Jattack: a WebRTC load testing tool

A. Amirante*, T. Castaldi*, L. Miniero*, S. P. Romano[†*]

*Meetecho S.r.l., Via C. Poerio 89/a, 80121 Napoli, Italy

{alex, lorenzo, tobia}@meetecho.com

[†]University of Napoli Federico II, Computer Science Department, Via Claudio 21, 80125 Napoli, Italy

spromano@unina.it

Abstract—We present *Jattack*, an automated stressing tool for the analysis of the performance of WebRTC-enabled server-side components. Jattack has been initially conceived with the primary objective of performing a thorough scalability analysis of the well-known Janus WebRTC gateway. As such, it re-uses most of the Janus core stack components in order to reliably emulate the behavior of a dynamically adjustable number of WebRTC clients. The specific testing scenario can indeed be programmatically reproduced by writing a small "controller" component, which takes on the responsibility of properly orchestrating the scenario itself. The general-purpose nature of the tool, together with its flexibility deriving from the controller-based programmable approach, makes Jattack also suitable for stress-testing other WebRTC-enabled servers.

I. INTRODUCTION

The motivation behind this paper stems from the concrete need for a flexible, lightweight and reliable testing tool for the assessment of the scalability of WebRTC-enabled servers. These components definitely represent strategic assets of a number of Web-based real-time multimedia systems that are starting to be used at very large scales. The authors of this paper are the developers of the Janus[TM] WebRTC gateway [1], a well-known open source component that is currently leveraged by a number of small and medium size enterprises having in common the need for WebRTC-enabled interaction among their clients, as well as interoperability with "legacy" real-time communication frameworks based, e.g., on SIP (Session Initiation Protocol) [2]. Just to cite a very well-known example, Slack[TM] is currently using Janus for the implementation of the audio call functionality of their application [3].

Given the increasing level of deployment of Janus as a key backend component of the aforementioned systems, the need has arisen to properly assess its scalability properties in a reliable way. When we started to tackle such an issue, we had to confront ourselves with a number of challenges, especially when trying to automatically generate a suitable stress testing load capable to properly mimic the behavior of real-world WebRTC clients. Indeed, more often than not we found ourselves struggling with the issue of properly scaling the client-side of the integrated WebRTC system under test before being capable of arriving at a load which might be deemed suitable in order to assess the scalability of even a single Janus instance. The testing campaigns, most of the times deployed in the cloud, took us a lot of effort, in terms of both platform configuration and cost.

We hence decided to work on a suitable load generator which might be leveraged in a programmatic fashion in order to rapidly prototype customized testing scenarios involving a significant number of WebRTC client instances capable to reproduce the typical behavioral profile of the users of a specific Janus-enabled framework. The idea was to become able to keep the load on the client-side as low as possible, hence moving the bottleneck to the server-side of the integrated system and allowing us to easily assess the scalability of the Janus servers residing in the backend.

The tool we devised has been conceived at the outset as a general purpose one. As it will be explained in the next sections of the paper, it is made of two main components: (i) a core implementing the fundamental WebRTC functionality and related stack; (ii) a programmable controller allowing to easily build and drive a specific testing scenario. The adoption of the principle of separation of concerns, with special regard to the availability of the programmable orchestrating part, indeed allows to use our stress-testing tool for the assessment of server-side WebRTC components other than Janus.

II. RELATED WORK

The automated testing of WebRTC applications and services definitely represents a challenging issue. Being able to design and execute a stress test campaign can be of paramount importance when evaluating a new service, a new WebRTC endpoint or any kind of WebRTC component. Stress testing, in fact, allows to assess the scalability and the performance of the target. Lately, there have been several efforts devoted to this area of research.

The most popular means for testing WebRTC applications in an automated fashion is by using the well known Selenium Framework [4], which allows for a complete simulation of a browser's behavior. This framework, in fact, allows developers to remotely control and drive browser instances through software modules called "webdrivers". A controller application, then, can be written by leveraging different language bindings (e.g., Java or Python) to decide things like which page to open, what user input to simulate, and so on. As such, it is usually fairly easy to deploy headless instances of browsers (e.g., Chrome, Firefox) on dedicated servers and have them open and "navigate" properly crafted web pages that allow for a partial, if not full, automation of a user's interaction. When these interactions involve the creation of WebRTC PeerConnections, this allows for the automated creation of

multiple WebRTC resources that can be used for the purpose. The advantage of this solution is that it allows users to re-use most of the resources already available when designing a test campaign: in fact, most of the times you can run tests on the very same pages that would actually be accessed by users themselves, taking advantage of the scripting and programmable features of both JavaScript and the language used in the controlling application. We ourselves have used this approach repeatedly in our tests, and have shared our results in a recently published paper [5]. The same approach has been adopted in [6], where the authors evaluated the performance of Kurento Media Server [7] in various scenarios. To the best of our knowledge, Selenium is also the foundation of a popular WebRTC testing service as well, TestRTC [8]. That said, while easy and effective, this approach is nonetheless quite demanding in terms of resources. In fact, this requires actual browser instances to be deployed on server machines that can be used for the testing. Since browsers are typically not very lightweight applications, and given the fact that there is a lot of overhead in terms of what such browser instances actually do when opening a target web page, the number of concurrent PeerConnections that can be established and maintained on a single machine is limited. This makes it hard to effectively assess the scalability of a component without an adequate number of client machines available for testing.

For this reason, when either the availability of resources is constrained or there is a need for more efficiency, it is in general advisable to rely on some kind of application that allows for a better management of resources (e.g., to only implement the WebRTC communication), while leaving the rest of the interactions (e.g., authentication, signaling, automated input, and so on) to other tools. Unfortunately, just a few tools can satisfy this requirement. In fact, most of the available automated web tools are specifically targeted at stress testing the web-based behavior of an application, e.g., in terms of HTTP and/or WebSocket [9] requests.

One notable exception is Jitsi Hammer[10], a tool devised and implemented by Jitsi as a traffic generator for their Videobridge application. This tool is a Java application that can be configured to connect to a Jitsi-Meet conference, create fake users, and generate/receive RTP traffic on behalf of the simulated users within the room [11]. This is a very good example of how resources can be better managed for the purpose of assessing the performance of a component. In fact, while it is true that the same results might be achieved through the Selenium Framework (e.g., by having multiple browser instances all access the same web page and provide credentials in an automated way to join a meeting room), it is fairly obvious that a dedicated application allows for a much more optimized management of resources. In this case, the Jitsi Hammer tool takes care of the interaction with the backend on its own and, through configurable parameters, can be instructed to behave in different ways, all using less resources than a full browser would require. While a very effective tool, though, to the best of our knowledge Jitsi Hammer is specifically tailored to the Jitsi Videobridge, which means it cannot be used, for instance, as a stress testing tool to interact with different applications, possibly based on different backends.

Finally, there are some commercial solutions for testing WebRTC services [12], [13]; unfortunately, to the best of our knowledge, there is no public documentation on the tools they leverage to do performance evaluation.

These were the considerations that eventually led us to work on such a tool ourselves. While the main aim was, at first, an efficient tool we could use to test Janus instances while managing resources more efficiently, we soon found out that it could actually be generalized, and used to assess the performance of any WebRTC application, provided that the test campaign is designed properly.

III. A GENERAL-PURPOSE LOAD TESTING TOOL

Jattack (which stands for "Janus Attack", or the French "J'attaque") is a tool we conceived to quickly generate multiple WebRTC connections, for the purpose of stress testing WebRTC applications. We took the Janus WebRTC Gateway's open source core and slightly modified it for the purpose. As such, it is mainly composed of the same WebRTC stack of Janus, with a few differences.

Jattack is supposed to be a generic WebRTC stressing client tool, thus it is not strictly limited to Janus itself. Given its programmable nature, with a few customizations it can be easily fit to interact with different server-side components.

By itself, Jattack does no signaling at all: it only works when used with a *controller* that handles that part for it, and which orchestrates its actions. This means that, in general, users will have their own application talking to Janus (or their Janus-based service) and handling the signaling (JSEP [14], SDP [15], trickle ICE candidates [16]), and then bridging this info between Janus and Jattack. This allows for the realization of heterogeneous scenarios: for instance, the controller can orchestrate the creation of multiple PeerConnections of different types to simulate real scenarios (e.g., a multi-party conference, or a large webinar). Multiple Jattack instances can be used by the same controller for larger scale scenarios. Furthermore, multiple PeerConnections can share the same media source in Jattack, thus allowing for a very lightweight setup of active sources. It is worth noting that, with this approach, Jattack is completely relieved of any encoding/decoding burden, which is known to be the most resource-intensive task a WebRTC client has to perform.

The Jattack architecture is depicted in Fig. 1.

IV. CONTROLLER: THE BRAIN

Jattack uses WebSockets to connect to the controller, receive commands, and send responses/events. This means that a wannabe controller needs to implement the server-side of the `jattack-protocol` we conceived.

While we will not delve into the details of the protocol itself for the sake of brevity, it may be worth summarizing its syntax, with special regard to the inner workings of the request/response mechanism, as well as to how asynchronous

Fig. 1: Jattack architecture

notifications can be delivered within the context of an existing session. Jattack commands (requests) have the following syntax:

```
{
    jattack: '<command>',         // Command name, mandatory
    transaction: '<unique ID>',   // Transaction ID, mandatory
    id: <session ID>,             // Media session ID, optional
    body: {                       // Command specific payload, optional
        [..]
    }
}
```

A command, when handled, always results in a response from Jattack to the controller. Responses are formatted like this:

```
{
    jattack: 'success|error',     // Command result
    transaction: '<unique ID>',   // Transaction ID, same as request
    id: <session ID>,             // Media session ID, optional
    body: {                       // Command specific response, optional
        [..]
    }
}
```

As anticipated, though, not all the communication between Jattack and a controller is synchronous. Commands sent to Jattack can subsequently result in asynchronous events generated by the tool (e.g., to notify new candidates or ICE state changes). Events are formatted like this:

```
{
    jattack: '<event>',           // Event name
    id: <session ID>,             // Media session ID, optional
    body: {                       // Event specific payload, optional
        [..]
    }
}
```

We notice that events do not have a "transaction" field, as they are, again, asynchronous, and hence never explicitly related to any command. These events can be quite helpful to track the "life" of a PeerConnection, and at the same time collect statistics that can provide valuable information.

For the testing campaign we conducted, we wrote different controller logics, all implementing, on the Jattack side, the above mentioned protocol. Such controller logics handled signaling toward different Janus plugins, e.g., the Janus *Streaming*, *AudioBridge*, *VideoRoom* and *EchoTest* plugins. For the sake of simplicity, we chose to focus on the Janus

Streaming and *VideoRoom* plugins for this paper, as they are the most commonly used by developers for their Janus-based applications. As to the controller implementation, we chose node.js as the runtime environment, and talked to Janus through its WebSocket transport module.

V. JATTACK: THE ARM

The actions to undertake in order to create and manage a Jattack-based testing campaign are briefly listed below:

1) Jattack connects to, and registers at, the controller;
2) the controller properly handles Jattack's registration request;
3) one or more media sources are created, if needed (more on that later);
4) as many sessions as needed get created in a dynamic fashion (each session will be associated with a Peer-Connection);
5) one or more Janus instances get contacted and instructed to create as many PeerConnections as needed with the available Jattack instances;
6) if Jattack instances need to generate media, a media source is attached to them;
7) optionally, Jattack instances can be instructed to record the incoming frames (not mandatory, you can receive without recording).

The following subsections dig further into the details of the above mentioned procedures.

A. Registration

Upon connection, Jattack sends a `register` message to the controller. This is the only time that Jattack sends an active command. In general it will always just receive commands, while sending back responses and/or events.

B. Commands

Jattack can receive the following commands:

- `add-media-source`: add a new source for active RTP sessions (e.g., an external GStreamer/FFmpeg script sending media); such a source can be shared across multiple sessions at the same time; without a source, a Jattack session is receive-only;
- `remove-media-source`: destroy an existing media source;
- `create-session`: create a new media session; this is basically the same as a session+handle in Janus, meaning that a session is associated with a single PeerConnection;
- `destroy-session`: destroy an existing media session (destroys PeerConnection too, if it existed);
- `generate-offer`: have Jattack generate an offer in a session to create a new PeerConnection;
- `generate-answer`: have Jattack generate an answer in a session to create a new PeerConnection;
- `handle-offer`: pass a remote offer to Jattack in a session to create a new PeerConnection;
- `handle-answer`: pass a remote answer to Jattack in a session to create a new PeerConnection;

- `hangup`: hangup an existing PeerConnection;
- `get-state`: get the internals of an existing session (very similar to Janus admin API info).

A *media source*, in Jattack, is whatever can be used to have Jattack actively generate RTP data. Jattack does not capture/generate media by itself, meaning it does not access your microphone, your webcam, a file or anything like that. The only way to have a Jattack PeerConnection send media is by creating a media source and attaching it to the session. A media source listens on a couple of ports, and everything it receives on them (normally valid RTP) is sent over the session's PeerConnection by Jattack. Sending media to such ports can be achieved through external tools like *GStreamer* or *FFmpeg*. As already anticipated, multiple sessions can share the same media source in Jattack, which is completely relieved of any encoding/decoding task.

A Jattack *session* is the main purpose of Jattack itself. This is where a media session is created, that is something that will be strictly associated with a PeerConnection in Jattack itself. In Janus terms, it can be seen as a condensed session+handle, i.e., a session that only contains a single handle.

When a session is created, there is no PeerConnection yet. A PeerConnection becomes only available after a successful WebRTC negotiation. This involves SDP offers and answers, trickle ICE candidates, ICE connectivity checks, DTLS handshakes, and so on.

In order to create a PeerConnection in Jattack, and more specifically within a specific session that has been created, we need to first decide who will send the offer and who will instead provide the answer. This typically depends on the scenario we want to achieve and, in case Janus is involved, on the plugin that will be used for the purpose. To use either the Janus EchoTest, or the AudioBridge or even to create a VideoRoom publisher, for instance, the client is supposed to send the offer and the target plugin module on the Janus side to answer back. To create a VideoRoom subscriber or a Streaming viewer, instead, the offer always has to come from the Janus plugin, and the client has to answer back. For this reason, when Jattack acts as the offerer, a `generate-offer` command is always followed by a `handle-answer`; when the peer is the offerer, instead, a `handle-offer` is followed by a `generate-answer`. In both cases, the result is an attempted setup of a PeerConnection. This means that Jattack starts gathering and trickling ICE candidates, tries to send connectivity checks when it has enough information, starts a DTLS handshake when ICE is done and exchanges RTP/RTCP packets when WebRTC is ready. During the negotiation process, `handle-offer`, `handle-answer`, and responses to `generate-offer` and `generate-answer` have to carry an SDP body. As anticipated, this negotiation process starts a chain of activities and events that eventually lead to the setup of a WebRTC PeerConnection. All these activities will be notified to the controller via events so that the controller can use them as appropriate. A notable example of an important event to handle is `trickle`, as it is the event used to report ICE candidates Jattack has gathered for itself: the controller must forward those trickle candidates to the peer, or the WebRTC connectivity will most likely fail.

A PeerConnection stays alive until we destroy the session or simply hangup. Destroying the session means we would have to create a new one if we wanted to create a new PeerConnection, while only hanging it up allows us to re-use the existing session for a new PeerConnection. Tearing down a PeerConnection results in events sent to the controller.

In order to check the internal state of a Jattack session, including the state of the associated WebRTC PeerConnection, we introduced an ad-hoc request called `get-state`, which acts like the Janus admin API and returns similar info.

C. Events

Jattack can generate several different events towards the controller at any time. These include events related to any change in the state of the PeerConnection, the health of the media session, etc.. No action is needed from the controller when an event is received: it just needs to be prepared to receive and optionally handle them.

The events Jattack can generate are the following:

- `source-done`: a previously created source is over;
- `trickle`: Jattack gathered a new candidate in a session (the controller must forward it);
- `ice-state`: the ICE state of a Jattack session PeerConnection changed;
- `selected-pair`: an ICE pair was selected in a Jattack session PeerConnection;
- `webrtcup`: a PeerConnection in Jattack has just become active;
- `media`: Jattack started/stopped receiving media from the peer;
- `recording`: a Jattack recording state changed (started/completed);
- `hangup`: a PeerConnection in Jattack was just closed;
- `destroyed`: a Jattack session was just destroyed.

All these events can be very helpful to follow the evolution of an existing PeerConnection, and controllers may choose to save/store them in a structured way for offline evaluations.

VI. J'ATTAQUE! A CAMPAIGN AGAINST THE JANUS WEBRTC GATEWAY

In this section we show the results of the testing campaign we conducted against Janus. Our objective was to assess the performance of both Janus and Jattack itself. The primary parameters we took into account are the CPU and memory loads on both ends. We also leveraged the information provided by Jattack's `get-state` command in order to roughly estimate the quality perceived by the simulated users, more specifically by analyzing the number of negative acknowledgments (NACKs) sent/received on each PeerConnection. NACKs, in fact, are used to inform a sender of the loss of particular RTP packets, and thus represent a quality indicator of the media streams flowing across a PeerConnection.

A. The testbed

The testbed we set up uses Docker containers [17] in order to isolate the single functions described in the previous sections. Namely, we made use of `docker-compose` to build a microservices-oriented architecture. The testbed was deployed on a machine equipped with 8 Intel Core i7-4770S CPUs @ 3.10GHz and 16 GB of RAM. All containers are based upon the Ubuntu 16.04 OS. We also leveraged the Docker's `cpuset` option to dedicate different CPUs to different containers, so that the performance statistics of Janus were not impacted by Jattack, and vice-versa. Specifically, we assigned 4 cores to Janus, 3 cores to Jattack and 1 core to the controller.

We note that Janus, Jattack and the controller all ran on the same physical host, so all network communication happened in localhost. We did this on purpose, as we did not want network performance and congestion to affect the results of our analysis. Our aim was twofold: (i) evaluate the performance of the Jattack tool itself; (ii) spot any performance issue that strictly depends on the Janus source code, which can then be fixed/optimized.

B. Stressing the VideoRoom plugin

Fig. 2 shows the CPU evolution of both Janus and Jattack in the presence of 10 publishers and 90 subscribers. In this experiment we had Jattack generate clients with a frequency of 10 per second, starting with the publishers and then allocating viewers. As the VideoRoom plugin implements the Selective Forwarding Unit (SFU) logic, we had a total number of 1000 PeerConnnections maintained by Janus and Jattack; furthermore, as each client has to establish 10 PeerConnections when it connects to Janus, we were approximatively generating one PeerConnection per second. We observe the CPU levels are very similar to each other, as we expected since Jattack has been built upon the Janus core. Such a scenario took up to 200% of CPU on the Janus' side, and slightly more on the Jattack's side. We recall that Janus was running on four dedicated cores, while Jattack on three. Hence, Janus could take up to a value of 400% and Jattack up to 300%.

Fig. 3 shows the memory load, which proved to be very low in both cases.

Fig. 4 depicts the CPU load when there was only one publisher and 1000 viewers, i.e., 1000 PeerConnections maintained by Janus and Jattack. It is very similar to the one attained in the scenario envisaging 10 publishers and 90 viewers, as the overall number of PeerConnections is the same.

During the campaign described above, we periodically issued `get-state` requests to Jattack in order to retrieve information about the health state of each PeerConnection. Fig. 5 shows a client's perspective with respect to negative acknowledgments (NACKs) sent. We notice that no NACKs were sent/received until the overall number of PeerConnections approached 800. Then, NACKs started flowing, indicating a degradation of the quality perceived by the simulated users. One of the possible causes might be the inability for the media router to enqueue, on that specific instance, 800 packets
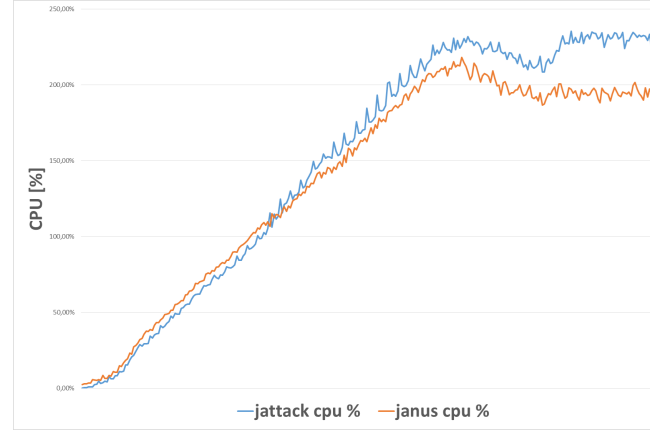

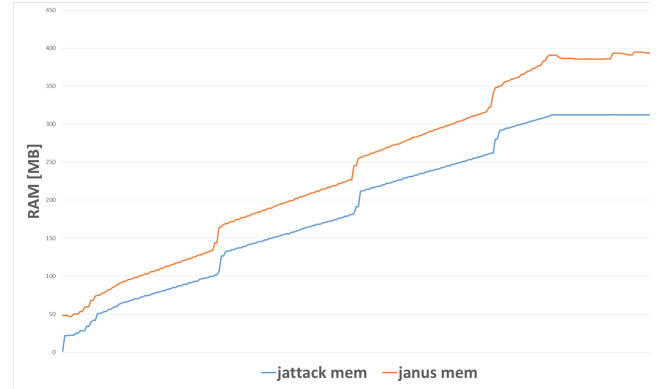Fig. 2: VideoRoom CPU with 10 publishers and 90 viewers.

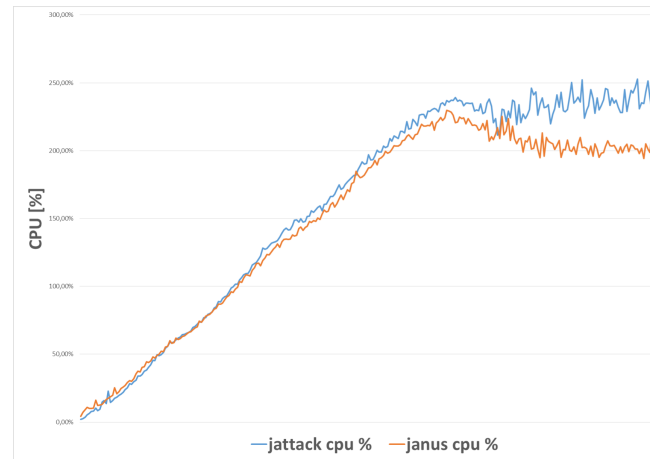Fig. 3: VideoRoom RAM with 10 publishers and 90 viewers.

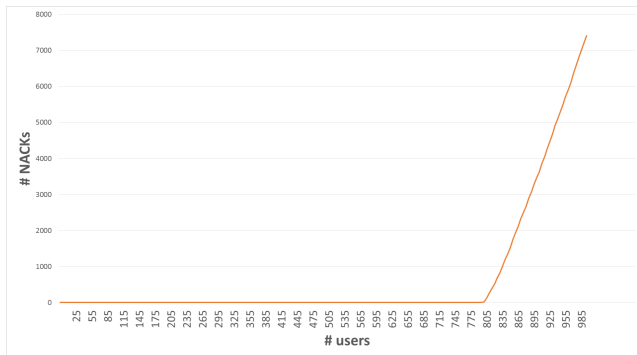Fig. 4: VideoRoom CPU with 1 publisher and 1000 viewers.

Fig. 5: Number of NACKS sent over a PeerConnection

for delivery to different recipients in the short time between two packets. Another possible cause might be ascribed to the significant number of CPU context switches: at the time of this writing, in fact, Janus spawns a dedicated thread to transmit packets to each receiver. We have already started investigating solutions to this problem. One possible approach, which we are working on in a development branch of Janus, is to delegate the one-to-many routing job within the Janus plugin to helper threads, which can then alleviate the burden a single thread has to bear right now. An alternative approach might be to leverage a pool of sender threads handling multiple recipients each.

C. Stressing the Streaming plugin

When we addressed our testing campaign towards the Streaming plugin, we obtained the very same results discussed in the previous subsection about the VideoRoom plugin when a single publisher was envisaged. So, we do not present any new chart regarding these specific experiments.

VII. CONSIDERATIONS AND FUTURE WORK

We introduced a general-purpose WebRTC stressing tool called Jattack. We explained the motivations that led us to design and implement such a tool, and also described some sample test campaigns, aimed at assessing the performance of our Janus WebRTC server. In this context, we focused our attention on both the performance of the target (Janus) and the stressing tool (Jattack), in order to evaluate the effectiveness of Jattack as a tool for performance assessment purposes.

We were delighted to find out that Jattack behaved exactly as we hoped it would, and that it was actually able to simulate the activities of multiple (both passive and active) WebRTC sessions. The greatest result was probably in terms of resources needed on the client side to implement the test campaign. While implementing a Jattack controller required some more complexity in terms of managing signaling and out-of-band communication with the target, when compared to implementing a Selenium controller (where, as anticipated, you typically can re-use existing web pages and JavaScript code), in order to set up 1000 PeerConnections we needed more than 10 servers when using Selenium. We only needed a single one to setup the same number of PeerConnections on

Jattack, instead, and it was not even the maximum number of sessions it could create on the server we hosted it on. This whole order of magnitude can prove extremely helpful when it is time to scale up the number of tests and start simulating hundreds of thousands of users, as in this case you can leverage multiple Jattack instances, each of which can support a very high number of sessions.

That said, there is room for improvement. In fact, as anticipated, we currently only tested Jattack to test the Janus server. Considering they share most of the same code-base in terms of WebRTC stack, it might be actually more interesting to start using it in different contexts as well. This includes using it both in purely peer-to-peer scenarios (i.e., to browsers directly) and to interact with third-party components, as a tool to help assess the performance of those as well.

Another interesting use case for Jattack is actually not related to stress testing, but to WebRTC clients implementation in general. In fact, the configurable nature of Jattack and the fact that it can relay plain RTP media via WebRTC on behalf of a programmable controller means it can also be used as a simple, while effective, client-side WebRTC gateway.

REFERENCES

[1] A. Amirante, T. Castaldi, L. Miniero and S.P. Romano, *Janus, a general purpose WebRTC gateway*, Proceedings of the Conference on Principles, Systems and Applications of IP Telecommunications, IPTComm '14, 2014

[2] J. Rosenberg, H. Schulzrinne, G. Camarillo et al. SIP: Session Initiation Protocol. RFC 3261, RFC Editor, June 2002. URL http://tools.ietf.org/html/rfc3261.

[3] Faraz Khan, *Calls: Is it you or is it me?*. URL https://slack.engineering/calls-is-it-you-or-is-it-me-f5d36749e8ed.

[4] SeleniumHQ web site, http://www.seleniumhq.org/

[5] A. Amirante, T. Castaldi, L. Miniero and S.P. Romano, *Performance Analysis of the Janus WebRTC Gateway*, Proceedings of the 1st Workshop on All-Web Real-Time Systems, AWeS '15, 2015

[6] C. C. Spoiala, A. Calinciuc, C. O. Turcu and C. Filote, *Performance comparison of a WebRTC server on Docker versus virtual machine*, 2016 International Conference on Development and Application Systems (DAS), Suceava, Romania, 2016, pp. 295-298.

[7] Kurento Media Server web site, http://www.kurento.org/

[8] TestRTC web site, http://www.testrtc.com

[9] Alexey Melnikov and Ian Fette, The WebSocket Protocol, RFC 6455, RFC Editor, Oct. 2015. URL https://rfc-editor.org/rfc/rfc6455.txt.

[10] Jitsi Hammer project page, https://github.com/jitsi/jitsi-hammer

[11] Grozev, Boris and Marinov, Lyubomir and Singh, Varun and Ivov, Emil, Last N: Relevance-based Selectivity for Forwarding Video in Multimedia Conferences, Proceedings of the 25th ACM Workshop on Network and Operating Systems Support for Digital Audio and Video, NOSSDAV '15, 2015.

[12] Valid8 WebRTC load tester web site, https://valid8.com/WebRTC_Load_Tester.html

[13] Load Multiplier web site, https://loadmultiplier.com/

[14] Justin Uberti and Cullen Jennings and Eric Rescorla, Javascript Session Establishment Protocol, draft-ietf-rtcweb-jsep-15, July 2016 (work in progress). URL https://tools.ietf.org/html/draft-ietf-rtcweb-jsep-15.

[15] J. Rosenberg and H. Schulzrinne. An Offer/Answer Model with the Session Description Protocol (SDP). RFC 3264, RFC Editor, June 2002. URL http://tools.ietf.org/html/rfc3264.

[16] E. Ivov, E. Rescorla, J. Uberti, P. Saint-Andre. Trickle ICE: Incremental Provisioning of Candidates for the Interactive Connectivity Establishment (ICE) Protocol draft-ietf-ice-trickle-03, July 2016 (work in progress). URL https://tools.ietf.org/html/draft-ietf-ice-trickle-03

[17] Docker web site, http://www.docker.com