B. Garcia, F. Gortazar, L. Lopez-Fernandez, M. Gallego and M. Paris (2017). WebRTC Testing: Challenges and Practical Solutions. *IEEE Communications Standards Magazine*, 1(2), pp. 36-42.

WebRTC Testing: Challenges and Practical Solutions

Boni García, Francisco Gortázar, Luis López-Fernández, Micael Gallego, Miguel París
Universidad Rey Juan Carlos
{boni.garcia, francisco.gortazar, luis.lopez, micael.gallego, miguel.paris}@urjc.es

Abstract

WebRTC comprises a set of novel technologies and standards that provide Real-Time Communication on Web browsers. WebRTC makes simple the embedding of voice and video communications in all types of applications. However, releasing those applications to production is still very challenging due to the complexity of their testing. Validating a WebRTC service requires assessing many functional (e.g. signaling logic, media connectivity, etc.) and non-functional (e.g. quality of experience, interoperability, scalability, etc.) properties on large, complex, distributed and heterogeneous systems that spawn across client devices, networks and cloud infrastructures. In this article, we present a novel methodology and an associated tool for doing it at scale and in an automated way. Our strategy is based on a black-box end-to-end approach through which we use an automated containerized cloud environment for instrumenting Web browser clients, which benchmark the SUT (System Under Test), and fake clients, that load it. Through these benchmarks, we obtain, in a reliable and statistically significant way, both network-dependent QoS (Quality of Service) metrics and media-dependent QoE (Quality of Experience) indicators. These are fed, at a second stage, to a number of testing assertions that validate the appropriateness of the functional and non-functional properties of the SUT under controlled and configurable load and fail conditions. To finish, we illustrate our experiences using such tool and methodology in the context of the Kurento open source software project and conclude that they are suitable for validating large and complex WebRTC systems at scale.

Introduction

Web applications and services are among the most popular ones in today's Internet. Due to this, Web developers are subject to an increasing demand for delivering more complex, scalable and reliable applications in less and less time. The most limiting factor for satisfying this demand is software verification and validation that may account for more than 70% of the total development effort [1]. Because of this, during the last few years, we have witnessed an explosion of DevOps methodologies and tools that aim to simplify the validation process. Thanks to these, nowadays Web developers are able to automate unit and system tests. These tests typically comprise one or several probes that query the System Under Test (SUT), or any of its components, and gather simple outcomes as a result. After that, a set of testing assertions evaluate the system correctness by comparing such outcomes with pre-set or model-inferred values. An assertion (or predicate) is as a mechanism for determining whether a test has passed or failed. It involves comparing an outcome of the SUT with the expected value.

The popularization of multimedia technologies among Web developers and, very particularly, the emergence of WebRTC as a built-in feature of Web browsers, is significantly increasing the complexity of the testing. WebRTC comprises a set of standards designed for embedding RTC (Real-Time Communication) capabilities as part of Web browsers' APIs [2]. A recent report predicts that with Apple and Microsoft supporting WebRTC in their browsers, there might be 7 billion devices compliant with WebRTC by 2020 [3].

WebRTC applications cannot be tested using the usual simple comparison-based assertion. For example, validating the functional correctness of a WebRTC application requires the ability to evaluate aspects such as media connectivity (e.g. whether the media bits are being sent end-to-end) or media continuity (e.g. whether the media is decodable). Assessing non-functional properties is even more complex. The real-time nature of WebRTC traffic makes QoS (Quality of Service) parameters such as network latency, network jitter or packet loss to affect significantly, and in non-trivial ways, the end-user's QoE (Quality of Experience). Measuring accurately such QoE and enabling assertions to automate their validation is a huge challenge.

As an additional complication, testing WebRTC applications is challenging due to the heterogeneous nature of the networks to be traversed in order to enable a P2P communication. Some problems include clients consuming media might be behind a NAT or a firewall. As a result, WebRTC services involve complex, distributed and heterogeneous network topologies where failures or inefficiencies may prevent the service to operate offering a successful user experience.

All in all, creating reliable and automated tests for WebRTC applications, services and infrastructures is a task that cannot be performed in a simple way basing on commonly available state-of-the-art DevOps tools and methodologies. In this article, we contribute a solution to this problem proposing both a strategy and technological toolbox that are suitable for the automated testing of WebRTC applications and services. Our approach is based on black-box end-to-end tests that capture QoS and QoE indicators. Basing on these indicators, developers can create simple test assertions. In addition, we provide customizable capabilities to exercise the SUT in different network scenarios in order to simulate real world WebRTC potential issues at development stage.

The remainder of this article is structured as follows. In the next section, we compare our approach to some existing ones in the literature. Next section describes the overall methodology and contributions of this work. Then, we describe how our proposal has been used to carry out a case study in which a WebRTC broadcasting application is assessed in terms of scalability. Finally, we summarize the conclusions and findings of this work.

Related work

Traditionally, tools such as Apache JMeter has been used as a load testing tool for analyzing and measuring the performance of a variety of services, with a focus on Web applications [4]. Nevertheless, these kinds of tools are not valid to test WebRTC applications because it is required to use browsers implementing the WebRTC stack.

In the WebRTC arena, we find that very few solutions exist aimed to verify WebRTC applications in the literature. One of these is WebRTCBench, an open source tool for performance assessment of WebRTC implementations [5] which allows testing applications making use of video and audio through WebRTC standards and collects performance indicators. It consists of a NodeJS application with a HTML5 client, supporting Chrome and Firefox browsers. Another is Jattack [6], a general-purpose WebRTC stressing tool capable to simulate the activities of multiple WebRTC sessions. This tool uses Janus media server to emulate the behavior of a dynamically adjustable number of WebRTC clients to stress the SUT. Then, several physical parameters of the SUT (CPU and memory) are monitored while gathering the number of negative acknowledgments (NACKs) as an estimator of the overall quality of the WebRTC session.

However, these approaches are very limited when comparing with the toolbox presented in this article. As a differential advantage, our approach supports advance QoE/QoS indicators to carry out complete testing of WebRTC application at scale while forcing different network scenarios.

Automated testing of WebRTC applications: methodology and contributions

Our approach is the result of our experiences on assessing WebRTC applications, services and infrastructures that have been developed in the context of the Kurento open source software project. We have designed a testing strategy and we have enforced it through a toolbox that enables the creation and automation of tests. Figure 1 shows a high-level description of the approach. Our toolbox relies on browser instrumentation techniques for emulating the behavior of end-users accessing the WebRTC applications. The strategy of instrumenting browsers for performing end-to-end tests is not new and it has been widely used for the validation of Web applications thanks to open source software tools such as Selenium. However, as a novelty, our instrumented browsers have been extended to probe the SUT and collect diverse WebRTC QoS and QoE metrics using complex media processing techniques that include speech analysis and computer vision. The metrics thus collected are made available to developers through simple APIs so that they can implement assertions based on their individual or statistical values seamlessly.

This is a quite convenient method for exposing to developers some meaningful indicators on top of which assertions can be built, but it has a relevant drawback: all browser instrumentation and automation technologies are greedy consumers of computing resources. This is due to the need of launching and executing a complete Web browser stack per session tested. As a result, this technique has relevant limitations at the time of testing and benchmarking for scalability as the required computing costs for spawning a reasonable number of browser probes is typically prohibitive.

For avoiding this, our strategy introduces a mechanism based on limiting the number of such browsers during the tests to a small value. In order to test for scalability, our toolbox provides the ability of loading the system through light-weight WebRTC clients that we call *fake clients*. Fake clients do not implement the complete WebRTC media processing pipeline but only the parts that are strictly required for being indistinguishable from a real browser form the WebRTC infrastructure or application (i.e. negotiation and transport). Therefore, these fake clients can gather QoS metrics although they cannot gather any QoE indicators. In exchange, their CPU and memory fingerprint is very low and tests may instantiate thousands of them without needing huge computing resources. The toolbox exposes coherent APIs that developers can use for instantiating and controlling the behavior of fake clients at runtime during the tests, so that customized load patterns can be produced.

Moreover, the toolbox introduces the concept of *test scenario*, which can be seen as the collection of browsers in which a given test case is going to be exercised. This test scenario can be defined in a simple JSON notation, in which the type (e.g. Chrome, Firefox) and version of browser can be configured. This is a small step for testers but one giant leap for DevOps. On the one hand, testers can focus on the test logic, typically using a local browser for test development. On the other hand, DevOps can focus on test data, customizing the test executions with rich test scenarios in JSON notation included in the job configuration on the Continuous Integration (CI) server, and thus performing compatibility tests in a seamless way.
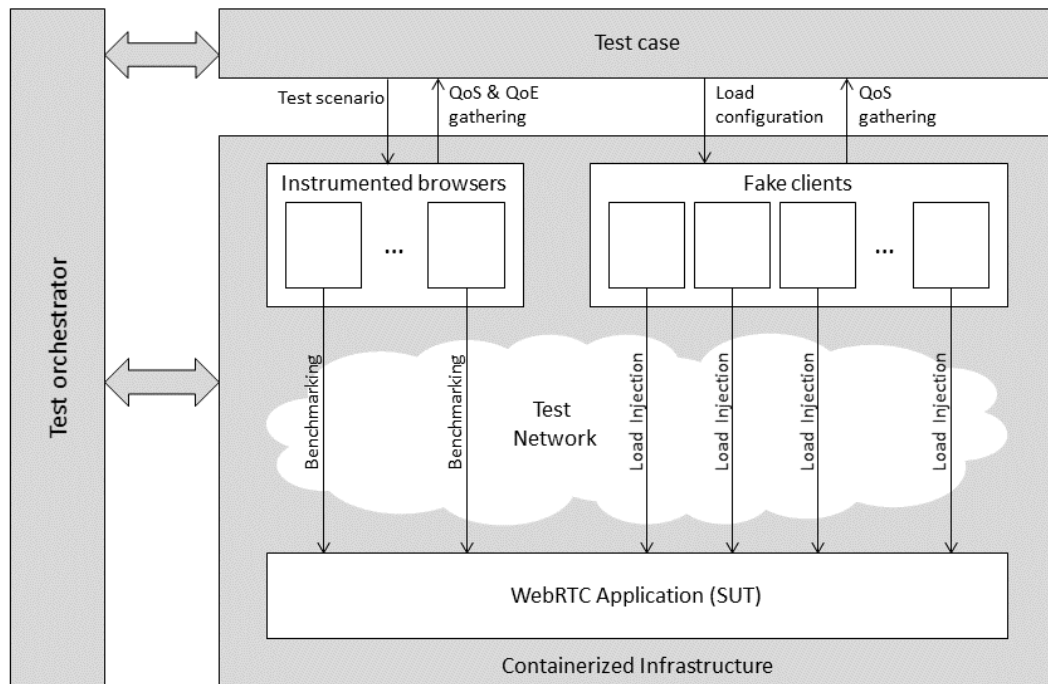
Figure 1. WebRTC testing architecture.

As an additional requirement, we wish the tests thus created to be manageable through state-of-the-art CI tools (e.g. Jenkins). Due to this, all our testing logic is built around a *test orchestrator* capability that has been created by extending JUnit, one of the most popular Java testing frameworks that is supported seamlessly and off-the-shelf by most CI systems. Thanks to this, all the tests are created as JUnit testing jobs or workers that rely on our toolbox for instantiating and controlling instrumented browsers and fake clients. A relevant challenge that emerges with this approach is on how to manage the testing computing resources. Both, instrumented browsers and fake clients require computing and networking resources that, depending on the scale of the test, may be relevant to scale in or out dynamically for adapting to the traffic needs. For this, we need to use some kind of IaaS (Infrastructure as a Service) capability suitable for provisioning and managing the virtual computing resources we need. After some experimentation, and given our agility requirement, that mandates our test to execute consuming the less possible time, we decided to use containers (i.e. Docker) instead of traditional hypervisor-based virtualization technologies. Containers provide a very relevant reduction in virtual computing node startup time, expose better performance and are a lighter-weight solution with more reduced memory and CPU fingerprint [7].

WebRTC functional evaluation

As a first approach for functional assessment, our test orchestrator provides a subscription capability for media events. WebRTC media is played in browsers using the HTML5 video element. Several media events are triggered by the video tag, for instance, play, pause, seek, change of volume, mute, or change of the playback rate. Within the test logic, developers are able to create subscriptions to any of these video tags events, allowing to create simple but powerful test assertions aimed to evaluate if the subscribed events happen in a given time. For example, this mechanism can be used to evaluate the call setup time, simply measuring the time between the session start request and the actual time in which the playing event is received in the video tag.

Another challenge is to find out if the media reaching the browser is as expected in an automated manner. A simple but effective approach to address this question is analyzing the color of the received media in the HTML5 video tags. To do that, our toolbox provides an API to assert the expected color given the Cartesian coordinates within a video tag. Then, a built-in comparison method based on the calculation of the RGB (Red, Green, Blue) component distance from the expected and the real color can be used to assess whether or not the received color is equal (or similar, using a threshold) to the expected one.

WebRTC non-functional evaluation

As any experienced tester knows, testing activities are aimed to provide useful answers about a SUT. To do that, testers should ask the proper questions in form of test cases. So far, we are able to assess whether or not WebRTC media is reaching different types of browsers. But once the call is established, how do we find out if that media is as expected in an automated manner? As Dr. Alfred Lanning (I, Robot) would say: "That, detective, is the right question".

In order to address this question, WebRTC services first need to be measured objectively based on key performance indicators (KPI). In this context, we need to understand that RTC applications should enable and ease as much as possible the human-to-human communication. That means that media (typically audio and video) should have a minimum quality to be understood by the receiver. Moreover, RTC applications imply a hard restriction related to having a low latency (less than 300ms [8]) to allow a conversational (bidirectional) communication. All in all, our approach allows measuring these indicators into two groups: network-dependent QoS and media-dependent QoE.

Regarding QoE, the most widely accepted way to classify the metrics is based on subjective or objective methods. Subjective methods are conducted to obtain information on the quality of multimedia services using opinion scores, while objective methods measure the difference between the original media and the resulting media after processing (encoding, transmission through the network, decoding, rendering, etc.). Subjective measurement is time consuming, and is not particularly applicable in test environments. Instead, objective methods can be used, namely [9]:

- Traditional point-based metrics. For example, peak signal-to-noise ratio (PSNR), which is the proportion between the maximum signal power and the corruption noise power [10].

- Natural visual characteristics oriented metrics. For example, structural similarity (SSIM), which predicts the perceived quality of images and videos based on its differences. The SSIM index is a decimal value between 0 and 1, where 1 is only reachable for two identical sets of data [11].

- Perceptual oriented metrics. For example, perceptual evaluation of speech quality (PESQ), which analyzes the degraded audio signal through the network. As a result, a mean opinion score (MOS) is provided. With this metric, the audio quality is assessed using a five-point scale: 1 = bad, 2 = poor, 3 = fair, 4 = good and 5 = excellent [12].

For the sake of completeness, all these QoE metrics (PSNR, SSIM, and PESQ) are supported out-of-the-box by our test orchestrator. For that, the toolbox uses the local media rendered by the sender browser as original media, and the remote media rendered by receiver browser. On the one hand, the MOS value provided by PESQ can be directly used as the expected outcome in a test assertion (for instance, a test case can be declared as failed when MOS is lower than 3) to assess the audio quality. On the other hand, PSNR and SSIM can be used in test assertions using statistics or threshold values to evaluate the video quality (for instance, declaring a test failed when the average SSIM is less than 0.8).

Regarding QoS, our toolbox offers the possibility of gathering all the standard WebRTC statistics implemented by instrumented browsers and fake clients to developers. Among all these data, the following indicators are key to find out the QoS of the underlying network:

- Packet loss. By default, WebRTC uses UDP as transport protocol for media. That means that IP packets can be dropped by routers to mitigate potential congestion issues. Even though WebRTC implements adaptive hybrid packet retransmissions based on NACK and forward error correction (FEC) to handle packet loss [13], overall video quality due to packet loss can be significant.
- Bandwidth. WebRTC codecs (e.g. G.711, OPUS, VP8, among others) are lossy, meaning that they encode to save on space usually by making assumptions around the human eye and ear and also by external restrictions such as the bandwidth available.
- Jitter, which is the variation in the latency on a packet flow between two systems, when some packets take longer to travel than others. Jitter in RTC applications can lead to audio and video unintended deviations that degrade the quality of communications.

As said before, having a low latency in RTC applications is critical. Strictly speaking, latency is a QoS indicator due to the fact it depends directly on the underlying network, but it can also be affected by the mechanisms used to mitigate jitter (jitter buffer, which adds some latency to provide continuous and smooth media flow) or packet loss (retransmissions). Taking this into account, our toolbox is able to measure the total end-to-end latency using the synthetic video provided by Chrome to substitute the real user media. This video consists on a green video with a spinner which completes a spin and beeps each second. The content of the video includes a timer and the number of frames, which is sent within the media (see Figure 3 in next section). Our test orchestrator uses an Optical Character Recognition (OCR) applied to local media rendered by the sender browser and the remote media rendered by the receiver browser. The end-to-end latency is calculated as the time recognized in the presenter less the time recognized in the viewer.

WebRTC test network

WebRTC applications have to deal with the heterogeneous nature of the networks, for example, NAT traversal, firewalls, etc. Web developers expect having HTTP request and responses to pass through NATs and firewalls, because the related ports are usually open by default, but this is not the case for RTC communication, where developers need to carefully configure their applications to ensure the WebRTC connectivity in all network topologies.

This is why WebRTC's standard Interactive Connectivity Establishment (ICE) [14] was developed. ICE's main objective is to discover a network path between two peers that can be used to maintain the connectivity. The discovering process may need the help of additional servers like STUN and TURN. On the one hand, STUN is used by an endpoint to determine the IP address and port allocated to it by a NAT. On the other hand, TURN server acts as communication relay that is used when peers cannot communicate directly for some reason (for instance a firewall prevents UDP packets, the NAT involved does not provide hair-pinning, etc.).

To allow developers to test if their applications are correctly configured and deployed, we have included a customizable network environment (labeled as *Test Network* in Figure 1) between the clients and the SUT. The test orchestrator is able to reproduce all possible network scenarios so that we can assure that our application will work independently of the network topology and conditions, namely:

- Both, SUT and browser, behind a NAT. In this case, a STUN server is provided and configured automatically in both elements. As shown in Figure 2, the STUN server is connected to the Docker network because it simulates public Internet.
- Both, SUT and browser, behind a NAT, one of them is only capable of TCP connections (UDP is disabled in that peer). In this case, a TURN server is also provided.
- Only SUT or browser behind a NAT, the other peer can be reached directly. In this case, a STUN server is provided.
- Only SUT or browser behind a NAT, the other peer can be reached directly. However, in this scenario, the peer behind NAT has also a firewall preventing UDP packets (only TCP connections on that peer are enabled). In this case, a TURN server is provided.

In addition, the test orchestrator also provides advanced capabilities to simulate different traffic scenarios in the network (*Network traffic control* component in Figure 2), namely:

- Packet filtering. Different rules can be added in order to simulate losses in the UDP or TCP traffic.
- Latency configuration. Test orchestrator is able to add extra latency in the test network in order to simulate real-world latency scenarios.
- Bandwidth configuration. The toolbox offers some APIs to configure maximum bandwidth limitations per port. This can be used to simulate networks such as DSL, 3G, or 4G.
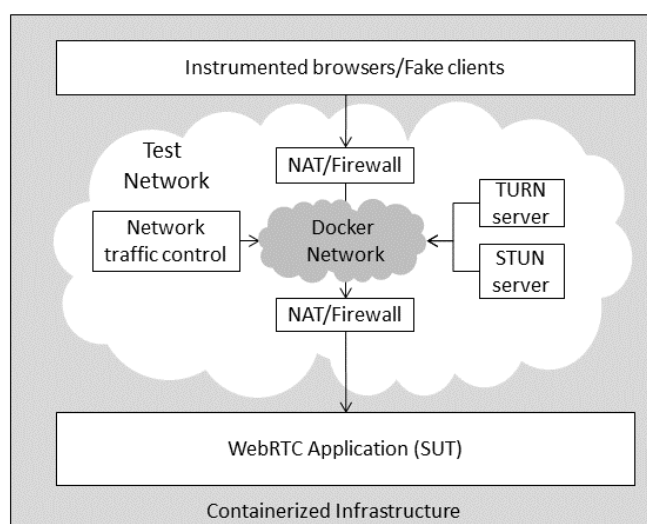

Figure 2. Test network structure.

All in all, the test network component provides a rich set of scenarios to ensure that a RTC communication can be established no matter which network topologies are involved between the SUT and the clients. This way, the test orchestrator is able to setup most real network conditions that can be found on the Internet.

Case study: scalability of WebRTC broadcasting

In order to validate our proposal, we have carried out a case study using one the most demanded services in the WebRTC arena: the broadcasting (i.e. one-to-many video communication). In this service, there are two kinds of entities involved in the communication: presenter (peer sharing media) and viewers

(peers receiving the media from the presenter distributed by a media server). Our application (see Figure 3) uses Kurento as Media Server and has been deployed on NUBOMEDIA, which is an open source elastic cloud PaaS (Platform as a Service) specifically designed for real-time interactive multimedia services [15]. The machine hosting the service is a medium-size cloud instance (2 VCPU, 4 GB RAM).
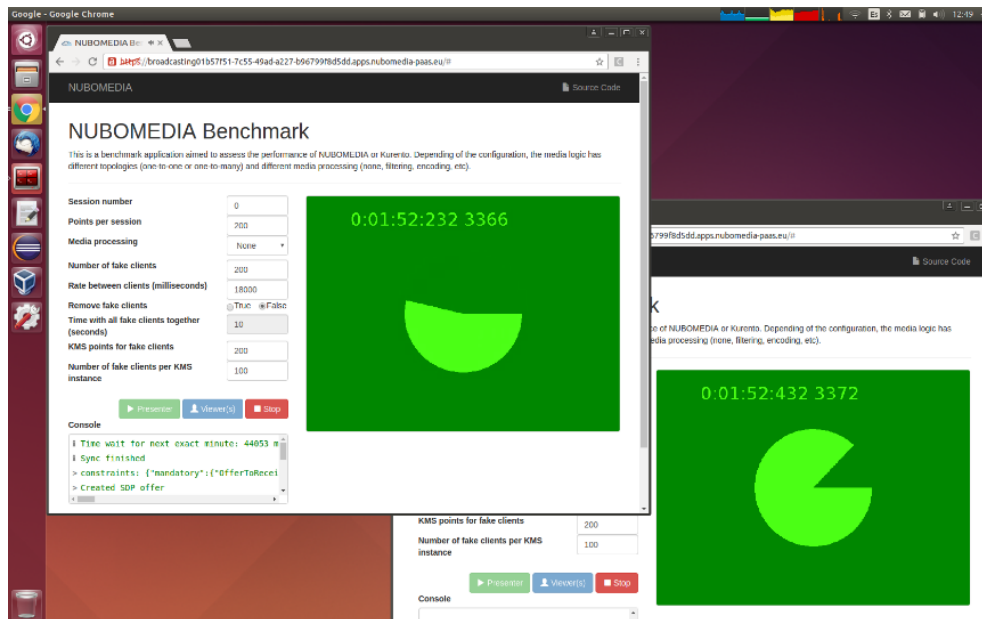

Figure 3. NUBOMEDIA Benchmark WebRTC application.

The research question driving this case study has to do with the scalability of the service, and it can be formulated as follows: How many viewers can be connected to a presenter keeping good levels of quality? To address this question, we use our toolbox to implement a test case which benchmarks the SUT. Regarding the test scenario, we define two instrumented browsers: one for the presenter and other for the first viewer. Then, up to 200 fake clients acting as viewers are connected to the broadcasting session. The application consists on a Spring-Boot Java application which consumes a cloud Kurento Media Server hosted on NUBOMEDIA. This application together with the browsers is containerized inside the testbed. During the test execution, different metrics are collected: end-to-end latency and SSIM video quality. The WebRTC user media has the following features:

- Video: resolution 640x480px, 30fps, VP8 codec
- Audio: 1 channel, 48.0kHz sample rate, Opus code

In order to introduce a control variable, we repeat the experiment introducing a packet loss of the 25% in the UDP traffic. The rest of network parameters (i.e. latency and bandwidth configuration) are not changed in this case study.

After the test execution, we analyze the gathered data. Figure 4 illustrates the results obtained for the end-to-end latency. As can be seen, the mean latency for the first experiment (without packet loss) is around 200ms, whilst in the second experiment is around 400ms. An important finding is that both scenarios find the same bottleneck when the number of clients reaches approximately 175. This number identifies the upper threshold for this broadcasting service supported by a medium-size cloud instance of Kurento Media Server (over 175 viewers the service cannot be considered real-time anymore).
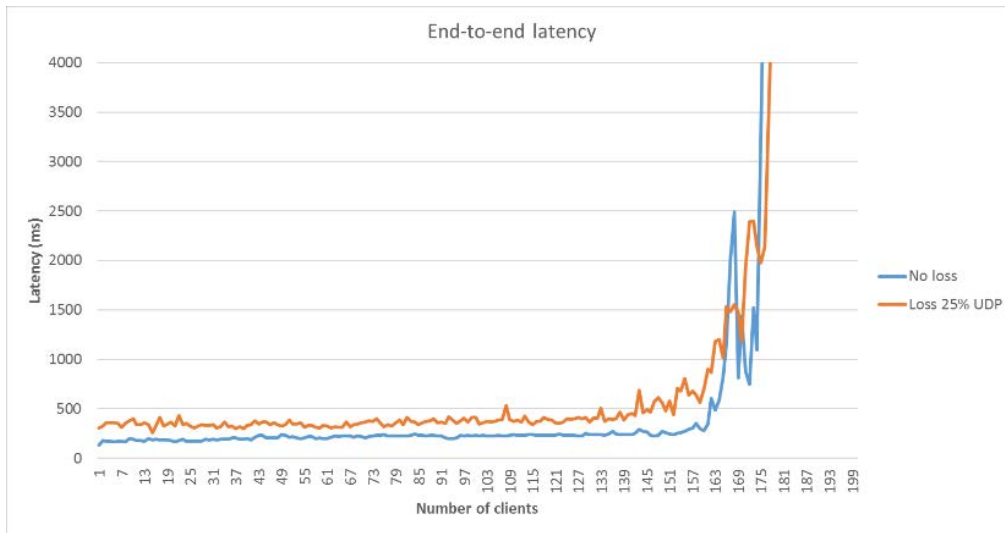
Figure 4. End-to-end latency results.

On the other hand, Figure 5 depicts the results in terms of video quality using the SSIM metric. This chart is interesting since a different boundary of the system is exposed. Regarding the case without loss, the same limit (around 175 clients) is shown clearly in the chart, since the quality remains stable so far, with a mean value around 0.965. Nevertheless, in the scenario with UDP losses the scalability of the system is reduced, because the upper bound of clients falls to 121, when the resulting SSIM drops to values near to 0. This is due to the packet loss detected by the congestion control of WebRTC, which has a direct consequence of a bit rate reduction and therefore the video resolution reduction. As a result, the media in the viewers is degraded too much and the perceived quality falls below the operational margin.
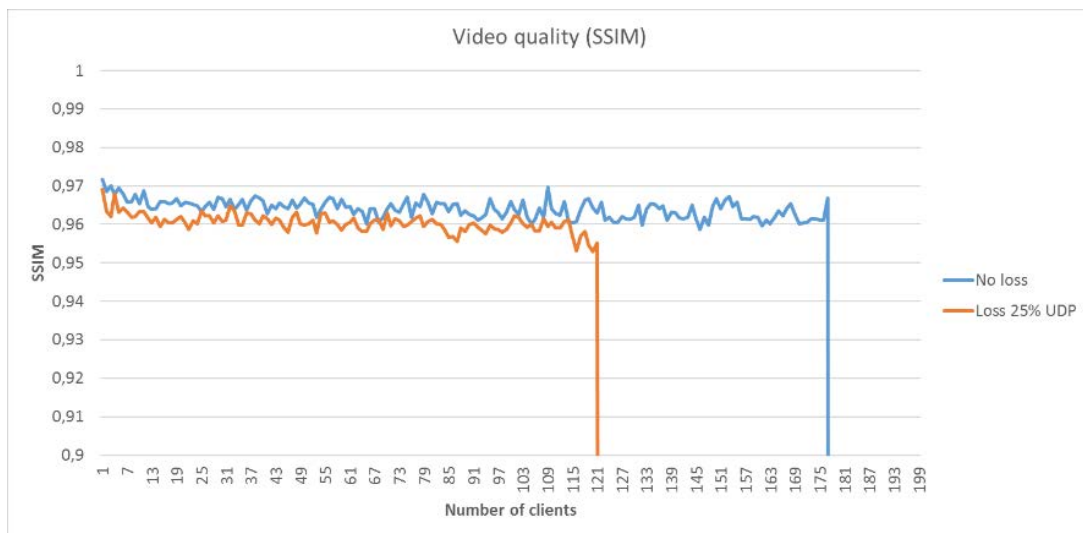

Figure 5. Video quality results.

Due to the benchmarking nature of this case study we have not included any predicate in the test case. Nevertheless, it would be straightforward to add assertions in order to obtain a verdict about the test execution, declaring the test as failed when for example the average latency is above 1 second or the SSIM index is below 0.8.

Conclusions

WebRTC is a set of standards aimed to provide media capabilities in an easy way to Web applications. However, there is still a need to test these applications to ensure a proper user experience. This is where standard testing practices, as known and applied by Web developers, are not enough. Testing media in Web applications requires functionalities like inject media in the system, measuring media quality, timings, testing whether media is being received, etc. In addition, developers may want to test common network scenarios, like having clients connecting from DSL, 3G or 4G networks, or initiating a communication behind a NAT or firewall. All these scenarios need to be considered when testing and benchmarking a WebRTC enabled application.

In our journey on developing Kurento, an open source media server that complies with the WebRTC standards, we have made a big effort on assessing WebRTC applications and services. As a result, we have designed a methodology and an associated tool based on well-known frameworks such as JUnit and Selenium. This technology is able to provide high level semantics in terms of QoE/QoS indicators, which enables them to write rich test assertions in functional tests. The metrics are gathered by instrumented browsers started as Docker containers, which also enables to carry out compatibility tests in a seamless way. Moreover, our test infrastructure enables to perform load benchmarking without wasting a lot of computing resources by introducing fake clients which implement a lightweight standard WebRTC stack. This allows to evaluate WebRTC services at scale. Finally, the use of Docker containers enables configuring complex networking scenarios where NAT traversal, firewalls or different traffic loss, latency, and bandwidths are enforced. This allows developers to perform end-to-end tests where the experience of the end user is assessed by running tests in situations as close to the real world as possible. As part of the future work, we plan to enhance the network traffic control component by supporting network handover, i.e. the capability of switching between a network setup to another (for instance from 4G to 3G) during a WebRTC session.

References

[1] Y.-F. Li, P. K. Das, and D. L. Dowe, "Two decades of Web application testing: A survey of recent advances", *Information Systems*, vol. 43, pp. 20–54, 2014.

[2] A. Johnston, J. Yoakum, K. Singh, "Taking on WebRTC in an enterprise", *IEEE Communications Magazine*, vol. 51, no. 4. pp. 48–54, 2013.

[3] S. Sale and T. Rebbeck, "Operators need to engage with WebRTC and the opportunities it presents", Analysis Mason Oct. 2014; http://www.analysysmason.com/About-Us/News/Insight/WebRTC-operator-opportunities-Oct2014-RDMV0

[4] E. H. Halili, *Apache JMeter: A practical beginner's guide to automated testing and performance measurement for your websites*. Packt Publishing Ltd, 2008.

[5] S. Taheri et al., "WebRTCbench: a benchmark for performance assessment of WebRTC implementations", *13th IEEE Symposium on Embedded Systems for Real-time Multimedia (ESTIMedia),* pp. 1–7, 2015.

[6] Amirante, A. et al, "Jattack: a WebRTC load testing tool", In *Principles, Systems and Applications of IP Telecommunications (IEEE IPTComm),* pp. 1–6, 2016.

[7] D. Bernstein, "Containers and cloud: From LXC to Docker to Kubernetes", *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, 2014.

[8] W. Cellary and A. Iyengar, *Internet Technologies, Applications and Societal Impact: IFIP TC6 / WG6.4 Workshop on Internet Technologies, Applications and Societal Impact*. Springer, 2013.

[9] H. A. Tran et al., "QoE-based server selection for content distribution networks", *IEEE Transactions on Computers*, vol. 63, no. 11, pp. 2803–2815, 2014.

[10] Q. Huynh-Thu and M. Ghanbari, "Scope of validity of PSNR in image/video quality assessment", *Electronics letters*, vol. 44, no. 13, pp. 800–801, 2008.

[11] Z. Wang et al., "Image quality assessment: from error visibility to structural similarity", *IEEE Transactions on Image Processing*, vol. 13, no. 4, pp. 600–612, 2004.

[12] M. Viswanathan and M. Viswanathan, "Measuring speech quality for text-to-speech systems: development and assessment of a modified mean opinion score (MOS) scale", *Computer Speech & Language*, vol. 19, no. 1, pp. 55–83, 2005.

[13] S. Holmer, M. Shemer, and M. Paniconi, "Handling packet loss in WebRTC", *IEEE International Conference on Image Processing*, vol. 9, 2013.

[14] Müller, A., Carle, G., Klenk, A., "Behavior and classification of NAT devices and implications for NAT traversal", *IEEE Network*, vol. 22, no. 5, pp. 14–19, 2008.

[15] B. García et al., "NUBOMEDIA: an Elastic PaaS Enabling the Convergence of Real-Time and Big Data Multimedia", *IEEE International Conference on Smart Cloud (SmartCloud)*, pp. 45–56, 2016.

Biographies

Boni García has a PhD degree on Information and Communications Technology from Universidad Politécnica de Madrid (Spain) in 2011. His main research interests are software testing, web engineering and computer networking. Currently he is working as a Researcher at Universidad Rey Juan Carlos and Assistant Professor at Centro Universitario de Tecnología y Arte Digital (U-tad) in Spain. He is member of Kurento project, where he is in charge of the testing framework for WebRTC applications.

Francisco Gortázar holds a PhD degree on Computer Science. He is Associate Professor at Universidad Rey Juan Carlos. His main research activities are related to Software Engineering and Software Testing. He is member of the Kurento project, where he is focused on developing infrastructure support for testing distributed applications. He leads the ElasTest project focused on end-to-end testing on large distributed software systems.

Luis Lopez is associated professor at URJC and lead of the Kurento.org open source software project. Dr. Lopez research interests are concentrated on the creation of advanced multimedia communication technologies and on the conception of Application Programming Interfaces on top of them. His ideas

have generated more than 60 technical publications and have been included into important research and industrial projects including FIWARE and NUBOMEDIA.

Micael Gallego earned a PhD on Computer Science from Universidad Rey Juan Carlos (Spain) in 2008. Among other scientific publications of high impact, he is the coinventor with an AT&T researcher of an American patent. He has participated in three national research projects, from the Spanish Research Agency, and in two European research projects. He is software architect in the Kurento project, where he is focused on developing scalable and fault tolerance distributed systems.

Miguel París is software engineer and has a MSc in Telematics Systems. He works as researcher in new multimedia systems and is the manager of real-time communication area in Kurento team, where is the responsible of the WebRTC stack. He has participated in the design of Kurento architecture and APIs and in the development of Kurento media elements. In addition, he has contributed to GStreamer community with some patches and discussions about RTP stack.