Tanwir Ahmad | Fredrik Abbors | Dragos Truscan | Ivan Porres

Model-Based Performance Testing Using the MBPeT Tool

Turku Centre for Computer Science

TUCS Technical Report
No 1066, February 2013

Model-Based Performance Testing Using the MBPeT Tool

Tanwir Ahmad
Fredrik Abbors
Dragos Truscan
Ivan Porres

Åbo Akademi University, Department of Information Technology
Lemminkäisenkatu 14 A, 20520 Turku, Finland
{tahmad, fabbors, dtruscan, iporres}@abo.fi

TUCS Technical Report
No 1066, February 2013

Abstract

This document describes a model-based performance testing tool, called MBPeT, used for generating synthetic workload from probabilistic models. The purpose of the tool is two fold: (a) to simplify the creation and updating of test specifications by using abstract models for the user profiles and (b) to create, based on a certain level of randomness introduced by the probabilistic models, synthetic workload which is closer to the real workload as compared to the workload generated from static scripts. MBPeT has a distributed architecture, where one master node controls multiple remote slave nodes. Each slave node is capable of generating synthetic workload based on the specified load profile. Besides measuring different key performance indicators of the system under test, the tool monitors permanently the resource utilization of its slave nodes and (when possible) the one of the system under test. At the end of each test run, the measurements are aggregated by the master into a detailed test report. This report describes the implementation details behind MBPeT and provides several experiments highlighting the capabilities of the tool.

Keywords: load generation, performance testing, probabilistic timed automata, tool support

TUCS Laboratory
Software Engineering Laboratory

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 *system under test* (SUT). 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 defect 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 *synthetic workload*. The purpose of the synthetic 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 is typically achieved by running a series of users concurrently, 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 major drawback to this approach is that it exercises only a limited combination of user actions which is a simplification of the real usage. Traditionally, areas of interest in performance testing are the system response times, throughput, scalability, reliability, resource usage, etc. Today, these areas are equally important to that of functionality.

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 a lot of manual work, especially in creating test cases. A software system typically undergoes a lot of changes during its lifetime. Whenever a pieces of code is changed, a test have 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.

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 an abstract model, representing the behavior or the use of the SUT. Tests are then automatically generated form the model. In MBT, the focus has shifted from manually creating tests to creating and maintaining an abstract model. Due to

the fact that tests are automatically generated from that model, MBT copes better with changing requirements and code compared to traditional testing. Research has shown that MBT could reduce the total testing costs with 15 percent [5].

There exist a plethora of commercial performance testing tools. In the following, we briefly enumerate a short list of the most popular performance testing tools. FABAN is an open source framework for developing and running multi-tier server benchmarks [10]. 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 [11] 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 scripts that are run concurrently. Httperf [3] 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 [4] 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.

MBPeT is tool for load generation and system monitoring. Load is generated from *probabilistic timed automata* (PTA) models describing the behavior of groups of virtual users. The load is applied on the system while being generated and different *key performance indicators* (KPIs) such as response times, throughput, memory, CPU, disk, etc. are monitored. The MBPeT tool has a distributed architecture where one master node controls several slave nodes which are the actual load generators. This means that the MBPeT tool is very suitable for a cloud environment where computational nodes can be rented in an on-the-fly manner. Besides monitoring, the tool also produces an performance test report at the end of the test. The report contains information about important KPIs, such as response times, throughput etc, but also graphs showing how CPU, memory, disk, network utilization varied during the test session.

The rest of the report is structured as follows: In Section 2, we briefly describe the load generation process. In Section 3, we overview the architecture of the tool. In Section 4, we discuss the load generation process used by the tool. Section 5, discusses the tool implementation and libraries used. In Section 6, we present a auction web service case study and a series of experiments using our tool including an evaluation of our approach against JMeter. Finally, in Section 7 we present our conclusions and discuss future work.

2 The Load Generation Process

The MBPeT tool takes as input a set of Probabilistic Timed Automata (PTA) [6] models as shown in Figure 1, applies the load against the SUT and generates a test report in HTML format. We use these PTA models to describe the behavior of different groups of *virtual users* (VUs). In brief, a PTA is composed of *locations* connected by *transitions*. Transitions have associated probability distributions which allow a PTA to decide based on a probabilistic choice what is the next transition to take. Transitions can have associated time actions which specify how time advances (e.g., how long to wait before taking the transition) or actions (what is being executed once the transition is enabled). An example PTA is given in Figure 2. In other words, if the time has not passed a certain set value, that particular transition can not be fired. A location can have more than one outgoing transitions. In that case, the transition to fire is based on a probabilistic choice. If a transition has a action attached to it, that action is carried out when the transition is fired. Every PTA has an exit location (depicted with a double circle) that will eventually be reached.

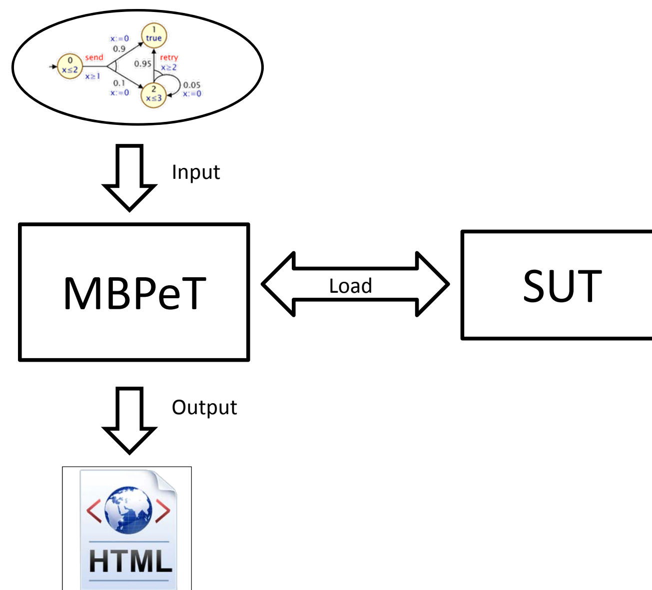


Figure 1: Overview of the MBPeT tool

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 system under test (SUT) and the clocks represent the user *think time*.

Load is generated from these models by executing an instance of the model

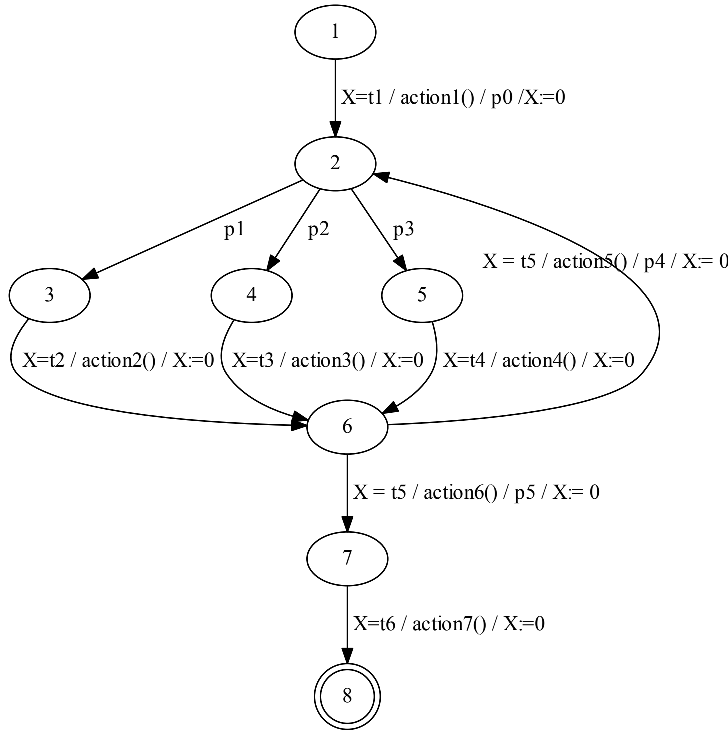


Figure 2: Example of a probabilistic timed automaton

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 for every simulated user and throughout the whole load generation process. During load generation, the MBPeT tool monitors the SUT the whole time. At end of the load generation process, an HTML report is created. The HTML report information about important metrics and shows graphs for different KPIs. More details will follow in the next sections.

3 Tool architecture

MBPeT has a distributed architecture. It consists of two types of nodes: a master node and a slave node. A single master node is responsible of initiating and controlling multiple remote slave nodes, as shown in Figure 3. Slave nodes are designed to be identical and generic, in the 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 test session configuration and sends that information to all the slave nodes. Once all slaves are initialized, the master begins the load generation process by starting a single slave while rest of the slaves are idling.

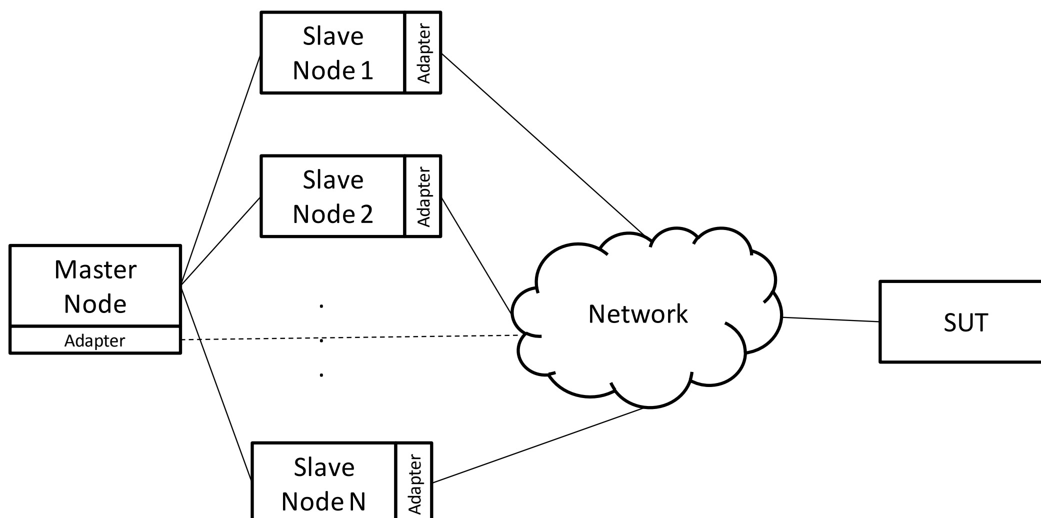


Figure 3: Distributed architecture of MBPeT tool

3.1 Master Node

The internal architecture of the master node is shown in Figure 4. It contains the following components:

3.1.1 Core

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:

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 [2].

2. *Test Configuration:* It is a collection of different parameters, that are defined in a *Settings* file, which is 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 several mandatory parameters in the *Settings* file, which are 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.

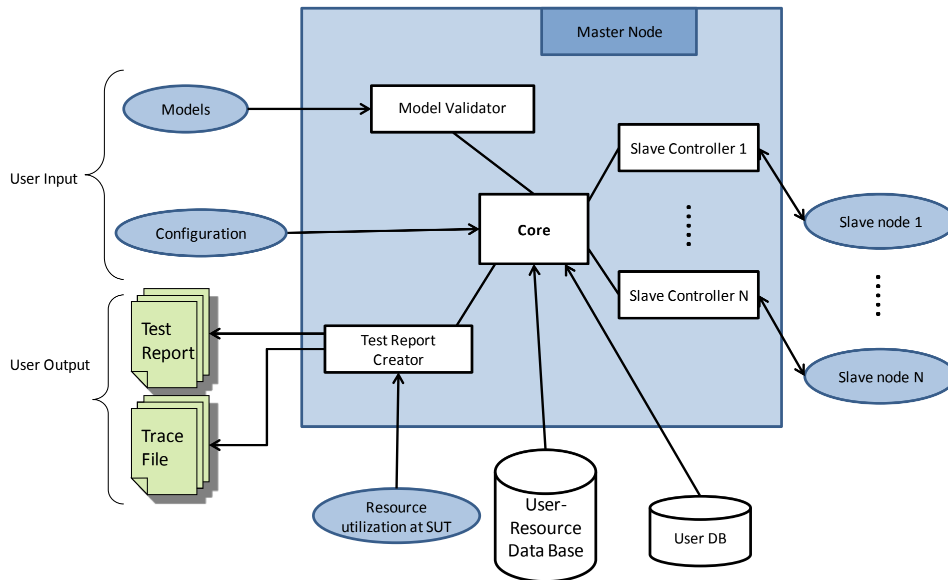


Figure 4: Master Node

(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 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 5, as illustrated in the Listing 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 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.

. . .

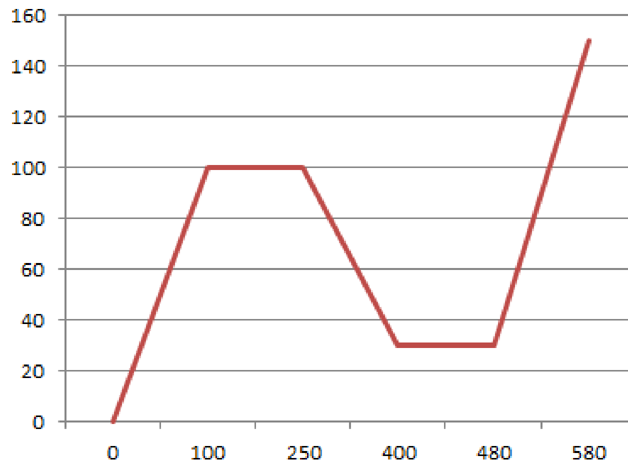


Figure 5: Example ramp function

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

Listing 1: Ramp section of Settings file

(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 abstract 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 are participating in the test session. As detailed later, at startup, the master node waits for all slaves to report theirpresence before proceeding to load generation.

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 a row of information for each resource column after every update, as shown in Figure 6. 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.

```
----total-cpu-usage---- -dsk/total- -net/total- ---paging-- ---system--
usr sys idl wai hiq siq| read  writ| recv  send|  in   out | int   csw 
  1   6  87   4   1   2| 894k   19M|   0     0 | 14B  689B|3982  8877 
  1  18  73   0   2   7|2724k   66M|  50M  973k|   0     0 |  12k   19k
  0  18  73   0   1   7|2768k   65M|  49M  954k|   0     0 |  12k   19k
  0  15  78   0   1   6|2432k   50M|  42M  816k|   0     0 |  10k   15k
  1  16  75   0   2   7|3200k   61M|  46M  884k|   0     0 |  10k   17k
  1  18  73   1   2   6|3388k   66M|  48M  904k|   0     0 |  11k   18k
  0  20  71   1   2   6|3724k   64M|  49M  854k|   0     0 |  13k   21k
  1  19  72   0   2   7|2980k   64M|  49M  846k|   0     0 |  12k   21k
  0  17  75   0   2   6|2388k   60M|  47M  918k|   0     0 |  11k   16k
  1  18  73   1   2   6|3644k   66M|  50M  975k|   0     0 |  12k   18k
  0  18  72   1   2   7|3432k   62M|  47M  818k|   0     0 |  11k   19k
  1  21  70   0   2   7|3920k   66M|  49M  834k|   0     0 |  13k   21k
  1  17  75   0   2   6|3428k   55M|  45M  800k|   0     0 |  11k   18k
  0  16  76   0   1   7|2892k   60M|  46M  903k|   0     0 |  11k   17k
  0   9  87   0   0   4|1228k   26M|  26M  515k|   0     0 |5895   10k
  0   3  76  21   0   0| 504k   21M| 192B  484B|   0     0 |1276  6645 
  0   3  71  24   0   0| 784k   19M|4370k   53k|   0     0 |1674  7095 
  0   7  78  10   1   4|1236k   17M|  30M  537k|   0     0 |7130   10k
  1  20  71   0   2   6|2372k   63M|  49M  843k|   0     0 |  13k   22k
  1  18  74   0   2   6|1828k   57M|  46M  798k|   0     0 |  12k   19k
  1  17  75   0   2   6|1800k   55M|  44M  766k|   0     0 |  11k   17k
  1  21  70   0   2   7|2084k   62M|  50M  812k|   0     0 |  13k   21k
  1  19  71   1   2   6|2240k   64M|  45M  729k|   0     0 |  12k   21k
  1  19  71   0   3   7|2048k   60M|  49M  825k|   0     0 |  13k   20k
```

Figure 6: Dstat log file example

3.1.2 Model Validator

The *Model Validator* module validates the user models. It performs the different number of syntactic checks on all models and generates a report similar to a report

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

presented in Listing 2. This report describes the error description and the location in model where it is occurred. A model with syntax anomalies could lead to the 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

```
======Running Test # 1 : test_InitialFinalState
Description : Checks that a model should have one initial
   and one final state.
PASSED:: Model contains only 1 initial state: ['1']
PASSED:: Model contains only 1 final state: ['9']
======Running Test # 2 : test_actionCheck
Description : Checks whether the all transitions have
   action names or not
FAILED::Action name is not specified for transition from 2
   to 3 with probability 0.35
FAILED::Action name is not specified for transition from 2
   to 4 with probability 0.21
FAILED::Action name is not specified for transition from 5
   to 2 with probability 1.0
FAILED::Action name is not specified for transition from 7
   to 2 with probability 1.0
FAILED::Action name is not specified for transition from 6
   to 2 with probability 1.0
FAILED::Action name is not specified for transition from 8
   to 2 with probability 1.0
======Running Test # 3 : test_isolationCheck
Description : Checks for isolated states in a model
PASSED::There is no isolated state in the model.
======Running Test # 4 : test_probabilityCheck
Description : Checks the sum of probabilities of all
   outgoing transitions from a state must be equal to 1.
PASSED::Probability sum of each state is equal to 1.
```

Listing 2: Sample report of a PTA model syntax validation

3.1.3 Slave Controller

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 session. The benefit of this architecture is in keeping 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.

3.1.4 Test Report Creator

This module performs two tasks: Data Aggregation and Report Creation. In the first task, it combines the test result 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.

3.2 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.

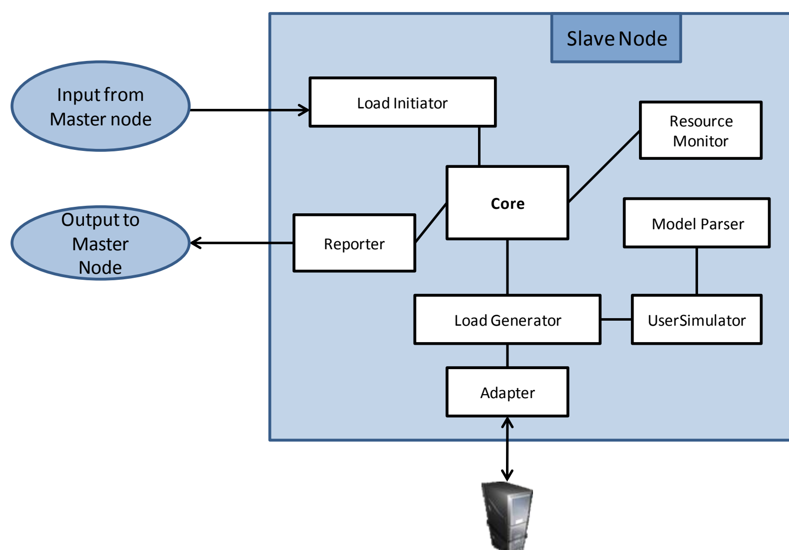


Figure 7: Slave Node

3.2.1 Load Initiator

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.

3.2.2 Model Parser

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.

3.2.3 Load Generator

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.

3.2.4 Resource Monitor

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.

3.2.5 Reporter

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.

4 Load Generation

In each test setup, there is one master node that carries out the entire test session and generates a report. The user only interacts with the master node by initializing it with the required parameters (mentioned in the section 3.1.1) and getting the test report at the end of the test run, as illustrated in Figure 8. The master core uses the given information to set 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. If the user decides to continue or the validation was successful, then the master enters into the next phase.

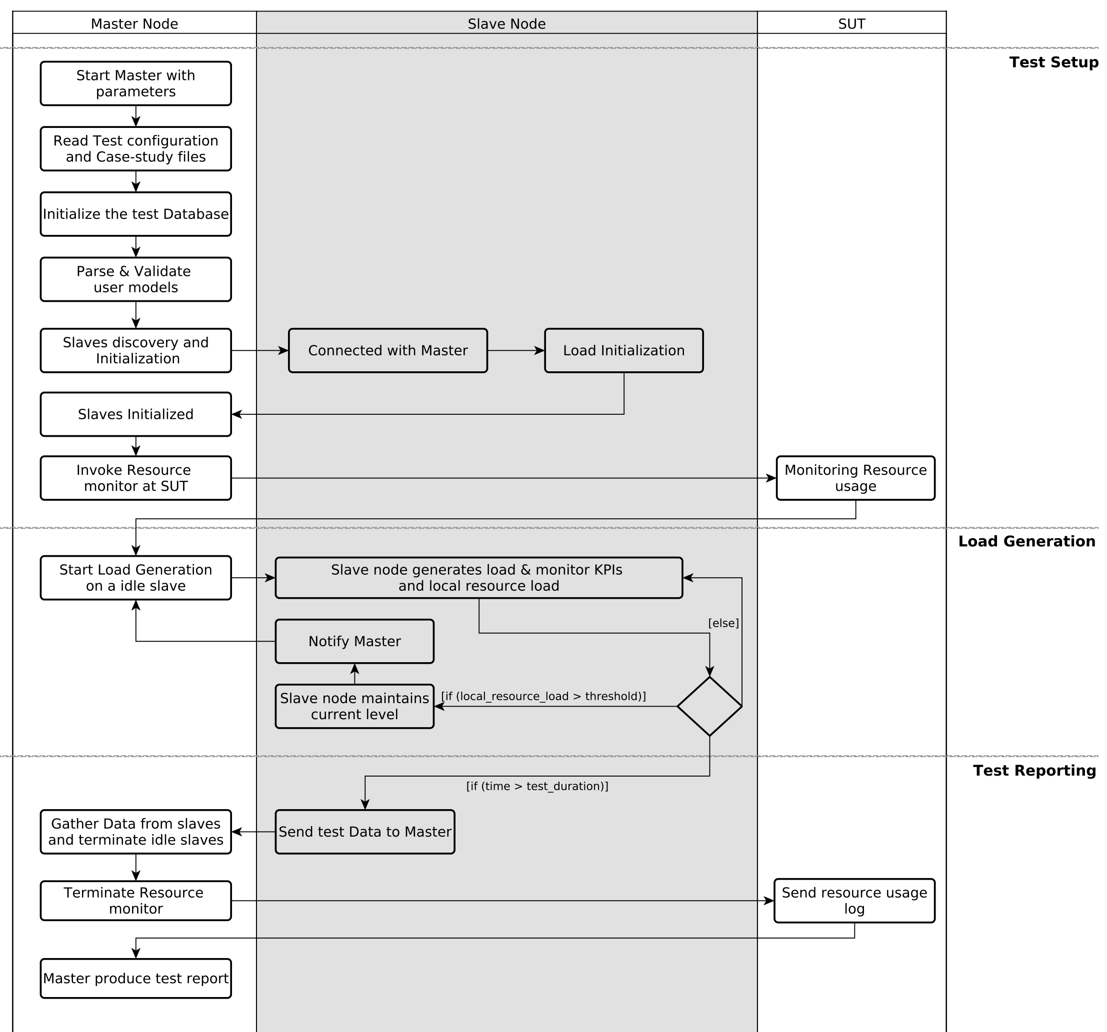


Figure 8: MBPeT tool activity diagram

4.1 Test Setup

This phase consists of two parts: slave discovery and slave initialization, as follows.

4.1.1 Slave Discovery

In this phase, master starts listening for incoming connections from the slave nodes. It opens the socket that listens for TCP connection on given TCP/IP port. And all slaves are by default configured to look for master on a same port at the given IP address. Whenever the master receives a new connection request from the slave, it accepts the request and initializes the new internal process called *SlaveController* for the slave. Subsequently, it passes the new slave connection object to the newly created SlaveController.

The master node only accepts the defined maximum number of slave connections. As soon as all slaves have made a connection with the master, the master core process stops listening for the new connections. Afterwards, the master core waits until all SlaveControllers inform the core process that they have successfully initialized all the slaves and they are ready to generate load.

4.1.2 Slave Initialization

The SlaveController starts the slave initialization process as soon as it connects with the remote slave node. It sends the all the information that is required by the slave nodes to generate load. All SlaveController instances use the information that has already been setup by the master core module.

Each SlaveController sends the following informations to the slave node:

- *Command-line arguments:* SlaveController also forwards the command-line arguments to the slave node.
- *Test-database:* Test-database (described in Section 3.1.1) is a collection of two databases: a user database and a user resource database. It is utilized during the load generation process. The test database is retrieved by the master core module and sent to all slaves. Because of that, each slave node does not have to retrieve the database separately from the SUT, it speeds up the load initialization phase.
- *Test Configuration, Case-study and model files:* In order to minimize communication delays as much as possible, all necessary files are packed into one zip file and send it to slave node. Later on, this zip file is extracted by the slave node to get the files back.

Once the slave node is initialized, it acknowledges the master node and starts waiting for the response in an idle state. At this point, the slave could receive

either of the following two commands: *kill* or *run* command. The master node sends a kill command to all idle slave nodes to terminate themselves at the end of each test session. In the later case, the master node sends a run command with the following parameters: 1) target number of users, 2) number of users generated by other slaves and 3) test start time. The test start-time is a timestamp when the first slave node started the load generation process. If this node is the first slave node started by the master then the test started time value would be zero. Then, before starting load generation, the slave node sends a message to the master with test starting time along with the list of all possible actions a user can execute. The list of actions is trivial in the report generation process and it is only sent by the first slave, all other slaves only send the starting time of load generation. After sending that message, the slave node initiates the Resource Monitor module and enters into load generation phase.

4.2 Load Generation

As load initialization phase is completed on all slave nodes, the user is notified by the master core that the test environment has been setup and the tool is ready to start the test session. After getting the confirmation from the user, the master selects one slave from the pool of idle slave nodes and starts the load generation on the slave, while all other slave nodes wait in their idle state.

Next, the master remotely invokes the *dstat* tool on SUT by sending the command via SSH protocol. This tool monitors the system resources utilization information after every specific time interval and appends that information in a log file, as shown in Figure 6.

During the load generation phase, the slave node performs the following steps in a loop until the test duration has been completed:

1. It uses the ramp function (described in Section 3.1.1) to calculate the number of users required at the given moment in time.

2. Then it queries the Resource Monitor module to check whether the resource utilization level has gone over the given threshold value or not. And if it has crossed the limit then it stops creating more virtual users and notifies the master along with the information of how many virtual users that have been spawned.

3. If the resource load is still under the threshold and the output of the ramp function suggested for more users to be added, then slave core starts creating more virtual users by instantiating the *UserSimulator* module. Each instance of the UserSimulator module represents an individual virtual user, capable of generating synthetic workload. Once the UserSimulator module is initialized with a new user ID, it continues to execute the following steps until the end of the test session or ramp decrease:

(a) In addition to the user models, each case-study defines a so call *root model*, which specifies how many user types are used in the load mix and what is their probabilistic distribution. The UserSimulator module simulates the *root model* of the case-study by selecting a type of the user to simulate. An example of a root model with three user types will be shown in Figure 12).

(b) Once the UserSimulator module obtains the user type, it starts simulating the corresponding user model.

(c) Whenever a transition is traversed in the user model, a user's action is retrieved. The action is sent to the SUT through the Adapter module.

(d) After performing the action, the UserSimulator stores the response time of the action and status code for the KPI analysis.

(e) The UserSimulator module ends the current simulation of a VU if it encounters the *exit* action during the simulation or the test session is completed. In the first case, it will go back to Step a) and choose a new type of the user, and in second case it will terminate itself. If neither of the previous two conditions are true, then it will wait for a specific period of time (i.e. think time) and go back to Step c).

4. Next, the slave node checks whether the test duration has been completed or not, if not then it goes back to Step 1, otherwise to the next step.

5. At this step, each user performs the *exit* operation and dumps all the data which has been gathered during load generation to a file.

While generating load, the slave nodes monitor the KPIs of the SUT and their own local resource utilization by using the Resource Monitor module. As the local resource utilization of the slave node approaches to a given threshold value, the slave node stops ramping up the number of concurrent users and notifies the master node. As a response, the master node invokes one of the remaining idle slaves to start generating load. This mechanism is employed to maintain the given load generation rate. For example, the resource utilization threshold values is set to 80% and during load generation a slave discovers that CPU utilization is crossing the threshold, then instead of saturating the local resources and dropping the load generation rate, the slave notifies the master which will distribute the load further to new slaves. The slave node is only marked once as a saturated node and it cannot be changed afterwards, even if the ramp decreases. The reason is that the slave node does not terminate the UserSimulator processes when the ramp is decreasing, it suspends the processes. The processes are resumed if the ramp starts increasing again. This approach is more beneficial than creating and destroying the processes, because it requires less processing overhead.

The master node waits for two conditions to happen: 1) the last active slave node started has reported back that it has been saturated and is unable to generate

more users; 2) test duration is completed. In the first case, the master will select the next idle slave and starts the load generation. In the second case, all running virtual users on the slave the nodes perform the *exit* action and the *Load Generator* module shuts down itself. The slave nodes report back the KPI values monitored over the test period to the master. Then, it is the job of the master node to aggregate the test result data from all active slave nodes and send a kill signal to all idle slaves. Afterwards, *Test Report Creator* module at the master node is initiated to create the test report.

4.3 Test Reporting

The Test Report Creator module gathers the test result data from all the slaves. It uses a naming convention to the data depending upon the slave from where it originates. The Test Report Creator module aggregates multiple instances of data in a way that it can be interpreted as a test result data generated by a single node with the combined computational power of all participated slave nodes.

In addition to KPI information from slaves, the master has another source of information, resource utilization log of SUT. The log file is used to compose the graphs that represent the resources utilizations trends over the test period. These graphs can be used in a comparison with the KPI graphs to investigate the different reasons for poor performance. For instance, which resources are extensively being used and thus cause delays in server response, and how effectively the resources are being utilized.

After the data assembly, the *Test Report Creator* module starts rendering a test report based on the assembled data. In the report, most of the information is summarized with the help of different visual elements (i.e. tables and graphs), to enhance the report readability and interpretability.

In addition to the HTML test report, the MBPeT tool produces a trace file (shown in Listing 3) containing all the traces executed during the load generation. It sorts the traces in descending order of their number of executions. This is useful to identify which traces have been executed most frequently.

```
browse,get_auction:157
browse,get_auction,get_bids:148
search,get_auction,get_bids:115
search,get_auction:114
browse,get_auction,get_bids,bid:73
search,get_auction,get_bids,bid:51
browse:44
browse,browse,get_auction,get_bids:29
browse,get_auction,get_bids,bid,get_bids:25
search,get_auction,get_bids,bid,get_bids:23
browse,browse,get_auction:22
search:20
browse,get_auction,get_bids,browse,get_auction,get_bids:16
browse,browse,get_auction,get_bids,bid:15
...
```

Listing 3: Sample trace file

The tool is equipped with an auxiliary script which analyzes the trace file to provide answer to several questions. The script takes two arguments, the trace file and a threshold value, and produces a trace analysis report. The analysis report consists of several sections. One question that the script can solve is *"what is the distribution of different user actions in the generated load?"*. For instance, Listing 4 demonstrates the first section of Trace analysis report, which is generated from the sample trace file (in Listing 3) with the threshold value equals to 0.23. The first line of the report (i.e. *Total traces executed*) indicates the total number of executions of all the unique traces in the trace file. Further, it shows the total number of invocations of all the actions and the individual number of invocations and percentage per action. For example, in Listing 4 *get_auction* action is executed more than other actions, it is invoked 1380 times out of 4798, which is 28.76 % of the total number of invocations.

```
 ############# original statistics
    ##############################
Total traces executed:  1263

TOTAL:  4798
browse      1231        25.6565235515 \%
get_auction       1380        28.7619841601 \%
search      548      11.421425594 \%
bid     440      9.17048770321 \%
get_bids       1199        24.9895789912 \%
```

Listing 4: Trace analysis - Sec 1: Number of Actions

The script can be used to analyze *"which are the most executed traces accounting for a given percentage of the load."*. As such, the second section (Listing 5) of the script estimates the number of actions required according to the given threshold value. It traverses through the list of all traces in descending order, and keep selecting the traces until the total number of actions in the selected traces becomes greater or equal to the required number of actions according to the given threshold value. As it is shown in the Listing 5, the top four traces have been selected because these traces have been executed 534 times and the total number of invocations in these selected traces is equal to 1331 which is close to 1103. The script also reports the total number of invocations of each action in the selected traces.

```
 ############# SELECT TRACES2 ##############################
Actions:  4798 threshold  1103.54 0.23
selecting actions....
Total traces:  534
('browse,get_auction', 157)
('browse,get_auction,get_bids', 148)
('search,get_auction,get_bids', 115)
('search,get_auction', 114)
Selected traces:  4

 create list of actions in traces....

TOTAL:  1331  of 4798  =  27.7407253022 %
browse      305      22.9151014275 \%
get_auction       534      40.1202103681 \%
search      229      17.2051089406 \%
get_bids       263      19.7595792637 \%
```

Listing 5: Trace analysis - Sec 2: Selecting traces

In the third section (Listing 6), the script scales up the selected traces by scaling up the number of times the actions have been executed. First, the script calculates the scaling factor based on the number of time an action has been executed in the selected traces and the total number of time all actions have been executed in all the traces. For example, the total number of actions executed in the selected traces and in all the traces are 1331 and 4798 respectively, as shown in Listings 5. The number of executions of each action in the selected traces is scaled up by using the scaling factor and based on the results the number of executions of each selected traces is amplified, as illustrated in Listing 6. These scaled traces can be used in the other performance testing tools like *JMeter* to generate the same amount of load of all the traces.

```
 ############ SCALING UP ACTIONS...same number of actions
    executed
scaling_factor 3.60480841473  resulting total:  4798.0

Scaled action list
browse      1099        22.9151014275 %
get_auction       1924        40.1202103681 %
search      825      17.2051089406 %
get_bids       948      19.7595792637 %
TOTAL actions:  4796 Original actions:  4798

Scaled trace list
browse,get_auction : 565
browse,get_auction,get_bids : 533
search,get_auction,get_bids : 414
search,get_auction : 410
TOTAL traces:  1922 Original traces:  1263
```

Listing 6: Trace analysis - Sec 3: Scaling up traces based on number of actions

The last section (Listing 7) also scaled up the selected traces but in a different way. Instead of scaling up the number of executions of each action in the selected traces, it scales up the number of executions of the selected traces. It computes the scaling factor based on the total number of executions of the selected traces and of all the traces.

```
 ############ SCALING UP TRACES...same number of users/
    traces executed
scaling_factor 2.36516853933  resulting total actions:
   3148.03932584
Total traces (target):  1263

Scaled trace list
browse,get_auction : 371
browse,get_auction,get_bids : 350
search,get_auction,get_bids : 271
search,get_auction : 269
TOTAL traces:  1261 Original traces:  1263

Scaled action list
browse      721      22.9151014275 %
get_auction       1263        40.1202103681 %
search      541      17.2051089406 %
get_bids       622      19.7595792637 %
TOTAL actions:  3147 Original actions:  4798
```

Listing 7: Trace analysis - Sec 3:Scaling up traces based on number of traces

5 Implementation

MBPeT tool has been developed using the Python[2] language. Python has many built-in libraries for network programming which allowed implementing the distributed architecture seamlessly. Moreover, in order to make our tool more productive and autonomous, we have also employed the following third-party libraries to carry out the different tasks.

- pydot: This library act as interface between Python and dot models. The *Model Parser* and the *Model Validator* modules utilize this library to parse the dot model.
- matplotlib: It is a very useful library for plotting 2D graphs and other figures like histograms, power spectra, bar charts, etc. It requires few parameters as input to produce quality figures. Test Report Creator module uses this library to generate figures in the test report.
- psutil: This library provides the real-time information about the system resources utilization. Resource Monitor module uses this library to keep track local resource load of slave node.
- paramiko: It is an implementation of SSH protocol for Python. It allows the tool to connect securely with the SUT. After establishing a successful connection, the tool could send commands to the SUT and transfer files over an encrypted session.

6 Experiments and Evaluation

In order to demonstrate the efficiency of the tool, we have performed different experiments. We used our tool to evaluate the performance of an auctioning web service running on the Apache web server. The host machine features 8-cores CPU, 16 GB of memory, 1 GB Ethernet, 7200 rpm hard drive and Fedora 16 operating system.

6.1 Case study: 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.

[2]http://www.python.org/

The web application is implemented in Python language using the Django[3] web-framework. In addition to HTML pages, YAAS also has a RESTful [7] 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.

6.1.1 Test data

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.

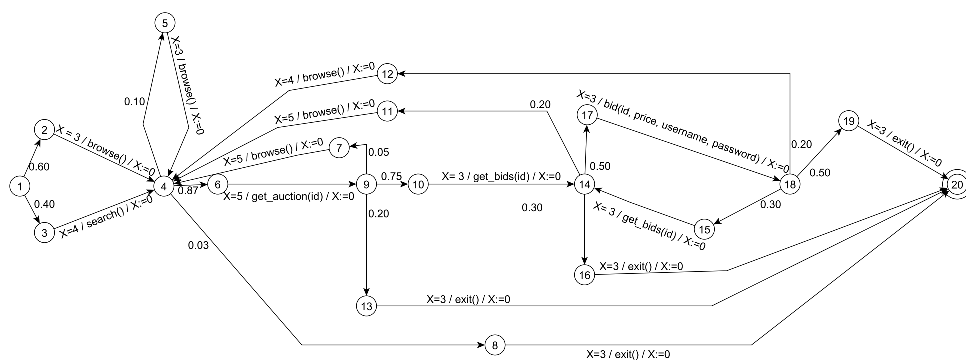


Figure 9: Aggressive User type model

In order to identify the different type of users for the YAAS application, we have used the AWStats[4] 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. For each user type a profile model has been created, the aggressive type (Figure 9) of users describes those users, who make bids more frequently as compared to other types of users. And the passive users (Figure 10) 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 the auctions instead of making any

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

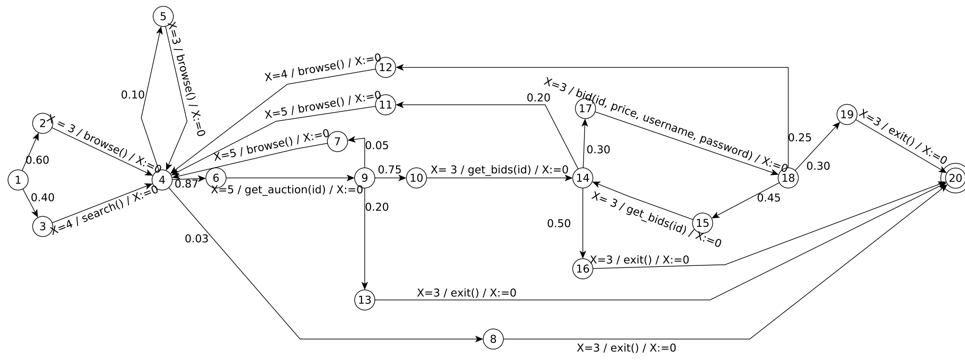


Figure 10: Passive User type model

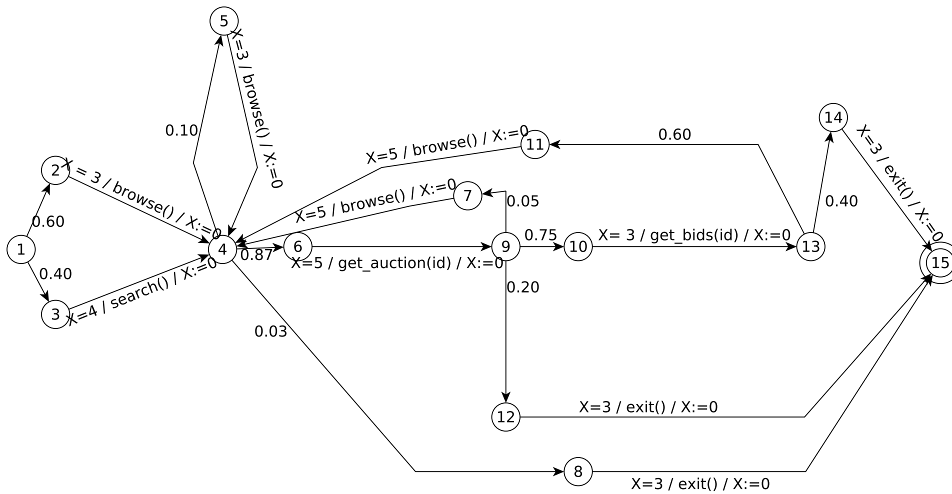


Figure 11: Non-bidder User type model

bids, known as non-bidders (Figure 11). The root model of the YAAS application, shown in Figure 12, 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 12.

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 9, 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

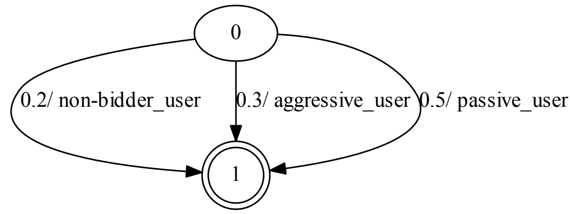


Figure 12: YAAS Root model

the same structure but with different probabilities and distribution of actions.

6.1.2 Test Architecture

A setup of the test architecture can be seen in Figure 13. The server runs as instance of the YAAS application on top of the 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.

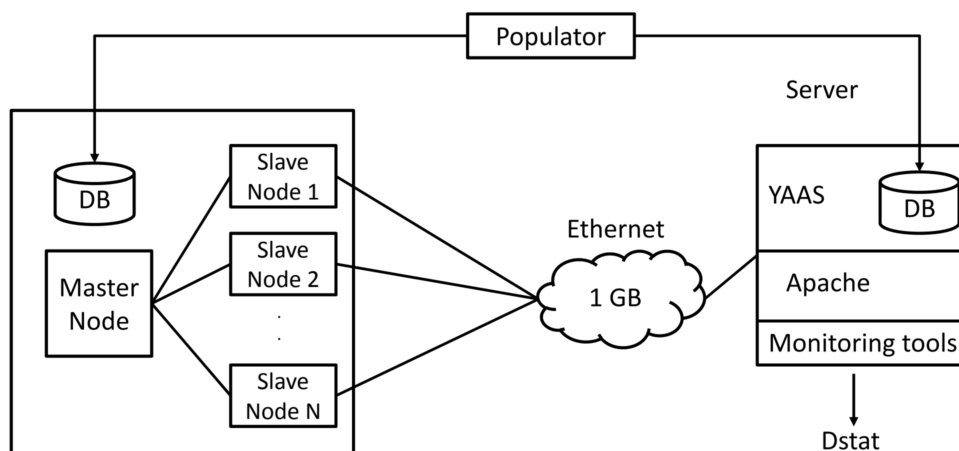


Figure 13: A caption of the test architecture

6.2 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 14, contains the information about the test session including, test started time, test duration, target number of concurrent of users, etc. The *Total 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 12). This section is useful to see the summarized view of the entire test session.

#################### 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 14: Test Report 1 - Section 1: global information

In the second section, 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 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.

In section three (shown in Figure 16) of the test report presents a comparison of the SUT's desired performance against the measured performance. As we had defined the target response time for each action in the test configuration, in this section we could actually observe how many concurrent users were active when

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

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 251 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 16 have breached the target response time except the *BID* action in *NON-BIDDER_USER* column because *non-bidder* users do not bid.

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

The fourth section (Figure 17) presents the stress level on the slave nodes caused by the load generation. This section is useful to observe whether the slave nodes were saturated during the test session or not. If the nodes were saturated then it is possible that slave nodes would not have generated the load at the targeted rate.

In the *Local Resource Usage* section, report portrays the information that had been gathered by the *Resource Monitor* module during the load generation by the slave nodes. Figure 18 and 19 shows the slave local resource utilization trend over the test duration. These figures depict that the utilization of all resources increased as the number of concurrent users increased, however utilization rate

#################### System Stats #######################

Total Disk read bytes: 0.00 Bytes
Average Disk read bytes: 0.00 Bytes/s
Total Disk write bytes: 10.06 MB
Average Disk write bytes: 8.59 KB/s
Total Network sent bytes: 50.51 MB
Average Network sent bytes: 43.10 KB/s
Total Network received bytes: 1.93 GB
Average Network received bytes: 1.64 MB/s
Virtual Memory Usage: 0.0 %
Physical Memory Usage: 19.33 %
Slave 0 CPU 0 Usage: 13.2 %
Slave 0 CPU 1 Usage: 3.07 %
Slave 0 CPU 2 Usage: 0.99 %
Slave 0 CPU 3 Usage: 1.93 %
Slave 0 CPU 4 Usage: 0.15 %
Slave 0 CPU 5 Usage: 0.1 %
Slave 0 CPU 6 Usage: 0.14 %
Slave 0 CPU 7 Usage: 0.29 %

Figure 17: Test Report 1 - Section 4: Aggregated resource load statistics of slave nodes

of some resources grew more rapidly than the others, for example memory usage increased more steeply than the CPU. Further, the CPU usage remained almost constant over the test duration whereas the memory utilization increased as the concurrent number of users ramped up.

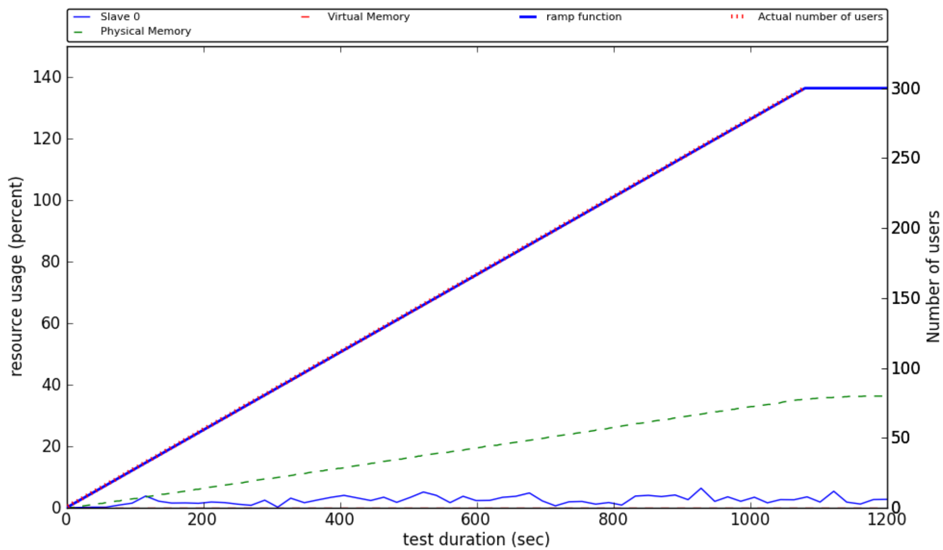


Figure 18: Test Report 1 - Slave physical and virtual memory, and CPU utilization

Figures 20 and 21 display the resource load at the SUT during load generation. These graphs are very useful to identify which resources are being utilized more

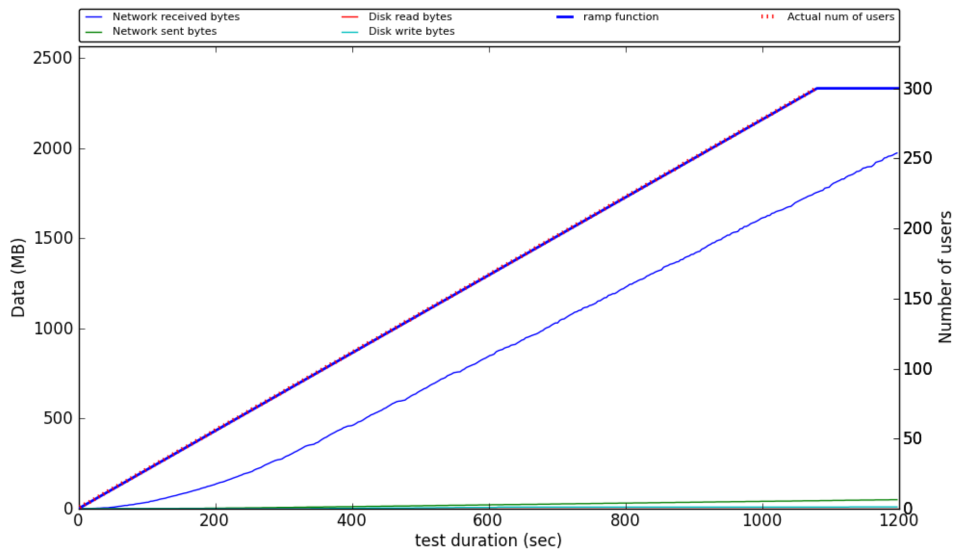


Figure 19: Test Report 1 - Slave network and disk utilization

than the others and limiting the performance of SUT. For instance, it can be seen from Figure 20 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.

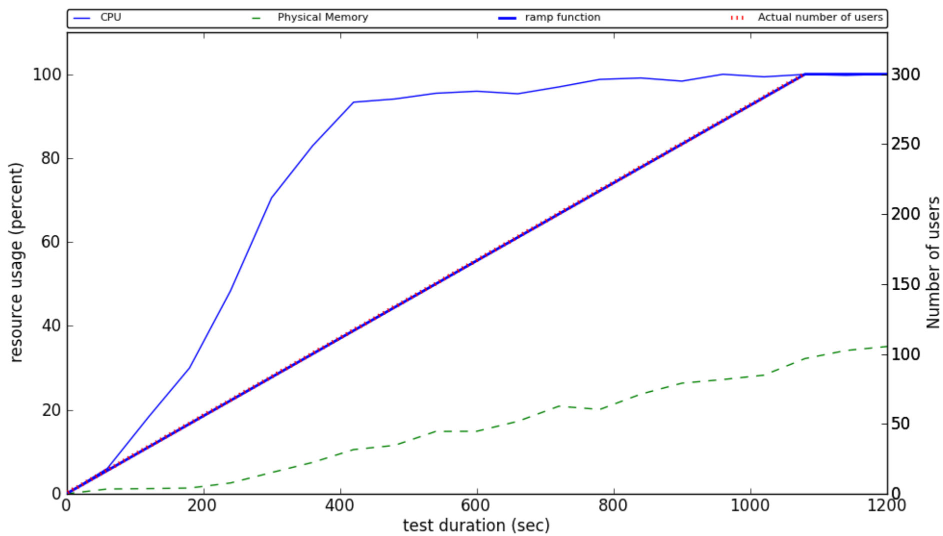


Figure 20: Test Report 1 - SUT CPU and memory utilization

Figures 22, 23 and 24 illustrate that the response time of each action per user type increases proportionally to the number of concurrent users. These figures

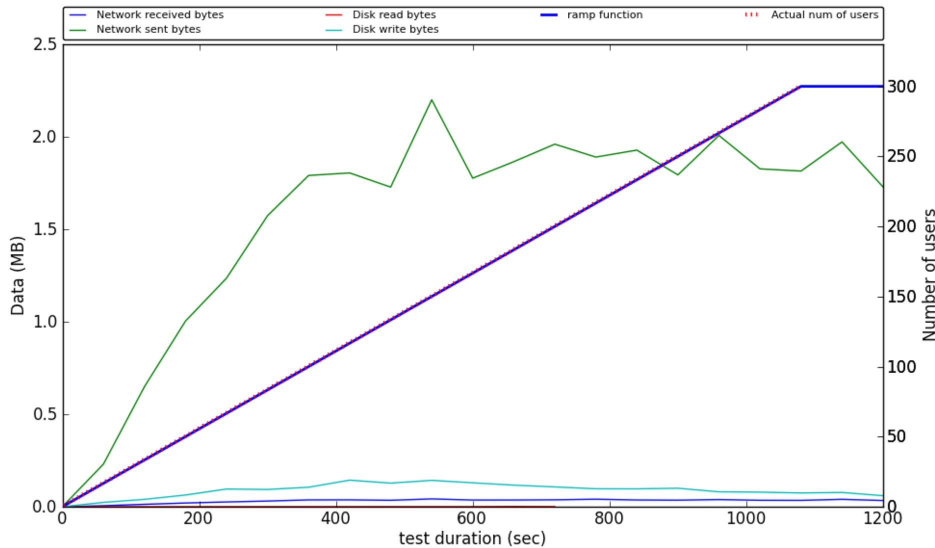


Figure 21: Test Report 1 - SUT network and disk utilization

also point out which actions response time is increasing much faster than the other actions and requires optimization. For example the response time of *BID(ID, PRICE, USERNAME, PASSWORD)* action in *aggressive* and *passive* types of users, 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 *SQLite*[5] 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 five of the test report provides miscellaneous information about the test session. For instance, *Error Rate* in Figure 25 indicates the average error rate. It is noticed that the SUT usually returns an error with the status code 500 when the request rate is high enough for the SUT to respond to all requests correctly. It might be because if the server receives more requests than it can handle, it denies the new connection requests with an error response. For example, the first erroneous response was recorded at 520 seconds (according to Figure 26) and at that time tool was generating the load at the maximum rate that is 1600 actions/seconds, shown in Figure 27. Similarly, Figure 26 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

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

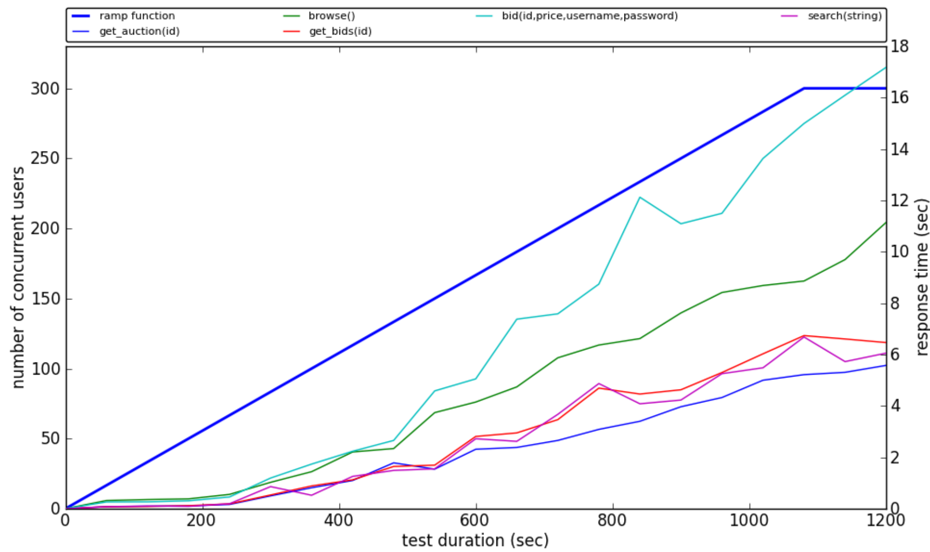


Figure 22: Test Report 1 - Response time of aggressive user type per action

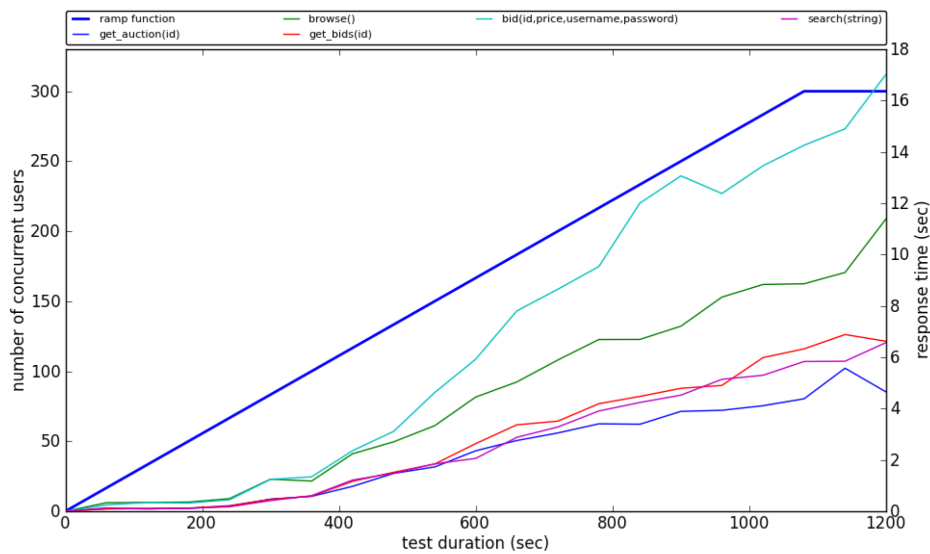


Figure 23: Test Report 1 - Response time of passive user type per action

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 [9]. This could also delay the processing of a write operation and thus cause a larger response time.

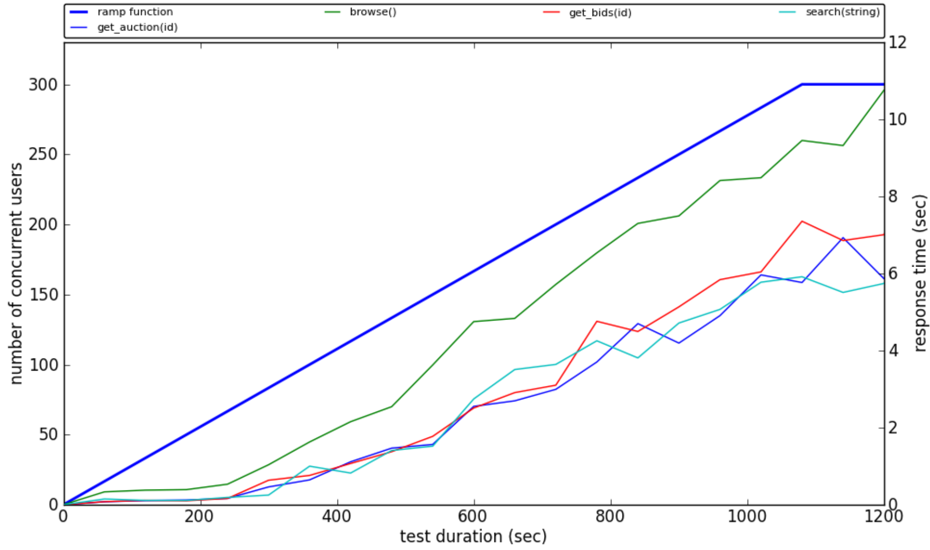


Figure 24: Test Report 1 - Response time of non-bidder user type per action

Misc

Average request size: 1.56 KB
Average user life time: 38.15 sec
Average user think time: 3
Error Rate: 8.4%
This test run ended at: 2013-07-01 17:14:47

Figure 25: Test Report 1 - Section 5: Miscellaneous information

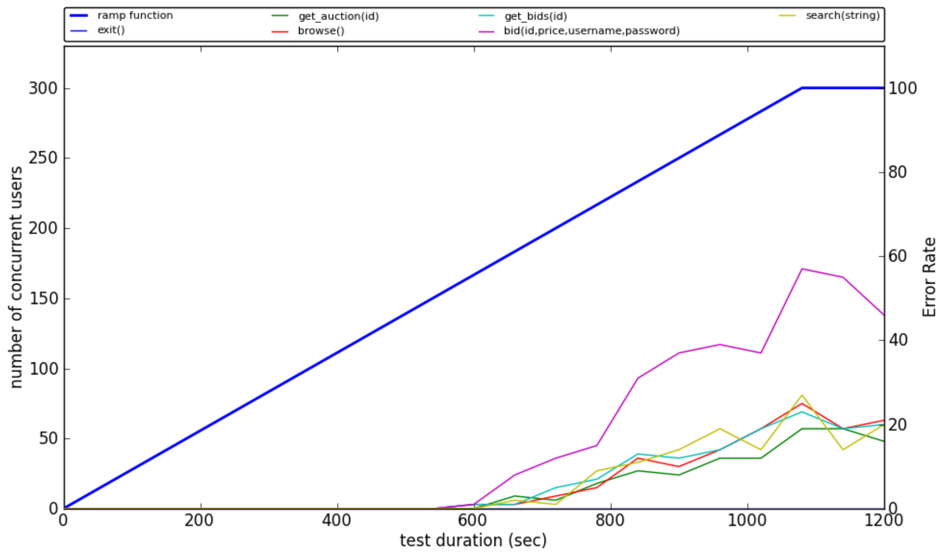


Figure 26: Test Report 1 - Error rate

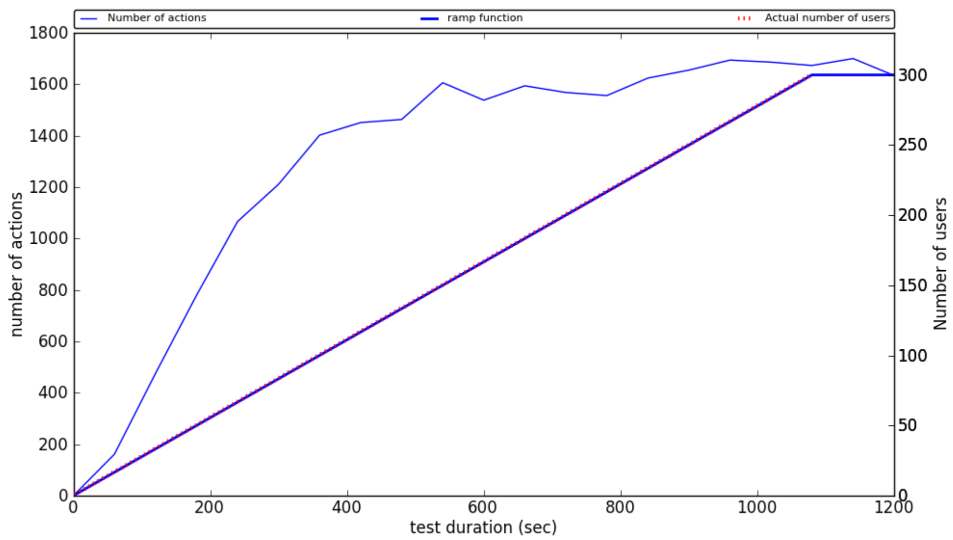


Figure 27: Test Report 1 - Average number of actions

6.3 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 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* [6]. The main motivating factor of using the PostgreSQL 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 [8].

#################### 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 28: Test Report 2 - Section 1: global information

In the section 1 of Test report 2 (Figure 28) 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 30), the average and maximum response time of all action decreased by almost 47%. Moreover, the error rate section (Figure 29) depicts that there was no error until the number of concurrent users was below 182, that is 21% more users than the last experiment.

Figure 31 shows that the response time of aggressive type of users is decreased by 50% approximately in comparison with the previous experiment in Figure 22. In summary, all of these indicators suggest significant improvement in the performance of SUT.

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

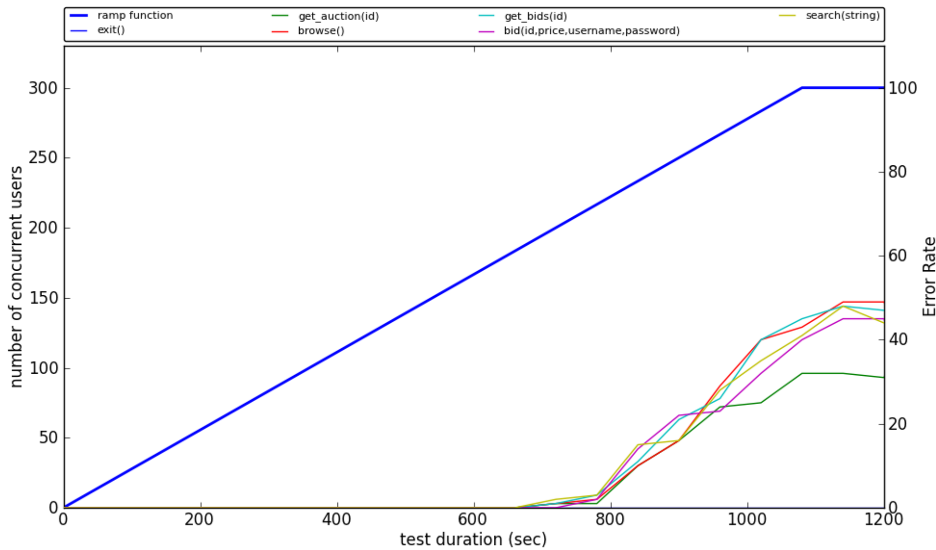


Figure 29: Test Report 2 - Error rate

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

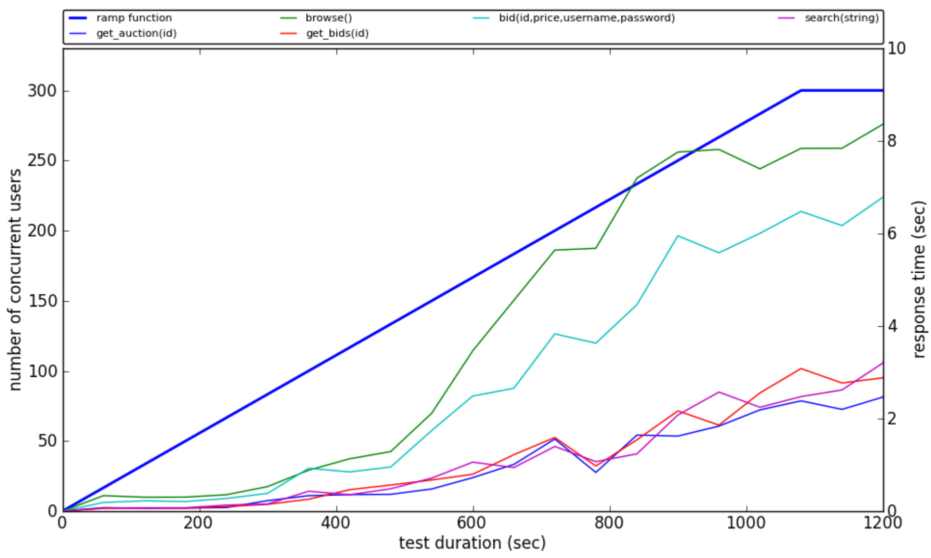


Figure 31: Test Report 2 - Response time of aggressive user type per action

6.4 Experiment 3

The primary objective of running this experiment was to discover eventual memory leaks in the SUT. The experiment ran for 3 hours. The concurrent number of users was set to 150 because it can be observed from the previous experiment that the SUT does not return errors as long as the number of concurrent users remains under 182.

The Figure 32 shows that the response time of all the actions performed by the aggressive type of users, increased very rapidly during the ramping up period. Similarly, Figure 33 shows that the CPU consumption was over 90 % at 1900 seconds but as the number of concurrent users stopped ramping up, the response time and SUT's resource load started to decline steeply. After 6000 seconds since the experiment started, the response time of all the actions remained stable for the rest of the test session. This phenomenon could be explained by considering the caching feature of the SUT. Once the web server had cached the data which was mostly accessed by the virtual users, the response time of the actions started to stabilize.

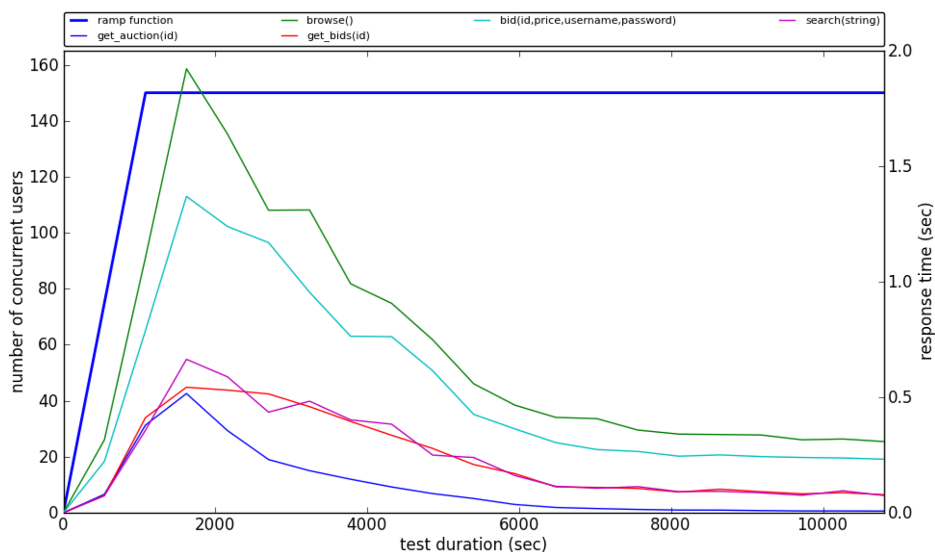


Figure 32: Test Report 3 - Response time of aggressive user type per action

Moreover, it is noticed that the CPU and the memory utilization at the slave node does not depend upon the test duration. Figure 34 shows that the resource load at the slave node increased only during the ramp up period and remained stable afterwards. For instance, when the slave node was generating the load with the maximum number of concurrent users (i.e. 150), the memory utilization remained stable at the 20%. The amount of memory used by one VM depends on three things: First of all the python interpreter itself. The size of an empty python interpreter draws about 4.5 MB of memory. Second, the amount of libraries im-

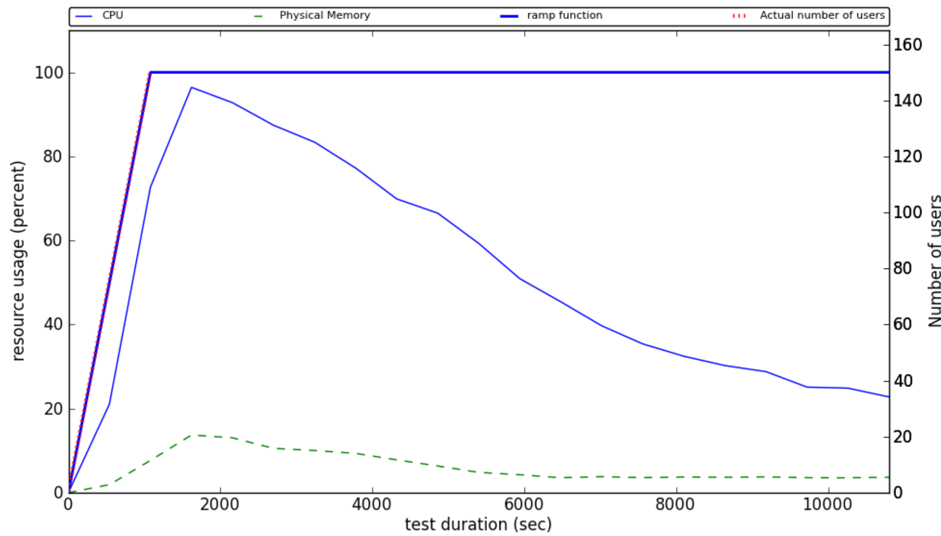


Figure 33: Test Report 3 - SUT CPU and memory utilization

ported for running the code. This can vary drastically depending on the number and size of the libraries that are imported. To keep the memory impact at a minimum it is recommended that the programmer only import the classes that are needed and not the library as a whole. The final factor to impact on memory consumption is time. The longer the test takes more memory will be consumed due to the fact that the python interpreter will be filled up with interpreter objects. The host machine used in this experiment for the slave node featured with 16 gigabytes of physical memory and it is calculated that each virtual user instance on the slave node is consuming almost 21.8 megabytes of the physical memory. Therefore, it can be estimated that the host machine for the slave node can simulate up to 750 virtual users. In this case the test was run for 3 hours which led to the high memory impact of the VUs. In other experiment where the adapter code has been optimized the memory impact has roughly been halved and was around 10 MB per VU.

6.5 Experiment 4

In order to validate our approach and tool chain, we decided to compare our generated workload with a similar workload generated by the JMeter [11] tool using static scripts. We choose JMeter because it is a mature tool and has been available to users for over 10 years. We wanted to find out if the MBPeT tool is able to the generate syntectic workload at the same rate as JMeter. We divided the experiment into two different benchmarks. In the first benchmark we ran simple atomic actions and sequences with both MBPeT as well as with JMeter and compared the results. In the second benchmark, we ran bit more complex load and compared

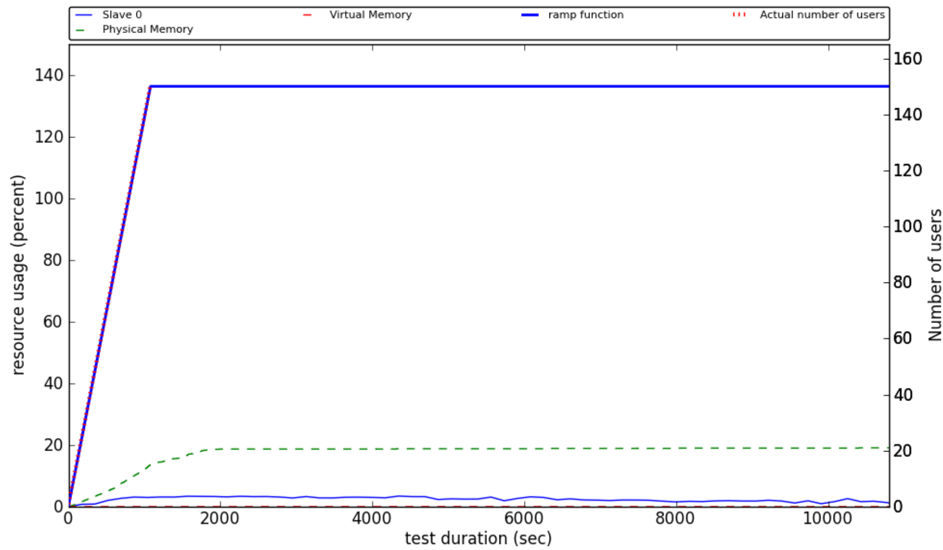


Figure 34: Test Report 3 - Slave physical and virtual memory, and CPU utilization

the results. Both JMeter and MBPeT were run on the same host configuration and tested against the same application as mentioned in Section 6.1.

6.5.1 Benchmark 1

In our first benchmark, we ran simple atomic actions and test sequences repeatedly for 5 min with 100 users running them in parallel. We created a new model for every single test. Every test was run three times and an average was computed. Between each action a uniform *think time* of 3 seconds was used. For example, for the sequence *browse, get_auction* in Table 1 we would construct a PTA with only one possible trace and with 3 second *think time* in between the action. We would do the same in JMeter. Table 1 shows the result of our comparison. The table shows that e.g., action *browse* was executed 4196 times when testing the YAAS system with JMeter, compared to 4510 times when testing with MBPeT. This corresponds to a 7,48 percent difference in speed. The column *Nr of Traces* refers to how many times a particular trace has been executed. Similarly, *Nr of Actions* refer to the number of actions that was executed. If a trace contains only one action, then the two values are the same.

We note that we did not benchmark the *bid* action separately since it requires that the user first searches for an auction item, then gets the bids for that particular auction, and then places a higher bid. We also note that there is a significant difference in the throughput when the *browse* action is executed compared to the *search* action. This is because the database contained over 1000 auctions, and the *browse* action triggered to system to return all actions in the database, while the *search* action only returned one result.

	JMeter				MBPeT				
Sequence of Actions	Nr of Traces	Nr of Actions	Throughput	Request Rate	Nr of Traces	Nr of Actions	Throughput	Request Rate	Percent
browse	4196	4196	3,7 MB/s	14,0 req/sec	4510	4510	4,0 MB/s	15,3 req/sec	7,48 %
search	9128	9128	17,1 KB/s	30,4 req/sec	9520	9520	26,6 KB/s	33,2 req/sec	4,29 %
get_action	9190	9190	17,2 KB/s	30,6 req/sec	9519	9519	28,4 KB/s	33,2 req/sec	3,58 %
get_bids	9105	9105	37,2 KB/s	30,4 req/sec	9463	9463	26,4 KB/s	33,0 req/sec	3,93 %
browse,get_action	3503	7086	3,1 MB/s	23,6 req/sec	3776	7606	3,5 MB/s	26,4 req/sec	7,79 %
search,get_action	4564	9178	25,3 KB/s	30,6 req/sec	4607	9282	110,0 KB/s	32,3 req/sec	0,94 %
browse,get_action,get_bids	2659	8098	2,4 MB/s	27,0 req/sec	2698	8194	2,5 MB/s	28,3 req/sec	1,47 %
search,get_action,get_bids	3017	9153	31,0 KB/s	30,5 req/sec	3044	9240	96,1 KB/s	32,2 req/sec	0,89 %
browse,get_action,get_bids,bid	2003	8162	1,85 MB/s	27,2 req/sec	2077	8308	1,89 MB/s	27,7 req/sec	1,78 %
search,get_action,get_bids,bid	2229	9062	40,5 KB/s	30,1 req/sec	2340	9352	77,4 KB/s	31,2 req/sec	3,20 %

Table 1: Benchmark 1 with JMeter: Running 100 users in parallel for 5 minutes. A uniform think time of 3 seconds between actions was used

6.5.2 Benchmark 2

In our second benchmark, we wanted to find out how our approach compares to JMeter when the workload is more complex. In this experiment, we ran 5 different test sequences with 100 concurrent users and a test session of 20 minutes with a ramp-up period of 120 seconds. Between each action a uniform *think time* of 3 seconds was used. The tests were run three times and an average was computed. The test sequences used in this experiment was selected based on a previous test run of the YAAS application, where the top 5 most executed sequences were selected for this experiment. We constructed a model containing the 5 selected trace in MBPeT and did the same in JMeter. Figure 35 shows a caption of the selected sequences implemented in JMeter. From the figure one can see that each sequence has been implemented as a separated Thread Group. As the name implies, a thread group is controller that controls the number of threads JMeter will use to execute the tests. For example, the sequence *browse, get_action* contains two containers of actions (Browse Actions and Get_Action). Each action or container contains a http request, a pre- or post-processor for extracting something from the response, and a constant timer set on 3 seconds. This sequences is controlled by a thread group called *browse, get_action*. That tread group was set to run 22 users in parallel and have a ramp-up period of 120 second.

The test sequences and the distribution between them can be seen in Table 2. The table shows that the JMeter tested on average a total of 13343 test sequences while the MBPeT tested on average a total of 13483 test sequences. This corresponds to a 1 percent speed advantage for the MBPeT tool. We note that there is a difference in the percentages in which test sequences where executed. This is because the MBPeT tool uses probabilistic models, from which load is generated, and due to the randomness in the models it is difficult to control the exact distribution between test sequences.

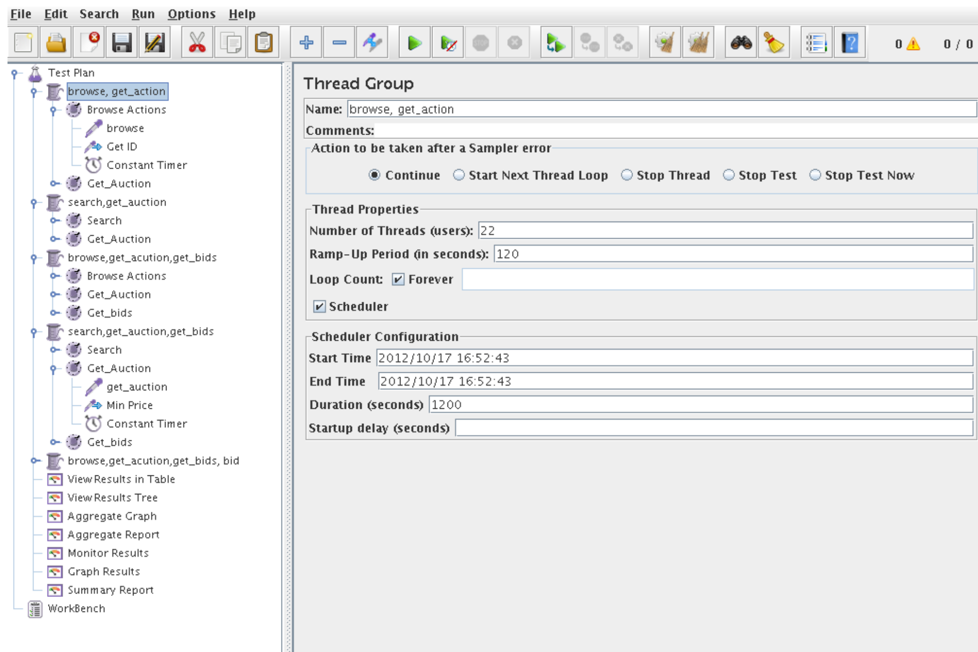


Figure 35: JMeter

Traces	JMeter		MBPeT	
	Nr of Traces	Percent of Traces	Nr of Traces	Percent of Traces
browse,get_auction	3706	27,77 %	3682	27,31 %
search,get_auction	3101	23,24 %	3218	22,87 %
browse,get_auction,get_bids	2427	18,19 %	2447	18,15 %
search,get_auction,get_bids	2306	17,28 %	2305	17,10 %
browse,get_auction,get_bids,bid	1803	13,51 %	1831	13,58 %
total	13343	100%	13483	100%

Table 2: Benchmark 2 with JMeter: Running 5 different traces with 100 users in parallel for 20 minutes. A uniform think time of 3 seconds between actions was used

6.6 Experiment 5

In this experiment, we wanted to find out if the randomness in the PTA models affect the response times all. To be more precise, we wanted to find you if PTA models can show worse performance compared to static scripts. In this experiment we excluded JMeter from the experiment since we do not have an exact control over how JMeter generated the load and only focus on static execution vs random execution. We decided to run an experiment where we compared the result of a normal test run with MBPeT when probabilistic execution is turn on with the result of an other test run where have a more static setup.

To achieve this we ran a simple performance test with 150 parallel users everyone using the model found in Figure 9 for 30 minutes. We then studied the most executed traces in the trace file and the average response time of each action.

Listing 8 shows a caption of the 12 most executed traces found the trace file.

```
aggressive_user::
browse,get_auction,get_bids:1351
browse,get_auction:1210
search,get_auction,get_bids:903
search,get_auction:833
browse,get_auction,get_bids,bid:681
search,get_auction,get_bids,bid:467
browse,get_auction,get_bids,bid,get_bids:328
search,get_auction,get_bids,bid,get_bids:225
browse:218
browse,get_auction,get_bids,browse,get_auction:186
browse,get_auction,get_bids,browse,get_auction,get_bids:162
browse,browse,get_auction,get_bids:151
browse,get_auction,get_bids,bid,get_bids,bid:149
browse,get_auction,get_bids,bid,browse,get_auction,get_bids
    :147
```

Listing 8: Caption of the trace file showing the 12 most executed traces

We then selected the 5 most executed traces in the trace file and had the trace file analysis tool, described in Section 4.3, to scale up the number of traces accordingly to match the initial load put on the system. We then created a PTA model with the 5 most executed traces with the individual probabilities suggested by the trace analysis tool and had MBPeT execute the model. Figure 36 shows the PTA that we constructed. We had MBPeT run the model for 150 parallel users for 30 minutes and we again observed the average response time of individual actions. Table 3 shows the average response time of the individual actions for each test run. For example, the average response time of action *browse* drops from 1,55 seconds to 0,98 when having a more static execution of the PTA model. In both of the test the same amount of actions was executed. We note that the difference of 0,57 sec might at a first glance not seem as much but the difference in percent is roughly 58%. The biggest difference was as much as 84%. This test showed that by adding a bit of randomness to the test sequences and letting the VUs choose actions based on a probabilistic choice, the average response time can increase significantly. We believe that this is due to the order of actions being executed. Certain caching mechanisms of the SUT might be triggered which in turn might affect the results if static scripts are used and actions are always executed in the same order.

	Test 1	Test 2	
Action	response time (sec)	response time (sec)	Increase
browse	1,55	0,98	58,16%
search	0,50	0,34	47,06%
get_auction	0,35	0,19	84,21%
get_bids	0,52	0,36	44,44%
bid	1,19	0,77	54,55%

Table 3: Differences in average response time between MBPeT in normal mode and forced mode

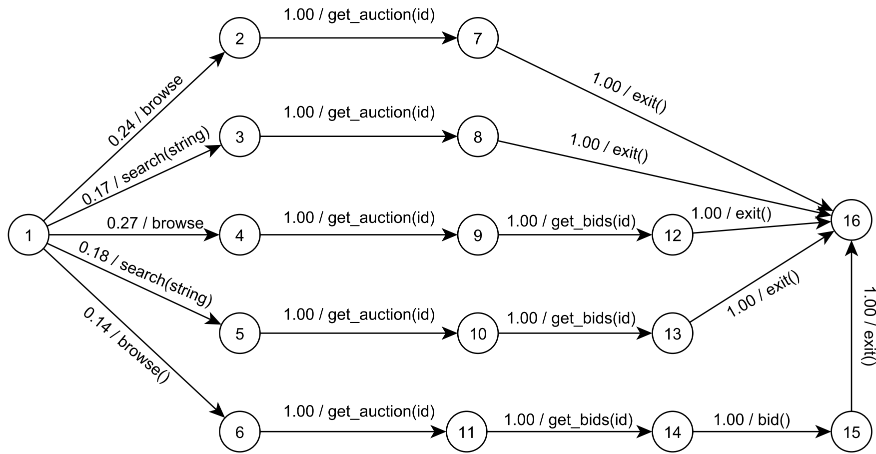


Figure 36: PTA model of the reduced traces

7 Conclusions

In this document, we have presented our tool, MBPeT. The tool receives PTA models and Test Configuration from user, generates synthetic workload for the SUT. The Test configuration defines the test environment and other parameters. And the models specify the probabilistic distribution of actions and different types of users. The tool has distributed architecture where one master node controls several slave nodes in parallel. This makes the tool high scalable and capable of generating load at high rate. All slave nodes and SUT node monitor their own local resource load and send the information back to the master. Based on the information of resource utilization at nodes and test simulation data from slave nodes, master node renders a graphical test report.

The slave nodes store the values of different KPIs during the test generation e.g. response time, error rate. Later from these KPIs, we could observe the performance of SUT under the given load and the level of the load at that particular moment when KPIs crossed the target threshold.

For future development, we have planned several features to incorporate in our tool. By allowing the user to add more information in the model, so that the tool can generate synthetic workload closer to the real workload. In addition to defining a target KPIs value for an action, user would also be able to specify the target KPIs value for a particular action sequence. Further, the load generation process can be further optimized, so that the slave node can generate workload at high rate while not consuming lot of resources. Moreover, instead of having a command line interface, we are planning to provide a fully featured GUI dashboard to the user. The advantage of having a GUI interface is that it provides a standard and convenient method for performing a given task rather than creating a set of commands for each operation. It would also allow the user to monitor the

status of different slave nodes during load generation process.

References

[1] Ashlish Jolly. Historical Perspective in Optimising Software Testing Efforts. http://www.indianmba.com/Faculty_Column/FC139/fc139.html, February 2013.

[2] E. Gansner, E. Koutsofios, and S. North. Drawing draphs with dot. Documentation, January 2006.

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

[4] HP. HP LoadRunner. http://www8.hp.com/us/en/software-solutions/software.html?compURI=1175451#.URz7wqWou8E, February 2013.

[5] ITEA 2. ITEA 2 D-mint project result leaflet: Model-based testing cuts development costs. http://www.itea2.org/project/result/download/result/5519?file=06014_D_MINT_Project_Leaflet_results_oct_10.pdf, February 2013.

[6] M. Kwiatkowska, G. Norman, D. Parker, and J. Sproston. Performance analysis of probabilistic timed automata using digital clocks. *Formal Methods in System Design*, 29:33–78, 2006.

[7] L. Richardson and S. Ruby. RESTful web services. O'Reilly Media, May 2007.

[8] PostgreSQL. Concurrency control. http://www.postgresql.org/docs/9.1/static/mvcc-intro.html. retrieved: March, 2013.

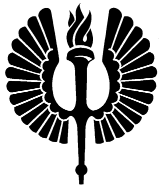
[9] SQLite. File locking and concurrency in sqlite version 3. http://www.sqlite.org/lockingv3.html. retrieved: March, 2013.

[10] Sun. Faban Harness and Benchmark Framework. http://java.net/projects/faban/, February 2013.

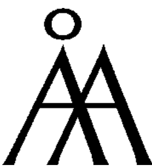
[11] The Apache Software Foundation. Apache JMeter. http://jmeter.apache.org/. Retrieved: October, 2012.

TURKU CENTRE *for* COMPUTER SCIENCE

Joukahaisenkatu 3-5 B, 20520 Turku, Finland | www.tucs.fi



University of Turku
- Department of Information Technology
- Department of Mathematics



Åbo Akademi University
- Department of Computer Science
- Institute for Advanced Management Systems Research

Turku School of Economics and Business Administration
- Institute of Information Systems Sciences

ISBN 978-952-12-2847-6
ISSN 1239-1891