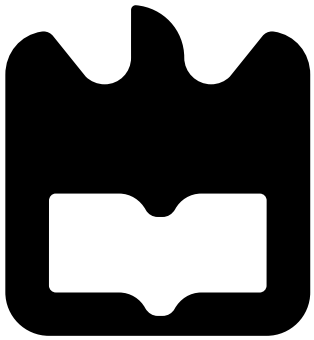
Universidade de Aveiro Departamento de Electrónica, Telecomunicações e Informática,
2017

Sara Cristina Pereira Loureiro

Qualidade de Experiência em Navegação Web

Quality of Experience in Web Browsing



Sara Cristina Pereira Loureiro

Qualidade de Experiência em Navegação Web

Quality of Experience in Web Browsing

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica do Doutor João Paulo Silva Barraca, Professor Auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro.

o júri / the jury

presidente / president

Professor Doutor Joaquim João Estrela Ribeiro Silvestre Madeira
Professor Auxiliar da Universidade de Aveiro

vogais / examiners committee

Professor Doutor Óscar Emanuel Chaves Mealha
Professor Associado com Agregação da Universidade de Aveiro

Professor Doutor João Paulo Silva Barraca
Professor Auxiliar da Universidade de Aveiro

agradecimentos/
acknowledgments

Ao meu orientador, o Professor Doutor João Paulo Barraca, pelo apoio e disponibilidade.
À equipa da Altice Labs que me acolheu no desenvolvimento desta dissertação de mestrado, o meu agradecimento pela oportunidade.
Aos meus amigos, pelo apoio e companhia ao longo desta caminhada.
Em especial, à minha família, a quem dedico este trabalho, por me terem proporcionado todas as condições para a concretização deste objectivo e estarem sempre disponíveis e presentes em todos os momentos, o meu muito obrigada!

palavras-chave

Qualidade de Experiência, Qualidade de Serviço, Automação de browsers, Navegação Web headless

resumo

Esta dissertação de Mestrado tem por objectivo o estudo e o desenvolvimento de uma ferramenta de aferição de métricas de Qualidade de Experiência, no contexto da navegação Web, de modo a percecionar, da forma mais real possível, a experiência dos utilizadores comuns na navegação em páginas Web arbitrárias. A obtenção de métricas que permitam determinar a Qualidade de Experiência e de Serviço são uma ferramenta essencial para as operadores de telecomunicações, de forma a diagnosticar o nível de qualidade dos serviços prestados e, consequentemente, perceber o grau de satisfação dos clientes. Neste contexto, a solução desenvolvida foi uma ferramenta que atua de forma independente e efetua testes que permitem aferir a experiência a que um utilizador normal estaria sujeito face à navegação numa página Web, num browser real. Assim, a aplicação desenvolvida é capaz de: extrair intervalos temporais para inferir a qualidade da experiência, informações acerca da navegação e do estado da página Web, bem como dos seus componentes. Para além disso, permite configurar cenários de interação simples ou complexos com a página, tirar capturas de ecrã em vários momentos durante o correr do programa e ainda, permite descarregar todos os ficheiros que compõem a página Web, que foram carregados. Esta solução foi sujeita a testes de averiguação do desempenho e da confiabilidade dos valores devolvidos (métricas) pela mesma, quando comparados com uma navegação real.

keywords Quality of Experience, Quality of Service, Browser Automation, Headless Web Navigation

abstract This Master dissertation has the aim of study, and further develop, a tool that allows to gather Quality of Experience metrics, in the Web browsing context, in order to perceive, as close to reality as possible, the experience that real users would have during the navigation in webpages. The gathering of Quality of Experience and Service metrics are essential for telecom operators, in order to diagnose the quality of services and, consequently, perceive the level of satisfaction of its clients. In this context, the solution was an application that acts independently and performs tests, that allow to infer the experience that a real user would face during the access to a webpage, through a real browser. The solution is able to: extract time metrics, information related with the Web navigation, the status of the webpage and its resources. Furthermore, it allows to configure simple or more complex interaction test scenarios with the webpage, take screen captures during the test execution and even download all the files belonging to the webpage that were loaded. The solution was tested in order to evaluate its performance and returned values (metrics) reliability, comparing with a real navigation values.

Contents

List of Figures

Listings

List of Tables

Acronyms

API	Application Programming Interface
ARP	Address Resolution Protocol
CDN	Content Delivery Network
CPE	Customer Premises Equipment
CPU	Central Processing Unit
CSP	Communications/Content Service Provider
CSS	Cascading Style Sheets
DNS	Domain Name System
DOM	Document Object Model
FTP	File Transfer Protocol
GB	GigaByte
GUI	Graphical User Interface
HAR	HTTP Archive
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
HTTPS	HyperText Transfer Protocol Secure
ICMP	Internet Control Message Protocol
IE	Microsoft Internet Explorer
IEEE	Institute of Electrical and Electronics Engineers
IETF	Internet Engineering Task Force

IGMP Internet Group Management Protocol

IP Internet Protocol

ISP Internet Service Providers

ITU International Telecommunication Union

JS JavaScript

JSON JavaScript Object Notation Syntax

KPI Key Performance Indicators

MDN Mozilla Developer Network

MOS Mean Opinion Score

OS Operating System

OSI Open Systems Interconnection

OTT Over-The-Top

PHP Hypertext Preprocessor

POP Points Of Presence

QoE Quality of Experience

QoS Quality of Service

RAM Random Access Memory

RIP Routing Information Protocol

RTT Round-trip delay time

SNMP Simple Network Management Protocol

SMTP Simple Mail Transfer Protocol

SOCKS Socket Secure

SSL Secure Sockets Layer

SWF Shockwave Flash File

TCP Transmission Control Protocol

TCP/IP Transmission Control Protocol/Internet Protocol

TLS	Transport Layer Security
TTFB	Time To The First Byte
TTFP	Time To The First Paint
UDP	User Datagram Protocol
URL	Uniform Resource Locator
URI	Uniform Resource Identifier
UX	User Experience
VNC	Virtual Network Computing
VOD	Video On Demand
VoIP	Voice over IP
W3C	World Wide Web Consortium
WWW	World Wide Web
XML	Extensible Markup Language
Xvfb	X Virtual FrameBuffer
Xvnc	X-based Virtual Network Computing Server

1

Introduction

Motivation

Nowadays, with the fast evolution of the Web, increased contents available, new technologies involved and faster retrieval of information due to speedy and reliable networks and, also, the rising of solutions to bring data closer to the end-users - the use of Content Delivery Network (CDN) and Cloud CDN solutions -, raised the exigency and expectations of the users about their Internet service and navigation experience. Thereby, in order to provide the best Web navigation experience to their clients, the Communications/Content Service Provider (CSP) - commonly, the telecom operators - need to keep improving the Quality of Service (QoS) delivered to their costumers, by evolving their networks and bet on the investigation of new solutions of software and hardware, to maintain the clients and markets already acquired, but also, conquer new ones. Consequently, it urged the need to create solutions that allow the possibility of a more reliable perception of the end-User Experience (UX) in terms of their navigation experience, known as the Quality of Experience (QoE) in Web browsing.

The QoE aims are to reveal the users experience on what they see during their navigation time, to better grant the CSP clients satisfaction about the service they applied to and reduce the *churn* rate, that is the amount of clients who leave the subscription of CSP services.

Objectives

In this work, it will be addressed the perceived UX quality during Web browsing - Web Experience -, considering the user-agent type, the accessed webpage and further manipulation

(for example, login into a website to determine the time needed to the return of the action result, that is, a visible answer to the user), within a maximum time interval threshold (*timeout*) considered reasonable to receive the response for the test specific allocated tasks. The main aim is to perceive what would be the end-user QoE in a real navigation from the returned results of the browsing simulation test. The results returned can be further analyzed to measure the experience quality and to infer what would a real user feel about the browsing experience.

The aim of this work is to provide an application that works independently and that mimics a real end-user behavior, while navigating on the Web on a real browser, in order to acquire metrics that can relate its experience.

In that way, the solution developed is capable of extracting temporal metrics and information about the webpage, such as its code status and loaded components. Also, it can take screen captures in several temporal instants during the program running time. It is possible, as well, to configure testing scenarios for interactions with the webpage, using three different scenario possibilities: no interactions configured, one interaction (simple scenarios) and more than one interaction (complex scenario).

Furthermore, this application was aimed to be included in a probe that would trigger tests and provide the informations extracted to the support unit of the *Altice Labs ArQoS* team. As such, this application was developed in the scope of *Altice Labs ArQoS NG* project, which aim was to develop this application for future integration on their *ArQoS NG* probes.

Contributions

This dissertation is included in the scope of the *Altice Labs* project, the *ArQoS*.

The *ArQos* is a performance monitoring system of services and telecommunication networks. This system is composed by probes and a management system, which communicate between each other. The probes (vide Figure 1.1), fix or mobile, can be configured to perform several tasks and, further, send the results to the management system. The management system is the unit that schedules the tasks requests and performs the results analysis, besides other functionalities, such as, for example, alarm management or reporting.

Figure 1.1: *ArQoS* Probe.

In this work was developed a tool to access webpages, through real browsers, in order to gather metrics and perform interactions in an automated way.

The need to automate a web navigation urged with the need to evolve from the *ArQoS* existing solution, which focus is on the extraction of QoS metrics. Moreover, the existing solution does not interact with the webpage in the same way as a real browser. As long as it uses command line tools, the *wget* and *libcurl*, to perform the Uniform Resource Locator (URL) request, the navigation does not correspond to a real experience. The main difference is on the webpage loading, as the *wget* loads the totality of the webpage at once, which does not correspond to a real user-scenario. By contrast, the real browsers are able to partially load the webpages, and the remaining as the users scrolls on the page.

Since that, there was the need to have a solution that could approximate to a real user navigation, by using real browsers, in order to gather QoE-related metrics and also perform interactions, for example: login, logout and scroll actions. Furthermore, the new solution should be compatible with devices with no graphical capabilities, as it must run on the *ArQoS* devices, the *ArQoS NG Probes*. These probes run *ArQoS* tasks, which are scheduled by the the management system team. The probes run the tasks that were scheduled and, afterwards, send the response to the management system, where the response is collected and analyzed. The solution developed throughout this dissertation is, also, a task that must run on these probes.

Hence, the solution proposed and developed, throughout this dissertation work, allows the access to webpages, to gather QoE-related metrics, take screenshots of the scenario at certain instants, and execute user-like interactions, through an inputted configuration. Afterwards, this solution returns the results, which can be further analyzed.

Structure

This master dissertation is divided in six chapters, below there is the list and the description of each of them:

Chapter 1 - Introduction: In this chapter are presented the motivation that triggered this project and, also, the objectives to be fulfilled.

Chapter 2 - State-Of-The-Art: This chapter aim was to investigate the topics directly connected with this theme and that were important in the construction of the *SmartBrowsing* solution.

Chapter 3 - *SmartBrowsing*: Solution: In this chapter it is described the solution, along with the use cases, flow and architecture diagram.

Chapter 4 - Implementation: In this chapter it is described the way the application solution is implemented.

Chapter 5 - Evaluation And Analysis: In this chapter the solution was tested and the results analyzed, in order to prove that the solution is adequate to the task envisioned.

Chapter 6 - Conclusion: The last chapter has the main conclusions taken and the future work.

2

State-Of-The-Art

In this chapter it will be approached the themes related with this master dissertation aim.

The themes will be addressed by order, that is, from the most core concepts to the most corner related others. Also, it will be described tools and solutions that deal with the QoE evaluation.

Quality of Experience (QoE)

The user experience perception is an aim that both websites developers and content service providers (CSPs) want to acquire, as such may help to improve the contents availability and access speed to the service end-user. Consequently, both website owners and CSPs can maintain their user targets satisfied and loyal to their services or contents. Therefore, the concept of Quality of Experience urged due to that need of knowing what is the user acceptance in relation to a target product or service (for example: the users feel boredom or satisfaction or if they quit after an awaiting time) and translate that sensations into data that can be treated and analyzed, in order to trace an ideal profile of the users requirements to better fulfill their expectations about the product or service.

The concept of QoE is still, nowadays, a fresh and important theme as there is no final answer to the question of how to perform its measurement. Due to the enormous variety of factors associated to the user experience and also, the difficulty to obtain exact values to define the real perception of what is being sensed, the objective is to get as closer as possible to the best approximation of what seems to be a good performance and behavior to the human users.

The Quality of Experience concerns about what is the consumer impression about a prod-

uct or a service. Thus, it seeks what distinguish a good experience from a bad one and which are the factors that affect it.

In this field, there are multiple research papers and experiments that relate what can influence the QoE and which are the metrics that seem to have a bigger impact on the final QoE. Some works [1] separate the QoE influences in terms of:

- Human Factors (extrinsic and intrinsic);

- Context Factors (the environment of use);

- System Factors (the characteristics intrinsic to the service or product being analyzed).

The Human Factors relate to the users characteristics, in terms of gender, age, culture, personality, and others intrinsically related to the users mind. This influencing factors forms the particularities set of each individual [1].

The Context Factors relate to the environment characteristics, such as, the experiment location, the day time, the space/environment physical conditions [1] and, also, the task type and its imperativeness for the end-user [2].

Lastly, the System Factors are connected to the specific characteristics of the target product or service. An ITU Recommendation [2] defines some sub-types of system influencing factors for web-browsing QoE: "server-related" factors, that are linked to the response time of the system components, like the Central Processing Unit (CPU), Operating System (OS), memory or software in use, but also the server link capacity to the Internet; "Content-related" factors, that relates to the content available on the World Wide Web (WWW), such as text, media content - images, audio and video -, HyperText Markup Language (HTML), JavaScript (JS), Cascading Style Sheets (CSS) files and others. Otherwise, the page organization and structure affects widely the time of page load, due to aspect such as, the amount of objects and elements, its length and the order they are loaded; "Delivery network" that consider factors as the network capacity, its contribution to the transaction time (i.e., "is the sum of all the component times for a given transaction type, where the number and direction of exchanges and device configurations are specified" [3]) and the presence of elements along the network that allow caching of requests, resulting in a reduced server response time, consequently reducing the Round-trip delay time (RTT) (that is, the time it takes to send and process a request until the respective acknowledge is received); "Client influencing factors" that aggregate the "resource (webpage) loading procedure", "processing power"/capacity, browser type, "TCP/IP stack and configuration" and the OS. "All of these IF's (influencing factors) impact the user perceived performance of the webpage display when requested by the user" [2].

The Figure 2.1 shows an illustration of the location where these factors are applied in an access to a webpage, immediately after an end-user request:

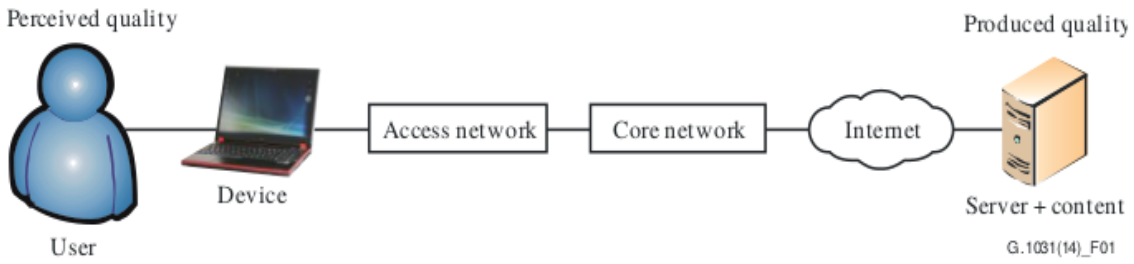


Figure 2.1: "Delivery chain for a typical webpage."[2].

As such, the Figure 2.1 shows that the user is the agent who perceives the quality of what is shown on its device. The access network, core networks and Internet allow the return of QoS metrics, which can impact on the quality observed by the user (this topic will be further addressed on the subsection 2.1.2). Finally, the server contains the data content to be requested and sent to the destination, which by itself has characteristics that can also influence on the quality of the data received by the user.

And, hence, the QoE is commonly defined as "the overall acceptability of an application or service, as perceived subjectively by the end-user"[4], which are influenced by the factors referenced above.

This strategy of differentiating the factors in three different aspects is an International Telecommunication Union (ITU) Recommendation that several researchers use as a starting point for their paper works, about this thematic. This entity is an agency specialized in the fields of telecommunications, information and communication technologies (ICTs) [2], whose work aim is to provide Recommendations for technical, operating and tariff questions, in order to provide standardized rules worldwide.

However, all the concept definitions are complex to be translated into standards, due to its subjectivity, as they rely on human characteristics that are composite and difficult to measure and complicated of being read automatically by a system.

Some papers about the QoE, besides addressing the concept along with the factors which influence it, also state features of QoE, that characterize the experience for of each individual using a service and how that characteristic influences its quality. Accordingly with the *Qualinet White Paper* about QoE [1], there are four levels of features: "Direct perception" based on "sensory" information obtained during the experience, for example, image quality or synchronism of video image with sound; "Interaction" that comprehends the interaction between human-to-human and human-to-machine, like responsiveness to actions or efficiency of the communication; "Usage situation", that is, in under what conditions does the service is used, for example, the accessibility or stability of the service; And the "service" level is how this one is sensed aesthetically, its "usability, joy and ease of use". The same work [1], also states that "in applications like web browsing or HTTP-based media streaming, the percep-

tion of waiting times before streaming or webpage loadings is an important feature to measure and monitor QoE of such systems", which means that the time of page loading and of its components are an important metric to measure the QoE, as it is a "direct perception" for the end-users. This is a particularly important fact for this work, as this states an important metric to measure the QoE.

This question, the QoE measurement, is applied to many areas, but with more focus on the following:

- **Video and audio quality**, whose QoS measurement can be measured with certainty, as its metrics are well defined and can be determined by calculate time metrics acquired on the network response to actions. However, the QoE is more challenging to be known, as it depends on the end-user perception of the video or audio experience. But, in fact, QoE measurement is important to increase the end-user satisfaction, which is advantageous for the media providers, that want more visualizations, in order to get more profit from it. [5]

- **Gaming**, is an area with particular interest to measure the QoE, as the end-user experience is the most important, in order to avoid loosing players due to a poor experience during the action. This is particularly important to interactive, online or multiplayer games, that depend on the game responsiveness. Game producers are very interested in having tools that can help them perceive what their users are sensing, so they can improve their games quality, as well as, keep the users attached to the game or acquire new ones, due to a good game reputation. Since this will be translated into revenues increase to the game enterprise. [5]

- **Web Browsing**, is an area to which the QoE measurement is important both to telecommunication operators and webpage developers, in order to perceive how the service provided is sensed by the end-user, the responsiveness observed or how appealing is the webpage to him. This is critical to these stakeholders, as it may entail services termination of contract, leading to a churn rate increase, otherwise, it can cause a decrease in the amount of time the users spend on a webpage, reducing the number of visitors after some time and, consequently, lowering the webpage owners profits. The QoE measurement, in this area, depends on several factors that match the ones mentioned above (Human, Context and System), as the web browsing experience depends on the device type used (Random Access Memory (RAM) available, CPU model and other characteristics), the type of Internet connectivity, the end-users intrinsic characteristics that will imply in the way they sense the experience. But also, the network connectivity and the webpage host server characteristics can imply on the QoE, although they are more associated with QoS measurement. And that is the key problem of retrieving a QoE value from a navigation experience. [5]

QoE Metrics

In this subsection, it will be addressed some metrics that relate to the topic of QoE, with focus on the Web, and that are known to be helpful in the QoE measurement, considering some paper works in this area.

As it was referred before, QoE is an important thematic as the WWW has been increasing its number of users, which, as a result, opens possibilities of business for website owners. As such, it is important to keep the focus on the end-users satisfaction, in order to get in return the highest profit possible, from the content they publish.

Nowadays, web navigators are becoming more demanding about the experience they have during the time they navigate on the web. This fact, introduces more complexity to the problem of measurement, as there are more factors involved and with more relation between each others. Moreover, the factors, as mentioned above, are also, in its majority, subjective, because they involve intrinsic characteristics of the users, which are diverse and complex to measure, as well as the environmental factors, which are very diversified too.

However all this complexity, there are already some studies and works that try to state some metrics to acquire the QoE.

- *Page Load Time*: that is the average time needed to completely load a webpage;
- *Time to the First Byte*: that is the time interval needed to receive the first byte response from the server, immediately after a request;
- *Time to the First Paint*: the average time to obtain the first visible painting on the webpage.
- *Time to the Above The Fold / Time to The Full Screen*: this metric was proposed by *Google* and corresponds to the time it takes to the webpage being visible and loaded on the screen of the user, even if in its totality it is still processing and loading resources. This metric states the importance of having the first part of the webpage ready to be showed to the user and giving the idea of a completely loaded and responsive webpage, even if the invisible part for the user is still not complete.[6]
- *Mean Opinion Score (MOS)*: arithmetic mean of the scores given by each individual that participated on the classification test. The scores are between 1, that represents the worst score, and 5, that is the best score possible.

  The *MOS* is a common test to obtain the real end-user perception of a service, generally, it is associated with tests on telephony systems.

The MOS is calculated as the mean of individual scores, given by real persons, in a range scale of 1 (one - the worst score) to 5 (five - the best score):

$$\text{MOS} = \frac{\sum_{i=1}^{N}(Xi)}{N}$$

, where Xi is an individual score value (within 1 to 5) and N is the total number of individuals scores. [7]

The scores given by each individual classify the service in the perspective of the user experience, enabling the comprehension of how the system is seen by the overall users. The Mean Opinion Score value is calculated as the mean of all answers of each individual, which can result in a deceiving final score, due to the opinions distribution. Because the scores given can be very sparse, mixing very good classifications with very bad ones, the average final classification may not be representative of all user opinions. Besides that, additional calculations would help to understand if the resulting score is more or less trustworthy, like the scores standard deviation, that would detect the amount of opinions dispersion.

Although it is not very common to use this measure in other areas rather than in telecommunication systems, the MOS can be useful to obtain a better understanding of how the target users see a product or service. The users, that participate in the MOS tests, contact directly with the system or product to be classified and score it accordingly with the experience. This is useful to understand if the service or product serve well the target audience.

In the web, the MOS can be useful to classify webpages, according with the responsiveness that the page users experience. This measure is an important factor that influence the websites quality of experience classification, since it is the direct contact between the end-users and the webpage being evaluated, i.e. the scores relies on the persons who use it, which provide a more realistic perception of the QoE.

Comparison between QoE and Quality of Service (QoS)

Since the emergence of the QoE concept that it is correlated with the topic of Quality of Service (QoS). The QoS concept is defined in standards that are specific to products or services that obey to defined performance rules. [8]

Some of the most used metrics to calculate the QoS are: the *response time*, that is defined as the time between a request and the beginning of its response; the *latency*, that is the time between the request sending and the response receiving; *jitter*, that is the variation of the latency on a packet flow being transmitted, showing if there is more delay in the transmission of some packets in relation to others from the same flow; *throughput*, that is the rate of packets delivered per time unit; the *packet loss*, that is the amount of packets lost during a transmission. [8]

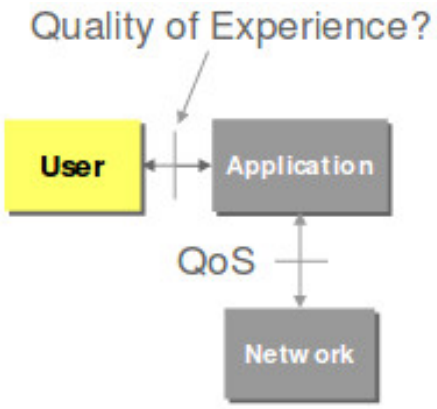


Figure 2.2: QoE versus QoS. [9]

As the Figure 2.2 depicts, the QoS metrics are related with the network performance, since the QoE is more related with the way the user perceives what is being seen on the application used. But, since the application performance depends on the network, the network factors themselves impact on the user-perceived QoE. And that is the reason why the QoS metrics are also important on the measure of the QoE.

Internet

The Internet is an international network of networks that connects computational devices all over the world, via fiber, copper, radio or satellite means, using communication protocols.

The Internet had its start in an experiment on connecting computer networks. This project was leaded by an agency belonging to the United States of America (USA) Department of Defense, the Advanced Research Projects Agency (ARPA, today known as DARPA) along with other institutions, like universities or private companies, that helped raise the Internet. This experiment started with a network compound by four nodes and using the packet-switching technology, in order to communicate in-between the nodes. The packet-switching

technology relies on the split of chunks of data (packets) to be sent to the destination where they are reassembled, in case the packets arrive out of order, as the packets could choose any path available to reach the destination. The network gets redundant and reliable, because, in case of failure of one or more nodes or links, the packets can reach the destination, while there are paths available to reach it. [10]

This investigation project had the aim to explore the paths to achieve a secure and reliable way of communication. Each node was known as an IMP (Interface Message Processor), representing the first gateways, that then evolved to the nowadays routers. However, questions of compatibility, in terms of the communication signals switched between devices with different characteristics (mainly, different operating systems), were raised and the solution came with the emergence of protocols. The protocol term can be defined as the rules that define the way the communication should occur, but also, the abstraction needed to hide the differences between the systems that communicate. The first protocol used was the *Network Control Protocol* (NCP), that at the time allowed the users to remotely access machines, to transport files between computers and the switch of electronic mails. But, years later, the NCP protocol was discontinued and ARPANET and all attached subnetworks started to use TCP/IP protocols. [11]

After the online release, the ARPANET started to grow with the join of organizations private networks, like government and universities, to the initial one. The ARPANET had a policy that prohibited its commercial use, known as the *Acceptable Use Policy* (AUP). Due to its amount of traffic, the ARPANET become overloaded and this problem triggered the need of a response, that came from the National Science Foundation (NSF), with the NSFNET network solution. Meanwhile, the ARPANET was deactivated. The NSFNET was made of regional and peer networks connected to the its core backbone. After that, the NSFNET started to connect educational institutions and research campuses to the regional networks connected to its backbone.[10]

But, since there were other parties interested in using this network for commercial communications, Internet Service Providers (ISP) started to appear, in order to give a response to these requests. Nowadays, the Internet is a set of Points Of Presence (POP) located on several regions that allow users to have connection to the Internet through their region network service providers. [10]

In fact, with the increasing number of Internet users and amount of contents there is a growing concern about the contents' quality, which has been changing the way the Internet provides them to the end-users. Those changes are, for example, the integration of new elements that increase the contents' delivery speed, such as the case of CDNs.

Transmission Control Protocol/Internet Protocol (TCP/IP)

The TCP/IP, also called Internet Protocol Suite, is a set of protocols that rule the communications in the Internet, with the aim to allow the communication between networks with different characteristics. This protocols set is divided in four layers, listed below [12]:

- **Application**: This is the top layer on the stack of protocols of the TCP/IP model. And the protocols belonging to it provide the interface needed to communicate between hosts. Some of this layer protocols are: HTTP, File Transfer Protocol (FTP), Simple Mail Transfer Protocol (SMTP), Telnet, DNS, Routing Information Protocol (RIP) and Simple Network Management Protocol (SNMP). These protocols operate over the protocols belonging to the layer below to it, the transport layer. They allow the user to perform connections to remote hosts, packet routing, information transfers, by using services over the lower layers protocols. [13]

- **Transport**: The main protocols of this layer are the TCP and User Datagram Protocol (UDP). Since that, this layer has the responsibility to transport data either on a reliable way, by using TCP connections, or a non-reliable way, using UDP connections. [12]

- **Internet**: In this layer are treated the addressing, routing and packaging. The main protocols belonging to this layer are Internet Protocol (IP), Address Resolution Protocol (ARP), Internet Control Message Protocol (ICMP) and Internet Group Management Protocol (IGMP).[12]

- **Network**: This layer does the management of the TCP/IP packets that circulate on the network and it is responsible for the incomings and outgoing packets [12].

There is another layer model, the Open Systems Interconnection (OSI) Model, that is compound of seven layers, as shown in Figure 2.3: the application, presentation, session, transport, network, data-link, physical layers. These layers have a correspondence to the TCP/IP model, although it is not a direct correspondence, that is, one layer from the TCP/IP model has common protocols with one or more layers of the OSI model.

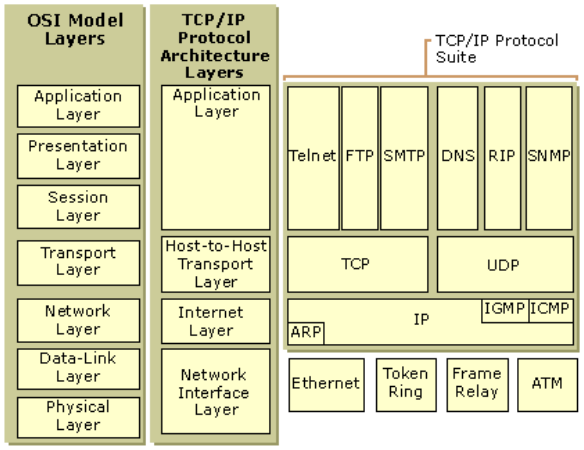


Figure 2.3: TCP/IP Protocol Architecture. Source: [12]

These protocols are the means of communication and delivery of data on the Internet, consequently, they are related with the services' quality provided to the users and, further, their experience. For example, the time took on the lookup for the DNS is an indicative metric of the quality of service.

World Wide Web Concepts

The WWW, also known as Web, was born by the hands of an British scientist, Tim Berners-Lee, in the early nineties. The WWW is part of the Internet, as it is a service on the top of its infrastructure, that uses the HTTP protocol to spread data and make it available. By using an URL, on a browser, it is possible to access webpages available on the Web, as it describes the location of their documents (for example: HTML documents, CSS, JS or multimedia). The URL is a subset of Uniform Resource Identifier (URI)s, that identifies and describes the location of a resource [14].

The technologies that rely on the base of WWW are: the HTML, that describes the content of the webpage; Languages as JS and CSS are part of the majority of the webpages structure and are related with the page look and interaction (respectively); The HTTP protocol, that allows the retrieval of resources across the web.

The Figure 2.4 shows the languages that are used on the Web and the protocols under which the Web is supported by.

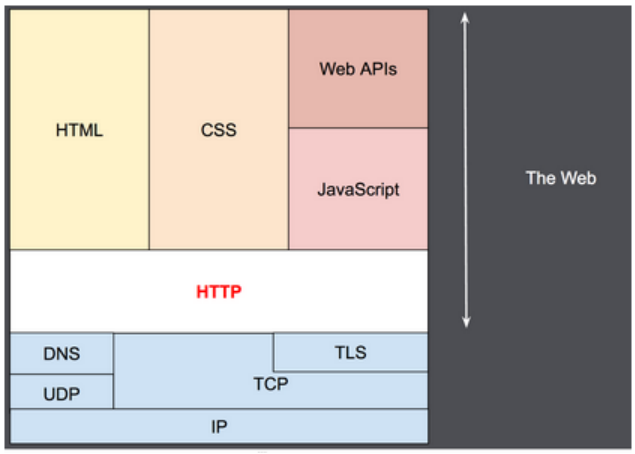


Figure 2.4: Diagram of the technologies that compound the Web.[15]

The IP, as the name indicates is the protocol under which all the Internet is sustained, and makes possible the communication between hosts connected to the Internet, due to the attribution of IP addresses that will identify them.

The DNS performs the match between IP addresses and domain names (strings), in order to be easier for humans to remember than the correspondent IP addresses.

The UDP and TCP are used to transport data between systems, although they are different from each other, as the UDP is a stateless protocol, i.e., the exchanged messages between hosts does not imply the confirmation that the messages were receipt successfully [16], and so it is a non-reliable protocol. In contrast, the TCP is a stateful protocol and so a reliable one, as it implies the confirmation that each message sent was receipt properly, i.e., the message was not corrupted nor lost. Usually, this confirmation is made by the reception of an acknowledge message type on the sender side, from the message receiver [17]. In brief, both these protocols are used to transport data across the Internet, belonging to the transport-layer.

The Transport Layer Security (TLS) is a protocol that enables the transport of messages in a secure way, as it aims to provide integrity and privacy to the data switched between entities [18].

Following, it will be addressed with more detail, the HTTP protocol due to its importance to the Web. Further, the concepts of webpage and the way it is organized and structured will be studied.

Hypertext Transfer Protocol

The HTTP protocol is a client-server protocol, as its usage relies on the switching of messages between servers and the client requester. As such, the client, known as the user-agent, performs *requests* that are sent to a server that will provide the correspondent *response*

message that answers to the request made. Furthermore, this protocol is also a stateless protocol, as none of the parts involved wait for a response of acknowledge after each message sent. Stated that, the exchanged messages between client-server are sent over the Internet by using TCP or TLS over TCP connections, or other identical protocol that serves the same purpose, in order to keep the data secure during its transmission across the Internet and also, grant the reliability of the communication [15]. The TCP connections are stateful, that is, the requester and the sender agents keep a history of the messages switched, to prevent loss of data and make the communication reliable, what would not be possible if they were not used due to the reasons explained above.

The HTTP protocol is used for "distributed, collaborative, hypermedia information systems" at the application level, which is in use since the year of 1990 [19]. Since then, there were released some versions that improved the amount of features available. The first HTTP was HTTP/0.9 that allowed the transfer of raw information data across the Internet. This was followed by the HTTP/1.0 version, that allowed to pass meta-information about the data being transferred. Still, this version was not sufficient for the needs of proxies, caching, persistent connections an virtual hosts usage. Then, the version HTTP/1.1 arrived, in order to supply those needs.[19] More recently, the birth of the HTTP/2.0 represents the optimization of the previous version, with characteristics such as: the decrease of the latency effects, with the adding of header field compression and the allowance of single-connection multiple transactions. Although that, the HTTP/1.1 is still the most widely used. [20]

In essence, this protocol aim is the fetch of resources, such as HTML, CSS, JS, images or videos, in order to build an entire document that form the webpage [15]. The resources, that compose the webpage document, to be fetched can be located in different places, known as servers, on the Internet, as it is illustrated in Figure 2.5. In this Figure 2.5, the webpage document is compound by several files, some of them located in other servers, different from the server that contains webpage base files, such as the HTML and CSS. The server (represented on the Figure by the name of "Web server") hostages the base webpage resources: the style sheet (CSS file, represented on the Figure by the name of "layout.css"), the image (represented on the Figure by the name of "image.png") and the HTML file (represented on the Figure by the name of "page.html"). The other resources: the video (represented on the Figure by the name of "video.mp4") is located on a different server (represented on the Figure by the name of "Video server"), as well as the the "Ads" image (represented on the Figure by the name of "ads.jpg"), that is located on another server (represented on the Figure by the name of "Ads server").

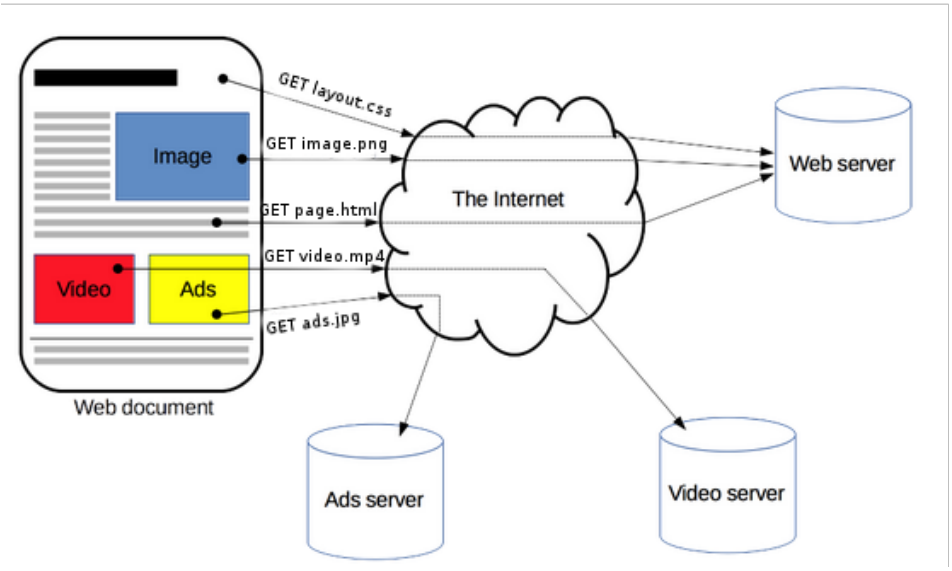


Figure 2.5: Fetching of documents located in different servers.[15]

The HTTP response of a resource return provides informations like a status code. There are several types of status codes, that are represented by a number, which the first number indicates to which category it belongs to.[13] The categories of status codes are the following:

- **Informational - 1xx**: the codes belonging to this category have the purpose to inform about intermediate stages before the final response.
- **Successful - 2xx**: the codes from this category indicate success to the response and can also provide more information about it, like the type or amount of content transferred (for example: "206 Partial Content").
- **Redirection - 3xx**: These codes provide indications to the user-agent, in terms of the actions in need to be performed (for example: "305 Use Proxy").
- **Client error - 4xx**: these codes, as the name of the category states, provide information about client-side errors, for example, "400 Bad Request".
- **Server error - 5xx**: this category provide the codes to inform about server errors, for example, the "500 Internal Server Error".[13]

In the WWW, it is common to observe websites that use the HTTP over TLS, the HyperText Transfer Protocol Secure (HTTPS), in order to provide a secure navigation on the WWW, by transferring the traffic through a different server port than the one used to forward the HTTP traffic. For that, the client triggers the process of the TLS handshake with the server, and after this process is finished, it is performed the HTTP request, under a secure session.[21]

Webpage

A webpage is a document that can be displayed on web browsers, and the collection of webpages that are interconnected is known by the term of website. The host of a website is denoted as a web server. These concepts are important as they are elements that are part of the WWW.

There are two main types of webpages: static and dynamic. The dynamic pages can be separated, for matters of definition, in two different processing modes, the *client-side scripting* and the *server-side scripting*. Although, nowadays, dynamic pages are the most common on the Web, as static webpages are on the verge of extinction, due to the fact of only return static content and use few technologies, which make them less attractive to the end-user (as it will be further explained in this section).

A static webpage, also named flat page or stationary page, always presents the same content, that is, the webpage shows to every user the stored content on its web server.

The client (browser of the user computer) performs a *HTTP Request* of the intended webpage URL, and in return the web server, that hosts the webpage, returns the *HTTP Response* with the corresponding HTML document of the requested URL. This message switching process is illustrated on the Figure 2.6.

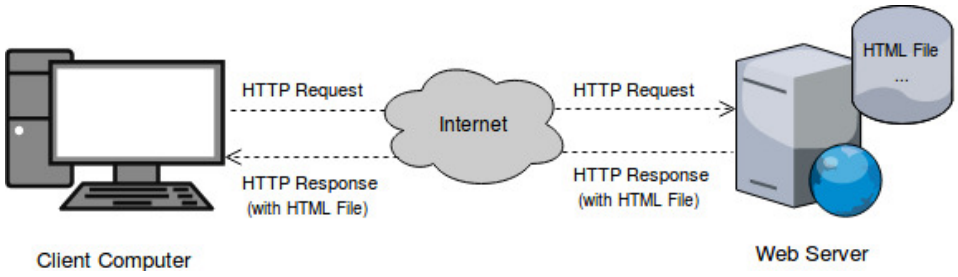


Figure 2.6: Process of request and return of a static webpage.

A dynamic webpage displays different contents on each access, which can depend on circumstances as the user login on a webpage, or other type of interaction that show the user interests, or the user history. Other conditions like changing the contents depending on the day, month, or time are also events of a dynamic webpage. As such, there are two types of dynamic webpages: *client-side webpages* and *server-side webpages*.

The initial process of a dynamic webpage request, to the web server, is identical to the process of a static webpage.

In the case of a *client-side webpage*, which processing is called *client-side scripting*, the content displayed to the user changes accordingly with his actions on the page (e.g., button clicks, menu selections, enable/disable audio/video). The page contents only depends on the user interaction with the page, as the user browser has all the code (web scripts) needed and

the content displayed is generated locally, on the client-side - browser. Hence, the process request, as referred above, occurs in the same way as in the case of a static webpage (with *HTTP Request* from the client-side to the web server). After that, the web server answers with an *HTTP Response* with the webpage scripts. The web scripts, like HTML, CSS, JS, Shockwave Flash File (SWF) files, returned by the web server response, will trigger changes on the webpage look and content after user interactions with the page. This mechaninsm is illustrated on the Figure 2.7.

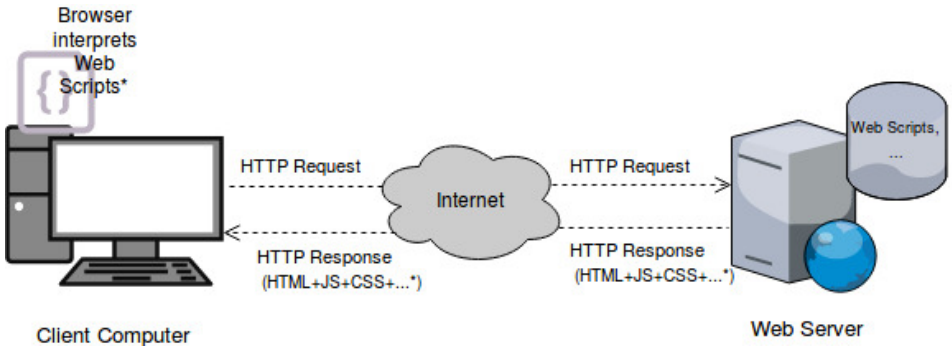


Figure 2.7: Client-side scripting process.

In the other dynamic page type, the *server-side webpage*, known as the process of *server-side scripting*, the content displayed is customized according with informations retrieved by the user (for example, login, frequent or related searches). The page content displayed is defined on the web server by scripts that run there. These scripts/programs are programmed with back-end languages. There are multiple types of languages and frameworks to run on the server-side, some of the most common ones are Hypertext Preprocessor (PHP), Python, Java, Ruby (languages), ASP.NET, Django, Ruby on Rails (frameworks).

After the client webpage request to the web server, the server will process it, by executing the programs on the server-side and, therefore, the web server will return the data needed to compose the webpage on the client-side. This process is illustrated on Figure 2.8.

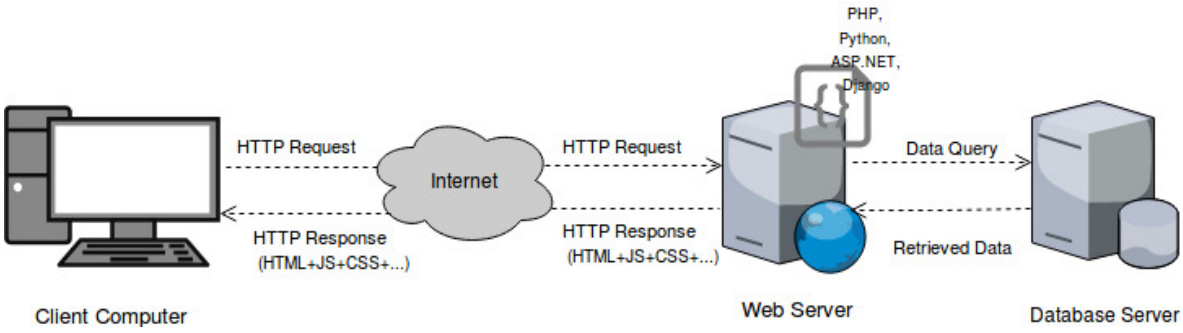


Figure 2.8: Server-side scripting process.

Normally, both dynamic types coexist, that is, both the server-side scripting and the client-side scripting happen together in a dynamic webpage.

Related with the term of webpage is the website, that consists on a set of webpages linked with the same root domain name.

Document Object Model (DOM)

The Document Object Model (DOM) is an interface for HTML and Extensible Markup Language (XML) documents that permits the representation of the webpage document as objects and nodes, so languages, like JS, can access and manipulate the webpage document content by accessing its elements. [22]

Some of the data types made available on DOM API and that can be accessed by scripting languages, in order to perform actions over them, for example: the *document*, *element* and *attribute*. These types allow to execute methods that return objects by searching its id or name, or DOM events, like scroll on the page or return the status of the webpage document, that can be[22]:

- *uninitialized*: that indicates that the document loading is not initialized yet;
- *loading*: indicates that the document is being loaded;
- *loaded*: indicates that the document already loaded;
- *interactive*: indicates that the webpage has loaded enough to allow the interaction by the user, but some resources may not still be loaded;
- *complete*: indicates that the webpage is fully loaded and also its resources.[22]

Also, scripting languages, like JS are allowed to execute scripts, to do so, it is needed to specify in the HTML document the element *script*, that contains a scripting language function, that will be triggered when the document is being loaded.[22]

The increasing complexity of the webpages raised the worrying about the users experience during the process of accessing, in terms of the loading perceived lag and how responsive they appear to be.

Browser Automation

This section will address the theme of web browsers automation, in order to allow the manipulation of the browser without human intervention. Furthermore, it will compare tools for that purpose and also, to use the browsers in a headless way, that is, use the browser without the need of using a Graphical User Interface (GUI). Also, it will be made an analysis of the current browsers market share.

Browsers

A browser is a software that allows users to navigate on the WWW, in order to access and render websites or resources, like multimedia (images or videos) and non-multimedia content that compound a webpage (JS, CSS, HTML,..), and these contents are located by their URL or URI, inputted by the end-users on the browser. The URL address supports the HTTP protocol, but also other prefixes (URI), such as FTP, for the transference of files from remote servers or *file* prefix, for the opening of local files.

Moreover, a browser can save history and cookies, that is data retrieved by the website that is saved on the user desktop, with the purpose of save it for a future access to the same webpage. This way, the end-user avoids the need of reintroduce repeated information, for example, login credentials or content preferences.

The first browser was invented by the scientist Tim Berners-Lee, that is also the inventor of the WWW. Nowadays, the scientist is the Director of the W3C, that develop technologies for the WWW [23].

Currently, the most used popular browsers, that is, with the highest number of users, are Google Chrome, Safari, Internet Explorer and Edge, Mozilla Firefox and Opera.

The Figure 2.9 shows a graphic with the evolution of the browsers market share along the period between July 2016 and July 2017. It is possible to observe that, by far, the Google Chrome browser has the highest market share of them all, showing a growth along the time period considered. Following is the Internet Explorer browser that, although having the second higher market share, is experiencing a decline. Next, the Mozilla Firefox browser appears right below with a little growth of its market share along the time period. Finally, the least used browsers, and which show a constant market share, are the Microsoft Edge followed by Safari.

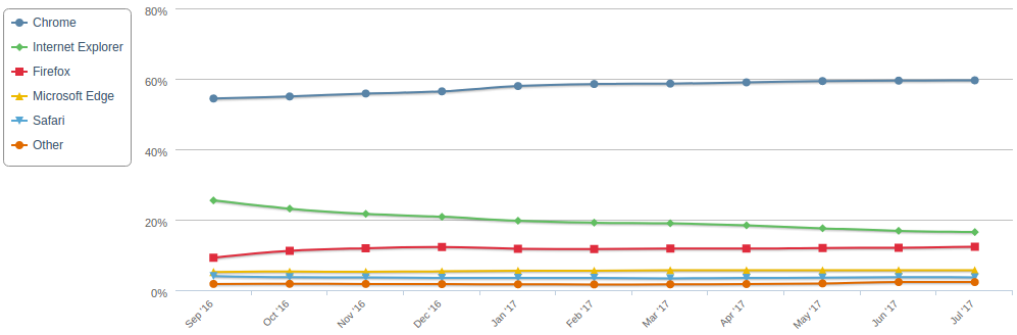


Figure 2.9: Desktop Browsers Market Share Chart, between the month of July from year 2016 to the same month of year 2017. [24]

The Figure 2.10 shows the percentage of market share mean for the same time period. The conclusions to be observed are the same stated above, by the observance of the Figure 2.9.

Google Chrome [25] is undoubtedly the one with the highest market share, also other webpages (for example: W3Schools webpage [26] or *StatCounter* webpage [27] or User Agent Breakdowns Wikimedia Foundation webpage [28]) that have studies about desktop browsers market share agree that the highest slice of the market belongs to it. This browser is free of costs for the end-user and also compatible with several operating systems (Windows, Linux, macOS) [25]. The other browsers market share statistics vary according with the market share statistic source consulted, although the ones referred above are the most popular and have, next to Google's browser, the higher market shares. For example, accordingly with the *w3schools.com* webpage [26], that counts with 45 million visitors per month, users accesses were made mostly from Google Chrome and Mozilla Firefox browsers (vide Table 2.1). The least used browsers were Microsoft Internet Explorer (IE)/Edge, Apple Safari, Opera and others less known or older.

Browser Name	**Market Share %**
Chrome	75.1
Firefox	14.1
IE/Edge	4.8
Safari	3.6
Opera	1.0
Others	1.4

Table 2.1: Most Popular Browsers Market Share, on March 2017 (Source: [26])

Mozilla Firefox [29] is a free and open-source browser, that belongs to the group of most popular browsers. Beyond these advantages, this browser is also compatible with several

operating systems (Linux, Windows and macOS).

Internet Explorer [30] is a product of Microsoft and its system requirements only allow its usage on Microsoft operating systems (Windows). Microsoft Edge [31] is also a Microsoft browser, whose characteristics in terms of system requirements are identical to the Internet Explorer, as the browser only works on Windows 10 operating system.

The **Safari browser** [32] is, as the Internet Explorer or Microsoft Edge, an exclusive browser of macOS operating systems, as it is a creation of Apple. This characteristic of system requirements exclusivity, reduce the number of possible users of these browsers, which, consequently, result on lower market shares.

The **Opera** browser [33] is free of costs and is compatible with Linux, Windows and macOS operating systems and was developed by Opera Software. It has the lowest market share in the scope of the most popular browsers, as it is visible on the Figure 2.10.

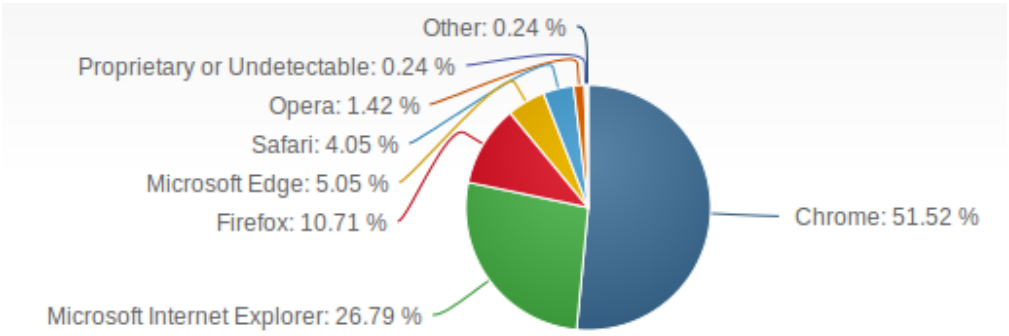


Figure 2.10: Desktop Browsers Market Share Chart percentage mean, between the month of July from year 2016 to the same month of year 2017. [24]

User-agents

When the web users request a webpage, the browser sends the user-agent in the HTTP header of the request. The user-agent is important because it has information about the browser and which version is being used and which is the device's operating system from where the request for the webpage came from. With this information, the webserver[1] can provide different content or return a different layout of the webpage, accordingly with the characteristics of the requester user-agent.

The user-agent is sent as a string that obeys to certain format defined by a Internet Engineering Task Force (IETF)[2] standard. That standard states that the user-agent field

[1] Briefly, a webserver is a server that hostages the source files of a webpage or a website and that accept HTTP requests and answers with HTTP responses.

[2] "is a large open international community of network designers, operators, vendors, and researchers concerned with the evolution of the Internet architecture and the smooth operation of the Internet" [34]

is compound by one or by a sequence of products identifiers (name and its version), optionally followed by comment(s) that together represent the user-agent software and its subproducts.[35] Although there is this standard, it allows some freedom in what concerns to its content, there is no string pattern to be followed by every user-agents.

Beyond the desktop and mobile browsers user-agents, there are several other types of them. For example, there are user-agents for: offline browsers, crawlers, gaming consoles, e-mail clients and collectors, feed readers, link checkers, validators.[36] Although that, the focus of this work will be in the desktop and mobile browser user-agents.

Below, there is an example of a desktop user-agent and the respective explanation of the information that can be extracted from it:

- *Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:47.0) Gecko/20100101 Firefox/47.0*

  This user-agent string means that the browser is Mozilla compatible with the version 5.0. It uses a *X Window System*, the operating system is Linux and runs over a sixty four bit Intel CPU and the version of the Mozilla layout engine (Gecko) is 47.0, the browser's build date was 1st of January of 2010. Finally, the name of the browser, Firefox, and its version, 47.0.

For the mobile browser user-agents, it is common that their user-agent has the word "Mobile" on the string. An example of a mobile user-agent is the following:

- *Mozilla/5.0 (Linux; Android 6.0.1; SM-G920V Build/MMB29K) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/52.0.2743.98* ***Mobile*** *Safari/537.36*

Headless Browsers

An headless browser is a browser without GUI. This type of browser is useful to be used in devices that have no graphical capabilities, or to be used to automation and tests. In this section, it will be addressed some examples of this type of browsers:

PhantomJS [37] is a headless browser, that is, a browser without a GUI, but is also a non-real browser as it is not used by real users (unlike Google Chrome or Mozilla Firefox browsers), so its purpose is only the automation of webpage interactions. This "fake" browser has a JS API and allows to [37]:

- Access and render webpages;
- Take screen captures, by capturing what is on the webpage;
- Monitor page loading, such as, the sniffing of all the webpage resources requests and responses;

- Export the HTTP Archive (HAR) files: log file that gathers the data captured during the monitoring task.[37]

Splash[38] is a JS rendering browser that is controlled via its HTTP API. This browser's features include:

- The processing of multiple webpages in parallel;
- Take screen captures;
- Disable elements in the webpage, like block images or ads, in order to enhance the webpage rendering performance;
- Execute JS in the webpage;
- Return HAR files containing rendering information;
- Write scripts in Lua[3] programming language. [38]

BrowserKit Component[40] is not a browser, but simulates the behavior of one. This tool allows to:

- Perform a HTTP request for a webpage;
- Perform clicks and submit forms in an automated way;
- Store, retrieve and create cookies;
- Store the navigation history.[40]

These headless browser tools are useful for webpage developers that want to test their webpages, but to measure the QoE they are not, although there is the option to automate some user-like tasks, the browsers are not real and consequently, the experience is not close to a real one.

Thus, to accomplish this objective, to get as closer as possible to a web user experience, there are tools that can automate real browsers and other tools that can make the automation headless, as addressed on the next subsections.

[3]The Lua language is commonly used on development of applications such as games or web applications, but is also used on image processing. [39]

Browser Automation Tools

In order to create an application that simulates the behavior of a real user, while accessing to a webpage from a browser, it is imperative to automate the browser. For that purpose, there are tools that can perform that automation, such as:

Selenium (Selenium Webdriver) [41] is a free of costs tool, open-source and free of license for commercial usage, as it uses the Apache License 2.0 [42], which does not restrict the commercialization of the product that uses it. This tool is compatible with real browsers, such as: Google Chrome, Mozilla Firefox, Internet Explorer, Safari and Opera. Moreover, it has a community of users and documentation, which helps to reduce the learning curve. It also has support for several programming languages, they are: Python, Ruby, NodeJS, C# and Groovy. In matter of features, it allows the interaction with webpages, through click functions or text input on a webpage, for example.

A Webdriver is a interface that allows to control the behavior of browsers remotely and is commonly used to write tests for browser automation.[43]

For example, the *Google Chrome* browser webdriver is the *ChromeDriver*, that is an open-source tool available for several operating systems, like *MacOS, Linux and Windows* on Desktop devices, but also for devices running the *Android* operating system. The Chromedriver's features are, for example: webpage navigation, user inputs and JS execution to trigger actions on the webpage.[44]

Another example of a known browser webdriver, is the *Mozilla Firefox* webdriver, named *Geckodriver* that uses the *Marionette*[45] remote protocol, in order to allow, similarly to the *ChromeDriver*, to replicate user-like actions. This way the webdrivers allow the manipulation of user-agents in order to simulate the users behavior while using a browser. [46]

Sahi [47] is a tool that allows browsers automation, that has two types of features packages: a non-free version (Sahi Pro) or the free version (Sahi Open-Source), this one with limited functionalities available. This fact, of features restrictions, limits the possibilities for browsers automation. The support and community, due to the reason above, is limited and closed, having focus on its clients. This tool, as it happens with Selenium, permits the usage of real browsers: Google Chrome, Mozilla Firefox, Microsoft Edge, Internet Explorer and Safari.

Watir [48] is an open-source Ruby library for browsers automation, similarly with Selenium, it can use real browsers. This library is powered by Selenium, as a derivation of the Selenium 2.0 API. Despite that fact, the *Watir* only has support for the Ruby language.

Headless Experience Tools

In order to run applications without a GUI, there are at least three possibilities, that are very similar to each other. Following, it will be addressed the characteristics of each of the three identical tools. All the tools addressed below are based on the free *X Window System*, commonly known by *X*, that provides management of GUIs on computers that use Linux or Unix-based operating systems, with cross-platform support that uses the client-server model. [49]

By cross-platform it is meant to refer that the *X* has compatibility with systems with different characteristics, such as: different operating systems, different CPUs or different hardware computing systems. [49]

And the use of a client-server model refers to the existence of two principal entities: the server and the client. The client, that commonly refers to the user machine, consumes/accesses the services provided by the servers, by performing a request to this one. The server, that commonly refers to a remote machine, can handle several clients requests and be able to process the requests and construct the responses that will be sent to the respective clients.[50] Despite that definition of a common client-server model, the *X* system uses the client-server model in an inverted way, that is, the *X* server, that is a program, runs locally on the user device and the *X* client, that is an application program, runs on a local or remote machine, and the *X* server can access to one or more *X* clients. The *X* server has the following tasks: manage the access to local graphics cards, display screens and input devices (like mouses or keyboards) and, also, perform the requests of the intended graphical operations. In that way, the *X* server services are:

- *Input handling*: sending of events from input devices, such as keyboards or mouses, to the clients via the *window manager* client, which will manage windows-related operations (for example: close, open, move or resize). [51]

- *Window services*: compound by the tasks of create or destroy windows or give information about the them to the client. [51]

- *Graphics*: related with drawing and bitmap operations. [51]

- *Text and Fonts*: include text input in certain locations using specific fonts and sending information about the fonts available. [51]

- *Resource Management*: provide database for the clients. [51]

The *X* client displays the result of the requested graphical operations on the *X* server, without using graphical resources from the machine where the *X* server in running on. The communication between server and client in the *X Window System* is the *X* protocol, which rules the messages switched between them. [51]

Next, there are the three tools that can be used in order to run an application in a headless way. Despite their similarities, referred above, they work in different modes, and so, it will be addressed their specificities. The tools are:

- **X Virtual FrameBuffer (Xvfb)** : a *X* server, compatible with *Unix* systems, that emulates a framebuffer[4] with an unreal display that runs all its graphical operations on a virtual memory. And so, it can run applications in a headless way, making it compatible with systems without graphical hardware or with none input devices. In that way, the Xvfb allows the configuration of display (fake display) characteristics (however it allows other types of configurations), for example: width, height, depth. [52]

- **X-based Virtual Network Computing Server (Xvnc)**: is a *Unix* Virtual Network Computing (VNC) server, based on the X server standard, that is free and cross-platform. It behaves like two servers, as for the applications it is an *X* server, where the applications display themselves, but for the VNC users it is a VNC server. [53]

  The VNC is a software for remote control and interaction with computers desktops and is compound by a server and a viewer. The server generates the display and the viewer is responsible for the drawing of the display on the user screen. [54]

  The Xvfb tool allows the configuration of some options for the display creation (although it allows other types of configurations related with the connections or where the application can be run from), for example: width, height and depth. [53]

- **Xephyr**: it is a kdrive[5] based *X* server that display its outputs on a host *X* display, that is used as a framebuffer. [56]

  The Xephyr also allows the configuration of the display dimensions: width and height. [53]

Existing Solutions

This dissertation was triggered by the need to evolve from the ArQoS NG solution, that has its focus on the QoS of the services provided by the telecommunication operator, to some solution that could allow the extraction of metrics that gather the QoE, with the maximum approximation to the QoE perceived by real users.

These days, there are several web tools focused on webpage developers, that want to keep their pages on the top of the most well scored by the web engines, and make the page more appealing to the user and reduce its early abandonment, increasing the time the user spends on the webpage visit. To do so, there are tools which focus is on the webpage structure and

[4] Area of the RAM memory where the information is kept to be displayed on the device screen.
[5] *X* server designed for environments with low memory capacities. [55]

organization, but are also good tools to perceive part of what the user perceives, as they return time metrics about the page responsiveness. Although, in fact, these tools cannot, by themselves, return the QoE of the page, as they do not consider webpage interactions.

Google PageSpeed Insights[57] analyzes the content of a webpage, for desktop and mobile devices. It evaluates the HTML structure, the organization of external resources (images, JS and CSS files) and the server configuration. Then, returns suggestions to improve the webpage performance, in terms of time to above-the-fold load and time to full page load and classifies the page with a score (within 0 and 100) and a classification ("Good", "Needs Work" and "Poor"). [57]

Pingdom[58] is a non-free tool that evaluates several aspects of a webpage, such as: webpage availability, interactivity, page speed (like, load time). But also provides alerts for webpage malfunctions, tests the webpage from multiple locations and the webpage structure and external resources usage. Then, provide reports to the user. [58]

WebPageTest[59] is an open-source tool that can test a webpage from different locations using real browsers (Google Chrome and Internet Explorer). It allows the user to test his webpage on scenarios of multi-step transactions, video capture and content blocking, but also in terms of load time, time to the first byte and time to start the render. [59]

But, in fact, the above tools are not easy to automate nor embrace all the stakeholders, for example, the telecommunication operators.

Today, there is an increase of works that address the theme of QoE estimation. In this context, there are several papers that state the metrics that can provide clues about the quality perceived by the end-users. Some of the research is also about the development of software tools that can acquire and return the quality of experience for services, and the web browsing is one of the concerns, specially in the mobile scenario. This way, following are presented some solutions which aim is the measure of the experience and service quality

Intraway QX Suite

This is a tool that emulates the behavior of the end-user and measures the experience through the use of automatically programmable probes.

The *Intraway* solution has the ability to: detect defects outside the network, that have impact on the clients experience; Perform measures on applications with Over-The-Top (OTT) contents, that consist on multimedia content (audio, video or other media type) that is transported over the Internet as an isolated service instead of using an operator service, and are examples of this type of OTT content applications: the *Youtube*, the *NetFlix*, the *WhatsApp*

or the *Skype*; Perform measures on the operator Video On Demand (VOD) (systems that display media content that the user selects when, where and which part to see) platform, in order to evaluate the connections reliability to the online games servers.

The tool also allows to perform: Speed tests, in order to measure the connection downstream and upstream bandwidths and latency; For the web browsing it provides the times to full page load, but also, the time to the first frame received and the network interface bandwidth; It performs pings to acquire the network latency, jitter and packet loss; It does download and upload of files to measure the download and upload bandwidths and times, bytes sent and received and the time it takes to connect to the server that hosts the files; In the video streaming field, it measures the streaming bandwidth, and identifies buffering and pauses timings; It also measures the UDP performance (in gaming, Voice over IP (VoIP) and others) in terms of latency, jitter, packet loss; It measures the DNS lookup time for IPv4 and Ipv6; And, performs packet sniffing to catch the TCP/UDP packets.

The *Intraway QX Suite* also provides reports with the Key Performance Indicators (KPI) collected, and the possibility to define thresholds to each of them and display the data by history or by specific time intervals. Also, it provides information about the time the peaks occur. [60]

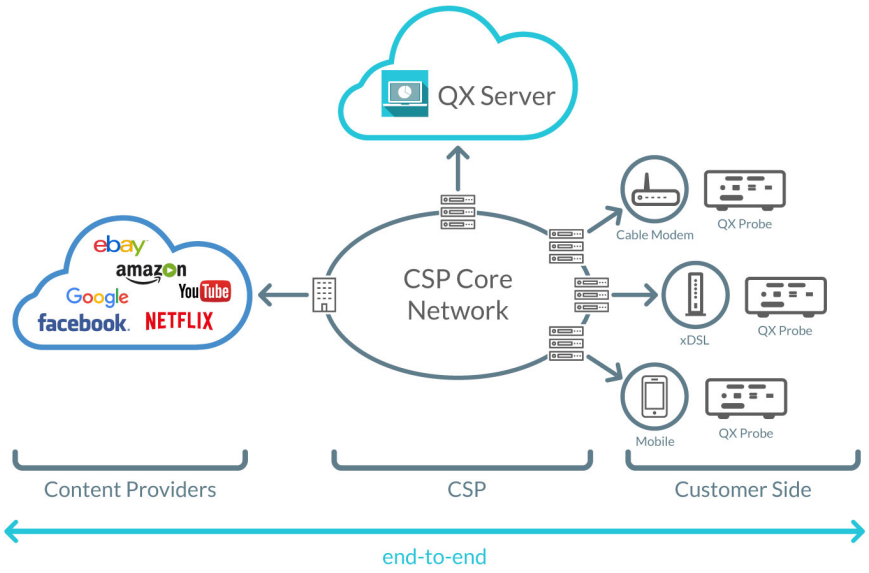


Figure 2.11: Intraway Qx Architecture.[60]

The tool uses three types of probes to perform the measures, like it is shown in the Figure 2.11: the *Small-PC Probe*, which focus is on the measurement of the user experience on video streaming; The *Router Probe*, based on *Linux*, is a probe that makes measures of the most

common KPIs of the QoS and also performs measures on video streaming, to the level of its transmission across the network; And lastly, the *Smartphone Probe* is an *Android* application developed to evaluate the *Android* mobile device users, as well as, measure KPIs related with the quality and performance of the mobile network. [60]

The probes used are deployed behind a Customer Premises Equipment (CPE), fix or mobile, in order to return the status of the network service. [61]

Firelog

The *Firelog* is an hybrid probe that performs measures in active and passive ways in the domain of Web navigation. Its aim is to detect low QoE (for example, a high page load time) and diagnose what are the causes.

This tool is constituted by three different modules: a non-real headless (without a GUI) browser, the *PhantomJS* (vide 2.3.3); a *network packet analyzer*, that performs measurements at the transport-layer; And an *active measurements monitor*, which aim is to gather information from the network layer. These components perform events and metrics capture, and store them on a local database, in order to evaluate the current web browsing session, in search of the causes for high timings of web page loading.

This tool tests are performed in three steps: The first step corresponds to the URL browsing, in order to extract the metrics on the application level and gather all the fetched objects from the webpage; The second step corresponds to perform *traceroute* commands to all the webpage objects loaded, in order to collect path informations to all the webpage objects and the measuring of the RTT to the webpage IP address. In the final step, all the information gathered is stored for later analysis.

The main metrics collected are the following: the page load time, the time to the reception of the first byte, immediately after the HTTP GET object request, the TCP handshake time, the object reception time and its length, the RTT to the destination and the size of the page, defined as the sum of all objects sizes.

This tool is compatible with Unix/Linux environments. [62]

QoE Doctor

This tool uses automation techniques to reproduce the interaction of a real user using a mobile device. The tool automates interactions, such as the publishing of posts on *Facebook* and, at the same, performs time measurement of the latency perceived by the user, through the changes that occur on the mobile screen.

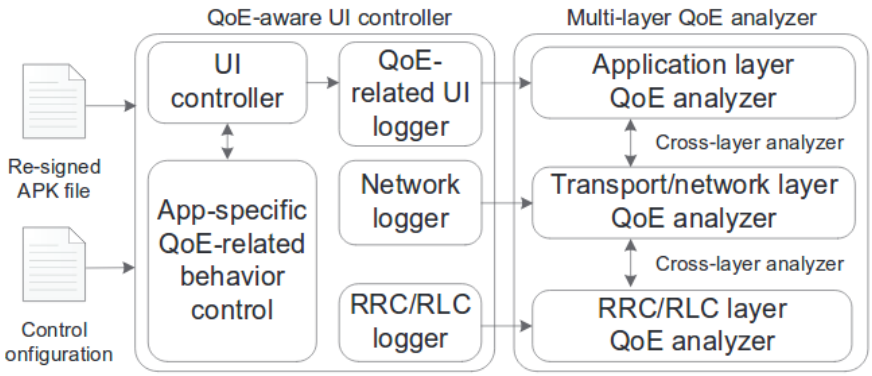


Figure 2.12: QoE Doctor Overview.[63]

The *QoE Doctor* analyzes several layers of the network (application, transport, network and radio), as can be observed on Figure 2.12, with the objective of detecting the causes of a good or a bad user experience. Although that, this tool was developed only for *Android* platforms and only performs measurements for popular applications, such as the case of *Facebook* (publish posts and update the news feed), *Youtube* (video visualization), *Google Chrome* and *Mozilla Firefox* (to load a webpage).

The components of the *QoE Doctor* shown in Figure 2.12 are: the *QoE-aware UI controller* that runs on the mobile device and automates the user interface, in order to reproduce the user behaviors, like button clicks or page scroll, and collects, simultaneously, data from the application, transport, network and radio layers; The data collected from the layers, referred previously, is further analyzed in offline mode, by the *Multi-layer QoE Analyzer* component.

The metrics obtained are: the user perceived latency, in user-tasks, for example, the news feed update or the upload of a post to *Facebook*, or the stalls during a video playing (on *Youtube*); It is also gathered the video initial loading time and the re-buffering ratio (on *Youtube*); In web browsing, it is measured the webpage load time; The tool also gathers traffic logs from the transport and network layers, allowing to measure the mobile data consumption, during the user behavior replaying; And gathers information from the radio link layer that allows the measurement of the device battery energy consumption, during the test.[63]

Comparison of QoE Estimation Solutions

The Table 2.2 gathers the main metrics collected by the tools described above, the *Intraway QX Suite*, the *Firelog* and the *QoE Doctor*.

Metrics	*Intraway QX Suite*	*Firelog*	*QoE Doctor*
DNS Lookup Time	Yes	–	–
TCP Connection Time	–	Yes	—
Time To The First Byte (TTFB)	Yes	Yes	–
Response Time	–	Yes	–
Page Load Time	Yes	Yes	–
Initial Page Load Time	–	–	Yes
Update news feed Time (*Facebook* website)	–	–	Yes
Re-buffering Ratio (*Youtube* website)	–	–	Yes
File Upload / Download Time	Yes		–
Packet Loss	Yes		–
Latency	Yes		Yes

Table 2.2: Table with the main metrics collected by the three solutions.

There are other differences like, the devices where the tools run, as can be seen on Table 2.3.

Device	*Intraway QX Suite*	*Firelog*	*QoE Doctor*
Mobile	Yes	–	Yes (Android)
Desktop	Yes	Yes	—

Table 2.3: Table with the target devices of the referred tools.

As previously referred, these tools have different characteristics and collect different metrics, although with some similarities between some of them, as can be observed on Table 2.2. The main differences, besides the metrics returned by each of them, are the devices where the software applications run. Some are focused only on mobile devices, like the *QoE Doctor*,

others, like the *Firelog*, has it focus on a performing tests on a "fake" browser (not used by real users), that however, allows the emulation of different user-agents. And lastly, the *Intraway QX Suite* can test several devices types, as it does not run on them, but in probes behind the user devices.

In fact, these tools have the same purpose of measuring the QoE perception of a real user on scenarios like the web browsing, that is the focus of this master thesis.

Conclusion

This chapter intent was to approach all the themes that were related with this dissertation title, in order to serve as an introduction and contextualization to the next chapters. For that, were approached themes from the most core concepts, as the Internet concepts to tools and existing solutions that could be taken as an input to new research work on the Web QoE measurement area.

3

SmartBrowsing Solution

In this chapter, it will described the proposed solution. The *SmartBrowsing* solution has the aim to automate a web navigation scenario (the access to a webpage), in order to extract informations about it. Moreover, the solution must be able to automate scenarios of common user interactions. Hence, in this chapter, the solution will be presented through the description of the requirements and diagrams, such as, use cases, flow and the system's architecture.

Overview

The collection of experience-related metrics implies the need to extract data from a real navigation. Therefore, this work focus is on the development of an application that will impersonate the behaviour of a real user, while accessing to a website. This tool must access to webpages by using a common browser, in order to approximate this automation to a real web navigation. Hence, the objective is to mirror a real UX, by using real browsers and simulate different types of user-agents, along with the possibility of automate interactions with the webpage.

This solution came in the sequence of the need to evolve from an earlier existing solution: the *ArQoS* solution, which focus is on the extraction of QoS metrics and does not allow the testing of generic interaction scenarios (such as, login, logout and scroll actions). In order to download the webpage and obtain some measures, related to its access, the existing *ArQoS* solution uses the *wget* package, which is a command line tool.

The tests performed with the application should be made under different conditions of network and hardware capabilities, allowing the possibility to trace profiles based in all these variants, with the aim of inferring about what does have a significant impact in the real end-user Quality of Experience. This means that, although there is no option to configure

the network or the hardware device, the application should run in different devices connected to different networks, if the device is configured with the needed packages and tools. The *SmartBrowsing* must compatible with the *ArQoS* probes, for future integration.

This program (bot) should have a configuration file as input, in order to define the conditions under which the tests will be executed. In this configuration file, it should be possible to setup several fields, in order to have the desired testing scenario. Some of the setups available to be configured must be: the browser, the URL of the webpage to be tested, the "display"[1] resolution to be emulated, the user-agent, the interaction(s), the enable or disable of the webpage download (all files and media that are part of the webpage), and the paths to where the output files, such as logs and screen captures must be saved to.

The program should start by performing the access to the URL, of the configured webpage to be tested, and then, extract metrics, load information about the webpage components. And, also return, for example, the status code of the webpage and for each of its loaded components.

Moreover, it should allow to take "screen" captures along the test, with the purpose of being possible a future analysis of those screen captures. These screen captures can serve as a proof for some of the information extracted or, otherwise, help to diagnose problems occurred during the test.

Additionally, as referred above, the application must have the ability to perform interactions with the page. In order to approximate the simulation to a real human user experience. These interactions should be performed by doing clicks and scroll tasks on the webpage, and further measure the time it took to accomplish each interaction, and gather the loaded components information.

This program should perform the tests on a real browser as if a real-user was accessing it in the anonymous/private/incognito mode. Since that, topics of cache and cookies were not considered in this solution.

Use Cases

In Figure 3.1 are visible the options available to configure the input file. This configuration file is a JSON compound by several fields, where can be defined: the browser, with two possibilities of being filled with: *Google Chrome* or *Mozilla Firefox*; The webpage URL; The user-agent, that if it is left empty will be assumed the default of the device system in use; The "display" resolution, that can be overlapped by the resolution of the user-agent defined; the possibility of download the full webpage, i.e., all the media, for example, images or videos, and files of the page, such as the HTML, JS and CSS files.

The browser to be used, during the test execution, should be an option of two possibilities,

[1]The word "display" appears surrounded by quotation marks, due to be a headless display and not a real GUI, this will be further addressed on the system requirements, in this chapter.

the *Google Chrome* or *Mozilla Firefox* browsers. The reason for these choices rely, mainly, on these browsers market share, that will be further explained (vide section 4.1 from chapter 4), and on the system requirements (vide section 3.3). The availability of two browser options is to allow the analysis of the differences between tests executed on different browsers.

The URL to be configured defines the webpage that will be tested, however, when login or logout interactions are configured, the solution must return, besides the time it took to perform the interaction, the time it took to load the webpage requested after the interaction (for example, in the login case, it must return the time it took to load the webpage requested after the login success).

The user-agent field must be configurable or not, in case of no user-agent setup, the program should assume the default one, which is the system's user-agent.

The "screen" resolution is a required field to be configured.

The download of the webpage and its resource must be optional, that is, it must be possible to define, in the configuration file, if it is to be done or not. If the test is configured to download the webpage, this download must be outputted to a compressed folder. Besides that, it is needed to define the location to where this compressed folder will be saved to.

Other possible configurations should be interactions with the webpage, such as: login, logout, scroll down to the bottom of the page, or to a certain position on the page. In the interactions case, it must be possible to configure one or more by test. Although, it must be optional to setup interactions on the configuration file, as it must be possible to run a test without interactions configured.

The interactions are performed by the order of setup in the input configuration file.

To setup the configurations to be performed, it is needed to define which is the type of interaction and which are the actions to execute, by order, to accomplish the task. Also, it is needed to specify the element, which visibility on the "screen", will state whether the interaction was successfully finished or not.

Finally, the configuration file needs the paths (compound by the directory path plus the file name) to where the output files (logs, results, "screen" captures and, eventually, the zipped folder, containing all the webpage's files loaded) are going to be saved on the device's system, where the test is executed.

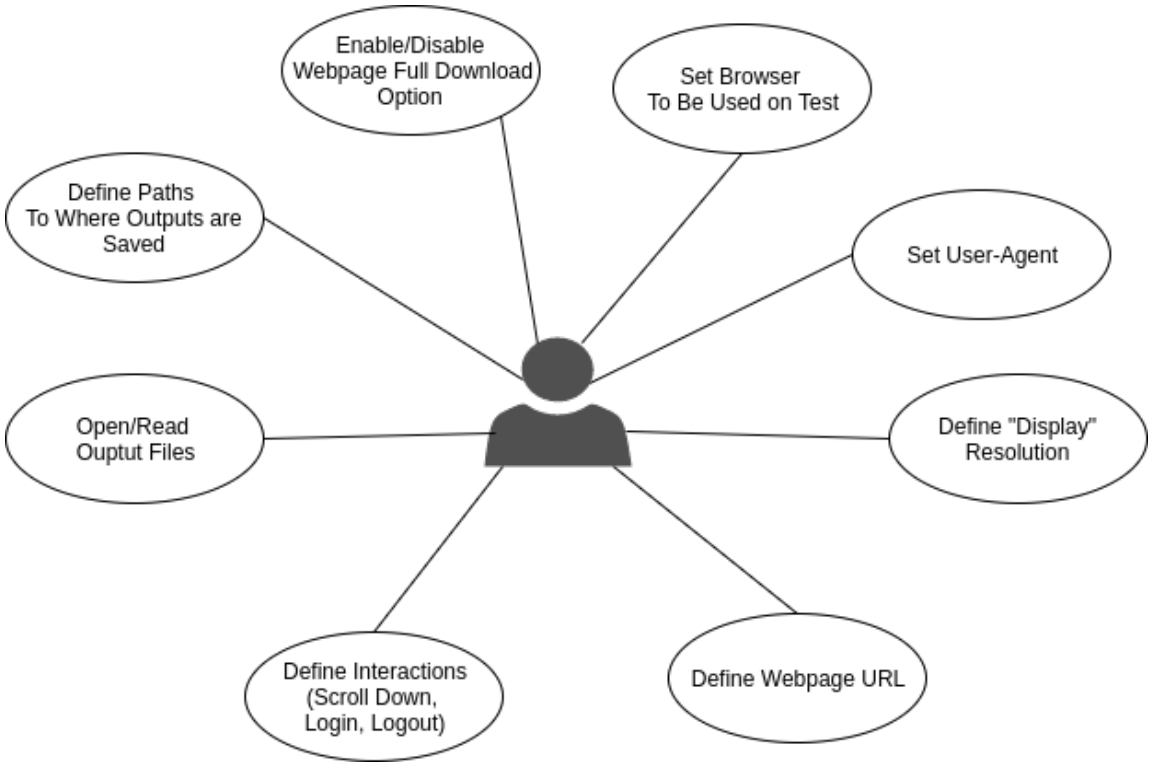


Figure 3.1: *SmartBrowsing* Use Case Diagram.

Requirements

This system should be developed under certain requirements that are crucial to the compatibility of this solution with the *ArQoS* probes. These requirements will justify some of the choices made in terms of the technologies adopted. Those requirements are: the *SmartBrowsing* solution must have no graphical interface, that is, no GUI, in order to be possible the integration of the solution in the ArQos Probes, as these probes have no graphical capabilities.

Furthermore, the solution must run on a *Linux* system. Besides the fact that *Linux* is an open-source operating system, it is the operating system running on the ArQos Probes. Since that, it is imperative to have a solution that can run on *Linux*.

Beyond these requirements, there are others needed to fulfill the purpose of the solution in matters of QoE. The proposed solution must be able to return metrics that could help the measure of QoE, and also be able to take "screen" captures, although the requirement of running in devices without graphical processing capabilities as previously referred.

Furthermore, the solution needed to perform tests on a scenario as much real as possible, in order to be closer to a real navigation on the Web. In that way, the use of real browsers,

that is, browsers commonly used by real users, like *Google Chrome* and *Mozilla Firefox* was imperative. The reason beyond the browsers choice is explained in the following chapter, in section 4.1.

Also, the solution must consider the possibility of emulating different user-agents, to have more variables in the equation to determine if these differences (device and browsers used) impact on the QoE.

Other requirements that add value to the application would be the possibility to configure tests to save the full webpage accessed, and return files: with results (JSON file), to be processed by a management system; Log files, one for the clients of the application service with few details about the steps taken during the test, and another for the support team, with more specific details of the steps and tasks performed by the program, to help in case of an abnormal behavior or error of the solution.

Overall, the solution must provide a totally automated tool, that allows the configuration of tests. The tests must be configurable to acess a webpage, through the use of real browsers. The solution must have the capability to gather information and perform interactions without human intervention.

Flow Diagram

Figure 3.2 shows the program flow: which actions are performed, which decisions need to be taken and which are the program inputs and outputs.

Initially, before the program start, it is inputted a configuration file with the informations wanted to test a specific scenario. This input file is a JSON that will be read right after the start of the program is triggered and its configurations are assumed, to be considered along the program run.

After the reading and processing of the inputted configurations, the headless "display" is created and initialized, in order to run the application program without the need of a graphical interface.

Following, is the initialization of the *Selenium Webdriver*, that is the tool that will allow the browser automation, and which is the mean to perform interactions with the page. The *Selenium Webdriver* is initialized accordingly with the browser chosen and specified on the input configuration file: if the browser chosen was the *Google Chrome*, then, the webdriver used is the *Chromedriver*; Otherwise, if the chosen browser was the *Mozilla Firefox*, then the driver to be used is the *Geckodriver*.

The tests are performed without using cached data, as the webdriver runs a browser session as if it was in anonymous/private mode. And, also, only one access is made to the configured URL.

Next, with all the needed initializations concluded, it is triggered the access to the web-

page, using the URL read from the configuration file.

After, are extracted webpage informations, such as the status code, and temporal instants, that correspond to triggered events, that will be used to do further metrics calculations.

Furthermore, it is returned a list of the webpage components, that were loaded after the access to the webpage, along with informations about each, such as: its identification *href* (that is a HTML attribute with the URL of the destination webpage), the status code and temporal instants.

From this point, the program can proceed to the closing and saving, on the directory path defined on the configuration file, of the log files and the application program ends. Or, if there were one or more interactions configured on the input file, then the program proceeds to perform the interaction(s), and measure the time it took to complete it. The interactions available to be configured are: login, logout or scroll down to the bottom of the page or to a certain position of this one.

Immediately after each interaction, are extracted the loaded components and also informations and temporal instants of each of them. This succession of events occur till there is no more interactions on the list read from the configuration file. So, when the interactions lists reach the end, the program proceeds to the close of the writing to the output files (the debug and clients log, that have informations about the events that occur along the program run and the JSON file that only has the values and informations extracted) and the program is finished.

In the end, there is another output that can be also saved, a compressed *zip* folder that contains the webpage source code and all the components that are part of the page, although this is only an output of the program if it is specified on the configuration file that the webpage download must be done.

Along this flux of events and actions are taken "screen" captures along the run, that are saved on the directory path defined on the input configuration file.

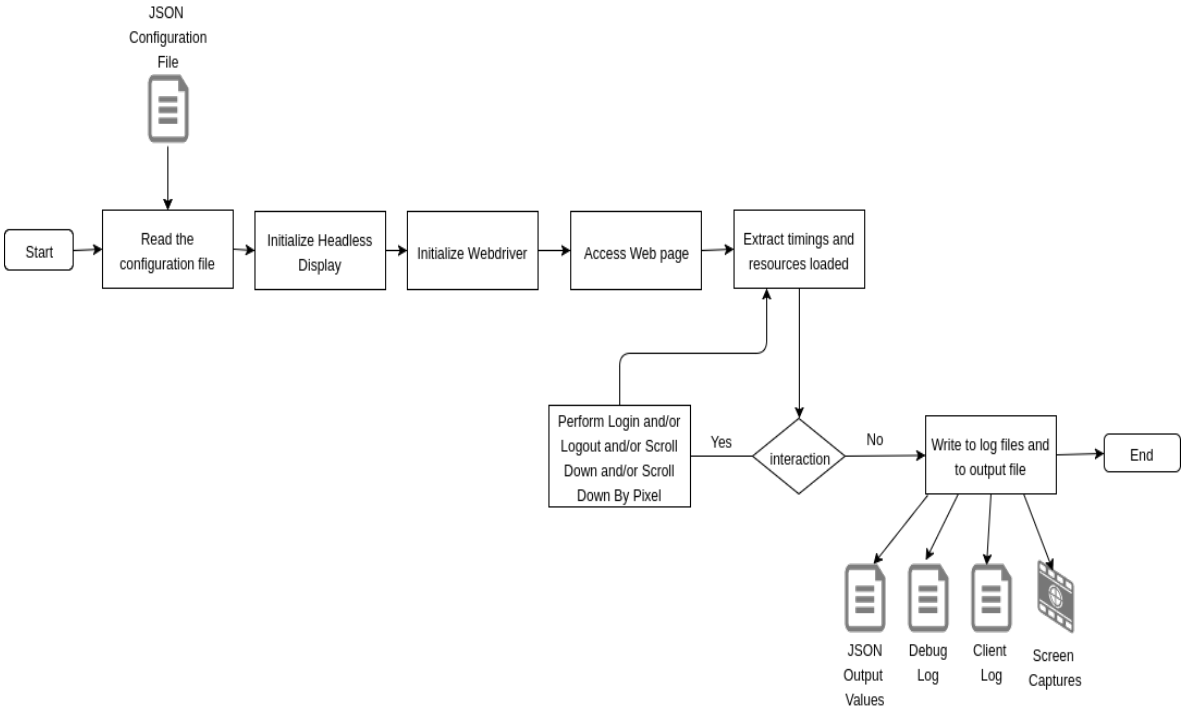


Figure 3.2: *SmartBrowsing* Flow Diagram.

Architecture

In this section it will be proposed the application architecture, in order to describe the overall components of the solution and how they should be connected to each other.

The Figure 3.3 depicts the *SmartBrowsing* solution.

To run *SmartBrowsing* solution, it must be used a computer device (with or without graphical capabilities), running Linux and have real browsers installed (vide section 3.3).

Furthermore, the test performed, using the solution, should access the webpage specified in a configuration file, through a real browser. This access must be made along with a tool that allows to automate a navigation, such as the *Selenium Webdriver*. This is needed because, the tests must run without human intervention, and gather information from it, such as: timings that would allow the calculation of metrics, take screen and perform user-like actions (login, logout and scroll interactions).

Furthermore, the use of a automation tool, like the *Selenium Webdriver*, implies its in-

stallation on the device. Also, the device must have the webdrivers installed, for the browsers to be considered, the *Google Chrome* webdriver (*ChromeDriver*) and the *Mozilla Firefox* webdriver (*Geckodriver*), in order to allow the browsers running.

These webdrivers, *ChromeDriver* and *Geckodriver*, have functionalities that allow the automation and interaction with their respective real browsers. Those webdrivers will perform the access to the webpage, by making an HTTP GET request.

Afterwards, will execute a JS on the browser in use, to obtain the timings from the webpage accessed.

If there are interactions to be made, the webdrivers use their methods, such as, clicking, submit forms or scroll, in order to trigger actions on the webpage.

After each interaction, the API must be called via JS execution, as referred, to extract the webpage resources loaded and its informations.

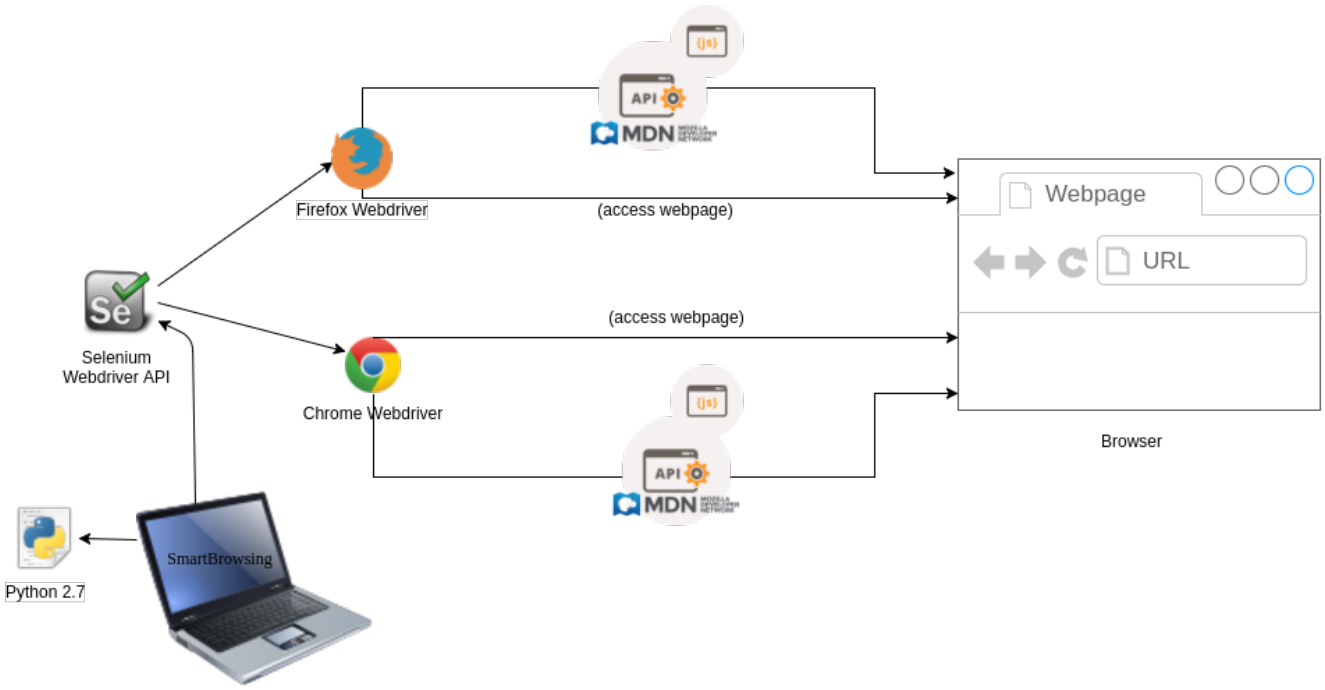


Figure 3.3: *SmartBrowsing* Architecture Diagram.

Conclusion

This chapter intended to provide the overall scenario of the solution developed in the context of this master thesis.

In order to provide the *SmartBrowsing* solution description, this chapter approached details, such as: the overall description, the solution requirements and the use cases, flow and architecture diagrams.

4

Implementation

In this chapter it will be described the adopted technologies and explained the details of the implementation of the solution, along with some code snippets, in order to better illustrate the deployment.

Adopted Technologies

In this section, it will be described all the technologies adopted in the creation of the *SmartBrowsing* solution. Beyond that, the choices of the technologies adopted will, also, be justified.

The first base of this solution was to automate a browser, since that, it was used the *Selenium Webdriver* (vide 2.3.4) tool. This tool was chosen instead of the others, because its use is free of licensing for commercial uses, which is a big advantage as this project can become a product, since it is licensed under *Apache 2.0*. Other advantages are the fact that it is compatible with real browsers, for example, *Google Chrome*, *Mozilla Firefox*, *Internet Explorer*, *Safari* and *Opera*. Beyond this, it counts with a large user community and open documentation, allowing a more efficient development and a lower learning curve, compared with other tools which support is limited, due to being payed tools or less popular (smaller community) (vide 2.3.4). Finally, it has the advantage to be compatible with several programming languages: *Python*, *Ruby*, *Java*, *NodeJS*, *C#* and *Groovy* and has webpage interaction functionalities, like clicks or text insertion, besides others.

In order to emulate a reality-close experience, real browsers were used to perform the tests. For the browsers choice it was taken into consideration the need of being compatible with *Linux* systems and also, rankings of the most used and popular nowadays. Hence, the browsers that fulfilled these requirements and were the selected are: *Google Chrome* and

Mozilla Firefox. As can be observed on the subsection *Browsers* 2.3.1, on the *State-Of-The-Art* chapter:

- Figure 2.9 depicts the *Google Chrome* as the one with the highest percentage of market share, comparing with the other browsers, and also shows some growth.

  Although *Internet Explorer* has the second better market share, it presents a decline. Also, it has the disadvantage of being incompatible with *Linux* systems, which is one of the solution requirements. For the same reason, the *Safari* browser was excluded, since it is only compatible with the *Apple* operating systems, *MacOS*.

- Table 2.1, from a *W3Schools* website study, and the Figure 2.10, from the *NetMarket-Share* website, show that the *Firefox* browser has one of the three bigger market shares and is *Linux* compatible. These advantages were the influence for the choose of this browser. The *Opera* browser, which is also compatible with the *Selenium*, has a very little percentage on the market share, which was the reason for not being chosen.

Hence, were chosen two browsers that were representatives of the major slice of the market share, in order two have two options and, also, to infer if the browsers impact on the QoE.

To connect the use of these browsers to the *Selenium*, it was required to obtain the correspondent browsers drivers, the *Chromedriver* (for *Google Chrome*) and the *Geckodriver* (for *Mozilla Firefox*). These browsers drivers allow the connection between the *Selenium* automation tool and the respective real browser, *Chrome* and *Firefox*.

The programming language used to develop this application was *Python*, since it was one of the languages with compatibility with the chosen automation tool, the *Selenium Web-driver*. Also, as the Figure 4.1 depicts, *Python* has an emerging popularity, along with an increased demand for programmers with knowledges on this language. The Figure 4.1 shows a popularity ranking of programming languages, that relies on a study published in the year of 2017, made by the *IEEE Spectrum*.

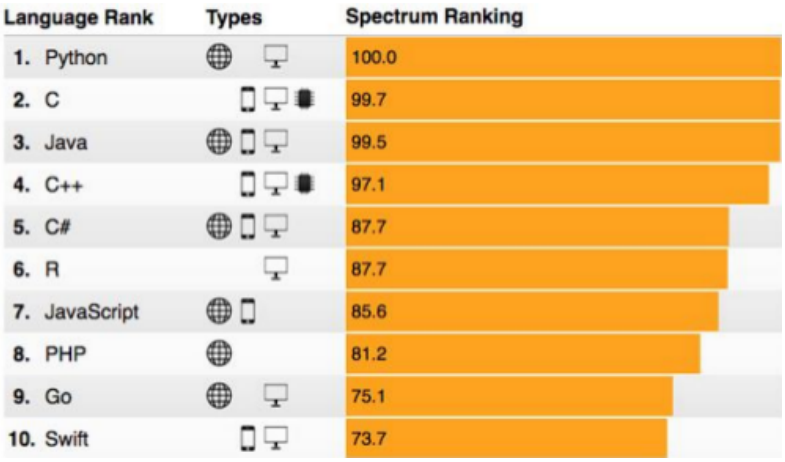


Figure 4.1: Programming languages ranking, according with a study made by *IEEE Spectrum*. (Image source: [64])

The ranking shows that *Python* is the most popular, along with *C* and *Java*. Since that, from the top three languages, *Python* and *Java* are compatible with the automation tool, *Selenium Webdriver*, so it was chosen the *Python* language due to the advantages previously stated, that rely on its popularity growth along programmers and even on enterprise context.

Those advantages, previously stated, are particularly important, as the long term support is imperative if this solution becomes a product in the future.

Since a critical requirement (vide 3.3) to this solution was the ability to run in a headless environment, in order to be possible the future integration on the *ArQoS* probes, the Xvfb tool (vide 2.3.5) was used. This tool runs the graphical operations in a virtual memory. There was other two similar options beyond this, as they could be used for the same purpose and with similar capabilities. In that way, any of them could have been used.

In order to integrate this headless tool on the solution, it was needed to use the *Python* library, *PyVirtualDisplay*, that is a wrapper for the Xvfb. This way, the solution opens the browser in a virtual framebuffer, the Xvfb, so the solution can run in a headless mode.

The Figure 4.2 depicts the *PyVirtualDisplay* hierarchy. The *PyVirtualDisplay* is a library that provides a wrapper to the virtual displays *Xvfb*, *Xvnc* and *Xephyr*, so, applications that are implemented using the *Python* programming language can use these tools.

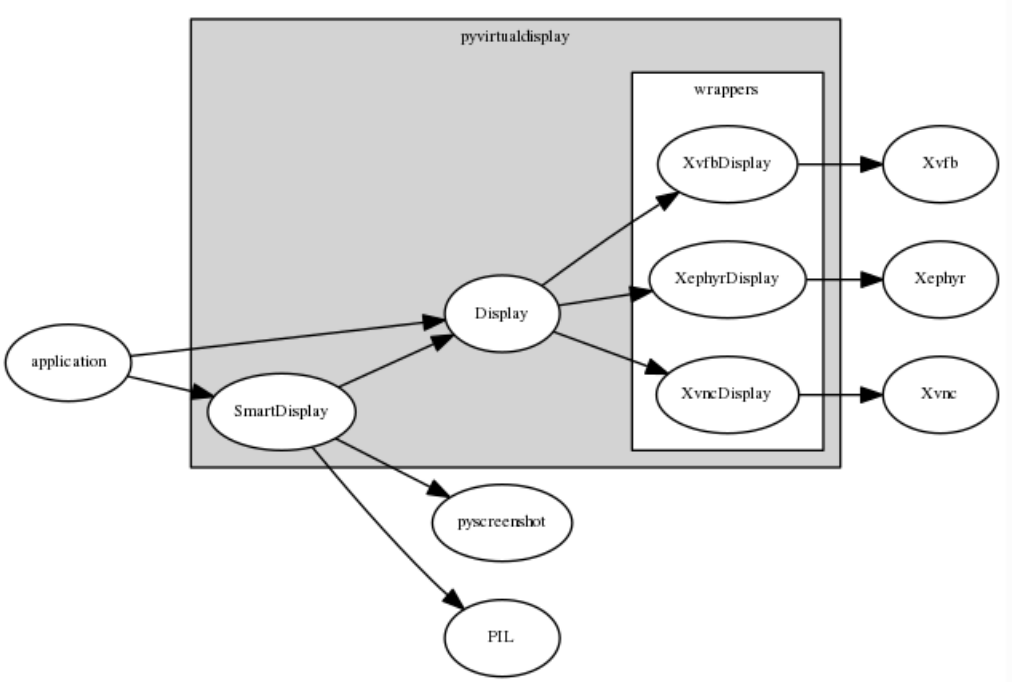


Figure 4.2: *Python*'s library, *PyVirtualDisplay*, Hierarchy. (Image Source: [65])

As the main purpose of this work was to extract QoE metrics, it was used the Performance MDN API [66], that, by executing JS requests, can return several temporal instants along with other informations, such as: the list of external resources loaded, along with temporal instants for each and other informations like its identification HTML attribute - *href*. And this API has compatibility with the browsers chosen: *Google Chrome* and *Mozilla Firefox*.

The MDN Performance API is based on a non-normative webpage processing model for a browser, and it belongs to W3C. [67]

That W3C model can be observed on Figure 4.3, where there are shown the different events that occur between the input of the URL on the browser till the webpage is totally rendered on the browser window and visible to the user. However, as the non-normative word suggests, these timeline events can occur in another order than the one presented. This depends on the browser used, on the webserver hosting the webpage, on the network technology or even on the users device processing capabilities.

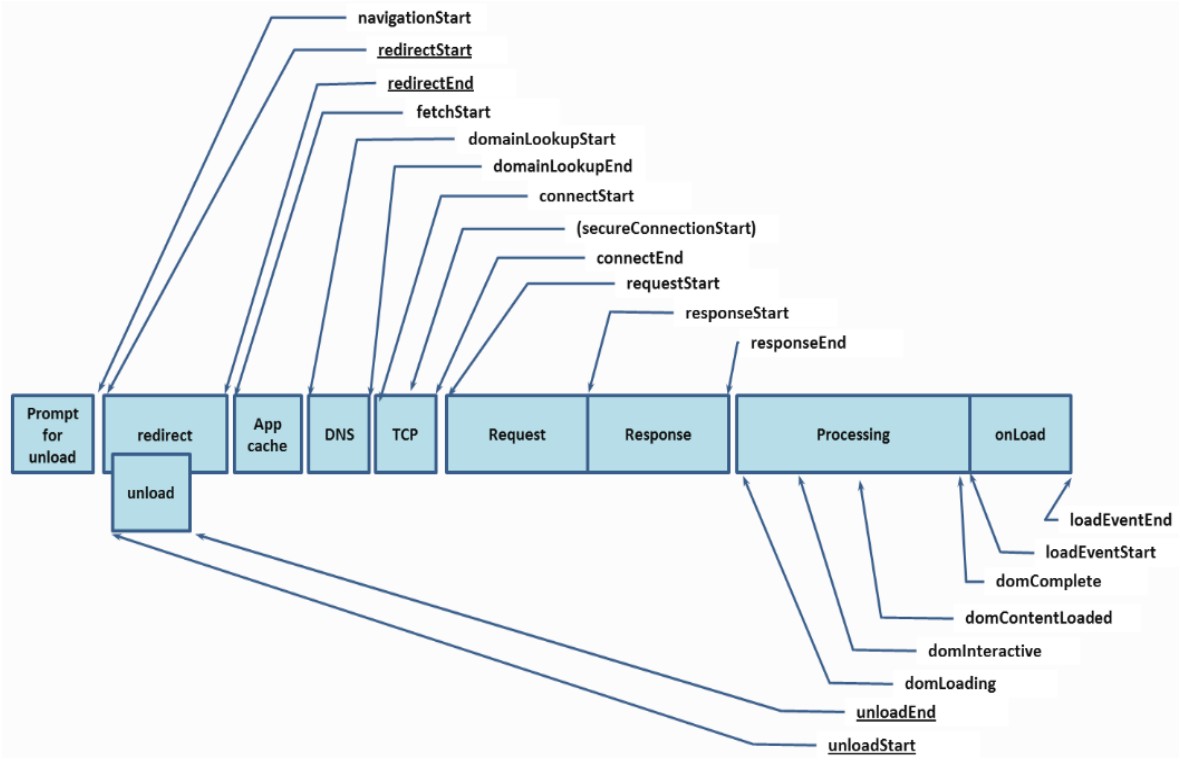


Figure 4.3: Browser Non-normative Webpage Processing Model, from W3C.[67]

The events, returned by the API in milliseconds and presented on Figure 4.3, were used to perform the metrics calculation, that will be further explained in this chapter on section 4.5. Each of the events will be briefly explained on the topics below. Those topics, that describe each of the time events from the non-normative model of W3C (vide Figure 4.3), will appear underlined in case they were the ones taken into consideration to perform the metrics calculation extracted in this solution. This will be further addressed on the Subsection 4.5.

- *navigationStart*: Temporal instant that occurs immediately after to the user-agent perform the *prompt for unload* of the previous document (webpage).

  If there is no previous document, the temporal instant returned will have the same value of the *fetchStart* event.

- *unloadEventStart and unloadEventEnd* (*unLoad* block): The *unloadEventStart* is triggered immediately before the beginning of the previous document unload, by the user-agent. This event occurs if there is a previous document, and if it has the same origin from the current one. On the opposite, i.e., if there is no previous document, or if this one does not have the same origin from the current one, then the value returned by the *unloadEventStart* will be equal to zero.

For the *unloadEventEnd* case, the contrary occurs, i.e., the temporal instant returned is the one immediately after the user-agent ends the previous document unload. Similarly to the *unloadEventStart*, if the previous document does not have the same origin from the current one or does not exist, the temporal instant returned will be zero.

Thus, if the previous document is still not totally unloaded, then the *unloadEventEnd* should return zero.

- *redirectStart and redirectEnd* (*Redirect* block): The redirect block is defined by both this time instants, the *redirectStart* and the *redirectEnd*. This block represents the time interval between the instant of the beginning of the fetch of this redirect (*redirectStart*) and the instant immediately after it (*redirectEnd*), which corresponds to the reception of the last byte of the response from the last redirect.

  This block only returns these time instants if there are HTTP redirections with the same origin, otherwise, the value returned by these instants, *redirectStart and redirectEnd*, will be zero.

- *fetchStart*: Time instant that occurs immediately before the user-agent begins the search for applications in cache. This is considered only if the component is fetched by HTTP GET requests or equivalent, otherwise, the value returned will be zero.

- *domainLookupStart and domainLookupEnd* (DNS block): The time instants that delimit this block, *domainLookupStart and domainLookupEnd*, allow the calculation of the domain lookup time for the current document, in case the connection in use is a non-persistent[1] one.

  If the connection is persistent, or the current document is saved on cache or on local resources, the value of both these time instants, *domainLookupStart and domainLookupEnd*, is equal to the *fetchStart*.

- *connectStart and connectEnd* (TCP block): The time instants that delimit this block, *connectStart and connectEnd*, compound the time interval of the TCP connection to the server. However, if the connection is persistent, or the resources are saved locally or in cache, both events (*connectStart and connectEnd*) will return the same value, equal to the one returned by the *domainLookupEnd*. If the connection to the server fails and the user-agent performs another attempt to reconnect to the server, the *connectStart and connectEnd* will return the values correpondent to the new established connection.

  The instant correspondent to the *connectEnd* includes the Secure Sockets Layer (SSL) handshake time and Socket Secure (SOCKS) authentication.

[1] A non-persistent connection means that the TCP connect session is not the same between the sending and receiving of HTTP requests/responses pairs. That way, for each request/response pair it is established a new TCP connection.

- *secureConnectStart* (TCP block): This event is only applied to HTTPS webpages, and corresponds to the time instant immediately before to the establishment of a new secure connection (handshake).

  If the webpage is not HTTPS, or the user-agent does not have this attribute available, the returned value will be zero.

- *requestStart* (*Request* block): This event marks the time immediately before to the user-agent request for the current document to the server, or to the cache or local resources.

- *responseStart and responseEnd* (*Response* block): This block represents the time interval it takes to the server to provide a response to the previous request.

  The *responseStart* corresponds to the instant immediately before to the reception, by the user-agent, of the first byte of the response. And, the *responseEnd* occurs immediately after the reception of the last byte of the requested document, or immediately after to the server connection close.

- *domLoading* (*Processing* block): This event belongs to the processing block, and returns the instant immediately before to the document change of state to "loading", by the user-agent.

- *domInteractive* (*Processing* block): This event, also part of the processing block, returns the instant immediately before to the document change of state to "interactive", by the user-agent.

- *domContentLoaded* (*Processing* block): This event, from the processing block, represents the time immediately after to the completing of the document loading.

- *domComplete* (*Processing* block): This event returns the time immediately before to the user-agent changes the state of the current document to "complete".

- *loadEventStart and loadEventEnd* (*onLoad* block): These two time events, from the *onLoad* block, mark the trigger of the document load start (*loadEventStart*), and the time when the document has finished loading (*loadEventEnd*).

Since the MDN Performance API returns the same value for the instants that allow the calculation of the DNS lookup time (*domainLookupStart and domainLookupEnd* instant events) and the TCP connection time (*connectStart and connectEnd* instant events), as described above, there was the need to have an alternative to calculate this metrics.

Thus, for both the domain lookup time and connection time to the webserver, the *Python*'s *subprocess* library was used . This library allows to execute a customized *curl* command, in

order to obtain both time intervals, the domain lookup time and the TCP connection time metrics.

In order to obtain the status code of the target webpage and, also, the status code from each of its external resources loaded, it was used a Python library, the *selenium-requests*[68]. This library is an extension for the Selenium Webdriver features that is compatible with the browsers chosen to be used in this solution (*Google Chrome* and *Mozilla Firefox*).

Configuration File

Firstly, the program reads the configuration file, that has the instructions of what tasks will be performed, during the program run. Since that, the configuration file is a JSON file that allows the configuration of: the browser to be used, the webpage URL, the user-agent, screen size (width and height), interactions to be performed (login, logout, scroll down to bottom or scroll down by pixels), the possibility to download the page source along with the external resources, the paths to the directories where the outputs should be saved to and the path to the Firefox binary (for Chrome it is not needed).

In the case of one or more interactions configuration, it is needed to specify the type of interaction and the elements to be clicked, with the *click()* webdriver method, or where to input text, using the *send_keys()* webdriver method. These elements can be selected by their name, id, XPath, class and others, and need to be specified in the JSON configuration file. The elements identifier is found by using, for example, the webdriver *find_element_by_id()* method, in the case the element identifier, specified on the configuration file, corresponds to the element id.

The Table 4.1 presents the fields available on the configuration file and the respective description.

The configuration file has a JSON field, named "*interaction*", that is compound by a boolean field and a JSON dictionary with the interactions to be performed (one or more). The Table 4.2 shows the fields that can be configured on the "*interaction*" field, from the configuration file.

Parameters	**Description**
browser	The browser short name. There are two possibilities available: *Google Chrome* or *Mozilla Firefox*.
timeout	The time in seconds to receive a response (bytes) from server. If server returns no response within timeout seconds, then a error code is raised.
userAgent	The browser user-agent to be emulated. If the field is left empty, the user-agent used is the default from the system where the test is being performed.
screenWidth	The value to set the screen resolution width.
screenHeight	The value to set the screen resolution height.
webpage	The URL of the webpage to be tested.
cacheEnable	Boolean entry to set if the cache is enabled (true) or disabled (false). By now, this field has no function, since the tests only open a browser session, that is, the browser is never closed neither are done several accesses to more than one website.
screenshot	Boolean to set if the test should return screenshots or not.
interaction	JSON to set the interaction(s) that should be performed or if no interaction should be made during this test. (vide Table 4.2)
downloadPage	Boolean value for the option of download the page source plus its external resources.
firefoxPath	Path to where the *Mozilla Firefox* browser is located.
firefoxWebdriverPath	Path to the *Geckodriver* is located.
chromePath	Path to where the *Google Chrome* browser is located.
chromeWebdriverPath	Path to the *ChromeDriver* is located.
pathToOutputsDir	Path to directory where the outputs should be written to.
pathDirToWebpageZip	Path and filename to where the page source and resources should be written to. If empty, the program will not return the webpage and neither its resources.
pathToDebugLog	Path and filename to where the debug log file should be written to.
pathToClientlog	Path and filename to where the client log file should be written to.
pathToOutputValues	Path and filename to where the extracted informations file should be written to.

Table 4.1: Table with entries of the configuration file.

Parameters	**Description**
performInteraction	Boolean value that state if the test should perform interaction(s) with the webpage or not.
interactionType	String of the type of interaction to be performed during the test. The interactions available are: login, logout and scroll down and scroll down by pixel.
interactionTimeout	Value, in seconds, during which the *interactionType* needs to be completed. If the interaction timeouts, it is considered unsuccessful and it is returned an error code to the output files (debug and client logs and to the values file).
username	Username or email or other data used to identify the user account on the webpage.
password	Password string credential matching the *username* specified in the above field.
usernameElem	JSON with element HTML identifier (e.g.: id, class, XPath) key and HTML element label, corresponding to the key, to where the *username* will be inserted.
passwordElem	JSON with element HTML identifier (e.g.: id, class, XPath,...) key and HTML element label that indicates the location where the *password* will be inserted.
loginElemClick	JSON dictionary composed by element(s) HTML identifier(s) key (for example: id, class, name, XPath) and the HTML element(s) label to be clicked, in order to complete the login.
loginElemSuccess	JSON with element HTML identifier (e.g.: id, class, XPath,...) key and HTML element label that should be visible, in order to consider the login action successful. Otherwise, if the element identifier is not visible, it is considered that the action was not successfully performed.
logoutElem	JSON with element(s) HTML identifier (e.g.: id, class, XPath,...) key(s) and HTML element(s) label to be clicked, in order to logout from the webpage.
logoutElemSuccess	JSON with element HTML identifier (e.g.: id, class, XPath,...) key and HTML element label that should be visible, in order to consider the logout action successful. Otherwise, if the element identifier is not visible, it is considered that the action was not successfully performed.
scrollBottomElem	JSON with element(s) HTML identifier (e.g.: id, class, XPath,...) key(s) and HTML element(s) label to be found, in order to know if the page bottom was reached.
pixelsNumber	Integer value that corresponds to the pixels number to scroll the page by.
cookieButtonClick	JSON with cookie element HTML identifier (e.g.: id, class, XPath,...) key and HTML element label to be clicked if the page of login or logout has a pop-up that needs to be closed, in order to proceed with the interaction login action. It is an optional field, as it can be left empty and only used when needed.

Table 4.2: Table with entries of the JSON *interaction* field from the configuration file.

Following there are two examples of configuration files for the four types of interactions available: login, logout, scroll by pixel and scroll down.

```
{
  "browser" : "firefox",
  "timeout" : 25,
  "userAgent" : "",
  "screenWidth" : 1170,
  "screenHeight"  : 890,
  "webpage" : "https://www.facebook.com/",
  "cacheEnable" : false,
  "screenshot" : true,
  "interaction" : {"performInteraction" : true,
          "actions" :
          [{
            "interactionType" : "login",
            "interactionTimeout"  : 25,
            "usernameElem"  : {"id" : "email"},
            "passwordElem"  : {"id" : "pass"},
            "username"  : "chophipaco@inaby.com",
            "password"  : "chophipaco@inaby.com",
            "loginElemClick"  : {"id" :"loginbutton"},
            "loginElemSuccess"  : {"id" : "userNavigationLabel"},
            "cookieButtonClick" : ""
          }, {
            "interactionType" : "logout",
            "interactionTimeout"  : 25,
            "logoutElem"  : [{"id" : "userNavigationLabel"},{"x_path": "//li
    [12]/a/span/span"}],
            "logoutElemSuccess" : {"id" :"loginbutton"},
            "cookieButtonClick" : ""
          }]},
  "downloadPage"      :   false,
  "firefoxBinaryPath" : "/usr/bin/firefox",
  "pathToOutputsDir"  : "/opt/QoE/outputs/",
  "pathDirToWebpageZip" : "/opt/QoE/outputs/webpage_content.zip",
  "pathToDebugLog"  : "/opt/QoE/outputs/debug_log.log",
  "pathToClientlog" : "/opt/QoE/outputs/client_log.log",
  "pathToOutputValues"  : "/opt/QoE/outputs/values.json"
}
```

Listing 4.1: Example of a test configuration file (JSON) with login and logout interactions.

The Listing 4.1 is an example of a JSON configuration file to perform an access to the *Facebook* webpage, with further login and logout interactions scheduled, with the purpose of

illustrate the previous explanations.

```
{
"interaction" : {"performInteraction" : false,
        "actions" :
        [{
          "interactionType" : "scrollbypixel",
          "interactionTimeout"   : 25,
          "pixelsNumber"     : 100
        },{
          "interactionType" : "scrolldown",
          "interactionTimeout"   : 25,
          "scrollBottomElem"   : {"class":"footer"}
        }]},
}
```

Listing 4.2: Example of a test configuration file (JSON) with scroll by pixel and scroll to the bottom interactions.

This Listing 4.2 illustrates how the interactions of scroll down to the webpage's bottom and the scroll by pixel are configured, on the configuration file.

The Listing 4.3 depicts a *Python* code snippet, with the JS executed by the webdriver, in order to perform a scroll down by a certain amount of pixels and a scroll down to the bottom, respectively. In this case, it is done the execution of a JS, by the webdriver, to scroll down to the page height (bottom).

```
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as
    ExpectedConditions
from selenium.webdriver.common.by import By

...

self.driver.execute_script("window.scrollTo(0,document.body.scrollHeight);")
wait = WebDriverWait(self.driver, INTERACTION_TIMEOUT)
wait.until(ExpectedConditions.visibility_of_element_located((By.ID, val)))
```

Listing 4.3: Code snippet to perform a scroll down to the bottom of the webpage.

In the Listing 4.3 it is illustrated what was used to detect elements on the page and how the timeout waiting is done. So, in order to wait till the interaction is completed, it was introduced a timeout waiting, that along with the *selenium.webdriver.support.ui* library it is allowed to have a waiting condition that is non-blocking, till the event that is wanted occurs. In this snippet case, the waiting condition duration is till it reaches the timeout

specified, or till the element id (this is done with the help of the *selenium.webdriver.common.by* library) is detected visible on the webpage, this by using the *expected_conditions* from the *selenium.webdriver.support* library. For the scroll by pixel interaction, it is also executed a JS: *"window.scrollBy(0, pixels_to_scroll)"*.

Headless Display

In order to create a headless display, besides having the Xvfb installed, the *PyVirtualDisplay* library, which is a wrapper for the Xvfb, was imported. Then, to create the headless display it is needed to specify some of this class parameters: the backend and the screen resolution (width and height), although there are others that could be specified, such as, the color depth, background color, or the visibility of the screen. Although this latter (screen visibility) is an alternative to specify the backend, because if it is set to *True or 1*, then the *Xephyr* tool is used , else if it is set to *False or 0*, is the Xvfb that is used. And, finally, is it started the display by using the *start()* method.

The Listing 4.4 is a code snippet with the method example of how it is created and started the headless display. It is used the Xvfb backend and the screen size used is the one specified by the *screenWidth* and *screenHeight* fields of the configuration file.

```
1 from pyvirtualdisplay import Display
2
3 # Method to create headless display
4 def create_headless_display(screen_width, screen_height):
5
6   #Display with no interface
7   display = Display(backend='xvfb', size=(screen_width, screen_height))
8   display.start()
```

Listing 4.4: Code snippet of the creation and starting of a headless display using Xvfb.

Webdrivers

After the headless display is created and started, the next step is to initialize the webdrivers (vide Listings 4.5 and 4.6). It is needed to import the *Selenium Webdriver API* and then use its *Chrome* and *Firefox* classes.

```
1 from selenium import webdriver
2
3 ...
4
```

```
# if not defined: use user-agent of the device performing the test
if USER_AGENT == "":
  driver = webdriver.Chrome()

# else if the user agent is defined: setup the specified user-agent
else:
  chrm_opts = webdriver.ChromeOptions()
  chrm_opts.add_argument('--user-agent=' + USER_AGENT)
  driver = webdriver.Chrome(chrome_options=chrm_opts)
```

Listing 4.5: Code snippet with example of the Google Chrome webdriver initialization.

In the initialization of the *Google Chrome* webdriver (vide Listing 4.5), if the user-agent field was left empty on the configuration file, then the user-agent used is the default, which is the user-agent with the characteristics of the device and browser in use.

Otherwise, if a user-agent was specified in the configuration file, then that is the one assumed on the test. In this case, in order to set the user-agent specified on the configuration file, it was used the *ChromeOptions* class, whose *add_argument()* method enables the possibility of adding this characteristic to the webdriver initialization as an input parameter.

```
from selenium import webdriver
from selenium.webdriver.common.desired_capabilities import DesiredCapabilities

...

dc = DesiredCapabilities.FIREFOX
dc['binary'] = FIREFOX_BINARY_LOCATION

# if not defined: use user-agent of the machine performing the test
if USER_AGENT == "":
  driver = webdriver.Firefox(capabilities=dc)

# else if the user agent is defined: setup the specified user-agent
else:
  profile = webdriver.FirefoxProfile()
  profile.set_preference("general.useragent.override", USER_AGENT)
  driver = webdriver.Firefox(capabilities=dc, firefox_profile=profile)
```

Listing 4.6: Code snippet with example of the Mozilla Firefox webdriver initialization.

Similarly, the *Mozilla Firefox* webdriver uses the default user-agent of the device and browser being used, in case of no user-agent was specified in the configuration file (vide Listing 4.6).

Otherwise, the *FirefoxProfile* class is used, in order to allow the setup of the user-agent specified on the configuration file, by using the *set_preference()* method. In the case of this

webdriver, opposing to the *Google Chrome* webdriver, there is the need to define the Firefox binary location, which is setup by using the *DesiredCapabilities* class, that is further used as an input parameter on the webdriver's inititalization (vide Listing 4.6). In the case of the *Google Chrome* webdriver, the path to the binary does not need specification (vide Listing 4.5).

Outputs

Webpage's Status Code

In order to obtain the status code of the webpage and of its external resources, it was used the *selenium-requests Pyhton* library [68] (vide Listing 4.7).

```
from seleniumrequests import Chrome
from seleniumrequests import Firefox


# Method to obtain the status code of a webpage or external resource
# Args:
#   - browser_name: Name of the browser (firefox or chrome)
#   - url: URL of the webpage or resource
def get_status_code(browser_name, url):
  #var init
  status_code = None

  if browser_name == 'firefox':
    status_code = Firefox().request('GET', url)

  elif browser_name == 'chrome':
    status_code = Chrome().request('GET', url)

  # Apply filter to the string returned by the request() method
  #to return only the status code number.
  status_code = filter(str.isdigit, str(response))
  return status_code
```

Listing 4.7: Code snippet example of how the status code was obtained.

The Listing 4.7 shows how it is returned the status code of an URL. For that purpose, the *selenium-requests* library was used, and both Webdrivers were imported (*Google Chrome* and *Mozilla Firefox* webdrivers).

Then, depending on the browser specified on the configuration file, the webdriver is selected by the conditional *IF* statements, and it is made a HTTP *GET request* to the webpage

or external resource URL. Then, the response is filtered, in order to obtain only the code of the URL status.

Metrics Extracted And Other Informations

As referred on section 4.1, of the "*SmartBrowsing Solution*" chapter, for both the domain lookup time and connection time to the webserver, it was used the *Python*'s *subprocess* library. This library allowed to execute a customized *curl* command, in order to obtain both time intervals, the domain lookup time and the TCP connection time metrics.

The Listing 4.8 depicts the extraction of the DNS lookup time, and the Listing 4.9 shows the extraction of the TCP connection time.

For the *DNS Lookup Time*:

```
1  import subprocess
2
3  # Method to obtind the domain lookup time
4  # in seconds
5  # Args:
6  #   - page URL
7  def get_domain_lookup_time(self, url):
8      try:
9          cmd = subprocess.Popen(["curl", "-s", "-w", '%{time_namelookup}', "-o",
       "/dev/null", "" + str(url) + ""],
10                                 stdout=subprocess.PIPE)
11     except Exception, ex:
12         print str(ex)
13         return
14
15     lookup_time = cmd.communicate()[0]
16     return float(lookup_time.replace(",","."))
```

Listing 4.8: Code snippet example of the *DNS Lookup Time* extraction.

For the *TCP Connection Time*:

```
import subprocess

# Method to obtind the domain lookup time
# in seconds
# Args:
#   - page URL
def get_tcp_connection_time(self, url):
    try:
        cmd = subprocess.Popen(["curl", "-s", "-w", '%{time_connect}', "-o", "/dev/null", "" + str(url) + ""],
                               stdout=subprocess.PIPE)
    except Exception, ex:
        print str(ex)
        return

    tcp_time = cmd.communicate()[0]
    return float(tcp_time.replace(",","."))
```

Listing 4.9: Code snippet example of the *TCP Connection Time* extraction.

To calculate both these metrics, the *Popen* class method, from the *subprocess* library, was used. The *Popen* accepts, as input, an array of arguments that, in this case, compounds the command *curl*, and the output's location, that in this case is the command's output result, which is sent to a new pipe opened to the standard stream (*subprocess.PIPE*).

Then, the data sent to the *stdout* is read by the *communicate()* method, and then truncated to the first element of the command response, which corresponds to the value of the metric being extracted, the *DNS Lookup Time* or the *TCP Connection Time*.

The command *curl* is customized in both the methods of these code snippets examples, in order to output the value of the intended metric: the *"%time_namelookup"* argument for the DNS time and the *"%time_connect"* for the TCP time. The other command arguments, apart from the webpage's URL to be measured, are:

- *"-s" (–silent)*: is the silent mode, meaning that the curl command will not output a progressing bar.

- *"-w" (–write-out FORMAT)*: this argument states the format of the command resulting output. That, in this case, will be the *"%time_namelookup"* or *"%time_connect"* value, depending on the time requested on the command.

- *"-o"* (-output FILE): this tells to write the output to the path instead of outputting to the *stdout*, in this case, the *FILE* is the *"/dev/null"* path. This *"/dev/null"* path represents a null device on the UNIX operating systems, that accepts all data sent to it without storing it. In this situation, the content that is received and then excluded

from the command output is correspondent to the webpage's HTML of URL requested on the command.

In order to extract the other time metrics, the time events from the MDN Performance API (on section 4.1 of the *"SmartBrowsing Solution"* chapter) were used. In order to extract those metrics it is needed to execute a JS with *Selenium Webdriver*. In the Listing 4.10 is shown the request for the *navigationStart* event timestamp, which is further added to a dictionary, for later use in metric calculation.

```
x['navigationStart'] = driver.execute_script("return window.performance.timing
    .navigationStart;")
```

Listing 4.10: Code snippet example of the extraction of the *navigationStart* event, from MDN Performance API.

After the extraction of all webpage's events timings, returned by the API, the metrics are calculated, besides the TCP connection time and DNS lookup time, already referred upper. Those metrics and its calculation equations are the following:

$$RequestTime = (responseStart - requestStart) * 0.001(seconds) \tag{4.1}$$

Accordingly with the Mozilla documentation[69], the request and response times could be given by subtracting the *responseEnd* to the *requestStart* event. That way, both the request and response times can be calculated separately, as it was assumed on this solution, based on the W3C processing model (referred on section 4.1 of the *"SmartBrowsing Solution"* chapter).

This Equation 4.1 returns the time interval of the URL request to the webpage hostage server, which is given by the elapsed time at which the *requestStart* event is triggered and the time at which the *responseStart* is triggered. And, since the event timings are returned in milliseconds, the multiplication concerns to seconds conversion.

$$ResponseTime = (responseEnd–responseStart) * 0.001(seconds) \tag{4.2}$$

The Equation 4.2 returns the time interval of the response from server, which is given by the elapsed time at which the *responseStart* event is triggered and the time at which the *responseEnd* is triggered.

$$InitialReponseTxTime = (responseStart–domainLookupStart) * 0.001(seconds) \tag{4.3}$$

In order to return a metric that gives the total network delay till the start of the response transmission (Tx), it was considered that it was given by the elapsed time at which the

domainLookupStart was triggered, which is relative to the start of the URL translation, till the time at which the *responseStart* is triggered, i.e., the time at which the server started to send the response (Equation 4.3).

$$TotalResponseTxTime = (responseEnd–fetchStart) * 0.001(seconds) \quad (4.4)$$

The Equation 4.4 allows to calculate the webpage and its resources latency, accordingly with the W3C model recommendation, which includes all the process, from the entering of the webpage URL till the end of the response receiving.

$$TTFB = (responseStart–navigationStart) * 0.001(seconds) \quad (4.5)$$

The Equation 4.5 gives the total time taken till the first byte was received (*responseStart*), since the very beginning of the navigation start, i.e., since the time it was entered the URL on the browser.

$$PageLoadTime = (loadEventEnd–navigationStart) * 0.001(seconds) \quad (4.6)$$

Once again, accordingly with the *Mozilla* documentation for the API, it was considered that the time it takes to the page to load (Equation 4.6) is the time interval between the beginning of the navigation and the end of the webpage loading on browser.

$$HTMLProcessingTime = (domComplete–domLoading) * 0.001(seconds) \quad (4.7)$$

The render time (Equation 4.7) is also considered by the Mozilla documentation to be the time elapsed from the DOM loading start, till the time the DOM has completed its loading. Although that, for the tests performed and presented on chapter 5, this equation was not used. Following, it is presented the equation used (Equation 4.8), due to the fact of being more embracing in terms of the webpage processing events occurred on the client-side.

$$Client - sideProcessingTime = (loadEventEnd–domLoading) * 0.001(seconds) \quad (4.8)$$

The Equation 4.8 is similar to the previous one (Equation 4.7), due to the same start time considered. However, this equation measures the total time it takes to the client-side to process and load the webpage, by assuming the time elapsed at which the DOM starts its loading till the loading ends.

$$TimeToWebpageInteractivity = (domInteractive–navigationStart) * 0.001(seconds) \quad (4.9)$$

This Equation 4.9 indicates the time interval it took to the webpage to become interactive, that is, to allow the user to interact with it. Although that, it does not give information about the state of the webpage resources loading, for example, the images, CSS files and others. Anyhow, it can be a good indicative of the page responsiveness.

Furthermore, another metric was extracted, the Time To The First Paint (TTFP) (Equation 4.10). However, this one is only available for tests that use *Google Chrome* browser, due to two time instants (*firstPaintTime* and *startLoadTime*) that are only returned by this browser.

$$TTFP = (firstPaintTime - startLoadTime) * 0.001(seconds) \tag{4.10}$$

This Equation 4.10 calculates the time interval needed to the appearance of the first paint in the webpage, i.e., the first visual element. Thereby, it is considered an important metric as the users tend to appreciate the appearance of visual elements, as it is indicates that the loading of the webpage is in progress. This is more satisfying for the user perception of the webpage responsiveness, rather than waiting, without any clue of what is happening in the background, and then having all the webpage loaded and visible at the same time on the screen.

Furthermore, besides all the previous referred metrics measured after the webpage access, the API also returns the list of external resources loaded after the webpage access. For each of the external resources loaded, the previous metrics are calculated, and the status code is obtained. These results are obtained using the same ways as the ones used to obtain the same informations for the webpage.

The Listing 4.11 shows the JS executed, in order to obtain a JSON with the resources loaded, along with the previously referred time events associated to each of them, the identification *href* HTML attribute and the bytes transferred. Further, the previous equations are performed, and all the informations are then written to the output files.

```
1 json_rsc_list = driver.execute_script("return JSON.stringify(window.performance
    .getEntriesByType(\"resource\"));")
```

Listing 4.11: Code snippet example of how the webpage resources are extracted.

After the execution of the code depicted on Listing 4.11, it is returned a list of external resources (belonging to the webpage content). This list is compound by the resources loaded till the webpage loading is finished. In case of heavy webpages, i.e., webpages compound by many resources and/or with a big height, the elements returned are the ones loaded till the load spinning wheel symbol stops, meaning that not all the webpage resources are returned on the list (only the loaded ones are). If a scroll down action is performed and the code (illustrated on Listing 4.11) runs once more, the list returned will include more resources,

that correspond to the ones which loading was triggered by the scroll action. If this situation occurs, the *SmartBrowsing* solution returns a new list with the external resources loaded after the scroll action.

Besides those informations, are also returned for each external resource, the correspondent transferred bytes (the resource bytes length plus the bytes of the header, that is, the transferred bytes are correspondent to the packet(s) length(s) that brings the resource). Then, each time new external resources are loaded, a sum of all the bytes of the loaded resources is performed, which are also written to the outputs files, in a JSON format.

User Interactions

In terms of the interactions, the solution measures the time it takes to perform the login, logout and scroll to the webpage bottom. For that measuring, the *time Python* library is used, and the measure is done by saving the time at which the action starts, when the first action that leads to the login/logout/scroll down to the bottom is triggered, till the time the element that indicates the success of the interaction is loaded and the element is visible on the page.

The time taken to perform each of the interactions: login, logout and scroll down to the bottom is written to the output files.

Screen Captures

In the beginning of the access, during an interaction and at the end of the navigation test, screen captures are taken and saved to the directory path, specified on the configuration file.

These screen captures serve has a proof that the test run within the expected, or in case of some abnormal situation (for example, an unsuccessful interaction or an unreachable webpage) it helps in the process of debugging, as the test runs on a headless environment and without human intervention.

The Listing 4.12 shows how the *Selenium Webdriver* tool performs screen captures, using a Xvfb display (headless).

```
1 driver.save_screenshot("screenshot_name.png")
```

Listing 4.12: Code snippet example of how screen captures are performed.

Webpage Download

Finally, the download of the page source and its external resources is also possible in this solution. The option to download the page source and its resources must be specified in the configuration, on field *"downloadPage"* (vide Table 4.1). The webpage files are downloaded into a compressed zip folder.

This can be useful if the webpage is opened in a browser in offline mode (no Internet connection and by clicking on the HTML file and having the other files in the same folder) with the purpose of viewing how was the webpage after the test ending (for example, to check if every resource hosted on the webpage webserver was loaded or not). This can work as an addition to the informations written into the output files that can confirm the informations on them.

In order to extract the page source and external resources, the webdriver's *page_source()* method was used. This method allows to extract the webpage HTML. After that, the extracted HTML is written into a file, in the *utf-8* encoding format. After that, the *BeautifulSoup Python* library is used in order to parse the HTML document, and search for the external resources that will, also, be downloaded. To find the webpage external resources, the *find_all()* method, of the *BeautifulSoup Python* library, is used, in order to search for all the *"src"* tags in the HTML file. After finding a *"src"* tag, with the use of *urlparse Python* library, the webpage URL is joined with the *"src"* tag value, by using the *urljoin()* method. After that, the resource is copied to the path specified on the configuration file, by using the *urlretrieve()* method, from the *urllib Python* library. Finally, all the files downloaded are zipped into a folder, by using the *zipfile Python* library.

The JSON output file, where all the information values and list of resources extracted are stored, have the fields presented on Table 4.3:

Parameters	**Description**
generalInfo	JSON with the informations configured: URL, screen size, user-agent, status code of the webpage, and the browser in use.
webPageTimings	JSON with the time intervals measured for the webpage.
pageInteractive	Time interval to the webpage status turn to *interactive*.
requestTime	Time interval of the request.
connectionTime	Time interval of the TCP connection with the webserver (webpage host).
pageLoadTime	Time interval correspondent to the webpage loading.
lookupTime	Time it took to perform the DNS lookup.
score	Score value, in a range of 1 (Bad) to 5 (Excellent), indicating the score of the webpage in terms of page load time. It was based on the premise that a webpage should load within two seconds.
TTFB	Time interval to the arriving of the first byte of the response from the webserver.
clientSideProcessingTime	Time interval correspondent to the processing on the client-side.
latency	Time interval correspondent to the latency.
webPageResourcesTimings	JSON array of JSONs with the loaded resources informations extracted for each (the referred above plus the resource identification *href*).
loginTime	Time it took to perform and complete successfully the login interaction.
logoutTime	Time it took to perform and complete successfully the logout interaction.
loginTimeAndPageLoadTime	Time it took to perform and complete successfully the login interaction plus the page load time of the page loaded after the login.
logoutTimeAndPageLoadTime	Time it took to perform and complete successfully the logout interaction plus the page load time of the page loaded after the logout.
scrollTime	Time it took to perform and complete the scroll down to the bottom interaction.
resourceTransferSize	The sum of the total bytes transferred of the resources loaded. This includes the size of the header and the payload of the responses.

Table 4.3: Table with entry fields of the informations extracted to the output file.

Conclusion

This chapter aim was to provide all the details of the implementation of the *SmartBrowsing* solution and, also, its deployment requirements, in order to give all the information about the way the solution was implemented. The information provided were: the technologies adopted to the solution implementation; The configuration and output files, along with illustrative code snippets for each approach. The system and deployment requirements to deploy the *SmartBrowsing* application can be consulted in the Appendix chapter.

5

Evaluation And Analysis

In this chapter, it will be made the analysis of the tests results performed in order to check the solution capabilities and analyze the the solution performance.

The tests performed were based on the metrics extracted, on different webpages (static and dynamic) and under none, one or complex (more than one) interactions on the webpage.

The application was analyzed in terms of performance in two computers (available at the time of the solution deployment) with different characteristics, essentially with focus on the CPU and RAM differences. It was also analyzed the differences in-between tests using two different browsers (the ones available of use on the solution: *Google Chrome* and *Mozilla Firefox*).

Afterwards, were made several test samples in order to check whether the render of a webpage would differ depending on its type: static or dynamic. Also, the page load time for each of these two types were analyzed, to inquire if there was a relationship between both, and how much the render time would affect the page load time.

To examine other metrics, like the DNS lookup, the TCP connection, the request and the response times, there were performed tests in parallel with packet capturing, by using the *tcpdump*[70] tool for the packet capture, and the *Wireshark*[71] to open the files and analyze them. These tests aim was to check the reliability of these metrics, extracted by the solution, and the time intervals extracted on the captures.

The other two metrics, *Network Delay Time* and the *TTFB* are included in the *request* time metric, which is analyzed along with the *response* time, on the scenarios further described in this chapter. This way, there were made no tests to those metrics (*Network Delay Time* and the *TTFB*) specifically, due to the fact that they are part of the time intervals of the *request* and *response* metrics formulas 4.5.2.

The tests described in this chapter were made without using cache, since each test/sample was executed in a new browser session in anonymous/private mode, with a single request for

each URL, and cache options were disabled for the *Selenium Webdriver* (as referred in chapter 3).

Program Execution Time

In these tests was examined the impact of using devices with different capabilities, focusing on the RAM and processing (CPU) characteristics. It was used a computer with 4 GigaByte (GB) of RAM along with an *Intel Core i3* CPU (PC Low), and another with 8 GB of RAM and an *Intel Core i5* CPU (PC High). The tests performed used both browsers, *Google Chrome* and *Mozilla Firefox*, to access a dynamic page, without any interactions scheduled. For this purpose was used the webpage `https://www.ua.pt/`, from the University of Aveiro.

For this scenario, were made fifty tests under the conditions referred above (for each browser), and the metrics returned from the tests executions annotated for further analysis.

The Figure 5.1 shows the tests execution timings (Y axis), in seconds, for each one of the fifty tests (X axis), executed on both browsers. This tests were made on a computer with 4 GB of RAM and an *Intel Core i3* CPU.

The conclusions that can be extracted from the observation of the results obtained are: that the tests performed with the *Google Chrome* browser, in all of the samples, presents a lower execution time than the tests performed on the *Mozilla Firefox* browser.

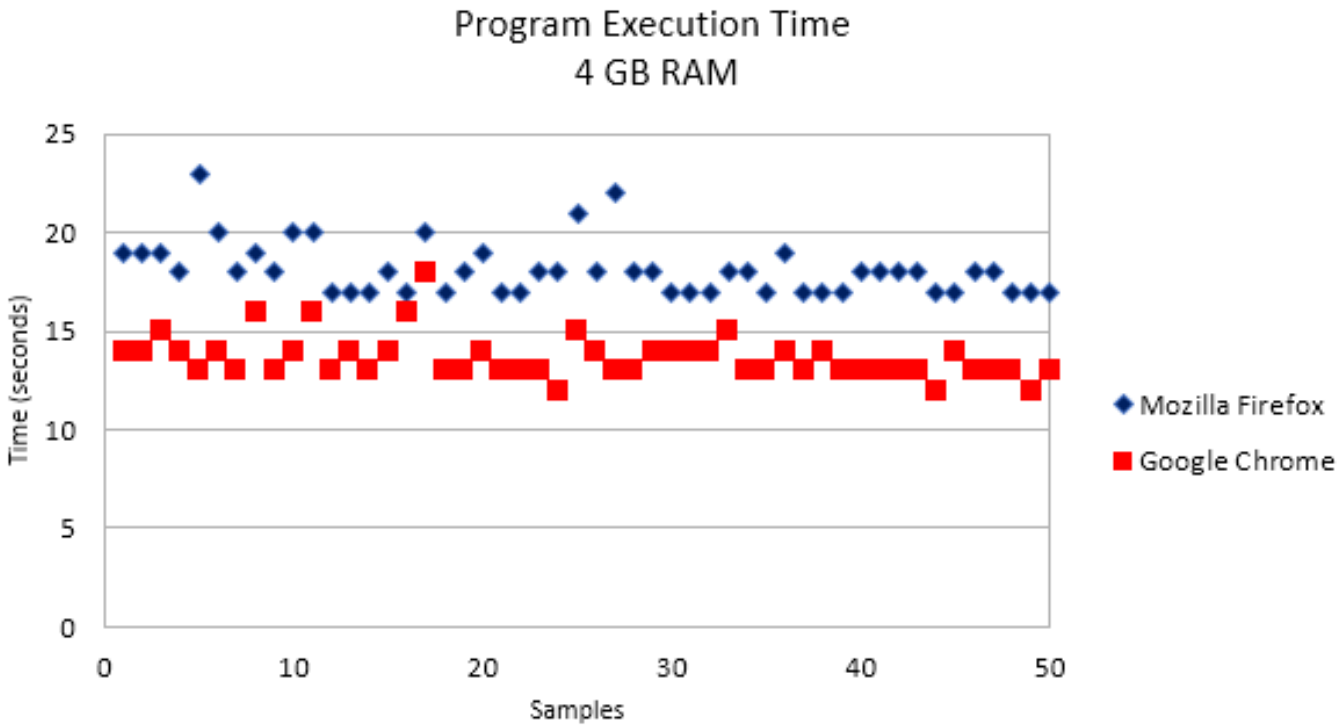


Figure 5.1: Program execution time test (with no interactions) using *Google Chrome* and *Mozilla Firefox* browsers, on a 4 GB RAM and *Intel Core i3 device*.

The difference between the execution times of each browser presents a gap of, approximately, four and a half seconds between them.

Browser	*Google Chrome*	*Mozilla Firefox*
Average (seconds)	13,63	18,14
Maximum (seconds)	18	23
Minimum (seconds)	12	17
Standard Deviation	1,12	1,36
Confidence Interval (Confidence Level = 95%)	[13, 32; 13, 94]	[17, 78; 18, 50]

Table 5.1: Table with the analysis of the "program execution time" fifty samples, for PC Low tests.

Table 5.1 contains the measured values for the average, maximum and minimum for each of the browsers considered. And also, presents the standard deviation and confidence interval, with a confidence of 95 percent. By calculating the coefficient of variation (given by the division of the standard deviation by the mean):

- For *Google Chrome*, it is obtained a coefficient of 0,082, which is inferior to one, indicating a low dispersion between the samples of program execution time.

- For *Mozilla Firefox*, it is obtained a coefficient of 0,075, which is inferior to one, indicating a low dispersion between the samples of program execution time, as well.

The same tests were performed on the PC High. Figure 5.2 shows that the difference between the execution times for each browser decreased, and it is less intuitive to take conclusions. Still, from the observation of the chart, it can be affirmed that the *Google Chrome* test samples present lower execution times than the *Mozilla Firefox*. This is similar to what was concluded for the device with lower capabilities.

In this case, the values for the program execution, in both browsers, are more similar than the ones observed for the PC Low (vide Figure 5.1). In addition, it is coherent that the PC High does not present a big difference between the results obtained for each browser. In fact, a device with higher capabilities has a better performance executing the running processes. Also, a PC with more RAM presents less visible differences between the navigation in each of the browsers considered. Consequently, the impact on the program execution time is lower.

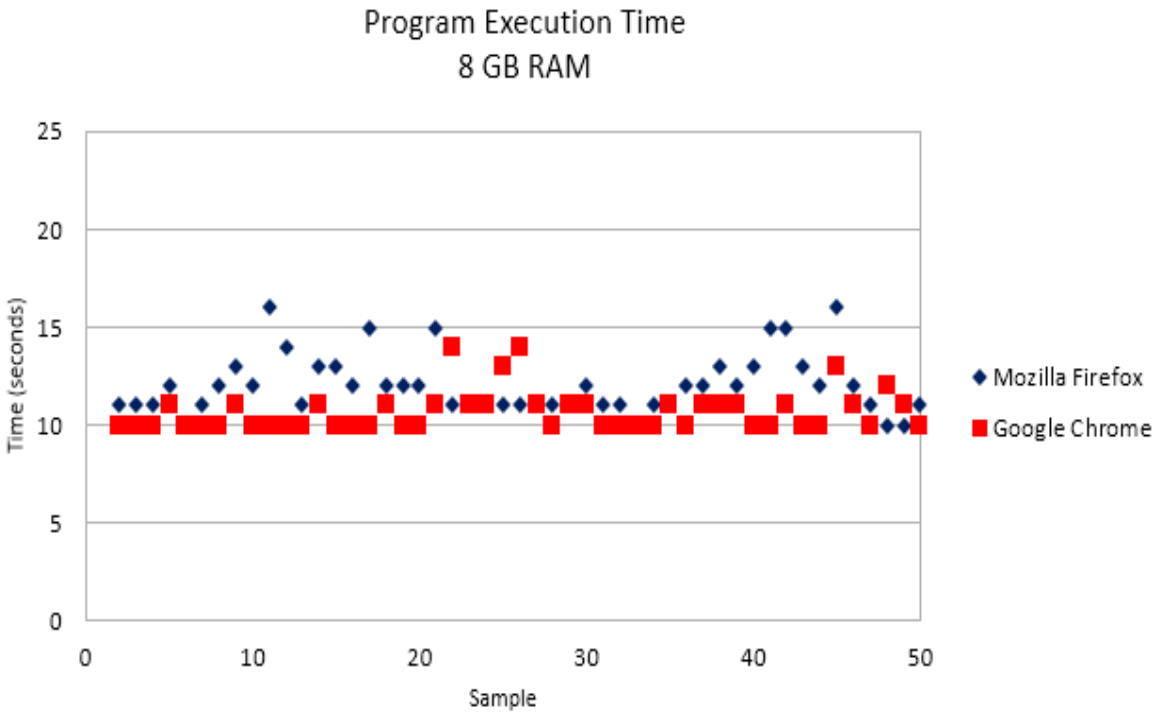


Figure 5.2: Program execution time test (with no interactions) using *Google Chrome* and *Mozilla Firefox* browsers, on a 8 GB RAM and *Intel Core i5* device.

Table 5.2 shows the average, maximum and minimum values in seconds, corresponding to the data presented in Figure 5.2. And also, presents the standard deviation and confidence interval values, with a confidence of 95 percent. By calculating the coefficient of variation:

- For *Google Chrome*, it is obtained a coefficient of 0,094, which is inferior to one, indicating a low dispersion between the samples of program execution time.
- For *Mozilla Firefox*, it is obtained a coefficient of 0,126, which is inferior to one, indicating a low dispersion between the samples of program execution time, as well.

Browser	*Google Chrome*	*Mozilla Firefox*
Average (seconds)	10,68	12,02
Maximum (seconds)	14	26
Minimum (seconds)	10	10
Standard Deviation	1,00	1,52
Confidence Interval (Confidence Level = 95%)	$[10,40; 10,96]$	$[11,60; 12,44]$

Table 5.2: Table with the analysis of the "program execution time" fifity samples, for PC High tests.

Table 5.2 reveals that the gap between the two browsers execution times is, approximately, of one point three seconds.

To conclude, from the data obtained, it can be affirmed that the device running the *SmartBrowsing* solution influences the time that it takes to execute a test. A device with less memory and less processing capabilities increase the time of execution of the test. It can also be concluded that the tests which use the *Mozilla Firefox* browser take, in average, more time to execute than the ones made using the *Google Chrome* browser. And, also, the *Mozilla Firefox* browser presents a bigger difference between the average values of the lower capability device and the higher capability device, of, approximately, six seconds (18,14 - 12,02 = 6,12). While the *Google Chrome* browser presents a difference between the average values of each device of, approximately, three seconds (13,63 - 10,68 = 2,95). In summary, the browser and the device used have impact on the *SmartBrowsing* execution performance.

Page Load Time

In this section, the same two devices and the same webpage, described on the section 5.1, were used. In this case with the purpose of taking conclusions about the influence on the page load time, and consequently, on the user QoE perceived when loading the page.

Firstly, considering the PC Low, in the sequence of the tests made on the section 5.1, were extracted the page load times for each browser, *Google Chrome* and *Mozilla Firefox*, in a total of fifty tests per browser.

Figure 5.3 shows the page load time, in seconds, for the fifty tests made for both browsers. It can be observed that the page load times measured in each browser are not conclusive of a distinguishing difference.

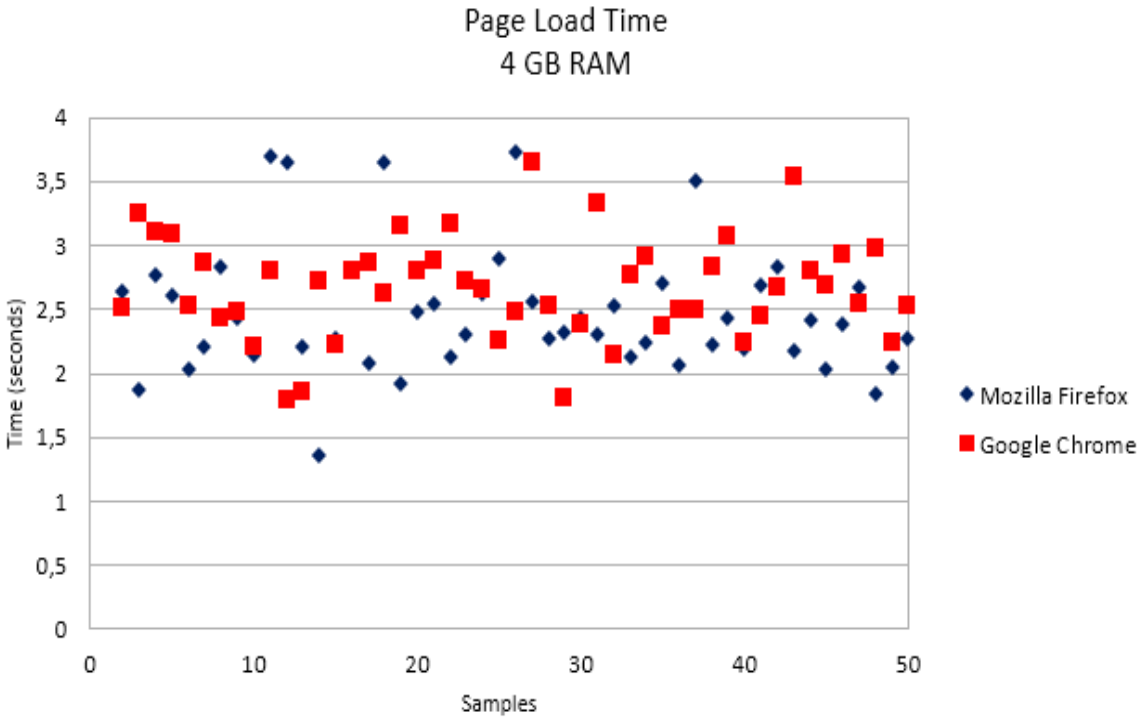


Figure 5.3: Page Load Time (with no interactions) using *Google Chrome* and *Mozilla Firefox* browsers, on a 4 GB RAM and *Intel Core i3* device.

Table 5.3 shows the average, maximum and minimum values, in seconds, for the set of fifty samples. And also, presents the standard deviation and confidence interval values, with a confidence of 95 percent. By calculating the coefficient of variation:

- For *Google Chrome*, it is obtained a coefficient of 0,149, which is inferior to one, indicating a low dispersion between the samples of program execution time.
- For *Mozilla Firefox*, it is obtained a coefficient of 0,202, which is inferior to one, indicating a low dispersion between the samples of program execution time, as well.

In fact, the difference of page load times between each browser is not significant and the range of values is very similar.

Browser	*Google Chrome*	*Mozilla Firefox*
Average (seconds)	2,68	2,47
Maximum (seconds)	3,65	3,72
Minimum (seconds)	1,79	1,36
Standard Deviation	0,4	0,5
Confidence Interval (Confidence Level = 95%)	$[2,57; 2,79]$	$[2,33; 2,61]$

Table 5.3: Table with the analysis of the "page load time" fifty samples, for PC Low tests.

Considering the PC High, in the sequence of the tests made in section 5.1, it extracted the page load times for each browser in a total of fifty tests per each.

In the Figure 5.4, it can be observed that the *Google Chrome* presents a shorter range of values, and also more stable values of page load timings than the *Mozilla Firefox*, that presents a wider range of values for the page load time (higher difference between the maximum and minimum points).

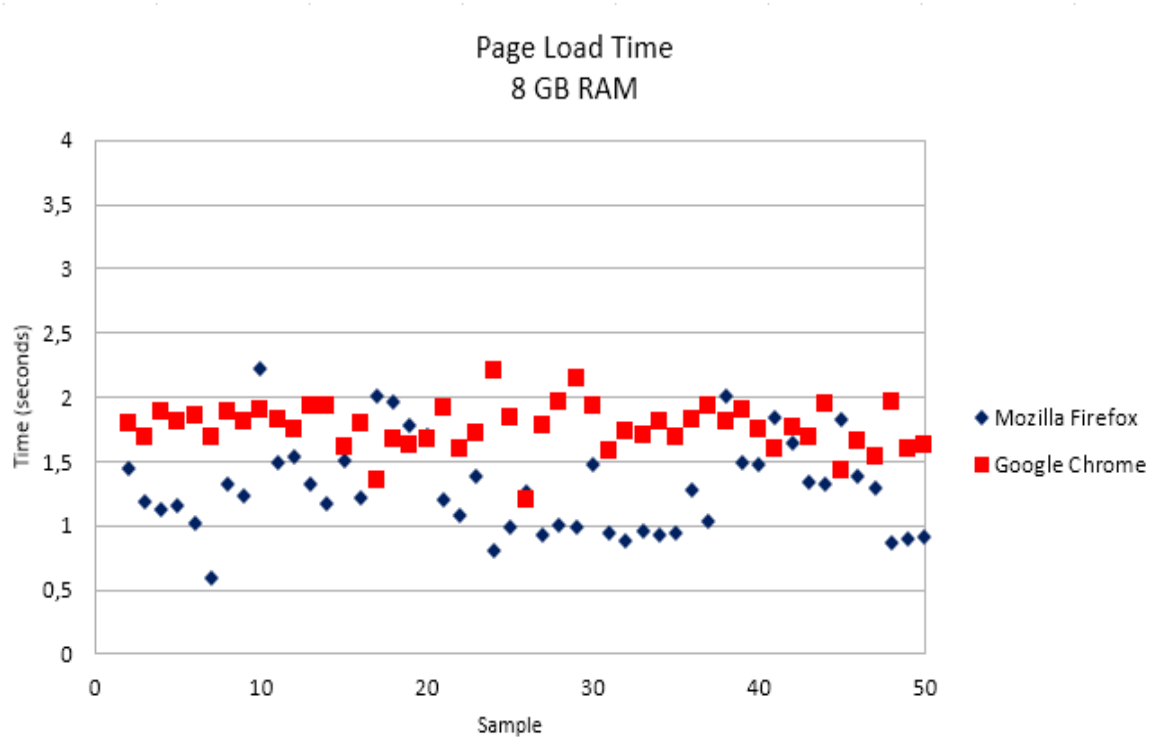


Figure 5.4: Page Load Time (with no interactions) using *Google Chrome* and *Mozilla Firefox* browsers, on a 8 GB RAM and *Intel Core i5* device.

Table 5.4 show the values of average, maximum and minimum for the set of fifty samples. And also, presents the standard deviation and confidence interval values, with a confidence

of 95 percent. By calculating the coefficient of variation:

- For *Google Chrome*, it is obtained a coefficient of 0,102, which is inferior to one, indicating a low dispersion between the samples of program execution time.
- For *Mozilla Firefox*, it is obtained a coefficient of 0,279, which is inferior to one, indicating a low dispersion between the samples of program execution time, as well.

It can be concluded that the difference of page load time between each browser is of approximately, two seconds. *Google Chrome* has the higher page load times.

Browser	*Google Chrome*	*Mozilla Firefox*
Average (seconds)	1,77	1,29
Maximum (seconds)	2,21	2,22
Minimum (seconds)	1,2	0,59
Standard Deviation	0,18	0,36
Confidence Interval (Confidence Level = 95%)	$[1,72; 1,82]$	$[1,19; 1,39]$

Table 5.4: Table with the analysis of the "page load time" fifty samples, for PC High tests.

The conclusion to be taken from the observance of this data is that the device capabilities also impact on the way of how the webpage loading is perceived by the user and, consequently, on the QoE. As such, the device with lower capabilities (PC Low) has increased values of page load time, but without a perceptible difference in terms of the browser used, as they have similar page load timings. Otherwise, the device with higher capabilities (PC High) present a lower range of page load timings and a tenuous difference between the times obtained for each browser in use, by Figure 5.4 observance. The difference between the page load times for each device is of, approximately, one second (for *Google Chrome*: 2,68 - 1,77 = 0,91 seconds; for *Mozilla Firefox*: 2,47 - 1,29 = 1,18 seconds).

Static versus Dynamic Webpages

In this section, it will be analyzed the impact on the webpage render time and on the page load time, depending on whether the page is static or dynamic.

For this, the tested scenarios were: perform twenty tests without interaction to the static webpage `https://srlrr.github.io/test/` (webpage made for this purpose), composed by images, links and a HTML file; Perform twenty tests without interaction to the dynamic webpage `https://www.ua.pt/`. The focus of the tests were the *Client-side Processing Time* and the *Page Load Time* metrics, as they are the most relevant to compare dynamic versus static webpages.

Figure 5.5 represents the *Client-side Processing Time* values obtained on the tests executed, for the static and dynamic webpages.

In Figure 5.5 it is explicitly visible that the dynamic page takes more time to be loaded than the static one, which is fair, as the static one always loads the same content and has less types of files associated, for example, no JS files, reducing the processing needed on the client-side.

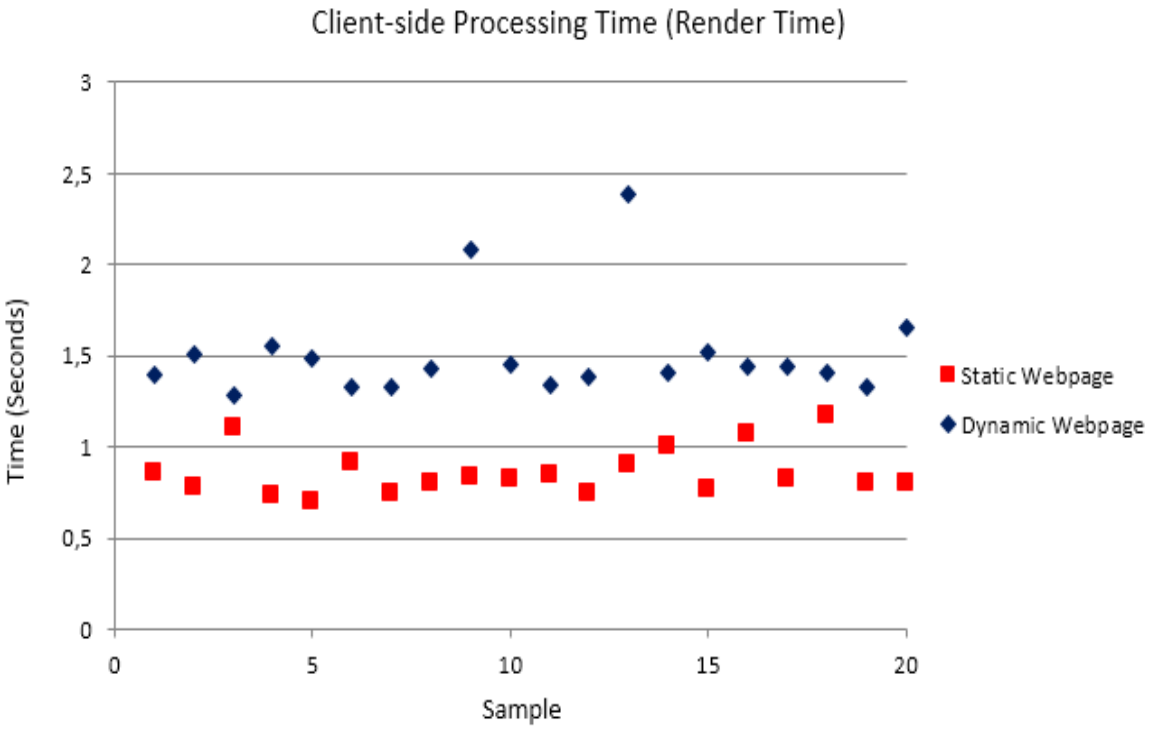


Figure 5.5: Client-side processing time or the time it takes to complete the load the webpage or the first part of the webpage, for a static and dynamic webpages.

In Figure 5.6 it is represented the page load times extracted on the twenty tests performed for each page type (static and dynamic).

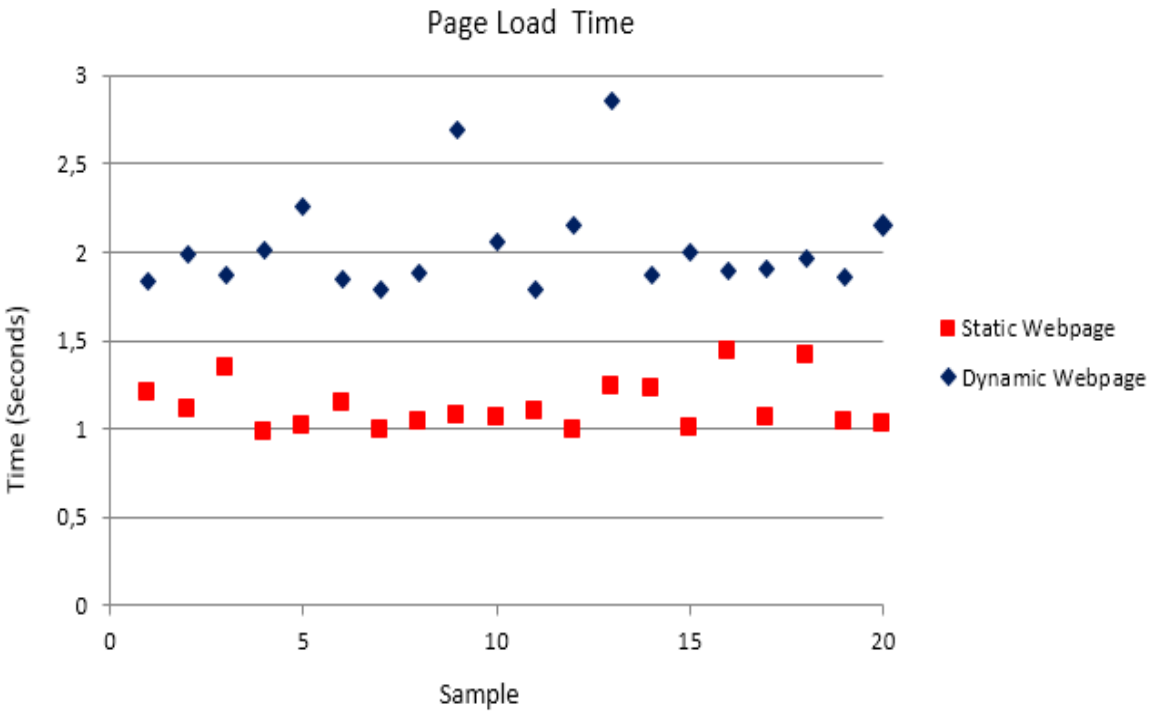


Figure 5.6: Page Load Time measured on a static and on a dynamic webpage.

In agreement with Figure 5.5, it is visible that the dynamic webpage takes longer on the process of page loading than the static one, showing that the type of webpage impacts on the client-side time, as well as on the total time of the webpage loading. Since that, it can be concluded that the render and DOM loading events are important on the comparison of a static with a dynamic webpage, as the charts 2.7 and 5.6 are very similar.

SmartBrowsing Navigation

In this section, it will be analyzed the coherence of the DNS Lookup time, the TCP connection time, the request time and response time metrics on a real navigation and on an emulated navigation performed by the *SmartBrowsing* solution on the access to the `http://www.sapo.pt/` webpage. For that purpose, it was performed a packet capturing during a real access to the webpage and also during a *SmartBrowsing* test to access the webpage.

For the case of the *SmartBrowsing* access to the webpage, the partial JSON output file is presented in Listing 5.1, with the metrics obtained after the webpage access, along with some informations about the test scenario.

```
{
"webPageTimings" : {
        "networkDelayTime" : 0.092,
        "pageInteractive" : 1.208,
        "requestTime" : 0.051,
        "connectionTime" : 0.05,
        "latency" : 0.185,
        "pageLoadTime" : 5.896,
        "lookupTime" : 0.061,
        "score" : 1,
        "responseTime" : 0.092,
        "TTFB" : 0.104,
        "clientSideProcessingTime" : 5.787
    },
    "generalInfo" : {
        "url" : "http://www.sapo.pt/",
        "screen" : {
            "screenHeight" : 730,
            "screenWidth" : 1053
        },
        "userAgent" : "Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:47.0) Gecko
    /20100101 Firefox/47.0",
        "browser" : "firefox",
        "statusCode" : 200
  }
}
```

Listing 5.1: Partial JSON output file, obtained from a test to the `http://www.sapo.pt/` webpage.

Domain Name System (DNS) Lookup Time

Figure 5.7 shows two captured packets, while the *SmartBrowsing* solution was running a test to access the `http://www.sapo.pt/` webpage. These packets correspond to the DNS lookup query and response for the IPv4 address matching the URL. By subtracting the packets timestamps (3,185056 - 3,182687), it is obtained a DNS lookup time of 0,002369 seconds.

```
3.182687    192.168.1.100    192.168.1.1      DNS    71 Standard query 0x911b A www.sapo.pt
3.185056    192.168.1.1      192.168.1.100    DNS    87 Standard query response 0x911b A www.sapo.pt A 213.13.146.142
```

Figure 5.7: IPv4 DNS request and response captured packets in parallel with a *SmartBrowsing* test.

Figure 5.8 is referent to the captured packets of the DNS lookup for the IPv6 address,

correspondent to the `http://www.sapo.pt/` webpage. These two packets were captured, also, while the *SmartBrowsing* solution was running the access to the *SAPO* webpage. The DNS lookup time for IPv6 is obtained by subtracting the packets timestamps (3,185799 - 3,182745), with a result of 0,003054 seconds.

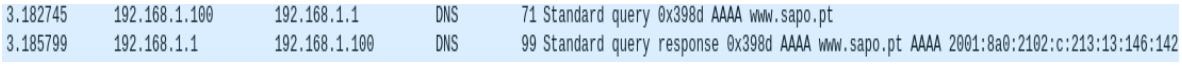


Figure 5.8: IPv6 DNS request and response captured packets in parallel with a *SmartBrowsing* test.

In the partial output JSON file, presented in Listing 5.1 in the beginning of this subsection *"SmartBrowsing Navigation"*, the DNS lookup time obtained by the solution was six milliseconds, which is not exactly the same as the sum of both of the DNS values calculated from the captured packets timestamp (0,002369 + 0,003054) which is, approximately, five milliseconds. However, this does not mean that the *SmartBrowsing* solution is returning a wrong time, it can mean that the DNS queries and responses for IPv4 and IPv6 are made in parallel, or one of them can be triggered while the other response has not arrived, or made with a temporal space in-between. Thus, the difference is not inconsistent, as both values are of the same order of magnitude in terms of units (milliseconds), and with very close values, approximately one millisecond of difference. Still, the value returned by the solution, compared with the one given by the subtraction of the packets timestamps, can be considered accurate. It is, the *SmartBrowsing* solution does not introduce delays on the DNS lookup time that is returned, compared to the value from the captures.

Transmission Control Protocol (TCP) Connect Time

The packets in Figure 5.9 represent the three-way handshake connection with the web-server. By subtracting the timestamps of the last and the first packets (2,655039 - 2,610855), the time interval for the connection was, approximately, of forty four milliseconds. This value dist from the one returned by the *SmartBrowsing* solution by, approximately, six milliseconds. However, this difference may be due to the a delay introduced by the *curl* command that performs the request for the TCP connection time.

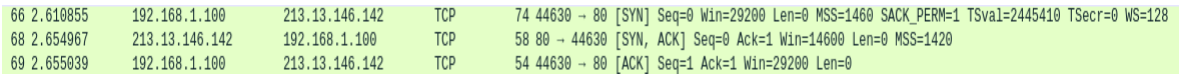


Figure 5.9: Three-way handshake TCP connection packets.

Request* and *Response Time

Figure 5.10 shows the packets correspondent to the *HTTP GET request* and to the *HTTP response*. By subtracting the packets timestamps (4,175272 - 4,032341) the time of request plus the response is of 0,142931 seconds, which coincides, approximately, with the sum of the values of request and response returned by the *SmartBrowsing* solution (0,051 + 0,092 = 0,143 seconds).

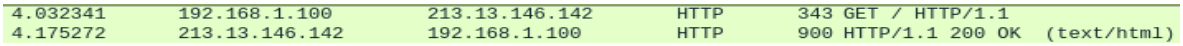


Figure 5.10: Captured packet of a webpage HTTP GET request and its respective HTTP response.

Interactions

In this section will be shown examples of the interactions available to be performed, by showing the partial JSON output file, to illustrate the timings and informations that are outputted and also the screen captures taken during the tests performed.

Login and Logout

The scenario tested for the login and logout interactions were performed to the `https://www.facebook.com/` and it will be presented the screen captures taken during the test, by order of capture:

In the first capture (vide Figure 5.11), it is visible the screen capture taken after the webpage access.

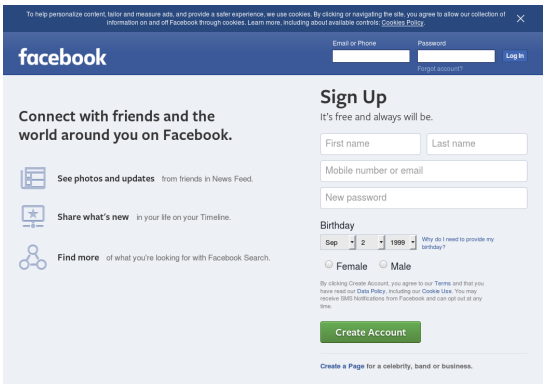


Figure 5.11: Screen capture of the initial page captured after the webpage access.

Figure 5.12 depicts the action before the trigger of the login interaction: the credentials insertion on the correspondent places, that were defined on the configuration file.

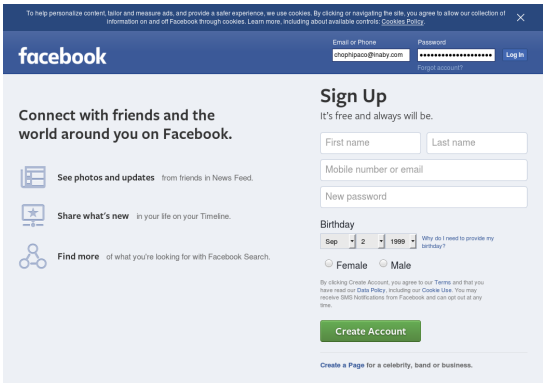


Figure 5.12: Screen capture taken after the insertion of the credentials to perform the login.

Figure 5.13 shows the screen capture taken after the login trigger, by the click on the login button. This screen capture (Figure 5.13) shows the webpage loaded after the login: the user's news feed.

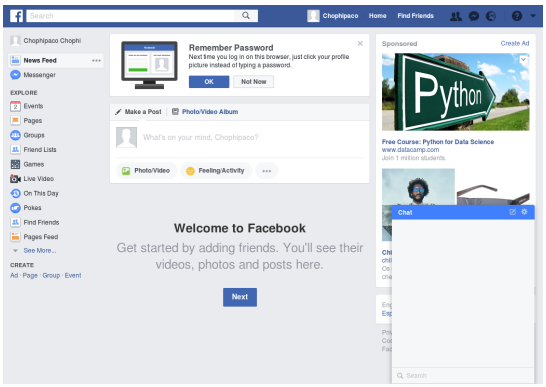


Figure 5.13: Screen capture of the webpage loaded after the login has been successfully performed.

Figure 5.14 shows the webpage loaded after the logout interaction. The logout action implies the click on the menu on the right corner of the webpage (vide Figure fig:loginDone) and, then, on the logout option of that menu. In the webpage loaded after the logout (vide Figure 5.14) it is visible the indication of the user that logged out, on the left side of the webpage, showing that the logout was successful.

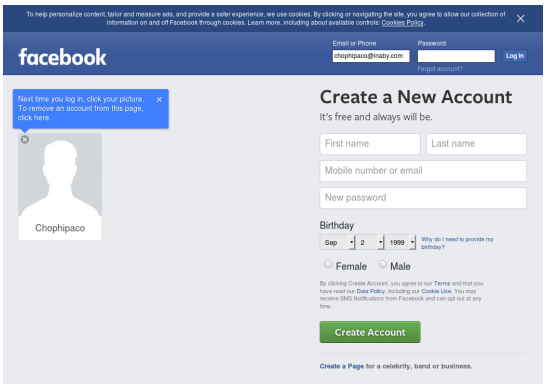


Figure 5.14: Screen capture taken after the logout has been successfully performed.

Listing 5.2 shows part of the outputted JSON file, which is composed by the informations gathered during the test. These informations include: the time metrics extracted after the URL fetch; The informations about the testing scenario configured (*generalInfo*), such as, the webpage tested and its status code, the user-agent, the screen size and the browser used. In this JSON, the list of loaded resources is not represented.


```
{
    "logoutTimeAndPageLoadTime" : 5.123,
    "webPageTimings" : {
        "networkDelayTime" : 0.391,
        "pageInteractive" : 1.26,
        "requestTime" : 0.208,
        "connectionTime" : 0.057,
        "latency" : 0.488,
        "pageLoadTime" : 2.78,
        "lookupTime" : 0.061,
        "score" : 1,
        "responseTime" : 0.097,
        "TTFB" : 0.406,
        "clientSideProcessingTime" : 2.351
    },
    "loginTimeAndPageLoadTime" : 11.272,
    "generalInfo" : {
        "url" : "https://www.facebook.com/",
        "screen" : {
            "screenHeight" : 730,
            "screenWidth" : 1053
        },
        "userAgent" : "Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:47.0) Gecko
    /20100101 Firefox/47.0",
        "browser" : "firefox",
        "statusCode" : 200
    },
    "logoutTime" : 2.921,
    "loginTime" : 5.451
}
```


Listing 5.2: Partial output written to the JSON file, of a test with login and logout interactions

To prove that the login and logout timings were accordingly with a real experience, both tasks were made by a real user, by accessing the *Facebook* website through the *Mozilla Firefox* browser. The browser session was opened in a private/anonymous session mode, in order to have a scenario with no cached credentials, as it occurs on tests performed using the *SmartBrowsing* solution.

The times for the login and logout were measured with the help of a chronometer, taking

into consideration the elements that needed to be visible on the screen (the ones used on the *SmartBrowsing* test configuration in Listing 4.1) as the ending point of the interaction (login or logout) time measurement and the click on the login/logout button (the same configured for the *SmartBrowsing* test) as starting point. Since that, the times measured were 4,10 seconds for the login task and 2,57 seconds for the logout task time. These values are different from the ones returned by the application, however, the difference between them is reasonable, as they are within the same order of magnitude. The login value resulting from the automated test dist from the real login value of 1,351 seconds. And the logout time resulting from the automated test dist from the real one in 0,351 seconds. The values measured may have a delay (increased or decreased value), not only related with the user reaction time while starting and stopping the chronometer, but also, with the user perception of the elements appearing on the screen.

However, this comparison allows the conclusion that the automated test for these interactions does not dist in much from the times of real interactions.

Scroll Down

In the *SmartBrowsing* solution there are two options of scroll down interactions: scroll down to the bottom of the webpage or scroll by a certain amount of pixels.

Following, will be presented the screen captures outputted by a test to the `https://lifestyle.sapo.pt/` webpage, that performed both types of scroll interactions. This webpage was chosen due to its big height, in order to better perceive the two types of scroll interactions on Figures 5.16 and 5.17.

Figure 5.15 shows the screen capture taken after accessing to the webpage with the *SmartBrowsing* solution, in order to have the webpage view after the webpage request.

Figure 5.15: Screen capture taken after the webpage request.

Then, the program starts to trigger the defined interactions, in this case, the first one is the scroll down by pixels. And, similarly, the program takes a screen capture after the accomplishment of the interaction (vide Figure 5.16). If there were, still, resources unloaded before the scroll interaction(s), they will be listed and outputted to the JSON file.

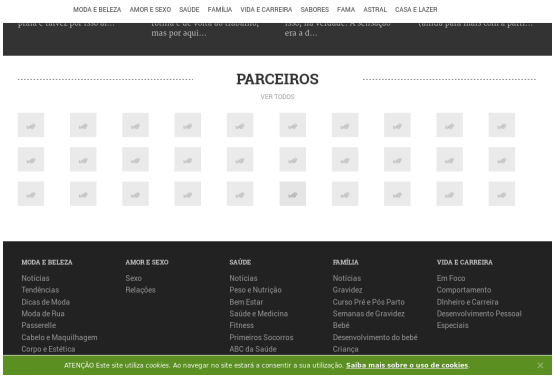


Figure 5.16: Screen capture taken after a scroll down by the specified amount of pixels on the configuration file.

Figure 5.17 corresponds to the last interaction, the scroll down to the bottom of the webpage.

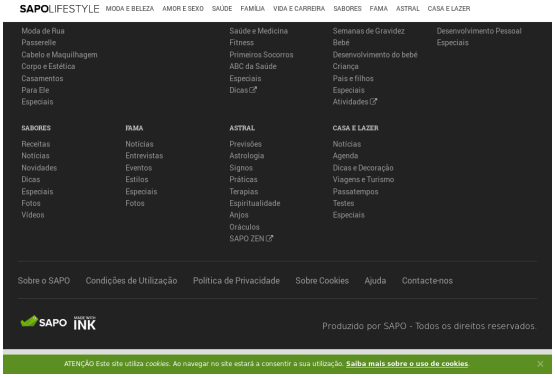


Figure 5.17: Screen capture taken after a scroll down to the bottom of the webpage.

Listing 5.3 represents part of the JSON output file, containing the general information and the metrics and interactions timings extracted during the test. In this JSON, it is not represented the list of loaded resources.

```
{
  "webPageTimings" : {
        "networkDelayTime" : 0.335,
        "pageInteractive" : 2.273,
        "requestTime" : 0.048,
        "connectionTime" : 0.163,
        "latency" : 0.467,
        "pageLoadTime" : 10.376,
        "lookupTime" : 0.125,
        "score" : 1,
        "responseTime" : 0.132,
        "TTFB" : 0.348,
        "clientSideProcessingTime" : 10.026
    },
    "scrollDownTime" : 0.108,
    "generalInfo" : {
        "url" : "https://lifestyle.sapo.pt/",
        "screen" : {
            "screenHeight" : 730,
            "screenWidth" : 1053
        },
      "userAgent" : "Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:47.0) Gecko
    /20100101 Firefox/47.0",
      "browser" : "firefox",
      "statusCode" : 200
  }
}
```

Listing 5.3: Partial output written to the JSON file, of a test with scroll interactions.

Comparison with a Web Tool Results

This section presents the tests performed, with two of the previously webpages used, on the *WebPagetest* tool[59]. This web tool was approached on chapter 2.

The tests to the two webpages, using the web tool, were made under the same conditions, that is, same location and browser.

The *WebPagetest*[59] allows to specify the location from where the test will be executed. However, Portugal was not in the options of the locations available. The location used was

Paris (France), because it is in Europe and was the most close location to Portugal, from all the options available.

The comparison will be performed between the results obtained from the web tool and from the *SmartBrowsing* solution. The values used were extracted from the *SmartBrowsing* tests, previously presented (vide Listings 5.1 and 5.2). Hence, the browser used on the *WebPagetest* tool was the *Mozilla Firefox*, because it was the used on those previous *SmartBrowsing* tests.

Table 5.5 depicts the metrics returned by the *WebPagetest* tool, for two of the websites used on previous sections: `www.sapo.pt` and `www.facebook.com`.

	SAPO	Facebook
Page Load Time (seconds)	7,613	2,297
TTFB (seconds)	0,171	0,493

Table 5.5: Table with the metrics measured (in seconds), by the *WebPageTest* tool, for the webpages: `www.sapo.pt` and `www.facebook.com`.

Table 5.6 depicts the same metrics returned by the *SmartBrowsing* solution, for two of the websites used on previous sections: `www.sapo.pt` and `www.facebook.com`. These values were extracted from the Listings 5.1 and 5.2, respectively.

	SAPO	Facebook
Page Load Time (seconds)	5,896	2,78
TTFB (seconds)	0,104	0,406

Table 5.6: Table with the metrics measured (in seconds), by the *SmartBrowsing* solution, for the webpages: `www.sapo.pt` and `www.facebook.com`.

The values obtained are very similar for both metrics. The page load time for the SAPO webpage, measured with the *WebPageTest* tool, dist from the *SmartBrowsing* value by 1,717 seconds. Also, the TTFB dist by 0,067 seconds. For the *Facebook* webpage, the page load time dist by 0,483 seconds, and for the TTFB there is a difference of 0,087 seconds.

The page load time metric is compound by network and device factors. Hence, the page load time considers the interval between the beginning of the URL access and the time at when the webpage stops loading. This interval considers events are dependent from: traffic, amount of requests to the webpage host server, delaying the response, or even the existence of CDN that can cache the most accessed content by users, and, consequently, reduce the waiting time. Besides connection factors, the device used, which performs the URL request, also impacts on the page load time. The device impact is in terms of the webpage rendering, as previously discussed on section 5.2.

The TTFB corresponds to the time it took to receive the first byte of the response from the server, from the time of the URL request. Since that, this metric also includes factors of network and device capabilities.

From the observance of Tables 5.5 and 5.6, the values are within the same order of magnitude, indicating that the *SmartBrowsing* solution returns identical values to the ones returned by *WebPageTest*, for these metrics.

The SAPO webpage results present a bigger difference comparing to the Facebook results. This can be due to the fact that the SAPO webpage target users are the Portuguese population, and so, the web servers are located in Portugal. Consequently, the tests made to the SAPO webpage with the *SmartBrowsing* tool show that the webpage responsiveness is higher than for the *WebPageTest* tool, which test location was in France. That is due to the distance that the packets need to through.

Although the differences stated with the SAPO webpage, the tests to the Facebook webpage present very similar results in both tools. The reason for these similarities can be due to the interest of the Facebook company for the all the users of the world. This webpage is used by millions of people af over the globe, since that, the Facebook needed to create strategies in order to have a uniform responsiveness of its website. That strategies can include the presence of CDNs on several places of the world, that bring the contents closer to the users, in order to provide a better responsiveness and, hence, a better experience for all the users, even the ones that are far from the Facebook hosting servers. This can explain the close results for each metric extracted with each tool.

Conclusion

This chapter aim was to address the tests performed, in order to proof the solution effectiveness on the emulation of a real experience on webpage access through a real browser.

Since that, the tests performed intended to analyze the overall scenarios of the *SmartBrowsing* solution. For that purpose, the solution was run and were made several tests: to verify if the device used has impact on the solution performance; Packet captures, to proof the reliability of the results; “Screen” captures to better illustrate the program flow for tests with interactions; And, a comparison between the metrics available for both the *WebPageTest* tool and SmartBrowsing solution.

In this chapter can be concluded that the device in use has impact on the webpage loading time, and on the *SmartBrowsing* program execution time, consequently, impacting on the QoE. Moreover, the timings gathered by the *SmartBrowsing* solution were considered reasonable, after the packets analysis. In the interactions scenarios, when compared to the

values measured on real interactions, the *SmartBrowsing* values were proved to be reliable.

The network-related metrics extracted are dependent on several factors, such as the location from where the webpage access is done, the traffic or requests amount to the webpage host server at the time. Since that, the values obtained on different tools that perform tests from different locations are likely to have differences on the values returned.

Although that, companies like *Facebook* that has users from all over the world, needs to improve the responsiveness of the webpage. For that purpose, this companies use strategies such as, the presence of elements like CDNs in strategic places of the world, allow to improve and uniform the experience for all the world users. That justifies the fact of the very close values obtained for both metrics in the test scenario that compared the *SmartBrowsing* with the *WebPageTest* tool.

6

Conclusions And Future Work

This master thesis studied the possibilities to automate the browsers and perform interactions on the webpages, in order to simulate a real-user experience. This way and by analyzing the metrics extracted along with the screen captures, it is possible to debug the causes that differentiate a good and a bad experience. Also, this could be completed with real users opinions, in order to calibrate the labeling of good and bad experience perception.

The *SmartBrowsing* solution is a useful tool both for web developers, that can debug their webpages and see what are the biggest influences on their webpage performance in terms of QoE, but can also, for example, help enterprises in the improvement of their web services, like the pages of authentication to have access to a wireless connection, in places such as, restaurants, hotels or free access hotspots available in certain locations.

The proposed objectives were all accomplished as it was implemented a solution that can perform user-like interactions in real browsers and obtain the times associated with the webpage access.

Further in time, the solution can be improved in terms of including other test scenarios, like the possibility to test on mobile browsers, although the *SmartBrowsing* solution allowance to define an user-agent, even a mobile one. It also supports the specificities of a mobile interaction, but by now, it is not possible to use a mobile browser, rather than the *Google Chrome* or the *Mozilla Firefox* ones.

Next improvements can also include a more complex and viable formula to score the webpages. Also, it is needed to test and integrate the solution on the *ArQoS* probes and management system.

There is also the possibility to evolve from the metrics extracted to others, as the QoE topic evolves over time, other metrics appear.

Furthermore, there is the need to keep the technologies used, on the deployment of this solution, updated, in order to keep the solution updated and with more possibilities to be

implemented as the technologies used evolve.

The conclusions taken are that the device characteristics and the browser in use impact on the QoE and impact on the execution times of the *SmartBrowsing* application. Also, the packets captured proved that the metrics tested were reliable, and can be inferred that the other metrics, not directly analyzed, are also reliable.

Overall, this solution allows to test different scenarios and retrieves several metrics that can be further evaluated by a person, in order to debug the causes of a good or a bad experience.

The metrics measured in each web access are different, due to its dependency on several factors. The factors influencing that difference are, for example, the location from where the access is performed, the traffic or the amount of requests to the web server at the time. All these factors have impact on the users QoE. Besides those factors, the presence of some elements in the network, like the CDNs, can improve the user-experience while navigating on the Web, as the contents requested are closer to the end user. In addition, the device, the browser used and the type of webpage accessed, have impact on the QoE perceived by the end-user.

The measuring of the QoE, while navigating in the Web, is complex and implies the need to have in consideration all the factors previously referred, along with each user individuality on the experience perception. The classification of the web navigation QoE needs research to verify which are the most impacting factors on a web navigation experience.

This master dissertation intends to give a contribute on the automation of the QoE evaluation perceived by real users, while navigating on a Web browser.

Bibliography

[1] K. Brunnström *et al.*, *Qualinet White Paper on Definitions of Quality of Experience*. Qualinet - European Network and Services, 2013.

[2] T. S. S. O. ITU, "ITU-T Rec. G.1031 (02/2014) QoE factors in web-browsing," ITU - International Telecommunication Union, Tech. Rep., 2014.

[3] T. S. S. O. ITU, "ITU-T Rec. G.1040 Network contribution to transaction time, institution = ITU - International Telecommunication Union," Tech. Rep., 2006.

[4] T. S. S. O. ITU, "Vocabulary for performance and quality of service, amendment 1: New appendix i – definition of quality of experience (qoe)," ITU - International Telecommunication Union, Tech. Rep., January 2007.

[5] F. Kuipers, R. Kooij, D. De Vleeschauwer, and K. Brunnström, "Techniques for measuring quality of experience," in *Proceedings of the 8th International Conference on Wired/Wireless Internet Communications*, ser. WWIC'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 216–227.

[6] E. Bocchi, L. De Cicco, and D. Rossi, "Measuring the quality of experience of web users," in *Proceedings of the 2016 Workshop on QoE-based Analysis and Management of Data Communication Networks*, ser. Internet-QoE '16. New York, NY, USA: ACM, 2016, pp. 37–42.

[7] T. Hoßfeld, P. E. Heegaard, M. Varela, and S. Möller, "QoE beyond the MOS: an in-depth look at QoE via better metrics and their relation to MOS," *Quality and User Experience*, vol. 1, no. 1, p. 2, sep 2016.

[8] ETSI, "Humman Factors (HF): QoE requirements for real-time communication services," Tech. Rep., 2009.

[9] K. Kilkki, "Quality of experience in communications ecosystem," vol. 14, no. 5, pp. 615–624, March 2008.

[10] S. Halabi, "Evolution of the Internet," in *Internet Routing Architectures*. Cisco, 2000, ch. 1, pp. 1–50.

[11] C. Wareham, "On the Moral Equality of Artificial Agents," in *Moral, Ethical, and Social Dilemmas in the Age of Technology: Theories and Practice*. IGI Global, 2013, ch. 5, pp. 70–78.

[12] Microsoft, "Tcp/ip protocol architecture," 2010. [Online]. Available: http://technet.microsoft.com/en-us/library/cc958821.aspx [Accessed: 2017-09-07]

[13] Parziale *et al.*, *TCP/IP Tutorial and Technical Overview*, 8th ed. IBM's International Technical Support Organization, 2006, vol. 1, no. December 2006.

[14] World Wide Web Foundation, "History of the Web – World Wide Web Foundation," 2008. [Online]. Available: http://webfoundation.org/about/vision/history-of-the-web/ [Accessed: 2017-05-5]

[15] P. S. Wang, "An Overview of HTTP," pp. 1–28, 2017. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview [Accessed: 2017-08-02]

[16] J. Postel, "User datagram protocol - rfc 768," United States, 1980.

[17] IETF, "Transmission Control Protocol - RFC 793," 1981.

[18] T. Dierks, "The transport layer security (TLS) protocol version 1.2 - RFC 5246," 2008.

[19] R. Fielding *et al.*, "Hypertext Transfer Protocol – HTTP/1.1 - RFC 2616," 1999.

[20] M. Belshe *et al.*, "Hypertext Transfer Protocol Version 2 - RFC 7540," 2015.

[21] E. Rescorla, "HTTP over TLS - RFC 2818," 2000.

[22] M. D. Network, "Introduction to the dom - gecko dom reference — mdn." [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction [Accessed: 2017-09-09]

[23] W3C, "Tim Berners-Lee," 2016. [Online]. Available: https://www.w3.org/People/Berners-Lee/ [Accessed: 2017-07-12]

[24] NetMarketShare, "Mobile/Tablet Browser Market Share," 2015. [Online]. Available: https://netmarketshare.com/ [Accessed: 02/08/2017]

[25] Google, "Download and install Google Chrome - Computer - Google Chrome Help." [Online]. Available: https://support.google.com/chrome/answer/95346?co=GENIE.Platform%3DDesktop&hl=en [Accessed: 2017-08-03]

[26] W3Schools, "Browser Statistics," p. 5, 2015. [Online]. Available: https://www.w3schools.com/browsers/default.asp [Accessed: 2017-05-02]

[27] StatCounter, "StatCounter Global Stats - Browser, OS, Search Engine including Mobile Market Share," 2013. [Online]. Available: http://gs.statcounter.com/ [Accessed: 2017-08-03]

[28] W. Foundation, "Dashiki: Simple Request Breakdowns." [Online]. Available: https://analytics.wikimedia.org/dashboards/browsers/#desktop-site-by-browser/browser-family-timeseries [Accessed: 2017-08-03]

[29] Mozilla, "Firefox." [Online]. Available: https://www.mozilla.org/en-US/firefox/ [Accessed: 2017-08-03]

[30] Microsoft. Internet explorer system requirements ie9. [Online]. Available: https://support.microsoft.com/en-us/help/11531/internet-explorer-system-requirements [Accessed: 2017-08-03]

[31] Microsoft., "Microsoft Edge requirements and language support (Microsoft Edge for IT Pros) - Microsoft Docs." [Online]. Available: https://docs.microsoft.com/en-us/microsoft-edge/deploy/hardware-and-software-requirements [Accessed: 2017-08-03]

[32] Apple, "macOS - Safari - Apple." [Online]. Available: https://www.apple.com/safari/ [Accessed: 2017-08-03]

[33] Opera. Best browser for linux — download — fast & safe — opera. [Online]. Available: https://www.opera.com/computer [Accessed: 2017-08-03]

[34] IETF, "About the IETF," 2015. [Online]. Available: https://www.ietf.org/about/ [Accessed: 2017-08-17]

[35] R. Fielding and J. Reschke, "Hypertext Transfer Protocol (HTTP 1.1) - Semantics and Content - RFC 7231," 2014.

[36] UserAgentString, "UserAgentString.com - List of User Agent Strings." [Online]. Available: http://www.useragentstring.com/pages/useragentstring.php [Accessed: 2017-08-17]

[37] A. Hidayat, "PhantomJS — PhantomJS," 2016. [Online]. Available: http://phantomjs.org/ [Accessed: 2017-08-09]

[38] ScrapingHub, "Splash - A javascript rendering service — Splash 2.3 documentation." [Online]. Available: http://splash.readthedocs.io/en/stable/index.html [Accessed: 2017-08-17]

[39] Lua, "Lua: getting started." [Online]. Available: http://www.lua.org/start.html [Accessed: 2017-08-17]

[40] SensioLabs, "The BrowserKit Component (The Symfony Components)." [Online]. Available: https://symfony.com/doc/current/components/browser{_}kit.html [Accessed: 2017-08-17]

[41] S. Stewart, "Selenium WebDriver," 2012. [Online]. Available: http://www.seleniumhq.org/docs/03_webdriver.jsp [Accessed: 2017-07-03]

[42] "SeleniumHQ/selenium is licensed under the Apache License 2.0." [Online]. Available: https://github.com/SeleniumHQ/selenium/blob/master/LICENSE [Accessed: 2017-07-04]

[43] W3C, "WebDriver." [Online]. Available: https://www.w3.org/TR/webdriver/ [Accessed: 2017-08-29]

[44] Google, "ChromeDriver - WebDriver for Chrome." [Online]. Available: https://sites.google.com/a/chromium.org/chromedriver/ [Accessed: 2017-08-19]

[45] Mozilla, "Marionette - Mozilla — MDN." [Online]. Available: https://developer.mozilla.org/en-US/docs/Mozilla/QA/Marionette [Accessed: 2017-08-29]

[46] Mozilla., "WebDriver - Mozilla — MDN." [Online]. Available: https://developer.mozilla.org/en-US/docs/Mozilla/QA/Marionette/WebDriver [Accessed: 2017-08-29]

[47] Sahi, "Automation Testing Tool For Web Applications — Free - Sahi." [Online]. Available: http://sahipro.com/ [Accessed: 2017-08-07]

[48] Watir, "Watir is... – Watir Project – Watir stands for Web Application Testing In Ruby. It facilitates the writing of automated tests by mimicking the behavior of a user interacting with a website." [Online]. Available: http://watir.com/ [Accessed: 2017-08-07]

[49] Linux Information Project, "An introduction to X by The Linux Information Project (LINFO)," 2006. [Online]. Available: http://www.linfo.org/x.html [Accessed: 2017-08-16]

[50] IBM, "IBM Knowledge Center - The client/server model." [Online]. Available: https://www.ibm.com/support/knowledgecenter/en/SSAL2T_8.1.0/com.ibm.cics.tx.doc/concepts/c_clnt_sevr_model.html [Accessed: 2017-08-16]

[51] Linux Information Project, "X server definition by The Linux Information Project," 2005. [Online]. Available: http://www.linfo.org/x_server.html [Accessed: 2017-08-16]

[52] D. P. Wiggins, "XVFB." [Online]. Available: https://www.x.org/archive/X11R7.6/doc/man/man1/Xvfb.1.xhtml [Accessed: 2017-08-16]

[53] T. Richardson, "Xvnc - the X-based VNC server," 2013. [Online]. Available: http://www.hep.phy.cam.ac.uk/vnc_docs/xvnc.html [Accessed: 2017-08-16]

[54] A. Laboratories, "Getting Started with," 1999. [Online]. Available: http://www.hep.phy.cam.ac.uk/vnc_docs/start.html [Accessed: 2017-08-16]

[55] J. Chroboczek, "The KDrive Tiny X Server." [Online]. Available: https://www.irif.fr/~jch/software/kdrive.html [Accessed: 2017-08-16]

[56] M. Allum, "Xephyr," 2004. [Online]. Available: ftp://www.x.org/pub/X11R7.5/doc/man/man1/Xephyr.1.html [Accessed: 2017-08-16]

[57] Google, "PageSpeed Insights," 2015. [Online]. Available: https://developers.google.com/speed/pagespeed/insights/?hl=en-EN [Accessed: 2017-08-16]

[58] Solarwinds, "Website And Performance Monitoring — Pingdom." [Online]. Available: https://www.pingdom.com/ [Accessed: 2017-08-16]

[59] P. Meenan, "WebPagetest - Website Performance and Optimization Test," 2016. [Online]. Available: https://www.webpagetest.org/ [Accessed: 2017-08-16]

[60] Intraway, "Intraway - Quality of Experience Smart Probes." [Online]. Available: http://www.intraway.com/w01/solutions/customer-experience/qx [Accessed: 2017-08-09]

[61] Intraway., "Intraway qx - brochure." [Online]. Available: http://web.intraway.com/brochures/QX/Intraway_QX-Brochure-ENG.pdf [Accessed: 2017-08-09]

[62] mPlane Consortium, "PUBLICATIONS Building an Intelligent Measurement Plane for the Internet." [Online]. Available: http://www.ict-mplane.eu/public/firelog [Accessed: 2017-08-09]

[63] Q. A. Chen *et al.*, "QoE Doctor," *Proceedings of the 2014 Conference on Internet Measurement Conference - IMC '14*, pp. 151–164, 2014. [Online]. Available: http://web.eecs.umich.edu/~alfchen/alfred_imc14.pdf [Accessed: 2017-08-15]

[64] IEEE Spectrum, "The 2016 Top Programming Languages - IEEE Spectrum," 2016. [Online]. Available: http://spectrum.ieee.org/computing/software/the-2016-top-programming-languages [Accessed: 2017-08-18]

[65] P. S. Foundation, "PyVirtualDisplay — PyVirtualDisplay 0.2.1 documentation." [Online]. Available: http://pyvirtualdisplay.readthedocs.io/en/latest/ [Accessed: 2017-08-23]

[66] M. D. Network, "Performance - Web APIs — MDN." [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Performance [Accessed: 2017-08-21]

[67] Z. Wang, "Navigation Timing," 2012. [Online]. Available: https://www.w3.org/TR/navigation-timing/ [Accessed: 2017-08-21]

[68] P. S. Foundation, "selenium-requests 1.3 : Python Package Index." [Online]. Available: https://pypi.python.org/pypi/selenium-requests/ [Accessed: 2017-08-21]

[69] M. D. Network, "Navigation Timing API - Web APIs — MDN." [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Navigation_timing_API [Accessed: 2017-08-25]

[70] Tcpdump/Libpcap, "TCPDUMP/LICPCAP public repository," 2014. [Online]. Available: http://www.tcpdump.org/ [Accessed: 2017-08-29]

[71] Wireshark Foundation, "Wireshark. Go deep." p. 27.05.2010, 2010. [Online]. Available: http://www.wireshark.org/ [Accessed: 2017-08-29]

[72] SeleniumHQ, "Platforms Supported by Selenium." [Online]. Available: http://docs.seleniumhq.org/about/platforms.jsp [Accessed: 2017-08-22]

[73] Google, "Chrome system requirements - Chrome for business and education Help." [Online]. Available: https://support.google.com/chrome/a/answer/7100626?hl=en [Accessed: 2017-08-22]

[74] Mozilla Firefox, "Firefox 51.0.1 System Requirements." [Online]. Available: https://www.mozilla.org/en-US/firefox/51.0.1/system-requirements/ [Accessed: 22-08-2017]

A

Appendix

Minimum Operating Requirements

In this section are presented the requirements of the operating system and the technologies versions installed in order to get this application functional. These requirements are focused only on the solution requirements 3.3, on the chapter "*SmartBrowsing Solution*", that is, it will only be addressed the compatibilities with the *Linux* system.

The technologies versions installed in the deployment of the solution were:

- The *Selenium Webdriver* version installed was 3.0.2.

- The *ChromeDriver* version installed was 2.27.440175.

- The *Geckodriver* version installed was 0.14.0.

- The *Google Chrome* version installed was 57.0.2987.98.

- The *Mozilla Firefox* version installed was 51.0.1.

Accordingly with the *Selenium* team, this tool is mainly tested on *Ubuntu* operating systems, but should work well on other variations of *Linux* and also be functional if the browsers manufacturers support those operating systems [72], which got proved as this solution was tested on a *Linux*-based operating system.

The *Google Chrome* browser requirements in terms of the Linux-based operating system are (accordingly with the *Google Chrome* Support [73]):

- For *64-bit Ubuntu*: version 14.04 or newer.

- For *Debian*: version 8 or upper.
- For *openSUSE*: version 13.3 or upper.
- For *Fedora Linux*: version 24 or upper.

And for the *Mozilla Firefox* browser, the requirements were in terms of the following libraries and packages [74]:

- *GTK+* with version 3.4 or higher;
- *GLib* with version 2.22 or higher;
- *Pango* with version 1.14 or higher;
- *X.Org* with version 1.0 or higher. Although, the installation of version 1.7 or higher is recommended;
- *libstdc++* with version 4.6.1 or higher.

But, the *Mozilla* support states that for a optimal functioning of the browser, it is recommended the following packages and libraries [74]:

- *NetworkManager* with version 0.7 or higher;
- *DBus* with version 1.0 or higher;
- *GNOME* with version 2.16 or higher.

Solution Deployment

In this section, it will be addressed the steps and downloads needed to have the run the solution in a device compatible with the versions addressed on the previous section "*Minimum Operating Requirements*" A.1.

Firstly, to install the *Python* programming language, can be used the following commands:

```
1 $ wget --no-check-certificate https://www.python.org/ftp/python/2.7.12/Python
    -2.7.12.tgz
2 $ tar -xzf Python-2.7.11.tgz
3 $ ./configure
4 $ make
5 $ sudo make install
```

Then, to install the headless display, the Xvfb, the commands used were:

```
$ sudo apt-get install xvfb
```

After that, it was installed the *PyVirtualDisplay*, with the following commands:

```
$ sudo apt-get update
$ sudo apt-get install python-pyvirtualdisplay
```

For the installation of the *Selenium* it was used the following command:

```
$ pip install -U selenium
```

In order to install the webdrivers, *ChromeDriver* and *Geckodriver*, the commands were the following:

For the *ChromeDriver*:

```
$ sudo apt-get install unzip
$ wget -N http://chromedriver.storage.googleapis.com/2.10/chromedriver_linux64.
    zip -P <path_to_destionation_directory>
$ unzip <path_to_zip_file_directory>/chromedriver_linux64.zip -d <
    path_to_destination_directory>
$ chmod +x <path_to_webdriver_directory>/chromedriver
$ sudo mv -f <path_to_webdriver_directory>/chromedriver /usr/local/share/
    chromedriver
```

The available versions of the *ChromeDriver* can be consulted on the following link: `http://chromedriver.storage.googleapis.com/`.

For the *Geckodriver*:

```
$ wget https://github.com/mozilla/geckodriver/releases/download/v0.14.0/
    geckodriver-v0.14.0-linux64.tar.gz
$ tar -xvzf geckodriver*
$ chmod +x geckodriver
$ export PATH=$PATH:/path-to-extracted-file/geckodriver
```

The available versions of the *Geckodriver* can be consulted on the following link: `https://github.com/mozilla/geckodriver/releases/`.

Finally, were installed the following libraries:

The *Selenium-requests Python* library:

```
$ sudo pip install selenium-requests
```

The *beautifulsoup4 Python* library:

```
$ pip install beautifulsoup4
```