Performance analysis of the Janus WebRTC gateway

Alessandro Amirante Tobia Castaldi
Lorenzo Miniero
Meetecho s.r.l.
Via C. Poerio 89/a, Napoli, Italy
{alex, tobia, lorenzo}@meetecho.com

Simon Pietro Romano
University of Napoli Federico II
Computer Engineering Department
Via Claudio 21, Napoli, Italy
spromano@unina.it

Abstract

This paper takes an in-depth look at the performance of the Janus WebRTC gateway. Janus is a modular, open-source gateway allowing WebRTC clients to seamlessly interact with legacy real-time communication technologies, both standard and proprietary, and with each other. This is achieved by attaching technology-specific plugins on top of a barebones core implementing all of the functions and protocols mandated by the RTCWEB/WebRTC specification suites. The paper focuses on assessing the scalability of the Janus architecture, by selecting three representative use cases, followed by a detailed analysis of a real-world scenario associated with multi-point audio conferencing.

Categories and Subject Descriptors H.4.3 [*Communications Applications*]: Computer conferencing, teleconferencing, and videoconferencing

Keywords WebRTC, RTCWEB, gateway, MCU, SFU, SIP, performance.

1. Introduction

Web Real-Time Communication (WebRTC) is a new standard that lets browsers communicate in real time using a peer-to-peer architecture. It is about secure, consent-based, audio/video (and data) peer-to-peer communication between HTML5 browsers. This is a disruptive evolution in the web applications world, since it enables, for the very first time, web developers to build real-time multimedia applications with no need for proprietary plug-ins.

The most general WebRTC architectural model draws its inspiration from the so-called SIP (Session Initiation Protocol) Trapezoid [8]. In the WebRTC Trapezoid model, both browsers are running a web application, which is downloaded from a different web server. Signaling messages are used to set up and terminate communications. They are transported by the HTTP or WebSocket protocol via web servers that can modify, translate, or manage them as needed. It is worth noting that the signaling between browser and server is not standardized in WebRTC, as it is considered to be part of the application. As to the data path, a PeerConnection allows media to flow directly between browsers without any intervening servers. The two web servers can communicate using a standard signaling protocol such as SIP or Jingle [5]. Otherwise, they can use a proprietary signaling protocol. For what concerns negotiation, the Session Description Protocol (SDP) [7] is mandated for all compliant WebRTC implementations.

The above mentioned scenario is the most straightforward one, since it envisages the presence of a browser talking directly to another browser. Indeed, the WebRTC APIs are designed around the one-to-one communication scenario, which represents the easiest to manage and deploy. In such a scenario, the built-in audio and video engines of the browser are responsible for optimizing the delivery of the media streams by adapting them to match the available bandwidth and to fit the current network conditions. There are other situations, though, where the one-to-one approach is not at all the best option. As an example, in a WebRTC conferencing scenario (or N-way call), each browser has to receive and handle the media streams generated by the other N-1 browsers, as well as deliver its own generated media streams to N-1 browsers. In such a case, in fact, the application-level topology is a mesh network. While this is a quite straightforward scenario, it is nonetheless difficult to manage for a browser and at the same time calls for increased network bandwidth availability. For these reasons, video conferencing systems usually rely upon a star topology where each peer connects to a dedicated server that is simultaneously responsible for:

- negotiating parameters with every other peer in the network;
- controlling conferencing resources;
- multiplexing (or mixing) the individual streams;
- distributing the proper (mix of) streams to each and every peer participating in the conference.

The WebRTC API does not provide any particular mechanism to assist the conferencing scenario. The criteria and process to identify the MCU are delegated to the application. However, this is a big engineering challenge because it envisages the introduction of a centralized infrastructure in the WebRTC peer-to-peer communication model. The upside of such a challenge clearly resides in the consideration that being capable of establishing a PeerConnection with a proxy server adds the additional services provided by the proxy server itself to the benefits offered by WebRTC. Among such additional services, we cite the possibility of letting the proxy act as a bridge towards 'legacy' technologies (e.g., SIP, Flash, etc.). In such a case, the proxy acts as a gateway allowing for the seamless interaction among WebRTC clients on one side and non WebRTC-enabled devices on the other.

The authors of this paper have recently worked on a general purpose WebRTC gateway called *Janus* [1], whose main objective is to make available a highly flexible architecture for the implementation of scenarios like those illustrated above. We herein present the results of a thorough experimental campaign aimed at assessing the performance attainable by the gateway, in a number of different configurations under varying load conditions.

AWeS '15, April 21, 2015, Bordeaux, France

http://dx.doi.org/10.1145/2749215.2749223

The paper is organized in five sections. Sec. 2 presents an overview of the Janus modular architecture, composed of a WebRTC core and a set of configurable plugins. In Sec. 3 we delve into the details of the aforementioned experimental campaign, by focusing on some of the most interesting plugins and hence analyzing a real-world audio-conferencing use case. A discussion about related work in the scientific research community are provided in Sec. 4. Sec. 5 concludes the paper and summarizes the most relevant directions of our future work.

2. Janus

The Janus WebRTC Gateway has been conceived as a general purpose gateway. As such, it does not provide any functionality per se other than implementing the means to set up a WebRTC media communication with a browser, exchanging JSON (JavaScript Object Notation) messages with it, and relaying RTP/RTCP messages between browsers and the server-side application logic they are attached to. Any specific feature/application needs to be implemented in server-side plugins, that browsers can then contact via the gateway to take advantage of the functionality they provide. At the time of this writing, eight different plugins have been implemented and publicly released as part of the project:

1. an Echo Test plugin, that simply bounces back whatever it is sent, together with some simple application logic to control the media (i.e., enable/disable audio or video, limit the bandwidth);
2. a Video Call plugin, that allows two peers to interact with each other with media relayed through the gateway, with the same knobs provided by the Echo Test plugin to have control over the media transfer;
3. a Streaming plugin, that provides means for creating on-demand and live WebRTC streams out of local files or media provided by third-party tools (e.g., *FFmpeg* or *GStreamer*);
4. a SIP Gateway plugin, that allows for a simple interaction with existing SIP infrastructures, hiding most of the complexity associated with them;
5. an Audio Bridge plugin, that allows multiple WebRTC participants to join an Opus-based mixed audio room, and have the conference call recorded;
6. a Video MCU plugin, which provides a configurable MCU to allow for any collaboration scenario ranging from webinars (one-to-many) to video conferences (many-to-many) in a controlled way (e.g., bandwidth limitations);
7. a Voice Mail plugin, a simple audio recorder module that records media provided by a WebRTC peer and stores it into a '.opus' file that can be replayed later.
8. a Record & Play plugin, which implements the ability to record an audio/video message via WebRTC and subsequently replay it within the context of a new WebRTC PeerConnection; these recordings can then be post-processed to a WebM and/or Opus file for fruition in external tools.

Janus, which has been released as open-source software[1], has been implemented using the C programming language, since we wanted something with a small footprint and that we could only equip with what was really needed (hence pluggable modules). This allows us to deploy either a full-fledged WebRTC gateway on the cloud, or a small 'nettop/box' to handle a specific use case.

An overview of the Janus architecture is depicted in Fig. 1. Herein, we do not provide further details on it. The interested reader may refer to [1].

[1] https://github.com/meetecho/janus-gateway

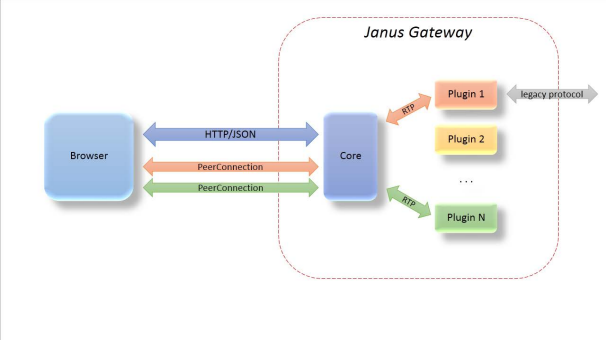

Fig. 1: Janus modular architecture

In the following subsections, we introduce the Janus plugins that have been the target of the performance analysis we conducted, which is in turn presented in Sec. 3.

2.1 The *videoroom* plugin

This is a plugin implementing a videoconferencing *Selective Forwarding Unit* (SFU) for Janus. An SFU is capable of receiving multiple media streams and then decide which of these media streams should be sent to which participants. The plugin implements a virtual conferencing room peers can join and leave at any time. This room is based on a Publish/Subscribe pattern. Each peer can publish her/his own live audio/video feeds: each feed becomes an available stream in the room the other participants can attach to. This means that this plugin allows the realization of several different scenarios, ranging from a simple webinar (one speaker, several listeners) to a fully meshed video conference (each peer sending and receiving to and from all the others).

For what concerns the subscriber side, there are two different ways to attach to a publisher's feed: a generic 'listener', which can attach to a single feed, and a more complex 'Multiplexed listener', which instead can attach to more feeds using the same PeerConnection. The generic 'listener' is the default, which means that if you want to watch more feeds at the same time, you will need to create multiple 'listeners' to attach to any of them. The 'Multiplexed listener', instead, is a more complex alternative that exploits the so called RTCWEB 'Plan B' [9], which multiplexes more streams on a single PeerConnection as well as in the SDP. Another alternative, not yet implemented anywhere, is the RTCWEB 'Unified Plan' [6], which tries to address the same requirement in a different way. While more efficient in terms of resources, these approaches are still experimental, and currently only available in its 'Plan B' form on Google Chrome. At the time of this writing, work on Plan B is ongoing, and as such its support in Janus is still flaky.

2.2 The *videocall* plugin

This is a simple video call plugin for Janus, allowing two WebRTC peers to call each other through the gateway. The idea was to provide a similar service as the well known *AppRTC* demo[2], but with the media flowing through the gateway rather than being peer-to-peer. Such an approach is of paramount importance whenever the gateway needs to have access to the media for any reason (e.g., session recording or bandwidth shaping).

The plugin provides a simple "fake" registration mechanism. A peer attaching to the plugin needs to specify a username, which acts as a 'phone number': if the username is free, it is associated with the peer, which means she/he can be called using that username by

[2] https://appprtc.appspot.com

another peer. Peers can either call another peer, by specifying their username, or wait for a call. All frames (RTP/RTCP) coming from one peer are relayed to the other.

2.3 The *audiobridge* plugin

This is a plugin implementing an audio conference bridge for Janus, specifically mixing Opus streams. The plugin implements a Multipoint Control Unit (MCU) with audio mixing functionality. This means that it replies by providing in the SDP only support for Opus, and disabling video. Opus encoding and decoding functions are implemented using libopus[3]. The plugin provides an API to allow peers to join and leave conference rooms. Peers can then mute/unmute themselves by sending specific messages to the plugin: every time a peer mutes/unmutes, an event is delivered to the other participants, so that it can be rendered in the UI accordingly.

2.4 The *SIP* plugin

This is a simple SIP plugin for Janus, allowing WebRTC peers to register at a SIP server (e.g., Asterisk) and call, or be called by, SIP user agents through the gateway. This plugin implements the gateway logic, as it lets the SIP and WebRTC worlds interoperate. Specifically, when attaching to the plugin, peers are requested to provide their SIP information, i.e., the address of the SIP server and their username/secret. This results in the plugin registering at the SIP server and acting as a SIP client on behalf of the web peer. Most of the SIP states and lifetime are masked by the plugin, and only the relevant events (e.g., INVITEs and BYEs) and functionality (call, hangup) are made available to the web peer: peers can call URIs at the SIP server or wait for incoming INVITEs, and during a call they can send DTMF tones.

The concept behind this plugin is to allow different web pages associated with the same peer (and hence the same SIP user) to attach to the plugin at the same time and yet just do a SIP REGISTER once. The same should apply for calls: while an incoming call would be notified to all the web UIs associated with the peer, only one would be able to pick up and answer, in pretty much the same way as SIP forking works but without the need to fork in the same place. This specific functionality, though, has not been implemented as of yet.

2.5 The *streaming* plugin

This is a streaming plugin for Janus, allowing WebRTC peers to watch/listen to pre-recorded files or media generated by another tool. Specifically, the plugin currently supports three different types of streams:

1. on-demand streaming of pre-recorded media files (different streaming context for each peer);
2. live streaming of pre-recorded media files (shared streaming context for all peers attached to the stream);
3. live streaming of media generated by another tool (shared streaming context for all peers attached to the stream).

For what concerns types 1. and 2., the only pre-recorded media files that the plugin currently supports are raw PCM mu-law and a-law files: support for other additional widespread formats will be added in the future. For what concerns type 3., instead, the plugin is configured to listen on a couple of ports for RTP: this means that the plugin is implemented to receive RTP on those ports and relay them to all peers attached to that stream. Any tool that can generate audio/video RTP streams and specify a destination is good for the purpose, e.g., *GStreamer*[4], *FFmpeg*[5], *LibAV*[6] or others. This makes it really easy to capture and encode whatever desired using one's own favorite tool, and then have it transparently broadcast via WebRTC using Janus.

[3] http://opus-codec.org

3. Performance evaluation

In this section we present the results of a thorough testing campaign we conducted in order to assess Janus performance. The experiments we ran were focused on server-side CPU, memory and bandwidth consumption. The testbed we set-up envisages on the server side a single Janus instance (v0.0.8) running on a machine equipped with 16 Intel Xeon E5-2640 v2 @ 2.00GHz CPU cores, 32 GB of RAM, and hosting an Ubuntu 12.04.5 LTS operating system. On the client side, we leveraged the Selenium 2.0 framework[7] in order to simulate the browser's behavior: a machine acting as 'grid master' dispatched browser allocation requests to a number of registered hosts, each capable to run browser instances (see Fig. 2). Such hosts were all Linux PCs with 8 Intel Core i7-4770S @ 3.10GHz CPU cores and 16 GB of RAM. Selenium allows to launch and remotely control any browser through the appropriate "webdriver". We chose to rely on the latest stable version of Firefox (35.0.1) since, despite Chrome, it does not implement audio-video BUNDLE techniques [3]: this means that audio and video streams do not share the same ICE component[8], thus doubling the network resources Janus has to use when both audio and video are negotiated as part of a session. In such a way, we put ourselves in the "worst case" with respect to the server-side resources. In our tests, we used "fake" media devices for both audio and video by passing the proper flag to the `getUserMedia()` constructor: this allowed us to make the simulations much easier and lighter to handle on the clients side, while still resulting in actual audio and video streams being sent and received.

All tests have been conducted over a dedicated gigabit LAN.

The following subsections present the performance figures devised by exploiting three of the plugins mentioned before, namely *videoroom* (see Sec. 3.1), *audiobridge* (see Sec. 3.2), and *SIP* (see Sec. 3.3). We do not provide detailed results related to the *streaming* plugin, since streaming can be seen as a particular SIP scenario characterized by one-way media flows. The same applies to video calls (i.e., the *videocall* plugin), which can be just considered as two-participants video rooms when looking at performance. We studied the system behavior from two different angles: (i) by performing a "stress test", i.e., letting an ever-increasing number of participants join the room; (ii) by reproducing a real-world scenario, i.e., an audio multi-point conference call with 20 participants. In the former case, we analyzed each plugin performance individually, while in the latter we used collected data to compare different possible approaches to provide the same service (Sec. 3.5).

3.1 Testing the *videoroom* plugin

In this subsection we present the performance figures derived from stress testing of the Janus videoroom plugin. Two different roles are envisaged: *subscribers* only receive remote streams, while *publishers* both send and receive Opus-encoded audio and VP8-encoded video. During our tests, we first made 10 publishers join the videoroom, then we let 140 subscribers join as well. As already an-

[4] http://gstreamer.freedesktop.org/

[5] http://www.ffmpeg.org/

[6] http://libav.org/

[7] http://www.seleniumhq.com

[8] *ICE Component*: A component is a piece of a media stream requiring a single transport address; a media stream may require multiple components, each of which has to work for the media stream as a whole to work.

Fig. 2: Selenium grid

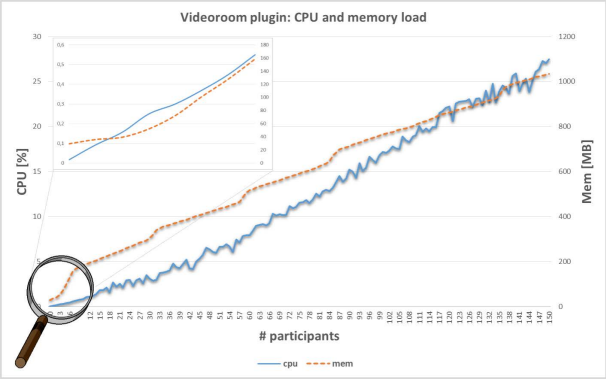

Fig. 3: Videoroom plugin: CPU and memory

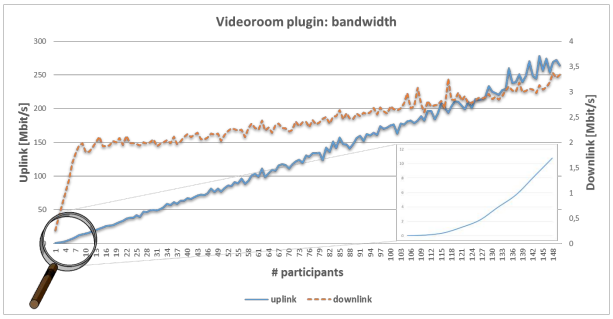

Fig. 4: Videoroom plugin: bandwidth

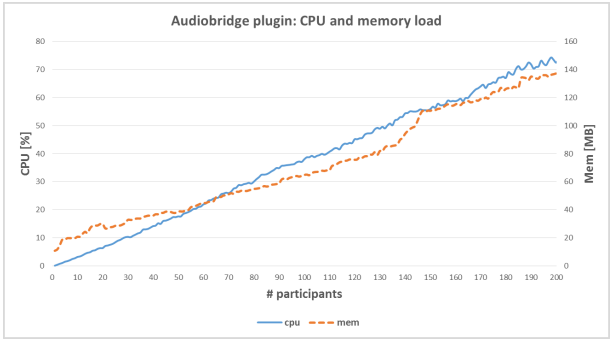

Fig. 5: Audiobridge plugin: CPU and memory

ticipated, the videoroom plugin implements the SFU logic, hence the 150 participants envisaged by this scenario correspond to 1500 PeerConnnections maintained by Janus. Fig. 3 shows the CPU utilization level and memory occupation in such a scenario. We notice how memory load increases quadratically with the number of publishers (participants 1 to 10). This was expected since every time a new publisher joins, Janus creates a new PeerConnection per each pre-existing participant. Then, it increases in a roughly linear manner with the number of subscribers. We notice periodic "steps" in the memory evolution, which we are still in the process of investigating in further detail through fine-grained profiling techniques. Intuitively, we believe such a phenomenon can be ascribed to the intensive use of dynamic memory allocation both in the core of Janus and in most of its plugins. With this type of memory management mechanisms, the system typically reserves memory slots in the heap and starts gradually filling them up. As soon as it runs short on free memory, it makes a new reservation. The CPU load, instead, does not follow this pattern and always keeps the same growth trend.

Fig. 4 plots the bandwidth usage. Each publisher sends both Opus audio and a 640x480 VP8 video, generating around 180kbit/s towards the server. Hence, the downlink traffic generated by publishers grows linearly to 1.8Mbit/s. This value slightly increases toward 3Mbit/s as more and more subscribers join, accounting for signalling traffic, RTCP feedback and possible retransmissions on each PeerConnection. On the other hand, uplink traffic has a quadratic evolution with the number of publishers, for the same reasons already discussed for memory load. Then, it increases almost linearly with the number of subscribers.

3.2 Testing the *audiobridge* plugin

When we aimed our stress tests at the audiobridge plugin, we found out that mixing and transcoding media flows led the CPU load to increase in a roughly linear way with the number of flows to be mixed. Such operations are quite demanding in terms of CPU resources: 200 participants in a wideband (16kHz) audio mix took up around 73% of CPU as depicted in Fig. 5. Memory occupation, instead, kept relatively small, as it did not exceed 150 MB.

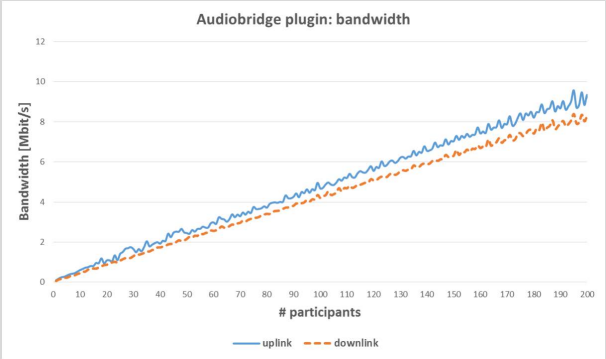

Fig. 6: Audiobridge plugins: bandwidth

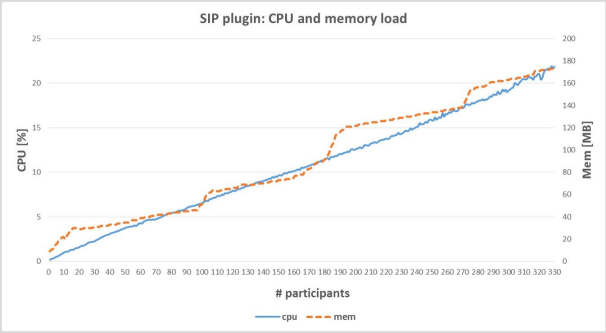

Fig. 7: SIP plugin: CPU and memory

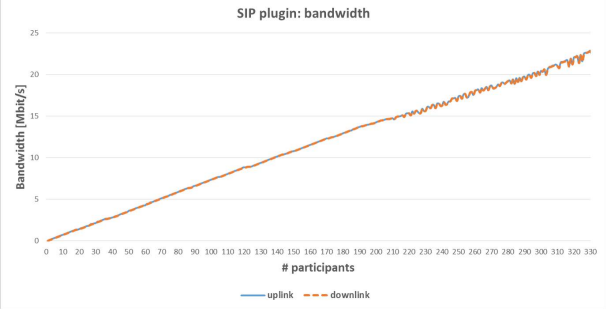

Fig. 8: SIP plugin: bandwidth

Finally, Fig. 6 shows how uplink and downlink traffic followed the same trend, as expected.

3.3 Testing the *SIP* plugin

This subsection shows the performance attained when we put under test the SIP plugin. In this case, each participant joining made Janus generate a SIP dialog with an external server, namely an Asterisk PBX and its echo-test application. As Fig. 7 shows, the CPU level increases linearly. 330 participants took up around 22% of the CPU. The memory consumption, instead, presented the same behavior already discussed in Sec. 3.1.

For what concerns bandwidth, uplink and downlink levels are exactly the same, as expected (see Fig. 8).

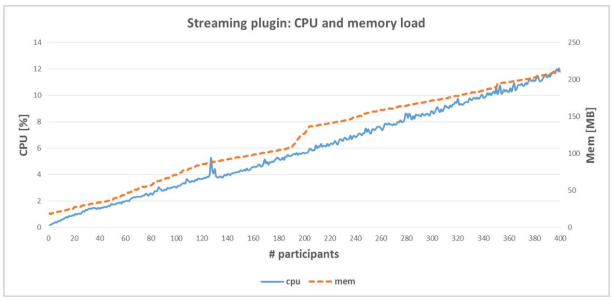

Fig. 9: Streaming plugin: CPU and memory

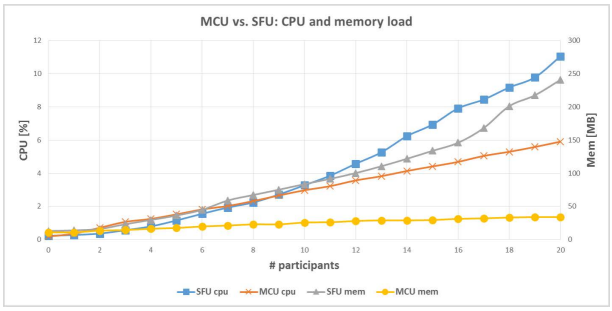

Fig. 10: MCU vs. SFU: CPU and memory

3.4 Testing the *streaming* plugin

As already anticipated, the streaming scenario can be seen as a particular SIP scenario characterized by one-way media flows. As such, the same considerations apply. CPU and memory evolutions are depicted in Fig. 9.

3.5 A real-world scenario: multi-point audio conference

In this subsection, we take a sample real-world scenario, namely a multi-point audio conference involving 20 participants. Such scenario may be realized by exploiting either the videoroom, or the audiobridge or the SIP plugin, with different performance attained as we show in the following.

3.5.1 MCU vs. SFU

In this first comparative example, we implemented the aforementioned audio conference service by leveraging the videoroom and audiobridge plugins. In the former case, of course, we disabled video functionality to obtain an audio-only SFU. Fig. 10 shows CPU and memory loads of the two approaches. CPU evolution over time is also depicted in Fig. 11, while bandwidth consumption is shown in Fig. 12.

The results demonstrate that, using a wideband (16kHz) mixer in the audiobridge, MCU wins over SFU whenever the number of participants goes beyond 8, as it requires less CPU, memory, and bandwidth. This seems to contradict the general belief that mixing flows requires more resources than just forwarding them: wideband Opus audio mixing proved to be a lightweight operation that can be easily performed without taking too much CPU cycles. It is worth remarking that the results provided do not take into account video flows. Video mixing, in fact, is a heavy task, which may lead to completely different outcomes.

3.5.2 Local vs. remote mixing

In this subsection we compare local and remote mixing approaches. We refer to *local* as the mixing functionality provided by Janus

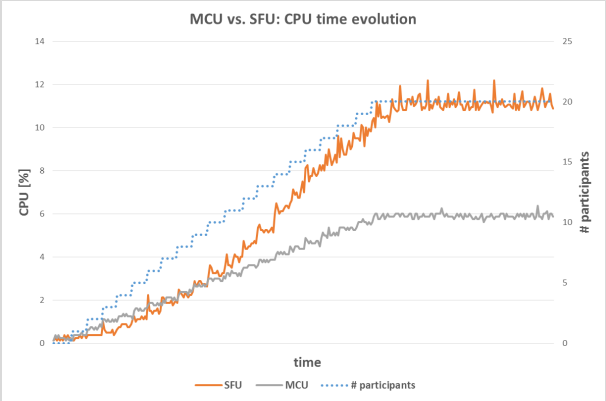

Fig. 11: MCU vs. SFU: CPU time evolution

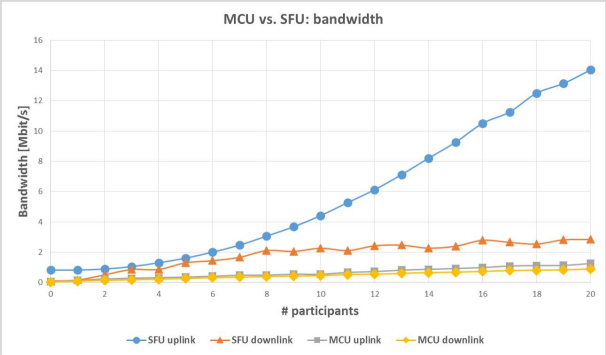

Fig. 12: MCU vs. SFU: bandwidth

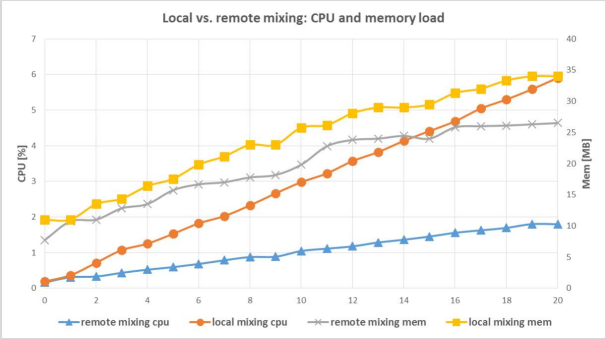

Fig. 13: Local vs. remote mixing: CPU and memory

itself, while *remote* are mixing features provided by an external component. To implement such scenarios, in the former case, we leveraged the Janus audiobridge plugin; in the latter, we exploited the SIP plugin which in turn forwarded flows to an Asterisk server taking care of mixing through its *ConfBridge* application.

As clearly stated by Fig. 13, remote mixing is definitely preferable when looking at CPU and memory consumption as key performance indicators. On the other hand, whenever bandwidth is considered more critical (e.g., on Amazon AWS instances), local mixing may be the best choice, as shown in Fig. 14.

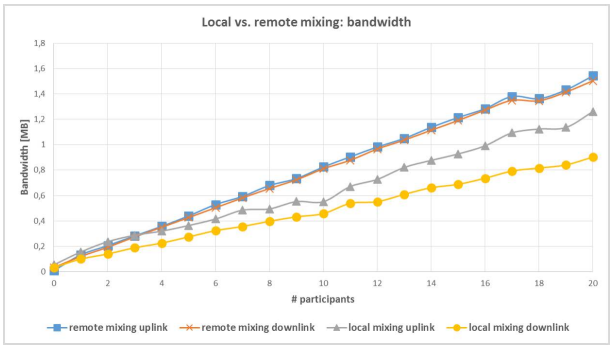

Fig. 14: Local vs. remote mixing: bandwidth

4. Related work

Performance assessment of server-side WebRTC code is gaining more and more interest. A discussion on this has recently been started on the *discuss-webrtc* group[9] on Google Groups, the main discussion forum on the WebRTC space.

T. Levent-Levi provides in [4] a theoretical analysis of different approaches to WebRTC server-side media processing, even though no experimental data is presented. He also proposes the two definitions of MCU and SFU we adopted in this work.

B. Grozev and E. Ivov conducted in [2] a thorough performance evaluation of the Jitsi Videobridge, an open source WebRTC SFU, by also making available experimental data. The performance attained seems to be close to the figures we presented in Sec. 3.1: they claim to support 10 publishers with 3.1% CPU usage, while in our tests the same number of publishers took less than 1% of CPU. On the other hand, in their scenario each publisher sent 515kbit/s toward the server, against the 180kbit/s per publisher we sent in our experiments.

Several other implementations of WebRTC server-side components are currently available, many of which come in the form of open source code like Janus. We mention *webrtc2sip*[10], a WebRTC-to-SIP gateway, *Licode*[11], a WebRTC SFU, *Medooze*[12], a WebRTC-enabled MCU and media server, *Kurento*[13], a media server with WebRTC support. To the best of our knowledge, no performance analysis of these software is currently available.

5. Conclusion and Future Work

In this paper we presented the results of a wide-range experimental campaign aimed at assessing the key performance indicators of the Janus WebRTC gateway. The tests we conducted have indeed allowed us to reach a twofold goal. First, during the first phases of testing, we have seized the opportunity to leverage test results in order to spot out latent issues hidden in our software and properly fix them. This has allowed us to arrive at a higher maturity level of the code our system's architecture relies upon. Subsequent tests have instead represented the ideal means for us to properly benchmark some of the most typical configurations of Janus and its plugins along the three main directions associated, respectively, with memory, CPU and bandwidth utilization. The mentioned parameters are indeed of interest if one is willing to make an informed decision

[9] https://groups.google.com/forum/#!forum/discuss-webrtc

[10] http://webrtc2sip.org/

[11] http://lynckia.com/licode/

[12] http://www.medooze.com/

[13] http://www.kurento.org/

on how to optimally configure one's own system in a real-world deployment scenario.

The results look encouraging, since they clearly demonstrate how the current structure and organization of the Janus architecture allow users to implement a variegated set of advanced services in a scalable fashion. In the next few months it is our intention to push testing one step further by both extending measurements to the set of plugins we have not yet analyzed in detail and digging further into the details of some of the phenomena for which we have not come out with a clear explanation as of yet. With respect to this last point, as already anticipated we are interested in investigating, through fine-grained memory profiling techniques, the exact behavior of our system for what concerns the optimized management of dynamic memory slots.

Acknowledgments

This work was partially funded by the Italian Ministry of Education, University and Research (MIUR) within the framework of projects PON01 01007 "PLATform for INnOative services in future internet" (PLATINO) and PON04a2_C ("SMART HEALTH").

References

[1] A. Amirante, T. Castaldi, L. Miniero, and S. P. Romano. Janus: a general purpose WebRTC gateway. In *Proceedings of the 7th International Conference on Principles, Systems and Applications of IP Telecommunications (IPTComm), Chicago, USA*, October 2014.

[2] B. Grozev and E. Ivov. Jitsi Videobridge Performance Evaluation. https://jitsi.org/Projects/JitsiVideobridgePerformance.

[3] C. Holmberg, H. Halvestrand, and C. Jennings. Negotiating Media Multiplexing Using the Session Description Protocol (SDP). Internet Draft, January 2015.

[4] T. Levent-Levi. Seven Reasons for WebRTC Server-Side Media Processing. Technical report, 2015.

[5] S. Ludwig, J. Beda, and P. Saint-Andre. XEP-0166: Jingle. Technical report, XMPP Standards Foundation, December 2009. URL `http://xmpp.org/extensions/xep-0166.html`.

[6] A. Roach, J. Uberti, and M. Thomson. A Unified Plan for Using SDP with Large Numbers of Media Flows. Internet-Draft draft-roach-mmusic-unified-plan-00, IETF Secretariat, July 2013.

[7] 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`.

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

[9] J. Uberti. Plan B: a proposal for signaling multiple media sources in WebRTC. Internet-Draft draft-uberti-rtcweb-plan-00, IETF Secretariat, May 2013.