MBPeT: A Model-Based Performance Testing Tool

Fredrik Abbors, Tanwir Ahmad, Dragoş Truşcan, Ivan Porres
Department of Information Technologies, Åbo Akademi University
Joukahaisenkatu 3-5 A, 20520, Turku, Finland
Email: {Fredrik.Abbors, Tanwir.Ahmad, Dragos.Truscan, Ivan.Porres}@abo.fi

***Abstract*—In recent years, cloud computing has become increasingly common. Verifying that applications deployed in the cloud meet their performance requirements is not simple. There are three different techniques for performance evaluation: analytical modeling, simulation, and measurement. While analytical modeling and simulation are good techniques for getting an early performance estimation, they rely on an abstract representation of the system and leave out details related for instance to the system configuration. Such details are problematic to model or simulate, however they can be the source of the bottlenecks in the deployed system. In this paper, we present a model-based performance testing tool that measures the performance on web applications and services using the measurement technique. The tool uses models to generate workload which is then applied to the system in real-time and it measures different performance indicators. The models are defined using probabilistic timed automata and they describe how different user types interact with the system. We describe how load is generated from the models and the features of the tool. The utility of the tool is demonstrated by applying to a WebDav case study.**

***Keywords*-Load Generation. Model-Based Performance Testing. Monitoring. Probabilistic Timed Automata. Models.**

I. Introduction

With the recent advancements in cloud computing, we constantly see more software applications being deployed on the web. This opens up a broader window to reach out to new users. As traffic increases, the overall quality of such applications becomes an even more important factor, since most of the processing is done on the server side. Evaluating that these kinds of systems meet the performance requirements is no longer a trivial task. High response times, technical issues, and display problems can ultimately have a negative impact on the customer satisfaction and ultimately the profitability of the company. As a result, effective performance testing tools and methods are essential for verifying that systems meet their performance requirements.

The idea behind performance testing is to validate the system under test in terms of its responsiveness, stability, and resource utilization when the system is put under certain synthetic workload in a controlled environment. The idea behind the synthetic workload [1] is that it should imitate the expected workload [2] as closely as possible, once the system is in operational use. Otherwise it is not possible to draw any reliable conclusions from the test results.

Jain [3] suggests three different techniques for performance evaluation: analytical modeling, simulation, and measurement. While analytical modeling and simulation are good techniques for getting early performance estimation, they rely on an abstract representation of the system and leave out details related to the system configuration. Such details are problematic to model or simulate, however, they can be the source of the bottlenecks in the system. With the measurement technique one has to wait until the system is ready for testing while with the two former techniques one can start testing while the system is being developed.

Traditionally, performance tests usually last for hours, or even days, and only test a predefined number of prerecorded scenarios that are executed in parallel against the system under test (SUT). The major drawback with this approach is that it certain inputs that the system will face might be left untested. Therefore, we suggest the use of models that describes how the virtual users (VUs) interact with the system and a probabilistic distribution between actions. The synthetic workload is then generated from these models by letting virtual users execute these models.

In this paper, we present a tool that evaluates the performance of a system. The main contribution of this work is that the load applied to the system is generated in real-time from models, specified using Probabilistic Timed Automata (PTA). A tool designed in-house is used to generate the load and monitor different performance indicators.

We use our tool to answer the following questions about the system under test:

- What are the values of different Key Performance Indicators (KPIs) of the system under a given load? For instance, what are the mean and max response times and throughput for a given number of concurrent users?
- How many concurrent users of given types does the system support before its KPIs degrade beyond a given threshold?

The rest of the paper is structured as follows: In Section II we discuss the related work. In Section III we give an overview of challenges with load generation and in Section IV we present our tool. Section VI presents a case study and a series of experiments using our approach. Finally, in Section VII, we present our conclusions and we discuss future work.

II. Related Work

There exist a plethora of commercial performance testing and load generation tools. However, most of them generate load from static scripts or pre-recorded scenarios that are scripted and executed in batches. In this section, we have focused our attention on tools that use models as input.

Denaro et al. [4] propose a tool for testing the performance of distributed software when the software is built mainly with middleware component technologies, i.e. J2EE or CORBA. The authors claim that most of the overall performance of such a system is determined by the use and configuration of the middleware (e.g. databases). The authors 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. Their tool differs from ours in the sense that they target middleware components only and they make use of stubs for components that are not available during the testing phase.

Barna et al. [5] present a model-based testing tool that tests the performance of different transactional systems. The tool uses an iterative approach to find the workload stress vectors of a system. An adaptive tool framework drives the system along these stress vectors until a performance stress goal is reached. Their tool differs from ours in the sense that they use a model of the system instead of testing against the real system. The system is represented as a two layered queuing model and they use analytical techniques to find a workload mix that will saturate a system resource.

Another similar approach is presented by Shams et al. [6]. There, the authors have developed a tool that generates 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 EFSMs. Their tool outputs traces that can be used in well known load generation tools like *httperf* [7]. Their approach differs from our in the sense that they focus on off-line trace generation while we apply the generated load on-line to the system.

Ruffo et al. [8] have developed a tool called *WALTy*. The tool that generates representative user behavior traces from a set of Customer Behavior Model Graphs (CBMG). The CBMGs are obtained from execution logs of the system and a modified version of *httperf* is used to generate traffic from these traces.

III. Load Generation Challenges

In performance testing, one of the main challenges is the load generation. The reason why load generation is such a challenge is that there are so many ways to get it wrong. For instance, important user types may not have been identified. These important user types might have a significant impact on the performance of the system. Another example is that the users that one is simulating during testing behave differently than users in the real world. This can lead to the fact that the generated load does not conform to the load that real users would put on the system. In other words, for load generation to be successful, one needs to be able to generate load that represent the real user load as closely as possible. Failure to do so, often leads to incorrect decisions regarding the performance of the system.

In real life, users need some time to reflect over the information that they have received. This is what usually is referred to as *think time*. The think time specifies how long the user normally waits before sending a new request to the system. Defining a think time for an action is not always as simple as it might seem. For example, to get really accurate results, one needs to consider the time it takes for a web page to be rendered in the client machine and the time it takes for the user to find a new action. Usually the think-time is different for different actions. Hence, in the load generation process, there need to be a way to define a think time value for each individual action.

Traditionally, load generation has been achieved by defining static scripts or pre-recorded scenarios that are run or played back in batches or certain quantities. Even if the scripts are somewhat parameterized, they do not behave like real life users would do. For performance testing, and especially load generation, to make sense one must allow the virtual user to behave as dynamically as real users.

IV. MBPeT Tool

In our approach towards model-based performance testing, we have developed a tool called *MBPeT*. The tool has essentially three high levels purposes: (1) to generate load according to input parameters and send it to the system, (2) to monitor the key performance indicators (KPIs) and other system resources, and (3) to present the results in a test report. The key performance indicators [9] or the KPIs are quantifiable values that one wants to measure and track. Example of typical KPIs are: response time, mean time between failure, number of concurrent users, throughput, etc.

MBPeT accepts as input a set of models expressed as probabilistic timed automata, the target number of virtual users, a ramp function, duration of the test session, and it will provide a test report describing the measured KPIs.

A. Performance Models

The behavior of virtual users is described with probabilistic timed automata (PTA) [10]. The PTA (see Figure 1) describes a set of locations and a set of transitions that take the automaton from one location to another. A transition can have four different labels: a clock zone, a probability value, an action, and a reset. The clock zone is an integer value describing discrete time. The clock zone specifies how long the PTA waits until firing a transition and, in our case, it is the equivalent of the think time. In the figure below, this is represented with the variable X. It is, however, possible to define more than one clock variable. In case of a branch in the PTA, the transition that is taken is based on its probabilistic value. Consider location *6* in the PTA figure below. One can reach location *7* with a probability of *p5* or reach location *2* with the probability of *p4*. Upon taking a transition, the associated action is being executed against the system. Whenever the action is executed there is a possibility to reset the clock variable. In the PTA below this is represented with *X:= 0*. Every PTA has an end location, depicted by a double circle, which eventually will be reached.

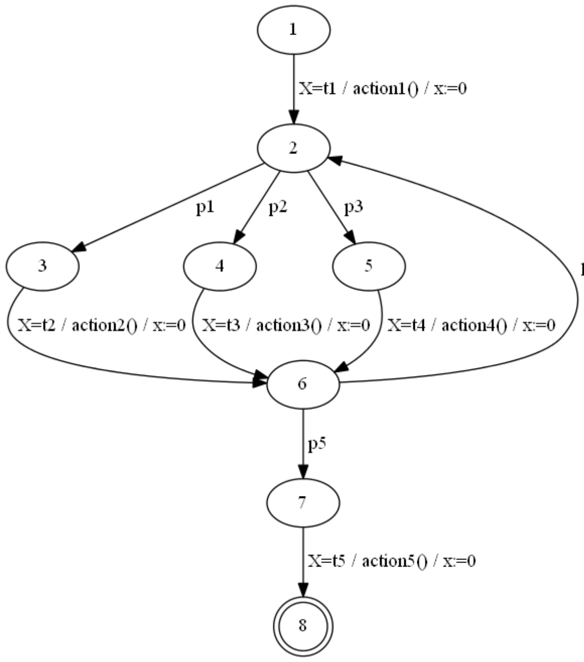

Figure 1. An example of a probabilistic times automata.

We believe that the PTA models are well suited for model-based performance testing and that the probability aspect that the PTA holds is good for describing dynamic user behavior, allowing us to include a certain level of randomness in the load generation process. This is important because we wanted the VUs to mimic real user behavior as closely as possible and real users do not follow static instructions. With the help of the probability values we can make it so that a certain action is more likely to be chosen over another action, whenever the VU encounters a choice in the PTA.

B. Tool architecture

The tool has a distributed architecture: a *master node* controls several *slave nodes* (see Figure 2.) The actual load generation is performed on the slave nodes. The master node just controls the load generation and initializes more slave nodes when needed. Each slave node is responsible for generating load for the VUs. The number of VUs a slave node can support is dependent on its capacity. In addition, each slave monitors its local resource utilization, collects KPIs for the system under test and reports the values to the master node.

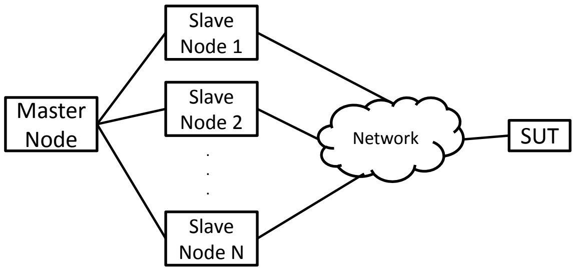

Figure 2. Master-Slave architecture for the MBPeT tool.

The internal architecture of the master node (Figure 3) includes the following components:

The **Model Parser** is responsible for reading the input models and building an internal representation of the model. In addition, it validates the models with respect to basic well-formness rules such as: all locations are connected, there is entry and an exit state, the sum of the probabilities of the transitions originating from a given node equals to 1, etc. We chose the *dot language* as a *plain text* representation for the PTAs. The reason for choosing the dot language is that we wanted to have a simple and lightweight way of representing models that both humans and machines can understand.

The **Core** module is the most important component of the master node. It takes care of reading the input parameters, initializing the test configuration by distributing relevant data to slave nodes, initiating load generation and collecting individual test reports from slaves. The test configuration contains information about the IP-addresses for the slave nodes and the master, the length of the test duration, a ramping function, and the number of concurrent users.

The master node uses two different **Test Databases**: *User DB* and *User-Resource Data Base*. The *User DB* contains data about the users, for instance user name and password, whereas the *User-Resource Data Base* contains information about the resources (documents, pictures, folders, etc) the users have on their own space on the server. The core module is responsible for initializing the data bases before the load generation begins.

The **Test Report Creator** module is in charge of producing an HTML test report once all the slave nodes have reported back to the master node all the gathered data from the test run. The report creator module aggregates the data and computes mean and max of the monitored values and for the specified KPIs.

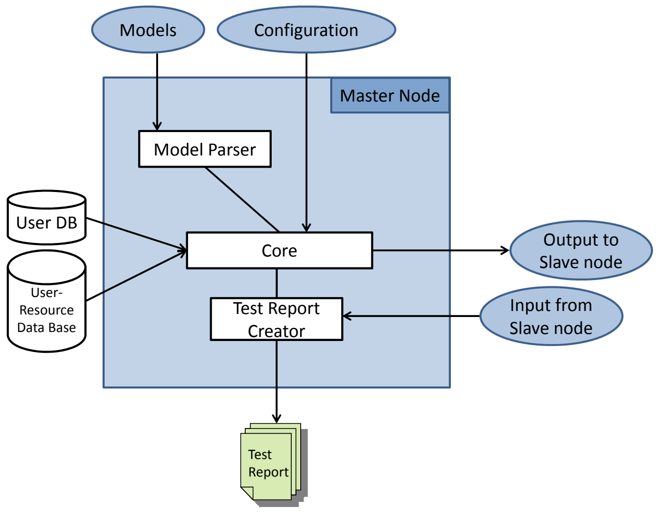

Figure 3. The structure of the master node.

The slave nodes consist of several modules (see Figure 4). The input received from the master node includes: the internal representations of the test models and specific test configuration. All the slave instances have identical configuration and implementation.

The **Load Generator** is in charge of generating load that is sent to the system under test. One instance of the PTA models is used by each VU to generate traces which results in sequences of actions based on specified probabilities and thin times that are sent to the SUT.

The models that we use contain abstract actions and can therefore, as such, not be directly used directly against the SUT. An **Adapter** module is used is to concretize every action into machine readableS format. For example, in the case of a HTTP-based system, a `login()` action needs to be implemented in the adapter code to be sent as a POST request over the HTTP protocol. A second example below shows how an *upload_file(image/jpg)* action on a WebDav server could be translated by an adapter:

```
PUT /webdav/user1/picture.jpg HTTP/1.1
Connection: Keep-Alive
Host: www.examplehost.com
Content-Type: image/jpg
```

A new adapter has to be implemented for each new SUT, however, it is possible to use different libraries in the adapter code to make the adaptation much easier. For example, in the case of the ”login()” action describe above, a standard HTTP library could be used to send the login to the SUT.

During the testing process each slave node monitors, via a **Resource Monitor**, the local resource utilization (CPU, memory, disk, and network) in order to make sure that the slave itself does not saturate and become a bottleneck in the configuration. If, for instance, the CPU utilization goes over a certain threshold, we can not guarantee anymore that the load is generated at the same rate as it should be. The slave node also monitors the response time for each action sent to the SUT and the error rate of these actions.

The **Reporter** module is in charge of putting the measured values together in an organized form and reports them back to the master node at the end of the testing process. The reporter is also responsible for notifying the master node if the local resource utilization threshold has been exceeded.

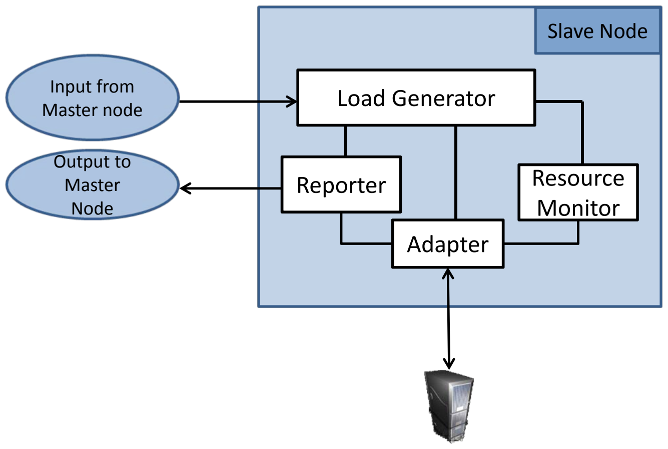

Figure 4. The structure of a slave node.

V. Performance testing process

In our approach, the testing process (see Figure 5) contains three distinct phases: test setup, load generation and test reporting.

A. Test Setup

The test setup phase takes care of initializing the test databases and the configuration of the slave nodes. This is done before the actual test run in order to avoid any negative impact on the bandwidth or resource utilization of the tester.

1) Test database initialization: One of the main challenges in performance testing is providing test data and configuring the system under test with a configuration as close as possible to the production environment [9]. As such, every time before starting the load generation phase, we configure the system under test and the tool with synthetic data using a populator script: on the system side, the script will automatically configure the web server with the given user configuration and if needed with the corresponding user spaces. On the MBPeT tool side the script will populate the user and test data databases with user credentials and corresponding information/files that the user will eventually upload to the server.

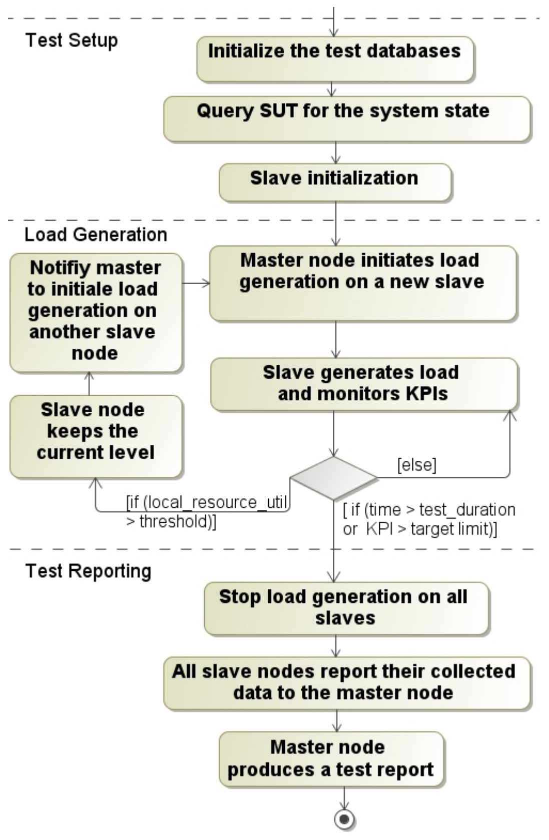
Test Setup
Initialize the test databases
Query SUT for the system state
Slave initialization
Load Generation
Notifiy master to initiale load generation on another slave node
Master node initiates load generation on a new slave
Slave generates load and monitors KPIs
Slave node keeps the current level
[else]
[if (local_resource_util > threshold)]
[if (time > test_duration or KPI > target limit)]
Test Reporting
Stop load generation on all slaves
All slave nodes report their collected data to the master node
Master node produces a test report

Figure 5. Activity diagram describing the load generation process.

2) System state: In certain cases, the current state of the SUT has to be captured before starting the test run. This is useful in stateful systems, in situations where the load generation should start from a given state of the SUT. For this, the master node queries the SUT for the user space resources, and stores the information in the URDB.

3) Slave initialization: When starting the testing process, the number of available slave nodes and their configuration (e.g., IP addresses) is provided to the master node. As mentioned previously, the master node will distribute the load incrementally on the slave nodes, one at the time, until the node saturates. However, all the available slave nodes are initialized with necessary test data and they are just idling.

B. Load generation

Different parameters of the testing process are provided as command line parameters. The tool can be used in *two* modes. The *first* use is to run with a certain target number of concurrent users. The tool will then slowly ramp up to the target number of users, run for the specified test duration and report back the aggregated values. The *second* use, and maybe the more interesting one, is to specify the target KPI value, for instance a target average response time, and let the tool find out the maximum number of concurrent users that the SUT can serve without exceeding the specified threshold.

To generate the load from the models, a few additional things have to be specified. First, one should input the models and a test configuration to the tool. Second, one needs to specify a stopping criterion for when the tool should stop generating load. This stopping criterion can be of two types: a time duration or a given threshold value. If a time duration is given, the tool will generate load based on the given models and target number of concurrent users, and stop generating load after the given amount of time has passed. If threshold values are given for a particular resource, e.g., the CPU, the tool will monitor that resource and ramp up the number of users until the threshold value is reached. All this information is specified in the configuration file. The load generation process will be discussed in more detail in Section VI

C. Test Reporting

When the specified test duration runs out or the tool detects that a certain threshold KPI value has been exceeded, the testing process is aborted and the test run is summarized. Consequently, each slave node reports back to the master node the data that it has collected during the test run. Based on the collected data the master node produces a test report of the test run.

The test report contains information such as, the duration of the test, number of generated users, amount of data sent to the system, response times for different actions, etc. The test report also shows diagrams of how various monitored values changed over time when the user amount was increased, e.g., response time, CPU, and resource utilization.

VI. EXPERIMENTS

In this section, we demonstrate our tool by using it to test the performance of a Webdav [11] file server. Webdav (Web Distributed Authoring and Versioning) is an extension to the HTTP protocol and provides a framework for users to create, change, and move their documents and files stored on web servers. Webdav also maintains the file properties, e.g., author, modification date, file locking, etc. These features facilitate creation and modification of files and documents stored on web servers.

The SUT featured a Linux machine with 8-core CPU, 16GB of memory, 1Gb Ethernet, 7200 rpm hard drive, and Fedora 16 operating system. The file server ran a WebDav installation on top of an Apache web server. The system was configured for 1500 users, each with its own user space. The slave nodes that generated the load had the exact same configuration and were connected via a 1Gb Ethernet network to the SUT. In total we have used 3 slave nodes, but nothing prohibits us from extending this configuration.

By analyzing the Apache server logs of a previous WebDav installation with the AWStats [12] tool, we identified three user type: heavy, medium and light user, respectively, based on the average bandwidth each user type used for

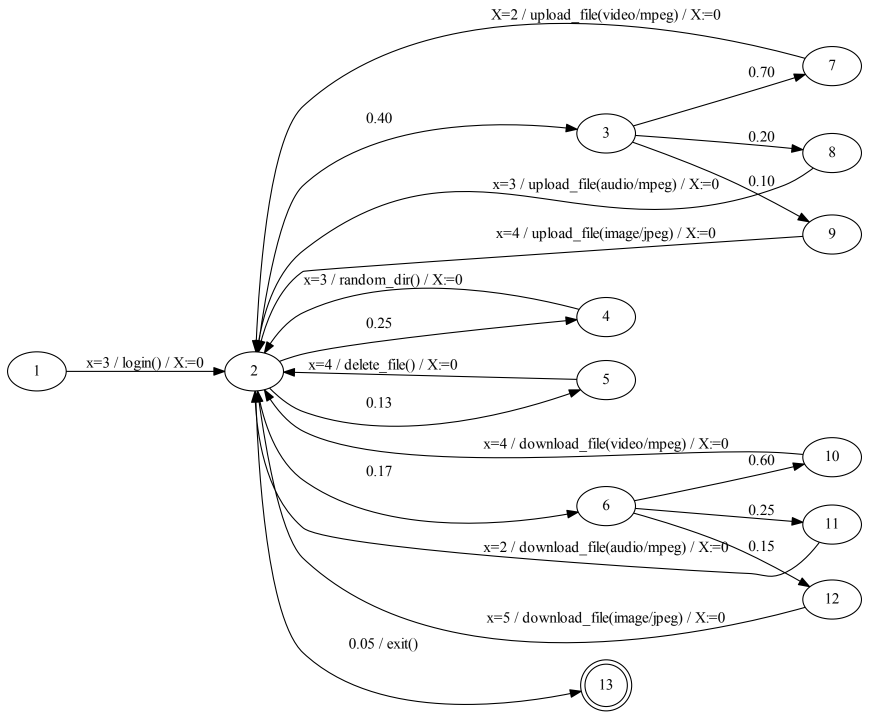
X=2 / upload_file(video/mpeg) / X:=0
0.70
0.40
0.20
x=3 / upload_file(audio/mpeg) / X:=0
0.10
x=4 / upload_file(image/jpeg) / X:=0
x=3 / random_dir() / X:=0
0.25
x=3 / login() / X:=0
x=4 / delete_file() / X:=0
0.13
x=4 / download_file(video/mpeg) / X:=0
0.60
0.17
0.25
x=2 / download_file(audio/mpeg) / X:=0
0.15
x=5 / download_file(image/jpeg) / X:=0
0.05 / exit()
1
2
3
4
5
6
7
8
9
10
11
12
13

Figure 6. Probabilistic time automaton for a *heavy_user*.

transferring. The model depicting the distribution of these users is shown in Figure 7. We have also identified three types of file types (jpeg, mp3, avi) that the users usually transferred having on average file sizes of 3 MB, 5 MB, and 10 MB, respectively.

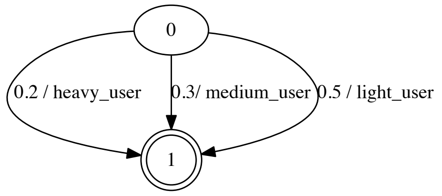

Figure 7. Distribution between different user types.

Figure 6 shows a PTA of the *heavy_user* type in terms of user actions, the probability for those actions, and the think time for each action. Eventually the user will find an *exit()* action and leave the system. Similar models were created for the *medium_user* and the *light_user* types. The PTA models for each user type can be completely different or be similar only varying in the distribution between actions. In our experiments we had the latter option.

The load generation process proceeds as follows: the master node takes as input the performance models, the test duration, the ramp function, the number of concurrent users and the target KPIs. The master node initiates load generation on the slaves in an incremental order. Each slave node monitors its local resource utilization and the KPIs of the SUT during the load generation. If the threshold for the local resource utilization on the slave node is exceeded, the slave node notifies the master node that it can not anymore generate new virtual users. The master then initiates load generation on another slave node. If the threshold of the measured KPI has been exceeded (in case a target KPI has been specified) or the test duration has ended the slave nodes notifies the master node and the load generation is stopped.

During the load generation on the slaves, the slave nodes execute the PTA models describing the user behavior as specified in Section IV-A in parallel processes. For each user the slave node starts a new process. The slave node then selects a user type if several are specified. The user type is selected based on probabilistic choice, see Figure 7. The virtual user then executes the PTA that belong to the selected user type inside its own process. Consider location *1* of the PTA in Figure 6. A possible execution of the PTA would be as follows: A virtual user waits until the clock variable X reaches *3* and then fires the transition. Upon firing the transition the action *login()* is sent to the adapter of the slave. The adapter creates a HTTP message, gets the user credential from the *User DB*, and sends the action to the SUT. In the adapter a timer is started to measure the response time. When the response is received it is checked in the adapter for the status code and the response time is stored. After that the clock variable *X* is reset to zero and

Table I
RESPONSE TIME MEASUREMENTS FOR USER ACTIONS WHEN RUNNING WITH 1000 CONCURRENT USERS.

	Light Users		Medium Users		Heavy Users	
Action	Average (sec)	Max (sec)	Average (sec)	Max (sec)	Average (sec)	Max (sec)
upload_file(video/mpeg)	82.3	133.0	81.5	133.5	85.1	133.3
upload_file(audio/mpeg)	158.3	217.4	143.7	214.3	126.2	210.5
upload_file(image/jpeg)	56.9	134.1	54.7	126.2	47.2	119.1
download_file(video/mpeg)	0.16	2.8	0.16	2.8	0.12	3.6
download_file(audio/mpeg)	0.15	3.0	0.18	3.2	0.18	3.0
download_file(image/jpeg)	0.13	3.1	0.15	3.7	0.16	1.4

the PTA moves from location *1* to location 2. In location *2* the transition to fire is based on the probabilistic values. For example, location 4 is reached with a probability of 0.25. In location *4* the transition is fired when the clock variable X reaches 3. The *random_dir()* action is sent through the adapter to the SUT. In this case, the adapter uses the *User-Resource Data Base* to select a folder for the user. Upon receiving the response the clock is reset and the PTA moves back to location 2. The process is repeated until the *exit()* action is fired and the end state is reached. The slave then chooses a new user type and the PTA corresponding to that user type is executed in a similar way. In a nutshell, every user runs independently of each other and decides for itself which actions to execute.

We have run two experiments with our tool, based on its two usage modes described in Section V-B

Experiment 1. In the first experiment, we wanted to answer the following question: *What are the mean and max response times of all actions when system is under the load of 1000 concurrent users?* We ran the test for 1 hour.

In this experiment we found out that the SUT had a bottleneck, namely the hard disk. Table I shows the average and max response times values for the actions and user types.

From the table one can see that the response times for the three upload actions are considerably higher that the ones for download. This is because a lot of data had to be written to the hard disk on the SUT, while the slave nodes simply discarded the data that the virtual users downloaded. Figure 8 shows the average response time plotted over time for the three upload actions for the *heavy_user type*.

Experiment 2. In the second experiment, we wanted to know *how many concurrent users of given types the system supports before the response time degrades beyond a given threshold?* The target threshold limits from the actions can be seen in Table II.

To figure this out, we had the tool to ramp up the number of user from 0 to 150 following the user type distribution in Figure 7 and the tool reported back when the measured response times exceeded any of the threshold values set for user actions specified in Table II. Figure 9 shows the average response times for the three upload actions plotted over time for the *medium_user* type when ramping up from 0 to 150 users. Similar graphs were created by the tool for the *light* and *heavy* user types. The test report also includes two tables for this experiment (see Table II and III).

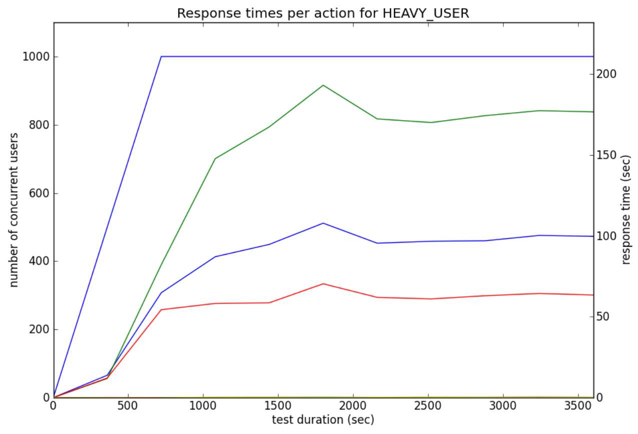

Figure 8. Average response time for uploading picture (bottom), video (middle), and music (top) when running with 1000 concurrent users.

Table II shows the time and number of users at which the threshold value for individual actions was exceeded. Table III shows the average and max response times for individual action over the entire test durations. The tool reported that the average and max response times were exceeded for all of the three upload actions. However, the response time for *upload_file(audio/mpeg)* for the *medium_user* type went over the set threshold of 3.5 seconds at 8 minutes and 44 seconds (524 seconds) into the test run. The tool was then testing with 74 concurrent users. The distribution between user types was the following: 50% *light_users*, 28% *medium_users*, and 22% *heavy_users*.

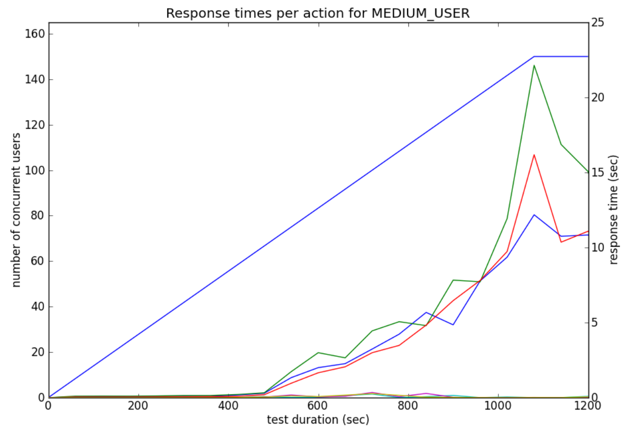

Figure 9. Average response times for uploading video (bottom), picture (middle), and music (top) when ramping up from 0 to 150 concurrent users.

VII. CONCLUSION AND FUTURE WORK

In this paper, we have presented a model-based performance testing tool that uses probabilistic models to generate

Table II
TIME AND NUMBER OF USERS AT WHICH THE THRESHOLD VALUE WAS EXCEEDED WHEN RAMPING UP FORM 0 TO 150 USERS

Action	Target Response Time		Light Users (50 %)		Medium Users (28 %)		Heavy Users (22%)		Verdict
	Average (sec)	Max (sec)	Average (users)	Max (users)	Average (users)	Max (users)	Average (users)	Max (users)	Pass/Fail
upload_file(video/mpeg)	4.5	12	78 (555.0 sec)	94 (669.0 sec)	86 (613.0 sec)	94 (670.0 sec)	78 (558.0 sec)	112 (801.0 sec)	Failed
upload_file(audio/mpeg)	3.5	10	76 (543.0 sec)	94 (670.0 sec)	74 (524.0 sec)	94 (670.0 sec)	74 (528.0 sec)	94 (671.0 sec)	Failed
upload_file(image/jpeg)	2.5	8	77 (545.0 sec)	80 (572.0 sec)	74 (524.0 sec)	80 (572.0 sec)	78 (555.0 sec)	101 (719.0 sec)	Failed
download_file(video/mpeg)	4.5	12	Passed	Passed	Passed	Passed	Passed	Passed	Passed
download_file(audio/mpeg)	3.5	10	Passed	Passed	Passed	Passed	Passed	Passed	Passed
download_file(image/jpeg)	2.5	8	Passed	Passed	Passed	Passed	Passed	Passed	Passed

Table III
RESPONSE TIMES WHEN RAMPING UP USERS FOR 0 TO 150 USERS

	Light Users		Medium Users		Heavy Users	
Action	Average (sec)	Max (sec)	Average (sec)	Max (sec)	Average (sec)	Max (sec)
upload_file(video/mpeg)	4.78	79.17	4.08	40.79	4.72	64.09
upload_file(audio/mpeg)	5.34	92.79	5.57	90.20	6.60	92.79
upload_file(image/jpeg)	4.22	93.44	4.25	93.04	5.04	87.84
download_file(video/mpeg)	0.05	1.98	0.05	1.57	0.04	1.57
download_file(audio/mpeg)	0.04	1.44	0.07	2.11	0.04	1.33
download_file(image/jpeg)	0.05	2.04	0.05	2.10	0.06	1.99

synthetic workload which is applied to the system in real-time. The models are based on the Probabilistic Timed Automata, and include statistical information that describes the distribution between different actions and think time. The tool has a scalable distributed architecture with a master node that controls several slave nodes. The slave nodes monitor the target KPIs and the local resource utilization, and after the test duration has ended the monitored values are sent to the master node which produces a test report.

We have described how load is generated from the PTA models and we have also discussed the most important features of the tool. We demonstrated the utility of the tool on a WebDav case study. We use our tool to answer the two questions about the system under test: What are the values of different KPIs when the system is under a particular load and how many users of given types does the system support before its KPIs degrade beyond a given threshold?

In the future, we will focus on the creation of the models and try to optimize the algorithm for load generation even further. We will strive to have a more formal approach on how we go from requirements to model. Also we will look further into load generation, for instance, develop methods to specify a minimum number of user action that has to be fulfilled (trace lengths) before the user can exit the system.

We are currently performing larger scale experiments to evaluate the capabilities of the tool against existing tools like, *JMeter* [13] and *httperf* [7]. Further, we plan to add target KPI values, for instance response time and performance requirements, in the models. By doing that, we can have performance requirements and address target response time values for individual actions.

REFERENCES

[1] D. Ferrari, "On the foundations of artificial workload design," in *Proceedings of the 1984 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, ser. SIGMETRICS '84. New York, NY, USA: ACM, 1984, pp. 8–14.

[2] J. Shaw, "Web Application Performance Testing – a Case Study of an On-line Learning Application," *BT Technology Journal*, vol. 18, no. 2, pp. 79–86, Apr. 2000.

[3] R. Jain, "The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation and Modeling (Book Review)," *SIGMETRICS Performance Evaluation Review*, vol. 19, no. 2, pp. 5–11, 1991.

[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*, ser. WOSP '04. New York, NY, USA: ACM, 2004, pp. 94–103.

[5] C. Barna, M. Litoiu, and H. Ghanbari, "Model-based performance testing (NIER track)," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 872–875.

[6] 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*. New York, NY, USA: ACM, 2006, pp. 54–61.

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

[8] G. Ruffo, R. Schifanella, M. Sereno, and R. Politi, "WALTy: A User Behavior Tailored Tool for Evaluating Web Application Performance," *Network Computing and Applications, IEEE International Symposium on*, vol. 0, pp. 77–86, 2004.

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

[10] M. Jurdziński, M. Kwiatkowska, G. Norman, and A. Trivedi, "Concavely-Priced Probabilistic Timed Automata," in *Proc. 20th International Conference on Concurrency Theory (CONCUR'09)*, ser. LNCS, M. Bravetti and G. Zavattaro, Eds., vol. 5710. Springer, 2009, pp. 415–430.

[11] *HTTP Extensions for Web Distributed Authoring and Versioning (WebDAV)*, http://www.webdav.org/specs/rfc4918.pdf, Network Working Group pdf, retrieved: October, 2012.

[12] AWStats, http://awstats.sourceforge.net/, retrieved: October, 2012.

[13] The Apache Software Foundation, "Apache JMeter," http://jmeter.apache.org/, retrieved: October, 2012.