Improving the Testing Efficiency of Selenium-based Load Tests

Shahnaz M. Shariff*, Heng Li*, Cor-Paul Bezemer†,
Ahmed E. Hassan*, Thanh H.D. Nguyen‡, Parminder Flora‡
*Queen's University, †University of Alberta, ‡BlackBerry, Canada
*{sharrif, hengli, ahmed}@cs.queensu.ca, †bezemer@ualberta.ca

***Abstract*—Web applications must be load tested to analyze their behavior under various load conditions. Typically, these load tests are automated using protocol-level HTTP requests (e.g., using JMETER). However, there are several disadvantages to using protocol-level requests for load tests. For example, protocol-level requests are only partially representative of the true usage of a web application, as the web application is not actually executed in a browser. It can be difficult to abstract complex behavior, such as a login sequence, into requests without executing the application. Browser-based load testing can be used as an alternative to protocol-level requests. Using a browser-based testing framework, such as SELENIUM, tests can be executed more realistically — inside a browser. Unfortunately, because a browser instance must be started to conduct a test, browser-based testing has a high performance overhead which limits its applicability for load tests. In this paper, we propose an approach for reducing the performance overhead of running SELENIUM-based load tests. Our approach shares browser instances between test user instances, thereby reducing the performance overhead that is introduced by launching many browser instances during the execution of a test. Our experimental results show that our approach can significantly increase the number of user instances that can be tested on a test machine without overloading the load driver. Our approach and the experiences that we share in this paper can help software practitioners improve the efficiency of their own SELENIUM-based load tests.**

I. INTRODUCTION

Modern web applications typically have many users that send many concurrent requests. To ensure that an application can handle the number of concurrent requests (i.e., the *load*) it is supposed to, the system must be thoroughly load tested. Load testing is performed to determine an application's behavior under various load conditions [7]. Neglecting a load test can have catastrophic consequences. For example, in 2016 a Statistics Canada website was broken for three hours because it could not handle the traffic [2].

Typically, to conduct a load test, practitioners use a tool such as JMETER that sends varying numbers of protocol-level HTTP requests to the Application Under Test (AUT). The performance of the AUT (e.g., the response time or resource utilization) is then measured throughout the test and analyzed to understand how the AUT responds to the various levels of load. However, many modern web applications rely on complex interactions within the browser, making them difficult to be abstracted into a sequence of HTTP requests without rendering the application's responses. For example, the requests that are necessary to execute a secure login sequence are hard to generate without using the application's logic.

One way to overcome this disadvantage is by using a browser-based testing framework, such as SELENIUM. SELENIUM is widely used to automate functional tests of web applications by simulating user behavior in a web browser [3, 4, 6]. Browser-based load tests have several advantages over request-based load tests. For example, the aforementioned secure login sequence can be executed easily with SELENIUM by loading the login page, filling out and submitting the form. Because the application is actually rendered inside the browser, the application logic and the browser will take care of the rest of the login sequence. Hence, as there is no need to simulate complex dynamic behavior of the web application, SELENIUM-based tests give a more realistic view on the end-to-end behavior of an application under load.

Unfortunately, there are several limitations when using a browser-based test framework for load tests. The most important limitation is the requirement of having to start a browser for each test user, which causes a considerable amount of overhead compared to request-based load tests.

In this paper, we take the first important step towards improving the efficiency of load testing using SELENIUM. We share our experiences of using SELENIUM for load testing, and we propose an approach to improve the efficiency of SELENIUM-based load testing. Our approach shares browser resources among the test users in a load test, thereby reducing the required resources per test user. The main contributions of this paper are:

- An approach that increases the number of test users in a SELENIUM-based load test by at least 20% using the same amount of hardware resources.
- A systematic exploration of various testing scenarios for SELENIUM-based load testing.

Our approach and shared experience can help software practitioners improve the efficiency of their own SELENIUM-based load tests.

The paper is organized as follows: Section II gives background information about SELENIUM. Section III presents our experimental design. Section IV presents our experimental results. Section V presents the threats to validity of our study. Section VI presents the related work and Section VII presents our conclusion.

```
# launch browser using Chromedriver
driver = webdriver.Chrome("/path/to/chromedriver")
# go to URL
driver.get("http://example.com/")
# locate element using XPATH
more_information_link = driver.\
     find_element_by_xpath("/html/body/div/p[2]/a")
# click on the element
more_information_link.click()
```

Code Listing 1: A SELENIUM test example.

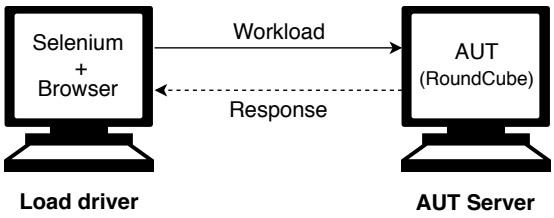

Fig. 1: Our test environment.

II. LOAD TESTING USING SELENIUM

SELENIUM[1] is a browser automation tool that is used to test the functionality of a web application by simulating the user's interactions with an actual browser. SELENIUM provides an API with which a tester can specify and replay a test scenario automatically in a modern web browser (such as Google Chrome, Mozilla Firefox or Safari). To allow for automated testing in command line-based environments (e.g., for continuous integration environments), SELENIUM supports headless browsers which provide the same functionality as regular browsers without a graphical user interface.

Listing 1 shows an example of a SELENIUM test scenario. First, a browser is opened by the SELENIUM WebDriver, the SELENIUM component that controls the browser. Second, the web application is loaded and an element on the page (a link) is located using an XPath query. Finally, the WebDriver clicks on the link.

Although SELENIUM is predominantly used for functional testing of web applications, it can be used for load tests as well. To use SELENIUM for load tests, multiple browsers are launched simultaneously (one for each user in the test). Unfortunately, this introduces considerable performance overhead, limiting the applicability of SELENIUM for load tests. In this paper, we investigate how we can improve the efficiency of executing a SELENIUM test, thereby taking an important step towards making SELENIUM more appealing for conducting load tests.

III. EXPERIMENTAL SETUP

In this section, we describe the experimental setup that we used to investigate how we can improve the efficiency of executing a test in SELENIUM. In particular, we describe our AUT, the test environment, the design of our load test, and the way in which we monitor the resource usage during the test.

A. Application Under Test

We use RoundCube[2] version 1.3.4 as our AUT. RoundCube is an open-source email client of which the front-end runs in a browser while providing an application-like user interface. It provides features such as MIME support, address book, folder manipulation, message searching and spell checking. We chose RoundCube because its front-end makes extensive use of the AJAX technology (i.e., dynamic loading of web page content) for its user interface, for example, for its drag-and-drop message management.

[1] https://www.seleniumhq.org/
[2] https://roundcube.net/

B. Test Environment

Figure 1 shows our test environment. Our test environment consisted of two dedicated desktop machines: a load driver and an AUT server. SELENIUM and the front-end of RoundCube (i.e., inside of a web browser) were executed from the same machine (i.e., the load driver), as SELENIUM must interact with the browser instances. The load driver consisted of an AMD Phenom desktop with 6 cores (2.70 GHz) and 8GB of RAM running Ubuntu 16.04. We ran SELENIUM tests using Google Chrome version 69 and the Chrome WebDriver version 2.37. We deployed RoundCube's back-end on the second machine (i.e., the AUT server) running Ubuntu Linux version 14.04, Nginx version 1.4.6, MySQL version 5.5.47 and PHP version 7.0.27. We used Postfix and Dovecot as our SMTP and IMAP servers. The back-end was installed on an Intel Core i7 (3.6 GHz) desktop with 8 cores and 16GB of RAM.

C. Load Test Design

Test Suite. In order to test our AUT, we created a test suite that consists of eight tasks covering the typical actions that are performed in an email client: composing an email, replying to an email, replying to everyone in an email, forwarding an email, viewing an email, browsing contacts, deleting an email and permanently deleting an email. Login and logout actions are added to the beginning and the end of each task, respectively.

In our load test, we simulate the scenario in which multiple users connect to an email server and perform email tasks through their web browsers. We initialized the mailbox and contacts of each user with 250 emails and 5 contacts to have a fully functioning email service.

Test Schedule. We based our test schedule on the MMB3 benchmark (Messaging Application Programming Interface Messaging Benchmark), which was designed by Microsoft for measuring the performance of Microsoft Exchange installations. Although the benchmark itself was retired in 2008 [11], the test schedule provides a realistic mix of tasks that are performed by users of an email application. The MMB3 benchmark specifies the number of times that each task is performed during a day, modelled around a typical user's working day of eight hours. According to the benchmark, 295 tasks are scheduled to run in an 8-hour period ([5]). In our study, we reduce the overall execution time while keeping the

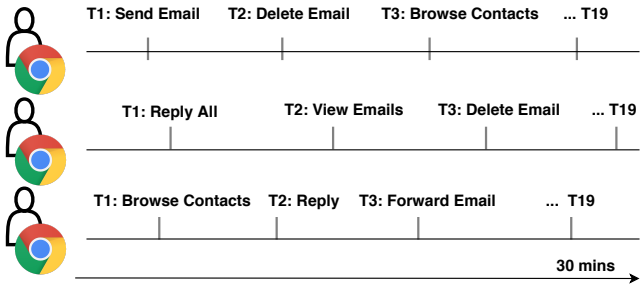

Fig. 2: The *one-user-per-browser* setting.

same intensity of tasks as performed in the MMB3 schedule. Specifically, we run 19 tasks in a 30-minute period. Each of these 19 tasks is randomly chosen (with repetition) from the 8 email tasks as specified by MMB3. Each of the tasks is randomly scheduled within the testing period using a uniform distribution. In a purely random schedule, one task might be scheduled immediately after another. However, in realistic usage, users always finish one task before starting another. Therefore, we set a minimum gap of 30 seconds between the scheduled time of two consecutive tasks. Prior to starting a load test, every user's mailbox is cleared and loaded with new emails. This initialization step is done to ensure that all emails in the mailboxes have the same status (i.e., "unread") when starting the test.

D. Configurations for Executing a Load Test

As explained in Section II, SELENIUM employs browsers to test a web application.Traditionally, one dedicated browser instance is opened for each user instance (e.g., each user of the email application in our load test). However, this approach is very resource-heavy. In this paper, we experiment with several configurations for executing a load test. In particular, we vary (1) the number of users per browser instance and (2) the type of browser instance that is used. Table I gives an overview of the configuration settings that we used in our study. Below we describe each setting.

1) The number of users per browser instance: We use two settings for the number of users per browser instance. In the *one-user-per-browser* setting, which is traditionally used in SELENIUM load tests, there is one dedicated browser instance for each user that remains open while there are still tasks left for the user to execute. This setting is depicted by Figure 2.

In this paper, we propose the *many-users-per-browser* setting, in which a browser instance is shared between users in the load test. In this setting, a scheduler is employed that selects the next task to execute and assigns it to a browser instance from a pool of available browser instances. The scheduler has a separate thread for each browser instance in the pool. The rationale behind this setting is that in the *one-user-per-browser* setting, there is a considerable amount of idle time in which a browser is not used (and hence is wasting resources). In the *many-users-per-browser* setting, a user can execute a task during the idle time of other users of the browser.

Sharing browsers between users. To share browsers between user instances, we first combine the schedules of all the user

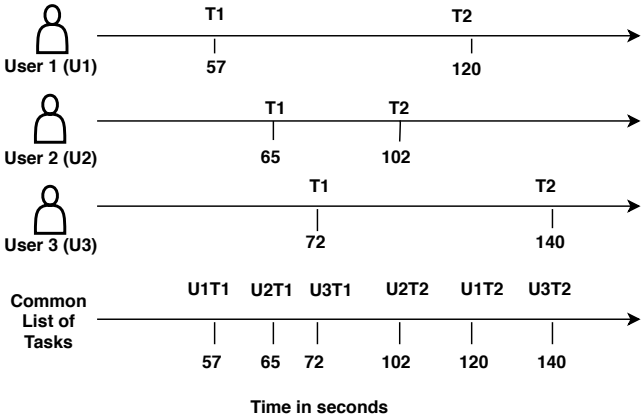

Fig. 3: Merging the scheduled tasks of two user instances into a common list of tasks.

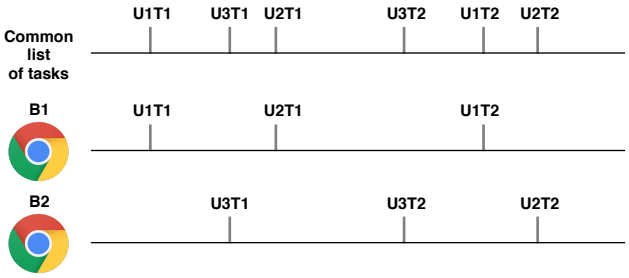

Fig. 4: The *many-users-per-browser* setting.

instances in a common list of tasks. The tasks in the common list are sorted based on their scheduled starting time (starting from the earliest task). Figure 3 shows how the tasks are gathered from three user instances to form a common list of tasks.

The scheduler selects the next task from the common list, and executes it in an available browser instance (as illustrated in Figure 4). Figure 5 shows how a task is selected and assigned to a browser. While a browser is executing a task, it is removed from the pool of available browsers. Once the task is finished, the browser is added back to the pool. One can experiment with the size of the pool of available browsers to make optimal use of the available resources. The process of assigning tasks to available browsers are running in multi threads (the number of threads is equal to the number of available browsers), such that all the browsers are continually running the tasks in parallel until all the tasks in the common list are executed.

To avoid conflicts between user tasks (e.g., when reading and deleting an email at the same time for a user), the scheduler also maintains a pool of available users. When a user task is assigned to a browser instance, that user is removed from the available pool. When the user for a scheduled task is not available, the scheduler selects the next task from the task list. The current task is not ignored; it is selected later when the user for the task becomes available.

2) The type of browser instance: As shown in Table I, we use three settings for the type of browser instances in our experiments. In the *regular browser* setting, we use the

TABLE I: The configuration settings for executing a load test that are used in our study.

	Concept	Definition
Number of users per browser instance	One per browser	A dedicated browser instance is started for each user instance
	Many per browser (our proposed approach)	Browsers are shared among several user instances
Type of browser	Regular browser	A regular browser (e.g., Google Chrome)
	Headless browser	A simplified browser without a graphical user interface
	XVFB browser	A regular browser with its display transferred to an in-memory display

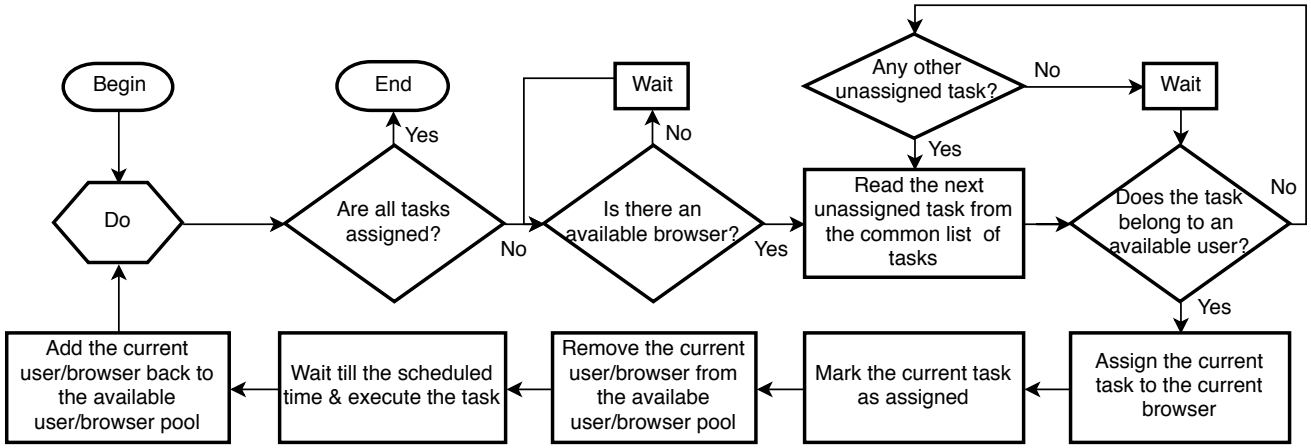

Fig. 5: Our process of assigning a task from the common task list to an available browser. The process is running in multi threads (the number of threads is equal to the number of available browsers).

Chrome browser. In the *headless browser* setting, we use the headless version of the Chrome browser. As not all modern browsers have headless versions, we also consider the XVFB *browser* setting in our experiments, which is considered an alternative to the headless approach [1]. In the XVFB *browser* setting, we use a regular Chrome browser of which the display is transferred to an in-memory display using the XVFB application.

E. Performance Metrics for Monitoring the Resource Usage

To compare the aforementioned configurations for executing a load test, we use the following metrics.

- **The CPU & memory usage.** The combined CPU and memory usage of the load testing processes (SELENIUM, Chrome, ChromeDriver, and XVFB when using XVFB browsers) running on our load driver machine. We monitor the CPU and memory using the `pidstat`[3] application. We calculate the median and the 95th percentile values of the CPU and memory usage recorded at every second. The median resource usage values give an overall estimate of the used resources during a load test, while the 95th percentile values show the spikes of resource usage (or peak usage). We also monitor the overall system CPU and memory usage, using the `sar`[4] application, to understand the overall status of the load driver system.

[3]https://linux.die.net/man/1/pidstat
[4]https://linux.die.net/man/1/sar

- **Error ratio.** The error ratio is the proportion of tasks with execution errors. We assume that the implementation of the load test tasks is functionally correct for one user. Hence, if errors occur during the execution of a task during the load test, it is likely that these errors are caused by the load. We use the error ratio as the primary metric for determining the maximum number of user instances that can run on a load driver.
- **Delay ratio.** The delay ratio captures the proportion of tasks that missed their scheduled starting time. When the load driver is overloaded, tasks may take longer to finish, which could cause the next tasks to get delayed (i.e., miss their scheduled starting time). In order to follow the test schedule, we need to minimize the proportion of delayed tasks (i.e., less than 5%).
- **Maximum number of error-free user instances.** The maximum number of error-free user instances is the maximum number of user instances that can run on the load driver machine without overloading it. We increase the number of user instances in the load test until these thresholds are reached to identify the maximum number of error-free user instances. We repeat the load test five times to reduce variation in the measurements. We identify the maximum number of user instances when the median error ratio is 0 and the median delay ratio is less than 5% among the five repetitions. We use a different random schedule for each repetition to ensure that the results are not biased towards a certain task schedule. The random schedules are recorded to ensure that the same

TABLE II: The resource usage of the load test for each configuration setting for the type of browser instance.

Browser type	Median CPU (%)	Median memory (%)
Headless	49	5
Regular	121	12
XVFB	92	8

schedules are used across different experimental settings.

IV. Studying The Resource Usage of Selenium-based Load Tests

In this section we present our experimental results. First, we study the resource usage of SELENIUM-based load tests for each configuration setting for the type of browser instance. Second, we study the resource usage of SELENIUM-based load tests for each configuration setting for the number of users per browser instance.

A. Studying the Resource Usage of Different Types of Browsers

Approach: We run the load test discussed in Section III for all three configuration settings for the type of browser instance. To be able to compare the results across the settings, we fix the number of user instances to ten in this experiment. We launch a browser for each user instance. Hence, for each configuration setting, we start ten browser instances.

In addition, we study the resource usage of a *busy* and an *idle* browser for each browser type, to investigate how many resources are wasted by having idle browser instances. In this experiment, we run ten browser instances for ten minutes and monitor the resource usage. During these ten minutes, the busy browsers execute tasks one after another, and the idle browsers execute no tasks.

Results: **SELENIUM-based load tests with headless browsers use the least resources in terms of CPU and memory.** Table II shows the CPU and memory usage for each of the browser types. SELENIUM-based load tests with regular browsers use the most resources in terms of CPU and memory. `Xfvb` browsers use less resources than regular browsers but more resources than headless browsers. Hence, headless browsers are best suited for SELENIUM-based load testing in terms of resource usage. As regular browsers require considerably more resources, we focus the remainder of our study on headless browsers and XVFB browsers.

The peak resource usage of idle browsers is non-negligible compared to the peak resource usage of busy browsers. Table III compares the results of running busy and idle browsers. The median CPU and memory usage shows that running an idle browser consumes significantly fewer resources than running a busy browser. For example, running busy headless browsers consumes a median value of 328% CPU while running idle headless browsers consumes only a median of 2% CPU. However, Table III also shows the 95th-percentile CPU and memory usage, which can be considered the peak resource usage. These peak numbers show that idle XVFB browsers consume a 95th-percentile memory of 14%, which is more than one-fourth of the 95th-percentile memory (51%) of busy XVFB browsers. The high peak resource usage of idle browsers suggests that sharing browsers could reduce the peak resource usage of SELENIUM-based load tests, as the total number of browsers would be reduced. As a result, the capability of the load driver for generating workload would be increased.

TABLE III: The resource usage of headless and XVFB browsers in busy and idle status.

Browser type	CPU usage (%)		Memory usage (%)	
	Median	95th perc.	Median	95th perc.
Headless (busy)	328	454	28	34
Headless (idle)	2	67	2	9
XVFB (busy)	513	547	46	51
XVFB (idle)	1	76	6	14

> Headless browsers consume considerably less resources than other types of browser instances in SELENIUM-based load tests. In addition, even idle browsers can consume a significant amount of CPU and memory during a SELENIUM-based load test.

B. Studying the Resource Usage of Different Settings for the Number of Users per Browser

Approach: In the remainder of this section, we study the resource usage of SELENIUM-based load tests using the *one-user-per-browser* and *many-users-per-browser* configuration settings. In these experiments, we execute the load test as described in Section III. Hence, we keep increasing the user instances until we exceed the thresholds for the error and delay ratio.

Results: **Sharing browsers in SELENIUM-based load tests increases the maximum number of error-free user instances by 20% when using headless browsers.** The left side of Table IV shows the results of our experiments using headless browsers. The performance measures are the median values over the five repetitions of the load test (as discussed in Section III). For the *one-user-per-browser* setting, we started to see execution errors when the number of user instances reached 50. In comparison, for the *many-users-per-browser* setting (using 20 shared browsers), the error ratio was still zero when we ran 60 user instances on the same load driver. The delay ratio was zero in both cases. For 65 users, the performance metrics exceeded the thresholds for the error and delay ratio. Therefore, our proposed approach of sharing browsers increases the maximum number of error-free user instances from less than 50 to 60 (i.e., by at least 20%).

Table IV also shows the performance of the load tests with the *one-user-per-browser* and *many-users-per-browser* settings when both running 60 user instances using headless browsers. The median CPU, median memory, and 95th-percentile CPU usage is similar between the two settings. However, the *many-users-per-browser* setting uses significantly less 95th-percentile memory (41.4%) than the *one-user-per-browser*

TABLE IV: The performance of the load tests for each configuration setting for the number of users per browser, using headless and XVFB browsers. The threshold for determining whether the load driver is overloaded is 0% for the error ratio and 5% for the delay ratio.

	Headless browsers			XVFB browsers		
	One user per browser (**50 users**)	One user per browser (**60 users**)	Many users per browser (**60 users**)	One user per browser (**18 users**)	One user per browser (**22 users**)	Many users per browser (**22 users**)
Error ratio	**0.1**	**1.6**	**0.0**	**0.9**	**0.7**	**0.0**
Delay ratio	**0.0**	**0.0**	**0.0**	**0.0**	**1.0**	**1.0**
Median CPU	222.0	259.0	265.0	299.0	357.0	362.0
Median memory	23.2	27.6	28.3	24.6	27.7	27.1
95th percentile CPU	385.0	429.0	418.0	485.0	506.0	501.0
95th percentile memory	45.1	51.7	41.4	40.9	48.4	40.0
Median system CPU	349.0	383.0	381.0	429.0	490.0	486.0
Median system memory	93.4	92.3	53.7	93.4	96.8	84.2
95th percentile system CPU	497.0	537.0	514.0	555.0	589.0	584.0
95th percentile system memory	97.4	97.6	60.1	96.4	98.2	87.5
Load driver overloaded?	Yes	Yes	No	Yes	Yes	No

- All values are in %
- Notice that there are 6 cores available on the load driver machine, hence total CPU usage can go up to 600%

setting when running 60 user instances (51.7%), and even less than the *one-user-per-browser* setting when running 50 user instances (45.1%). When looking at the system-level resource usage, it shows that the *one-user-per-browser* setting uses a median of more than 92% memory (when running either 50 user instances or 60 users). In comparison, our proposed *many-users-per-browser* setting uses only 53.7% system memory in median. This large difference in overall system memory usage is due to the overhead that is required for the operating system to manage the open browser instances.

Sharing browsers in SELENIUM-based load tests increases the maximum number of error-free user instances by 22% when using XVFB browsers. The right side of Table IV shows the results of our experiments using XVFB browsers. In the *one-user-per-browser* setting, execution errors start to occur from 18 user instances, while the delay ratio is still zero. In the *many-users-per-browser* setting (using 10 browser instances), the error ratio is still zero for 22 user instances. The delay ratio increases to 1%, but this is still below our threshold. Therefore, sharing browsers between user instances increases the maximum number of error-free user instances from less than 18 to 22 (i.e., by 22%) when using XVFB browsers.

Table IV also shows the performance of the load tests with the *one-user-per-browser* and *many-users-per-browser* settings when both running 22 user instances using with XVFB browsers. Similar to the results for headless browsers, the *many-users-per-browser* setting with 22 user instances uses less 95th-percentile memory (40.0%) than the *one-user-per-browser* setting with 60 user instances (48.4%) and 50 user instances (40.9%). When looking at the overall resource usage of the system, we noticed that the median memory usage is more than 93% in the *one-user-per-browser* setting with either 18 user instances or 22 user instances. In comparison, the median memory usage is just 84.2% in the *many-users-per-browser* setting with 22 user instances.

> Sharing browsers between user instances in a SELENIUM-based load test increases the capability of the load driver for generating workload by at least 20%.

V. THREATS TO VALIDITY

Generalization of our approach to other AUTs and test schedules. We tested various configurations for executing a SELENIUM-based load test on a single AUT (i.e., RoundCube). Future studies should investigate how these configurations perform for other AUTs and in different test environments.

We designed a test schedule that was inspired by the MMB3 benchmark. We assumed that the individual tasks of a user instance are not executed directly after each other. Future work should study the resource usage of SELENIUM-based load tests with other test schedules.

Generalization of our approach to other browser-based test automation. The experimental results may vary for other browsers (e.g., Firefox). However, our approach of sharing browsers is not limited to a specific browser.

Cypress[5] is a framework that supports testing of web applications running in a Chrome browser. However, Cypress does not support running multiple instances of browsers. Therefore, we did not consider Cypress in this work.

Thresholds for detecting if the load driver is overloaded. We used thresholds for the error ratio and delay ratio to identify whether the load driver is overloaded. By not allowing any errors and only a low delay ratio, we set these thresholds fairly strict. Some web applications may prefer using other thresholds. Future studies should do a sensitivity analysis of the thresholds and search for generic optimal thresholds.

[5] https://www.cypress.io

VI. Related work

In this section, we give an overview of related work on Selenium-based test automation and load testing.

Several prior studies discussed automated test generation methodologies in Selenium using a combination of human written scripts and crawlers to fetch the dynamic states of the application [8, 9, 10]. The performance issues of Selenium were discussed by Vila et al. [12]. They highlighted that the Selenium WebDriver consumes a large amount of resources as the whole application needs to be loaded in the browser (including all the images, CSS and JavaScript files). Our experimental results confirm that Selenium-based testing is resource-intensive. Therefore, we proposed to share browsers between user instances to improve the efficiency of Selenium-based load testing.

There exists a large body of prior work on load testing, which was summarized by Jiang and Hassan [7]. However, this body of work has always focused on testing how the AUT responds to various levels of load. The focus of our work is quite different, as we focus on how we can improve the efficiency of the load driver, the component that generates load for the AUT.

To the best of our knowledge, we are the first to systematically study how Selenium can be used for load testing. Dowling and McGrath [4] suggested that Selenium can be used next to a request-based load testing framework, such as `JMeter`. However, we are the first to suggest a load testing framework that solely uses Selenium.

VII. Conclusion

Request-based frameworks for load testing such as `JMeter` are the de facto standard for executing load tests. However, browser-based load tests (e.g., using Selenium) have several advantages over such request-based load tests. For example, browser-based load tests can simulate complex user interactions within a real browser. Unfortunately, browser-based load testing is very resource heavy, which limits its applicability.

In this paper, we studied the resource usage of Selenium-based load tests in different configurations for executing the load test. Our most important findings are:

- Headless browsers consume considerably less resources than other types of browser instances.
- The capacity of a load driver (in terms of the number of users that it can simulate) can be increased by at least 20% by sharing browser instances between user instances.

We took the first important step towards more efficient load testing in Selenium. Practitioners can use our approach as a foundation to improve the capacity of the load drivers of their own browser-based load tests.

Acknowledgment

We are grateful to BlackBerry for providing valuable support and suggestions for our study. The findings and opinions expressed in this paper are those of the authors and do not necessarily represent or reflect those of BlackBerry and/or its subsidiaries and affiliation. Our results do not in any way reflect the quality of BlackBerry's products.

References

[1] BlazeMeter (2016). Headless Execution of Selenium Tests in Jenkins. https://www.blazemeter.com/blog/headless-execution-selenium-tests-jenkins. (Accessed on 02/01/2019).

[2] Census (2016). Census 2016: IT experts say Bureau of Statistics should have expected website crash. https://www.smh.com.au/national/census-2016-it-experts-say-bureau-of-statistics-should-have-expected-website-crash-20160809-gqosj7.html. (Accessed on 02/01/2019).

[3] Debroy, V., Brimble, L., Yost, M., and Erry, A. (2018). Automating web application testing from the ground up: Experiences and lessons learned in an industrial setting. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 354–362.

[4] Dowling, P. and McGrath, K. (2015). Using free and open source tools to manage software quality. *Queue*, 13(4):20:20–20:27.

[5] Exchange (2005). Exchange performance result. https://www.dell.com/downloads/global/solutions/poweredge6850_05_31_2005.pdf. (Accessed on 02/01/2019).

[6] Gojare, S., Joshi, R., and Gaigaware, D. (2015). Analysis and design of Selenium WebDriver automation testing framework. *Procedia Computer Science*, 50:341 – 346. Big Data, Cloud and Computing Challenges.

[7] Jiang, Z. M. and Hassan, A. E. (2015). A survey on load testing of large-scale software systems. *IEEE Transactions on Software Engineering*, 41(11):1091–1118.

[8] Milani Fard, A., Mirzaaghaei, M., and Mesbah, A. (2014). Leveraging existing tests in automated test generation for web applications. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, pages 67–78. ACM.

[9] Mirshokraie, S., Mesbah, A., and Pattabiraman, K. (2013). Pythia: Generating test cases with oracles for JavaScript applications. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 610–615.

[10] Stocco, A., Leotta, M., Ricca, F., and Tonella, P. (2015). Why creating web page objects manually if it can be done automatically? In *2015 IEEE/ACM 10th International Workshop on Automation of Software Test*, pages 70–74.

[11] The Exchange Team (2007). MAPI Messaging Benchmark Being Retired. https://blogs.technet.microsoft.com/exchange/2007/11/06/mapi-messaging-benchmark-being-retired/. (Accessed on 02/01/2019).

[12] Vila, E., Novakova, G., and Todorova, D. Automation testing framework for web applications with Selenium WebDriver: Opportunities and threats. In *Proceedings of the International Conference on Advances in Image Processing*, ICAIP 2017, pages 144–150. ACM.