Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Cost-effective load testing of WebRTC applications[☆]

Francisco Gortázar[*], Micael Gallego, Michel Maes-Bermejo, Iván Chicano-Capelo, Carlos Santos

Escuela Técnica Superior de Ingeniería Informática, Universidad Rey Juan Carlos, Spain

A R T I C L E   I N F O

Article history:
Received 8 April 2021
Received in revised form 30 April 2022
Accepted 14 July 2022
Available online 2 August 2022

Keywords:
Testing
Load testing
WebRTC

A B S T R A C T

Background: Video conference applications and systems implementing the WebRTC W3C standard are becoming more popular and demanded year after year, and load testing them is of paramount importance to ensure they can cope with demand. However, this is an expensive activity, usually involving browsers to emulate users.
Goal : to propose browser-less alternative strategies for load testing WebRTC services, and to study performance and costs of those strategies when compared with traditional ones.
Method: (a) Exploring the limits of existing and novel strategies for load testing WebRTC services from a single machine. (b) Comparing the common strategy of using browsers with the best of our proposed strategies in terms of cost in a load testing scenario.
Results: We observed that, using identical machines, our proposed strategies are able to emulate more users than traditional strategies. We also found a huge saving in expenditure for load testing, as our strategy suppose a saving of 96% with respect to usual browser-based strategies. We also found there are almost no differences between the traditional strategies considered.
Conclusion: We provide details on scalability of different load testing strategies in terms of users emulated, as well as CPU and memory used. We could reduce the expenditure of load tests of WebRTC applications.

1. Introduction

With the rise of remote jobs, businesses transiting online, or online courses, video conferencing has become an integral part of our lives. Video conference applications and systems are becoming more popular and demanded year after year. On top of this, the video conferencing market is experiencing high growth during the coronavirus outbreak. Whether this situation will persist after the coronavirus crisis or not is still something we do not know, nevertheless, the demand is pushing companies in the video conferencing market into a situation in which load testing their solutions is of paramount importance to ensure solutions can cope with demand.

Among the different technologies for real time video communication over the Internet, the World Wide Web's (W3C) Web Real-Time Communications (WebRTC) standard (WebRTC, 2022) is a popular one. Video conference tools like BigBlueButton (2022), Jitsi (2022), Whereby (2022), among others all rely on the WebRTC standard to enable video calls for a set of users.

WebRTC is a set of protocols and APIs that provides web browsers and mobile applications with Real-Time Communications (RTC) capabilities over peer-to-peer connections. It was conceived to allow connecting browsers without intermediate helpers or services, but in practice this P2P model falls short when trying to create more complex applications. For this reason, in most cases a central media server is required. Conceptually, a WebRTC media server is just a multimedia middleware where media traffic passes through when moving from source(s) to destination(s). Media servers are capable of processing incoming media streams and offer different outcomes, such as:

- Group Communications: Distributing among several receivers the media stream that one peer generates, i.e. acting as a Selective Forwarding Unit ("SFU").
- Mixing: Transforming several incoming streams into one single composite stream, i.e. acting as a Multipoint Conferencing Units ("MCU").
- Transcoding: On-the-fly adaptation of codecs and formats between incompatible clients.
- Recording: Storing in a persistent way the media exchanged among peers.

In WebRTC, users of a group call (from now on session) are usually browsers exchanging real-time video and audio through

☆ Editor: Prof Raffaela Mirandola.
* Corresponding author.
E-mail addresses: francisco.gortazar@urjc.es (F. Gortázar), micael.gallego@urjc.es (M. Gallego), michel.maes@urjc.es (M. Maes-Bermejo), ivan.chicano@urjc.es (I. Chicano-Capelo), carlos.santos@urjc.es (C. Santos).

https://doi.org/10.1016/j.jss.2022.111439

a WebRTC media server. The number of users can vary, and the media server is usually able to handle many of such sessions. Testing group calls is usually expensive. Test cases are usually executed using several browsers to impersonate real users by exchanging preconfigured video and audio in real-time through a media server. Web browsers are heavy resource consumers, and usually a single browser impersonates a single user thus limiting scalability and a better use of resources. The challenges of testing video conference systems have been discussed in the past in the literature (García et al., 2017a; Gouaillard and Roux, 2017; Bertolino et al., 2020; Garcia et al., 2017b).

In this paper: (a) we propose two new testing strategies that avoid the need of a browser, and significantly reduces the expenditure of testing WebRTC based video conference applications; (b) we present an open source framework for load testing WebRTC applications in the cloud that greatly simplifies this task, using any of the 5 testing strategies considered in this work; (c) we perform a comparative study of different testing strategies for real-time group communications at scale using the WebRTC standard. In the study, we used the OpenVidu WebRTC platform (OpenVidu, 2022), which is open source, and experiments were conducted in the cloud using Amazon Web Services.[1] We paid attention to benefits and drawbacks of the scrutinized strategies, the performance in terms of number of users per session, the use of resources, and the total expenditure of the tests.

We decided to compare traditional WebRTC testing strategies with our new proposals with two aims:

(1) To measure the scalability of each testing strategy within a single AWS machine (instance in AWS jargon) in terms of number of users that can be impersonated from that machine, by answering the following research questions:

- RQ_{1a}: What is the strategy that is able to emulate more users into sessions? Is this consistent with the session size?
- RQ_{1b}: How do each of the strategies perform in terms of CPU and memory with respect to the number of users?

(2) To compare the costs of our most promising testing strategy with traditional browser-based strategies in a load test scenario, where multiple machines are needed to impersonate users:

- RQ_2: Are there any cost savings for using our proposals with respect to using browsers when doing load testing of a WebRTC platform?

With the first set of research questions we try to understand the scalability and limits of each of the strategies in a single machine, and the correlation of the strategy with the use of resources. We used different testing scenarios, where a scenario consists of a given number of WebRTC sessions of a certain size (all sessions have the same size). This information is later used to answer the final research question, by conducting a load test scenario with the most promising testing strategy. In this scenario, several testing machines are used, up to the limits defined in the first set of research questions, to explore the capacity of a WebRTC server in terms of users, and determining the cost savings of the strategies considered.

The rest of the paper is structured as follows: Section 2 discusses previous research in WebRTC testing (including load or load testing of WebRTC services). In Section 3 we introduce the OpenVidu platform. Traditional testing strategies, and our new proposals for WebRTC load testing are presented in Section 4. We describe the methodology used in the study in Section 5. Results of applying the methodology are reported in Section 6, including a discussion on the threats to validity. Finally, Section 7 draws conclusions and presents further research.

1 https://aws.amazon.com/.

2. Background

Testing and quality assurance of web applications has the challenge of having to test heterogeneous applications (Li et al., 2014; Bertolino, 2007). According to Di Lucca and Fasolino (2006), the main testing activities for non-functional requirements that a Web application is usually required to accomplish are: performance testing, load testing, compatibility testing, usability testing, accessibility testing and security testing. In the remaining of the section strategies and tools from the literature (including gray literature) focused on generic testing, as well as load and load testing, are presented.

2.1. Generic WebRTC testing tools

Among the WebRTC testing tools, Selenium[2] is one of the most widely used, as was reported previously by Garcia et al. (García et al., 2017a). Selenium enables programmatic managing of browsers using different programming languages, such as Java, Python, PHP or JavaScript. As most browsers currently implement the WebRTC stack, any WebRTC-enabled Selenium managed browser can be used to impersonate users for WebRTC testing. However, using browsers implies a considerable expenditure of computational resources. To avoid investment in infrastructure, users can resort to commercial providers of browsers such as Saucelabs,[3] BrowserStack[4] and Nightwatch.js,[5] in a pay-per-use approach.

The Kurento Testing Framework (KTF) (García et al., 2016) provides a set of browser-based tools for testing WebRTC applications. It allows to automatically evaluate functional parameters such as media events or color detection (for instance, to build oracles based on video synchronization), it collects performance statistics, and it is able to extract quality of experience metrics (by evaluating audio quality). This testing framework is a part of Kurento (Kurento Media Server, 2022), a WebRTC platform, and it again uses Selenium for the connection with the browsers.

ElasTest[6] is a comprehensive open source platform that aims to significantly improve the efficiency and effectiveness of testing complex systems. It is a generic tool, not exclusively focused on WebRTC testing, although it includes interesting features for WebRTC testing. One of its strengths is observability, ElasTest platform provides a complete monitoring system for the software under test. When confronted with WebRTC testing, ElasTest, in addition to provide Selenium controlled browsers, it automatically obtains specific metrics of the WebRTC connection that browsers make available for inspection (the WebRTC stats defined within the WebRTC W3C standard).

KITE (Gouaillard and Roux, 2017) is another framework for WebRTC peer-to-peer test automation supported and managed by companies actively involved in the development of the WebRTC standard. It is supported by other technologies such as Selenium and browser vendors. It allows interoperability testing between different browsers and operating systems. The main purpose of the project is to detect the level of compliance of each browser vendor with the WebRTC standard.

2.2. WebRTC load testing tools

Some well-known load testing tools for web application such as Apache JMeter, Artillery or Gatling cannot be used for WebRTC

2 https://www.selenium.dev/.
3 https://saucelabs.com/.
4 https://www.browserstack.com/.
5 https://nightwatchjs.org/.
6 https://elastest.io/.

testing, as they do not use a browser or any implementation of the WebRTC stack. Therefore, these tools are out of scope. Other approaches like (Shariff et al., 2019) allow for a better utilization of resources by using waits in a browser for running several tests in parallel, however this would not work in WebRTC, as the media flows as a continuum, and there are no waits.

However, there are other tools in the literature aimed at load testing a WebRTC media server. WebRTCBench (Taheri et al., 2015) is a WebRTC benchmark which measures performance on peer-to-peer communication, allowing to evaluate the performance across different platforms and devices, including the most popular browsers for desktop and Android platforms. The benchmark is limited to two peers, and therefore cannot be used to analyze the performance of media servers at scale, where many peers are connected simultaneously arranged into different media sessions.

In Amirante et al. (2016), the authors introduced JAttack, a WebRTC load testing tool for performance analysis of the WebRTC media server Janus. This tool uses a modified Janus media server to generate multiple WebRTC connections in order to load the Janus media server (Amirante et al., 2014). Although the authors performed some experiments loading a Janus WebRTC media server, they only reported CPU usage of the testing tool and the media server. Furthermore, both the load testing tool (the modified Janus server) and the Janus media server were running together in the same machine, which raises some concerns about the validity of the results they got. Additionally, they did not perform a comparison and in depth analysis of the costs compared to traditional browser-based approaches. Finally, JAttack has not been released to the public.

The main commercial option is testRTC (2022), which includes a wide variety of solutions for WebRTC application testing, including load testing where network conditions can be configured to build realistic scenarios. As far as the authors can tell, testRTC prices are not publicly available, so cost analysis cannot be performed.

2.3. Testing in the cloud

Cloud computing can provide users cost-effective and flexible scalable computing power and services. According to Gao et al. (2011), the main benefits of cloud-based testing are: reduced costs of computing resources by leveraging them to clouds (using virtualized resources and shared infrastructure), existing on-demand test services for large-scale and effective real-time online validation and easily leverage scalable cloud system infrastructure to test and evaluate system performance and scalability.

Bertolino et al. (2019) did a systematic review on cloud testing. They divided their study into testing in the cloud (leveraging the elasticity of cloud for testing), testing of the cloud (testing cloud applications) and testing of the cloud in the cloud (mixing both approaches). The authors conclude that testing in the cloud has indeed received much attention, being test execution the most prominent usage of cloud providers in the context of the different testing activities.

In André et al. (2018), the authors tested the scalability of 4 WebRTC media servers (Medooze, Jitsi, Janus, and OpenVidu/Kurento), giving details of the infrastructure used. They used the AWS Elastic Compute Cloud (EC2) service, separating the media server and the web clients on different instances. Specifically, the authors used the c4.4xlarge (8 vCPUs, 30 GB RAM) instance type for the media servers and the c4.xlarge (4 vCPUs, 7.5 GB RAM) instance type for the clients. The paper focuses mainly on the number of users supported by each media server, using KITE as the WebRTC testing platform. No analysis of costs was included in the work.

3. WebRTC testing strategies

Before introducing the different WebRTC testing strategies considered in the study, we present an overview of the architecture of WebRTC services and applications. In this section, we first sketch how WebRTC applications work and the services involved, and then we resort to detail how they are tested. Finally, we present our proposals for cost-effective load testing of WebRTC applications.

3.1. Anatomy of a WebRTC application

A WebRTC application comprises several components: (1) the application itself, running in the browser; (2) a library that enables the application to embed videoconference into the page; and (3) a media server that receives and routes streams of audio and video to all users in a session. The library in (2) makes transparent for the application how the WebRTC protocols are used, thus easing the embedding of video into the app. For instance, the library instructs the browser to fetch video and audio through the corresponding devices (camera and microphone) and it sends the streams to a media server through the WebRTC API.

Usually, media servers are able to handle multiple sessions running in parallel. For each session, a user in the session (connected through a browser) sends audio and video streams to the media server, which in turn, sends the video and audio streams that it receives from all other users in the same session to the user. Given a set of users in a session $N, |N| = n$, each user $x \in N$ sends 2 streams and receives $2 * (n-1)$ streams from the other $n-1$ users, $y \in N, y \neq x$ in the session. Depending on how the application is implemented, this might impose a limit on the number of users that a browser is able to handle for the session. For instance, Whereby limits to 50 the number of users in a session.

At the media server, the number of streams to be managed grows with the number of users in a session. For a session with 2 users, the media server receives 4 streams (2 per user), and it sends 2 streams to each user. With 3 users, the media server receives 6 streams, and it sends 4 streams to each user. This means $4 * 3 = 12$ streams to be sent in total to 3 different users. Therefore, the number of streams s to be sent depends quadratically on the number of users: $s = 2n*(n-1) = 2n^2 - 2n$.

It is therefore of paramount importance to properly plan the size of the machine hosting and running the media server, as it will have to route many streams to different users in different sessions. In order to do so, proper load tests need to be performed on the media server. As it was mentioned in Section 2, traditional approaches are based on web browsers to impersonate users in media sessions. This strategy works well, but it has a considerable cost. It is worth noticing that this load testing might be needed for many scenarios, with different number of users in each session.

3.2. OpenVidu WebRTC platform

In order to study testing strategies for WebRTC we need a WebRTC application. OpenVidu (2022) is an open source platform to enable embedding video conferencing capabilities into modern web and mobile applications. It provides both the browser library and the media server (the Kurento Media Server, KMS). In this section, we describe the OpenVidu architecture and how we leverage it to perform load tests in order to find the limits of the platform. We will conduct our comparative study using the OpenVidu platform and the OpenVidu Loadtest Tool (OVLT).

The OpenVidu platform comprises three modules as shown in Fig. 1:

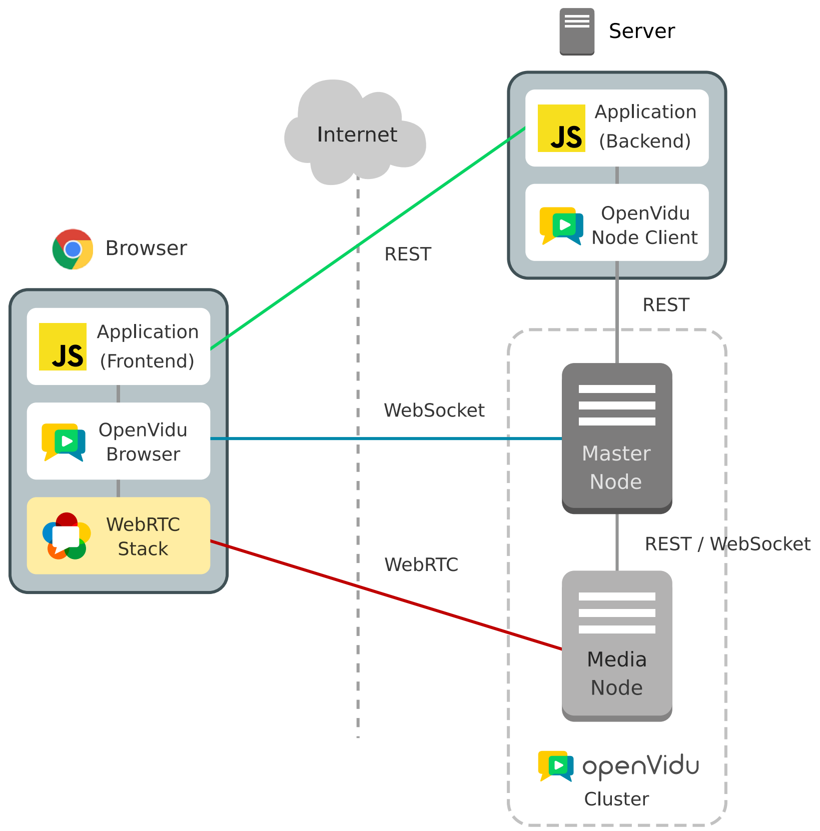

Fig. 1. OpenVidu architecture. The OpenVidu platform consists of a master node and one or more media nodes. An application using the OpenVidu platform consists of a backend and a frontend. The openvidu-browser library is used by the frontend of the application to make use of OpenVidu WebRTC capabilities. The backend is usually used to provide security. Upon client authentication (green line), negotiation between the openvidu-browser library and the master node (blue line) results in a media node selection with whom the media will be exchanged with the help of the browser's implementation of the WebRTC stack (red line). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

- **openvidu-browser**: A JavaScript library for the browser. It allows developers to embed video in their own applications. It is responsible for the control plane and leverages to the WebRTC stack provided by the browser for the media plane.
- **OpenVidu master node**: Executes an application that controls one or more media nodes. It takes care of the control plane, and it manages sessions, forwarding events and messages to clients and distributing the load across the available media nodes.
- **OpenVidu media node**: Executes the media server responsible for the media plane. OpenVidu currently uses the Kurento Media Server, an open source implementation of an WebRTC media server. OpenVidu platform can use several media nodes. They are the actual bottleneck of the OpenVidu platform and their number and size determine its capacity: more media nodes means more concurrent sessions possible, which results in an increased spending.

OpenVidu provides a basic application, but developers can build their own application using the OpenVidu APIs. A WebRTC application based on OpenVidu consists on a backend and a frontend. In the backend, authentication and authorization of users is managed. The frontend requests session joins for a user through the backend, and upon authorization, a token is returned to the frontend. This token is used by openvidu-browser to connect to the master node asking for a specific session to join, and the master node selects which media node to use for the session. Thus, the master node is in charge of the control plane, managing where each session is hosted (i.e., in which media node). When the media node is selected, openvidu-browser uses the browser WebRTC API to negotiate how to exchange media between the browser and the media server. The media server and the browser WebRTC API are in charge of the media plane.

Notice that all the users of a session must all be connected to the same media node. That is, a session is fully contained into a single media node. A media node is able to handle several sessions concurrently. The specific number of sessions that a media node is able to handle depends on the number of users connected to the session (the session size) and is somehow difficult and cost ineffective to calculate. Hence, we are interested in researching new cost-effective strategies for finding the limits of a media node for different session sizes that might be more cost-effective and result in less expenditure.

3.3. OpenVidu load testing

Little attention has been paid in the WebRTC arena on easing the task of load testing WebRTC applications, with a few exceptions (Amirante et al., 2016; André et al., 2018). Most load testing approaches are still based on user impersonation through browsers, thus emulating a video conference session by starting several web browsers that are connected into a single session through a media server. This approach has several advantages: it is possible to record the video conference session in each browser and apply state-of-the-art Quality of Experience (QoE) algorithms to automatically determine quality, or even record the browser window to gain better insight on how the application is performing from the users' perspective. However, it also has severe drawbacks: a browser is a big piece of software consuming a considerable amount of resources. Specifically, browsers tend

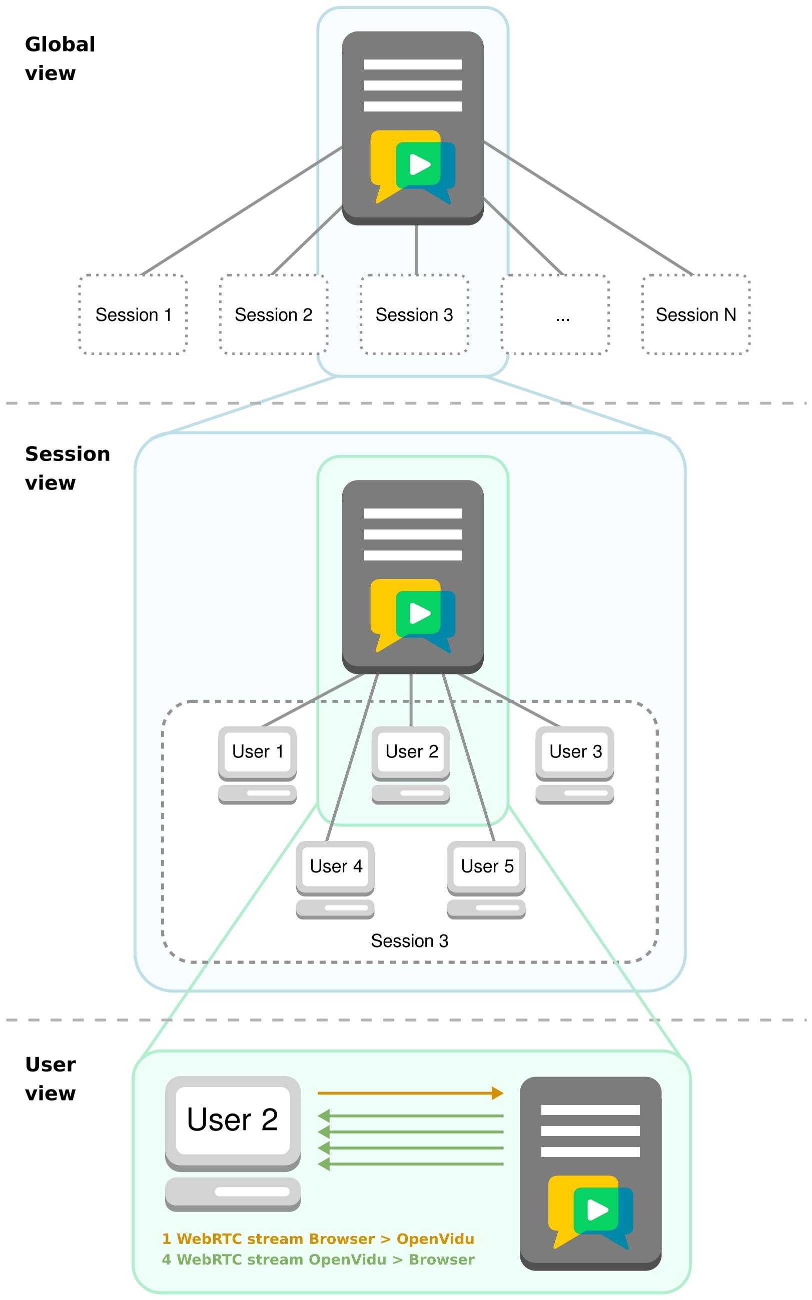

Fig. 2. OpenVidu testing scenario for sessions with 5 users. Notice how each user sends a stream to the media node, and receives 4 streams from it (one per any other user in the session). The session is completely hosted in a single media node. The media node can host several of these sessions.

to consume a lot of CPU and memory just to show a video on-screen. Considering that an important number of browsers will be needed in order to load a media server, this is a huge use of resources. When these load testing strategies are to be considered in the cloud within a continuous integration pipeline, the cost of running so many browsers is also something to take into account.

Despite the drawbacks, this is the most common way of testing WebRTC applications, and the one in use in OpenVidu. There is a specific load test performed frequently at the OpenVidu project: estimating the number of users supported by a single media node, which is a recurrent question in order to plan in advance the number of media nodes needed for a given load. To perform such a load test, the master node is deployed on a machine in the cloud, and a single media node is started and attached to cluster.

Then, a Test Orchestrator is responsible for starting browsers and connecting them to the media node until this media node is not able to handle more connections. Usually this condition is detected by the Test Orchestrator as a browser request to join a session that does not receive any response. At this moment, the test finishes, and the Test Orchestrator reports the number of sessions that were properly connected to the media server, i.e., those sessions for which all users successfully joined their respective sessions.

Usually, this load test is performed on AWS, where machines running browsers can be requested on demand. As shown in Fig. 2, the OpenVidu load test starts EC2 instances (virtual machines) on demand in an AWS environment. Each instance hosts a single Chrome browser managed through Selenium. This strategy is very resource consuming, as many instances might be needed before it can be determined that the media server is not able

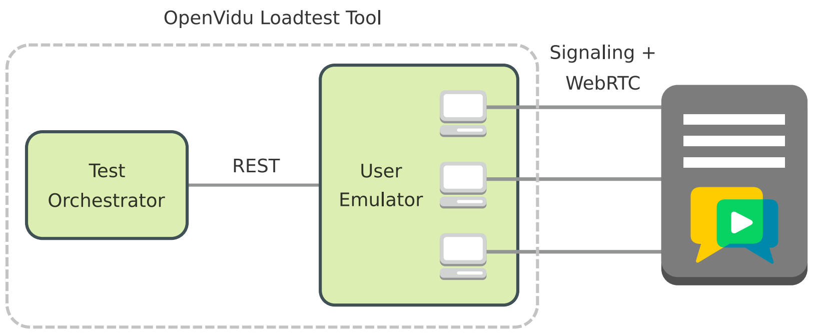

Fig. 3. The emulator service presents a simple API to the Test Orchestrator and hides the specificities of the strategy being used.

to handle more connections. Even for media servers running on small instances, dozens of instances hosting browsers might be needed.

4. Load testing strategies for WebRTC applications

In this paper we propose new load testing techniques for WebRTC applications, and adaptations of existing ones, that optimize the use of resources, hence reducing costs for testing video conference applications. All the strategies depicted here were implemented and are available in the OpenVidu Loadtest Tool.[7] This tool is another contribution of the paper, and enables load testing of WebRTC applications with ease.

The OpenVidu Loadtest Tool consists of a Test Orchestrator, that is responsible for starting all the user emulators according to the test scenario defined in its configuration file, and a browser emulator that is responsible for joining users into WebRTC sessions using the configured strategy. All the configuration is done through files that the tool reads at start up.

It is important to note that the only piece in the OpenVidu Loadtest Tool that depends on the media server is the openvidu-library. This library assumes a control plane protocol that the media server must provide so that the openvidu-library can make a user join a specific session. Given that it requires some experience working with the media server internals for someone to implement such a protocol, for this paper we chose to rely exclusively on the Kurento media server used by OpenVidu, with which the authors have an extensive experience. However, it is possible to use the OpenVidu Loadtest Tool with other media servers as well, although relevant experience on the target media server is needed.

We made an effort within the OpenVidu Loadtest Tool to make strategies interchangeable, by sharing some pieces. Specifically, all the strategies run a user emulator service that exposes a REST API to the Test Orchestrator that can be used to join users to sessions (see Fig. 3). The specificities of how users are joined depend on the different strategies as described below.

The test scenario is configured via a file that the Test Orchestrator reads at start up, and this file configures the session sizes and number of sessions that we want to run in our tests. Several session sizes can be provided and the Test Orchestrator will run a different test case for each session size. The tool can run the strategies locally (for instance, starting browsers in the same machine), or it can use AWS EC2 virtual machines, which allows for running bigger scenarios. Additionally, the OpenVidu Load Test Tool is able to send metrics to an ElasticSearch server, just by providing the corresponding URL of the ElasticSearch service in the configuration file.

4.1. Selenium driven browsers

The first strategy considered is the one described above of using web browsers to impersonate users in a video call. However, we can devise three slightly different ways in which this strategy can be applied that might result in different scalability, use of resources and costs: *browser*, *browser with recording* and *headless browser*.

In the most usual *browser* strategy we considered impersonating users using Chrome web browsers controlled through Selenium running in Docker containers as shown in Fig. 4. When using this strategy, a new Chrome browser is started for each user joining the session. Browsers run a small test app, as shown in Fig. 4 that uses the openvidu-browser library to join a specific session and sends a specific video and audio, instead of using the webcam and microphone, in order to have more control on the video being shared.

There is a second browser-based strategy, namely *browser with recording*, similar to the previous one except that the open source ffmpeg[8] tool is used to record the Chrome browser window. This approach might require some more resources, as the ffmpeg tool is recording the window, encoding it in a suitable format and saving the video to disk. All while the browser is emitting and receiving video. Although in this paper we do not use the recorded videos in any way, we understand that in many cases these videos are used for acceptance testing. That is why we decided to consider recording the browser window.

Finally, there is still a third strategy involving browsers that consists on running the browsers in headless mode. A *headless browser* is a standard browser without a UI. Everything within the browser works as usual except that the browser does not show the page on a graphical widget. We expect this headless mode to result in a reduction on the use of resources as page contents are not rendered on the screen.

In all these three strategies, each user is emulated through its own Chrome browser. The process is as follows: the Test Orchestrator sends a request to the user emulator service, specifying which session should the user join. The user emulator will then start a Docker container with a Chrome browser running a test application, configured to join the session specified. The test application leverages the openvidu-library to join the session and, once joined, a predefined video and audio available within the Docker container is sent to the media node.

4.2. Cost-effective testing solutions for WebRTC

In order to reduce the costs when load testing WebRTC services, we propose two new strategies that do not need to rely on a web browser. Notice that not using browsers would make

7 https://github.com/OpenVidu/openvidu-loadtest.

8 https://www.ffmpeg.org/.

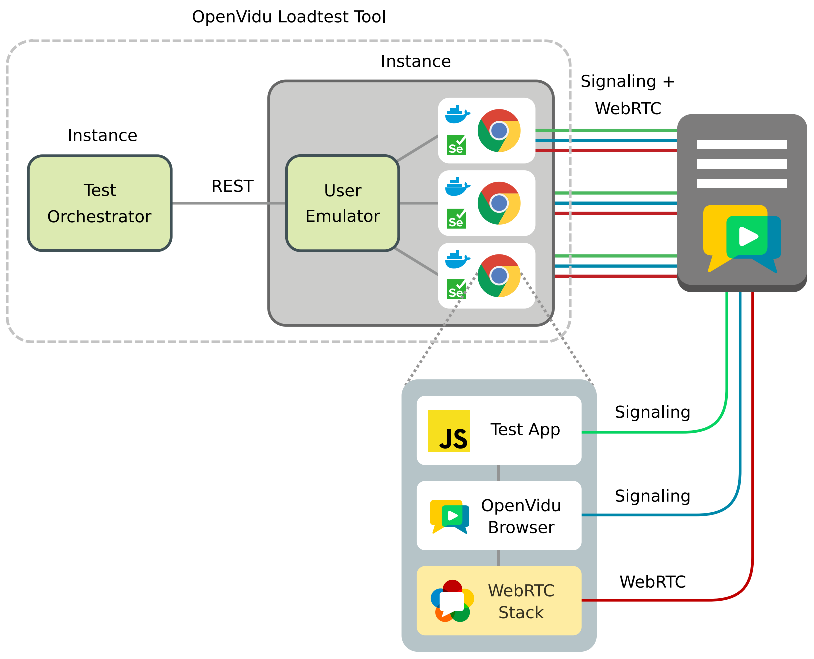

Fig. 4. OpenVidu Loadtest Tool using browsers to impersonate users. Each browser is running in a Docker container and controlled through Selenium.

browser-dependent failures to go unnoticed. However, in this work we are interested in cost-effective solutions for load testing, and we assume that browser-dependent failures would be detected in other test suites.

Traditionally, browsers are used for WebRTC testing because they already implement the WebRTC APIs needed to establish and share media among peers. Therefore, it is relatively easy to build a small test application code to run in the browser that will join a specific WebRTC session. However, if we want to remove the browser from the test scenario in order to save some resources, then we need to consider a replacement that understand the WebRTC API, so that video and audio can be exchanged with the WebRTC service.

In this section we describe two new strategies that do not use browsers, to impersonate users to a WebRTC media server. The first of these strategies, namely *node-webrtc* leverages a Javascript WebRTC API implementation to impersonate users to a WebRTC media server. The second strategy, namely *kms-webrtc*, uses the Kurento WebRTC media server (that implements the WebRTC API) to impersonate users. This strategy is an adaptation of JAttack (Amirante et al., 2016), already described in Section 2.

4.2.1. The node-webrtc approach

Our new proposal, *node-webrtc*, consists on using the WebRTC JavaScript library[9] that implements the WebRTC APIs. This library is able to impersonate many users from a single lightweight process (see Fig. 5). To enable this library to send video and audio, we implemented a video and audio generator that generates and sends the media through the library.

This strategy is one of the main contributions of this work, and it is completely different to the previous ones. No browsers are used in this strategy. Instead, the *node-webrtc* library, that wraps the standard *libwebrtc* library for the Node platform, is able to impersonate users by directly connecting to a media node. The library implements parts of the WebRTC API, enabling the openvidu-browser library to use it as if it were the API of a real browser, thus doing the necessary configuration to send and receive video. With this strategy we get rid of browsers, thus we hypothesize that using this library should result in important savings in CPU and memory. Notice that *node-webrtc* library provides the standard WebRTC API found in browsers, so any WebRTC based library could be used instead of openvidu-browser with that strategy. Therefore, the results presented here can be considered valid regardless the media server used, as we are basically interested in reducing the testing costs implied by impersonating users with web browsers.

4.2.2. The kms-webrtc approach

The second of our contributions is the *kms-webrtc* strategy, inspired partially in Garcia et al. (2017b) and JAttack (Amirante et al., 2016). The *kms-webrtc approach* uses the Kurento media server to impersonate users. The Kurento media server implements de WebRTC protocol, and is able to exchange video and audio with other WebRTC peers through this protocol. On top of that we implemented a developed library that implements the WebRTC API, so that a test can use Kurento as if it was a browser, in the same way that *node-webrtc* works (see Fig. 6).

Thus, the main difference of our *kms-webrtc* strategy with previous strategies is that we implemented a library that wraps the specificities of the media server and provides the standard WebRTC API found in browsers, as in the *node-webrtc* strategy. The openvidu-browser library (or any other WebRTC signaling library) can then be used unmodified with *kms-webrtc*. This is a huge difference with the JAttack approach that needs specific test code in order to deal with the WebRTC protocol that media servers understand. Other differences worth mentioning with respect to JAttack are the following: (a) we used an unmodified standard Kurento Media Server, instead of a modified version of Janus media server (i.e., no modification of the media server was needed); (b) video and audio generation is integrated in our proposal, reducing the complexity of the setup; and (c) in order to use JAttack, a component called Controller has to be implemented to adapt the signaling to a specific media server (Janus).

In the proposed OpenVidu Loadtest Tool the Test Orchestrator plays the role of the JAttack Controller. However, in our case the Test Orchestrator already implements typical load testing scenarios that can be selected with a high level configuration. The

[9] https://github.com/node-webrtc/node-webrtc.

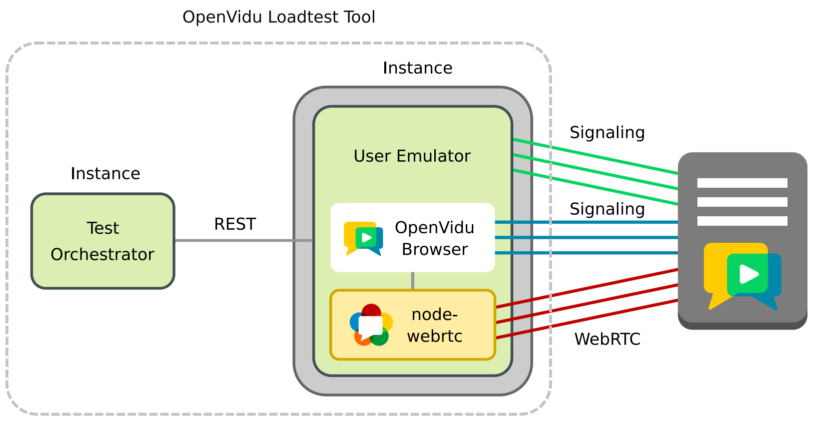

Fig. 5. OpenVidu Loadtest Tool using the *node-webrtc* implementation to impersonate users.

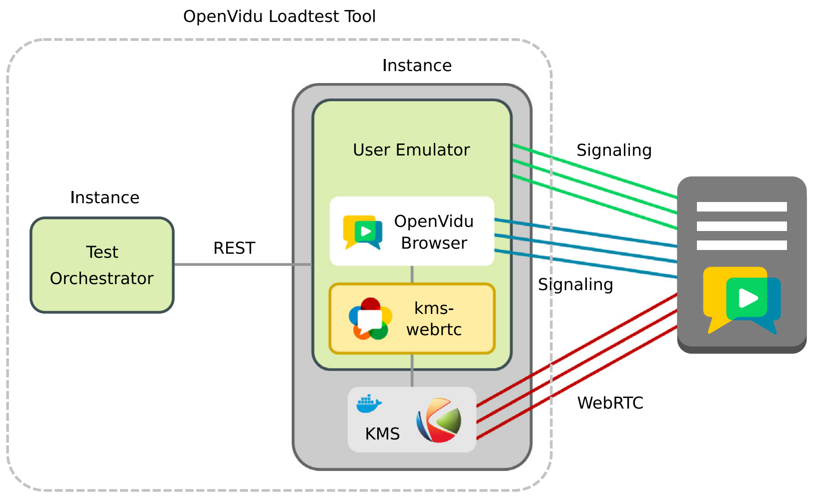

Fig. 6. OpenVidu Loadtest Tool using the *kms-webrtc* strategy to impersonate users.

authors would have considered JAttack in our final experiment comparison, but this tool is closed source, and according to its creators it will not be open-sourced in a near future.[10]

5. Methodology

Our comparison study has two parts: a preliminary experimentation and a final experimentation. Given a number of cloud servers that will be used to load test a WebRTC service, in our preliminary experimentation we determined the limits of each testing strategy, defined as the number of users they can emulate per cloud server. We do it by answering the following research questions:

- RQ_{1a}: What is the strategy that is able to emulate more users into sessions? Is this consistent with the session size?
- RQ_{1b}: How do each of the strategies perform in terms of CPU and memory with respect to the number of users?

In our final experimentation, we studied the cost of the most promising strategy and compared it with a traditional strategy using browsers in a real load test scenario, and used the results to answer RQ_2 (Are there any cost savings for using our proposals with respect to using browsers when doing load testing of a WebRTC platform?). Both experiments were performed in AWS, and consisted on: (1) an OpenVidu deployment (master node and media nodes), and (2) deployment of the necessary infrastructure for each strategy, as discussed in the rest of this section.

5.1. Preliminary experiment

First, we study how each of the proposed testing strategies perform in terms of: (a) number of users that can be impersonated; and (b) memory and cpu usage as we increase the number of users.

Given that the number of media streams exchanged grows up quadratically with the number of users in a session, as described in Section 3, we considered four different scenarios with different session sizes: 2, 3, 5, and 8 users per session. Each scenario has therefore a fixed session size (all sessions have the same size). Then, for each testing strategy, we run four different scenarios with a different session size each. Therefore, by performing the experiment with different session sizes we captured the overall behavior of the testing strategies and by analyzing the results we can answer RQ_{1a}. Notice that we had to run 20 tests in total, as we had to run four scenarios for each of the five testing strategies (*browser*, *browser with recording*, *headless browser*, *node-webrtc* and *kms-webrtc*).

In RQ_{1a} we are interested in measuring the scalability of the different strategies defined as the number of sessions each strategy can emulate in a single instance of a given size, for different session sizes. Therefore, we considered a single AWS instance to impersonate all the users, namely the testing instance. The testing

10 https://groups.google.com/g/meetecho-janus/c/HwdfG82kZ_M/m/FVzq8dp1EAAJ.

instance size chosen was a AWS *t3.xlarge* instance with 4 vCPUs and 16 Gb of memory, big enough to ensure all strategies were able to hold at least one session in the biggest scenario (8 users per session). We used the same video for all the users emulated. The video had a resolution of 640 × 480, at 30 frames per second. All the details can be found in the reproduction package.

The OpenVidu platform (master and media nodes) was deployed on a sufficiently big instance so that the media nodes are never loaded (we are interested in estimating the number of users each testing strategy can handle in the given instance size). Specifically, we used c5.xlarge (4 vCPUs, 8 Gb) instances for both the master node and the media node. We carefully monitored the cpu and memory usage of media nodes to ensure they are never loaded.

All the instances used in the experiment were properly monitored, by running probes in each instance, and metrics were sent to and stored in an Elasticsearch[11] deployed in a different AWS instance. Data stored in this Elasticsearch was used for answering all RQs, and has been exported and made available in our reproduction package.

Each of the 20 tests to run consisted of a testing code that proceeded as shown in Algorithm 1.

Algorithm 1 Preliminary experimentation testing procedure

```
procedure GETUSERSANDSESSIONSCREATED($t$, $sessionSize$)
    $p \leftarrow 0$                                  ▷ Number of users
    $ns \leftarrow 0$                                 ▷ Number of sessions
    $stop \leftarrow false$
    while $!stop$ do
        $sessionId \leftarrow newSession()$
        $i \leftarrow 0$
        for $i \leftarrow 1, sessionSize$ do
            $stop \leftarrow addUser(t, sessionId)$
            if $!stop$ then
                $p \leftarrow p + 1$
            else
                break
            end if
            wait(3)
        end for
        if $!stop$ then
            $ns \leftarrow ns + 1$
        end if
    end while
    return ($p$, $ns$)
end procedure
```

The algorithm introduces load by increasing the number of sessions and for each session, by increasing users until no more users can be added. Specifically, in the Algorithm t is the testing strategy, *sessionSize* is the number of users for each session (2, 3, 5, or 8), p is the total number of users already added (across all sessions) and ns is the number of complete sessions. The variables p and ns are increased during the test execution each time a new user is added and a new session is completed, respectively. A session is considered complete whenever all users have been added to the session (i.e., if the scenario consists on 5 users per sessions, all 5 users have been successfully added). The *newSession* method creates a new session in the OpenVidu platform and returns its id. Then, we resort to add users to this new session until all *sessionSize* users have been added. There is a waiting time of 3 s between one user and the next one in the session to give some time to the session to stabilize after adding the new user. If at any time *addUser* fails to add a new user, the test stops, and reports the number of users added and the number of complete sessions successfully created. Notice that when the test stops there is a session yet to be completed. The maximum number of sessions completed ns for the strategy t will be used in the second part of our study.

Notice that after we successfully complete a session we wait for a reasonable time. This is to give some time to a new connection to stabilize. If we immediately request a new user to be added, the request might fail even when there are still available resources for the new user. This is due to the instance still working on adding the new user and the new streams to the other users in the same session. Therefore, to avoid stopping the test before the machine has been exhausted, we wait for some time, then we resume with the next session.

As was mentioned above, the *addUser* method works differently depending on the strategy t to use. For browser-based strategies, it uses Selenium to start a new browser for the new user within the AWS instance. This method returns a value indicating the stop condition when: (a) Selenium is not able to start the browser; or (b) the browser is not able to join the session (the request times out). For *node-webrtc*, the *addUser* method uses the *node-webrtc* API to instruct the library to add a new user. Similarly, for the *kms-webrtc* strategy, the *addUser* method uses the KMS API to instruct KMS to connect to a session. The error conditions for these two strategies are the same as for browser-based strategies.

In order to answer RQ_{1b}, during the experiment we collected CPU time and memory consumption metrics, using a metricbeat agent[12] that was run in the testing instance.

5.2. Final experiment

In our final experiment we performed a load test against the OpenVidu platform using two different testing strategies: the most common strategy (*browser*) and the most scalable one according to our preliminary experiment (*kms-webrtc*). In this case we do not want to exhaust the testing instances, but to actually load the media server, with the aim of studying the differences in costs of both strategies (*browser* and *kms-webrtc*).

For this experiment we considered a single scenario with $sessionSize = 5$. Given we do not want to overload testing instances, we did not want to use the theoretical maximum number of sessions determined by the preliminary experiment. Indeed, we wanted to ensure that all the sessions would have good quality. Therefore, before resorting to our final experiment, we run both methods with a session size of 5 participants, but for each session, one of the participants was started on a different instance as a Chrome browser, and its window was recorded. We used the videos for both methods to assess at which session the quality decayed. Three of the authors independently watched the videos of each session and method, and gave a score from 1 (worst) to 5 (best) in a Likert scale. All the authors agreed that the quality of the videos recorded of the *browser* strategy already degraded when the second session started, whereas the quality of the videos recorded of the *kms-webrtc* strategy degraded at session 12. Therefore, we could conclude that the maximum practical values in number of sessions for both methods, *browser* and *kms-webrtc*, are 1 and 11, respectively. Notice that we used these values for the final experiment.

The OpenVidu deployment consisted on a master node ($c5.2xlarge$ instance with 8 vCPUs and 16 GB of memory) and two

[11] https://www.elastic.co/elasticsearch/.

[12] https://www.elastic.co/beats/metricbeat.

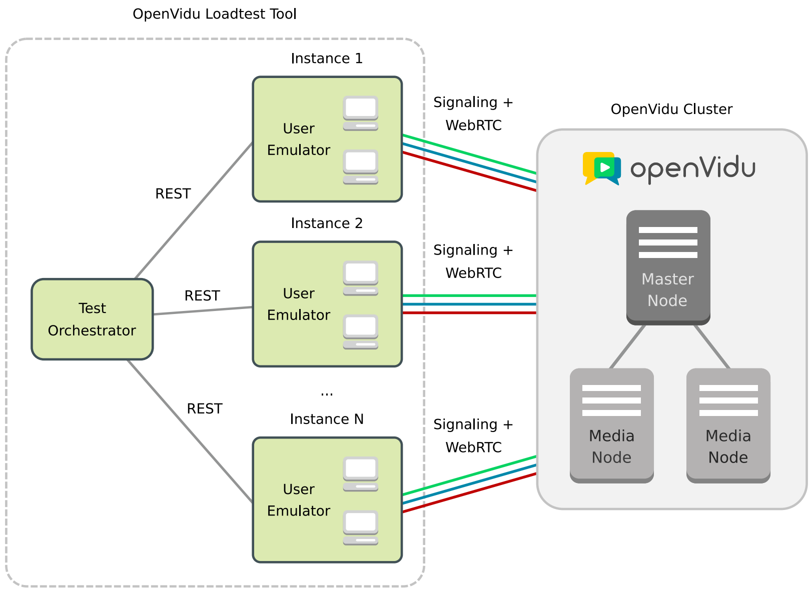

Fig. 7. OpenVidu Loadtest Tool with several instances to perform load testing against an OpenVidu cluster with two media nodes.

additional *c*5.*xlarge* instances for the media nodes (4 vCPUs and 8 GB of memory).

The load test in this case added sessions until no more sessions could be started, i.e., until the platform became unresponsive. Usually, the number of sessions needed to make the system unresponsive are much more than the number of sessions that a single testing instance can hold. Therefore, for this experiment, several testing instances were used. In our final experiment, we run the two strategies mentioned above on a representative scenario, and recorded the total number of sessions and users added, the running time of the test, and the number of instances used in total to impersonate those users. The topology of the final testing experiment is shown in Fig. 7.

In this experiment, the test uses several instances, and it will introduce load by adding sessions to each of them in turn until no more sessions can be added. The process is depicted in Algorithm 2, where *t* is the testing strategy, *sessionSize* is the number of users per session for this test execution, *maxSessions* is the maximum number of complete sessions to be started within each testing instance, and *IPs* is a list of IPs of available instances to be used to impersonate users. Notice that we started a sufficiently large number of instances in advance, and that the test does not start instances on demand (to save some time that otherwise would require starting each instance before being available). The algorithm then proceeds to initialize the number of users joined, sessions started and instances used to zero. Then, while users can be added it proceeds by choosing the first IP (line 8), and starting *maxSessions* within that instance (lines 9–25), one session at a time (lines 10–24), and waiting until the session stabilizes (line 18), just like in our preliminary experiment. When all the *maxSessions* have been started, we resort to the next instance (line 8). During the process we update the number of users (line 14), sessions completed (line 21) and instances used (line 7). The method *addUser* is exactly the same as in Algorithm 1 and will return a code indicating a stopping condition under the same assumptions. The test returns the number of users that were successfully added, the number of complete sessions and the number of instances used. Externally, we also recorded the CPU wall clock time of the test execution.

With the outcome of this experiment we could calculate the cost of each testing scenario with the two testing strategies selected. As the cost of the master and media nodes is fixed,

Algorithm 2 Final experimentation testing procedure

```
1:  procedure OPENVIDULOADTEST($t$, $sessionSize$. $maxSessions$, $IPs$)
2:      $p \leftarrow 0$                                  ▷ Number of users
3:      $ns \leftarrow 0$                                 ▷ Number of sessions
4:      $ni \leftarrow 0$                                 ▷ Number of instances used
5:      $stop \leftarrow false$
6:      while !$stop$ do
7:          $ni \leftarrow ni + 1$
8:          $IP \leftarrow IPs[ni]$
9:          for $j \leftarrow 1, maxSessions$ do
10:             $sessionId \leftarrow newSession(IP)$
11:             for $k \leftarrow 1, sessionSize$ do
12:                 $stop \leftarrow addUser(IP, t, sessionId)$
13:                 if !$stop$ then
14:                     $p \leftarrow p + 1$
15:                 else
16:                     break
17:                 end if
18:                 wait(3)
19:             end for
20:             if !$stop$ then
21:                 $ns \leftarrow ns + 1$
22:             else
23:                 break
24:             end if
25:         end for
26:     end while
27:     return ($p$, $ns$, $ni$)
28: end procedure
```

we considered exclusively the cost of the instances hosting the user emulators. We considered costs of the eu-west-1 AWS region where the experiments were run. All testing strategies made use of the same instance size (t3.xlarge), which had a cost of 0.1824 $ per hour, 0.00304 $ per minute at the time of the experiment. Let m_t be the number of test instances needed for strategy t until the test stops, *time* be the time that it takes to complete the test (in minutes) and X be the cost of the instance (in $/min), then the cost of strategy t can be calculated as $cost_t = time * X * m_t$.

Table 1
Results for the preliminary experiment.

Session size	Testing strategies	Max complete sessions	Max users
2	browser	6	12 (12)
	browser w/recording	5	10 (11)
	headless browser	7	14 (14)
	node-webrtc	18	36 (37)
	kms-webrtc	64	128 (129)
3	browser	4	12 (12)
	browser w/recording	3	9 (10)
	headless browser	4	12 (14)
	node-webrtc	13	39 (40)
	kms-webrtc	26	78 (78)
5	browser	2	10 (12)
	browser w/recording	2	10 (10)
	headless browser	2	10 (11)
	node-webrtc	5	25 (26)
	kms-webrtc	12	60 (62)
8	browser	1	8 (13)
	browser w/recording	1	8 (10)
	headless browser	1	8 (11)
	node-webrtc	1	8 (15)
	kms-webrtc	2	16 (21)

6. Experiment results

All the experiments were performed on AWS, using instances of different sizes as described in Section 5. We performed two different experiments. In our preliminary experimentation we tried to find the limits of each of the load testing strategies on a single AWS instance. In our final experimentation we tried to find the limits of the OpenVidu platform, using two of the strategies (the most common one and the best one from the preliminary experimentation) with the aim of compare the spending of each.

We always used the same instance size for the testing strategies (i.e., all strategies have the same number of CPUs and memory available), and the same video (i.e., all users are sending the same video), except the *node-webrtc* strategy that could not send a real video, and a new fake one was specifically generated for it.

6.1. Preliminary experiment

In the preliminary experiment we considered the five strategies described in Section 4, namely *browser*, *browser with recording*, *headless browser*, *node-webrtc* and *kms-webrtc*. Table 1 reports the session size (number of users per session), the testing strategy, the maximum number of completed sessions that each strategy could manage for each session size before starting to error when adding new users, and the maximum number of users that successfully joined their sessions. The number before the brackets denote the number of users considering only those that belong to a complete session, whereas the number in brackets denote the actual number of users, of which some might belong to a session not yet completed. For instance, for session size 2 the strategy *browser with recording* reported 10 users (corresponding to the 5 users in each of the 2 complete sessions), but 11 were able to join. The last one corresponds to a new, incomplete, session that only one user could join before the stop criteria was met.

As shown in the table, the number of complete sessions that a t3.xlarge instance can hold decreases with the session size in all the strategies. In general, using browsers we were able to generate less load than using our proposed strategies *node-webrtc* and *kms-webrtc*. When the session size goes beyond 3 users, that is, in the case for 5 and 8 users per session, the number of sessions supported by any of the browser-based strategies remain the same (2 and 1 respectively), indicating that there is little overhead in using the UI or recording the browser. Therefore, when considering exclusively browser-based strategies, one could choose browsers with recordings, with the benefits of having a recording of each browser window.

The differences between the browser-based strategies and the other two strategies are huge, with *kms-webrtc* being the strategy that is able to generate more load (handle more complete sessions) in a single machine. This is even more clear in Fig. 8 where the number of sessions supported by each strategy can be graphically compared. The *kms-webrtc* strategy systematically outperforms any other strategy no matter the session size. Notice how for a session size of 8, *kms-webrtc* successfully joined 21 users, more than 2 and a half sessions. For small session sizes *kms-webrtc* is able to handle 6 to 10 times more sessions than browser-based strategies. Our *node-webrtc* proposal also outperforms strategies based on browsers (except for session size 8 where they tie), handling 2 to 3 times more sessions than browsers.

To understand the differences between *node-webrtc* and *kms-webrtc* we need to dive on how these strategies handle video. All the strategies need some prerecorded video that clients use in place of a real video from a real cam. When using a browser, the browser can be fed with a prerecorded video. This is a capability built in the browser when doing testing. However, these videos are handled differently in the *node-webrtc* and *kms-webrtc* strategies. In *node-webrtc* each client needs to read the video. The reading is performed by a ffmpeg process, which means the video is read many times and there are many ffmpeg processes. This is due to the way this approach works: the video is read from within the JavaScript code, and this code is invoked once per client we want to emulate. Furthermore, in *node-webrtc* the video is needed in a specific codec, and ffmpeg needs to transcode the video into the format required by *node-webrtc*. However, in *kms-webrtc*, the video is read once by the Kurento media server, and the same video can be used by different clients. Moreover, the video is read in VP8, the codec needed by WebRTC, without a need for transcoding it into a different format. We believe these are the main reasons why *kms-webrtc* outperforms *node-webrtc*.

RQ$_{1a}$: "What is the strategy that is able to emulate more sessions? Is this consistent with the session size?" The two strategies proposed that avoid the use of browsers outperform any browser-based strategy. The strategy with better scalability (the highest number of sessions and users) is *kms-webrtc*. It is able to handle 64 sessions of 2 users, 26 sessions of 3 users, 12 sessions of 5 users and 2 sessions of 8 users, and it is consistent with the session size, being always above any other strategy. When compared to the most usual strategy of using browsers, *kms-webrtc* is able to handle at least twice the sessions of the browser-based strategies. For small session sizes (2 and 3 users), it is able to handle about an order of magnitude more sessions.

In addition to retrieve the number of sessions and users, each AWS instance was running a probe for monitoring CPU and memory usage. In Fig. 9 we reported the usage of resources for each session size and strategy. We present the evolution of both metrics with the number of users.

Results show that *kms-webrtc* is again consistently below any other strategy in terms of CPU and memory usage. When considering session size 8, *node-webrtc* begins with a lower CPU and memory consumption, but as the number of users increase, *kms-webrtc* stays below *node-webrtc*. Browser-based strategies quickly

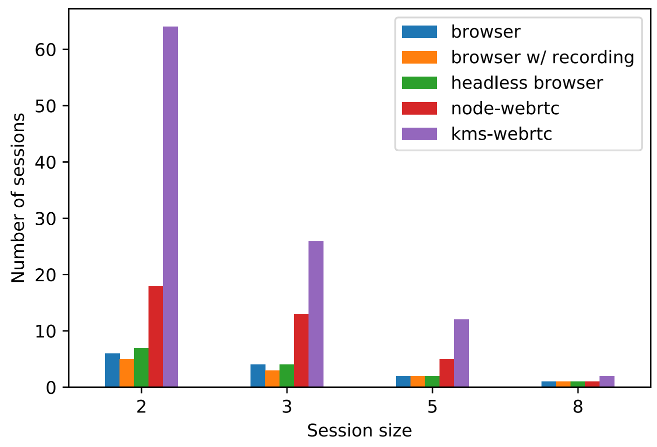

Fig. 8. A comparison of the number of sessions supported by each strategy.

Table 2
Results from our final experiment comparing costs of *browser* and *kms-webrtc* strategies.

Strategy	Max complete sessions	Max users	Workers	Test time	Cost
browser	28	140 (144)	32	60 min	5.8368$
kms-webrtc	33	165 (167)	3	22 min	0.20064$

get to a situation where the CPU is at 100%, and their memory consumption grows quickly, hence the differences in Table 1.

RQ$_{1b}$: "How do each of the strategies perform in terms of CPU and memory with respect to the number of users?" The two proposed strategies *node-webrtc* and *kms-webrtc* perform much better than any browser-based strategy in terms of memory. However, in terms of CPU, *kms-webrtc* performs much better than any of its competitors. When looking at the CPU, *node-webrtc* performs similarly to browser-based strategies. The steeper curves that strategies based on browsers expose in terms of memory are a clear indication that their scalability is poor, which means that not many browsers can be used within a single instance.

6.2. Final experiment

In our final experiment, we study the costs of a load test using the usual browser-based strategy and the *kms-webrtc*, which is the strategy that performed best in our preliminary experiment. As described in Section 5, in this case the Test Orchestrator used several testing instances for emulating clients.

Table 2 shows the results for the final experiment. For each strategy and session size the table reports the number of complete sessions before the OpenVidu platform started to fail, the number of users (the number between brackets is to be interpreted as in Table 1), the number of workers (number of testing instances that were used to host the sessions), the test time, and the cost. The cost is calculated as described in Section 5.2. The browser strategy needed 28 complete sessions to load the OpenVidu platform. To host all those sessions 32 workers were required, running for a total time of 60 min. The total cost of the load test with *browser* (rounded to the nearest hundredth) was 5.84$. When using the *kms-webrtc* strategy, 33 complete sessions were needed to load the OpenVidu platform. These 33 sessions were hosted in 3 instances, with a total expenditure of 0.20$. This means a 3.4% of the spending of the test using browsers.

In order to ensure a fair comparison, we recorded the CPU and memory usage of the master and media nodes. Graphs in Fig. 10 report these metrics, where data of the media nodes is averaged over the two nodes (hence a single graph for both media nodes). The CPU on the master node is below 60%, something expected as the master node is not handling the media, just forwarding users to one of the media nodes. However, the average CPU of the media nodes increases with the number of users. The behavior is similar in both strategies. The memory consumption is about 40% in the master node, without many variations. In the media nodes, it remains slightly below 40% and slowly increases with the number of users. As shown in the graphs, media nodes are CPU bound, that is, the number of users managed by a media node have a bigger impact on CPU than in memory.

RQ$_2$: "Are there any cost savings for using our proposals with respect to using browsers when doing load testing of a WebRTC platform?" Definitely, there is a huge cost saving by using *kms-webrtc*. Expenditures can be cut off by 96.6%, due to using much fewer instances: 3 instances were needed by our proposal, 32 instances were needed by the usual strategy of using browsers.

6.3. Threats to validity

Our studies are subject to construct validity issues, mostly due to the load testing strategies chosen. The most common testing strategies use a browser to impersonate users, sometimes recording the browser window. In the state of the art we could not find any reference to the use of headless browsers, nevertheless, we considered this an interesting strategy. Authors believe the strategies described for load testing WebRTC services are the usual ones. The usage of a single media server in our experiments is also a construct validity issue. Different media servers might support different congestion control protocols, which might impact the network bandwidth usage. This could have an impact on the workers used to generate load, as more clients could be impersonated per worker if the quality is dynamically adjusted. This is because the client might require less CPU and memory to decode the frames received, as these would have lower quality. However, using our tool with different media servers would have required an adapter on top of each of them, something that requires knowledge of the target media servers.

Our results are also subject to internal validity issues, due to how browsers adapt to environmental changes, mainly CPU available. In our preliminary experimentation, when the stop criteria is met, the test instance is reporting a CPU at 100%. This might cause the quality of the video to drop, and hence, the load over the media nodes decreases accordingly. To avoid this issue, we estimated the QoE by recording videos and did a manual assessment of their quality. Another internal validity threat is the stopping criteria itself in the preliminary experiment. In this experiment, we stop as soon as it is reported that one of the clients cannot join its session. This usually happens because the CPU is at 100%. In some cases, this might be just a peak, and the worker might reduce its load after a few seconds. Therefore, an additional attempt to recreate this client might succeed, and the total number of clients supported by the worker might be different. To limit the impact of this issue, we wait 3 s since one user joins its session before the next one is attempted. Furthermore, given the huge difference in terms of clients supported between the two approaches selected for our final experimentation, with 10 clients supported by the *browser* approach versus 60 supported by the *kms-webrtc* approach, even if the issue materializes it is unlikely that it would invalidate the results.

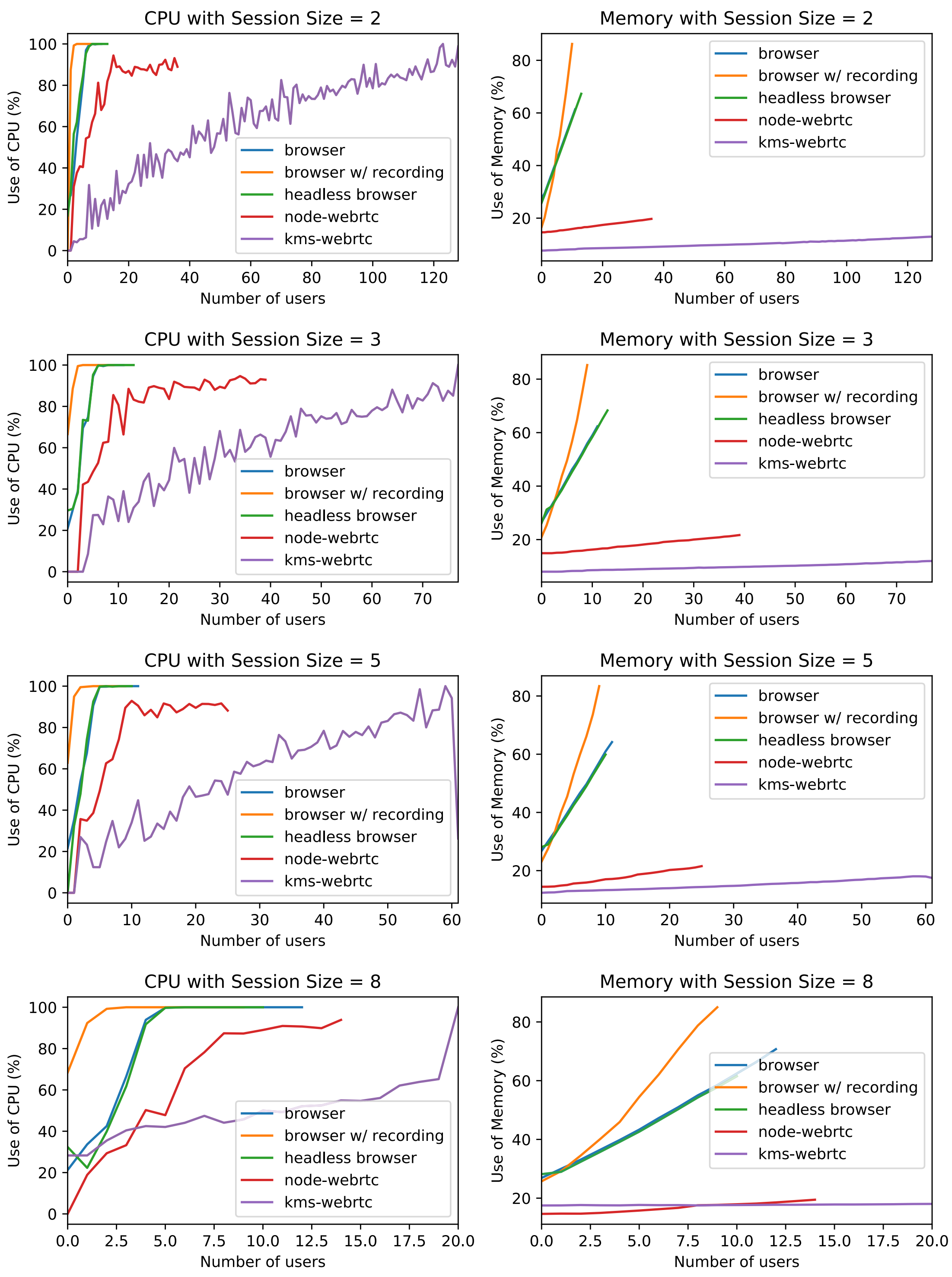

Fig. 9. A comparison on the use of resources (CPU and memory) by session size for each of the strategies.

Finally, we also have external validity issues. We run our experiments in AWS instances (virtual machines) that possibly share the host machine with other instances (not owned by us). When the host is under a huge load, the situation might somehow impact the metrics we took during the test executions. The metrics we took, however, are consistent with what we expected, and we do not think that this issue materialized.

7. Conclusions

In this paper we have proposed cost-effective testing strategies for load testing WebRTC applications, we have developed and open sourced a testing framework for load testing WebRTC applications with any of the 5 testing strategies studied in this work, and we have studied the scalability and cost of these

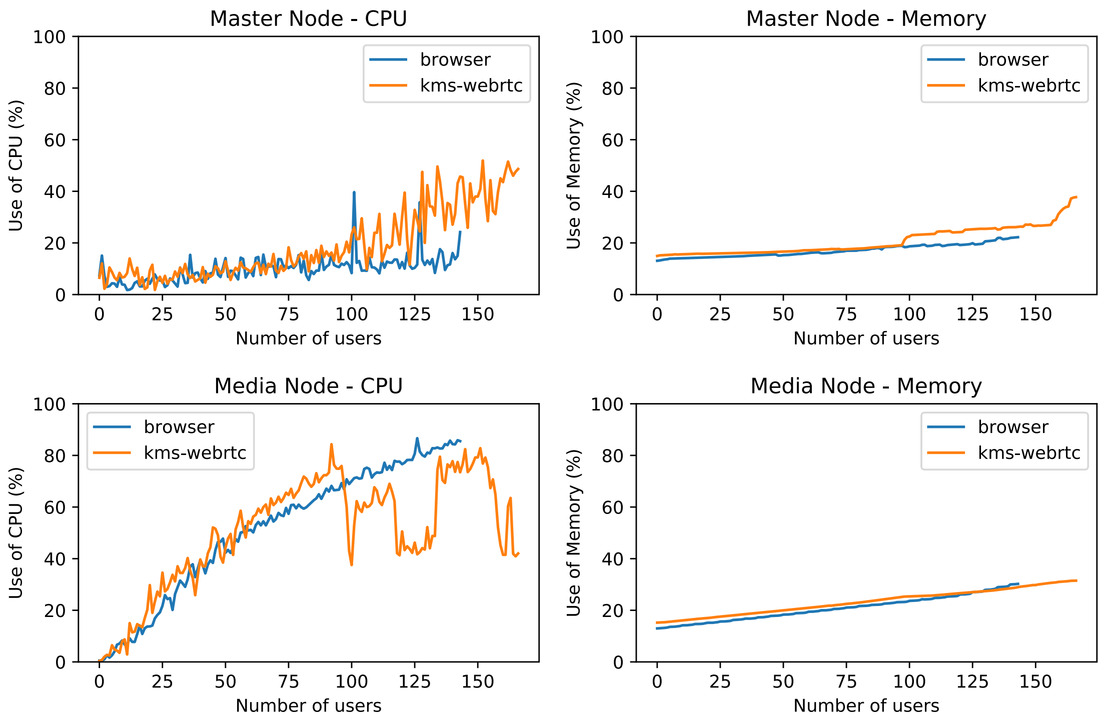

Fig. 10. CPU and memory usage of the master node and the two media nodes (averaged).

strategies, with a strong focus on comparing the new strategies proposed with the usual browser-based strategies in the field.

The main contributions of this paper are:

- A discussion of the most common browser-based strategies for load testing videoconference applications based on the WebRTC standard, and under which circumstances one strategy should be chosen over another. Our data confirms there is a big difference in terms of costs between browser-based and other strategies, with the latter being more cost-effective without impacting quality.
- A proposal of a new strategy, *node-webrtc*, not based on browsers, that is more efficient in terms of use of resources, number of users emulated, than the browser-based ones.
- An improvement over a previous proposal (Amirante et al., 2016) based on the usage of a media server to emulate users. Our strategy, *kms-webrtc*, includes improvements like adapting our strategy to the WebRTC APIs, or removing the need to implement ad-hoc testing code, among others. This strategy outperforms any other strategy in terms of number of users emulated. It also outperforms the *node-webrtc* strategy.
- An open source load testing tool (the OpenVidu Loadtest Tool) that automates the process of generating load through any of these five strategies to an OpenVidu deployment. This tool automatically starts and stops nodes in AWS to inject load into the application under test.
- A study of the scalability of each strategy in terms of number of users, as well as CPU and memory usage. These results suggest that there is little overhead between browsers with and without recording, a conclusion that could be used to favor the former and using the recorded videos as oracles in load testing WebRTC services.
- Our *kms-webrtc* proposal can save up to 96% of monetary costs when performing load testing of WebRTC services. According to these results, developers could use cost-effective strategies such as *kms-webrtc* in their daily CI jobs to ensure there are no regressions with respect to the load the application can handle, while expensive strategies such as browser-based ones could be used from time to time in order to spot UI bugs.

We tried to perform the studies in similar conditions for all the strategies. However, it might be interesting to research, for instance, how different methods behave under different conditions (like network issues).

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

A repository with the OpenVidu Loadtest Tool repository is available in GitHub.[13] Additionally, a reproduction package (Bellas et al., 2021) is available in Zenodo with raw results and a copy of the OpenVidu Loadtest Tool repository.

The authors would like to thank the reviewers for their insightful comments that helped improve the paper. This work has been supported by the Government of Spain through project "BugBirth" (RTI2018-101963-B-100), by the Regional Government of Madrid (Spain) (CM) through project EDGEDATA-CM (P2018/TCS-4499) cofunded by FSE & FEDER, and by the European Commission under the H2020 project "MICADO" (GA-822717).

References

Amirante, A., Castaldi, T., Miniero, L., Romano, S.P., 2014. Janus: a general purpose WebRTC gateway. In: Proceedings of the Conference on Principles, Systems and Applications of IP Telecommunications. pp. 1–8.

13 https://github.com/OpenVidu/openvidu-loadtest.

Amirante, A., Castaldi, T., Miniero, L., Romano, S., 2016. Jattack: a WebRTC load testing tool. In: 2016 Principles, Systems and Applications of IP Telecommunications (IPTComm). IEEE, pp. 1–6.

André, E., Le Breton, N., Lemesle, A., Roux, L., Gouaillard, A., 2018. Comparative study of WebRTC open source SFUs for video conferencing. In: 2018 Principles, Systems and Applications of IP Telecommunications (IPTComm). IEEE, pp. 1–8.

Bellas, F.G., Carrillo, M.G., Bermejo, M.M., Chicano, I., Santos, C., 2021. Dataset of paper "Cost-effective load testing of WebRTC applications". http://dx.doi.org/10.5281/zenodo.5553261.

Bertolino, A., 2007. Software testing research: Achievements, challenges, dreams. In: Future of Software Engineering (FOSE'07). IEEE, pp. 85–103.

Bertolino, A., Angelis, G.D., Gallego, M., García, B., Gortázar, F., Lonetti, F., Marchetti, E., 2019. A systematic review on cloud testing. ACM Comput. Surv. 52 (5), http://dx.doi.org/10.1145/3331447.

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

2022. Bigbluebutton. https://bigbluebutton.org/ (accessed August 16, 2022).

Di Lucca, G.A., Fasolino, A.R., 2006. Testing Web-based applications: The state of the art and future trends. Inf. Softw. Technol. 48 (12), 1172–1186.

Gao, J., Bai, X., Tsai, W.-T., 2011. Cloud testing-issues, challenges, needs and practice. Softw. Eng.: Int. J. 1 (1), 9–23.

García, B., Gallego, M., Gortázar, F., Jiménez, E., 2017a. WebRTC testing: State of the art. In: ICSOFT. pp. 363–371.

Garcia, B., Gortazar, F., Lopez-Fernandez, L., Gallego, M., Paris, M., 2017b. WebRTC testing: challenges and practical solutions. IEEE Commun. Stand. Mag. 1 (2), 36–42.

García, B., López-Fernández, L., Gallego, M., Gortázar, F., 2016. Testing framework for WebRTC services. In: Proceedings of the 9th EAI International Conference on Mobile Multimedia Communications. pp. 40–47.

Gouaillard, A., Roux, L., 2017. Real-time communication testing evolution with WebRTC 1.0. In: 2017 Principles, Systems and Applications of IP Telecommunications (IPTComm). IEEE, pp. 1–8.

2022. Jitsi. https://meet.jit.si (accessed August 16, 2022).

2022. Kurento media server. http://kurento.org (accessed August 16, 2022).

Li, Y.-F., Das, P.K., Dowe, D.L., 2014. Two decades of web application testing—A survey of recent advances. Inf. Syst. 43, 20–54.

2022. Openvidu. https://openvidu.io (accessed August 16, 2022).

Shariff, S.M., Li, H., Bezemer, C.-P., Hassan, A.E., Nguyen, T.H., Flora, P., 2019. Improving the testing efficiency of selenium-based load tests. In: 2019 IEEE/ACM 14th International Workshop on Automation of Software Test (AST). IEEE, pp. 14–20.

Taheri, S., Beni, L.A., Veidenbaum, A.V., Nicolau, A., Cammarota, R., Qiu, J., Lu, Q., Haghighat, M.R., 2015. WebRTCbench: a benchmark for performance assessment of webrtc implementations. In: 2015 13th IEEE Symposium on Embedded Systems for Real-Time Multimedia (ESTIMedia). pp. 1–7. http://dx.doi.org/10.1109/ESTIMedia.2015.7351769.

2022. testRTC. https://testrtc.com/ (accessed August 16, 2022).

2022. Webrtc. https://webrtc.org/ (accessed August 16, 2022).

2022. Whereby. https://whereby.com/ (accessed August 16, 2022).