Investigating Selenium usage challenges and reducing the performance overhead of Selenium-based load tests

by

Shahnaz Mohammedi Shariff

A thesis submitted to the

School of Computing

in conformity with the requirements for

the degree of Master of Science

Queen's University

Kingston, Ontario, Canada

February 2019

Copyright © Shahnaz Mohammedi Shariff, 2019

Abstract

SELENIUM automates the testing of web applications. Quality Assurance (QA) engineers usually write Selenium scripts to test the content that is rendered in a browser and simulate user interactions with the web content for web applications. Apart from test automation, Selenium is also used to perform load testing and web scraping tasks. Selenium has gained a great amount of attention from users as observed from the trend of Selenium-tagged questions on StackOverflow (a popular online development forum). The percentage of StackOverflow questions that are tagged with “Selenium" has been constantly increasing (i.e., it increased six folds, from 2011 to 2018), reaching around 0.6% in 2018.

In this thesis, we first explore the Selenium-related questions on StackOverflow to understand users' questions about Selenium, programming-language specific issues, browser-specific concerns and the difficult aspects of Selenium. Some of the notable findings from our empirical study are: Questions about Python have become

the most frequent and the fastest growing programming language related questions in Selenium, while questions on Java have become stable in recent years. We also find that Chrome tag is the most frequently used browser tag among other browser related tags associated with Selenium. However, different browsers may be used for different purposes. For example, PhantomJS widely used for web scraping while Firefox is frequently used with Selenium-IDE.

Several users discuss performance issues in Selenium tests on StackOverflow. Prior work on Selenium also points out that Selenium tests are resource intensive. As Selenium needs to launch a browser for each instance (e.g., a client) of a test, it is usually very resource consuming to run a load test using Selenium as one would need to run thousands of user instances. Therefore, we investigate the challenges in Selenium-based load tests by performing experiments using different type of browsers. We propose an approach to improve the testing efficiency of Selenium-based load tests. Our approach shares browser instances between user instances, thereby reducing the performance overhead that is introduced by launching many browser instances. The main contribution of this thesis is that our approach can significantly increase the number of user instances that can be tested on a test driver machine without overloading the machine. We improve the testing efficiency of Selenium-based load tests by at least 20%.

The insights that we share in this work can help the developers of Selenium understand how Selenium is used in practice and how they can offer better support for Selenium. Further, software practitioners can benefit from our approach to improve the efficiency of Selenium-based load tests.

Acknowledgments

I extend my greatest gratitude to Prof. Ahmed E. Hassan for being the best supervisor. His guidance and mentorship allowed me to get this thesis in shape. I am extremely grateful to have worked under him. I would like to thank everyone at SAIL (Software Analysis & Intelligence Lab), for creating a friendly and supportive environment to work in. I would like to especially thank my collaborators and mentors, Dr. Heng Li and Prof. Cor-Paul Bezemer. Their advice, critical comments and timely feedback helped me immensely.

My heart goes out to my mother (Nasreen) and my brother (Sarfraz), my greatest cheerleaders and support systems. I would like to dedicate my work to my father (Late. M.N Shariff). I hope that this achievement will complete the last dream that you had for me. Last but not the least, I would like to thank these Good Samaritans - Dev, Ganga, Gopi and my other family members for their constant support during my masters journey.

Table of Contents

Abstract i

Acknowledgments iii

List of Tables vi

List of Figures vii

1 Introduction 1
- 1.1 Thesis Statement 3
- 1.2 Thesis Overview 3
- 1.3 Thesis Contributions 5

2 Background and Related Work 7

3 An Empirical Study of Selenium Questions on Stack Overflow 12
- 3.1 Case Study Setup 16
- 3.2 Case Study Results 19
- 3.3 Threats to Validity 43
- 3.4 Conclusion 46

4 Reducing the Performance Overhead of Selenium-based Load Tests 47
- 4.1 Load testing using Selenium 50
- 4.2 Experimental Design 51
- 4.3 A systematic exploration of the performance overhead of Selenium-based load testing 64
- 4.4 Experimental Results 67
- 4.5 Threats to Validity 72
- 4.6 Lessons learned 75
- 4.7 Conclusion 78

5 Conclusions and Future Work 80

5.1 Summary 80
5.2 Future Work 82

Bibliography **84**

List of Tables

3.1 Stats for users in each of the programming language related tags in Selenium 31
3.2 Stats for the user communities of each of the browser and driver tags in Selenium 36

4.1 Browser definitions 55
4.2 Maximum number of user instances in each experimental setting 61
4.3 Performance measures in headless browsers 69
4.4 Performance measures in regular browsers (with Xvfb) 69

List of Figures

3.1 Trend analysis of 'Selenium' tag 13
3.2 Data extraction process 17
3.3 Most popular Selenium-related tags 23
3.4 Relationship between Selenium-related tags 24
3.5 Attention received by Selenium-related tags. The dotted lines indicate the median attention for each group. 25
3.6 Evolution of questions added in each group 27
3.7 Tag association between programming languages 29
3.8 Evolution of the asked questions in each of the programming language related tags 32
3.9 Tag association between browsers (combined with drivers) 35
3.10 Evolution of questions added in each browser (with the corresponding drivers) 37
3.11 Percentage of questions with accepted answers for each of the Selenium-related tag. The dotted lines indicate the median percentage of accepted answers for each group. 40
3.12 Answering speed for the top 35 Selenium-related tags. The dotted lines indicate the median time to get accepted answer (in hours) for each group. 42

4.1 Experimental Setup 51
4.2 Independent browsers setting - Persistent browsers 53
4.3 Independent browsers setting - Non-persistent browsers 57
4.4 The merging of tasks across different user instances to produce a common list of tasks 57
4.5 Steps to execute scheduled tasks in each browser 60
4.6 Our approach of using shared persistent browsers 60
4.7 Our approach of using fixed number of non-persistent browsers 60
4.8 Median CPU and Memory in different browsers 65
4.9 Median CPU and memory values for headless and regular browsers . . . 65
4.10 95th percentile CPU and memory values for headless and regular browsers 66

CHAPTER 1

Introduction

SELENIUM is becoming increasingly popular among developers and testers as it can be used for various software engineering tasks such as test automation, web-scraping, and load testing. The popularity of Selenium is evident from the percentage of Stack Overflow questions that are tagged with “Selenium" as it has been constantly increasing (i.e., it increased six folds, from 2011 to 2018), reaching around 0.6% in 2018. Until Nov 2018, there are 73,617 Selenium-related questions and more than 100,000 Selenium-related answers on StackOverflow.

Analysis of Selenium-related questions raised by thousands of users is essential for the developers of Selenium to understand the Selenium-related issues and how these issues are related to each other. Further, the programming languages and browsers supported by Selenium have to be studied to learn how each of them is used in practice

along with Selenium. The developers of Selenium can benefit from knowing which Selenium-related questions require a long time to get an answer and the factors that impact the speed of receiving a response to such questions.

The performance overhead of Selenium tests has been discussed by the users on StackOverflow (StackOverflow, 2017, 2018b,a). For example, one of the questions discusses the high memory consumption of Chrome instance and the overall CPU consumption of the server during the load test StackOverflow (2018c). Further, prior work (Vila et al. (2017)) also highlight that Selenium tests can be both time and memory intensive. Selenium tests consume a large amount of resources as browsers are launched for each test run. Therefore, load-testing using Selenium is extremely resource intensive as one would need to run thousands of browser instances to simulate thousands of user instances. Traditional load testing tools, like JMeter, generate load for thousands of user instances using HTTP requests. However, they fail to capture real usage scenarios or enable the verification of the rendered JavaScript of web applications.

As web applications are becoming increasingly more complex, browser-based load testing has to be performed instead of protocol-level load testing. Moreover, simulating and replaying even a secure login into a user session is extremely challenging using protocol-level load-testing as is done with JMeter. Further, the importance of using Selenium over JMeter has been discussed by Dowling and McGrath (2015). They discuss that JMeter cannot be used to check the rendering of a webpage and for executing Javascript. Due to these reasons, Selenium is a better choice for load-testing as it can effectively render JavaScript, test real usage scenarios and it is much easier than hard-coding protocol level HTTP requests. However, there is no tool or technique that improves the testing efficiency of Selenium-based load tests in order to increase the

number of user instances that can run on test driver machines.

1.1 Thesis Statement

Selenium-related questions on StackOverflow offer an overview of the challenges facing practitioners who use Selenium. Moreover, since protocol-level load testing tools cannot be used to verify the complex behavior of web applications, reducing the performance overhead of Selenium tests would enable us to effectively leverage browser-based tests for load testing.

1.2 Thesis Overview

In this section, we provide an outline of our thesis.

1.2.1 Chapter 2: Background and Related work

This chapter gives background information about the Selenium tool. We also provide an overview of prior research that is related to our work. In particular, we focus on prior research in the following three areas: (1) Selenium (2) Load testing (3) Stack Overflow data.

1.2.2 Chapter 3: An Empirical Study of Selenium Questions on Stack Overflow

In this chapter, we aim to understand the issues in Selenium and how the issues are related to each other. We provide a detailed analysis of Selenium-related questions that

are related to programming-languages and browsers. We also analyze which Selenium-related questions take a long time to receive an answer and the factors that are associated with such delayed answers. We observe that questions on programming languages are the most popular and fastest growing type of Selenium-related questions while browser-related questions concerning Selenium get the most attention. Python tag is the fastest growing among other programming language related tags, while Java tag has become stable in recent years. We also find that Chrome tag is the most frequently used browser tag among other browser tags related to Selenium. However, different browsers may be used for different purposes. For example, PhantomJS is widely used for web scraping while Firefox is frequently used with Selenium-IDE. We observe that less than half of Selenium-related questions get accepted answers. The time taken to get an accepted answer is statistically significantly impacted by the number, the median reputation, and the experience level of the answerers of each tag. Our findings could help the developers of Selenium and users understand how Selenium is used in practice and provide better support for Selenium.

1.2.3 Chapter 4: Reducing the Performance Overhead of Selenium-based Load Tests

In this chapter, we propose an approach to reduce the resource usage of Selenium-based load testing. Our approach shares browser instances between user instances, thereby reducing the performance overhead that is introduced by launching too 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 driver machine without overloading the machine. Our approach and our

experience can help software practitioners improve the efficiency of Selenium-based load testing.

1.3 Thesis Contributions

In this thesis, we study the Selenium-related questions on StackOverflow to understand the difficult aspects of Selenium and Selenium usage in practice. We also discuss the advantages of using browser-based load tests (using Selenium) over protocol level load tests (using JMeter). We propose an approach that shares browser instances among tested clients to reduce the performance overhead associated with launching many browser instances for load testing. In particular, our main contributions are as follows:

1. We present a detailed analysis of Selenium-related questions on StackOverflow. We believe that new users of Selenium will benefit from this study to know which programming language and browser to use depending on the use case. Our insights will also help Selenium developers prioritize their efforts to address the difficult aspects in Selenium.

2. This thesis is the first work to propose an approach that increases the number of user instances in Selenium-based load tests by at least 20% using the same hardware resources.

3. We provide a systematic exploration of various testing scenarios (i.e. headless vs. regular browsers and persistent vs. non-persistent browsers) for Selenium-based load testing.

4. We share our experience of using Selenium for load testing including the challenges encountered in designing Selenium scripts, our investigations of the performance issues in Selenium, and how we addressed them.

CHAPTER 2

Background and Related Work

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. User interactions for simpler applications such as telnet, ftp, passwd etc can be automated using tools such as Expect (Expect (2018)). Whereas, Selenium is used to automate the testing of modern web applications. Selenium is commonly used for testing by popular web applications such as Bugzilla, Mozilla Add-ons, Jenkins, and Wikia. Selenium consists of a set of different software tools each with a different approach to support test automation.

Selenium is composed of multiple software tools: Selenium Webdriver, Selenium

[1] https://www.seleniumhq.org/

Remote Control (RC), Selenium Integrated Development Environment (IDE) and Selenium Grid (Altaf et al. (2015)). Selenium 2 supports the Webdriver API along with the Selenium RC technology to provide flexibility in porting tests written for Selenium RC. One of Selenium's key features is the support for executing one's tests on multiple modern browsers such as Google Chrome, Mozilla Firefox, and Safari. The Webdriver uses browser-specific libraries (i.e., browser drivers) to launch and control browsers. For example, the Webdriver uses the Chromedriver to control a Chrome browser.

Listing 2.1: Selenium Code Snippet

```
# 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()
```

The Selenium WebDriver enables testers to write scripts to launch a browser, simulate various human interactions with the browser and support various approaches to verify the loading results of web pages. Selenium provides various locator strategies (to locate web elements) such as CSS selectors, XPath, ID, class name etc. Human interactions are simulated by locating web elements (using one of the locator strategies) and performing suitable actions on them (e.g., click a button). Code listing 3.2 shows

a sample test snippet. After the browser is launched, we navigate to a URL, locate an element using a suitable locator strategy (XPath in this case) and then click on the element.

Selenium tests are verified by using asserts that look for specific text, images, or other web elements. The verification checks depend on the Application Under Test (AUT) and the goal of the test case.

2.0.1 Prior research in Selenium

A large amount of Selenium-related studies focuses on automated test generation (Milani Fard et al., 2014; Stocco et al., 2015; Mirshokraie et al., 2013). Tests are generated using a combination of human written scripts and crawlers to fetch the dynamic states of the application. Researchers also discuss the challenges and drawbacks of Selenium, motivating them to develop new tools and frameworks (Gojare et al., 2015; Le Breton et al., 2013). Other studies on Selenium are experience reports (Dowling and McGrath, 2015; Debroy et al., 2018; Arcuri, 2018). Leotta et al. (2013) compare the maintenance effort of several locator strategies that Selenium provides. Kongsli (2007) show how Selenium can be used for Security testing. Our work is different from the aforementioned work since we analyse the Selenium-related questions and answers that are posted on StackOverflow. While prior work uncovers some of the limitations of Selenium based on the experiences of using it, our work analyzes Selenium-related questions and their associated answers that are posted by thousands of users on Stack Overflow.

The performance overhead of Selenium has been discussed by Vila et al. (2017). They highlight 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 js files). Our experimental results also show that Selenium-based testing is resource-intensive. Therefore, we propose an approach to improve the efficiency of Selenium-based load tests.

2.0.2 Prior research in load testing

Load testing is usually performed to determine a system's behavior under anticipated peak load conditions (Jiang and Hassan (2015)). Prior work proposes various approaches to simulate workloads in a load test. Step-wise load is one of the approaches where the load is increased steadily to study the stability of the Application Under Test (AUT). Neves et al. (2013) use step-wise workloads to study the performance of JAVA and PHP web server technologies. They identify that JAVA can handle a large number of simultaneous requests better than PHP. Similarly, in a study by Goldschmidt et al. (2014), the authors gradually added virtual users to the system. This was done in order to evaluate the scalability of time series databases in cloud-based systems. In this thesis, we employ a similar approach. We increase the number of user instances in small increments (2, 5 or 10) to identify the maximum number of error-free user instances.

Using JMeter for load testing is widely discussed in the past. For example, Abbas et al. (2017) compare the performance of popular load testing tools such as Apache JMeter, Microsoft Visual Studio (TFS), LoadRunner and Siege. Kiran et al. (2015) discuss the associated challenges of using JMeter for applications that use Unified Authentication Platform (UAP). However, there is no work on Selenium-based load testing. To the best of our knowledge, there exists no prior work that proposes an approach to use Selenium for load testing. As web applications are becoming increasingly more complex, browser-based load testing has to be performed instead of protocol-level

load testing. Moreover, simulating and replaying something like a secure login into a user session is extremely challenging using protocol-level load-testing as is done with JMeter. Further, the importance of using Selenium for load testing over JMeter has been discussed by Dowling and McGrath (2015). They note that JMeter cannot be used for checking the rendering of a webpage and for executing Javascript. Therefore, Selenium tests would have to be written separately to check rendering of the webpage and the execution of Javascript.

2.0.3 Prior studies on Stack Overflow

StackOverflow data has been widely analyzed in prior work. Some studies (Barua et al., 2012; Ponzanelli et al., 2014), use all the posts on SO within a certain time period and focus on providing insights on the entire user-base. Several prior studies analyze Stack Overflow data on one domain or aspect of Software engineering. For example, Venkatesh et al. (2016) perform an empirical study on 32 Web APIs to study the concerns of Web developers; Abad et al. (2016) study posts on requirements engineering; Yang et al. (2016) study security related questions on StackOverflow. Similarly, we study questions and answers related to Selenium. The work closest to ours is the work by Kochhar (2016). The author analyzes software testing related questions on StackOverflow by focusing on Software testing questions with the *test* tag. In comparison, we perform a detailed study of Selenium-related questions. For example, we analyze the difference between the Selenium questions that are associated with different programming languages or different browsers.

CHAPTER 3

An Empirical Study of Selenium Questions on Stack Overflow

Selenium [1] is a popular browser-based test automation tool that is used to test the functionality of web applications. Selenium supports the testing of web applications running on different browsers, such as Firefox and Google Chrome. Selenium provides direct support for programming languages such as Java, Python, C#, Ruby and provides third-party support for programming languages such as JavaScript, Perl, and PHP. While Selenium is predominantly used for test automation, it is being used for web-scraping as well (Chaulagain et al. (2017)).

Selenium started in 2004 as "JavaScriptTestRunner" and then the Selenium Remote-Control (RC) technology came into the picture. Selenium RC was followed by Selenium IDE and finally the WebDriver. Selenium RC uses a server to launch browsers and acts

[1] https://www.seleniumhq.org/

Figure 3.1: Trend analysis of ‘Selenium’ tag

as an HTTP proxy for web requests that arrive from the browsers. Selenium IDE is used to record scripts and replay them for later use. Selenium IDE comes as a plugin in popular browsers such as Chrome and Firefox. The WebDriver is the latest addition to the Selenium toolkit. The implementation of the WebDriver provides all the functions to write test automation scripts, such as the functions used to launch browsers, navigate to URLs and locate web-elements. Selenium is being used to test popular systems such as Bugzilla, [2] Mozilla Add-ons, [3] and Wikia [4].

Selenium-related issues are widely discussed on Stack Overflow. Until Nov 2018, there are 73,617 Selenium-related questions and more than 100,000 Selenium-related answers. Figure 3.1 shows that the percentage of Stack Overflow questions tagged with

[2]https://github.com/mozilla-bteam/bmo/tree/master/qa/t
[3]https://github.com/mozilla/addons-server/tree/master/tests/ui
[4]https://github.com/Wikia/selenium-tests

“Selenium" has been constantly increasing (i.e., it increases six folds, from 2011 to 2018), reaching around 0.6% in 2018 [5]. While the proportion might appear small, we wish to highlight that it still represents a sizable proportion given the wide scope of SO as it covers all aspects surrounding Software Development. For example, the proportion of questions on a popular IDE such as Eclipse is less than that of Selenium (0.4%). Similarly, JMeter, a popular load testing tool covers 0.1% of the total questions on SO. In this chapter, we explore Selenium-related questions to understand the needs and concerns of the developers and the support avenues for Selenium through online development forums such as StackOverflow.

In this chapter, we collect all the questions and answers related to Selenium that are posted on StackOverflow from August 11 2008 until November 11 2018. Since SO existed even before Selenium, we have a complete historical view of Selenium issues and challenges instead of a partial view. By examining these Selenium-related questions, we aim to understand the issues and challenges of using Selenium and the provided support by the Selenium community. In particular, our work proposes the following research questions:

RQ1: *What questions are being asked about Selenium?*

In this RQ, we extract and analyze the tags used in Selenium-related questions. We observe that tags related to programming languages (e.g., Python, Java, C# etc) are the most popular and fastest growing compared to other Selenium-related tags. Questions on browsers get more attention than the other Selenium-related questions; questions on testing frameworks have stabilized over the last few years

[5] https://insights.stackoverflow.com/trends?tags=selenium

while the number of new questions is increasing for most of the other Selenium-related questions.

RQ2: *Do users ask different type of Selenium-related questions for the different programming languages?*

We study the programming languages supported by Selenium in order to study the associations between programming language tags and the popular Selenium-related tags. For example, we identify that tags such as *web-scraping* and *python* have a strong association. On studying the user community in each of the programming language tags, we identify that *java* and *python* tags form the largest user base in Selenium. Further, the evolution of the number of new questions shows that only questions tagged with *python, java* and *C#* have an increasing trend.

RQ3: *Do users ask different type of Selenium-related questions for the different browsers?*

In this RQ, we analyze browser and driver tags associated with Selenium. We combine browser tags with their corresponding driver tags as they are used together in tests. For example, we combine *selenium-chromedriver* with *google-chrome*. *Google-chrome* related questions are the fastest-growing browser related questions used with Selenium. However, different browsers may be used for different purposes. For example, *PhantomJS* tag is widely used for web scraping while *Firefox* tag is frequently used with Selenium-IDE.

RQ4: *Which Selenium-related questions are difficult to answer?*

We identify the difficult aspects of Selenium using the time taken to receive an

accepted answer and the percentage of accepted answers as proxies for the difficulty of questions. We observe that less than half of Selenium-related questions get accepted answers. The answering speed varies across tags. Factors such as the age of the user on SO, size of the community and the reputation of the user are significant predictors of the answering speed, while the percentage of accepted answers is not impacted by those factors.

This is the first work to support Selenium users in making informed decisions when determining the appropriate browser and programming-language for their specific use-case i.e., test automation or web-scraping. Selenium developers can benefit from our analysis to understand the user community and the challenging aspects of Selenium. Our results also provide insights on the factors that influence the speed to get accepted answers and the ratio of accepted answers for Selenium-based questions.

Chapter organization. The remainder of the chapter is organized as follows. Section 3.1 describes the data source and our data extraction process. Section 3.2 presents our results for answering the research questions. Section 3.3 discusses the threats to the validity of our findings. Finally, Section 3.4 draws conclusions.

3.1 Case Study Setup

This section describes the data source and the data extraction process used in our case study.

3.1.1 Data Source

Stack Overflow is a popular online platform where users can exchange information related to computer programming and software engineering tasks. The website allows users to ask technical questions and answer existing questions, as well as to "vote" or "downvote" questions and answers based on the quality of the post. A score of each question or answer is calculated based on the sum of the votes and downvotes. Users can earn reputation points based on the upvotes that they receive for their questions and answers. Users are expected to add a maximum of 5 keywords (tags) to describe the topics of their question. For example, 'How to use Selenium Webdriver in Python?' would be tagged with *Selenium*, *Webdriver* and *Python*. In this example, the tags provide the main technologies/tools associated with Selenium.

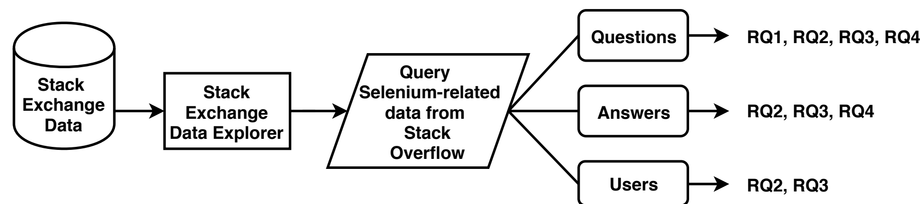

Figure 3.2: Data extraction process

3.1.2 Data Extraction

We query the data from Stack Overflow using the StackExchange Data Explorer (SEDE) [6]. SEDE is an open source tool for running arbitrary queries against public data from the Stack Exchange network (StackOverflow is a part of the StackExchange network). Figure 3.2 shows an overview of the data extraction process using SEDE. First, we query the *Posts* table to get all the questions with title containing words like 'selenium' or tags

[6]https://data.stackexchange.com/stackoverflow/queries

like ‘selenium’. We obtain a total of 73617 questions (56,906 questions with “selenium" tag, 16207 questions with tags containing Selenium in them and 504 questions with title of the post containing Selenium). As SEDE restricts the number of returned rows in each query to 50k, we repeat the same query twice to gather all the rows. We make use of the query parameters to manually enter the first and last IDs of posts to keep track of the returned posts in each query. Listing 1 shows the query used to extract Selenium-related questions from SEDE.

Listing 3.1: Query to extract Selenium-related ‘Questions’

```
/* Select unique questions from Posts table */
select distinct * from posts
/* Include questions that contains selenium-like tag
or title containing selenium-like word */
where (tags like '%selenium%'  OR
title like '%selenium%') AND
/* Use query parameters 'starting' and 'ending' to
enter the first and last post id */
id >= ##Starting:int## AND
id < ##Ending:int##
```

We query the *Posts* table again to obtain the answers on the extracted questions (as shown in Listing 2).

Listing 3.2: Query to extract Selenium-related ‘Answers’

```
/* Select unique posts */
select distinct * from posts as pp
```

```
/* Join Posts table to get only the answers (parentid
will be present only when the post is an answer) */
join
posts as p
on pp.id = p.parentid
/* Use query parameters ‘starting’ and ‘ending’
to enter the first and last post id */
where (pp.Tags like '%selenium%'
or pp.Title like '%selenium%')
/* Use a custom defined parameter to know the
starting point for the next query */
AND pp.id >= ##Starting:int##
AND pp.id < ##Ending:int##
```

In this chapter, we also study the user aspects of Selenium-related questions. Therefore, we executed another query that extracts data from both the *Posts* and *Users* tables. In this way, we extract all the necessary information related to Selenium posts that are needed to answer our research questions.

3.2 Case Study Results

In this section, we present the results of our research questions. For each research question, we present the motivation of the research question, the approach that we used to address the research question, and our experimental results.

RQ1: What questions are being asked about Selenium?

Motivation

Tags are keywords that the users pick to describe a question asked on StackOverflow. As tags are added by users themselves, they serve as an important piece of information to study the discussions on StackOverflow. Therefore, we use these tags to understand the techniques/tools that are associated with Selenium and the relationship between them. Our findings could help the developers of Selenium and users understand how Selenium is used in practice.

Approach

Identifying popular tags. We study the distributions of the tags that co-occur with the *Selenium* tag. We define a concurrent tag that co-occurs with the *Selenium* tag as a Selenium-related tag (e.g., Python, Firefox, web-scraping). We identify the 'popular tags' based on the number of times a tag has been used in Selenium-related questions i.e., we pick the popular tags based on a threshold - among the 73k studied questions, we pick the tags that appear in at least 730 questions i.e., comprising of 1% of the studied questions. We obtain the top 38 tags based on this approach. As some of the tags are duplicates, we manually merge them together. For example, we merged *selenium-webdriver* and *webdriver*, *automated-tests* and *automation.* We ended up with 35 popular tags which are used for the analysis throughout this chapter.

Grouping popular tags. We identify groups within the 35 popular Selenium-related tags:

- Programming languages *(java, Python, c#, JavaScript, ruby, php, node.js, angularjs)*
- Browsers and drivers *(chromedriver, Firefox, google-chrome, phantomjs, internet-explorer)*
- Selenium components *(webdriver, selenium-ide, selenium-grid, selenium-rc)*
- Testing frameworks *(testng, protractor, appium, cucumber, capybara, robot framework, junit)*
- General processes and tools *(automation, testing, web-scraping, android, maven, eclipse, jenkins)*
- Web elements and locators *(html, css, xpath, css-selectors)*

Identifying commonly occurring tag pairs. In order to study the strength of associations between tag pairs and the communities in Selenium, we first filter the data to account for associations that constitute 0.1% of the studied questions i.e., tag pairs that occur in at least 73 questions.

Calculating attention metric. Among the final set of popular tags, we identify the tags that receive a large amount of attention. We use the number of views as a proxy metric to capture the received attention in Selenium-related tags. As older questions tend to have more views, we calculate the median of normalized view count for questions in each tag. We choose median over other central tendency measures as the view count data is generally skewed i.e., contains outliers. The normalized view count is calculated using:

$$\textit{Normalized view count} = \textit{Number of views} / \textit{Age of the question in days}$$

Calculating the evolution of added questions. In order to study the trend in the added Selenium-related questions over time, we group the created questions every month for each of the tag groups that we identified.

Results

Programming languages and browsers are among the most popular tags associated with Selenium. Figure 3.3 shows the distribution of the number of questions for the popular tags in Selenium. From the top 5 tags, we see that 4 out of the 5 tags belong to the 'programming languages' group. We observe that Selenium questions are related to specific programming languages that Selenium supports. *Java, Python, C#* and *JavaScript* are the most popular programming language tags. Among tags related to browsers and drivers, *selenium-chromedriver, Firefox* and *google-chrome* are the most popular ones. Other popular Selenium-related tags are *webdriver, xpath, automation, testng* and *html*. Our analysis shows that *webdriver* is the most-asked Selenium component; *xpath* is the most-asked locator strategy. Leotta et al. (2013) show that the time spent for repairing XPath-based test cases is much more than the time spent for repairing the ID-based ones. XPath requiring more maintenance effort as shown by Leotta et al. (2013) could drive developers to ask more questions on it.

Selenium-related tags co-occur with clear patterns, as we observe closely-related groups within Selenium-related tags. The network graph in Figure 3.4 shows the association between the commonly occurring tag pairs. We use the Force Atlas layout in Gephi [7] to get the nodes with high input links towards the centre of the graph and the ones with high output links towards the periphery. The size of the node indicates the

[7] https://gephi.org/

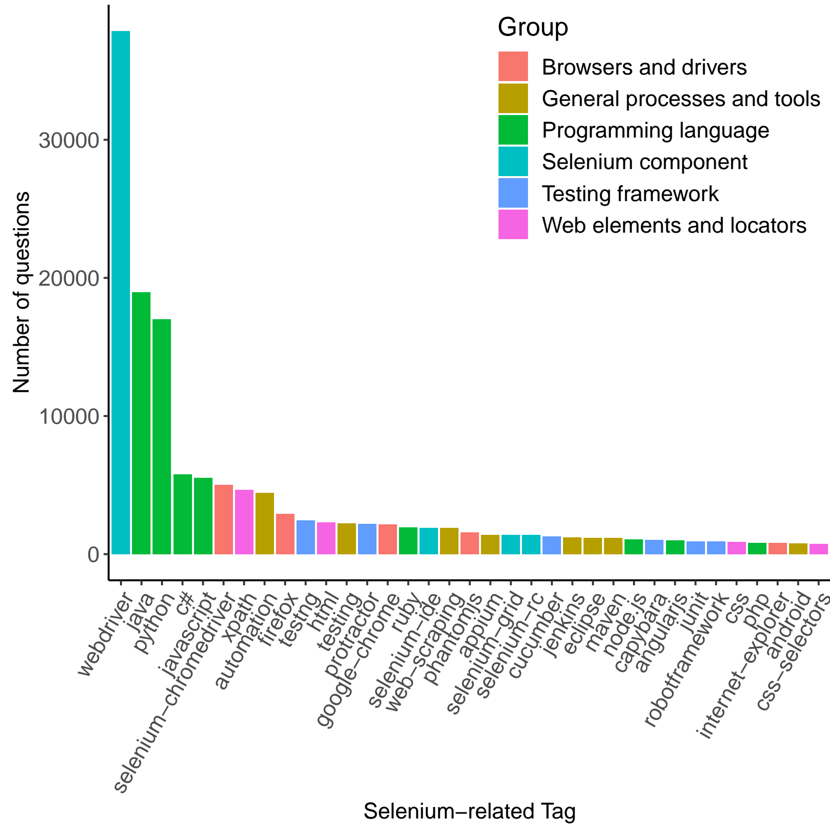

Figure 3.3: Most popular Selenium-related tags

strength of the incoming links (or the indegree). The nodes with same colors constitute a community. We use the modularity class function in Gephi for community detection (Blondel et al. (2008)). We observe that tools and frameworks along with the programming language that the tools and frameworks are written in constitutes a community. For example, *ruby, capybara and cucumber* and *JavaScript, node.js, angular.js and protractor* constitute a community. Similarly, java-related libraries and tools form a community. The community with *Python, web-scraping, selenium-chromedriver, google-chrome, robot framework* tags shows interesting associations such as the association

between *web-scraping* and *Python* tags.

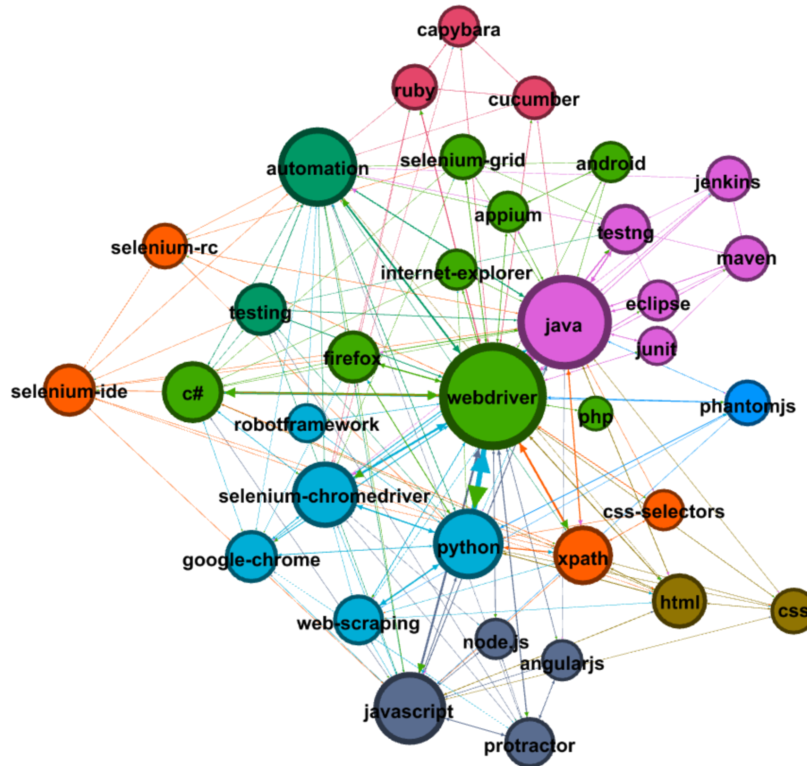

Figure 3.4: Relationship between Selenium-related tags

Questions related to browsers get the most attention while questions related to web-elements and locators receive the least attention. Figure 3.5 shows the median normalized view count for each of the Selenium-related tags. Among the 6 groups identified, the median normalized view count is the highest for 'browsers and drivers' group (0.55), followed by 'testing frameworks' (0.51). We observe that *google-chrome, robot-framework* and *selenium-chromedriver* are the most viewed tags among the Selenium-related tags. Vila et al. (2017) discuss the opportunities and threats in developing a new

framework based on Selenium WebDriver. They argue that the creation of automation testing framework with Selenium WebDriver could reduce time for development among other advantages. Reduced development time could be the reason why users adopt testing frameworks and therefore testing frameworks related questions receive a large amount of attention on StackOverflow.

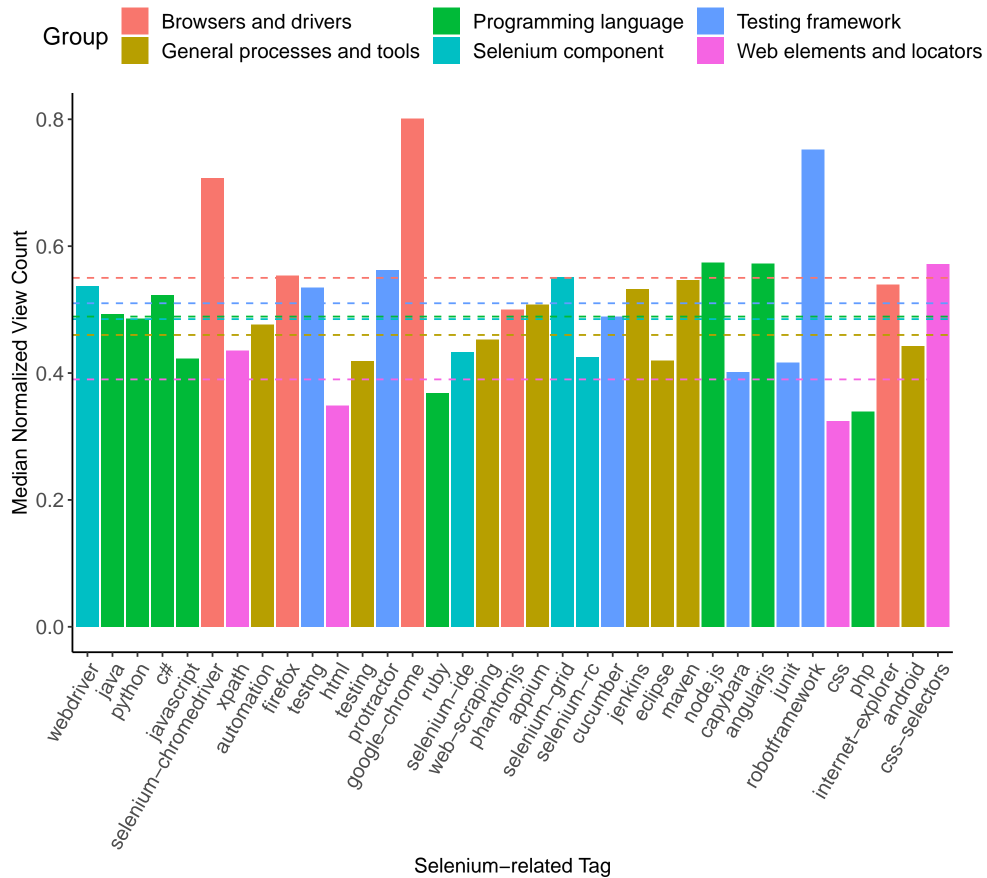

Figure 3.5: Attention received by Selenium-related tags. The dotted lines indicate the median attention for each group.

The questions with each category of tags have been increasing over time; among

them, the questions with programming languages-related tags have been increasing the fastest. We study the evolution of the popular tags categorized in the 6 groups obtained from RQ1. Figure 3.6 captures the overall trend of each of the groups. We observe that all groups have an increasing trend in the number of new questions from 2010 to 2018. Although programming language related tags receive less attention compared to the tags in other groups, Selenium questions that are associated with programming languages are increasing the fastest. Selenium questions on browsers, selenium components and general processes and tools are steadily increasing, whereas Selenium questions on testing frameworks and web elements and locators have been stable in the past years.

Programming languages are the most popular and fastest-growing tags; browser-related tags get the most attention. The number of new questions per month is increasing for most of the tags except for tags related to 'testing frameworks' which have stabilized over the last few years.

RQ2: Do users ask different type of Selenium-related questions for the different programming languages?

Motivation

From the result of RQ1, we observe that askers of Selenium-related questions ask programming-language specific questions. Although all the programming languages share the same functionalities, we observe that different programming languages are associated with different tools/techniques. Therefore, we study the differences in the questions that are associated with different programming languages in terms of the associated tags

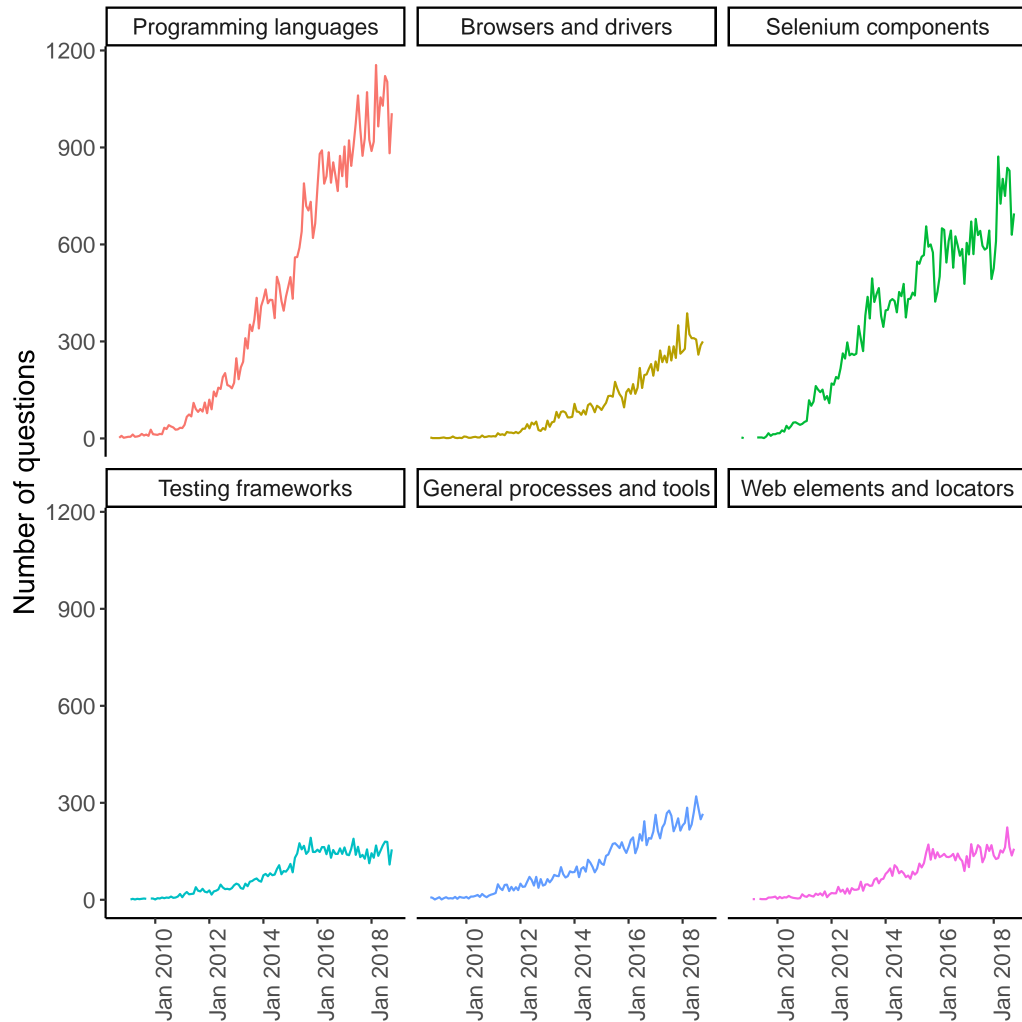

Figure 3.6: Evolution of questions added in each group

and the characteristics of the communities around these tags. We also study the evolution of the programming languages used for Selenium scripts.

Approach

Calculating the tag association metric. In order to calculate the association between programming language related tags and other tags (i.e., the tag association), we obtain

the normalized values of the number of times each of the popular tags appears with each of the programming language related tags. For example, if the total number of questions with the *web-scraping* tag is 1888 and 1673 of which occur with the *python* tag, then the tag association would be 0.89 (1673/1888). A tag association value close to 1 would indicate a high degree of association.

Identifying non-casual users. To study the user communities for different programming languages related tags, we use the Users dataset to calculate the distinct users, the median reputation and the median number of asked or answered questions by each of these users. We also define 'non-casual askers/answerers' as those users who asked or answered more than 1 question on SO. We also define 'casual askers/answerers' as those users who asked or answered 1 question on SO. We calculate the percentage of questions that are asked by non-casual askers/answerers to determine the amount of contribution by the non-casual users of Selenium on SO.

Results

Java and Python tags are associated with most of the popular Selenium-related tags. Figure 3.7 shows the tag association for the tags that correspond to all the programming languages that Selenium supports. We observe that most of the top popular tags are asked along with the *Java* tag, suggesting that Java is the most commonly used tag in Selenium. Apart from Java tag, *Python* tag is also widely used with the popular Selenium-related tags.

It is interesting to note that the *web-scraping* tag in Selenium mostly co-occurs with the Python tag than any other programming language related tag. This could be due

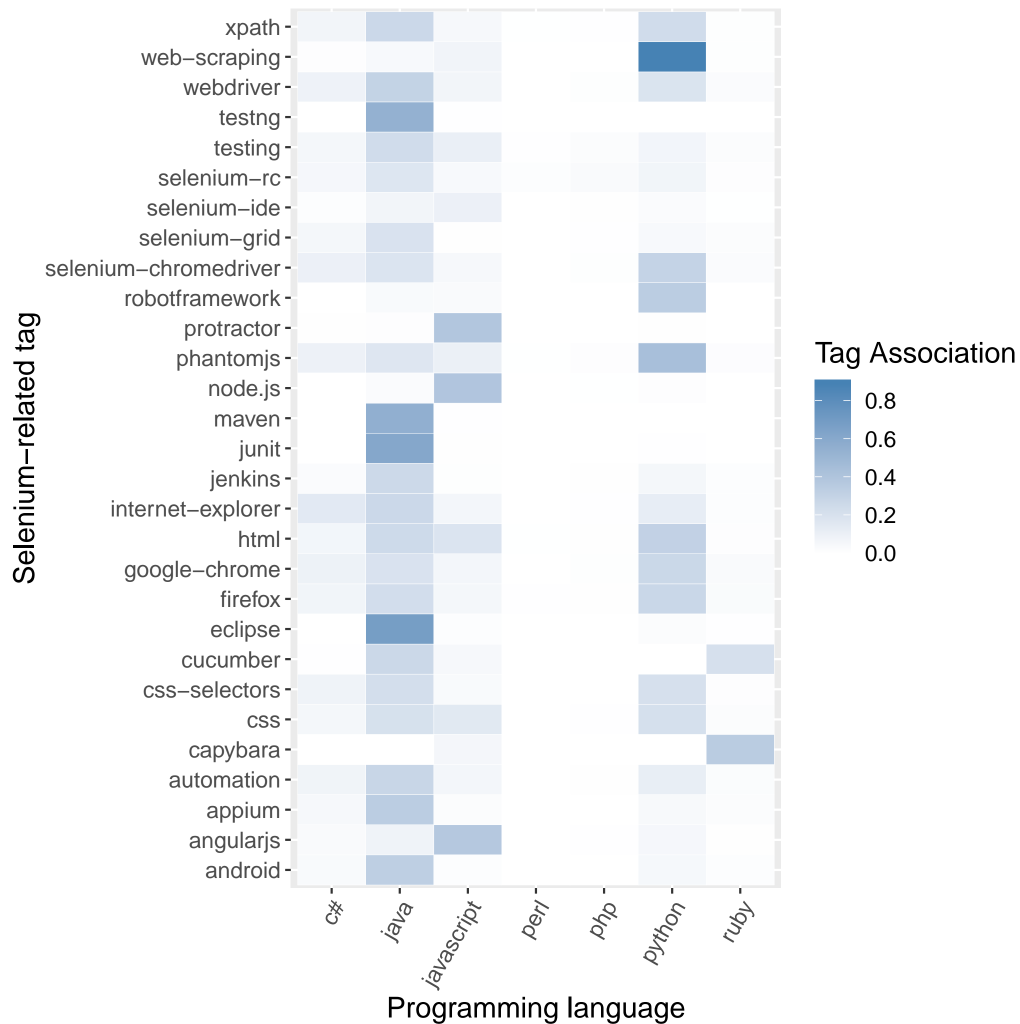

Figure 3.7: Tag association between programming languages

to the availability of several python scraping packages such as beautiful-soup [8] and scrapy [9]. These python packages can be easily integrated with Selenium scripts written in Python to navigate to several URLs and scrape data. *Appium*, a test automation tool for mobile apps is used with *Java* more than *Python*. Therefore, users who would want to use Appium for future projects can use Java for testing mobile apps as it has

[8]https://www.crummy.com/software/BeautifulSoup/
[9]https://scrapy.org/

better support on SO. We observe that JavaScript-based frameworks and libraries relate closely to *JavaScript*, e.g., *node.js*, *angularjs* and *protractor* tags. Similarly, users of *ruby* tag discuss about *capybara* and *cucumber* much more than the users of other programming language related tags. These results are consistent with the findings from the network graph shown in RQ1.

We observe that users of Java & Python tags use Selenium directly and users of other programming language related tags such as Ruby & JavaScript use several other tools built on top of Selenium. For example, user of ruby tag may most likely use *capybara* and *cucumber* while users of JavaScript tag may use *protractor*.

Users of PHP tag discuss tools and frameworks built to test applications written in PHP e.g., *behat, codeception*. Although tags that co-occur with the Perl tag are popular, the number of questions that contain both Perl tag and the corresponding Selenium-related tag is very few (around 10). Therefore, the tag association is not significant. This could be because Selenium does not provide direct support for these languages (PHP, Perl and Javascript).

Selenium users of Java and Python form a more mature community than other programming languages, in terms of the percentage of non-casual users, the percentage of questions asked/answered by non-casual users. Users of Java and Python tags form the largest community of users among Selenium-related tags, followed by the users of Javascript tag, which is consistent from the results obtained before. The non-casual askers of Python and Java contribute to a higher percentage of the Python and Java questions, compared to other programming languages. It is interesting to note that although users of Java tag form a mature community, the median reputation of answerers is the lowest among users of other programming language related tags.

	Programming language related tags							Overall
User stats	C#	Java	Python	Ruby	Javascript	Perl	PHP	Selenium
Distinct askers	3,411	10,729	9,279	1,210	4,334	117	669	39,782
Median questions asked	1	1	1	1	1	1	1	1
Percentage of non-casual askers	26.5	28.2	31.8	23	14.2	22	12.9	29.8
Percentage of questions asked by non-casual askers	55.9	59	61.8	51.2	31.74	48.8	28	61.6
Median reputation of askers	32	20	33	81.4	66.5	63	156	26
Distinct answerers	3,834	10,632	7,738	1,425	4,019	164	697	32,489
Median questions answered	1	1	1	1	1	1	1	1
Percentage of non-casual answerers	22.3	27	30.7	18.8	17.4	20	15.8	27.4
Percentage of questions answered by non-casual answerers	61.4	73.2	74.4	54.3	52.8	44.5	36.9	76.5
Median reputation of answerers	363	174	324	650	501	1,892	900	187

Table 3.1: Stats for users in each of the programming language related tags in Selenium

Slag et al. (2015) report that about half of the users ask or answer only once (casual users). Among Selenium users, we observe that users with 1 contribution is more than half of the overall community (70.2%). We identify that the small group of non-casual users contribute to majority of the posts on Selenium i.e., 29.8% of non-casual askers contribute to 61.6% of the questions and 27.4% of non-casual answerers contribute to 76.5% of the questions. Similarly, among programming languages, casual askers contribute to more than half of the Javascript, Perl, and PHP questions while casual answerers contribute to more than half of the Perl and PHP questions.

Answerers have much higher reputation than askers, especially for Perl questions; answerers of Perl, PHP, and Ruby have the highest reputation. The average reputation of general users in SO (as of Dec 16 2018) is 108. From Table 3.1, we observe that the median reputation of askers is less than the average reputation, while the median reputation of answerers is well above the average reputation of a user.

Python tag is the fastest growing programming language related tag of Selenium, while Java tag has become stable in recent years. In RQ1, we saw that 'Programming

languages' group has a steep increase in the number of new questions compared to the other groups. Therefore, in this RQ, we study the trend for each programming language tag that Selenium supports. Python, and C# questions are increasing while questions on other programming languages are stable. Python questions is the fastest-growing programming language related questions in Selenium. Java questions has been stable in the past few years.

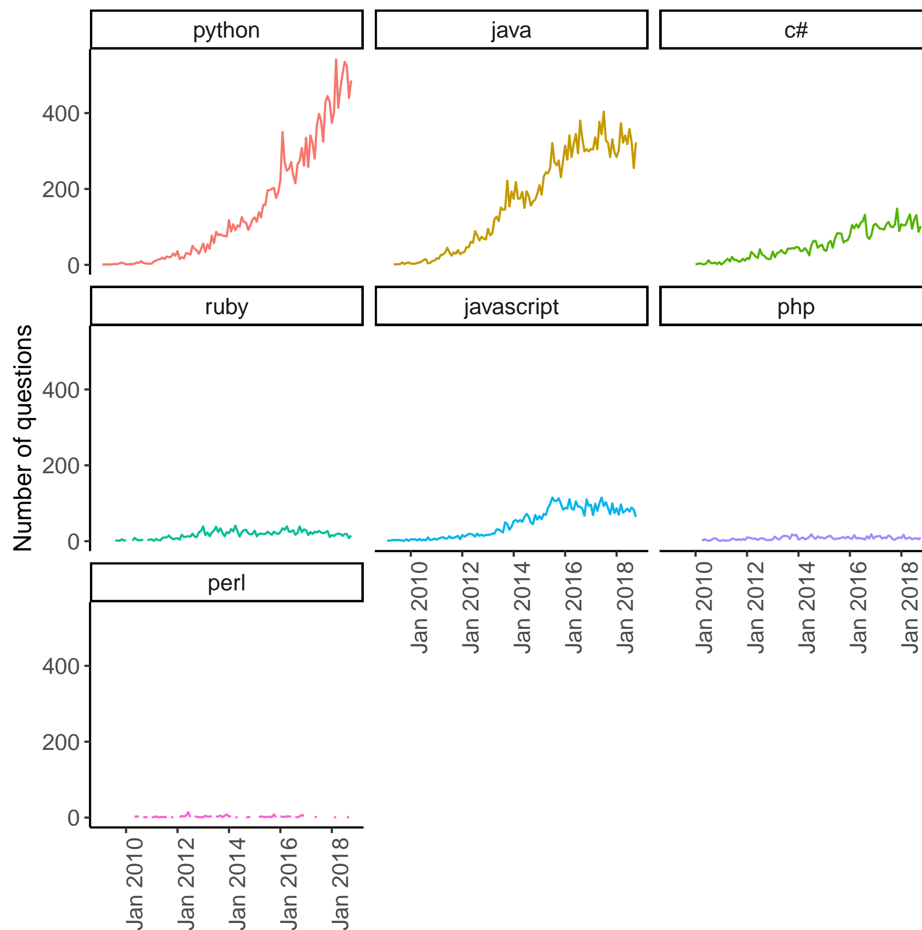

Figure 3.8: Evolution of the asked questions in each of the programming language related tags

Users of Java and Python tags of Selenium form more mature communities than the users of other programming language related tags. Questions about Python have become the most frequent and the fastest growing programming language related questions in Selenium, while questions on Java have become stable in the recent years.

RQ3: Do users ask different type of Selenium-related questions for the different browsers?

Motivation

In this RQ, we want to study the differences among the browser and driver tags that Selenium supports. Studying the commonalities and differences among browser tags will let a new user be aware of the shortcomings and possible strengths of certain browsers and drivers. Similarly, studying the characteristics of user communities for each of the browser and driver tags will be useful for Selenium-users to identify browser tags with good support on SO.

Approach

Merging browser and driver tags. As drivers are always used with their corresponding browsers, we combine *selenium-chromedriver* questions with *google-chrome*; *geckodriver* questions with *Firefox* and *selenium-iedriver* questions with *internet-explorer* in order to study the associations among tags, the user communities and the evolution of asked questions.

Calculating the tag association metric. We calculate the association between browsers

and other tags (i.e., the tag association) using the same approach followed for programming language related tags as shown in RQ2.

Results

Google Chrome tag appears with most of the popular Selenium-related tags; however, different browsers are used for different purposes. For example, we observe strong associations between PhantomJS and Web-scraping tags From Figure 3.9, we observe that, *google-chrome* tag co-occurs with most of the popular tags in Selenium compared to other browser tags, which is followed by the *Firefox* tag. We observe that *internet-explorer* tag is used less frequently with any of the popular tags. Another interesting observation is that *PhantomJS* tag occurs with *web-scraping* tag more than any of the other popular tags. PhantomJS is a headless browser used for automating web page interactions. Therefore, the strong association suggests that users prefer using headless browser instances to perform web-scraping as the GUI is not necessary to scrape data. We also observe that *selenium-rc* tag and *selenium-ide* tag occurs with *Firefox* tag more than the other browser related tags. The IDE was initially a Firefox browser plugin. However, now the Selenium IDE is also available for Chrome browsers as the Katalon Recorder [10].

We also observe that the *eclipse* tag co-occurs with the *Firefox* tag more than the tags that correspond with any other browser. By manually analyzing the questions with *Firefox* and *eclipse* tags, we observe that some users have trouble running selenium-rc tests with eclipse configurations especially when the scripts works from the console or

[10]https://chrome.google.com/webstore/detail/katalon-recorder/ljdobmomdgdljniojadhoplhkpialdid

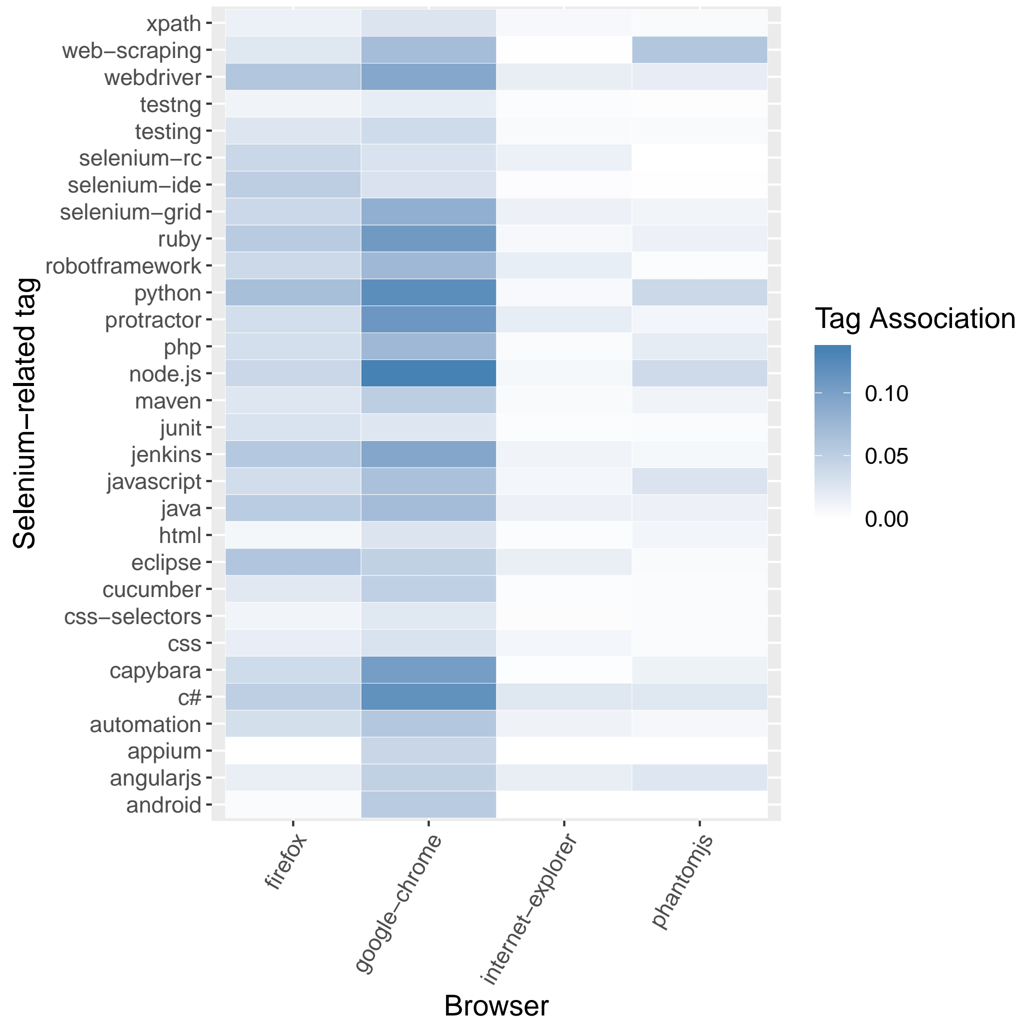

Figure 3.9: Tag association between browsers (combined with drivers)

command line, others report compatibility issues between selenium server and specific Firefox browser versions.

Users of Chrome and Firefox tags form mature communities; only a small percentage of PhantomJS tag users are non-casual - however, they have the highest reputation among the users of other browser related tags Table 3.2 summarizes the characterisitcs of users using different browser and driver tags. Users of *Google-chrome*

tag form the largest community among users of other browser tags. As per the stats from *google-chrome* and *Firefox* questions, we observe that about 27% of the users answer close to 60% of the questions; upto 40% of the questions are asked by non-casual askers. However, 80% of the users are casual askers. Internet-explorer tag users form the smallest community in SO, among users of other browser tags. However, the median reputation of their answerers is the second highest among the users of other browser tags. The non-casual askers and answerers of *PhantomJS* questions comprise only 11% of the total PhantomJS users. However, it is interesting to note that 11% of non-casual users answer upto 38% of the questions. The median reputation of askers and answerers is higher than askers and answerers of other browser tags.

	Browsers (with drivers)				Overall
User stats	google-chrome	Firefox	internet-explorer	phantomjs	Selenium
Distinct askers	5,291	3,179	826	1,310	39,782
Median questions asked	1	1	1	1	1
Percentage of non-casual askers	22.61	21.02	11.64	11.76	29.8
Percentage of questions asked by non-casual askers	42.39	39.78	25.08	25.39	61.6
Median reputation of askers	43	62.5	60	100	26
Distinct answerers	4,640	3,170	905	1,078	32,489
Median questions answered	1	1	1	1	1
Percentage of non-casual answerers	27.51	26.22	18.14	11.51	27.4
Percentage of questions answered by non-casual answerers	62.49	58.49	42.72	37.96	76.5
Median reputation of answerers	180	334.5	377	651	187

Table 3.2: Stats for the user communities of each of the browser and driver tags in Selenium

Chrome tag is the fastest growing tag among browser tags and it becomes the most frequently used browser tag used with Selenium, while the use of Firefox and PhantomJS browser tags is declining over the recent years. Figure 3.10 shows that only *google-chrome* tag has an increasing trend in the number of new questions. *PhantomJS* has a declining trend over the last few years. We also observe that *internet-explorer* tag

has a constant and negligible number of added questions every month. With *Firefox*, we observe that the number of new questions increased from 2016, but became constant after that.

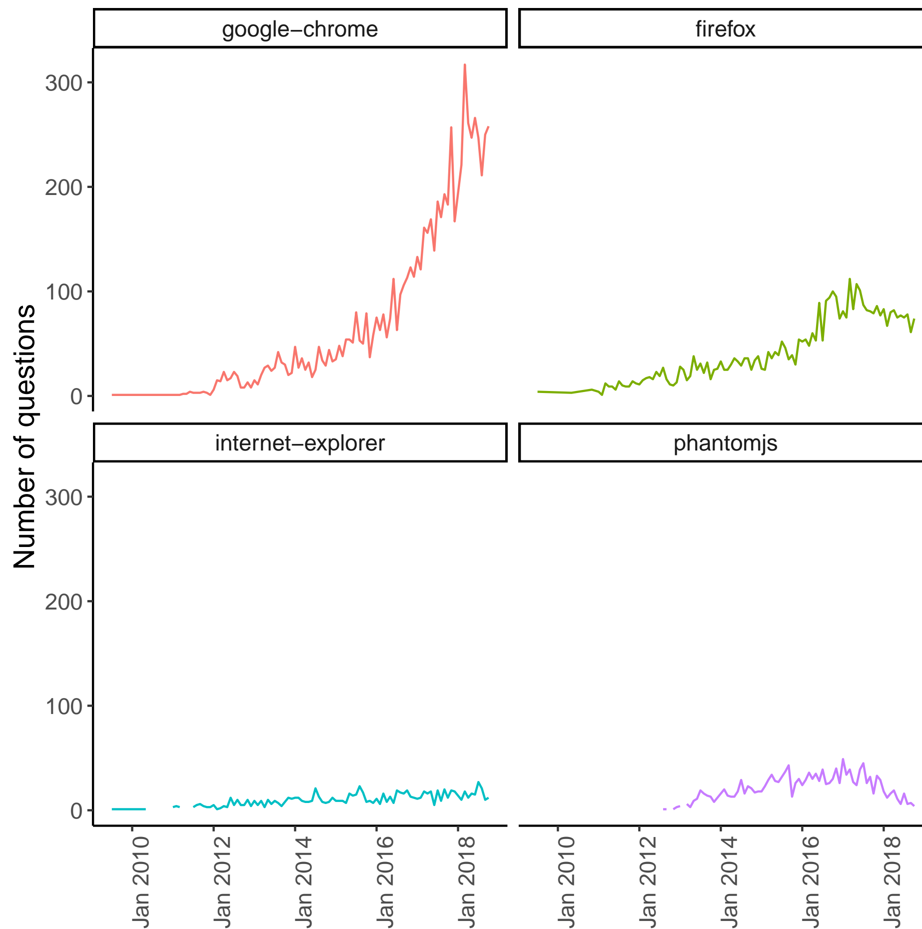

Figure 3.10: Evolution of questions added in each browser (with the corresponding drivers)

Chrome tag is the fastest growing tag among browser tags in Selenium. However, different browsers may be used for different purposes. For example, PhantomJS tag is widely used for web scraping while Firefox tag is frequently used with Selenium-IDE.

RQ4: Which Selenium-related questions are difficult?

Motivation

In this RQ, we study the difficult aspects in Selenium. We measure the difficulty based on the percentage of accepted answers and the time taken to receive an accepted answer for each tag. Understanding the difficult questions could benefit the developers of Selenium in offering better help on these difficult areas. It could also benefit Selenium users to understand how well they can get support from the community.

Approach

Metrics to calculate difficulty levels. We use two metrics to measure the difficulty level: (1) Percentage of accepted answers for the tag (2) Time to get accepted answers. We calculate the time to get accepted answers by subtracting the creation date of the accepted answer and the creation date of the question in each of the data subsets i.e., dataset specific to each tag.

Model Construction. We build linear regression models to study the relationship between the community of non-casual answerers and the speed and likelihood of getting an accepted answer. Non-casual answerers are users who answer more than 1 question on the platform. We consider the user communities in the top 35 popular tags to carry out this study.

$$\hat{Y}_i = \hat{\beta}_0 + \hat{\beta}_1 X_i + \hat{\epsilon}_i \quad (3.1)$$

The dependent variables for the models are (1) Median time to get accepted answers and (2) Percentage of accepted answers. We include features (i.e., independent variables) such as size of non-casual answerers (i.e., the number of non-casual answerers in each tag), the median reputation of non-casual answerers and the median age of the non-casual answerers.

Results

Less than half of the Selenium-related questions get accepted answers. In particular, questions related to browsers, and general processes and tools are less likely to get accepted answers. 53% of questions have accepted answers on StackOverflow (as of Dec 18 2018 - total number of questions = 16,928,570, total number of questions with accepted answers = 8,955,202). From Figure 3.11, we see that most of the tags have about 40% accepted answers. Therefore, Selenium questions have less accepted answers in general. We observe that questions that are tagged with *css-selectors*, *web-scraping* and *xpath* have the highest percentage of accepted answers. While questions about 'web elements and locators' and 'programming languages' have the most number of accepted answers, the majority of the questions on 'browsers and drivers' and 'general processes and tools' have the least number of accepted answers. The reason for 'web elements and locators' group to have the highest percentage of accepted answers could be because (1) the concept of locators is the most fundamental aspect in Selenium and the Selenium documentation explains the various locator strategies very

clearly (2) web elements such as HTML and CSS are generally easy and most developers know the basics of such concepts. Whereas general processes and tools such as *appium, android, jenkins* are not specific to Selenium and therefore lack accepted answers. Similarly, although Selenium supports several browsers and drivers, the documentation lacks details specific to each browser or driver.

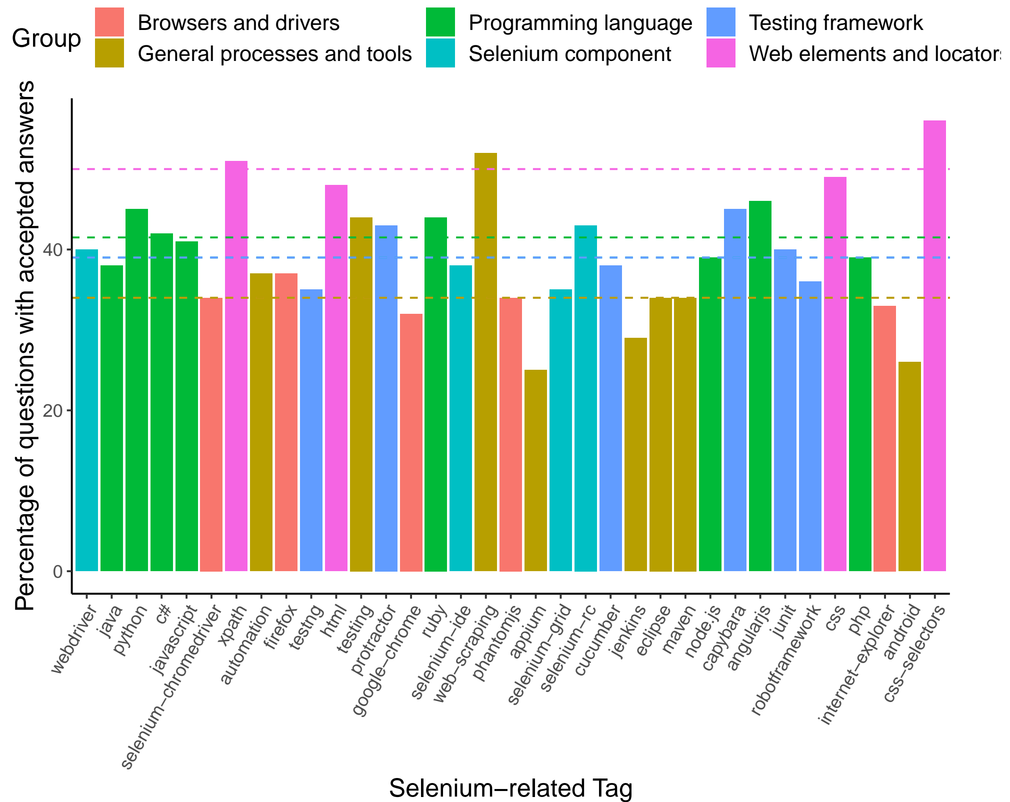

Figure 3.11: Percentage of questions with accepted answers for each of the Selenium-related tag. The dotted lines indicate the median percentage of accepted answers for each group.

Questions related to Selenium components, browsers, and general processes and tools take the longest time to get accepted answers; while questions related to web

elements and locators and programming languages get the accepted answers much faster. We observed that Selenium-related tags did not show a large amount of variation in the percentage of accepted answers i.e., the values are close to each other. However, the time taken to receive an accepted answer shows lots of variations among the Selenium-related tags. It is interesting to note that questions on Selenium components such as *selenium-grid* and *selenium-rc* take almost a day to receive accepted answers. While *selenium-rc* is deprecated now and therefore lacks support, it is also interesting to see that questions on other active components such as *selenium-grid* and *selenium-ide* also take a long time to receive accepted answers. The most difficult aspects in Selenium seem to be Selenium components. Other tags that take the longest time to get accepted answers include *android, internet-explorer* and *appium.*

Questions on "web elements and locators" seem to get accepted answers very quickly (around 1 hour). Question on 'web-scraping' also get accepted answers fairly quickly. Comparing these results with prior observations, we understand that questions on *css-selectors, web-scraping* and *xpath* are most likely to get accepted answers quickly. Programming languages such as Python, Java, C# and JavaScript receives answers within 4.5 hours. However, when compared to the median time to get an accepted answer on SO (i.e., 16 minutes Bhat et al. (2014), Selenium questions take much longer to receive accepted answers.

The time taken to get accepted answers is statistically significantly impacted by the number, the median reputation, and the experience level of non-casual answers of a Selenium-related tag. With median time to get accepted answers as the response variable, the results from linear model shows that the size of non-casual answerers ($p < .01$),

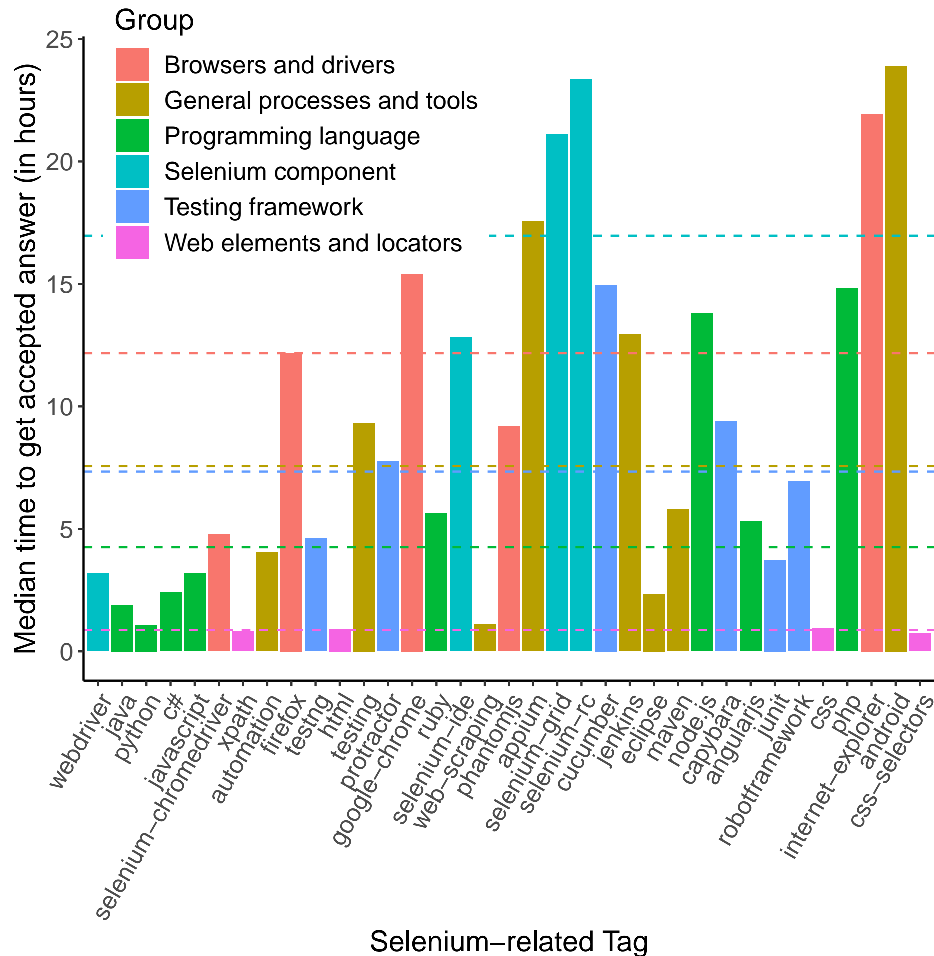

Figure 3.12: Answering speed for the top 35 Selenium-related tags. The dotted lines indicate the median time to get accepted answer (in hours) for each group.

median reputation of non-casual answerers (p <.02) and median age of non-casual answerers (p <.0005) are significant predictors. The overall model fit was Adjusted $R^2 = 0.34$. We observe that age of the non-casual answerers had the most significant impact on the answering speed in Selenium-based questions.

Our model shows that the size of the non-casual answerer community and the median reputation of the non-casual answerers have a positive correlation with the answering speed. Therefore, this result implies that a community with a large size and a high reputation (e.g., Java/Python) is more likely to get faster answers than a community with a small size and a low reputation (e..g, PHP). Further, the age of the non-casual answerers on the StackOverflow platform has a negative correlation with the answering speed. This shows that new users on SO are more likely to answer questions faster.

However, the results with percentage of accepted answers as the response variable has a contrasting result. The size of non-casual answerers ($p < .25$), median reputation of non-casual answerers ($p < .12$) and median age of non-casual answerers ($p < .64$) are not significant predictors.

Less than half of Selenium-related questions get accepted answers. The answering speed varies across different tags. The time taken to get accepted answers is statistically significantly impacted by the number, the median reputation, and the experience level of non-casual answerers of the Selenium-related tags.

3.3 Threats to Validity

External Validity. We only consider posts extracted from StackOverflow for this study. Although, there exists other QA sites and forums, the total number of questions posted is not high. For example, SQA forum has a total of 2,478 questions on Selenium [11]; Software Quality Assurance and Testing site [12] within StackExchange has 8,522 questions. In contrast, Stack Overflow has over 73,000 questions about Selenium. Some people may not use an online forum to ask questions. For example, they may use a mailing list.

[11] http://www.sqaforums.com/forums/selenium/
[12] https://sqa.stackexchange.com/

However, StackOverflow is the most popular QA site among software developers and therefore, we study only Selenium-related questions in this chapter. Moreover, since SO existed even before Selenium, we have a complete historical view of Selenium issues or challenges instead of a partial view. Further, our approach to analyze Selenium questions on StackOverflow is generalizable i.e., our approach can be used by others to study the Selenium questions on other forums.

Internal Validity. We extract posts based on tags like Selenium and title containing a word like Selenium. Although this approach captures most of the Selenium-related posts, there is a chance that we would have missed posts that discusses Selenium in the body of the post, without mentioning it in the title or including the tag. However, in such cases, the post would not have a strong association with Selenium. We assume that questions with the “selenium" tag or containing “selenium" in the title are more relevant to Selenium.

The extracted data may contain duplicate tags. Most of the synonymous tags are already identified by users in SO [13]. However, some of the tags are yet to be reported. For example, *selenium-firefoxdriver* and *geckodriver* refer to the same driver. We alleviate this threat by using our domain expertise to ensure that similar tags are merged before beginning our analysis.

In this chapter, we extract and analyze Selenium-related tags from StackOverflow. Most studies that use StackOverflow data use Latent Dirichlet Allocation (LDA), a popular topic modelling technique for finding discussion topics in natural language text. Due to this reason, we also run LDA on the data. We observed that the topics obtained from LDA are close to the tags extracted. Topics such *java code, automation, python*

[13]https://stackoverflow.com/tags/synonyms

code, chromedriver, xpath, web scraping, webdriver overlap with our top 35 popular tags. Therefore, we use only tags to perform our analysis. Further, LDA modelling suffers from other drawbacks such as determining an appropriate number of topics before running the model and deriving meanings from co-occurring words to assign labels to topics. Future practitioners can use only tags in such cases, in order to reduce the complexity of the analysis.

Among the six groups that were identified, we chose to study only 'programming languages' and 'browsers and drivers' as users have the option to pick among the several programming languages and browsers that Selenium supports. For example, users have the choice to pick between *Firefox* and *Chrome* or between *Python* and *Java*. Due to this reason, we did not study Selenium components as each component is designed for a specific task. For example, *selenium-ide* is used to record and replay tests whereas *selenium-grid* is used to run tests in different browsers and operating systems in parallel.

Construct Validity. A big threat to our study is that we use Stack Overflow questions to capture Selenium aspects such as programming languages and browsers. However, the questions on Stack Overflow may not represent the actual usages of Selenium.

We measure the difficult aspects in Selenium based on the time taken to receive an answer. We assume that difficult questions may take a longer time to get accepted answers. Moreover, Rosen and Shihab (2016) follow the same approach to measure the difficult aspects in mobile technology.

In order to identify the factors that impact the answering speed and the percentage of accepted answers, we use 3 characteristics of non-casual answerers such as the size, reputation and age on the platform. The results discussed in this chapter show only the

correlation between the characteristics of the non-casual answerers and the answering speed i.e., there is no causal relationship in either direction.

3.4 Conclusion

In this chapter, we perform an empirical study of Selenium-related questions on Stack-Overflow. We first identify the most popular Selenium-related tags. Next, we study the commonalities and differences in various programming language related tags and browser related tags that Selenium supports. Finally, we study the difficult aspects of Selenium and the factors that impact the answering speed and the likelihood of getting accepted answers. Our results show that questions on programming languages is the most popular and fastest growing among other Selenium-related questions while browser-related questions get the most attention. We also find that Chrome tag is the most frequently used browser tag among other browser tags related to Selenium. However, different browsers may be used for different purposes. For example, PhantomJS is widely used for web scraping while Firefox is frequently used with Selenium-IDE. Users of both Java and Python tags of Selenium form large communities than users of other programming language related tags. However, Python tag is the fastest growing among other programming language related tags, while Java tag has become stable in the recent years. Further, factors such as the age of the user on the SO platform, size of the community and the reputation of the user are significant predictors of the answering speed. We believe that our insights will help Selenium developers improve the support for the difficult aspects in Selenium and understand how Selenium is used in practice.

CHAPTER 4

Reducing the Performance Overhead of Selenium-based Load Tests

Load testing is usually performed to determine a system's behavior under anticipated peak load conditions (Jiang and Hassan (2015)). Load tests are crucial for large scale systems to avoid crashes. Several crashes and application failures have been reported in the past (Census (2016); NY Times (2013); CBC (2017)). These crashes were caused due to the inability of the systems to process large amounts of concurrent requests. These kinds of catastrophic events can be avoided if the software systems are tested with realistic or field-like workloads.

Practitioners use tools such as JMeter to create a production-like workload (with

thousands of users) on the System Under Test (SUT), and measure its performance (response time, resource utilization etc) under this load. However, JMeter's major disadvantage is that it is not trivial to capture complex application behaviour. For example, it is not possible to check the rendering of a webpage.

One way to overcome some of these disadvantages is by using browser-based tests. Selenium is an example of browser-based test tool. Selenium is widely used to automate functional testing of web applications that run in web browsers (Dowling and McGrath, 2015; Debroy et al., 2018; Gojare et al., 2015). Quality Assurance (QA) engineers of web applications usually write Selenium scripts to capture the web content that is rendered in a browser and simulate users' interactions with the web content. For example, Selenium can be used to determine whether a button element of a web application is loaded and displayed properly, and simulate users' actions to click the button. Selenium supports the testing of web applications in different browsers, such as Firefox and Google Chrome (Altaf et al. (2015)).

Apart from functional testing, Selenium can also be used to test the performance of web applications (a.k.a., performance testing or load testing). Selenium-based load testing has several advantages over tools using protocol-level requests. First, Selenium drives browsers to execute the Javascript in the loaded HTML pages, while protocol-level requests cannot. Second, Selenium-based load testing can capture the rendering of a web application in a browser, which is particularly important when testing the web applications that dynamically load web content (e.g., using AJAX techniques); protocol-level requests can only view the response as HTML but cannot render the HTML pages. Third, writing Selenium test scripts is much easier than hard coding protocol-level HTTP requests (e.g., Selenium tests leverage the rich functionality of

browsers to handle complex internet protocols such as the SSL certificates); Selenium-based load testing can reuse the scripts of Selenium-based functional tests (whereas protocol-level load testing needs separate scripts). Finally, Selenium allows the capturing of real user scenarios in load tests.

However, Selenium has its own disadvantages, most notably its resource overhead - as Selenium needs to launch a browser for each user instance (e.g., a client) of a load test, it is usually very resource consuming to run a load test that simulates thousands of users since one would need to run thousands of browser instances.

In this work, we take the first important step towards improving the efficiency of load testing using Selenium. We share our experience of using Selenium for load testing, include the performance issues that we encountered, our investigations of these performance issues, and how we addressed them. In order to leverage the advantages of Selenium-based load testing while reducing the high resource overhead, we propose an approach to improve the efficiency of Selenium-based load testing. Our approach shares browser resources among the instances of a load test, thereby avoiding the cost of launching a browser for each user instance. Therefore, our approach significantly improves the testing efficiency of Selenium-based load tests.

We believe that our approach and our shared experience can help software practitioners improve the efficiency of Selenium-based load testing. The main contributions of this work include:

- An approach that increases the number of user instances in Selenium-based load tests by at least 20% using the same hardware resources.
- A systematic exploration of various testing scenarios (i.e. headless vs. regular browsers and persistent vs. non-persistent browsers) for Selenium-based load

testing.

Chapter Organization. The remainder of this chapter is organized as follows: Section 4.1 discusses the background of Selenium-based load testing. Section 4.2 presents our experimental design. Section 4.3 outlines our systematic exploration of performance issues associated with Selenium-based load testing. Section 4.4 presents the experimental results. Section 4.5 presents the threats to validity of our study. Section 4.6 discusses our various observations. Section 4.7 presents the conclusion.

4.1 Load testing using Selenium

Although Selenium is predominantly used for testing the functionality of web applications, there have been some prior attempts to use it to perform load testing as well (Dowling and McGrath (2015)). In order to use Selenium for load testing, multiple browsers are launched simultaneously. Due to the heavy resource overhead of launching a browser, a simplified version of a browser (without GUI) or a headless browser is widely used in Selenium-based tests (Blazemeter (2018)).

Both regular and headless browsers are considered in our Selenium-based load tests. In the case of regular browsers, in order to reduce the overhead caused by GUI of several browsers, we make use of Xvfb displays (XVFB (2018)) as a virtual rendering engine that requires lower resources relative to rendering to an actual display. In order to perform load tests using Selenium, a browser instance is typically launched for every user instance.

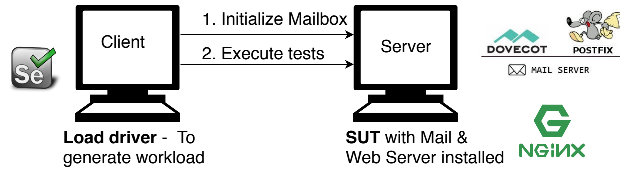

Figure 4.1: Experimental Setup

4.1.1 Load driver and the SUT

The load driver is the application that generates the workloads (i.e., Selenium scripts in our case) and the SUT is the system running the tested web application. Figure 4.1 shows the client and server machines consisting of the load driver and the servers respectively. Prior studies on load testing are mainly concerned with the resource usage of the SUT (Gao and Jiang, 2017; Svard et al., 2015), while we are concerned with the resource usage of the load driver. We measure the resource usages of the SUT to ensure that it is not overloaded. Therefore, any overload observed in our experiments is due to the overloading of the load driver.

4.2 Experimental Design

In this section, we describe our subject web application and our experimental design for load testing of our subject application.

4.2.1 Subject Application and Test Environment

In this work, we use RoundCube [1] as our subject application. RoundCube is an open-source mail client that 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 choose RoundCube because it makes pervasive 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.

Figure 4.1 illustrates our experimental setup. RoundCube can be installed and configured to run on a LAMP/LEMP server (Linux operating system, Apache HTTP Server/ Nginx, MySQL relational database management system, and PHP programming language). In this work, we deployed the Roundcube webmail client on an Nginx web proxy server. We configured Postfix and Dovecot for our SMTP and IMAP servers, respectively. We used MySQL as our mail database. The mail server and client is installed in a Intel Core i7 desktop with 8 cores and 16GB of RAM running ubuntu 14.04.

We run the Selenium tests on a different machine (Client machine in Figure 4.1) that consists of an AMD Phenom desktop with 6 cores and 8GB of RAM running Ubuntu 16.04. We run Selenium tests using Google Chrome (version 69) and Chromedriver (version 2.37).

4.2.2 Load Test Plan

Test Suite

In order to test our subject application, we create a test suite that consists of 8 tasks covering the typical actions that are performed in an email client: composing an email,

[1] https://roundcube.net/

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.

Test Schedule

In our load testing of the subject application, we emulate the scenario in which multiple user instances connect to a mail server and perform email tasks through web browsers. We initialized the mailboxes and contacts of all the virtual users in order to be able to have a fully functioning mail service.

We use an MMB3-based approach (MAPI (Messaging Application Programming Interface) Messaging Benchmark) to schedule the tasks for each of the emulated users. MMB3 is a well-adopted standard for testing mail services (Makhija et al., 2006; Brothers et al., 2008). MMB3 benchmark specifies the number of times that each task is performed during a day, modelled around a typical user's 8 hour work period. According to the benchmark, 295 tasks are scheduled to run in a 8-hour period Exchange (2005). In this work, we reduce the overall execution time while keeping the same intensity of tasks as performed in the MMB3 schedule. Specifically, we run 19 tasks in a 30-minute period.

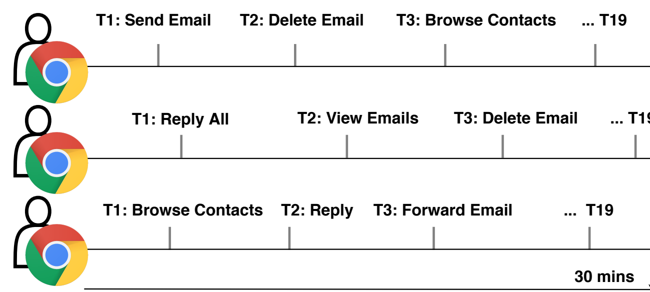

Figure 4.2: Independent browsers setting - Persistent browsers

Figure 4.2 illustrates the scheduling of the tasks in our tests. Each of these 19 tasks is randomly chosen from the eight email tasks as specified by MMB3. Each of the tasks is also randomly scheduled within the testing period such that each task has an equal probability to be scheduled at any point of the testing period (i.e., using a uniform distribution). In a random schedule, one task might be scheduled immediately after another. However, in a real usage case, users always finish one task before starting another. Therefore, we set a minimum gap of 30 seconds between the scheduled time of any two consecutive tasks.

Each user instance performs tasks according to its schedule. A task is not started until its scheduled time. If the load driver is overloaded, tests may take longer time to finish. Therefore, the start of a task might get delayed if the previous task has not been completed. When a task is delayed, it is performed immediately after the completion of the previous task. The browser waits until the scheduled time in order to execute the commands in the task (i.e. the email action). If the scheduled time has already passed, the browser executes the task immediately. We record the number of tasks that missed their scheduled time since that measure would indicate an overloaded state.

Prior to starting a load test, every user's mailbox is cleared and loaded with new emails. This initialization step is done in order to avoid varying pre-test status which might introduce noise that impact the results of tests. Further, as the tasks consists of actions that modify the number of emails in the mailbox, this step is crucial to avoid tests from failing due to the lack of emails in the mailbox. For instance, tests such as view emails and delete emails would fail if the mailbox does not contain any email.

Browser setting	Definition
Persistent	A browser that is opened at the start of the load test and terminated at the end of the load test i.e., the browser persists through the entire test session
Non-persistent	A browser that is opened at the start of each task and terminated at the end of the task i.e., the browser does not persist through the entire test session
Independent	A browser that is associated to one user instance
Shared	A browser that is shared among user instances
Headless	A browser that lacks a GUI component i.e., no display
Regular	A regular browser that is used by users for browsing purposes
Regular with Xvfb display	A regular browser with display transferred to a virtual frame buffer i.e., a dummy display

Table 4.1: Browser definitions

4.2.3 Load test Execution

We execute the load tests using independent browsers setting and shared browsers setting (our approach). Table 4.1 defines the various browser settings. We repeat the load test in each experimental setting for a given number of user instances and browser instances 5 times in order to obtain consistent results. We identify the maximum number of error-free user instances when the median error rate is more than 0 among the 5 repetitions of the experiment. Repetitions are done to ensure that the test errors are not due to random factors such as speed of the network. Further, we use a different random schedule for each repetition to ensure that the results are not dependent on the scheduler. We record the task schedules as the same schedules are used across different experimental settings.

Independent browsers setting

For the independent browsers setting, we have two possible scenarios: 1. Where browser instances are re-used for all tasks (Independent persistent browsers) and 2. Where browser instances are relaunched for every task (Independent non-persistent browsers).

Independent persistent browsers. In this scenario, tasks for each user instance are executed in the same browser, without relaunching browsers for every task, as illustrated

in Figure 4.2. For each user instance, we open an independent browser and execute all the tasks for that user instance according to the scheduled time. We terminate the browser once all the tasks of that user instance are completed. There are some drawbacks in using persistent browsers: apart from the difficulty in identifying the origin of errors, there is a dependence in consecutive tasks that must be properly handled in the tests, i.e. the failure of one task can affect the execution of the following task. For example, when testing a webmail client, if a previous task fails before logging out, the following task assumes that the current page is at the login screen and therefore fails when the username or password cannot be located. Therefore, extra checks are needed to ensure that the previous task does not impact the following task (e.g., to ensure that the following task starts with the login page). Persistent browsers also accumulate local and session storage data of all tasks as the browsers are active for the entire testing period. This causes a memory overhead in comparison to non-persistent browsers which are re-launched for every task.

Independent non-persistent browsers. In this scenario, browsers are launched for every task as shown in Figure 4.3. For each task scheduled for each user instance, a new browser instance is launched at the scheduled start time of each task; we terminate the browser instance once the task is completed. In this setup, task dependence is not a problem as using a new browser for each task ensures a clean session. Independent non-persistent browsers do not require extra checks to ensure a new session. We use the same task schedules as those used in independent persistent browsers in order to be able to compare the results between the two browser settings (persistent and non-persistent).

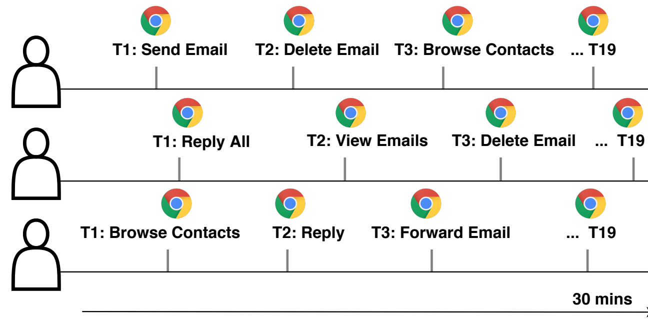

Figure 4.3: Independent browsers setting - Non-persistent browsers

Shared browsers setting

We developed an approach that can reduce the overhead introduced by the frequent relaunching of browser instances. Our shared browsers approach shares browsers among user instances in order to use the resources effectively and eventually increase the maximum number of error-free user instances that can run on the same testing machine. Tools such as JMeter also employ a similar approach to manage their thread groups to load test several users.

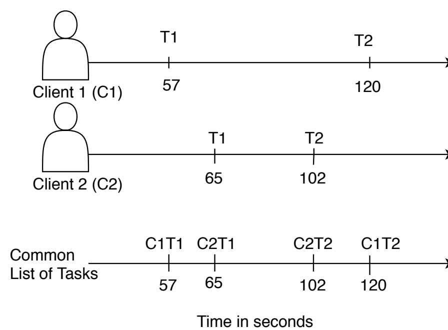

Figure 4.4: The merging of tasks across different user instances to produce a common list of tasks

As browsers are resource-intensive components, reducing the number of browsers intuitively reduces the resulting error rate. In order to execute tasks with fewer browsers,

we combine the schedules of all clients to make a common list of tasks sorted based on their scheduled start time (starting from the earliest task). Figure 4.4 shows an illustration of how the tasks are gathered from all the clients to form the common list of tasks. Client C1 has a task that is scheduled to start at the 57th second (Task C1T1) since the beginning of the test and another task that starts at the 120th second (Task C1T2); Client C2 has a task that is scheduled to start at the 65th second (Task C2T1) and another task that starts at the 102th second (Task C2T2). These tasks are executed in the following order: C2T1, C2T1, C2T2 followed by C1T2. This list of tasks is shared by all the browsers. We use a pool of available browsers to execute the scheduled tasks. Figure 4.5 illustrates the overall workflow of our approach. Each available browser picks the earliest task that has not been scheduled. A browser is removed from the pool of available browsers each time it picks a task to execute. A browser is restored back to the pool after performing its picked up task. One can set the size of the pool of available browsers, in order to control the maximum number of browsers running at the same time. As we limit the number of browser instances (using the size of the pool), our approach can reduce the peak usage of resources (CPU or memory) which is the main reason for the errors in the load driving machine.

Since the list of tasks to be executed is shared by all the browsers, we deal with the following two challenges:

- Preventing browsers from running the same task

  Since browser instances can access the task-queue at the same time, browsers can pick the same task. Therefore, we use a lock to ensure that only one browser instance accesses the list of tasks to perform read or write operations at any time.

- Preventing browsers from running tasks for the same client simultaneously

  In addition to the above problem of running the same task, two browsers can be assigned different tasks for the same client. This can cause conflicts during tests. For example, running delete email and reply email tasks of the same user instance at the same time could lead to a conflict if a browser is trying to reply to an email while the same email is being deleted by the other browser. In order to address this problem, we use a list to record the current active users. This list is updated throughout the test. Before a task is assigned to a browser, the list is checked to ensure that the user is not already active. If the task belongs to an active user instance, the next task is picked from the pool.

Similar to the independent browsers setting, our approach of sharing browsers applies to both the scenarios of using persistent browsers and non-persistent browsers:

Shared persistent browsers. In this scenario, we make use of fewer browsers to do the tasks for a larger number of user. Browsers are launched at the start of the test and re-used for all the tasks until all the tasks are executed. Figure 4.6 illustrates this setup.

We launch a specified number of browser instances. Figure 4.5 shows how each shared browser instance executes the scheduled tasks from the common list of tasks.

Shared non-persistent browsers. Similar to the shared persistent browsers setting, we use a fewer number of browsers to run a larger number of user instances. We follow the same steps as listed for the shared persistent browsers setting. The difference is that we use non-persistent browsers i.e., we relaunch browsers for each task by opening a browser at the scheduled start time of the task and terminate it after the task is executed.

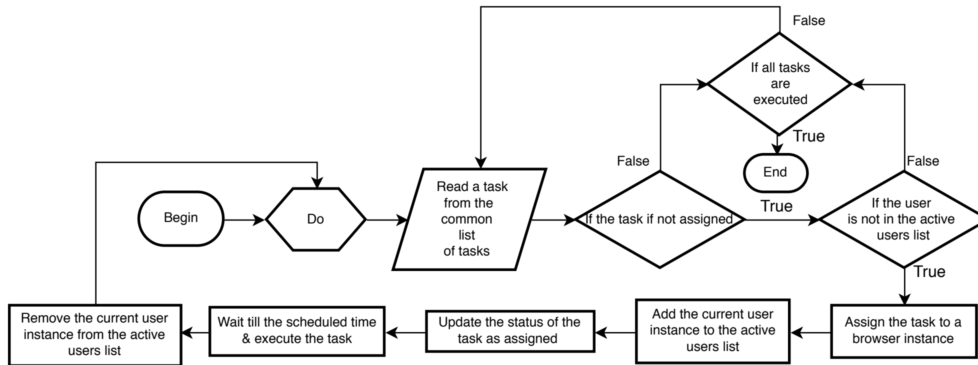

Figure 4.5: Steps to execute scheduled tasks in each browser

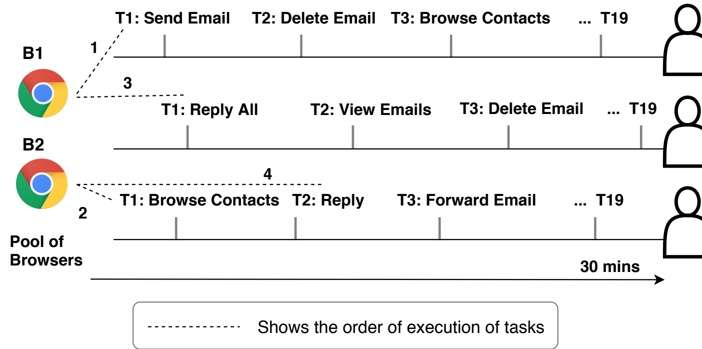

Figure 4.6: Our approach of using shared persistent browsers

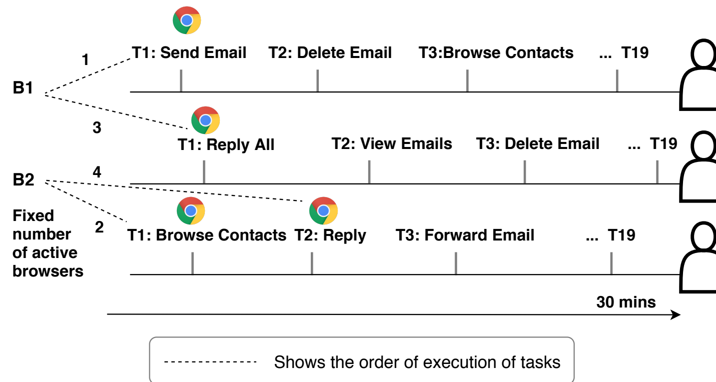

Figure 4.7: Our approach of using fixed number of non-persistent browsers

Table 4.2: Maximum number of user instances in each experimental setting

	Headless		Regular	
	Persistent	Non-persistent	Persistent	Non-persistent
Independent browsers setting[1]	50 user instances	50 user instances	18 user instances	20 user instances
Shared browsers setting[2]	60 user instances	60 user instances	22 user instances	25 user instances

[1] The maximum number of error-free user instances is less than the numbers listed. The numbers listed here indicate the maximum number of user instances when the test driver machine reaches an overloaded state (i.e., when the median error rate is more than 0).
[2] The maximum number of error-free user instances is slightly more than the numbers listed. The numbers listed here indicate the maximum number of user instances before the system reaches an overloaded state.

4.2.4 Load test analysis

The goal of this work is to improve the testing efficiency of Selenium-based load tests. We analyze the results from different experimental settings to identify the maximum number of user instances before the test machine (running the load driver) reaches an overloaded state. We ensure that the SUT itself is not overloaded. We use the following four metrics to measure the performance of our load testing:

Error rate

This metric is given by the number of failed tasks among the total number of tasks. Errors (i.e. performance errors) start to occur when the load driver is overloaded. Therefore, the error rate is used to indicate an overloaded state. We use error rate as our primary metric to determine the maximum number of user instances that can run on a load testing machine. In each experimental setting, we obtain the maximum number of user instances when the median error rate is more than zero across 5 repetitions of a test.

Delay ratio

Delay ratio is defined as the ratio of tasks that miss their scheduled start time. In an overloaded state, a task may take longer time to finish. Therefore, the following tasks might miss their starting time (i.e. get delayed) when the previous task is not completed before the scheduled time of the task under consideration. As the time taken by individual tasks can increase with the number of user instances, this metric serves as an indicator to detect an overloaded load driver. For example, for a given test schedule, when we execute a load test for 5-10 user instances, the average time to complete a task is 10 seconds. However, the average time is about 20 seconds when we execute more than 30 user instances. Therefore, the delay ratio of individual tasks is also an indicator of overloading. We report the median delay ratio based on the delay ratio obtained from the 5 repetitions of a test.

Resource usage

We record memory and CPU values of all processes (Chrome, Chromedriver and Selenium scripts) for every second; then we aggregate the values across threads (running browser instances). We calculate the median and 95th percentile values from the distribution of the recorded CPU and memory values every second. The median resource usage values give an overall estimate of the used resources during a load test and the 95th percentile values account for the spikes (or the peak usage) in resources. We monitor resources (i.e. CPU and memory) using pidstat. [2]

[2]https://linux.die.net/man/1/pidstat

Runtime

We also monitor the overall runtime (the time taken to execute the load test). When the load driver is not overloaded, all the tests should finish within the scheduled time (e.g., 30 minutes). From the runtime metric, we obtain another perspective about the load driver's overloaded state e.g., a runtime of over 31 minutes would indicate an overloaded state as the tasks are expected to complete within 30 minutes.

4.2.5 Identifying the maximum number of error-free user instances

As there is no tool or technique to identify the number of error-free user instances that can run on a system, we increase the number of user instances gradually in a step-wise manner (Goldschmidt et al. (2014); Neves et al. (2013)). We start with 5 user instances and increase the number of user instances based on the performance measures. In the independent browsers setting, we stop increasing the number user instances when the median error rate is more than 0 for 5 repetitions of the load test. Therefore, we identify the maximum number of user instances when the load driver has reached an overloaded state. Whereas in our approach, we report the maximum number of error-free user instances before the load driver reaches an overloaded state. Our goal is to identify the overloaded state using the current state-of-the-art practices (i.e., the independent browsers setting); then apply our approach to identify the maximum number of error-free user instances that can run in the same testing machine.

Although the stopping criteria is based on error rate (our primary metric), we monitor the delay ratio to tune our parameters (number of browsers for the number of user instances in the workload) in our approach. For example, we increase the number of browsers when the delay ratio is high. As the number of errors increase with more

browsers, it is important to use an optimal number of browsers that result in no errors and minimal delays.

4.3 A systematic exploration of the performance overhead of Selenium-based load testing

In this section, we investigate the performance overhead of Selenium-based load testing. We consider three different browser settings: 1) regular browser setting: the same browser setting that end users use when they browse web applications; 2) regular browser with Xvfb displays: same as the regular browser setting except that the display is redirected to a dummy device; 3) headless browser setting: a browser setting without graphical user interface.

In order to compare the performance overhead of Selenium for different browser settings, we repeat the load test for 10 user instances in all three browser settings. We use persistent browsers to perform the load test with the workload consisting of 19 tasks for a period of 30 minutes. We monitor the CPU and memory values during the duration of the load test. As shown in Figure 4.8, regular browsers with Xvfb displays are more efficient than regular browsers with actual displays; headless browsers are more efficient (in terms of CPU and memory) than regular browsers and regular browsers with Xvfb displays. Therefore, headless browsers are best suited for browser-based load testing in terms of testing efficiency.

Only a limited number of user instances can run in a given system as selenium tests are resource- heavy. Therefore, we perform experiments to identify the performance

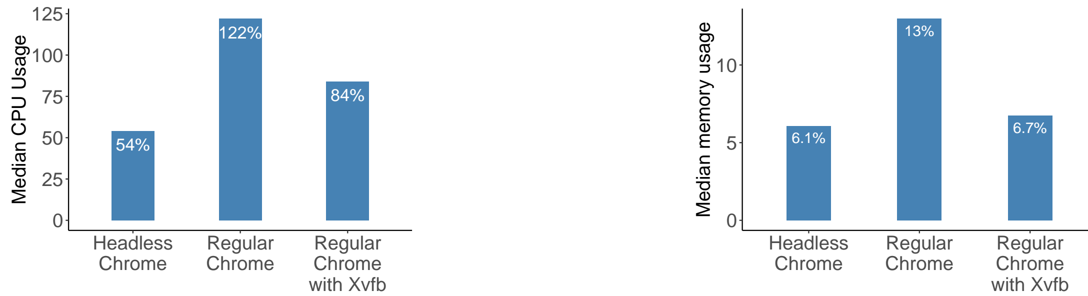

Figure 4.8: Median CPU and Memory in different browsers

critical components in a Selenium test. Broadly, a Selenium test has two main components: a browser that is used to execute the test and the Application Under Test (AUT). We focus on studying the performance of the browser under different settings.

Figure 4.9: Median CPU and memory values for headless and regular browsers

In our experiment, we compare the resource usage of 10 idle and busy browsers (i.e., 10 user instances) for a period of 10 minutes. An idle browser (as the name suggests) is a browser that does not perform any action after being launched, whereas a busy browser performs actions throughout the given period (10 minutes in our experiment). We perform this experiment for both headless and regular browsers (with Xvfb displays). From the median CPU and memory usage (as shown in Figure 4.9), we observe that headless browsers consume less resources compared to regular browsers.

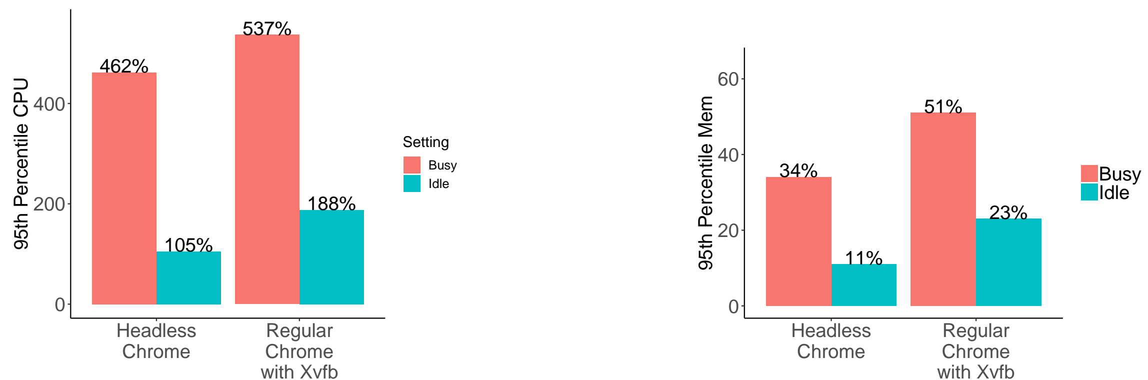

Figure 4.10: 95th percentile CPU and memory values for headless and regular browsers

Further, the median CPU for a busy headless browser is 324% whereas the median CPU for idle headless browser is 5.5%. The median memory for a busy headless browser is 28.5% whereas the median memory for idle headless browser is 3.4%. Therefore, we observe that each browser instance consumes resources irrespective of being idle or busy. Moreover, in an actual test with persistent browsers, an idle browser may take more resources as a web page might be loaded before it reaches an idle state.

We also measure the 95th percentile values (as shown in Figure 4.10) to understand the peak usage as we see errors when the system is in an overloaded state. Therefore, the 95th percentile values is directly related to errors. This metric is particularly important for our approach because our goal is to increase or maximize the number of error-free user instances. From the 95th percentile values of CPU and memory, we see that an idle browser instance consumes a huge amount of resources: almost 1/4th of the memory in headless instances and half of the memory in regular browser with Xvfb displays.

4.4 Experimental Results

We perform experiments using both the independent browsers setting and shared browsers setting (our approach) to compare the performance of the two settings and identify the maximum number of error-free user instances in our approach.

4.4.1 Independent vs Shared browsers in the headless browser setting

Table 4.2 shows the maximum number of user instances in each of the 4 experimental settings. In persistent and non-persistent headless browsers, we were able to increase the number of user instances that can run on the client machine by **20% (from 50 to 60 user instances)**.

Persistent browsers

We compare the number of user instances that can run in a persistent headless browser in both the independent and shared browsers setting. Table 4.3 shows the performance measures for our experiments using persistent headless browsers. We observed performance errors when we reached 50 user instances in the independent browsers setting. We also repeated the experiment in the independent browsers setting for 60 user instances. The median error rate increases when the number of user instances increases in the independent browsers setting i.e., from 50 to 60 user instances. Whereas, with our approach, we used 20 browsers to run up to 60 error-free user instances. Therefore, we end up increasing the number of error free user instances in this setting by 20%.

The median CPU values is reduced in our approach as shown in Table 4.3. Although

the median memory is higher in our approach, our approach reduces the 95th percentile values of both CPU and Memory as we run fewer browser instances. Reducing the peak resource usage values reduces the possibility of errors. The runtime value obtained for all 3 experiments (50 and 60 user instances using independent browsers setting, 60 user instances using shared browsers setting) indicates that the system is not overloaded. We also observe that the delay ratio is 0 in our approach indicating that the load driver is not overloaded.

Non-persistent browsers

We compare the number of user instances that can run in a non-persistent headless browser using the independent and shared browsers setting. In this setting, browsers are re-launched for every task. As shown in Table 4.3, performance errors occur when we reached 50 user instances in the independent browsers setting. We also tried running 60 user instances to compare the performance of our approach with the independent browsers setting. The experiment with 60 user instances in the independent browsers setting resulted in errors and delays. By using 15 browsers, we were able to run up to 60 error-free user instances. Therefore, we increase the number of error-free user instances in this setting by 20%.

As shown in Table 4.3, we observe that the CPU values are higher in our approach. However, the memory values are consistent across both settings. We also observe that the delay ratio is negligible in our approach. By restricting the number of active browser instances, we obtain zero errors even for 60 user instances.

	Persistent headless			Non-persistent headless		
	Independent Browsers Setting (50 users)	Independent Browsers Setting (60 users)	Shared Browsers Setting (20 browsers, 60 users)	Independent Browsers Setting (50 users)	Independent Browsers Setting (60 users)	Shared Browsers Setting (15 browsers, 60 users)
Median CPU	259	297	267	228	252	275
Median Memory	20.4	23.38	28.26	17.81	20.07	20.1
95th Percentile CPU	407.3	438	410.79	400	427.3	445.5
95th Percentile Memory	48.70	55.98	41.56	31.77	33.59	33.27
Runtime	1814	1823	1820	1818	1808	1811
Median Error rate	0.001	0.016	0	0.001	0.001	0
Median Delay ratio	0	0	0	0	0.001	0.003

Table 4.3: Performance measures in headless browsers

	Persistent regular chrome (with Xvfb display)			Non-persistent regular chrome (with Xvfb display)		
	Independent Browsers Setting (18 users)	Independent Browsers Setting (22 users)	Shared Browsers Setting (10 browsers, 22 users)	Independent Browsers Setting (20 users)	Independent Browsers Setting (25 users)	Shared Browsers Setting (6 browsers, 25 users)
Median CPU	279	331	343.335	129	159	161
Median Memory	20.77	30.02	27.69	18.11	22.58	15.485
95th Perc. CPU	477.4	496	500	294.75	331.95	328
95th Perc. Mem	40.182	50.39	43.218	28.136	34.47	22.791
Runtime	1840	1864	1822	1813	1807	1825
Median Error rate	0.009	0.007	0	0.005	0.004	0
Median Delay ratio	0	0.01	0.01	0	0.008	0.021

Table 4.4: Performance measures in regular browsers (with Xvfb)

4.4.2 Independent vs Shared browsers in the regular browser setting

Apart from emulating real-user behaviour, regular browsers come with all the features provided by Chrome such as access to chrome settings, add-ons, extensions etc. Although regular browsers are less efficient, they are still being used to perform browser-based load testing. Table 4.2 shows the maximum number of user instances in regular browsers. We were able to increase the number of user instances that can run on the client machine by **22% in persistent browsers** and **25% in non-persistent browsers** using the regular browser setting.

Persistent Regular Chrome

We compare the number of user instances that can run in a persistent regular browser (Chrome) using an Xvfb display in the independent and shared browsers setting. We observed performance errors when we reached 18 user instances in the independent browsers setting. We also executed the load test for 22 user instances in the independent browsers setting. Using 10 browsers, we were able to run up to 22 error-free user instances in our approach. Therefore, we were able to increase the number of error-free user instances in this setting by 22%.

From Table 4.4, we observe that our approach uses less median memory although the median CPU values are slightly higher. The runtime metric shows that our approach does not overload the load driver. However, the runtime for the experiments in the independent browsers setting (1864 seconds for 22 user instances) shows signs of overloading. This result further shows that the number of users identified is the maximum number of error-free user instances that can run on the testing machine.

Non-Persistent Regular Chrome

We compare the number of user instances that can run in a non-persistent regular browser (Chrome) with Xvfb display in independent and shared browsers setting. We observed performance errors when we reached 20 user instances in the independent browsers setting. By using 6 browsers, we were able to run upto 25 error-free user instances in our approach. Therefore, we increase the number of users in the setting by 25%. We also executed 25 user instances using the independent browsers setting and observed higher error rate and delay ratio compared to load testing with 20 user instances in the independent browsers setting, we reduce the median memory and the

peak values of CPU and memory using our approach.

In regular browsers, the maximum number of users is not the same in persistent and non-persistent settings unlike headless browsers. We observe that persistent regular browsers are less efficient compared to non-persistent regular browsers as seen in Table 4.4. The CPU and memory values in persistent browser is significantly higher than the non-persistent counterpart. Further, we use a smaller number of browsers in non-persistent regular browsers setting because relaunching browsers cause spikes in the CPU/Memory values.

In a persistent browser setting, a specific number of browsers is always active. However, in a non-persistent browser setting, the actual number of active browsers is the minimum of the specified number of browsers and the number of active tasks (a browser is only launched for each task). Such an effect is significant for regular browsers but less significant for headless browsers, as regular browsers are much heavier (i.e., requiring more resources) than headless ones. Therefore, non-persistent regular browsers have better performance than persistent regular browsers.

The number of delayed tasks is larger in the case of non-persistent regular browsers compared to persistent browsers. This is because we use a very small number of browsers (i.e. 6) to run 25 user instances. As the delays did not affect the error rate and the overall runtime, we identify that our approach (i.e. 6 browser instances for 25 user instances) is more efficient than the independent browsers setting.

4.5 Threats to Validity

4.5.1 Internal Validity

The scheduling of tasks

In this work, we design a test schedule following the MMB3 benchmark. Our approach assumes that the individual tasks of a user instance are not executed one after another i.e., with no pauses in between the tasks. A different schedule may impact our results. However, we follow the schedule of the MMB3 benchmark which is a well established standard for testing mail servers. Furthermore, for other general web applications, users usually have pauses between active tasks. Therefore, our approach can be applied to improve the testing efficiency for any type of web application.

Determining the overloaded state of load drivers

Due to the lack of existing metrics to evaluate Selenium-based load tests, we define custom metrics such as delay ratio and error rate to evaluate our approach. In order to have a holistic view of the load test, we also monitor the traditional performance measures such as runtime, CPU, memory and network resources. Since the network usage is very small (less than 400kbit/second), we observe that network is not a bottleneck in our load tests.

We use zero median error rate to derive the maximum number of user instances that could run on a client machine. We also calculate the delay ratio to ensure that the test schedule is not impacted by running too many user instances. Therefore, we believe that the metrics defined in this work (delay ratio and error rate) can be used by future practitioners as they detect an overloaded state for browser-based load tests.

The metrics defined in this work (delay ratio and error rate) helped us identify the maximum number of user instances before the system reaches an overloaded state. We believe that these metrics could help others as well. Nevertheless, others might wish to define their own metrics if needed.

4.5.2 External Validity

Generalizing our results

Our approach works on the the AUT (i.e., Roundcube) that makes pervasive use of AJAX technology. We expect our approach to work on other AJAX based web applications, which comprises of a large number of current web applications. Our approach can also be generalized to the testing of any web applications that can take concurrent requests from browsers. Practitioners using web applications built using other technologies should examine the applicability of our approach on their web applications.

Performance of browsers

We observed that headless browsers are the most efficient in terms of CPU and memory compared to regular browsers and regular browsers with Xvfb displays. We also observed that the number of error-free user instances is higher when headless browsers are used. However, we tested our approach only on regular Chrome and headless Chrome. The experimental results may vary for other browsers such as Firefox, IE etc. That said, Chrome is the most popular browser among users. Further, Mozilla Firefox's Gecko Quantum browser engine has been replaced by Google Chrome's engine, Chromium (Mozilla (2018)).

Other automation tools

Apart from Selenium, there are other browser automation tools used in end-to-end testing. However, some tools like the open-source version of Cypress (Cypress (2018)) cannot be used to run several instances of browsers. We also performed exploratory tests to check the testing efficiency of automation tools that use visual locators like Sikuli (Sikuli (2018)). Although Sikuli tests can be made to run in parallel, they are considerably more resource consuming. Due to these limitations, we develop an approach only for Selenium tests.

Additional factors

Our results can vary based on factors such as time when the experiment was scheduled, speed of the network etc. However, we mitigate this threat by repeating the experiments at least 5 times in order to obtain consistent results. We also carefully examined each test and removed any tests that were impacted by unexpected network issues (e.g., outages).

4.5.3 Construct Validity

Experimental Setup

Our results (i.e. the maximum number of error-free user instances) heavily depends on the hardware configuration of the client machine. The other factors that impact the results include the complexity of the AUT, browser configuration (headless or regular) and type of browsers (Firefox, chrome etc). Our approach improved the testing

efficiency in both headless and regular browsers. We believe that our results can generalize to other configurations.

Maximum number of error-free user instances

We arrived at the maximum number of error-free user instances in each experimental setting by increasing the number of user instances in small steps in each iteration, following the step-wise workload strategy. Therefore, the number obtained is not accurate. However, the focus of this work is to improve the testing efficiency of Selenium-based load tests and not detect the exact number of user instances that a specific load driver can support. Hence, we present an approximate result (i.e. maximum number of user instances) in each setting and scale by a minimum of 20%.

4.6 Lessons learned

4.6.1 Dealing with flaky tests to enable load testing

Although we use functional test scripts for the load tests, we make a few changes to the scripts to make them usable for load tests. In order ensure that each new task starts in a new session, we perform a logout towards the end of each task i.e. we re-do logout if the first attempt to logout failed. In the same way, we add precautionary steps in the start of each task i.e. if the username is not found, we assume that the old task did not complete and therefore, we execute commands to logout again. We verified that by adding such checks before and after a task, we can successfully avoid cases where the errors or failures of tasks are due to the failure of the previous task.

Apart from checks to ensure new sessions for new tasks, we had to solve the problem of intermittent failures or flaky tests. Flaky tests are caused due to wait times, concurrency and test order dependency:

1. We resolve flaky tests caused due to undesirable wait times by making changes to the polling frequency of Selenium scripts in order to locate HTML elements faster. For example, we changed the polling frequency from 500ms to 100ms to locate an alert box that appears for a very short duration. We observed that tests failed while locating that element even when about 5 user instances are running concurrently i.e. we resolved a functional issue rather than a performance issue.

2. We resolved flaky tests due to concurrent requests by identifying the order of execution of scripts (i.e., JavaScript) and by locating elements in our tests based on their order.

3. We resolved flaky tests due to test order dependency by initializing the mailboxes with sufficient emails for every user and by designing independent tests. We verified the same by running the tests in random order to ensure that there are no dependencies between tests. For example, *test_reply* has no dependency on *test_delete* and therefore, it can be executed in any order.

4.6.2 Dealing with the issues of using headless browsers and regular browsers with virtual displays

Clearing cache in headless Chrome

In order to reduce the memory overhead in persistent browsers, we wanted to clear cache before the task of every task. Using Selenium, it is possible to disable disk cache

using a chrome option to set the size of disk cache as 0. Selenium also provides a way to delete cookies. However, there is no straightforward way that Selenium provides to delete browser cache. With regular browsers, it's possible to delete cache by navigating to chrome://settings and clearing cache like a user would. This option is limited to regular browsers and its not possible to do so in headless browsers.Therefore, this is one of the limitations of persistent headless browsers.

Synchronizing webdriver creation to enable load testing using regular browsers with Xvfb displays

There is a need to synchronize web-driver creation in order to start several regular Chrome browsers. On the same note, Xvfb displays have to be terminated at the same time (i.e. end of the test) in order to avoid errors that occur due to driver-display mismatch. The display and driver instances are independent entities, therefore, even when one of the displays is stopped during the execution of the load test, the remaining tests in the workload fail. However, these additional steps are not necessary in the case of headless browsers as the GUI or lack of GUI is tied to the driver instance.

4.6.3 Removing functional issues in load testing

We observe that failures in Selenium tests can be due to functional or performance issues. In order to identify issues caused as a result of an overloaded system and thereby to evaluate the testing performance, we distinguish performance issues from functional issues: we identify functional errors as the errors that occur for one user instance or very few user instances i.e. when the system is not overloaded. On the other hand, an overloaded system causes performance or load errors. For instance, a timeout that

occurs when one user instance is tested is a functional error whereas a timeout that only occurs when many user instances are tested is identified as a performance error.

We ensure that our test suite has no flaky tests (due to functional errors) by following some of the best test design practices. We executed our tests several times to ensure that no tests are flaky. Wait conditions, concurrency and test order (Google Testing Blog (2016)) are the major causes of flaky tests (Luo et al. (2014)). We efficiently handled all such cases in our tests to ensure that the errors in load tests are only due to client-side performance issues.

One of the major challenges in designing Selenium scripts is estimating the wait time needed for a page to reach the *ready* state (Lee et al. (2018)). Commands fail when the scripts (i.e., javascript) associated with the previous command has not completed its execution. The browser handles scripts sequentially (Innoq Blog (2018)); therefore the current command could be blocked by one of the previous commands. For example, one of the issues we faced was to locate an element after an email is sent. Only after the script to reload the mailbox is complete (i.e., the script associated with the 'send email' command), the next command in the test (i.e. locating an element) will be executed. Therefore, it is crucial to design robust tests by adding explicit commands to ensure that the scripts have completed execution. Otherwise, Selenium tests fail with timeout exceptions while locating elements.

4.7 Conclusion

Browser-based load testing of web applications has many advantages over protocol-level load testing but its not used as much due to the performance issues in Selenium tests. In this chapter, we present an approach that makes use of a pool of browser

instances to execute tests for a large number of users. We perform experiments in different browser settings (headless vs regular, persistent vs. non-persistent) to identify the maximum number of user instances that can run in each setting. We also present our learned lessons using Selenium for load testing. Our approach improves testing efficiency by at least 20%. We believe that practitioners can benefit from our presented approach.

CHAPTER 5

Conclusions and Future Work

THIS chapter summarizes our work and presents potential opportunities for further work.

5.1 Summary

In this thesis, we study the challenges associated with the use of Selenium in practice through the mining of Selenium-related questions on StackOverflow. We study how various programming languages and browsers supported by Selenium are related to other Selenium-related tags. We study the StackOverflow user community for each of the programming language and browser related tags associated with Selenium. Finally,

we identify the difficult aspects of Selenium and the factors that impact the answering speed and the likelihood of getting an accepted answer.

Our results show that questions on programming languages is the most popular and fastest growing among other Selenium-related questions while browser-related questions get the most attention. The number of new questions is increasing for most of the tags except for tags related to 'testing frameworks' which has stabilized over the last few years. Python tag is the fastest growing among other programming language related tags, while Java tag has become stable in the recent years. We also find that Chrome tag is the most frequently used browser tag among other browser tags related to Selenium. However, different browsers may be used for different purposes. For example, PhantomJS is widely used for web scraping while Firefox is frequently used with Selenium-IDE. We observe that less than half of Selenium-related questions get accepted answers. The time taken to get accepted answers is statistically significantly impacted by the number, the median reputation, and the experience level of non-casual answerers of a Selenium-related tag.

We believe that new users of Selenium will benefit from this study by knowing which programming languages and browsers to use depending on the use case. For example, headless browsers such as *PhantomJS* are preferred for web-scraping tasks. Users will also know which topics or tags have good support online based on the time taken to get accepted answers. The insights shared in this work will also help Selenium developers improve their support for the difficult aspects in Selenium.

We learn about the performance issues in Selenium from users' questions on Stack-Overflow and from prior work on Selenium. Therefore, in the next part of the thesis, we investigate the performance overhead of Selenium-based tests in order to improve

the testing efficiency of Selenium-based load testing. We explore the advantages of using Selenium for load testing over Protocol level testing tools like JMeter. We understand that browser-based load testing of web applications has many advantages over protocol-level load testing but its not used as much due to the performance overhead of Selenium tests.

We propose an approach that makes use of a pool of browser instances to execute tests for a large number of users. We perform experiments in different browser settings (headless vs regular, persistent vs. non-persistent) to identify the maximum number of users that can run in each setting. We also present the lessons learned in using Selenium for load testing and the challenges faced in developing Selenium scripts for load testing. Our approach improves testing efficiency by at least 20%. We believe that software practitioners will benefit with the shared browsers approach that we presented in this thesis. The other shared experiences will help users better understand the performance overhead in Selenium tests, the resources consumption of browsers during a Selenium-based load test and how certain browser settings are more efficient than others.

5.2 Future Work

In this section, we explore potential opportunities for improving our work.

- We identify the most popular tags in Selenium based on the number of times they are used in Selenium-related questions. We study only those tags that occur in at least 1% of the total questions. Future research can study all the tags in Selenium in order to get more insights about how Selenium is used. For example,

our study fails to cover new technologies and tools as they might not have enough questions on StackOverflow.

- We perform LDA to verify that the obtained topics are equivalent to the ones that are extracted from the user. Future studies can use LDA to process text from the title and body of questions, answers and comments on Selenium-related posts to gain more details about the issues in Selenium.
- Finding the best combination of browser instances and user instances in our approach of sharing browsers involves a great amount of time and effort. We have to perform several experiments (by tuning parameters manually based on prior results) before we can arrive at the maximum number of error-free users for each of the experimental settings. Therefore, a possible future work would be to develop an automated approach to tune parameters dynamically during the execution of a load test.
- We identify that headless browsers are more efficient than regular browsers, yet many users still use regular browsers for load testing. Therefore, future practitioners can develop an automated approach to migrate test scripts with regular browsers to test scripts with headless browsers (fixing compatibility issues), to enhance the testing efficiency of their tests.

Bibliography

Abad, Z. S. H., Shymka, A., Pant, S., Currie, A., and Ruhe, G. (2016). What are practitioners asking about requirements engineering? an exploratory analysis of social q&a sites. In *2016 IEEE 24th International Requirements Engineering Conference Workshops (REW)*, pages 334–343.

Abbas, R., Sultan, Z., and Bhatti, S. N. (2017). Comparative analysis of automated load testing tools: Apache jmeter, microsoft visual studio (tfs), loadrunner, siege. In *2017 International Conference on Communication Technologies (ComTech)*, pages 39–44.

Altaf, I., Dar, J. A., u. Rashid, F., and Rafiq, M. (2015). Survey on selenium tool in software testing. In *2015 International Conference on Green Computing and Internet of Things (ICGCIoT)*, pages 1378–1383.

Arcuri, A. (2018). An experience report on applying software testing academic results

in industry: we need usable automated test generation. *Empirical Software Engineering*, 23(4):1959–1981.

Barua, A., Thomas, S. W., and Hassan, A. E. (2012). What are developers talking about? an analysis of topics and trends in stack overflow. *Empirical Software Engineering*, 19:619–654.

Bhat, V., Gokhale, A., Jadhav, R., Pudipeddi, J., and Akoglu, L. (2014). Min(e)d your tags: Analysis of question response time in stackoverflow. In *2014 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM 2014)*, pages 328–335.

Blazemeter (2018). Driving headless browser testing with selenium and python | blazemeter. https://www.blazemeter.com/blog/driving-headless-browser-testing-with-selenium-and-python. (Accessed on 11/01/2018).

Blondel, V. D., Guillaume, J.-L., Lambiotte, R., and Lefebvre, E. (2008). Fast unfolding of communities in large networks.

Brothers, T., Mandagere, N., Muknahallipatna, S., Hamann, J., and Johnson, H. (2008). Microsoft exchange implementation on a distributed storage area network. *International Journal of Computers and Applications*, 30(3):251–264.

CBC (2017). Design flaws crashed statscan's census website: documents | cbc news. http://www.cbc.ca/news/politics/census-statistics-canada-computers-online-webpage-1.3649989. (Accessed on 04/01/2018).

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 04/01/2018).

Chaulagain, R. S., Pandey, S., Basnet, S. R., and Shakya, S. (2017). Cloud based web scraping for big data applications. In *2017 IEEE International Conference on Smart Cloud (SmartCloud)*, pages 138–143.

Cypress (2018). Javascript end to end testing framework | cypress.io. https://www.cypress.io/. (Accessed on 11/05/2018).

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.

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

Exchange (2005). Exchange performance result. https://www.dell.com/downloads/global/solutions/poweredge6850_05_31_2005.pdf. (Accessed on 11/05/2018).

Expect (2018). Expect tool. https://core.tcl.tk/expect/index. (Accessed on 01/23/2019).

Gao, R. and Jiang, Z. M. (2017). An exploratory study on assessing the impact of environment variations on the results of load tests. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 379–390.

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.

Goldschmidt, T., Jansen, A., Koziolek, H., Doppelhamer, J., and Breivold, H. P. (2014). Scalability and robustness of time-series databases for cloud-native monitoring of industrial processes. In *2014 IEEE 7th International Conference on Cloud Computing*, pages 602–609.

Google Testing Blog (2016). Google testing blog: Flaky tests at google and how we mitigate them. https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html. (Accessed on 11/01/2018).

Innoq Blog (2018). How browsers load and process javascript. https://www.innoq.com/en/blog/loading-javascript/. (Accessed on 11/02/2018).

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.

Kiran, S., Mohapatra, A., and Swamy, R. (2015). Experiences in performance testing of web applications with unified authentication platform using jmeter. In *2015 International Symposium on Technology Management and Emerging Technologies (ISTMET)*, pages 74–78.

Kochhar, P. S. (2016). Mining testing questions on stack overflow. In *Proceedings of the 5th International Workshop on Software Mining*, SoftwareMining 2016, pages 32–38, New York, NY, USA. ACM.

Kongsli, V. (2007). Security testing with selenium. In *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*, OOPSLA '07, pages 862–863, New York, NY, USA. ACM.

Le Breton, G., Maronnaud, F., and Hallé, S. (2013). Automated exploration and analysis of ajax web applications with webmole. In *Proceedings of the 22Nd International Conference on World Wide Web*, WWW '13 Companion, pages 245–248, New York, NY, USA. ACM.

Lee, S., Chen, Y., Ma, S., and Lee, W. (2018). Test command auto-wait mechanisms for record and playback-style web application testing. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, volume 02, pages 75–80.

Leotta, M., Clerissi, D., Ricca, F., and Spadaro, C. (2013). Comparing the maintainability of selenium webdriver test suites employing different locators: A case study. In *Proceedings of the 2013 International Workshop on Joining AcadeMiA and Industry Contributions to Testing Automation*, JAMAICA 2013, pages 53–58, New York, NY, USA. ACM.

Luo, Q., Hariri, F., Eloussi, L., and Marinov, D. (2014). An empirical analysis of flaky tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 643–653, New York, NY, USA. ACM.

Makhija, V., Herndon, B., Smith, P., Roderick, L., Zamost, E., and Anderson, J. (2006). Vmmark: A scalable benchmark for virtualized systems. Technical report, Citeseer.

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*, ASE '14, pages 67–78, New York, NY, USA. ACM.

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.

Mozilla (2018). Goodbye, edgehtml - the mozilla blog. https://blog.mozilla.org/blog/2018/12/06/goodbye-edge/. (Accessed on 01/20/2019).

Neves, P., Paiva, N., and Durães, J. a. (2013). A comparison between java and php. In *Proceedings of the International C* Conference on Computer Science and Software Engineering*, C3S2E '13, pages 130–131, New York, NY, USA. ACM.

NY Times (2013). Inside the race to rescue a health care site, and obama - the new york times. https://www.nytimes.com/2013/12/01/us/politics/inside-the-race-to-rescue-a-health-site-and-obama.html?pagewanted=all. (Accessed on 04/01/2018).

Ponzanelli, L., Mocci, A., Bacchelli, A., and Lanza, M. (2014). Understanding and classifying the quality of technical forum questions. In *2014 14th International Conference on Quality Software*, pages 343–352.

Rosen, C. and Shihab, E. (2016). What are mobile developers asking about? a large scale study using stack overflow. *Empirical Software Engineering*, 21(3):1192–1223.

Sikuli (2018). Sikuli script - home. http://www.sikuli.org/. (Accessed on 11/05/2018).

Slag, R., d. Waard, M., and Bacchelli, A. (2015). One-day flies on stackoverflow - why the vast majority of stackoverflow users only posts once. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 458–461.

StackOverflow (2017). javascript - selenium starts running slower and slower - stack overflow. https://stackoverflow.com/questions/43824636/selenium-starts-running-slower-and-slower. (Accessed on 01/23/2019).

StackOverflow (2018a). performance - are all automated testing softwares slow? having speed issues with selenium - stack overflow. https://stackoverflow.com/questions/50959977/. (Accessed on 01/23/2019).

StackOverflow (2018b). performance - python selenium: Send keys is too slow - stack overflow. https://stackoverflow.com/questions/49956239/python-selenium-send-keys-is-too-slow. (Accessed on 01/23/2019).

StackOverflow (2018c). Selenium performance issue(1000+ instances). https://stackoverflow.com/questions/52879827/selenium-performance-issue1000-instances. (Accessed on 01/23/2019).

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.

Svard, P., Li, W., Wadbro, E., Tordsson, J., and Elmroth, E. (2015). Continuous datacenter consolidation. In *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 387–396.

Venkatesh, P. K., Wang, S., Zhang, F., Zou, Y., and Hassan, A. E. (2016). What do client developers concern when using web apis? an empirical study on developer forums and stack overflow. In *2016 IEEE International Conference on Web Services (ICWS)*, pages 131–138.

Vila, E., Novakova, G., and Todorova, D. (2017). 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, New York, NY, USA. ACM.

XVFB (2018). Xvfb. https://www.x.org/releases/X11R7.6/doc/man/man1/Xvfb.1.xhtml. (Accessed on 11/02/2018).

Yang, X.-L., Lo, D., Xia, X., Wan, Z.-Y., and Sun, J.-L. (2016). What security questions do developers ask? a large-scale study of stack overflow posts. *Journal of Computer Science and Technology*, 31(5):910–924.