Towards Automatic Performance and Scalability Testing of Rich Internet Applications in the Cloud

Niclas Snellman*, Adnan Ashraf*†‡, Ivan Porres*
*Department of Information Technologies, Åbo Akademi University, Turku, Finland.
Email: {niclas.snellman, adnan.ashraf, ivan.porres}@abo.fi
†Turku Centre for Computer Science (TUCS), Turku, Finland.
‡Department of Software Engineering, International Islamic University, Islamabad, Pakistan.
Email: adnan.ashraf@iiu.edu.pk

***Abstract*—The emergence of asynchronous techniques for building interactive web applications has led to the development of Rich Internet Applications (RIAs). RIAs offer greatly enhanced usability and the ability to deliver rich dynamic content. However, due to the widespread use of RIAs, there is a need to develop and test highly scalable RIAs. Furthermore, cloud computing introduces new opportunities for ensuring and extending performance and scalability of RIAs. This has necessitated the need to devise effective ways for doing automatic performance and scalability testing of RIAs. In this paper, we describe different problems and challenges in automatic performance and scalability testing of RIAs. We then propose the ASTORIA framework as a novel solution to the identified problems and challenges. The effectiveness of our proposed approach is demonstrated by building a working prototype for ASTORIA and by using it for conducting experiments.**

***Keywords*-Performance testing; scalability testing; rich internet applications; cloud computing**

I. Introduction

Rich Internet Applications (RIAs) [1] are web applications that provide a desktop-like application user experience by using asynchronous techniques for communication and dynamic content rendering such as Ajax (Asynchronous JavaScript and XML) or other advanced web technologies such as Adobe Flash and Microsoft Silverlight. The widespread use of RIAs has resulted in higher expectation levels concerning web application usability and performance. However, the technologies used in RIAs have also brought new challenges to performance and scalability testing of web applications.

Liu [2] defines performance as a measure of how fast an application can perform certain tasks, whereas scalability measures performance trends over a period of time with increasing amounts of load. If an application has good performance at a nominal load, but fails to maintain its performance before reaching its intended maximum load level, then the application is not considered scalable. Good performance at nominal load does not guarantee application scalability. On the other hand, an application that can not perform well at nominal load, also lacks scalability. An application should ideally maintain a flat performance curve until it reaches its intended maximum load level.

Cloud computing introduces new opportunities and dimensions for ensuring and extending the performance and scalability of RIAs. In a compute cloud, a RIA is hosted on one or more application server instances that run inside dynamically provisioned Virtual Machines (VMs). The growing size and complexity of RIAs and of the underlying infrastructure used for hosting these applications has necessitated the need to devise effective ways for doing automatic performance and scalability testing.

Conventional functional and non-functional testing of web applications generally involve generating synthetic HTTP traffic to the application under test. A commonly used approach [3]–[5] is to capture HTTP requests and then replay them with the help of automated testing tools. Fig. 1 presents an abstract view of how this works. However due to the complexity introduced by asynchronous technologies used for building RIAs, conventional load generation methods do not work. Instead we need to consider capturing and replaying of user actions on a higher level, that is user interactions with a web browser, as shown in Fig. 2. Subsequently, we are forced to generate load by automating ordinary web browsers. While web browser automation in itself is not particularly difficult, the task of automating large numbers of web browsers simultaneously in an effective manner is a non-trivial task.

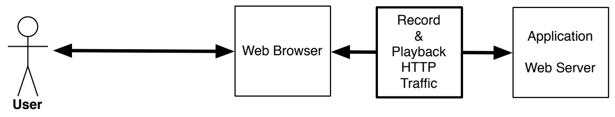

Figure 1. Record and playback HTTP requests

Cloud computing provides theoretically unlimited computing resources which we can use for generating test load and for simulating large quantities of web application users. However, we still need to use the cloud resources efficiently in order to minimize the costs of performing a test.

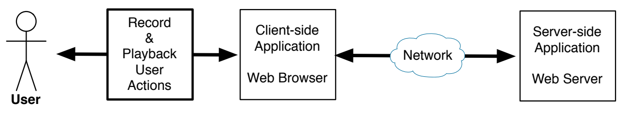

Figure 2. Record and playback user actions

In this paper, we describe different problems and challenges in cost-effective automatic performance and scalability testing of RIAs. We then propose the ASTORIA framework as a solution to the identified problems and challenges. One of the distinguishing characteristics of our proposed approach is that it does not require server-side monitoring of performance metrics or modification of any server configurations. The effectiveness of our proposed approach is demonstrated by building a working prototype for ASTORIA and then by using it for conducting experiments involving performance and scalability testing of RIAs.

The rest of the paper is organized as follows. Section II describes more precisely the problem that we try to tackle while Section III discusses the main challenges in automatic performance and scalability testing of RIAs. Section IV discusses load generation methods for RIAs while Section V presents the ASTORIA framework. Section VI describes data aggregation in the ASTORIA framework. An early version of our working prototype for the ASTORIA framework, along with an experiment and its results, is presented in Section VII. In Section VIII, we discuss related work and Section IX presents our conclusions.

II. Problem Statement

The most commonly used metrics for measuring performance and scalability of RIAs are response time and throughput. Response times measures the time required to complete one single action while throughput measures how many user actions are completed over a unit of time. By measuring and recording response time and throughput, we can answer following questions concerning performance and scalability of RIAs:

- What is the average response time *R* at a certain load *L* (number of simultaneous users)?
- What is the average throughput *T* (actions per second) at a certain load *L*?

The objective of this paper is to discuss how we can answer these questions empirically by carrying out a performance test using cloud computing resources. The two main concerns of our approach are how to automate and simplify from a developer's point of view the performance test process and how to reduce the costs of a performance test.

In order to automate and simplify the performance test we propose a testing approach that does not require the installation of any additional software in the application server(s). We also require that the provisioning of the testing hardware is handled automatically. Furthermore, we propose a method for automatically deciding when a performance test can be concluded. What this means in practice is that no user interaction is required after a test has been started.

Cloud providers usually have usage-based charging schemes, with different tariffs for different services. Except for the most obvious fee, i.e. VM provisioning fees, it is not uncommon for providers to charge for storage services (not VM local storage), bandwidth usage, load balancing services, etc. For cloud based load generation, the most important costs to consider are VM provisioning fees. Slightly simplified, the total cost for conducting a performance test using load generated in the cloud can be calculated using the formula:

$$cost = max(1, \lceil \frac{n}{n_{vm}} \rceil) \cdot c_{vm} \cdot \lceil t \rceil, \qquad (1)$$

where n is the desired number of simulated users and n_{vm} is the number of users that can be simulated on each VM. The test time is t and c_{vm} denotes the cost for renting on VM per time unit.

In order to minimize the cost, we should clearly try to minimize the cost per VM c_{vm} and the length of the test t and maximize the number of simulated users per VM n_{vm}. Minimizing the cost per VM can be achieved by selecting the cheaper cloud provider for the required level of service. This task is out of the scope of the article. In order to minimize the length of the test, we discuss in Section V-A2 how to stop the test once the test results are statistically significant. In order to maximize the number of simulated users per VM we use various techniques such as using headless browsers and introducing a resource control loop in each VM as described in Section V-C. Finally, in order to reduce bandwidth costs, we aggregate the test results in the load generation servers as described in Section VI.

III. Challenges

We now describe some specific problems and challenges in performance and scalability testing of RIAs.

A. What do we measure?

Response times can be measured at different levels as shown in Fig. 3. Server side monitoring methods which only record the time taken to process requests in the server are shown as level 3. Traditional non-functional testing methods usually record response time as the time it takes for the system under test to respond to the HTTP requests that are sent by the testing tool. This is shown as level 2 in Fig. 3. When testing RIAs, however, response time measures how fast actions are executed by the RIA in the web browser (level 1). Thus, response time includes the time it takes for the client to execute any JavaScript that is triggered by the

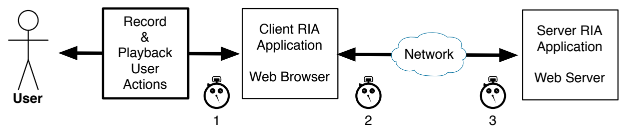

Figure 3. Different levels for measuring response time

action and the time it takes for the server to respond to any HTTP requests sent by the JavaScript.

Ideally we would like to exclude the time that is spent in the network as well as JavaScript execution time and any overhead generated by the web browser, since that would give us the most precise results. To properly exclude network time, we would have to employ server-side monitoring of requests. This approach would require a certain amount of tampering with the servers, which is something we would like to avoid altogether. The network overhead can, however, probably be ignored if the tests are performed in the same cloud as the RIA servers. Eliminating the overhead generated by the browser might be achieved by adding a proxy that forwards requests from the browser to the server and records the time for each request. Such methods are, however, outside the scope of this paper.

B. Uniqueness of HTTP Requests

Conventional methods for generating load on a web site involve sending large number of beforehand captured HTTP requests to a web site and measuring the response times. This is possible since simple web sites often do not keep track of user sessions. This is not the case for RIAs. For RIAs to work properly, the server has to keep track of the state of every individual user's session. This can be accomplished by, for instance, storing a token which identifies the session on the server and in a cookie on the client's computer. Thus, by passing the session token as a parameter with each request, a client can identify itself. A consequence of this is a security issue commonly known as Cross-Site Request Forgery (CSRF), which is a type of exploit where an unauthorized user of a web application identifies itself by using the session token of a user that the web application trusts. A simple capture-replay approach will therefore not work, since the recorded requests contain the token of the session that was active at the time when the requests were recorded.

C. Load Characterization

Since user actions are more complex than individual HTTP requests, we need to consider that they have different classes. The performance requirements for a user action to request information about a product may be different from the performance requirements of a user action that actually buys the product and processes the payment. Thus, each class of user actions can have a different average response time and then the values of response times would be distributed around its average. For example, if the action class A has an average response time of 0.3 seconds and the rest of the response times are distributed around 0.3 seconds, action class B might have an average response time of 1 second and initialization actions may have an average response time of 5 seconds. Thus, all classes of user actions should be identified and defined beforehand and the data aggregation and calculations need to be done separately for each action class.

Response time is a random variable and there need to be as many random variables as action classes. The samples of each action class should be distributed closely around the average response time in that class. If the action classes are poorly defined, however, the samples will be more dispersed. Thus, there need to be as many random variables as action classes. Throughput should also be considered in a similar manner. For each action class, there need to be a random variable for throughput. Thus, there should be twice as many random variables as classes of user actions.

D. How Many VMs and How Many Virtual Users on Each VM?

When utilizing VMs in a compute cloud for the purpose of generating load we need to know how many VMs we need to rent. Determining the number of required VMs is entirely dependent on the amount of load that we want to generate and how much load each VM is capable of generating. If we know that each VM is capable of simulating k virtual users and we need to simulate a total of N virtual users, then we simply divide N by k and get the required number of VMs. However, this is difficult to predict. Therefore, we need to have a control loop which has a goal to create N virtual users. Furthermore, the control loop should create VMs in such a way that the target number of virtual users can be reached as fast as possible. Then for each VM, the number of simultaneous virtual users to run needs to be decided. While doing this, we need to consider the resource constraints of the VMs, such as CPU capacity and network bandwidth. Hence, we need to have two control loops in total; one for deciding how many VMs are needed to reach a target of N users and a second loop for deciding how many maximum simultaneous virtual users can be run on each VM. The challenge is not only to do it, but to do it as automatic and as efficiently as possible.

E. When to Stop Testing?

In order to achieve automatic testing of RIAs, there need to be an automatic way of deciding when the results are good enough to stop the test. Additionally, we also need to evaluate when it is most appropriate to stop testing. For example, in the Amazon Elastic Compute Cloud (EC2) [6], the renting interval is one hour. If the testing results indicate that we can stop testing, then we need to decide if it is better to stop right now and get the results early or keep on running the tests until we reach to the end of the renting hour. When stopping near the end of the renting hour, we also need to consider the VM stopping time. VMs normally need a couple of minutes to shut down and therefore, to avoid paying for the next renting hour, they must be stopped a couple of minutes before the end of the renting hour.

The approach we suggest for deciding when a performance test has run for a sufficiently long time interval is to monitor the standard deviation of the response times. We calculate the standard deviation from the response times for each specific action or action class. Subsequently, we abort the test once the standard deviations reach a certain predetermined lower limit. For a scalable RIA, low standard deviations indicate that the load balancer has been able to successfully scale the server(s) to a level where the load can be handled comfortably and the test can be stopped.

IV. Load Generation

A. *Conventional load generation methods*

Conventional methods [3]–[5] for generating traffic for non-functional testing of web sites generally involve capturing and replaying HTTP requests. Requests are usually recorded while a typical usage scenario is conducted in a web browser. Sufficient load can consequently be generated by replaying the recorded requests many times simultaneously, thus simulating many concurrent users. Using this approach, it is possible to generate traffic equal to that of hundreds of users from each machine that is reserved for testing. In order to enable simulation of a more diversified user pool, it is sometimes possible to modify some requests (or parts of requests) to simulate varying usage scenarios.

The conventional load generation methods usually log various parameters describing the servers response to each request. Typical metrics are for example server response time, throughput, and latency. Such metrics can later be used to evaluate the system under test. Depending on the type of test, the data produced in the load generation machines may be sufficient, thus eliminating the need for server side data aggregation.

B. *Load generation for RIAs*

Unfortunately, the simple capture-replay approach is not applicable to RIAs due to different reasons. For instance, generating load representing realistic user traffic for web applications that relies on Ajax for providing an interactive UI will be challenging due to the randomness of the requests sent by Ajax. Another obstacle is due to the various methods that have been developed for preventing CSRF attacks. Recorded requests containing identification tokens from earlier sessions will inevitably be blocked. Thus, in order to use conventional methods for generating traffic for RIAs we would be required to disable CSRF prevention methods on the server. Ideally we would like to avoid having to do any modifications to the system under test.

A solution to these problems would be to rely on ordinary web browsers for load generation. In order to do so it is necessary to, instead of recording low level HTTP requests, record high level user interactions with the web browser. Such recordings (*test scripts*) can later be used to automate the behavior of the web browser in order to simulate a real user. However, using regular web browsers for generating traffic introduces several technical challenges for the test automation and orchestration.

While web browser automation in itself is not a major problem (there are, in fact, many different tools for automating various web browsers already in existence [7] [8] [9]), using ordinary web browsers for generating traffic is not an ideal option. Due to their heavy resource dependencies, regular web browsers are not suitable for effective load generation. For example, running one instance of Mozilla Firefox may consume up to (and possibly even more) a hundred megabytes of memory. This leaves us with a comparatively low number of web browser instances that we are able to run simultaneously per machine. Furthermore, we are limited to simulating only one user per web browser instance. Consequently, generating traffic with web browsers becomes a costly affair when the number of machines needed to generate a sufficient load increases.

C. *Headless browser*

Another approach is to use a headless web browser, which is a web browser without a graphical user interface (GUI). The lack of a GUI in a headless web browser enables web browsing with a smaller memory footprint. This would enable us to simulate a higher number of users per machine, thus reducing costs.

Headless web browsers are usually controlled programmatically by using an application programming interface (API) provided by the developers [10] [11]. Consequently, for us to be able to automate a headless web browser, we are forced to develop a custom driver application. While this may be more complicated than automating an ordinary web browser using tools already available, it offers us a wider range of options. Output data, for instance, can be customized to suit the needs of the types of tests we are conducting.

We also need to be able to control the headless web browsers in such a way as to simulate typical user behaviour. One approach for this is to make it take some form of test scripts describing usage scenarios as input. Ideally, the driver would also allow variables in the scripts, which would enable us to easily generate traffic with the characteristics of a diverse user pool. This would mitigate the need for creating multiple test scripts in order to simulate varying user scenarios.

D. Headless X Window System

A similar approach may be accomplished by exploiting the possibility of running the X Window System for Linux in a headless fashion [12]. This would allow us to use ordinary web browsers without being forced to use a GUI. The apparent advantages of this approach are ease of automation and the reliability and robustness of standard web browsers. There are, however, disadvantages. Those include, for example, the overhead that is generated by the X Window System, in spite of the lack of a GUI, and more difficult setup of VMs for testing.

V. ASTORIA Framework

ASTORIA (Automatic Scalability Testing of RIAs) is a framework for automatic performance and scalability testing of RIAs. It consists of the following main components: ASTORIA Master, ASTORIA Load Generator (ALG), ASTORIA Local Controller (ALC), and Virtual User Simulator (VUS). ASTORIA Master launches, manages and terminates VMs in a compute cloud, each acting as an ALG. In each ALG, there is an ALC that controls a number of VUSs. An overview of the ASTORIA framework is shown in Fig. 4.

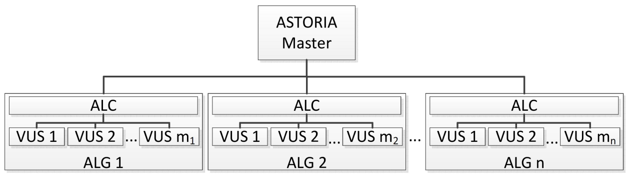

Figure 4. ASTORIA framework

A. ASTORIA Master

ASTORIA Master controls n VMs in an Infrastructure as a Service (IaaS) cloud such as Amazon EC2. There are two main challenges for ASTORIA Master:

- How to automatically decide an optimal number of ALGs based on the required number of virtual users to simulate?
- How to automatically decide when to stop testing?

1) Deciding the Optimal Number of ALGs: Unfortunately, we do not know beforehand how many users we will be able to simulate per VM. The reason for this is the greatly varying system resource requirements of the VUS. This is analogous to the varying system resource utilization required by ordinary web browsers when using different web applications. Depending on which IaaS we decide to use, there may also be different types of VMs available with different amounts of memory and CPU cores, which will have a significant influence on the number of virtual users per VM.

In order to deal with the first challenge, ASTORIA Master runs a control loop. The loop starts by instantiating a default number of ALGs and then it keeps on iterating until the target number of users N is reached. In each iteration, ASTORIA Master decides how many more ALGs need to be instantiated in proportion to the existing and required number of virtual users. Thus the ASTORIA Master acts as a proportional controller [13] that tries to reach the target number of virtual users as fast as possible. It uses the following formula for calculating the proportional number of ALGs:

$$N_{ALG} = \left\lceil \frac{N - \sum_{i=1}^{n} e_i}{\frac{\sum_{i=1}^{n} e_i}{n}} \right\rceil, \quad (2)$$

where N_{ALG} is the optimal proportional number of ALGs to instantiate in an iteration, N is the target number of virtual users, n is number of existing ALGs, and e_i denotes the number of existing virtual users on the i-th ALG. For example, if N is 1000, and after instantiating two ALGs in the first iteration of the control loop, e_1 is 101 and e_2 is 99, then in the second iteration, ASTORIA Master launches 8 ALGs in order to reach the target load as fast as possible. For simplicity, measurement delay and ALG instantiation time are not considered.

2) When to Stop Testing?: ASTORIA Master uses an approach called sequential sampling [14] to tackle the second challenge. The idea of sequential sampling is to keep on taking samples until a desired condition is met. For performance and scalability testing, the desired condition is specified in terms of performance (response time and throughput) requirements for an action class and the intended load level. For example, if for a certain action class, the intended load level is 1000 virtual users with an average response time of 500 milliseconds and an average throughput of 2000 requests per second, then this can be specified as the desired condition for sequential sampling. ASTORIA Master collects samples of performance data from all ALGs and keeps on aggregating it in order to calculate the average and standard deviation response time and throughput and as soon as the desired condition is met, the results are considered to be good enough to stop the test and the VMs are terminated.

B. ASTORIA Load Generator (ALG)

Each VM acts as an ALG, whose purpose is to generate load on the RIA under test. This is done by starting an ALC on the ALG which in turn controls a number of VUSs.

C. ASTORIA Local Controller (ALC)

On each ALG, an ALC controls m_i VUSs. The main responsibility of ALC is to determine m_i and then to ensure that each VUS creates as much load as possible, but without breaking the performance constraints of the VM. Similarly to ASTORIA Master, ALC works as a proportional controller that, in each iteration, calculates how many virtual users should be created or removed in proportion to the current and intended level of CPU and network bandwidth consumption. For this, ALC has two regulators; one for CPU consumption and one for network bandwidth. The proportional number of virtual users N_{user} in each iteration depends on these regulators, which use the following formulas:

$$N_C = \left\lceil \frac{C - \sum_{i=1}^{n} c_i}{\frac{\sum_{i=1}^{n} c_i}{n}} \right\rceil \; and \tag{3}$$

$$N_B = \left\lceil \frac{B - \sum_{i=1}^{n} b_i}{\frac{\sum_{i=1}^{n} b_i}{n}} \right\rceil , \tag{4}$$

where N_C is the optimal proportional number of users to instantiate in the next iteration based on the CPU consumption. C is the intended CPU consumption, n is the number of existing users and c_i denotes CPU consumption of the i-th user. Similarly, N_B is the optimal proportional number of users to instantiate in a next iteration based on the bandwidth consumption, B is the intended network bandwidth consumption, and b_i denotes network bandwidth consumption of the i-th user. N_{user} is then simply the minimum of N_C and N_B.

For example, if n is 2, C is 90%, c_1 is 6%, and c_2 is 4%, then the CPU consumption error is 80% and thus N_C is 16. Similarly, if B is 80%, b_1 is 5%, and b_2 is 5%, then the network bandwidth consumption error is 70% and thus N_B is 14. In this case, N_{user} would therefore be 14.

D. Virtual User Simulator (VUS)

Virtual users are simulated by the VUSs. The exact number of virtual users to simulate is controlled by the ALC.

VI. Data Aggregation

Due to bandwidth limitations and the cost of bandwidth usage, performance data (response time and throughput) for each action class is compiled into a smaller set of data locally on each ALG before being sent to the ASTORIA Master. While this may not be necessary for smaller tests, it may be neccessary for tests with a high number of simulated users in order to keep ASTORIA Master from being overloaded.

A. Throughput and Response Time Aggregation

Each instance of VUS logs the response time of each performed action locally on the ALG it is running on. At certain time intervals, each ALC aggregates the response time data from all VUSs on its ALG and calculates the average and standard deviation for the response times separately for each action class. Additionally, throughput is calculated separately for each action class. Thus, three values are associated with each action class: *average response time*, *response time standard deviation* and *throughput*. A list containing one set of these values for each action class is then sent to the ASTORIA Master. Once a list from each ALG has been received, ASTORIA Master calculates overall average and standard deviation for response times and throughput for each action class. This final set of data can then be used to determine the overall performance and scalability of the RIA under test.

B. Aggregating Averages and Standard Deviations

The following statistical formulas are used for combining the averages and standard deviations of several groups of samples (response times):

- For aggregating averages, we simply calculate the weighted average of averages.
- For aggregating standard deviations, we calculate combined standard deviation by using the method described by Langley in [15]. It uses the following formula:

$$S = \sqrt{\frac{B - \frac{A^2}{N}}{N - 1}}. \tag{5}$$

In (5), N is the size of combined samples from all groups, that is $N = n_1 + n_2 + n_3$, etc. A represents the total sum of observed measurements, that is, $A = A_1 + A_2 + A_3$, etc., where A_i is calculated as $m_i \cdot n_i$, where m_i is the average of the samples in the i-th class and n_i is the sample size. B denotes the sum of B_1, B_2, B_3, etc., where B_i is calculated as:

$$B_i = s_i^2 \cdot (n_i - 1) + \frac{A_i^2}{n_i}, \tag{6}$$

where s_i is the standard deviation of the ith class.

VII. Performance testing with the ASTORIA Prototype

In this section, we describe a prototype implementation of the ASTORIA framework. Additionally, the results of a test that was carried out using the implementation is shown.

A. Prototype

Our prototype consists of the main components described in Section V. ASTORIA Master instantiates, manages and terminates ALGs. ALGs are in practice VMs in the Amazon EC2 cloud. ASTORIA Master runs on a local machine or on a VM in the cloud.

ASTORIA Master takes *test scripts* as input along with a few parameters describing the test; such as number of users to simulate, ramp up time and sleep time between actions. When starting a test, ASTORIA Master launches the default number of ALGs. When the ALGs have been launched, the ALC on each ALG creates a number of VUS instances.

Each instance of VUS simulates a number of users by running multiple instances of a headless web browser simultaneously in parallel threads. The headless web browser used in our prototype is HTMLUnit [10]. HTMLUnit simulates user interactions by loading a web application and performing actions specified in a test script.

Test scripts are produced by Selenium IDE [7]. Selenium IDE is an extension for Mozilla Firefox which enables the recording of high level user interactions with the browser. Test scripts produced by Selenium IDE are stored as text formatted in HTML containing the actions performed and some additional information about the test. Actions are described using three variables; *action type*, *target* and *value*. Action type and target are self-explanatory. The value variable is needed for actions that require extra information. For instance, the coordinates of the mouse pointer when a click-action was performed or the string that was typed in a text insertion action.

Each VUS logs the response time of each performed action in a local file. At certain intervals, ASTORIA Master executes a script in the ALGs that aggregates the log files created by the instances of VUS that are running on the ALG. The script compiles the log files into a smaller set of data using the methods described in Section VI. This new set of data is saved in the form of a JavaScript Object Notation (JSON) formatted text file on the ALG. This file, which we call a *report*, is subsequently fetched by ASTORIA Master. After a report from every ALG has been collected, the reports are aggregated into one combined set of data using the same method that was used to aggregate the data locally in the ALGs.

Action classes are defined by specifying regular expressions that separate the actions based on their *action type* and/or their *target*. Before compiling the log files on the individual ALGs, the actions are sorted into action classes based on the regular expressions. Subsequently, the data in each action class is aggregated separately before being sent to the ASTORIA Master.

B. Experiment

The application under test in our experiment is an experimental RIA developed by Vaadin Ltd. The RIA is a mock-up of an on-line vendor for movie theater tickets called QuickTickets, implemented using the Vaadin web development framework [16]. QuickTickets was set-up by the Vaadin staff on two VMs of unknown size in the Amazon EC2 cloud. No CSRF prevention methods were disabled before conducting the tests.

One usage scenario was used to generate load. The test script describing the scenario was recorded when a user used QuickTickets to search for tickets, select tickets, and make an order. This resulted in a test script with more than 100 actions. A feature that allowed us to insert random integer variables in the script at runtime was implemented. This allowed us to use the same test script for simulating users buying tickets for different movie theaters and different shows.

The test was carried out with an early version of our prototype implementation in which the number of ALGs and the number of users that were simulated per ALG were static. The RIA was tested with 1000 concurrent users. In order to keep the number of VUS low on each ALG, we decided to utilize the maximum number of VMs that were available with a basic Amazon AWS account, which is twenty (20). A ramp-up time was used in order to slowly introduce the RIA to the generated load. The reason for keeping the number of VUS low was previously experienced memory leaks in HTMLUnit, which we feared could impede the VUS abilities to properly simulate users.

The testing was performed during the course of a few hours and reports were fetched from the ALGs with an interval of five (5) minutes. Throughput was not used as a metric for this test and therefore no such data was collected.

C. Results

The prototype showed promising results for the adequacy of the ASTORIA framework for generating load for non-functional testing. ALGs were launched and controlled by ASTORIA Master automatically without any user interaction after the test was started. The output data from the VUSs were properly aggregated first separately on the each ALG and then collectively on ASTORIA Master. Results from the test can be seen in Fig. 5. Since the system under test was able to comfortably handle the load generated by the ALGs, the graph only shows the first few steps. No surprises were found in the later stages of the test.

D. Interpretation and analysis

Generally, the response times in the performed test were slightly higher than expected. Also, some extremely high samples could be seen. We suspected that these were a result of multiple instances of HTMLUnit running simultaneously on the same VM. This was later confirmed when we compared the average response time for an action performed by an increasing number of concurrent HTMLUnit instances. A graph showing the results of this test can be seen in Fig. 6. There is a clear increase in response time when multiple instances of HTMLUnit are running at the same time. This motivates further research on VUS implementation.

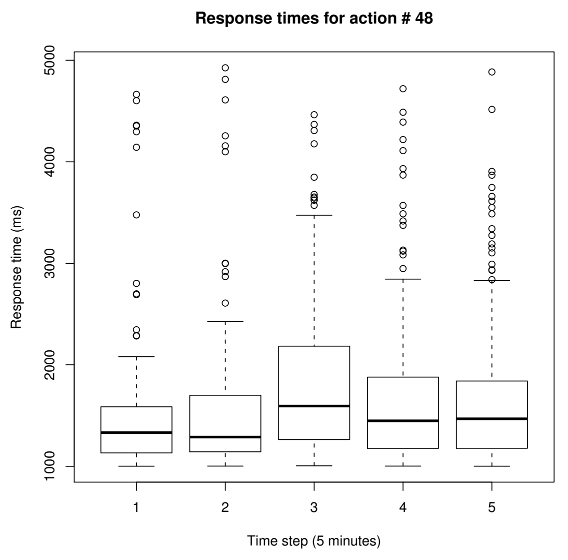

Figure 5. Box plot showing response times for time steps 1-5 with outliers removed

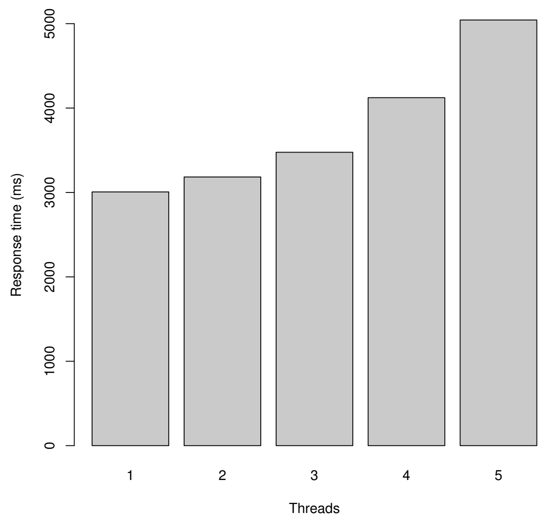

Figure 6. Results from HTMLUnit profiling

VIII. Related Work

Non-functional testing of web applications is a vastly explored area. Some of the earlier script-based approaches, such as [3], used a load generator with active monitoring of performance metrics. Some commercial load testing tools, such as HP LoadRunner [17], also use a script-based approach. Saddik [18] proposed a method for performance and scalability testing of web applications which are based on web services. Gao et al. [19] proposed a reactivity-based framework for automated performance testing of web applications. In addition to script-based approaches, some other approaches, such as [20], used stochastic form-oriented analysis models for simulating realistic user behavior.

There are also a number of open-source tools, such as Apache JMeter [21], WebLoad [22], and PushToTest [23]. Furthermore, there is a tremendously growing interest in exploiting cloud resources for doing performance testing of RIAs. Load testing services such as Load Storm [24], Load Impact [25], SOASTA [26], and BrowserMob [27] use VMs (from a compute cloud) for generating load on the application under test. However, most of the existing works and tools are applicable only for load testing. There have been relatively few works on other types of performance testing and on automatic scalability testing of RIAs.

When comparing to existing approaches, the ASTORIA framework is able to offer relatively cheap performance and scalability testing of RIAs by effectively automating headless web browsers in the cloud. By calculating an optimal proportional number of VMs required to create an intended number of virtual users for generating load on the RIA under test, a minimal number of VMs are used. The framework also supports automatic stopping of tests, which further helps in controlling the total cost of test execution. In contrast to server-side monitoring approaches that require modifications in the application server, the ASTORIA framework does not require server-side monitoring of performance metrics. Additionally, by using headless web browsers for generating load, no modifications of the system under test are required.

IX. Conclusions

In this paper, we described different problems and challenges that are associated with generating load for cost-effective performance and scalability testing of RIAs. We then presented the ASTORIA framework as a novel solution for overcoming these challenges and problems. The effectiveness of our framework was demonstrated by presenting experimental results of an early prototype implementation. As of March 2011, performance and scalability testing with ASTORIA prototype implementation costs €2 per hour per 1000 virtual users in Amazon EC2. This shows the cost-effectiveness of our approach.

We showed that headless web browsers can be automated across several VMs in a compute cloud in order to simulate users, thus to generate load. The use of headless web browsers allowed us to conduct a test without disabling any CSRF prevention methods in the application servers. Additionally, we were able to log response times in the headless web browsers, thus eliminating the need for server side monitoring. By using test scripts recorded with Selenium IDE, we were able to generate load with the characteristics of real user traffic. A diversified user pool was simulated by introducing random variables into the recorded usage scenarios.

The paper demonstrated how ASTORIA framework is used for load testing of RIAs. In addition to load testing, the framework can also be used to perform other types of performance and scalability testing of RIAs, such as endurance testing, stress testing, and spike testing [28].

References

[1] J. Farrell and G. Nezlek, "Rich Internet Applications The Next Stage of Application Development," in *Information Technology Interfaces, 2007. ITI 2007. 29th International Conference on*, June 2007, pp. 413 –418.

[2] H. H. Liu, *Software Performance and Scalability: A Quantitative Approach*. John Wiley and Sons, May 2009.

[3] D. Menascé, "Load Testing of Web Sites," *IEEE Internet Computing*, vol. 6, pp. 70–74, July 2002.

[4] J. Shaw, "Web Application Performance Testing – A Case Study of an On-line Learning Application," *BT Technology Journal*, vol. 18, pp. 79–86, 2000, 10.1023/A:1026732502654.

[5] J. Shaw, C. Baisden, and W. Pryke, "Performance Testing – A Case Study of a Combined Web/Telephony System," *BT Technology Journal*, vol. 20, pp. 76–86, 2002, 10.1023/A:1020899610791.

[6] "Amazon Elastic Compute Cloud." [Online]. Available: http://aws.amazon.com/ec2/

[7] "Selenium." [Online]. Available: http://seleniumhq.org/

[8] "Watir." [Online]. Available: http://watir.com/

[9] "iMacros." [Online]. Available: http://www.iopus.com/iMacros/

[10] "HTMLUnit." [Online]. Available: http://htmlunit.sourceforge.net/

[11] "PhantomJS." [Online]. Available: http://www.phantomjs.org/

[12] "Xvfb." [Online]. Available: http://www.xfree86.org/4.0.1/Xvfb.1.html

[13] J. Hellerstein, Y. Diao, S. Parekh, and D. Tilbury, *Feedback Control of Computing Systems*. John Wiley & Sons, 2004.

[14] M. DeGroot, *Optimal Statistical Decisions*, ser. Wiley Classics Library. John Wiley & Sons, 2005.

[15] R. Langley, *Practical Statistics Simply Explained*, ser. Dover Books Explaining Science Series. Dover Publications, 1971, no. v. 1971, pt. 3.

[16] "Vaadin." [Online]. Available: http://vaadin.com/home

[17] "HP LoadRunner." [Online]. Available: https://h10078.www1.hp.com/cda/hpms/display/main/hpms_content.jsp?zn=bto&cp=1-11-126-17%5E8_4000_100

[18] A. Saddik, "Performance Measurements of Web Services-Based Applications," *Instrumentation and Measurement, IEEE Transactions on*, vol. 55, no. 5, pp. 1599 –1605, Oct. 2006.

[19] T. Gao, Y. Ge, G. Wu, and J. Ni, "A Reactivity-based Framework of Automated Performance Testing for Web Applications," *International Symposium on Distributed Computing and Applications to Business, Engineering and Science*, vol. 0, pp. 593–597, 2010.

[20] C. Lutteroth and G. Weber, "Modeling a Realistic Workload for Performance Testing," in *Proceedings of the 2008 12th International IEEE Enterprise Distributed Object Computing Conference*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 149–158.

[21] "Apache JMeter." [Online]. Available: http://jakarta.apache.org/jmeter/

[22] "WebLoad." [Online]. Available: http://webload.org/

[23] "PushToTest." [Online]. Available: http://www.pushtotest.com/

[24] "Load Storm." [Online]. Available: http://loadstorm.com/

[25] "Load Impact." [Online]. Available: http://loadimpact.com/

[26] "SOASTA." [Online]. Available: http://www.soasta.com/

[27] "BrowserMob." [Online]. Available: http://browsermob.com/performance-testing

[28] J. Palomäki, "Web Application Performance Testing," Master's Thesis, University of Turku, Turku, Finland, 2009. [Online]. Available: https://oa.doria.fi/bitstream/handle/10024/52532/gradu2009palomaki.pdf