# Comparative Study of WebRTC Open Source SFUs for Video Conferencing

Emmanuel André*, Nicolas Le Breton*§, Augustin Lemesle*§, Ludovic Roux* and Alexandre Gouaillard*
*CoSMo Software, Singapore, Email: {emmanuel.andre, ludovic.roux, alex.gouaillard}@cosmosoftware.io
§CentraleSupélec, France, Email: {nicolas.lebreton, augustin.lemesle}@supelec.fr

***Abstract*—WebRTC capable media servers are ubiquitous, and among them, Selective Forwarding Units (SFU) seem to generate more and more interest, especially as a mandatory component of WebRTC 1.0 Simulcast. The two most represented use cases implemented using a WebRTC SFU are video conferencing and broadcasting. To date, there has not been any scientific comparative study of WebRTC SFUs. We propose here a new approach based on the KITE testing engine. We apply it to the comparative study of five main open-source WebRTC SFUs, used for video conferencing, under load. The results show that such approach is viable, and provide unexpected and refreshingly new insights on the scalability of those SFUs.**

***Index Terms*—WebRTC, Media Server, Load Testing, Real-Time Communications**

## I. Introduction

Nowadays, most WebRTC applications, systems and services support much more than the original one-to-one peer-to-peer WebRTC use case, and use at least one media server to implement them.

For interoperability with pre-existing technologies like SIP for Voice over IP (VoIP), Public Switched Telephone Network (PSTN), or Flash (RTMP), which can only handle one stream of a given type (audio, video) at a time, one requires media mixing capacities and will choose a Multipoint Control Unit (MCU) [1]. However, most of the more recent media servers are designed as Selective Forwarding Units (SFU) [2]; a design that is not only less CPU intensive on the server, but that also allows for advanced bandwidth adaptation with multiple encoding (simulcast) and Scalable Video Coding (SVC) codecs. The latter of these allow for even better resilience against network quality problems like packet loss.

Even when solely focusing on use cases that are implementable with an SFU, there are still many other remaining. Arguably, the two most popular use cases are video conference (many-to-many, all equally receiving and sending), and streaming / broadcasting (one-to-many, with one sending and many receiving).

While most of the open-source (and closed-source) WebRTC media servers have implemented testing tools (see section II-A), most of those tools are specific to the server they test, and cannot be reused to test others or to make a comparative study. Moreover, the published benchmarks differ so much in terms of methodology that direct comparison is impossible. So far, there has not been any single comparative study of media servers, even from frameworks that claim to be media server and signalling agnostic.



In this paper, we will focus on scalability testing of a video conference use case using a single WebRTC SFU media server. The novelty here is the capacity to run exactly the same test scenario in the same conditions against several different media servers installed on the same instance type. To compare the performance of each SFU for this use case, we report the measurements of their bit rates and of their latency all along the test.

The rest of the paper is structured as follows: section II provides a quick overview of the state of the art of WebRTC testing. Section III describes, in detail, the configurations, metrics, tools and test logic that were used to generate the results presented in section IV, and analyzed in section V.

## II. Background

Verification and Validation (V&V) is the set of techniques that are used to assess software products and services [3]. For a given software, testing is defined as observing the execution of the software on the specific subset of all possible inputs and settings and provide an evaluation of the output or behavior according to certain metrics.

Testing of web applications is a subset of V&V, for which testing and quality assurance is especially challenging due to the heterogeneity of the applications [4]. In their 2006 paper, Di Lucca and Fasolino [5] categorize the different types of testing depending on their non-functional requirements. According to them, the most important are performance, load, stress, compatibility, accessibility, usability and security.

### A. Specific WebRTC Testing tools

A lot of specific WebRTC testing tools exist; for instance, tools that assume something about the media server (e.g. signalling) or the use cases (e.g. broadcasting) they test.

WebRTCBench [6] is an open-source benchmark suite introduced by University of California, Irvine in 2015. This framework aims at measuring performance of WebRTC peer-to-peer (P2P) connection establishment and data channel / video calls. It does not provide for the testing of media servers.

The Jitsi team have developed Jitsi-Hammer [7], an ad-hoc traffic generator dedicated to testing the performance of their open source Jitsi Videobridge [8].

The creators of Janus gateway [9] have developed an ad-hoc tool to assess their gateway performance in different

configurations. From this initial assessment, they proposed Jattack [10], an automated stressing tool for the analysis of performance and scalability of WebRTC-enabled servers. In their paper, they claim that Jattack is generic and that it can be used to assess the performance of WebRTC gateways other than Janus. However, without the code being freely available, this could not be verified.

Most of the tools coming from the VoIP world assume SIP as the signalling protocol and will not work with other signaling protocols (XMPP, MQTT, JSON/WebSockets).

WebRTC-test [11] is an open source framework for functional and load testing of WebRTC on RestComm; a cloud platform aimed at developing voice, video and text messaging applications.

Finally, Red5 has re-purposed an open source RTMP load test tool called "bees with machine guns" to support WebRTC [12].

### B. Generic WebRTC Testing

In the past few years, several research groups have addressed the specific problem of *generic* WebRTC testing. For instance, having a testing framework or engine that would be agnostic to the operating system, browser, application, network, signaling, or media server used. Specifically, the Kurento open source project (Kurento Testing Framework) and the KITE project have generated quite a few articles on this subject.

*1) Kurento Testing Framework a.k.a. ElasTest:*

In [13], the authors introduce the Kurento Testing Framework (KTF), based on Selenium and WebDriver. They mix load-testing, quality testing, performance testing, and functional testing. They apply it on a streaming/broadcast use case (one-to-many) on a relatively small scale: only the Kurento Media Server (KMS) was used, with one server, one room and 50 viewers.

In [14], the authors add limited network instrumentation to KTF. They provide results on the same configuration as above with only minor modifications (NUBOMEDIA is used to install the KMS). They reach 200 viewers at the cost of using native applications (fake clients that implement only the WebRTC parts responsible of negotiation and transport, not the media processing pipeline). Using fake clients generates different traffic and behavior, introducing a de-facto bias in the results.

In [15], the authors add to KTF (renamed ElasTest) support for testing mobile devices through Appium. It is not clear whether they support mobile browsers and, if they do, which browsers and on which OS, or mobile apps. They now recommend to install KMS through OpenVidu, and propose to extend the WebDriver protocol to add several APIs. While WebDriver protocol implementation modifications are easy on the Selenium side, on the browser-side they would require the browser vendors to modify their (sometimes closed-source) WebDriver implementations, which has never happened in the past.

| | Media Server VM | Client VM |
|---|---|---|
| AWS instance type | c4.4xlarge | c4.xlarge |
| Rationale | Cost effective instance with dedicated computing capacity | Cost effective instance with dedicated computing capacity (4 vCPU) |
| RAM | 30 GB | 7.5 GB |
| Dedicated bandwidth | 2 Gbps | 750 Mbps |

TABLE I: VMs for the media servers and the clients.

*2) Karoshi Interoperability Test Engine: KITE:*

The KITE project, created and managed by companies actively involved in the WebRTC standard, has also been very active in the WebRTC testing field with an original focus on compliance testing for the WebRTC standardization process at W3C [16]. KITE is running around a thousand tests across 21+ browsers / browser revisions / operating systems / OS revisions on a daily basis. The results are reported on the official webrtc.org page.[1]

In the process of WebRTC 1.0 specification compliance testing, simulcast testing required using an SFU to test against (see the specification[2] chapter 5.1 for simulcast and RID, and section 11.4 for the corresponding example). Since there was no reference WebRTC SFU implementation, it has been decided to run the simulcast tests in Chrome browser against the most well-known of the open source SFUs.

*3) Comparison and Choice:*

KTF has been understandingly focused on testing the KMS from the start, and only ever exhibited results in their publications about testing KMS in the one-to-many use case.

KITE has been designed with scalability and flexibility in mind. The work done in the context of WebRTC 1.0 simulcast compliance testing paved the way for generic SFU testing support.

We decided to extend that work to comparatively load test most of the open-source WebRTC SFUs, in the video conference use case, with a single server configuration.

## III. System Under Test and Environment

### A. Cloud and Network Settings

All tests were done using Amazon Web Services (AWS) Elastic Compute Cloud (EC2). Each SFU and each of its connecting web client apps were run on separate Virtual Machines (VMs) in the same AWS Virtual Private Cloud (VPC) to avoid network fluctuations and interference. The instance types for the VMs used are described in Table I.

### B. WebRTC Open Source Media Servers

We set up the following five open-source WebRTC SFUs, using the latest source code downloaded from their respective public GitHub repositories (except for Kurento/OpenVidu, for which the Docker container was used), in a separate AWS EC2 Virtual Machine and with default configuration:

- Jitsi Meet (JVB version 0.1.1077)[3]

[1] https://webrtc.org/testing/kite/
[2] https://www.w3.org/TR/webrtc/
[3] https://github.com/jitsi/jitsi-meet

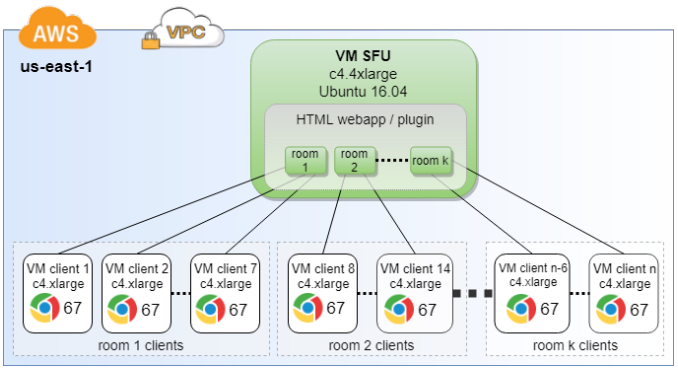



Fig. 1: Configuration of the tests.

- Janus Gateway (version 0.4.3)[4] with plugin/client web app[5]
- Medooze (version 0.32.0)[6] with plugin/client web app[7]
- Kurento/OpenVidu (from Docker container, Kurento Media Server version 6.7.0)[8] with plugin/client web app[9]
- Mediasoup (version 2.2.3)[10]

Note that on Kurento, the min and max bit rates are hard-coded at 600,000 bps in MediaEndpoint.java at line 276. The same limit is 1,700,000 bps for Jitsi, Janus, Medooze and Mediasoup as seen in Table IV column (D).

We did not modify the source code of the SFUs, so these sending limits remained active for Kurento and Jitsi when running the tests.

The five open-souce WebRTC SFUs tested use the same signalling transport protocol (WebSockets) and do not differ enough in their signalling protocol to induce any measurable impact in the chosen metrics. They all implement WebRTC, which means they all proceed through discovery, handshake and media transport establishment exactly the same standard way, respectively using ICE [17], JSEP [18] and DTLS-SRTP [19].

### C. Web Client Applications

To test the five SFUs with the same parameters and to collect useful information (getStats see section III-E and full size screen captures), we made the following modifications to the corresponding web client apps:

- increase the maximum number of participants per meeting room to 40
- support for multiple meeting rooms, including `roomId` and `userId` in the URL
- increase sending bit rate limit to 5,000,000 bps (for Janus only, as it was configurable on the client web app)

[4]https://github.com/meetecho/janus-gateway
[5]https://janus.conf.meetecho.com/videoroomtest.html
[6]https://github.com/medooze/mp4v2.git
[7]https://github.com/medooze/sfu/tree/master/www
[8]https://openvidu.io/docs/deployment/deploying-ubuntu/
[9]https://github.com/OpenVidu/openvidu-tutorials/tree/master/openvidu-js-node
[10]https://www.npmjs.com/package/mediasoup

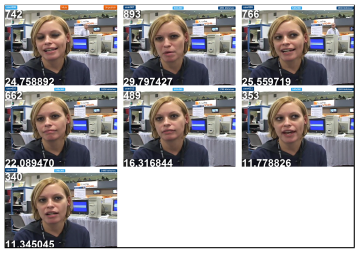


Fig. 2: Screenshot of Janus Video Room Test web app after modifications.

- support for displaying up to 9 videos with the exact same dimensions as the original test video (540×360 pixels)
- removal or re-positioning of all the text and images overlays added by the client web app so that they are displayed above each video area
- expose the JavaScript RTCPeerConnection objects to call getStats() on.

Each client web app is run on a dedicated VM, which has been chosen to ensure there will be more than enough resources to process the 7 videos (1 sent and 6 received).

A Selenium node, instrumenting Chrome 67, is running on each client VM. KITE communicates with each node, through a Selenium hub, using the WebDriver protocol (see Fig. 1).

Fig. 2 shows a screenshot of the modified Janus Video Room web client application with the changes described above. The sender video is displayed at the top left corner of the window. The received videos received from each of the six remote clients are displayed as shown on Fig. 2. The original dimensions of the full image with the 7 videos are 1980×1280 pixels. The user id, dimension and bit rates are displayed above the video leaving it free of obstruction for quality analysis.

### D. Controlled Media for Quality Assessment

As the clients are joining a video conference, they are supposed to send video and audio. To control the media that each client sends and in order to make quality measurements, we use Chrome fake media functionality. The sender and each client play the same video.[11] Chrome is launched with the following options to activate the fake media functionality:

- `allow-file-access-from-files`
- `use-file-for-fake-video-capture=e-dv548_lwe08_christa_casebeer_003.y4m`
- `use-file-for-fake-audio-capture=e-dv548_lwe08_christa_casebeer_003.wav`
- `window-size=1980,1280`

[11]Credits for the video file used: "Internet personality Christa Casebeer, aka Linuxchic on Alternageek.com, Take One 02," by Christian Einfeldt. DigitalTippingPoint.com https://archive.org/details/e-dv548_lwe08_christa_casebeer_003.ogg
The original file, 540×360 pixels, encoded with H.264 Constrained Baseline Profile 30 fps for the video part, has been converted using ffmpeg to YUV4MPEG2 format keeping the same resolution of 540×360 pixels, colour space 4:2:0, 30 fps, progressive. Frame number has been added as an overlay to the top left of each frame, while time in seconds has been added at the bottom left. The original audio part, MPEG-AAC 44100 Hz 128 kpbs, has been converted using ffmpeg to WAV 44100 Hz 1411 kbps.

| | Jitsi | Janus | Medooze | Kurento | Mediasoup |
|---|---|---|---|---|---|
| Number of rooms (7 participants in a room) | 40 | 70 | 70 | 20 | 70 |
| Number of client VMs | 280 | 490 | 490 | 140 | 490 |

TABLE II: Load test parameters per SFU.

The last option (`window-size`) was set to be large enough to accommodate 7 videos on the screen with original dimensions of 540×360 pixels.

### E. Metrics and Probing

*1) Client-side: getStats() function:*

The test calls getStats() to retrieve all of the statistics values provided by Chrome. The bit rates for the sent video, and all received videos, are computed by KITE by calling getStats twice (during ramp-up) or 8 times (at load reached) using the `byteReceived` and `timestamp` value of the first and last getStats objects.

*2) Client-side: Video Verification:*

Once a `<video>` element has been loaded, we verify that it displays a video and that it is neither blank, static nor a frozen image. We execute a JavaScript function that computes the sum of all the pixels in the image. The function is called twice with a 500ms interval. If it returns 0, then the video element is not receiving any stream and the display is empty. If the return values of the two calls for a given video element are positive, but the difference is null, then the video has frozen.

*3) Client-Side: Video Quality Assessment:*

Beyond the network measurements reported above, we have applied NARVAL (Neural network-based Aggregation of no-Reference metrics for Video quAlity evaLuation) video quality assessment tool [20]. NARVAL is a combination of state-of-the-art image and video metrics to evaluate the quality of the videos received from the sender and from each of the 6 participants in the rooms. The metrics used for this evaluation are BRISQUE features (Blind/Referenceless Image Spatial QUality Evaluator) [21], contrast, cumulative probability of blur detection (CPBD) [22], edges detected by Sobel and Canny, blur, image sharpness with Fast Fourier Transform and histogram of oriented gradient (HOG). Values computed for all these metrics are then used as input of a neural network trained specifically at evaluating quality of videos. The output of the network is a score in the range 0 (worst) to 100 (best) giving an objective estimation of the quality of the displayed videos.

Chrome will be launched with a fixed window size of 1980×1280 pixels, and the screenshots are taken using the Selenium WebDriver function which will generate a PNG file of the same dimensions. Those are used for video quality evaluation as described in section IV-D.

### F. Test Methodology

The tests are executed with KITE [16]. They are therefore written in Java and rely on Selenium WebDriver to launch and control the client browsers.

| | SFU | Page loaded | % | Sender video check | % | All videos check | % |
|---|---|---|---|---|---|---|---|
| PASS | Jitsi | 280 | 100% | 280 | 100% | 53 | 19% |
| | Janus | 448 | 100% | 448 | 100% | 256 | 57% |
| | Medooze | 481 | 100% | 481 | 100% | 434 | 90% |
| | Kurento | 118 | 100% | 118 | 100% | 69 | 58% |
| | Mediasoup | 488 | 100% | 488 | 100% | 476 | 97% |
| FAIL | Jitsi | 0 | 0% | 0 | 0% | 227 | 81% |
| | Janus | 2 | 0% | 0 | 0% | 195 | 43% |
| | Medooze | 0 | 0% | 0 | 0% | 47 | 10% |
| | Kurento | 0 | 0% | 0 | 0% | 49 | 42% |
| | Mediasoup | 2 | 0% | 0 | 0% | 15 | 3% |

TABLE III: Test results.
*Page loaded:* true if the page can load. *Sender video check:* true if video of the sender is displayed and is not a still or blank image. *All video check:* true if all the videos received by the six clients from the SFU passed the video check.

We began by executing several dry run tests before deciding on the achievable target loads. We realized that not all SFUs were able to support the same kind of load. As a result, we decided to limit the tests on Kurento to 20 rooms as that SFU usually failed before reaching that load. Similarly, we set a limit of 40 rooms for Jitsi as this SFU systematically failed when reaching 245 users in the test.

For the final load test, based on the results from the dry runs, we set parameters as shown in Table II. All tests were executed twice or more to validate the consistency of the results.

*1) Per SFU Test Logic:*

For each of the five SFUs we execute the following steps:

- KITE instantiates the required number of client VMs.
- For each room of a given SFU:
  - For each of the client VMs in a room (7 in our case):
    - fill the room and validate
    - upon successful validation, measure and probe (ramp-up)
  - Measure and probe (at target load)

*2) Room Filling and Validation Process:*

- Launch Chrome
- Open the client web app as given user in given room
- Wait for web app to load
- Do the video verification on the local video
- Do the video verification on all remote videos

*3) Measure and Probe:*

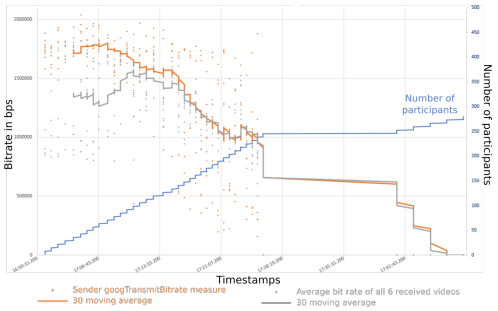
getStats is being called with one second interval n times, n=2 during ramp-up, once the room is filled with 7 users, and n=8 at target load, when all rooms have been created and filled.

- Take the first screenshot
- Call getStats and wait one second, n times.
- Take a second screenshot

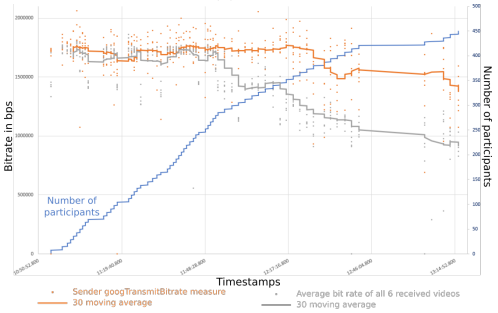
## IV. Results

### A. Quantitative Results

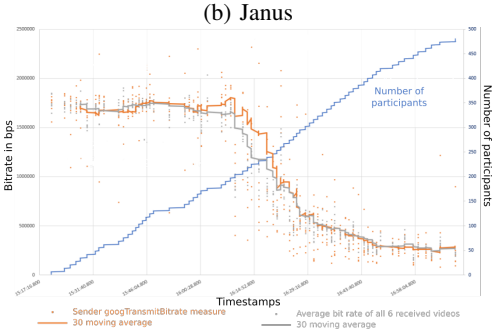
On Table III, we present the rate of SUCCESS and FAILURES that occurred during the tests (a success means that the



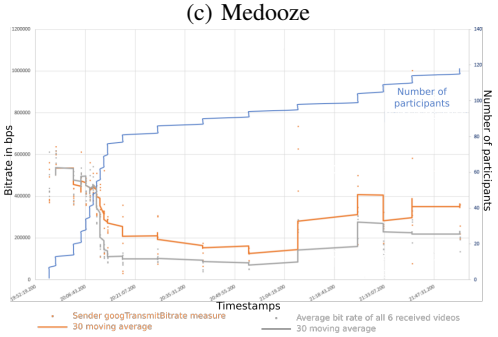

(a) Jitsi



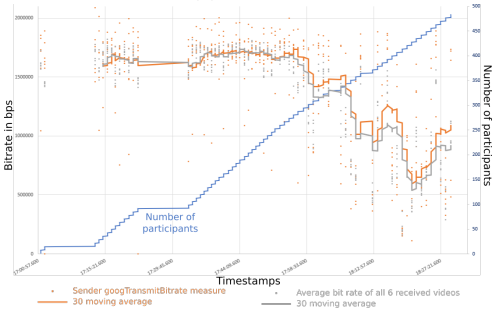

(b) Janus




(c) Medooze




(d) Kurento




(e) Mediasoup

Fig. 3: Transmission bit rates.

corresponding video is displayed and it is not a still or a blank image). No failure was reported concerning the display of the sender video. However, there have been some failures for the videos received from the 6 clients of a room.

The failure rate is very high for Jitsi at 81% because in most cases still images are displayed instead of the video (correspondingly the measured bit rates are zero). There is also a high failure rate of 43% for Janus and 42% for Kurento as many clients are missing one or more videos.

### *B. Bit Rates Measurements*

Bit rates measured for the sender's video and the average of the 6 valid receivers' videos in a room during the whole experiment are presented separately per SFU in Fig. 3, while Fig. 4a gives the graphs of the average reception bit rates of the 6 valid receivers. Zero or null bit rates are not included in the computation of the average, e.g. if the client only received 3 videos out of the 6, the values in Fig. 3 and in Fig. 4a are the average of those 3 videos.

It is clearly visible on Fig. 4a that Kurento has a hard-coded maximum bit rate of 600,000 bps, while the four other SFUs are able to use all the available bandwidth.

Apart from Kurento, all the SFUs have a rather similar behaviour regarding the average bit rate measured at receiver side. At some point, when the number of users to be served keeps increasing, the bit rate starts to decrease. This reduction in receiving bit rate happens first for Medooze at about 195 users, then for both Janus and Mediasoup at about 270 users. Jitsi simply stops sending any video when the number of users reaches 245, so at that point the bit rate drops down to zero.

In Fig. 3 we can see more finely the difference of behaviour of each SFU with the increase of number of rooms and participants. In these charts, the blue graph "number of participants" shows the increase of users with time. Each step means the addition of one room and 7 users to the test.

Ideally, this blue graph should have a very regular linear pattern. In reality, we can see some steps that are very long, which means that for a long time no new room has been added to the test. This is due to the video check that fails. The creation of a new room happens only when video check is successful for the current room (see validation check in subsection III-F2) or after a timeout set at 7 minutes (one minute timeout per video).

Fig. 3a is about Jitsi. The number of rooms and users increases very smoothly up to 34 rooms and 238 users. But when the 35$^{th}$ room is added, Jitsi seems to collapse: bitrate drops down to zero, and the check for validating the videos on the 35$^{th}$ room lasts for a very long time until a timeout.

For Janus, Fig. 3b, the number of rooms and users increases regularly for most of the test. However, from room 39 and subsequently, it takes a little more time to get all the 7 videos of a room to be displayed. It takes a very long time for room 60 only, then Janus is able to manage the additional rooms up to the target load of 70 rooms.

Medooze, Fig. 3c, achieve handling the load increase up to target load. However, at the beginning of the test it has

difficulties from time to time to have all of the 7 participants in a room to be up and active as one or more videos are missing

Kurento transmission bit rates are reported on Fig. 3d. Most of the time, check for videos in the latest created room takes a long time until timeout, except at the beginning of the test for the 10 first rooms.

At last Mediasoup, Fig. 3e, exhibits a very regular pattern for the number of participants graph, except for the second and the thirtieth rooms where a missing video delays the creation of the next room until timeout occurs.

Table IV gives an overview of the sender's video statistics collected on each client during the ramp-up phase of the load test. Looking at the average sending bit rate over time (column (E)), all the SFUs have a high average sending bit rate ranging from 1 Mbps (Medooze) to 1.6 Mbps (Janus). Only Kurento has a much lower bit rate of about 350 kbps. This can be explained in part because of its hard coded bandwidth limitation of 600,000 bps, and also because more than half of the video checks fail (see Table III).

The statistics about the bit rates for the 6 videos received by each client during ramp-up are reported in Table V. Here again, we notice a large difference between Kurento with a bit rate of roughly 300 kbps for the clients, and the other four SFUs for which the receiving bit rate ranges from about 1 Mbps (Medooze) to about 1.6 Mbps (Janus).

When target load is reached (Table VI), there are no data for Jitsi as videos are no more transmitted once the number of participants is above 245. On average, for Janus, sending bit rate is 1.18 Mbps and receiving bit rate is almost 1 Mbps at target load. For Mediasoup, the sending bit rate is 700 kbps and receiving bit rate is about 680 Mbps. For Medooze, both sending and receiving bit rates are much lower at about 215 kbps. At last, Kurento has a sending bit rate of 250 kbps and receiving bit rate of about 120 kbps.

### C. Latency Measurements

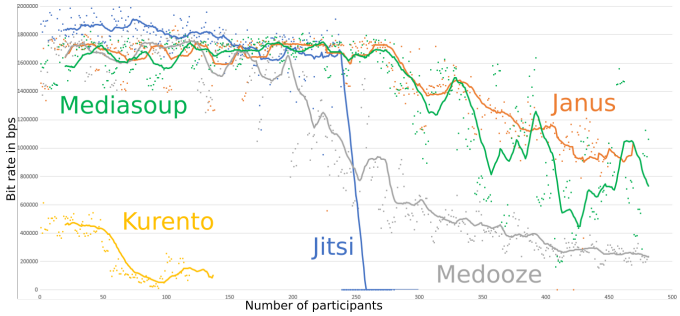
Fig. 4b gives an overview of RTT for the whole duration of the test. Note we had to use a logarithmic scale for this figure. Similarly as for the bit rates, variation of RTT is relatively similar for all the SFUs except Kurento.

Some detailed figures for the sender are only reported for RTT during ramp-up in Table IV column (F) and at target load in Table VI column (A). During ramp-up, Jitsi, Medooze and Mediasoup have a latency below 20 ms., for Janus it is 61 ms., while for Kurento it is already above half a second.

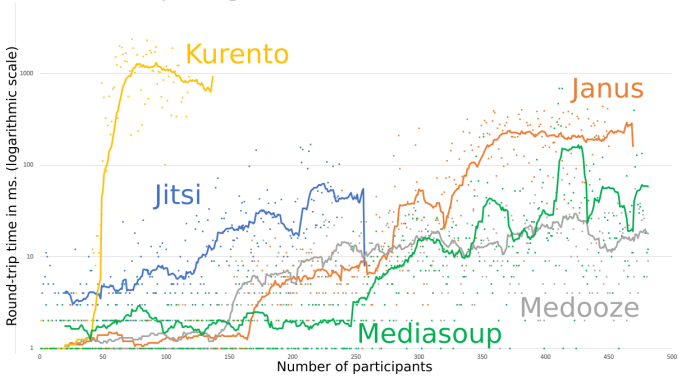
At target load, both Medooze and Mediasoup keep a low latency lower than 50 ms., for Janus it is 268 ms., and for Kurento it is slightly above one second.

### D. Video Quality Assessment

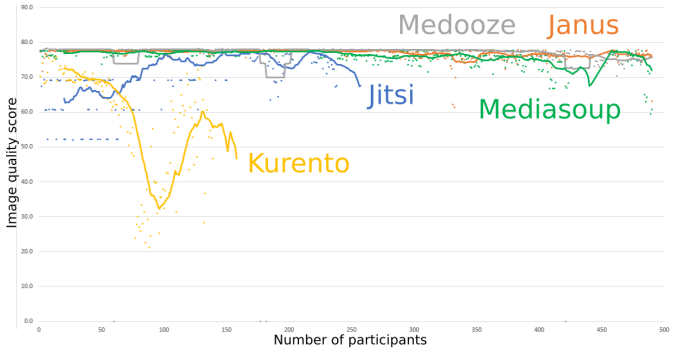
Estimation of video quality scores is presented in Fig. 4c. One may expect video quality to deteriorate as the average bit rate measured falls down, but the graphs of video quality remain remarkably flat until the end of the test. This counter intuitive result is explained by the ability of modern video




(a) Average reception bit rates for the 6 valid receivers.




(b) Average round-trip time for the 6 valid receivers *(logarithmic scale)*.




(c) Average video quality scores.

Fig. 4: Results over the number of participants.

codecs to make a video weight 15 to 30 times less than the original uncompressed video while still keeping a quality that is perceived as very good. For our test, each Chrome browser has encoded the video with VP8 before sending it to the SFU. After several experiments with ffmpeg, we noticed that the quality of the video we used for this load test becomes to be visibly damaged when the bit rate is set at about 150 kbps or lower. At target load, average bit rate for Medooze is about 215 kbps. This is still enough to transfer the selected video with a good perceived quality.

Kurento video quality varies awkwardly according to the load. It deteriorates almost immediately and reaches its lowest image quality at about 100 participants. Surprisingly, the image quality improves as more participants join the test, up

to about 130 participants, before dropping again.

## V. Analysis

This study exhibits interesting behaviours of the five SFUs that have been evaluated when they have to handle an increasing number of rooms and peers in a video conference use case.

Kurento is plagued by both a hard coded maximum bandwidth and a high RTT problem. When the number of participants is above 42, RTT rises sharply to reach and stay at about one second. The bit rate, already very low at the start of the test because of the hard coded limitation, falls quickly above 49 participants to a very low value before improving a little above 100 participants. Interestingly, video quality, which was worsening since the beginning of the test, starts to get better at about 100 participants just when the bit rates improve. But this improvement lasts only until 130 participants have joined the test.

Jitsi has some internal problem that makes it suddenly stop transmitting videos when there are more than 245 peers in the test.

All the three other SFUs tested behave roughly in a similar way. Bit rate is maintained at the same level while the number of peers increases, then at some point bit rate start to decrease. It happens first to Medooze at about 190 peers. Janus and Mediasoup are able to keep the maximum level of bit rate for a higher load as the decrease of bit rate starts above 280 peers. We note also that the decrease of bit rate is sharper for Medooze.

## VI. Conclusion and Future Work

We have shown that it is now possible to comparatively test WebRTC SFUs using KITE. Several bugs and oddities have been found and reported to their respective team in the process.

This work was focused on the testing system, and not on the tests themselves. In the future we would like to add more metrics and tests on client side, for example to assess audio quality as well, and to run the tests on all supported browsers to check how browser specific WebRTC implementations make a difference. On the server side, we would like to add CPU, RAM and bandwidth estimation probes to assess the server scalability on load.

We would like to extend this work to variations of the video conferencing use cases by making the number of rooms and the number of users per rooms a variable of the test run. That would allow to reproduce the results of [7] and [10].

We would like to extend this work to different use cases, for example broadcasting / streaming. That would allow, among other things, to reproduce the results from the Kurento Research Team.

## Acknowledgment

We would like to thanks Boris Grozev (Jitsi), Lorenzo Miniero (Janus), Iñaki Baz Castillo (Mediasoup), Sergio Murillo (Medooze), Lennart Schulte (callstats.io), Lynsey Haynes (slack) and other Media server experts who provided live feedback on early result during CommCon UK 2018. We would also like to thanks Boni García for discussions around the Kurento Team Testing research results.

## References

[1] M. Westerlund and S. Wenger, *RFC 7667: RTP Topologies*, IETF, Nov. 2015. [Online]. Available: https://datatracker.ietf.org/doc/rfc7667/

[2] B. Grozev, L. Marinov, L. Marinov, and E. Ivov, "Last N: relevance-based selectivity for forwarding video in multimedia conferences," in *Proceedings of the 25th ACM Workshop on Network and Operating Systems Support for Digital Audio and Video*, 2015.

[3] A. Bertolino, "Software testing research: Achievements, challenges, dreams," in *Future of Software Engineering*, 2007.

[4] Y.-F. Li, P. K. Das, and D. L. Dowe, "Two decades of web application testing — a survey of recent advances," *Information Systems*, 2014.

[5] G. A. Di Lucca and A. R. Fasolino, "Testing web-based applications: The state of the art and future trends," *Information and Software Technology*, 2006.

[6] S. Taheri, L. A. Beni, A. V. Veidenbaum, A. Nicolau, R. Cammarota, J. Qiu, Q. Lu, and M. R. Haghighat, "WebRTCBench: A benchmark for performance assessment of WebRTC implementations," in *13th IEEE Symposium on Embedded Systems for Real-time Multimedia*, 2015.

[7] Jitsi-Hammer, a traffic generator for Jitsi Videobridge. [Online]. Available: https://github.com/jitsi/jitsi-hammer

[8] B. Grozev and E. Ivov. Jitsi videobridge performance evaluation. [Online]. Available: https://jitsi.org/jitsi-videobridge-performance-evaluation/

[9] A. Amirante, T. Castaldi, L. Miniero, and S. P. Romano, "Performance analysis of the Janus WebRTC gateway," in *EuroSys 2015, Workshop on All-Web Real-Time Systems*, 2015.

[10] ——, "Jattack: a WebRTC load testing tool," in *IPTComm 2016, Principles, Systems and Applications of IP Telecommunications*, 2016.

[11] WebRTC-test, framework for functional and load testing of WebRTC. [Online]. Available: https://github.com/RestComm/WebRTC-test

[12] Red5Pro load testing: WebRTC and more. [Online]. Available: https://blog.red5pro.com/load-testing-with-webrtc-and-more/

[13] B. García, L. Lopez-Fernandez, F. Gortázar, and M. Gallego, "Analysis of video quality and end-to-end latency in WebRTC," in *IEEE Globecom 2016, Workshop on Quality of Experience for Multimedia Communications*, 2016.

[14] B. García, L. Lopez-Fernandez, F. Gortázar, M. Gallego, and M. Paris, "WebRTC testing: Challenges and practical solutions," *IEEE Communications Standards Magazine*, 2017.

[15] B. García, F. Gortázar, M. Gallego, and E. Jiménez, "User impersonation as a service in end-to-end testing," in *MODELSWARD 2018, 6th International Conference on Model-Driven Engineering and Software Development*, 2018.

[16] A. Gouaillard and L. Roux, "Real-time communication testing evolution with WebRTC 1.0," in *IPTComm 2017, Principles, Systems and Applications of IP Telecommunications*, 2017.

[17] A. Keranen, C. Holmberg, and J. Rosenberg, *RFC 8445: Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal*, IETF, July 2018. [Online]. Available: https://datatracker.ietf.org/doc/rfc8445/

[18] J. Uberti, C. Jennings, and E. Rescorla, *JavaScript Session Establishment Protocol [draft IETF]*, IETF, Oct. 2017. [Online]. Available: https://tools.ietf.org/html/draft-ietf-rtcweb-jsep-24/

[19] D. McGrew and E. Rescorla, *RFC 5764: Datagram Transport Layer Security (DTLS) Extension to Establish Keys for the Secure Real-time Transport Protocol (SRTP)*, IETF, May 2010. [Online]. Available: https://datatracker.ietf.org/doc/rfc5764/

[20] A. Lemesle, A. Marion, L. Roux, and A. Gouaillard, "NARVAL, a no-reference video quality tool for real-time communications," in *Proc. of Human Vision and Electronic Imaging*, Jan. 2019, [submitted].

[21] A. Mittal, A. K. Moorthy, and A. C. Bovik, "No-reference image quality assessment in the spatial domain," *IEEE Transactions on Image Processing*, 2012.

[22] N. D. Narvekar and L. J. Karam, "A no-reference image blur metric based on the cumulative probability of blur detection (CPBD)," *IEEE Transactions on Image Processing*, 2011.

| | SFU | (A) Available send bandwidth (bps) | (B) Actual encoder bit rate sent (bps) | (C) Transmit bit rate sent (bps) | (D) Target encoder bit rate sent (bps) | (E) Average bit rate video sent (bps) | (F) Sender googRtt (ms) |
|---|---|---|---|---|---|---|---|
| MIN (value > 0 only) | Jitsi | 113,118 | 153,952 | 156,744 | 113,118 | 130,881 | 1 |
| | Janus | 744,557 | 676,624 | 690,904 | 744,557 | 611,518 | 1 |
| | Medooze | 92,471 | 79,440 | 83,184 | 92,474 | 94,565 | 1 |
| | Kurento | 34,476 | 28,088 | 32,272 | 34,476 | 35,501 | 1 |
| | Mediasoup | 93,617 | 91,488 | 95,576 | 93,617 | 93,860 | 1 |
| MAX | Jitsi | 8,342,878 | 2,397,504 | 2,544,144 | 1,700,000 | 2,218,333 | 168 |
| | Janus | 5,000,000 | 2,081,072 | 2,108,224 | 1,700,000 | 1,963,089 | 435 |
| | Medooze | 9,130,857 | 2,782,136 | 4,660,656 | 1,700,000 | 4,052,553 | 103 |
| | Kurento | 600,000 | 893,912 | 1,003,728 | 600,000 | 676,504 | 2,371 |
| | Mediasoup | 7,184,000 | 2,049,872 | 2,088,776 | 1,700,000 | 2,131,814 | 688 |
| Average | Jitsi | 2,830,383 | 1,360,703 | 1,392,607 | 1,362,722 | 1,388,670 | 18 |
| | Janus | 4,210,276 | 1,647,705 | 1,677,538 | 1,641,714 | 1,682,062 | 61 |
| | Medooze | 2,309,558 | 1,000,979 | 1,045,173 | 1,009,288 | 1,044,573 | 10 |
| | Kurento | 369,775 | 335,393 | 359,979 | 356,457 | 359,540 | 576 |
| | Mediasoup | 2,326,640 | 1,385,142 | 1,416,096 | 1,401,947 | 1,414,368 | 18 |
| % bit rate > 1 Mbps (columns (A) to (E)) | Jitsi | 65.4% | 65.4% | 65.7% | 65.4% | 65.7% | 6.4% |
| | Janus | 98.9% | 98.4% | 98.4% | 98.4% | 98.9% | 27.8% |
| % RTT > 50 ms (column (F)) | Medooze | 47.4% | 47.4% | 47.8% | 47.8% | 48.6% | 1.9% |
| | Kurento | 0.0% | 0.0% | 0.8% | 0.0% | 0.0% | 59.3% |
| | Mediasoup | 77.6% | 76.1% | 76.9% | 77.6% | 76.3% | 6.5% |

TABLE IV: Ramp-up phase: Overview of the sender's video statistics collected on each web client
Values in columns (A), (B), (C), (D), (E), (F) as per getStats.

| | SFU | Video receive client 1 | Video receive client 2 | Video receive client 3 | Video receive client 4 | Video receive client 5 | Video receive client 6 |
|---|---|---|---|---|---|---|---|
| Average | Jitsi | 1,256,874 | 1,266,195 | 1,294,939 | 1,282,719 | 1,239,218 | 1,264,753 |
| | Janus | 1,612,145 | 1,599,621 | 1,609,458 | 1,597,015 | 1,576,674 | 992,048 |
| | Medooze | 1,044,738 | 1,034,351 | 1,002,854 | 1,052,769 | 966,064 | 984,617 |
| | Kurento | 284,624 | 293,876 | 286,889 | 326,107 | 344,997 | 329,671 |
| | Mediasoup | 1,384,331 | 1,367,255 | 1,378,114 | 1,378,644 | 1,375,112 | 1,312,928 |
| % bit rate > 1 Mbps | Jitsi | 54.6% | 58.9% | 60.4% | 60.4% | 58.6% | 57.9% |
| | Janus | 96.7% | 95.6% | 96.2% | 96.7% | 94.4% | 58.0% |
| | Medooze | 50.7% | 49.1% | 47.8% | 51.4% | 42.6% | 41.4% |
| | Kurento | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% |
| | Mediasoup | 73.9% | 72.7% | 74.9% | 74.5% | 73.7% | 69.6% |

TABLE V: Ramp-up phase: Average bit rates (in bps) for the 6 videos received on each of the 6 clients in a room.
For client i: avgBitrate $= \frac{((\text{bytesReceived at t2}) - (\text{bytesReceived at t1})) \times 8000}{\text{timestamp at t2} - \text{timestamp at t1}}$

| | SFU | (A) Sender googRtt (ms) | (B) Sender transmit bit rate (bps) | (C) Video receive client 1 | (D) Video receive client 2 | (E) Video receive client 3 | (F) Video receive client 4 | (G) Video receive client 5 | (H) Video receive client 6 |
|---|---|---|---|---|---|---|---|---|---|
| MIN (value > 0 only) | Jitsi | n/a | n/a | n/a | n/a | n/a | n/a | n/a | n/a |
| | Janus | 13 | 564,512 | 575,744 | 514,087 | 411,512 | 533,318 | 596,490 | 584,226 |
| | Medooze | 1 | 86,864 | 85,139 | 88,002 | 90,285 | 87,321 | 96,191 | 66,047 |
| | Kurento | 144 | 65,560 | 33,544 | 42,949 | 43,964 | 38,414 | 44,892 | 31,274 |
| | Mediasoup | 1 | 30,272 | 25,031 | 25,931 | 28,374 | 39,130 | 22,948 | 25,088 |
| MAX | Jitsi | n/a | n/a | n/a | n/a | n/a | n/a | n/a | n/a |
| | Janus | 1,794 | 2,254,864 | 1,700,033 | 1,603,051 | 1,508,647 | 1,542,904 | 1,539,257 | 1,524,721 |
| | Medooze | 131 | 627,328 | 425,070 | 599,465 | 504,749 | 419,465 | 505,301 | 426,254 |
| | Kurento | 2,344 | 596,336 | 476,430 | 471,115 | 278,106 | 472,683 | 231,422 | 225,831 |
| | Mediasoup | 803 | 1,933,720 | 1,608,261 | 1,624,194 | 1,619,031 | 1,604,730 | 1,616,282 | 1,631,401 |
| Average | Jitsi | n/a | n/a | n/a | n/a | n/a | n/a | n/a | n/a |
| | Janus | 268 | 1,239,144 | 1,007,965 | 1,015,112 | 1,006,940 | 1,014,453 | 1,019,429 | 1,019,508 |
| | Medooze | 19 | 222,004 | 224,558 | 222,902 | 220,621 | 214,895 | 218,703 | 215,345 |
| | Kurento | 1,052 | 253,297 | 126,523 | 124,644 | 120,520 | 134,441 | 118,047 | 122,425 |
| | Mediasoup | 48 | 787,675 | 706,256 | 693,924 | 701,539 | 705,602 | 698,421 | 693,795 |
| % RTT > 50 ms (column (A)) | Jitsi | n/a | n/a | n/a | n/a | n/a | n/a | n/a | n/a |
| | Janus | 93.7% | 99.5% | 92.0% | 90.8% | 93.2% | 91.8% | 94.1% | 79.3% |
| % bit rate > 1 Mbps (columns (B) to (H)) | Medooze | 4.2% | 58.4% | 65.5% | 63.4% | 60.7% | 63.4% | 59.2% | 56.7% |
| | Kurento | 100.0% | 63.6% | 5.9% | 5.9% | 5.1% | 7.6% | 3.4% | 1.7% |
| | Mediasoup | 23.1% | 91.6% | 93.5% | 93.7% | 93.9% | 93.7% | 92.9% | 89.4% |

TABLE VI: Target load reached: Sender's RTT (in ms) and transmit bit rates (in bps), and average bit rates for the 6 videos received, at the end of the test, when all the meeting rooms have been created and all clients are connected to the SFU.