Comparative study of WebRTC SFUs for Video Conferencing

1st Emmanuel Andre
KITE testing group
CoSMo Software
Singapore
emmanuel.andre@cosmosoftware.io

2nd Ludovic Roux
Research group
CoSMo Software
Singapore
ludovic.roux@cosmosoftware.io

3rd Alexandre Gouaillard
CoSMo Software
Singapore
alex.gouaillard@cosmosoftware.io

***Abstract*—WebRTC capable media servers are ubiquitous, and among them, SFUs 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 any WebRTC SFU. We propose here a new approach based on the KITE testing engine. We apply it to the comparative study of four 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 insight on those SFUs scalability.**

***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 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 need media mixing capacities and will choose a Multipoint Control Unit (MCU) []. However most of the newer media servers are designed as Selective Forwarding Units (SFU) design [] that not only is 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 later allowing even better resilience against network quality problems like packet loss.

Even when focusing only on use cases that are implementable with an SFU, there are many use cases left. Arguably the two most important 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 media server they test, and cannot be reused to test others, or to make a comparative study. Moreover the published benchmarks differ so much in term of testing 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 simple scalability testing of a video conference use case (one-to-many) using a single WebRTC SFU media server. The novelty here is the capacity to run the same test scenario against several media servers installed on the same instance type.

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

II. Background

Verification and Validation (V&V) is the set of techniques that are used to assess software products and services. For a given software, testing is defined as observing the execution of the software on specific subset of all possible inputs and settings and provide an evaluation of the output or behavior according to certain metrics. [1]

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. [2]

In their 2006 paper, Di Lucca and Fasolino 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 [3].

A. Dedicated WebRTC Testing tools

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

WebRTCBench [4] is an open-source benchmark suite introduced by University of California, Irvine in 2015. This framework aims at measuring performance of WebRTC P2P connection establishment and data channel / video call. It does not allow to test media servers. It has not been updated since August 2015.

The Jitsi team has developed Jitsi-Hammer [5], an ad-hoc traffic generator dedicated to test the performance of their open-source Jitsi Videobridge [6].

The developers of Janus [7], an open-source WebRTC gateway, have first developed an ad-hoc tool to assess the performance of their gateway in several different configurations. From this initial assessment, Janus team has proposed Jattack [8], an automated stressing tool for the analysis of performance and scalability of WebRTC-enabled servers. They claim in the paper that Jattack is generic, that it can be used to assess the performance of WebRTC gateways other than Janus. However the code not being available, it could not be verified.

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

WebRTC-test [9] 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.

At last, Red5 has re-purposed an open-source RTMP load test tool called "bees with machine guns" to support WebRTC [10].

B. Generic WebRTC Testing

In the past few years, several research groups have been addressing the specific problem of generic WebRTC testing, e.g. 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, renamed Elastest, and the KITE Projects have generated quite a few articles on the subject.

1) Kurento Test Framework a.k.a. elastest:

In [11], 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 1 server, 1 room and 50 viewers.

In [12], the authors add 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 also reach 200 viewers, at the cost of actually 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 [13], the authors add support to KTF (now called ElasTest) for testing mobile devices through Appium. It is not clear is they support mobile browsers, and if yes, which browsers on which OS, or mobile apps. They now install KMS through OpenVidu. They also propose to extend the WebDriver protocol to add several APIs. While WebDriver protocol implementation modifications are easy on the Selenium side, 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 be very active in the WebRTC testing field with an original focus on Compliance Testing for the WebRTC standardization process at W3C [14].

KITE is running around a thousand of tests across 21+ browsers / browsers revisions / operating systems / OS revisions on a daily basis, whose 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 all supported browsers, 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. Moreover 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 Conferencing 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 are been run on separate Virtal 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 setup the following 4 open-source WebRTC SFUs using the latest source code downloaded from their respective public GitHub (except for Kurento/openvidu, for which the Docker

[1]https://webrtc.org/testing/kite/

[2]https://www.w3.org/TR/webrtc/

container was used) in a separate AWS EC2 Virtual Machine, with default configuration:

- Jitsi Meet (JVB version 0.1.1043)[3]
- 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]

Note that on OpenVidu, the min and max bitrates are hardcoded at 600,000 bps in MediaEndpoint.java at line 276. Jitsi also sets a 650,000 bps limit for the sending bitrate. The limit is 1,700,000 bps for Janus and Medooze.

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

C. Web client Applications

To test the four SFUs with the same parameters and to collect useful information (getStats and full size screen captures see section III-E), 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 (only when configurable on the client web app)
- support for displaying up to 9 videos with the exact same dimensions as the original test video (540x360 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 seven streams (one sent and six 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 2).

Fig. 1 shows a screenshot of the modified Janus Video Room web client application with the changes described in **??**. The self video (in loopback) is displayed at the top left corner, then the videos received from each of the six remote clients are displayed as shown on Fig. 1. The original dimensions of the full image with the 7 videos are 1980x1280 pixels, and the user id, dimension and bitrates are displayed above the video leaving it free of obstruction for the quality analysis.

[3]https://github.com/jitsi/jitsi-meet
[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

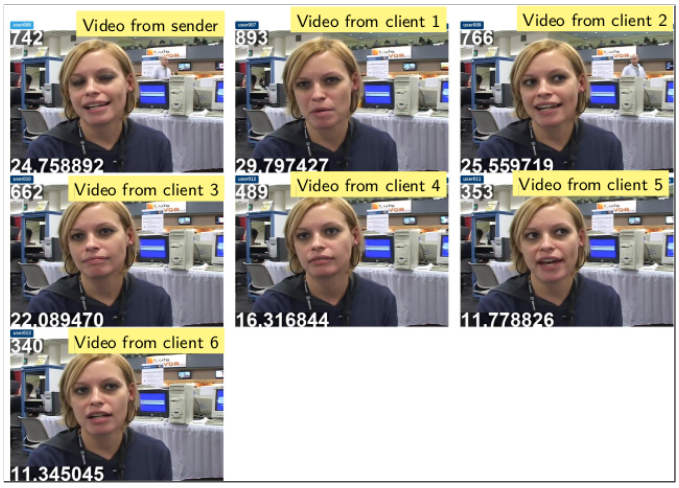


Fig. 1: Screenshot of Janus Video Room Test web app after modifications (yellow text boxes have been added to the screenshot to indicate the location of videos received from the sender and from each of the 6 clients, but are not present in the images collected in the process).

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 send, in order to make quality measurements, we use Chrome fake media functionality. Each client plays the same video.[10] 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`

The last option (`window-size`) was set to be large enough to accommodate 7 videos on the screen with original dimensions of 540x360 pixels.

E. Metrics and Probing

1) Client-side: getStats() function:

The test calls getStats() to retrieve all the statistics values provided by Chrome. The bitrates for the sent 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.

[10]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, 540x360 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 540x360 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 right. The original audio part, MPEG-AAC 44100 Hz 128 kpbs, has been converted using ffmpeg to WAV 44100 Hz 1411 kbps.

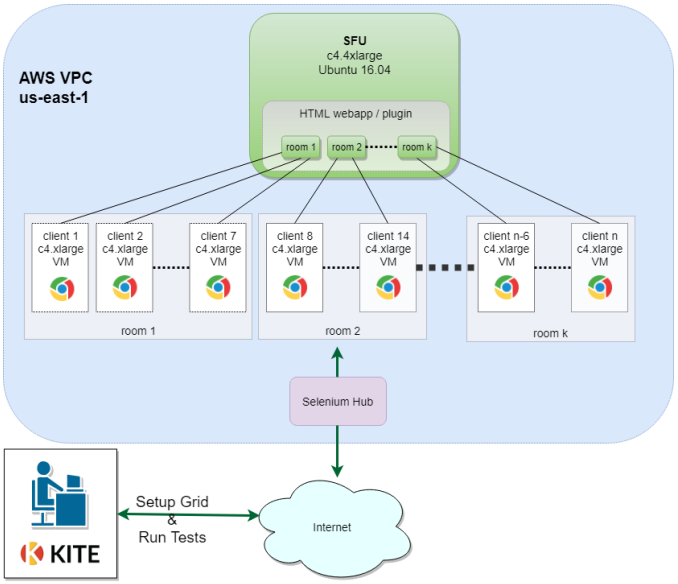


Fig. 2: Configuration of the tests.

2) Client-side: Video verification:

Once a `<video>` element has been loaded, we verify that it actually displays a video, and that it is neither blank nor static or 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 the video element is not receiving any stream and the display is black. If the return values of the two calls for a given video element are positive but the difference is null then the video is frozen.

3) Client-Side: Video Quality Assessment:

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

Beyond the networks measurements reported above, we have applied 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) [15], contrast, cumulative probability of blur detection (CPBD) [16], edges detected by Sobel and Canny, blur, 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. The network has been trained on the publicly available annotated video dataset IRCCyN IVC 1080i [17] [18].

F. Test methodology

The tests are executed with KITE [14]. They are therefore written in Java and rely on Selenium WebDriver to launch and control the client browsers. An overview of the configuration of the tests is provided in Fig. 2.

	Jitsi	Janus	Medooze	OpenVidu
Number of rooms	70	35	70	20
Number of client VMs	490	245	490	140

TABLE II: Load test parameters per SFU (there are 7 participants in each room).

1) per SFU test logic:

For Each of the 4 SFUs we execute the following steps.

- KITE will first instantiate the required number of client VMs.
- For each room in a given SFU (the number of rooms depends on the SFU, see II):
  - For each of the clients VM 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 (SUCCESS) or time out (FAILED)`
- `wait for self video to appear (SUCCESS) or time out (FAILED)`
- `wait for all remote videos to appear (SUCCESS) or time out (FAILED)`

3) Measure and probe:

getstats is being called with 1 second interval n times, n=2 while filling the room, and n=8 once the corresponding room is at capacity.

- `Take a first screenshot`
- `call getstats and wait one second, n times.`
- `Take a second screenshot`

4) Maximum Load Target:

We started by doing 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 OpenVidu to 20 rooms as that SFU usually failed before reaching that load, and to set a target of 35 rooms for Janus as we observed the bit rate became extremely low (below 10 kbps) once it reached 25 rooms

For the final load test and 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.

IV. Results

A. Quantitave Results

On table III, we present the rate of SUCCESS and FAILURES that occurred during the tests. No failure was reported

	SFU	Page loaded	%	Sender video check	%	All videos check	%
PASS	Jitsi	489	100%	489	100%	102	21%
	Janus	243	100%	243	100%	238	98%
	Medooz	481	100%	481	100%	434	90%
	OpenVi	118	100%	118	100%	69	58%
FAIL	Jitsi	0	0%	0	0%	387	79%
	Janus	0	0%	0	0%	5	2%
	Medooz	0	0%	0	0%	47	10%
	OpenVi	0	0%	0	0%	49	42%

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 SFU from the six clients passed the video check.

concerning the display of the self video. However, there have been some failures. A failure in that case means that at least one of the 6 remote videos is either not received (media not connected), blank (media not flowing) or frozen. The failure rate is very high for Jitsi at 79% because in most cases still images are displayed instead of the video (correspondingly the measured bitrates are zero). There is also a high failure rate of 42% for OpenVidu with many clients missing one or two videos. We can see a low failure rate of 10% for Medooze, and very low for Janus at 2%.

B. Bitrates measurements

1) Sending Streams :

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)), the highest is reported for Medooze with about 1 Mbps. This is the expected bitrate for the controlled input media encoded with modification. Both Jitsi and Janus have a similar sending bit rate, respectively 530 kbps and 505 kbps. While this was expected for Jitsi given the bandwidth cap in place in the server code, this is unexpected for Janus. OpenVidu's bit rate is the lowest at about 350 kbps. Here again, we cannot fully explain this behavior.

Table VI reports also the bit rates for the sender and the client at target load reached. These values have decreased sharply from the ramp-up phase. First let's have a look at the sender average bit rate. OpenVidu is the one which manages to have the lowest decrease for the sender average bit rate. At ramp-up it was about 350 kbps, and at target load it is about 250 kbps, roughly 30% less only. Medooze, which had the highest average bit rate of about 1 Mbps during ramp-up, see it falling down to about 215 kbps at target load, a drop of almost 80%. Jitsi and Janus experience an even stronger bit rate decrease between ramp-up and target load. Jitsi's average bit rate of about 500 kbps at ramp-up has fallen down to only 45 kbps at target load, a drop of 91%. For Janus, the figures are 530 kbps at ramp-up, down to slightly less than 30 kbps at target load, a drop of 94.5%.

2) Receiving Streams :

The statistics on the bit rates for the 6 videos received by each client are reported in table V. Medooze is the only one to manage to have the bit rate of each of the 6 clients almost equal to the bit rate of the sender at nearly 1 Mbps. Janus, with about 440 kbps for the clients, is not far away to the same bit rate of 530 kbps reached for the video from the sender. Same for OpenVidu. With a bit rate of about 300 kbps for the clients, it is not too far away from the 350 kbps of the sender. For Jitsi however, there is a large difference between the 125 kbps bit rate of the receiving clients, and the 530 kbps bit rate of the sender.

At last, we present the average bit rates for the clients. We observe virtually no transmission of video on the clients of Jitsi as the average receiving bit rate is only 435 bps. This is nothing compared to the 125 kbps bit rate observed during ramp-up. This confirms the information on Table III where the proportion of failures during video check (corresponding to a blank or still image) reached 79% for Jitsi. Average client bit rates for Janus is also extremely low, being only slightly above 10 kpbs at target load (it was about 440 kbps during ramp-up). Interestingly, despite such a low bit rate, video check was very good with only 2% failures. OpenVidu has an average client bit rate of about 120 kbps at target load, down from 300 kbps during ramp-up. Medooze delivers the best average bit rate at target load with about 250 kbps, just a quarter of the 1 Mbps observed during ramp-up.

C. Latency measurements

Last column of Table IV gives us the RTT measures during the ramp-up phase. The figures are very low for Medooze and Jitsi as they have respectively an average RTT of 10 ms and 15 ms. Although we are only during the ramp-up phase of the test, OpenVidu already exhibits a pretty high average RTT of 576 ms, while Janus has a terrible average RTT of more than 2.6 seconds.

After the ramp-up phase, i.e. when the target load for each SFU has been reached, we present in Table VI the RTT and the average bit rates. Concerning the RTT, the figures observed during ramp-up have increased. Roughly, they are the double of what they were during ramp-up. Both Medooze and Jitsi manage to maintain a very low average RTT, respectively 19 ms and 37 ms. OpenVidu has now an average RTT slightly over one second. And the already very long average RTT of more than 2.6 seconds during ramp-up for Janus is over 6.5 seconds at target load.

D. Video Quality assessment

Estimation of video quality scores is presented in Fig. 3. We may have expected the video quality to worsen as the number of participants increased, but it is not always true.

Medooze receives the best scores, and it is able to maintain a good image quality up to 400 participants. Above that figure, image quality starts to worsen.

Jitsi is able to keep about the same image quality score for the whole test. Even when 500 participants have joined the test, image quality remains stable, but quite lower than

	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	10,000	10,464	8,376	15,000	10,626	1
	Janus	10,000	8,288	9,144	18,635	9,420	1
	Medooze	92,471	79,440	83,184	92,471	94,565	1
	OpenVidu	34,476	28,088	32,272	34,476	35,501	1
MAX	Jitsi	6,385,172	1,081,750	1,956,270	650,000	1,432,475	967
	Janus	5,000,000	2,024,552	2,014,064	1,700,000	2,061,430	9,071
	Medooze	9,130,857	2,782,136	4,660,656	1,700,000	4,052,553	103
	OpenVidu	600,000	893,912	1,003,728	600,000	676,504	2,371
Average	Jitsi	1,741,272	482,678	527,541	536,767	516,150	15
	Janus	1,061,277	527,410	555,641	515,867	542,146	2,629
	Medooze	2,309,558	1,000,979	1,045,173	1,009,288	1,044,573	10
	OpenVidu	369,775	335,393	359,979	356,457	359,540	576
% bit rate > 1 Mbps (columns (A) to (E))	Jitsi	36.0%	0.2%	1.0%	0.0%	0.4%	2.9%
	Janus	22.6%	22.6%	25.1%	22.2%	25.1%	83.1%
% RTT > 50 ms (column (F))	Medooze	47.4%	47.4%	47.8%	47.4%	48.6%	1.9%
	OpenVidu	0.0%	0.0%	0.8%	0.0%	0.0%	59.3%

TABLE IV: Overview of the sender's video statistics collected on each client during ramp-up phase of the test. 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	125,061	135,241	127,464	135,951	125,232	122,008
	Janus	439,722	450,628	445,271	430,724	444,696	438,957
	Medooze	1,044,738	1,034,351	1,002,854	1,052,769	966,064	984,617
	OpenVidu	284,624	293,876	286,889	326,107	344,997	329,671
% bit rate > 1 Mbps	Jitsi	0.2%	0.8%	0.6%	0.6%	0.4%	0.6%
	Janus	18.5%	18.1%	18.5%	16.9%	18.1%	18.1%
	Medooze	50.7%	49.1%	47.8%	51.4%	42.6%	41.4%
	OpenVidu	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%

TABLE V: Average bit rates (in bps) for the 6 videos received during the ramp-up phase 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) Transmit bit rate sent (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	10	5,856	2,318	1,541	787	1,576	2,788	2,997
	Janus	1,092	6,456	2,239	1,875	1,910	1,928	2,244	2,750
	Medooze	1	86,864	85,139	88,002	90,285	87,321	96,191	66,047
	OpenVidu	144	65,560	33,544	42,949	43,964	38,414	44,892	31,274
MAX	Jitsi	148	436,744	4,174	8,211	9,008	1,576	3,009	4,018
	Janus	10,500	418,744	271,667	248,181	278,802	262,048	315,127	256,863
	Medooze	131	627,328	425,070	599,465	504,749	419,465	502,301	426,254
	OpenVidu	2,344	596,336	476,430	471,115	278,106	472,683	231,422	225,831
Average	Jitsi	37	46,285	341	797	724	61	338	351
	Janus	6,535	30,302	11,407	11,628	9,482	9,089	11,960	11,383
	Medooze	19	222,004	224,558	222,902	220,621	214,895	218,703	215,345
	OpenVidu	1,052	253,297	126,523	124,644	120,520	134,441	118,047	122,425
% RTT > 50 ms (column (A))	Jitsi	19.2%	3.8%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
	Janus	90.2%	1.6%	0.4%	0.8%	1.2%	0.4%	1.2%	0.8%
% 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%
	OpenVidu	100.0%	63.6%	5.9%	5.9%	5.1%	7.6%	3.4%	1.7%

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

Medooze. It is most certainly due to the fact that Jitsi uses simulcast on the sender-side, last-N in the server, and a specific client configuration were all but one video is displayed as a low resolution thumbnail. It biases the quality measure by comparing a low resolution video where for all the other SFUs clients would have a high resolution frame. Those extra Jitsi feature are supposed to bring extra scalability to Jitsi, and it would be interesting in a future work to take it into account for the quality measurement.

Videos delivered through Janus gateway are of pretty good quality when there are few participants. But very quickly, above 70 participants, image quality drops sharply and is really bad at 200 participants.

OpenVidu video quality varies awkwardly according to the load. It worsen almost immediately and reaches a lowest image quality at about 100 participants. As more participants are joining the test, surprisingly image quality improves steadily up to about 130 participants, then the quality drops again.

V. Analysis

This study exhibits interesting behaviours of the 4 SFUs that has been evaluated when they have to handle an increasing number of rooms and peer in a video conference use case.

Results per SFU across time are summarized in figure 4, first two columns from the left. Results per SFU as a function of the number of users connected are presented in the same figure, 3rd (and last) column from the left. Note that those graphs are not directly comparable as the range of the x-axis is not the same for each SFU. Finally, the consolidated Quality Assessment of each SFU as a function of the load (number of users) is presented in figure 3.

First, only Jitsi and Medooze were able to reach a load target of 70 rooms and 490 participants. While OpenVidu remains active when the number of rooms reaches or goes beyond its target load of 20, the video check fails, which means that on the web app, at least oen video is either totally missing, black, or frozen.

Among the 4 SFUs tested, Medooze is the one exhibiting the best performance. It delivers the highest bit rate and has the lowest RTT both during ramp-up and when target load is reached. As a result, videos transmitted to all the participants are of better visual quality. In addition, it has a pretty low failure rate on the video check. It would be interesting to see in a future work if the load on the server (CPU footprint, RAM footprint, NIC usage) is comparable. Also, it is possible that with Jitsi specific features taken into account, the video quality of Medooze and Jitsi end up being within a small tolerance range of each other.

Jitsi shows a very good ability to cope with a high load. RTT is very low, however the average bit rate measured is extremely low, resulting in lower video quality as compared to Medooze when videos are able to flow between peers. In fact, the bit rate is so low that few images of the videos are transmitted to the peers, resulting in many blank or still images, as shown by a very high 79% rate of failures on the video check.

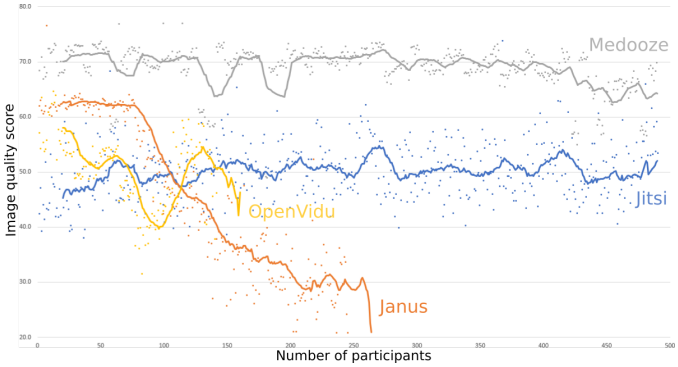


Fig. 3: Video quality scores over the number of participants.

On the contrary, Janus has a lot of difficulty to handle a lot of rooms and peers. Very quickly, its performance drops dramatically. RTT explodes, it is more than 6.5 seconds on average at target load, while the bit rates falls down. Consequently, the video quality becomes really bad when there are more than 140 participants in the test.

OpenVidu is not able to handle well more than 150 participants.

VI. Conclusion and Future Work

We have shown that it is now possible to comparatively test WebRTC SFUs using KITE. A lot of 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 test themselves. In the future we would like to add more metrics and tests on client side, for example to assess audio quality as well. On 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 number of users per rooms a variable of the test run. That would allow to reproduce the results of [5] and [8].

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.

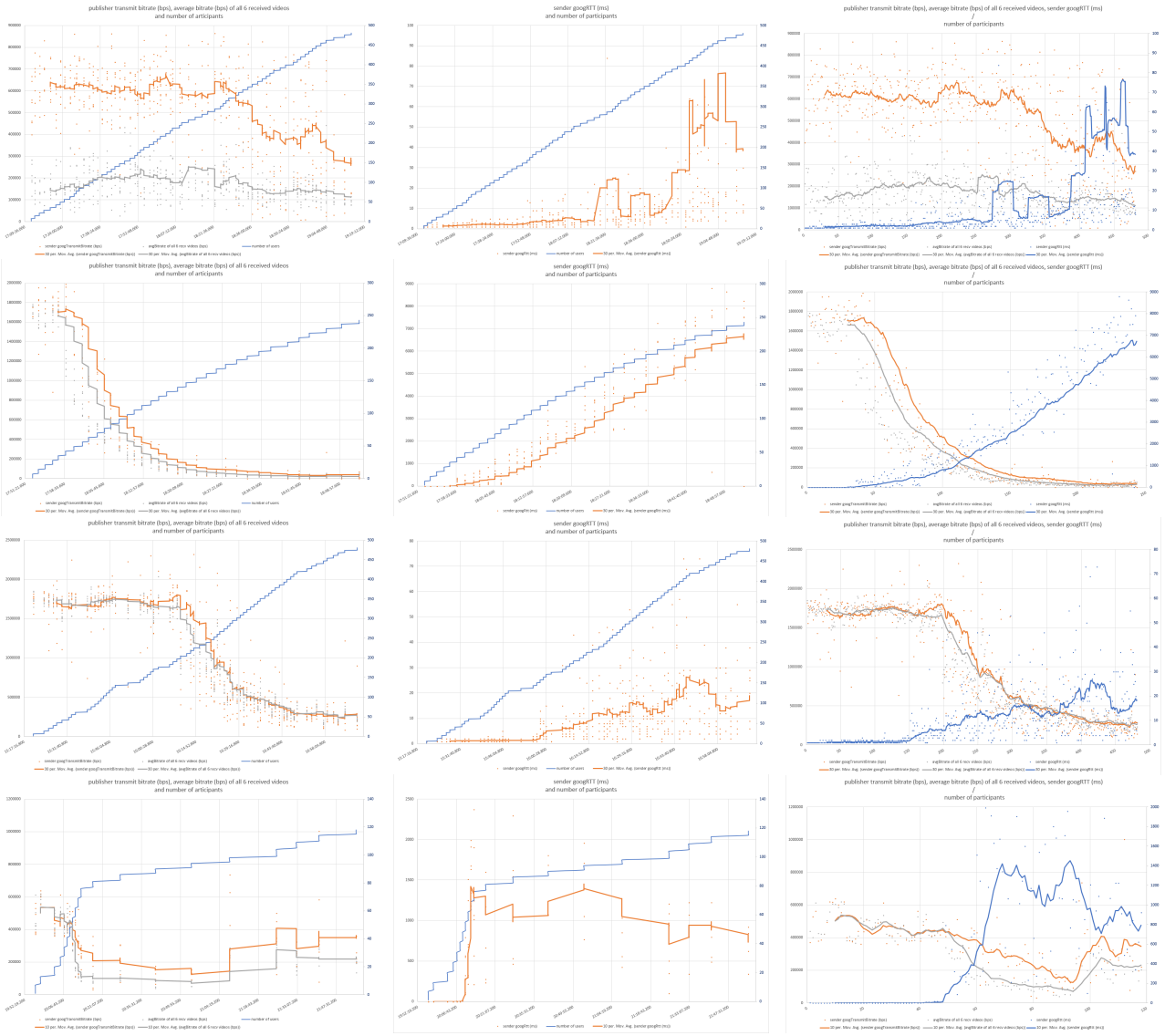
Acknowledgment

We would like to thanks Boris Grozev (Jitsi), Lorenzo Miniero (Janus), Inaki 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. We would also like to thanks Boni Garcia for discussions around the Kurento Team Testing research results.

References

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

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



(a) Transmission bitrate (TB).
Red: sender transmit bitrate (bps),
Grey: avg bitrate (bps) of received videos,
Blue: number of participants over time

(b) Round-trip time.
Red: sender googRTT (ms),
blue: number of participants over time

(c) TB and RTT over # participants.
Red: TB, Blue: RTT
Grey: avg bitrate (bps) of received streams

Fig. 4: **Results for all SFU**. Each row correspond to one SFU, from top to bottom: Jitsi, Janus, Medooze, KMS / OpenVidu.

[3] 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.

[4] 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 *ESTIMedia 2015, 13th IEEE Symposium on Embedded Systems For Real-time Multimedia*, 2015.

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

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

[7] 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.

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

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

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

[11] 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.

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

[13] 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.

[14] A. Gouaillard and L. Roux, "Real-time communication testing evolu-

tion with WebRTC 1.0," in *IPTComm 2017, Principles, Systems and Applications of IP Telecommunications*, 2017.

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

[16] 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.

[17] S. Péchard, R. Pépion, and P. Le Callet, "Suitable methodology in subjective video quality assessment: a resolution dependent paradigm," in *IMQA 2008, International Workshop on Image Media Quality and its Applications*, 2008.

[18] IRCCyN IVC 1080i video quality database. [Online]. Available: http://ivc.univ-nantes.fr/en/databases/1080i_Videos/