An automated parallel compatibility testing framework fo web-based systems

Yeisson Chicas, Stephane Maag

AN AUTOMATED PARALLEL COMPATIBILITY TESTING FRAMEWORK FOR WEB-BASED SYSTEMS

Yeisson Chicas and Stephane Maag
Télécom SudParis, Samovar, Institut Polytechnique de Paris, France

ABSTRACT

With the growth and wide use of web-based applications in many domains, it is crucial to check their conformance with regards to their requirements. Among the important set of testing types, compatibility testing is of high importance. Indeed, ubiquity is required and the way of interacting with these applications can be performed in many manners. Compatibility testing aims at determining if the web application is proficient enough to run in different browsers, database, hardware, operating system, mobile devices, networks, etc. In this context, one challenge for compatibility testing is how to execute multiple tests cases, in a correct and efficient way, that may cover several environments and functionalities of the tested applications, while reducing the consumed resources and time. In our work, we propose a methodology to efficiently perform compatibility tests through several environments with different versions of operating systems (OS) and browsers. We emphasize on resource consumption improvement by using parallel testing and containerization.

KEYWORDS

Compatibility testing, Automation testing, Selenium Grid, Docker, Web-based systems.

1. INTRODUCTION

Web-based applications are part of our daily lives. According to (WebsiteSetup Editorial, 2021) it is estimated that around 1.7 billion web pages currently exist used by 4.5 billions people all over the world.

With the growth and use of all these web-based applications in many domains (e.g., industry, financial, academia, security, etc.), they need to be safe, conform to their requirements in order to provide the expected functionalities. It is therefore necessary to test them. There exist several research works, approaches, methods and tools to test such webRTC applications (Al-Ahmad and Al Debei, 2020). Among the important set of testing types, compatibility testing becomes more and more important. Indeed, ubiquity is needed and the way of interacting with these applications can be performed in many manners.

Compatibility testing has diverse definitions. The one we will use here is the process to determine whether the web application is proficient enough to run in different browsers, database, hardware, operating system, mobile devices, networks, etc[1].

In this context, one challenge for compatibility testing is how to execute multiple tests cases, in a correct and efficient way, that may cover several environments and functionalities of the tested applications. Another important aspect to consider is the maintenance of testing processes when a web-based application is modified. Indeed, this can cause the whole process to become poor, slow, consuming too many resources. In our work, we propose a methodology to efficiently and correctly perform compatibility tests through several environments and contexts with different versions of operating systems (OS) and browsers. Furthermore, we emphasize on resource consumption improvement by using parallel testing and containerization. Finally, we summarize our main contributions.

- we propose a framework based on containerization and test parallelization for compatibility testing with resources and time reduction.
- we demonstrate that our approach improves the time processing when testing compared to traditional and sequential testing.
- thanks to our methodology, we show that, although not commonly used in the Internet, some browsers behave very well and pass many of our tests cases.

[1] https://www.softwaretestinghelp.com/software-compatibility-testing/

2. RELATED WORKS

WebRTC automation testing is studied for some years now (Garcia et al., 2017) and there exist several research papers dedicated to compatibility testing in many areas such as vulnerability detection (Hayek et al., 2019), GUI (Ki et al., 2019) or cross-browsers domains (Liu et al., 2019). However, there are few if we consider compatibility testing of web-based systems such as web applications. We cite in the following the related works we get inspired of.

A very first work on functional testing was proposed by Garcia et al in (Garcia et al., 2016). The authors present a framework based on Selenium for functional test cases and the assessment of quality of experience (QoE) while using web services. Although this work is not determined to compatibility testing, the metrics they use for the assessment of the quality of WebRTC applications are relevant.

Recently, Al-Ahamad et al provided an up-to-date survey on the testing methods for web applications (Al-Ahmad and Al Debei, 2020). Compatibility testing approaches are reviewed and compared. Giving detailed definitions, viewpoints, architectures and interesting challenges especially in terms of test cases execution. The authors highlight the main challenges and issues the tackle in this area.

In his Master's thesis (Heinonen, 2020), J. Heinonen shows the importance of parallelizing the testing process while performing compatibility testing through multiple browsers. Configurations and testbed setup is of high importance as mentioned into that paper. The author raised the difficulty to orchestrate docker and to distribute the test cases as well as the required resources. This is what we manage in our work by using Selenium Grid and Docker.

Another very recent and relevant work is (Bertolino et al., 2020). It deals with the distribution of test cases through the parallelization of the executions. They use a platform prototype, ElasTest, for evaluating the QoE of WebRTC applications. Their work is innovative and demonstrates the needs and efficiency of virtualization and parallel executions for testing. However, their approach is not dedicated to compatibility testing, the resources are not evaluated and real experiments are not deeply studied.

Besides, Tanaka provides formal definitions and ways for designing test suites in (Tanaka, 2019). The approach is dedicated to visual compatibility testing using selenium. However, the parallelization and containerization are not performed. This is the same in (Yu, 2019) in which relevant approaches for compatibility testing are presented demonstrating the importance of combinatorial tests of various settings, aspects considered in our work and eased by our methodology.

We also study the work of Villanes et al. (Villanes et al., 2020) devoted to test cases exploration and in particular, the ways to decide which use cases could be useful in terms of compatibility testing scenarios. This is somehow what we try to experience in our paper by providing our testing verdicts.

In our paper, we get inspiration of all these related works and present a novel approach based on containerization and parallelization of multiple test suites execution for web applications compatibility testing.

3. BACKGROUND

3.1 Selenium

Selenium[2] is an automated testing framework, open-source, based on JavaScript, used for web application testing. It allows to run tests directly with different browsers: popular browsers like Firefox, Google Chrome, Safari, enabling interactions with the desired websites. It reduces repetitive manual testing that consumes time and effort. Selenium is quite popular in the industry[3] and recent studies make mention that it is one of the best frameworks (Garcia et al., 2020).

[2] https://www.selenium.dev
[3] https://enlyft.com/tech/products/selenium

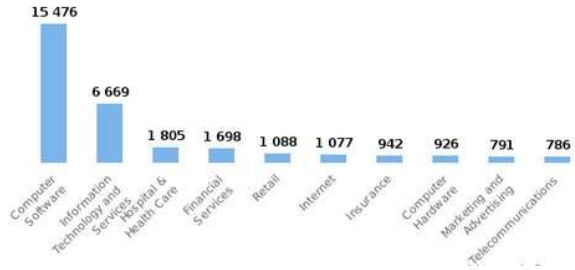


Figure 1. Distribution of companies using Selenium by Industry

Selenium has advantage of allowing testers to write their own tests by providing some templates and standards, as well as a lot of flexibility and portability across multiple operating systems such as Windows, Linux and Mac OS. Selenium can be controlled from various programming languages including Java, Python, PHP, C# and others. It also allows integration with other tools.

Selenium WebDriver It is a tool used to automate the testing of web applications and handling a browser in a native way. It handles the browser as a real user. Basically, WebDriver provides an interface to create and execute test scripts in an automated way, it communicates through an API which sends commands to control the different browsers.

Selenium Grid This tool enables to run tests in parallel across multiple machines at the same time, making a considerable reduction of time. It is an ideal tool for our approach that deals with parallel tests executions in different environments with different versions of browsers and OS. Selenium Grid gives the possibility to control and manage in a simpler way. It provides a hub that acts as a central point where Selenium sends commands to each node connected to it.

3.2 Docker

Docker is a platform[4] designed to create, deploy, distribute, and run applications using containers. Containers are kept running in isolation on top of the OS kernel. They allow developers to package an application with all its necessary parts, such as libraries and other dependencies and deploy them in a single package. It is an open-source platform in which many people contribute and keep updated so that they can add more features. Although Docker is widely used in many areas (e.g., cloud computing), this is not common for compatibility testing of web-based systems.

4. OUR METHODOLOGY

As above mentioned, and as studied in the cited related works, compatibility testing methods herein aim at evaluating web applications through diverse environments (OS, resources, scale, users, etc.). In order to assess the proper functioning of these applications, the testers need these environments reflecting the real contexts and users utilizing the web applications every day. Traditionally, this involves the use of browsers and devices to present and test all possible scenarios. It may lead to an increase of consumption and cost, factors that researchers and companies are constantly trying to reduce. Besides, they work on decreasing time processes to implement and complete the entire testing process from the generation of scripts, the creation of the necessary contexts and the execution of test cases. Basically, obtaining and processing such environments is often costly in terms of resources and time.

The methodology used in this work is based on the combination of different components and tools that lead to the execution of the compatibility tests as illustrated in Figure 2.

[4] https://www.docker.com

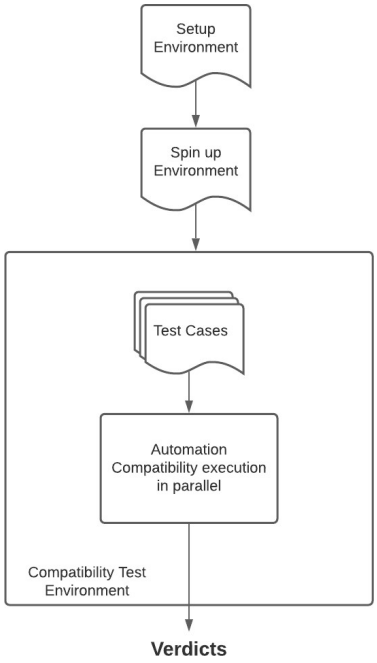


Figure 2. Our framework methodology.

- **Setup environment** This component aims at defining all initial configuration parameters by targeting in particular on the behaviors of our desired framework features. It allows to define the wished OS, environments, browsers, memory, space, number of entities/instances, users, etc. This component plays an important role on how the whole operations will behave and evolve.
- **Spin up environment** This component defines the deployment of our entire framework, using the previous setup configurations. It is responsible for executing the necessary commands so that we can deploy the entire environment from the main device to the different nodes. This obtained architecture will be utilized for the distribution and execution of the test cases.
- **Compatibility test environment** This is the main component where everything is put in place. The test generation element contains all the test cases to be evaluated. It is worth noting that any test case we need to evaluate within the framework can go here at runtime; they can be modified, extended, updated, removed. This element generates the test scripts that will be executed in parallel using Selenium Grid throughout the framework environment. Then, the automated parallel execution element executes each of the test scripts and verify its functionalities on the different deployed Docker nodes.

Note that the whole process of methodology ends up issuing verdicts which are defined on the basis of what is evaluated. In general, according to our testing purpose, obtained verdicts are PASS, FAIL or ERROR. They are depicted in the following sections.

5. EXPERIMENTAL STUDIES

5.1 Framework

Our framework is intended to meet the requirements of providing environments in which various web-based applications can be tested. Based on the above, the components that are part of the experimental framework are the following:

- *Host Device* one of the advantages of this framework is that it can be used on a variety of computers that meet the following minimum requirements:
    - 64bit processor

- Hardware virtualization support
- 4Gb system RAM

In our case, host device (computer) was used with the specification given in the Table 1.

Table 1. Framework host device specification

Operating system	MacOS version 11.2
Processor	2.5GHz Quad-Core Intel Core i7
Memory	16GB 1600 MHz
SSD	512GB

- *Docker* it enables to have different operating systems inside containers, called nodes, as well as to decide which browsers to install, in which nodes and what tests to deploy and where.
- *Browser versions* The Table 2 shows the different versions of browsers used in the implementation of our framework. These browsers were installed in the different nodes. Browser images with the latest version 5 were used, but it is possible to install earlier versions and other images.

Table 2. Web Browsers versions

Firefox	85.0.2
Google Chrome	88.0.4324.150
Opera	74.0.3911.107
Safari	14.1
Brave	88.0.43

- *Docker container images* the docker images that were used as nodes in our framework container the following specifications as show in Table 3.

Table 3. Docker images specifications

Architecture	Amd64
OS	Linux
Size	940829711
Node Port	5555

- *Selenium Grid* (v3.141.59) is the tool used in our framework to distribute our tests through the nodes. By using one of the main features of Selenium Grid, which is its client-server model, it is possible to connect within a Docker container to the hub and the other nodes that are also in containers. As above mentioned, our setup file generates in an automated way our test architectures. Once the architecture is built, it is necessary to use Selenium WebDriver to detect which browser is utilized and consequently to provide the required driver for communication and coordination. The Figure 3 shows the interactions between scripts, WebDrivers and browsers.

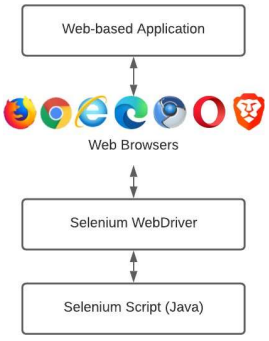


Figure 3. Selenium WebDriver.

One of our testing architecture is illustrated in Figure 4. Besides, our framework enables the creation of any amount of nodes instances and install browsers as we require in our experimental studies.

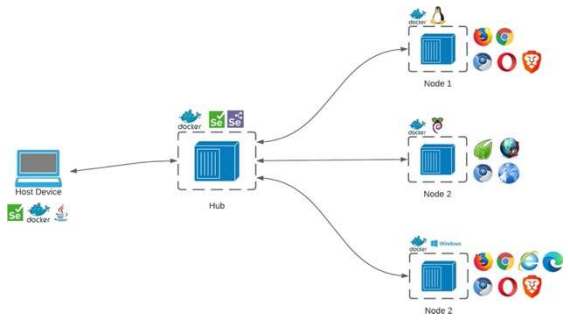


Figure 4. One of our testing architectures.

5.2 Experiments

This section is organized as it follows. First, we introduce the scenarios and test cases considered. This is followed by the framework requirements of the host device. Finally, we present the tuned testbed that runs the different scenarios.

5.2.1 Experimented scenarios and test cases

In our experiments, we propose four different assessed usage scenarios as shown in the Table 4. In these scenarios, we execute the two test cases defined as in the next section.

Table 4. Experimental scenarios

Scenarios	Description
Scenario 1	This scenario has 2 nodes, 1 node with Google Chrome browser, and 1 node Firefox browser.
Scenario 2	This scenario has 10 nodes, 2 nodes for each browser (Google Chrome, Firefox, Opera, Safari, and Brave).
Scenario 3	This scenario has 20 nodes, 4 nodes for each browser (Google Chrome, Firefox, Opera, Safari, and Brave).
Scenario 4	This scenario has 30 nodes, 6 nodes for each browser (Google Chrome, Firefox, Opera, Safari, and Brave).

5.2.2 Our test cases

- *Login Test* this Login test is executed when a user wants to log in to a web-based application as shown in Figure 5. We aim at testing the ability to enter information in the requested fields and thus evaluating the way the layout is displayed, verifying that it is the same in different browsers and operating systems, following policies that are not affected by the default settings that browsers bring and consequently, not show different behaviors noticeable from the requests/responses. Steps to perform for the test:
    - Obtain IP address and por of the node,
    - Identify the browser used on the node,
    - Request the WebDriver specified for this browser,
    - Identify the login fields within the application structure,
    - Enter the default values for each field,
    - Wait for a response from the application.

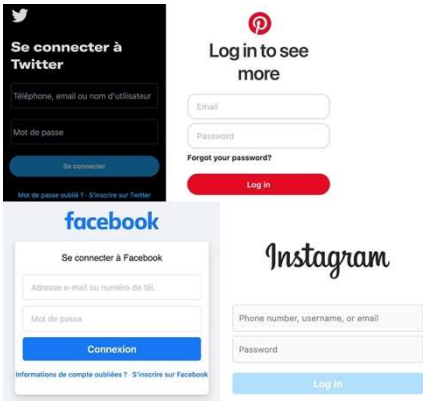


Figure 5. Login Test.

- *Broken links test* the test case named Broken links checks the operations of the internal links that the web-based application may contain as shown in Figure 6. It verifies that there are no broken links or links that cannot be reached, a malfunction of which may affect the overall experience and operation of the application. We can check what kind of error we get when we test a link. Note that we can use the HTTP status code to determine where the problem might be, for example, we get a status code 200 because it is a valid and working link, if we get a status code 400 because the error is on the client side (in the browser or in the operating system where the browser is running) or we get a status code of 500 because the error corresponds to the server providing the application. This is tested in different environments with different configurations to observe the behavior and get results. Steps to perform for the test:
    - Obtain IP address and port of the node,
    - Identify the browser used on the node,
    - Request the WebDriver specified for this browser,
    - Analyze and obtain all the anchor elements of the application,
    - Verify the HTTP response code of each link.

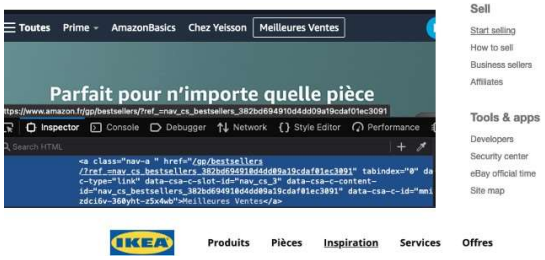


Figure 6. Broken Links Test.

5.3 Framework requirements

To implement and use the framework in our working environment, we did integrate:

- Docker
- Docker Compose
- Docker images
- Java SDK
- Eclipse
- TestNG library
- Selenium Grid

5.3.1 Framework tuning

While using the various elements of our framework, it is necessary to make a special adjustment where the difference is made and we achieve our goal. Therefore, we present below some of the important configurations in order to execute the proposed scenarios.

The entire architecture runs on Docker containers that gives the advantage to process a single configuration file to automate the deployment and nodes configuration, as well as the main node that will contain the Selenium Grid Hub. Next, we present the configuration of the main node, where we can set various variables, for example:

- Services - List of all images and configurations,
- image - Defines which image we will use for the container,
- ports - Ports used in this special format host:container,
- GRID MAX SESSION - This declares how many browsers can run in parallel at time.

```
services:
    hub:
        image: selenium/hubports:
            - "4444:4444"
        environment:
            GRID_MAX_SESSION: 100
            GRID_BROWSER_TIMEOUT: 3000
            GRID_TIMEOUT: 3000
```

In the same file named docker-compose.yml, we insert the nodes configurations with different OS, architectures and browsers. In this configuration, we set values for the Docker image to use if the deployment of our node depends on another, in these cases, all nodes depend on the central/main node (hub), as well as port configurations and how many browsers we want to have in each node. The following is the basic configuration of a node with google chrome browser.

- *container name* - Name to identify the container
- *depends on* - Is the required dependency previous to deploy the node, for example each framework node depends on the hub node
- *NODE MAX SESSIONS* - How many instances of a browser can run over the node
- *NODE MAX INSTANCES* - How many instances of different browsers can run in parallel in the same node

```
environment-chrome:
    image: selenium/node-chrome container_name: web_environment_chrome
    depends_on:
        - hub environment:
        HUB_PORT_4444_TCP_ADDR: hubHUB_PORT_444_TCP_PORT: 4444
        NODE_MAX_SESSIONS: 1
        NODE_MAX_INSTANCES: 1
    volumes:

        - dev/shm:/dev/shmports:
        - "9001:5900"
    links:
        - hub
```

5.3.1 Framework tuning

The Login Test: The script generated for its execution looks like the one illustrated in the Figure 7, where we have code blocks to identify the type of browser used, as well as the main method of testing the process of logging into a web-based application.

```
@Parameters("browser")
pubic void setup(String browser) throws Exception{
    //Check if browser is Firefox
    if(browser.equalsIgnoreCase('firefox')){
        //Create firefox instance
        System.setProperty('webdriver.gecko.driver', path_of_firefox_driver);
        driver = new FirefoxDriver();
    }
    //Check if browser is Chrome
    else if(browser.equalsIgnoreCase('chrome')){
        //Create chrome instance
        System.setProperty('webdriver.chrome.driver', path_of_chrome_driver);
        driver = new ChromeDriver();
    }
    //Check if browser is Opera
    else if(browser.equalsIgnoreCase('opera')){
        //Create opera instance
        System.setProperty('webdriver.opera.driver', path_of_opera_driver);
        driver = new OperaDriver();
    }
    //Check if browser is Safari
    else if(browser.equalsIgnoreCase('safari')){
        //Create safari instance
        System.setProperty('webdriver.safari.driver', path_of_safari_driver);
        driver = new SafariDriver();
    }
    //Check if browser is Edge
    else if(browser.equalsIgnoreCase('edge')){
        //Create edge instance
        System.setProperty('webdriver.edge.driver', path_of_edge_driver);
        driver = new EdgeDriver();
    }
    else {
        throw new Exception('Browser is not correct');
    }

}
```

```
public class LoginTest {

    @Test public void loginTest() {
        driver.get(website);

        webElement username=driver.findElement(By.id('username'));
        webElement password=driver.findElement(By.id('password'));
        webElement login=driver.findElement(By.xpath("//button[text()='Log in']"));

        username.sendKeys('username-example@gmail.com');
        password.sendKeys('password');
        login.click();
        String actualUrl = 'https://twitter.com/login';

        String expectedUrl = driver.getCurrentUrl();
        Assert.assertEquals(expectedUrl, actualUrl);
    }
}
```

Figure 7. Login scenario Java script.

The Broken Links Test: To determine if a web-based application has broken links, we need to follow the steps below:

- Collect the links that are present in the web-based application, these links can be found within the HTML structure with the < a > element, in this element there is the href attribute where we can find the URL that the link redirects to,
- Send a HTTP request to each link,
- Check the HTTP response codes,
- Evaluate the codes and determine whether it works or not.

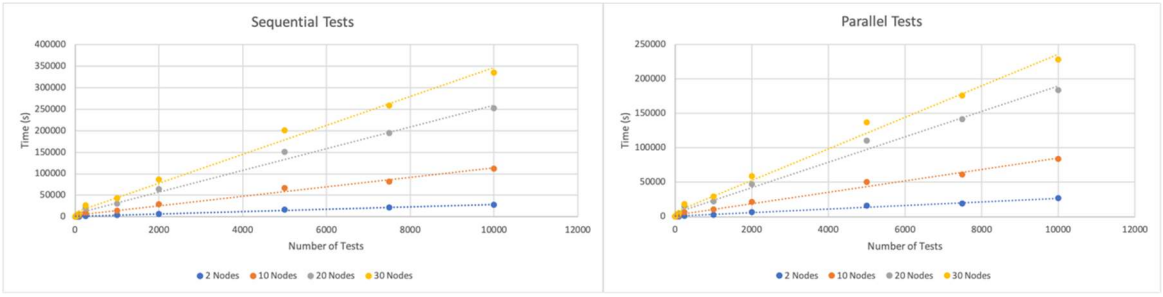
```
//Obtain all the a elements
List<WebElement> links = driver.findElements(By.tagName("a"));
Iterator<WebElement> iterator = links.iterator();
url = iterator.next().getAttribute("href");
link = (HttpURLConnection)(new URL(url).openConnection());link.connect();

httpResponse = link.getResponseCode();if(httpResponse >= 400){
    System.out.println(url
    + " is a broken link");
} else{
    System.out.println(url
    + " is a valid link");
}
```

5.4 Results and discussions

5.4.1 Framework runtime improvement

In order to assess the runtime improvement of our methodology and framework by running tests sequentially and in parallel, we illustrate in the Figure 8 the results obtained after implementing the four scenarios proposed above. In the Figure 8a, we have the results of executing the login tests in the scenarios and we notice a linear behavior where the more nodes we add, the more time it takes to execute each test and complete the tasks. In Figure 8b, we observe the behavior and results that prove our improvement after running the same tests in parallel.

Below we show a more detailed analysis between each scenario run sequentially and in parallel, in which we observe the different slopes of each graph as well as the speedup factor analysis for each scenario. The speedup factor refers to the improvement of the execution speed of a task in two ways, by executing the same tests sequentially and in parallel.



(a) Sequential Tests

(b) Parallel Tests

Figure 8. Execution time Sequential vs Parallel.

The speedup factor is the ratio between the linear regressions for sequential tests and linear regression for parallel tests. It is defined by the equation $speedup = \frac{sequential}{parallel}$

The linear regressions are computed as it follows. Let be X our data points obtained, it is important to mention that in our case, X is a matrix of values and the variable Y is the vector of times. As a result, we get the different coefficients in θ, if we do all these analyses, we get the equation of linear regression in the following way θ = (X ′X) −1X ′Y (X ′ being the transpose of the X matrix).

Below, the execution time of 10 to 10,000 tests in sequential and parallel execution are shown:

- **Scenario 1 – Sequential and parallel**
  - 10 tests execution time sequential: 52s
  - 10,000 test execution time sequential: 27,980s
  - 10 tests execution time parallel: 51s
  - 10,000 tests execution time parallel: 26,600s
  - Speedup factor = 1.067811
- **Scenario 2 – Sequential and parallel in Fig 9**
  - 10 tests execution time sequential: 208s
  - 10,000 tests execution time sequential: 111,920s
  - 10 tests execution time parallel: 156s
  - 10,000 tests execution time parallel: 83,940s
  - Speedup factor = 1.333333
- **Scenario 3 – Sequential and parallel in Fig 10**
  - 10 tests execution time sequential: 468s
  - 10,000 tests execution time sequential: 251,820s
  - 10 tests execution time parallel: 341.64s
  - 10,000 tests execution time parallel: 183,828.6s
  - Speedup factor = 1.369863
- **Scenario 4 – Sequential and parallel**
  - 10 tests execution time sequential: 624s
  - 10,000 tests exec. Time seq.: 335,760s (~4 days)
  - 10 tests execution time parallel (etp): 424.32s
  - 10,000 tests etp: 228,316.8s (~2.5 days)
  - Speedup Factor = 1.470588

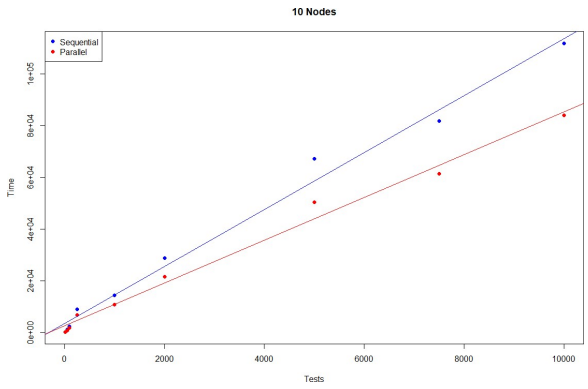


Figure 9. Scenario 2.

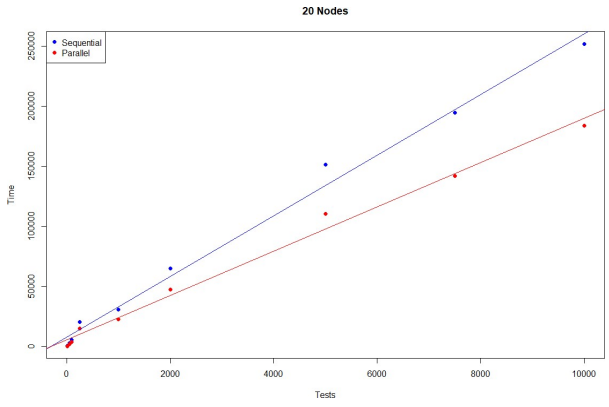


Figure 10. Scenario 3.

5.4.2 Compatibility testing automation

In order to test these scenarios, we define criteria for processing the final tests verdicts on test case 2 (Broken Links) as it follows:

- *PASS* is processed if the test manages to find the correct answer to all the links covered.
- *FAIL* is processed if one of the analyzed links returns an HTTP 400 code as response, it means that it is a broken link.
- We process an *ERROR* if the web page does not contain a link, if a null value is returned in our "links" variable after all elements have been captured, or if it is not possible to access the web app.

These are the results we obtained under the tests in our framework. By running Broken link tests, we get automation for compatibility testing that extends the coverage of our tests and is able to deploy them in multiple environments in parallel. It validates the performance of web-based applications across multiple operating systems, architectures, and browsers.

After deploying our different environments and having prepared the tests to be evaluated, in Table 5 we can observe the global market share of each of the browsers used in this test according to Kinsta[5] and the tests results obtained.

Table 5. Compatibility test against 20 websites

Browser	Market Share	Pass	Fail	Error
Chrome	77.0%	84%	15%	1%
Safari	8.87%	72%	16%	9%
Firefox	7.69%	85%	12%	3%
Opera	2.43%	63%	19%	18%
Brave	0.05%	79%	15%	6%

Note that the websites for this test case were randomly selected from the database of top websites that Alexa[6] provides, and all had properties optimized for most browsers. As part of the results, we can notice that Opera gets low results compared to the rest. Within the popularity of browsers, the most popular are Google Chrome and Firefox, though in terms of usage Safari makes the list, which is why it was added to these tests.

We also took into account the Brave browser, which is an open-source browser that is gaining popularity and offering improvements in security issues among other things. An important detail about Brave is that it is based on Chromium allowing to perform tests using the same Selenium WebDriver used by Google Chrome.

In the results, we notice that Opera is the one with the highest percentage of errors, this can be either because the webDriver did not get all the links on the page or because the web page could not be loaded correctly, remember that Opera is not one of the most used browsers, so there can be compatibility issues.

In Fig. 12, we can observe that we have better results with the Firefox browser, with which we obtained a higher percentage in tests with PASS verdicts, followed by Google Chrome. The behavior of Brave is pretty good, let us take into consideration how it was mentioned before Brave uses the same engine as Chrome, so we can conclude that this is the reason why it has a good coverage. Nevertheless, the overall results of this browser were surprising being the browser with the lowest usage percentage, its results were quite good compared to other competitors like Safari and Opera, which are more popular. In third place we have Safari, this browser has good coverage as well, which is interesting because it is the most used browser by Apple users, therefore, it makes sense that many web-based applications try to be optimized for this browser. Lastly, we have Opera, which impresses us with its performance as it is not that far behind the other browsers.

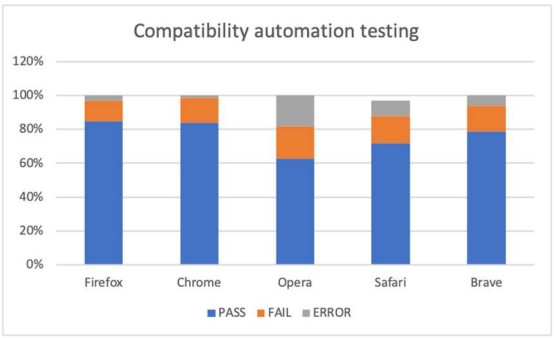


Figure 12. Compatibility Automation Testing results.

[5] https://kinsta.com/browser-market-share/
[6] https://www.alexa.com/topsites

At the end of the tests conducted over 20 different applications, the most popular browsers such as Firefox and Google Chrome perform best, both in PASS and ERROR verdicts. It is interesting to note Brave's results has a lower ERROR rate than Safari, as it is a relatively unused browser. What we also found in these tests is that many browsers have issues with pages that are entirely JavaScript based and how to handle that. But even with that, Safari is not that far away from good performance, as is Opera.

6. CONCLUSION AND PERSPECTIVES

In this paper, we have presented a novel framework for testing the compatibility of web-based applications in parallel. Containers have been processed with Docker to enable the design of a scalable testing architecture.

Thousands of test cases have been executed demonstrating the efficiency of our methodology in the running time improvement and the feasibility. Furthermore, these experiments have highlighted that the compatibility of under-used browser can be very good. Finally, these tests have also shown that applications containing most of its code in JavaScript could provide bad compatibility testing results and then issues.

As perspectives, we tackle a highest number of tested web applications, nodes, OS, browsers versions and test scenarios. Increasing these parameters will assess the scalability of our compatibility testing approach as well as raise existing issues within some web applications.

We also plan to perform a machine learning approach that we already applied in a previous work. We guess that such a technique could guide the testers in the execution of the test cases (providing priorities to specific nodes or test scenarios), in defining new test purposes (based on the obtained observations), and in refining the testing verdicts results (in terms of detailing the results of the obtained FAIL and ERROR).

ACKNOWLEDGEMENT

This research was partially supported by Labex DigiCosme (project ANR-11-LABEX-0045- DIGICOSME) operated by French ANR as part of the program "Investissement d'Avenir" Idex Paris-Saclay (ANR-11-IDEX-0003-02).

REFERENCES

Al-Ahmad, B. and Al Debei, K. (2020). Survey of testing methods for web applications. *IJST*, pp 9(12):1–22.

Bertolino, A., Calabró, A., De Angelis, G., Gortázar, F., Lonetti, F., Maes, M., and Tuñón, G. (2020). Quality- of-experience driven configuration of webrtc services through automated testing. In 2020 IEEE *20th International Conference on Software Quality, Reliability and Security (QRS)*, pp 152–159. IEEE.

García, B., Gallego, M., Gortázar, F., and Jiménez, E. (2017). Webrtc testing: State of the art. In *ICSOFT*

García, B., Gallego, M., Gortázar, F., and Munoz-Organero, M. (2020). A survey of the selenium ecosystem. *Electronics*, 9(7):1067.

García, B., López-Fernández, L., Gallego, M., and Gortázar, F. (2016). Testing framework for webrtc services. In Proceedings of the 9th *EAI International Conference on Mobile Multimedia Communication*s, pp 40–47.

Hayek, M., Farhat, P., Yamout, Y., Ghorra, C., and Haraty, R. A. (2019). Web 2.0 testing tools: A compendium. In 2019 *International Conference on Innovation and Intelligence for Informatics, Computing, and Technologies (3ICT)*,IEEE.

Heinonen, J. (2020). Design and implementation of au- tomated visual regression testing in a large software product.

Ki, T., Park, C. M., Dantu, K., Ko, S. Y., and Ziarek, L. (2019). Mimic: Ui compatibility testing system for android apps. In 2019 *IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp 246–256. IEEE.

Liu, Y., Zhang, T., and Cheng, J. (2019). Survey on crowdbased mobile app testing. In Proceedings of the 2019 *11th International Conference on Machine Learning and Computing,* pp 521–527.

Tanaka, H. (2019). X-brot: Prototyping of compatibility testing tool for web application based on document analysis technology. In *2019 International Conference on Document Analysis and Recognition Workshops*, vol. 7, IEEE.

Villanes, I. K., Endo, A. T., and Dias-Neto, A. C. (2020). Using app attributes to improve mobile device selection for compatibility testing. In *Proceedings of the 5th Brazilian Symposium on Systematic and Automated Software Testing*

WebsiteSetup Editorial (2021). How many websites are there in 2021 websitesetup. https://websitesetup.org/news/how-many-websites-are-there/. Accessed: 2021-02-03.

Yu, J. (2019). Exploration on web testing of website. In *Journal of Physics: Conference Series*, vol 1176, IOP Publishing.