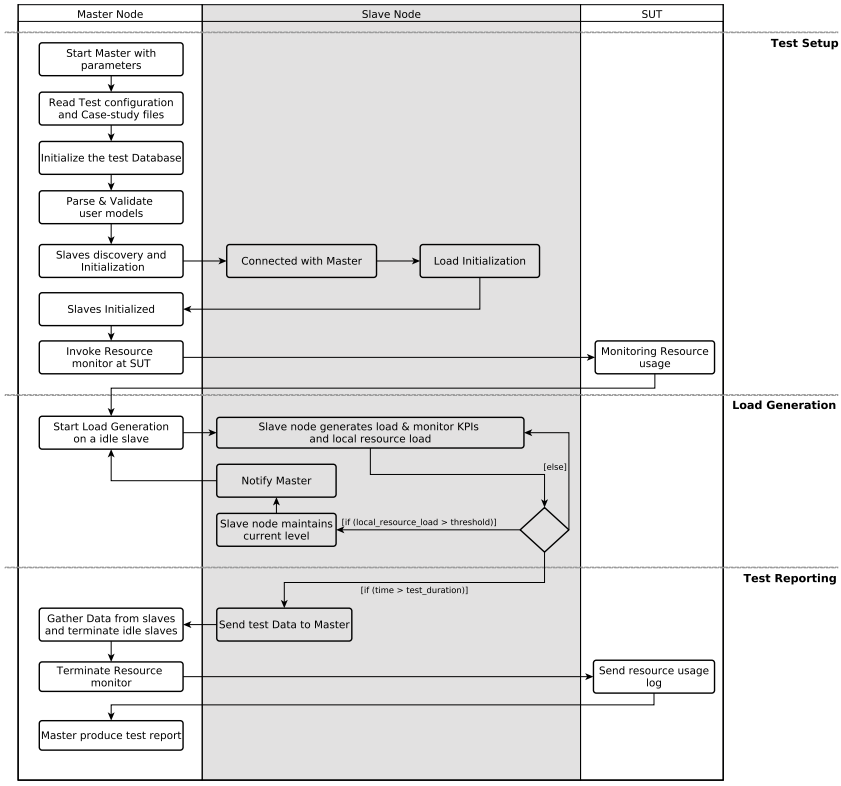

Figure 6.7: MBPeT tool activity diagram

will be ignored and the one specified in the GUI will be used.

Once the required information has been given, the master node sets up the test environment. After that, it invokes the *Model Validator*. This module validates the syntax of user models. If the validation fails, it gives the user a choice whether the user wants to continue or not to load generation. If the user decides to continue or the validation was successful, then the master enters into the next phase.

6.6.2 Load Generation

Load is generated for the models based on the same principles as described in section 6.5.2. The load generation is based on a deterministic choice with a probabilistic policy. This introduces certain randomness into the test process

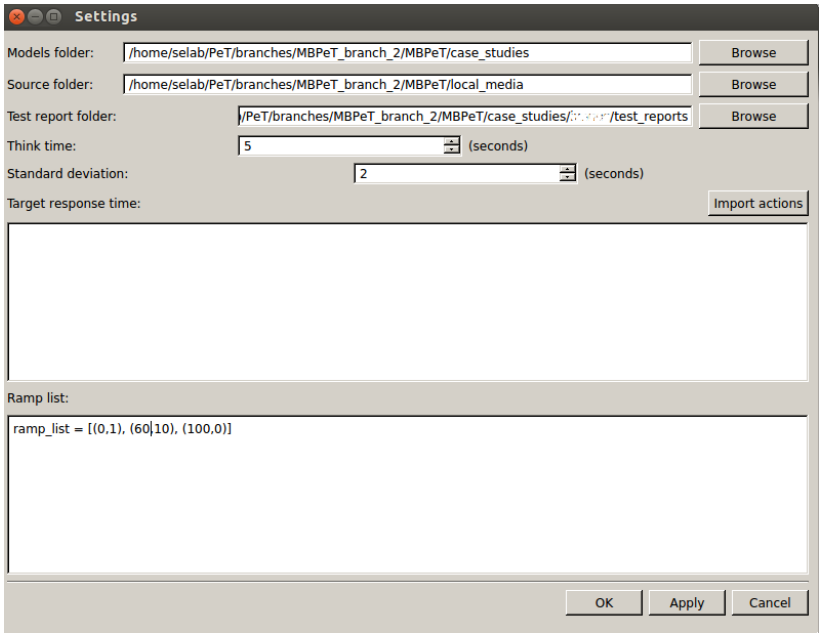

Figure 6.8: Settings window of the GUI

and that can be useful for uncovering certain sequences of actions which may have a negative impact of the performance. Such sequences would be difficult or maybe impossible to discover if static test scripts are used, where a fixed order of the actions is specified, and repeated over and over again. Every PTA has an *exit* location which will eventually be reached. By modifying the probability for the *exit* action, it is possible to adjust the length of the test.

The attributes of PTA models make them a good candidate for modeling the behavior of VUs, which imitate the dynamic behavior of real users. Actions in the PTA model corresponds to an action which a user can send to the SUT and the clocks present the user *think time*. In our case, the PTA formalism is implemented using the DOT notation.

Load is generated from these models by executing an instance of the model for every simulated VU. Whenever a transition with an action is fired, that action is translated by the MBPeT tool and sent to the SUT. This process is repeated and run in parallel for every simulated user throughout the whole test session. During load generation, the MBPeT tool monitors the SUT the whole time.

6.6.3 Test Reporting

After each test run the MBPeT tool generates a test report based on the monitored data. It is the slave nodes that are responsible for the monitoring and they report the values back to the master node which later creates the report.

Every slave node will monitor the communication with the SUT and collecting the data needed for test report. The slave node will start a timer every time and action is sent to the system. When a response is received, the timer is stopped and the response code together with the action name and response time is stored. This data is later sent to the master node which will aggregate the data and produce a report.

The slave node will also monitor its own resources so it does not get saturated and becomes the bottleneck during load generation. The slave node monitors is own CPU, memory, and disk utilization and sends the information to the master node. The master node the data is plotted in graphs and included in the test report.

It is the test report creation module of the master node that is responsible for creating test report. This module performs two tasks: aggregating data received from the slave nodes and creating a test report. Data aggregation consists of combining data received from the slave nodes together into and internal representation. Based on the received data, different kinds of statistical values are computer, e.g. mean and max response times, throughput, etc. Values such as response time and throughput plotted as graphs so the tester can see how the different values varies over time. Figures of the test report will later be shown throughout Section 6.7.

The final task of the test report creation module is to render all the values and graphs into a report. The final report is rendered as a HTML document.

6.7 Experiments

In this section we will describe a set of experiments carried out with the MBPeT tool on a case study. The system tested in the case study is an HTTP based auction web service.

6.7.1 YAAS

YAAS is a web application and a web service for creating and participating in auctions. An auction site is a good example of a service offered as a web application. It facilitates a community of users interested in buying or selling diverse items, where any user including guest user can view all the auctions

and all authenticated users, except seller of an item, can bid on the auction against other users.

The web application is implemented in Python language using the Django[2] web-framework. In addition to HTML pages, YAAS also has a RESTful [10] web service interface. The web service interface has various APIs to support different operations, including:

Browse API It returns the list of all active auctions.

Search API It allows to search auctions by title.

Get Auction This API returns an auction against the given Auction-ID.

Bids It is used to the get the list of all the bids have been made to a particular auction.

Make Bid Allows and authenticated user to place a bid on a particular auction.

6.7.2 Test Architecture

A setup of the test architecture can be seen in Figure 6.9. The server runs an instance of the YAAS application on top of an Apache web server. All nodes (master, slaves, and the server) feature an 8-core CPU, 16GB of memory, 1Gb Ethernet, 7200 rpm hard drive, and Fedora 16 operating system. The nodes were connected via a 1Gb ethernet over which the data were sent.

A populator script is used to generate input data (i.e., populate the test databases) on both the client and server side, before each test session. This ensures that the test data on either sides is consistent and easy to rebuild after each test session.

6.7.3 Load Models

The test database of the application is configured with a script to have 1000 users. Each user has exactly one auction and each auction has one starting bid.

In order to identify the different type of users for the YAAS application, we have used the AWStats[3] tool. This tool analyzes the Apache server access logs to generate a report on the YAAS application usage. Based on that report, we discovered three types of users; aggressive, passive and non-bidder.

[2]https://www.djangoproject.com/
[3]http://awstats.sourceforge.net

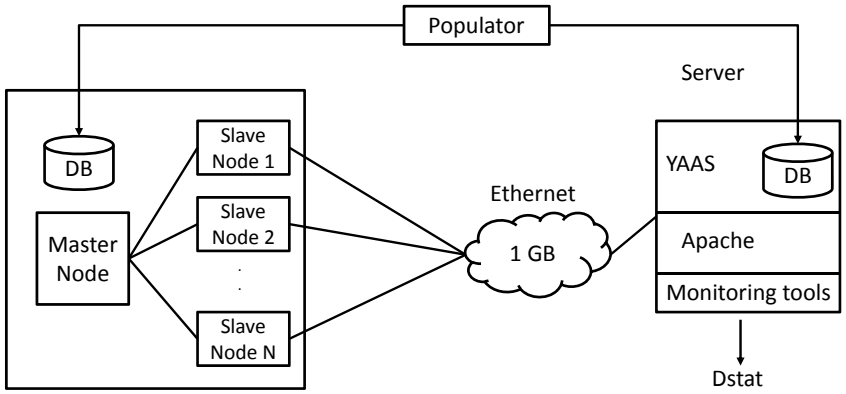

Figure 6.9: A caption of the test architecture

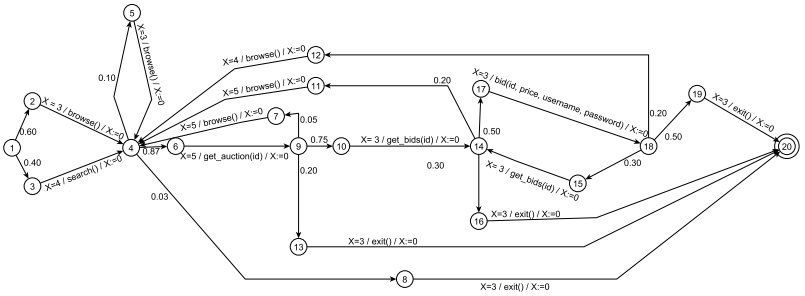

Figure 6.10: Aggressive User type model

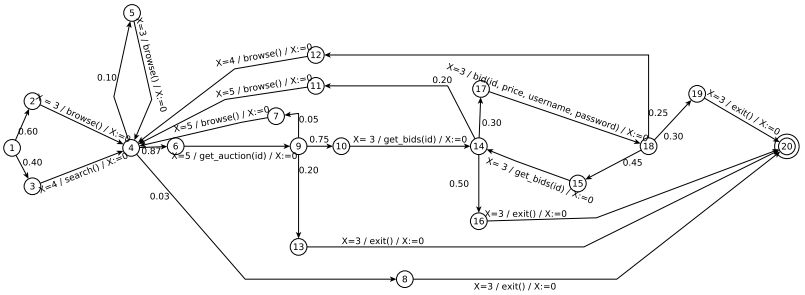

Figure 6.11: Passive User type model

Action	Dependency	Aggressive User		Passive User		Non-Bidder User	
		Think time	Frequency	Think Time	Frequency	Think Time	Frequency
search()		4	0,40	4	0,40	4	0,40
browse()		3	0,60	3	0,60	3	0,60
browse()	browse(),search()	5	0,10	3	0,10	3	0,10
get_auction()	browse(),search()	5	0,87	5	0,87	5	0,87
exit()	browse(),search()	3	0,03	3	0,03	3	0,03
browse()	get_auction()	5	0,05	5	0,05	5	0,05
get_bids()	get_auction()	3	0,75	3	0,75	3	0,75
exit()	get_auction()	3	0,20	3	0,20	3	0,20
browse()	get_bids()	5	0,20	5	0,20	5	0,60
bid()	get_bids()	3	0,50	3	0,30		
exit()	get_bids()	3	0,30	3	0,50	3	0,40
get_bids()	bid()	3	0,30	3	0,45		
browse()	bid()	4	0,20	4	0,25		
exit()	bid()	3	0,50	3	0,30		

Table 6.2: Think time and distribution values extracted from the AWStats report

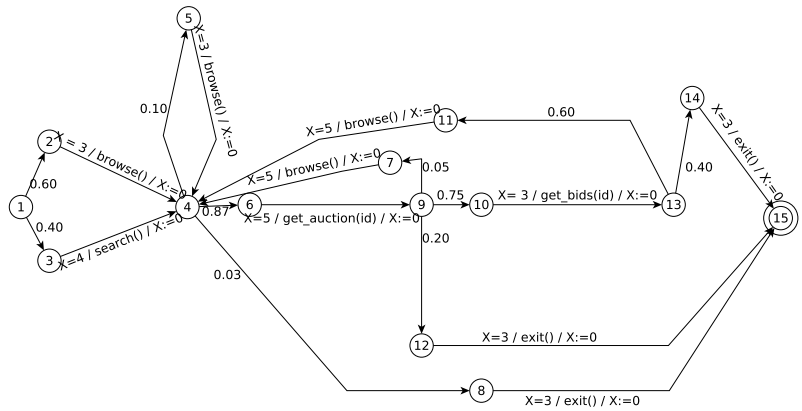

Figure 6.12: Non-bidder User type model

Table 6.2 shows the think time and distribution of actions for the three different types of users.

For each user type, a load model was created as describe in section 6.5. The aggressive type (Figure 6.10) of users describes those users, who make bids more frequently as compared to other types of users. The passive users (Figure 6.11) are less frequent in making bids, see for instance the locations 14 or 18 in the referred figures. The third type of users are only interested in browsing and searching for auctions instead of making any bids and are known as non-bidders (Figure 6.12). The root model of the YAAS application, shown in Figure 6.13, describes the distribution of different user types.

Based on the AWStats analysis, we determined that the almost 30% of total users who visited the YAAS, were very frequently in making bids, whereas rest of 50% users made bids occasionally. The rest of the users were not interested in making bids at all. This distribution is depicted by the model in Figure 6.13.

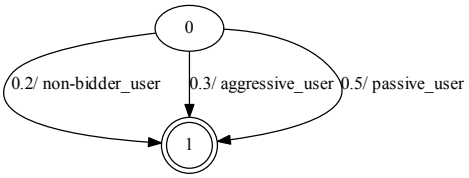

Figure 6.13: YAAS Root model

The models of all these user types were provided to the MBPeT tool to simulate them as virtual users. For example, the model of an *aggressive user type*, shown in Figure 6.10, shows that the user will start from the location *1*, and from this location the user will select either *browse* or *search* action based on a probabilistic choice. Before performing the action, the slave will wait for the think time corresponding to the selected action. Eventually, the user will reach the final location (i.e. location *20*) by performing the *exit* action and terminate the current user session. Similarly, the other models of *passive* and *non-bidder* user type have the same structure but with different probabilities and distribution of actions.

6.7.4 Experiment 1

The goal of this experiment was to set the target response time for each action and observe at what point the average response time of the action exceed the target value. The experiment ran for 20 minutes. The maximum number of concurrent users was set to 300 and the ramp up value was 0.9 that the tool would increase the number of concurrent users with the passage of time to achieve the value of 300 concurrent users when the 90% of test duration time has been passed.

The resulting test report has various sections, where each section presents the different perspective of the results. The first section, shown in Figure 6.14, contains the information about the test session including, test started time, test duration, target number of concurrent of users, etc. The *Total*

###################### Master Stats ######################

This test was executed at: 2013-07-01 16:54:47
Duration of the test: 20 min
Target number of concurrent users: 300
Total number of generated users: 27536
Measured Request rate (MRR): 27.68 req/s
Number of NON-BIDDER_USER: 6296 (23.0)%
Number of AGGRESSIVE_USER: 9087 (33.0)%
Number of PASSIVE_USER: 12153 (44.0)%
Average number of action per user: 91 actions

Figure 6.14: Test Report 1 - Section 1: General information

######## AVERAGE/MAX RESPONSE TIME per METHOD CALL ##########

	NON-BIDDER_USER (23.0 %)		PASSIVE_USER (44.0 %)		AGGRESSIVE_USER (33.0 %)	
Method Call	Average (sec)	Max (sec)	Average (sec)	Max (sec)	Average (sec)	Max (sec)
GET_AUCTION(ID)	3.04	23.95	2.85	23.67	2.93	24.71
BROWSE()	5.44	21.25	5.66	21.7	5.68	21.29
GET_BIDS(ID)	3.59	27.37	3.63	25.8	3.65	24.87
BID(ID,PRICE,USERNAME,PASSWORD)	0.0	0.0	8.26	33.44	8.11	36.84
SEARCH(STRING)	3.36	12.86	3.26	15.84	3.47	15.79

Figure 6.15: Test Report 1 - Section 2: Average and Maximum response time of SUT per action or method call

number of generated users in the report describes that the tool had simulated 27536 numbers of virtual users. The *Measured Request Rate (MRR)* depicts the average number of requests per second which were made to the SUT during the load generation process. Moreover, it also shows the distribution of total number of user generated which is very close to what we have defined in the root model (Figure 6.13). This section is useful to see the summarized view of the entire test session.

In the second section of the test report, we could observe the SUT performance for each action separately, and identify which actions have responded with more delay than the others, and which actions should be optimized to increase the performance of the SUT. As from the table in Figure 6.15, it appears that the action *BID(ID, PRICE, USERNAME, PASSWORD)* has larger average and maximum response time than the other actions. The *non-bidder* users do not perform the *BID* action that is why we have zero response time in the column of *NON-BIDDER_USER* against the *BID* action.

Section three (shown in Figure 6.16) of the test report presents a comparison of the SUTs desired performance against the measured performance. As we had defined the target response time for each action in the test config-

AVERAGE/MAX RESPONSE TIME THRESHOLD BREACH per METHOD CALL

Action	Target Response Time		NON-BIDDER_USER		PASSIVE_USER		AGGRESSIVE_USER		Verdict
	Average (secs)	Max (secs)	Average users (secs)	Max users (secs)	Average users (secs)	Max users (secs)	Average users (secs)	Max users (secs)	Pass/Fail
GET_AUCTION(ID)	2.0	4.0	70 (251)	84 (299.0)	70 (251)	95 (341.0)	70 (250)	95 (341.0)	Failed
BROWSE()	4.0	8.0	84 (299)	97 (345.0)	84 (299)	113 (403.0)	84 (299)	113 (403.0)	Failed
GET_BIDS(ID)	3.0	6.0	84 (298)	112 (402.0)	83 (296)	112 (402.0)	96 (344)	112 (401.0)	Failed
BID(ID,PRICE,USERNAME,PASSWORD)	5.0	10	Passed	Passed	97 (346)	113 (405.0)	112 (402)	135 (483.0)	Failed
SEARCH(STRING)	3.0	6	95 (341)	134 (479.0)	96 (342)	112 (402.0)	83 (296)	133 (476.0)	Failed

Figure 6.16: Test Report 1 - Section 3: Average and Maximum response time of SUT per action or method call

uration, in this section we could actually observe how many concurrent users were active when the target response time was breached. The table in this section allows us to estimate the performance of current system's implementation. For instance, the target average response time for the *GET_AUCTION* action was breached at 250 seconds for the *aggressive* type of users, when the number of concurrent users was 70. Further, this section demonstrates that the SUT can only support up to 84 concurrent users before it breaches the threshold value of 3 seconds for *GET_BIDS* action for the *passive* type of users. In summary, all the actions in Figure 6.16 have breached the target response time except the *BID* action in *NON-BIDDER_USER* column because *non-bidder* users do not bid.

Figures 6.17 and 6.18 display the resource load at the SUT during load generation. These graphs are very useful to identify which resources are being utilized more than the others and limiting the performance of SUT. For instance, it can be seen from Figure 6.17 that after 400 seconds the CPU utilization was almost equal to 100% for the rest of the test session, it means that the target web application is CPU-intensive, and it might be the reason of large response time.

Figure 6.19 illustrate that the response time of each action for the aggressive user type increases proportionally to the number of concurrent users. The figure also points out which actions response time is increasing much faster than the other actions and require optimization. Similar patterns was observed for the two other user types: passive users and non-bidder, respectively.

For example the response time of action *BID(ID, PRICE, USERNAME, PASSWORD)* for *aggressive* and *passive* user types increases more rapidly than the other actions. It might be because the *BID* action involves a write operation and in order to perform a write operation on the database file, the

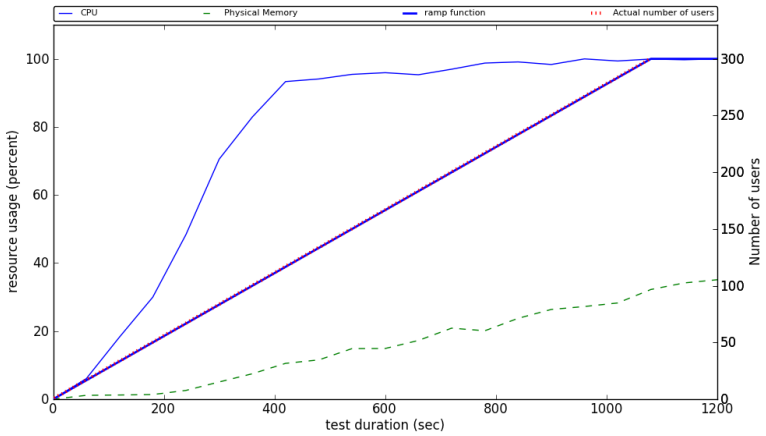

Figure 6.17: Test Report 1 - SUT CPU and memory utilization

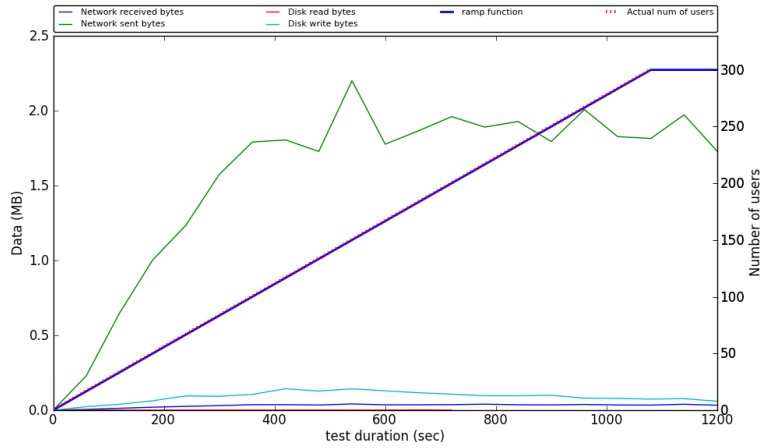

Figure 6.18: Test Report 1 - SUT network and disk utilization

SQLite[4] database has to deny the all new access requests to the database and wait until all previous operations (including read and write operations) have been completed.

Section four of the test report provides miscellaneous information about

[4]http://www.sqlite.org/

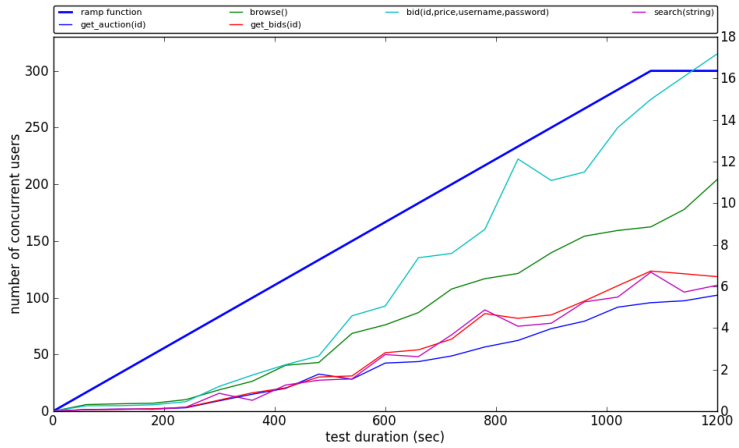

Figure 6.19: Test Report 1 - Response time of aggressive user type per action

the test session. For example, the first erroneous response was recorded at 520 seconds (according to Figure 6.20) and at that time the tool was generating load at the maximum rate, that is 1600 actions/seconds, shown in Figure 6.21. Similarly, Figure 6.20 displays that there was no error until the number of consecutive users exceeded 150, after this point errors began to appear and increased steeply proportional to the number of consecutive users.

A further deep analysis of the test report showed that the database could be the bottleneck. Owning to the fact a *sqlite* database has been used for this experiment, the application has to block the entire database before something can be written to it. It could explain the larger response time of *BID* actions compared to other actions. This is because the web application had to perform a write operation to the database in order to execute the *BID* action. Further, before each write operation, *sqlite* creates a rollback journal file, an exact copy of original database file, to preserve the integrity of database [17]. This could also delay the processing of a write operation and thus cause a larger response time.

6.7.5 Experiment 2

In the second experiment, we wanted to verify the hypothesis, which we proposed in the previous experiment: *database could be the performance bottleneck*. We ran the second experiment for 20 minutes with the same test

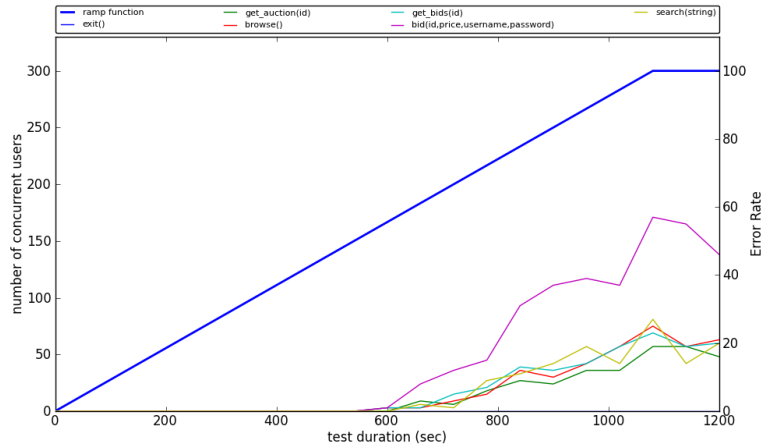

Figure 6.20: Test Report 1 - Error rate

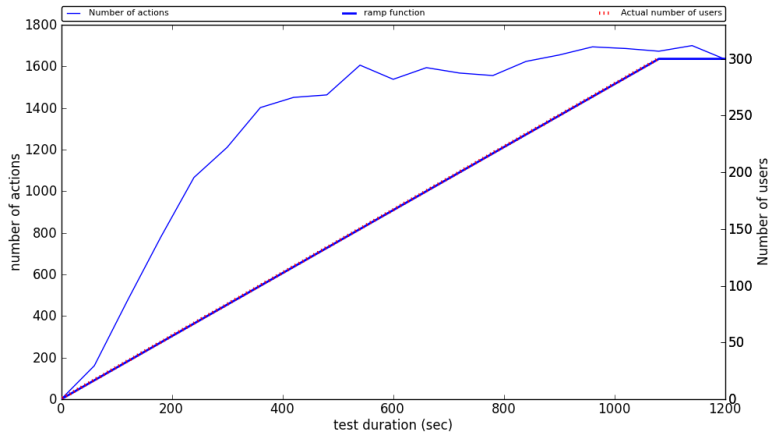

Figure 6.21: Test Report 1 - Average number of actions

configuration of the previous experiment. However, we did make one modification in the architecture. In the previous experiment, the *SQLite 3.7* was used as database server, but in this experiment, it was replaced by the *PostgreSQL 9.1* [5]. The main motivating factor of using the PostgreSQL

[5]http://www.postgresql.org

#################### Master Stats ######################

This test was executed at: 2013-07-01 17:37:38
Duration of the test: 20 min
Target number of concurrent users: 300
Total number of generated users: 35851
Measured Request rate (MRR): 39.21 req/s
Number of AGGRESSIVE_USER: 11950 (33.0)%
Number of NON-BIDDER_USER: 7697 (21.0)%
Number of PASSIVE_USER: 16204 (45.0)%
Average number of action per user: 119 actions

Figure 6.22: Test Report 2 - Section 1: global information

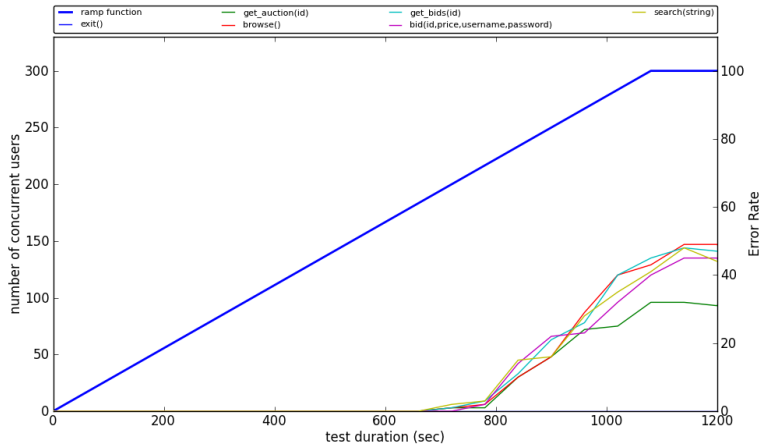

Figure 6.23: Test Report 2 - Error rate

database is that it supports the better concurrent access to the data than the SQLite. The PostgreSQL database uses the Multiversion Concurrency Control (MVCC) model instead of simple locking. In MVCC, different locks are acquired for the read and write operations, it means that the both operations can be performed simultaneously without blocking each other [14].

In the section 1 of Test report 2 (Figure 6.22) shows that the *Measured Request Rate (MRR)* increased by 42%. Additionally, each user performed averagely 30% more actions in this experiment.

Similarly in the second section (Figure 6.24), the average and maximum response time of all action decreased by almost 47%. Moreover, the error rate section (Figure 6.23) depicts that there was no error until the number

######## AVERAGE/MAX RESPONSE TIME per METHOD CALL ##########

	AGGRESSIVE_USER (33.0 %)		PASSIVE_USER (45.0 %)		NON-BIDDER_USER (21.0 %)	
Method Call	Average (sec)	Max (sec)	Average (sec)	Max (sec)	Average (sec)	Max (sec)
GET_AUCTION(ID)	1.18	15.58	1.1	15.95	1.25	15.8
BROWSE()	4.99	23.61	5.13	23.47	5.23	23.6
GET_BIDS(ID)	1.51	15.25	1.54	15.56	1.63	15.02
BID(ID,PRICE,USERNAME,PASSWORD)	3.25	18.65	3.25	18.37	0.0	0.0
SEARCH(STRING)	1.48	14.66	1.54	14.83	1.43	15.43

Figure 6.24: Test Report 2 - Section 2: Average and Maximum response time of SUT per action or method call

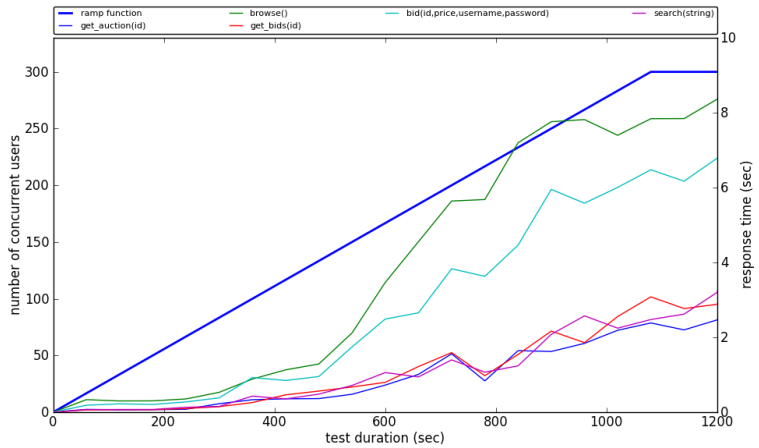

Figure 6.25: Test Report 2 - Response time of aggressive user type per action

of concurrent users was below 182, that is 21% more users than the last experiment.

Figure 6.25 shows that the response time of aggressive type of users is decreased by 50% approximately in comparison with the previous experiment in Figure 6.19. In summary, all of these indicators suggest significant improvement in the performance of SUT.

6.8 Conclusions

In this chapter, we have presented a tool-supported approach for model-based performance testing. Our approach uses PTA models to specify the probabilistic distribution of user types and of actions that are executed against

the system.

The approach is supported by the MBPeT tool, which has a distributed scalable architecture, targeted to cloud-based environments allowing it to generate load at high rates. The tool generates load in online mode and monitors different KPIs including the resource utilization of the SUT. It can be run both in command line and in GUI mode, respectively. The former facilitates the integration of the tool in automated test frameworks, whereas the latter allows the user to interact with the SUT and visualize in real-time its performance depending on the number of concurrent users.

Using our modeling approach, the effort necessary to create and update the user profiles is reduced. The adapter required to interface with the SUT has to be implemented only once and then it can be reused. As shown in the experiments, the tool allows quick exploration of the performance space by trying out different load mixes. In addition, preliminary experiments have shown that the synthetic load generated from probabilistic models has in general a stronger impact on the SUT compared to static scripts.

We have also showed that the tool us sufficient enough in finding performance bottlenecks and that the tool can handle large amounts of parallel virtual users. The tool benefits from its distributed architecture in the sense that it can easily be integrated in a cloud environment where thousands of concurrent virtual users need to be simulated.

Future work will be targeted towards improving the methods for creating the user profiles from historic data and providing more detailed analysis of the test results. So far, the MBPeT tool has been used for testing web services however, we plan also to address also web applications, as well as other types of communicating systems.

References

[1] Ashlish Jolly. *Historical Perspective in Optimising Software Testing Efforts*. 2013. URL: `http://www.indianmba.com/Faculty\_Column/FC139/fc139.html`.

[2] C. Barna, M. Litoiu, and H. Ghanbari. "Model-based performance testing (NIER track)". In: *Proceedings of the 33rd International Conference on Software Engineering*. ICSE '11. Waikiki, Honolulu, HI, USA: ACM, 2011, pp. 872–875. ISBN: 978-1-4503-0445-0. DOI: `10.1145/1985793.1985930`.

[3] M. Calzarossa, L. Massari, and D. Tessera. "Workload Characterization Issues and Methodologies". In: *Performance Evaluation: Origins and Directions*. London, UK, UK: Springer-Verlag, 2000, pp. 459–481. ISBN: 3-540-67193-5.

[4] G. Denaro, A. Polini, and W. Emmerich. "Early performance testing of distributed software applications". In: *Proceedings of the 4th international workshop on Software and performance*. WOSP '04. Redwood Shores, California: ACM, 2004, pp. 94–103. ISBN: 1-58113-673-0. DOI: 10.1145/974044.974059.

[5] E. Gansner, E. Koutsofios, and S North. *Drawing draphs with dot*. Online at http://www.graphviz.org/Documentation/dotguide.pdf. 2006. URL: http://www.graphviz.org/Documentation/dotguide.pdf.

[6] Hewlett-Packard. *httperf*. retrieved: October, 2012. URL: http://www.hpl.hp.com/research/linux/httperf/httperf-man-0.9.txt.

[7] HP. *HP LoadRunner*. 2013. URL: http://www8.hp.com/us/en/software-solutions/software.html?compURI=1175451\#.URz7wqWou8E.

[8] ITEA 2. *ITEA 2 D-MINT project result leaflet: Model-based testing cuts development costs*. 2013. URL: http://www.itea2.org/project/result/download/result/5519?file=06014_D_MINT_Project_Leaflet_results_oct_10.pdf.

[9] M. Jurdziński et al. "Concavely-Priced Probabilistic Timed Automata". In: *Proc. 20th International Conference on Concurrency Theory (CONCUR'09)*. Ed. by M. Bravetti and G. Zavattaro. Vol. 5710. LNCS. Springer, 2009, pp. 415–430.

[10] L. Richardson and S. Ruby. *RESTful Web Services*. O'Reilly Media. 2007.

[11] D. A. Menasce and V. Almeida. *Capacity Planning for Web Services: metrics, models, and methods*. 1st. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2001. ISBN: 0130659037.

[12] D. Mosberger and T. Jin. "httperfa tool for measuring web server performance". In: *SIGMETRICS Perform. Eval. Rev.* 26.3 (Dec. 1998), pp. 31–37. ISSN: 0163-5999. DOI: 10.1145/306225.306235. URL: http://doi.acm.org/10.1145/306225.306235.

[13] D. C. Petriu and H. Shen. "Applying the UML Performance Profile: Graph Grammar-based Derivation of LQN Models from UML Specifications". In: Springer-Verlag, 2002, pp. 159–177.

[14] PostgreSQL. *Concurrency Control*. retrieved: March, 2013. URL: http://www.postgresql.org/docs/9.1/static/mvcc-intro.html.

[15] G. Ruffo et al. "WALTy: A User Behavior Tailored Tool for Evaluating Web Application Performance". In: *Network Computing and Applications, IEEE International Symposium on* 0 (2004), pp. 77–86. DOI: http://doi.ieeecomputersociety.org/10.1109/NCA.2004.1347765.

[16] M. Shams, D. Krishnamurthy, and B. Far. "A model-based approach for testing the performance of web applications". In: *SOQUA '06: Proceedings of the 3rd international workshop on Software quality assurance*. Portland, Oregon: ACM, 2006, pp. 54–61. ISBN: 1-59593-584-3. DOI: http://doi.acm.org/10.1145/1188895.1188909.

[17] SQLite. *File Locking And Concurrency In SQLite Version 3*. retrieved: March, 2013. URL: http://www.sqlite.org/lockingv3.html.

[18] Sun. *Faban Harness and Benchmark Framework*. 2013. URL: http://java.net/projects/faban/.

[19] The Apache Software Foundation. *Apache JMeter*. Retrieved: October, 2012. URL: http://jmeter.apache.org/.