Janus: a general purpose WebRTC gateway

A. Amirante, T. Castaldi, L. Miniero, S. P. Romano
University of Napoli Federico II, Napoli, Italy
and
Meetecho s.r.l., Napoli, Italy
{alex, tcastaldi, lorenzo}@meetecho.com, spromano@unina.it

ABSTRACT

This paper deals with the design and implementation of Janus, a general purpose, open source WebRTC gateway. Details will be provided on the architectural choices we took for Janus, as well as on the APIs we made available to extend and make use of it. Examples of how the gateway can be used for complex WebRTC applications are presented, together with some experimental results we collected during the development process.

Keywords

Gateways, IETF, Rtcweb, WebRTC, Web communication, JavaScript, Conferencing, Streaming

1. INTRODUCTION

Web Real-Time Communication (WebRTC) is a new standard and industry effort that extends the web browsing model. For the first time, browsers are able to directly exchange real-time media with other browsers in a peer-to-peer fashion. The World Wide Web Consortium (W3C [11]) and the Internet Engineering Task Force (IETF [8]) are jointly defining the JavaScript APIs (Application Programming Interfaces), the standard HTML5 tags, and the underlying communication protocols for the setup and management of a reliable communication channel between any pair of next-generation web browsers. The standardization goal is to define a WebRTC API that enables a web application running on any device, through secure access to the input peripherals (such as webcams and microphones), to exchange real-time media and data with a remote party in a peer-to-peer fashion. The most general WebRTC architectural model draws its inspiration from the so-called SIP (Session Initiation Protocol) Trapezoid [23]. In the WebRTC Trapezoid model (Fig. 1), 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

IPTComm '14 September 30 - October 2, 2014, Chicago, IL, USA

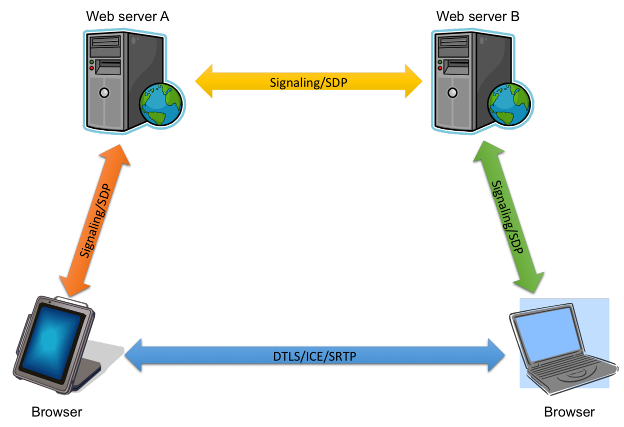


Figure 1: The WebRTC Trapezoid

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. It represents an association with a remote peer, which is usually another instance of the same JavaScript application running at the remote end. Communications are coordinated via a signaling channel provided by scripting code in the page via the web server, e.g., using XMLHttpRequest or WebSocket. Once a peer connection is established, media streams can be sent directly to the remote browser. The two web servers can communicate using a standard signaling protocol such as SIP or Jingle [17]. Otherwise, they can use a proprietary signaling protocol.

Besides the basic scenario associated with direct browser-to-browser communication, there are several alternative situations where an end-user may actually be interacting with an application, rather than another user. These include talking to an IVR system, a PSTN gateway or a conferencing server. Since day one, WebRTC has attracted the attention of two different worlds: those who envisaged the chance to create innovative applications based on a new paradigm, and those who basically just envisioned a new client to legacy services and applications. In the depicted scenario, the need has soon arisen for some kind of component to be placed between two or more WebRTC peers, thus going beyond (or even breaking) the end-to-end approach WebRTC is based upon. It is important to point out that, even in a simple peer-to-peer scenario, the two involved parties do not necessarily need to be browsers, but may actually run a different application. Such an application might be acting as a Multipoint Control Unit (MCU), a media recorder, an Interactive Voice Response (IVR) system, a bridge towards a different technology (e.g., SIP, RTMP, or any legacy streaming platform) or something else. It should implement most, if not all, the WebRTC protocols and technologies, and would actually represent a WebRTC Gateway: one side talks WebRTC, while the other might be talking some entirely different protocol (e.g., translating signaling protocols and/or transcoding media packets).

In the above depicted scenario, we have recently worked on an open source WebRTC gateway called *Janus*, which has been conceived at the outset as a general purpose approach to the design and implementation of a WebRTC-enabled mediation component. In this paper we will guide the reader through the common requirements and challenges application architects have to face when designing and implementing such a complex networked system.

2. CONTEXT AND MOTIVATION

There are several reasons why a gateway can be useful. Technically speaking, MCUs and server-side stacks can be seen as gateways as well, which means that, even when you do not step outside the WebRTC world and just want to extend the one-to-one/full-mesh paradigm among peers, having such a component can definitely help depending on the scenario you have in mind. Nevertheless, the main motivation comes from the significant number of legacy infrastructures that may benefit from a WebRTC-enabled kind of access. In fact, one might assume that the re-use of existing protocols like SDP [22], RTP [24] and others in WebRTC makes this a trivial task. Unfortunately, most of the times that is not the case. If for instance we refer to existing SIP infrastructures, even if we made use of SIP as a signaling protocol in WebRTC we would nevertheless incur serious issues, due to the many differences between the standards implemented by WebRTC endpoints and those available in current deployments. Just to make a simple example, most legacy components do not support media encryption, and when they do they usually only support SDES (*Session Description Protocol Security Descriptions*). On the other hand, for security reasons WebRTC mandates the use of DTLS (*Datagram Transport Layer Security* [18]) as the only way to establish a secure media connection. DTLS has been around for a while but has witnessed limited deployment in the existing communication frameworks so far.

The same incompatibilities between the two worlds emerge in other aspects as well, like the extensive use WebRTC endpoints make of ICE [21] for NAT traversal, RTCP feedback messages for managing the status of a connection or RTP/RTCP multiplexing, whereas existing infrastructures usually rely on simpler approaches like Hosted Nat Traversal (HNT) in Session Border Controllers (SBCs), separate even/odd ports for RTP and RTCP, and more or less basic RFC3550-compliant RTCP statistics and messages. Things get even worse when we think of the additional functionality, mandatory or not, that is being added to WebRTC right now, as *BUNDLE* [14], *Trickle ICE* [15] and new codecs (that the legacy media servers will most likely not support).

Researchers and implementors have hence started to work on gateways since the first WebRTC browsers have seen the light. We briefly touch on related work in Sec. 6. The main idea behind the Janus WebRTC gateway is to make available a component that is general enough to flexibly adapt to as many situations as possible, by relying on a lightweight WebRTC core that can be properly extended/customized through dynamic injection of application-specific plugins.

3. THE JANUS WEBRTC GATEWAY

As anticipated in Sec. 2, a need is arising for components able to bridge WebRTC endpoints to legacy architectures and technologies. Starting from this assumption, we designed our WebRTC gateway as a *general purpose* component that can be exploited in a programmable way.

We called this component, which we released as open source on GitHub [2], Janus, as a homage to the well known God of the Ancient Rome pantheon. In fact, it is known that Janus had two faces, one looking at the past, and one at the future, which is why January, as a bridge between two years, was called like that. For the same reason, we found Janus to be the perfect name for our component, as it always has at least two faces: one overlooking legacy technologies (the past), the other taking care of WebRTC (the future).

Coming to the design and implementation of the gateway, we conceived it as a modular architecture, with a core responsible for everything related to WebRTC itself. Since we wanted the implementation to be as lightweight and as fast as possible, we chose C as a programming language for the actual implementation. An overview of the overall Janus architecture is presented in Sec. 3.1, while the protocols implemented by the core are discussed in Sec. 3.2. To conclude, the API exposed by the gateway is presented in Sec. 3.3.

3.1 Modular architecture

Since the beginning, we conceived the Janus architecture as modular. Specifically, we designed it as a core with a specific set of responsibilities, and pluggable modules to provide specific features, namely support for legacy technologies and protocols. This approach was motivated by our former experiences within the IETF MEDIACTRL Working Group [12] . This WG was devoted to the definition of a standard way to implement an effective communication among Application Servers handling application logic, and Media Servers enforcing the related media manipulation tasks. Communication relied on so called *control packages*, allowing the usage of a generic protocol to drive the communication between an application and one or more packages providing specific functionality in a pluggable way.

We chose to follow a similar path for Janus as well, with a core handling the high level communication with users (sessions and handles management, WebRTC-related protocols) and server-side plugins to provide specific functionality in a way that is transparent to WebRTC, and as such independent from the web application. More details on the communication part are provided in Sec. 3.3, while an overview of the architecture and related interactions is depicted in Fig. 2.

The core is mostly responsible for three things: i) managing

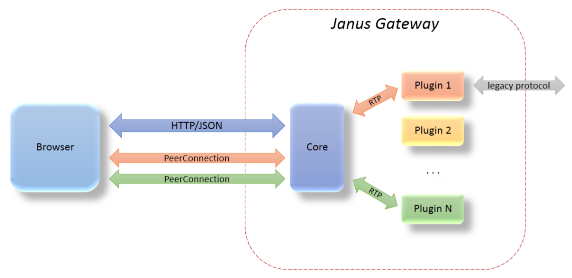


Figure 2: Janus modular architecture

application sessions with users through a REST-ful API; ii) implementing the WebRTC communication with the same users, by taking care of the whole WebRTC lifecycle (negotiation, establishment and management of PeerConnections); iii) attaching users to plugins, in order to allow them to exchange messages (based on a per-plugin ad-hoc protocol) and more importantly media (relaying plain RTP/RTCP). This allows plugins to easily communicate with WebRTC users, as most hassles associated with the WebRTC stack are masked by the gateway core. Janus plugins just need to implement the related plugin API to set up a specific session with users that want to take advantage of their features, and get prepared to receive and/or send RTP packets and related RTCP messages, in case of need. For instance, an echo test plugin will simply send back whatever it receives, while a video calling or MCU plugin would relay packets coming from one user to one or more different users. At the same time, more complex plugins involving different technologies may make use of those packets in a different way.

To showcase the pluggable nature of the modular architecture, we implemented a few simple plugins to address some typical use case scenarios: i) an Echo Test plugin; ii) a Video Call plugin, bridging two WebRTC users through the gateway; iii) a Streaming plugin, allowing to relay external sources (e.g., audio files or live RTP streams originated by a different tool) towards WebRTC users; iv) a SIP plugin, acting as a gateway between WebRTC users and an existing SIP infrastructure; v) an Audio Bridge plugin, mixing Opus streams coming from different WebRTC users, thus enabling audio conferencing scenarios; vi) a Video MCU plugin, acting as an MCU among several WebRTC users, thus enabling a number of different scenarios like video conferencing, webinars, screen sharing and so on; vii) a Voice Mail plugin, that simply records to an Opus file (and then returns back) whatever a WebRTC user says.

An online demonstration of how these plugins can be used for real-world applications is provided on the Janus online demos web page [3]. While definitely not an exhaustive list of features a gateway may provide, these simple plugins already allow for the realization of complex multimedia scenarios, especially if one looks at them like “bricks” that can be used for building more complex applications. More details about them are provided in Sec. 4, where we describe some applications that can be built using the existing set of plugins made available out of the box.

As already stated, the gateway exposes an API for creating new plugins: hopefully this will foster in the near future the publication of third-party plugins written by developers interested in exploiting the functionality of our gateway for purposes that go beyond what is currently available.

3.2 Standard protocols

While the plugins provide specific functionality, the Janus core is responsible for everything that is related to WebRTC itself. Specifically, the core makes sure that a WebRTC PeerConnection can be correctly negotiated and established with interested users, and that this PeerConnection can be used to send and receive media in a way that WebRTC users can understand. For this reason, we implemented the core so that it can handle the entire suite of protocols and technologies mandated by the WebRTC standard. Specifically, with the help of open source libraries where possible, we implemented support for the JavaScript Session Establishment Protocol (JSEP) [25], the Session Description Protocol (SDP) [22], the Real-time Transport Protocol [24] and its secure extensions [13], the Interactive Connectivity Establishment [21] and the Datagram Transport Layer Security extensions for SRTP [18], all as mandated by the latest media-related specification [20]. Besides, we also added support for new mechanisms currently under standardization, like BUNDLE [14], Trickle ICE [15] and Data Channels [16].

The core is hence responsible for the negotiation, through JSEP and SDP, of multimedia sessions with WebRTC users, as well as for the actual establishment of one or more SRTP/SRTCP multimedia channels with them by means of ICE (for NAT traversal and connectivity checks) and DTLS (to setup a secure channel). This process is the same no matter which plugin will be used afterwards to manipulate the multimedia channel itself. In fact, the core hides every aspect related to WebRTC from plugins, except for some media-related information the plugins may either require or provide (e.g., if a plugin expects a specific codec, or is interested in audio but not video). Once the multimedia channel is set up, the gateway has access to plain RTP and RTCP flows, which can be leveraged to let WebRTC users exchange media with specific plugins.

It is worth noting that, since the gateway will most of the time manipulate RTP and RTCP information (e.g., SSRCs), the core is also responsible for keeping them coherent when moving RTP packets and messages around. More information about this process is documented in a draft currently under discussion in the STRAW Working Group [19].

3.3 REST-ful API

We already clarified how the IETF and W3C chose not to mandate any signaling protocol for creating media sessions. This was a deliberate choice to allow for the maximum flexibility in creating web applications, while at the same time being strict with respect to the media negotiation and transport. While many applications, including existing gateways, chose to rely on well known protocols like SIP or XMPP for the purpose, we decided to take a different approach. In fact, considering the general purpose nature of the gateway, and the idea of fostering the usage of multiple plugin connections within the same application session, we resolved to design a REST-ful API that could be contacted by means of

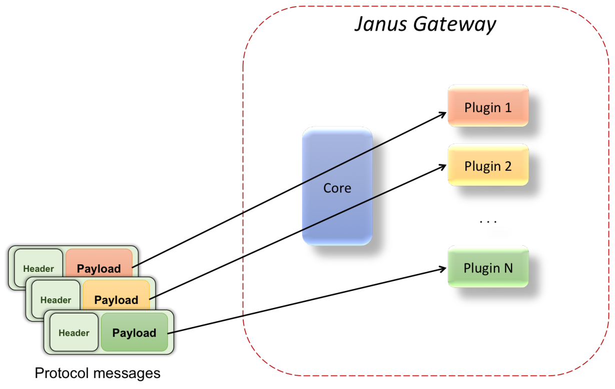


Figure 3: REST-ful API

an ad-hoc protocol based on the JSON (*JavaScript Object Notation*) format. This allowed us to dynamically create and manage gateway sessions (the generic session between a WebRTC user and the gateway, not to be confused with whatever application level session may be used in the web application itself) and plugin handles (everything related to web-to-plugin communication and WebRTC PeerConnections) using an extensible, yet simple, mechanism.

The API we conceived is based on the idea that every web page interested in exploiting the features made available by Janus opens a gateway session. Such a session creates a new endpoint on the gateway web server, which can be queried, e.g., by means of HTTP long polls, for events. Within the context of the session, a web page will hence be able to create one or more plugin *handles* (i.e., connections with specific plugins), to take advantage of their functionality. Each such handle will potentially be usable to setup a new PeerConnection whose usage will depend on the plugin it is attached to. When a plugin has been created, web applications can interact with it using a JSON-based protocol as a transport for asynchronous messages. Since each plugin will typically implement specific functionality, and might potentially be provided by different parties, the specifics of the protocol and syntax used to interact with it are opaque to the Janus JSON protocol itself. As anticipated in Sec. 3.1, this is the same approach that is used for *control packages* in MEDIACTRL, and is an effective way to provide extensibility in a transparent way.

An example of such an approach is depicted in Fig. 3. It is worth noting that we recently implemented a WebSockets interface as an alternative to REST, which uses exactly the same API. This allows for the setup of a sort of *control channel*, which is useful in scenarios where the Janus API is actually going to be wrapped on the server side by frameworks and applications.

To make things easier for web developers, we also implemented a JavaScript API that can be used as-is to integrate Janus within web applications. This API is based on a generic object that implements methods and callbacks, and that can be used with any plugin made available by the gateway instance it is going to be attached to. Having in mind the lengthy discussions within the standardization bodies with respect to whether the WebRTC API should be high level (and thus easier to use, but less flexible) or low level (more complicated, but much more flexible in terms of adaptability)[1], we chose to implement this API at a slightly lower level in terms of interaction with the gateway. In fact, different plugins will often require a customized communication syntax. This consideration motivated us to design a generic framework for such interaction. The choice we made will allow us in the future to also design and implement higher level APIs specifically conceived for the plugins we already made available. As an example, let us figure out a videoconferencing API designed on top of the Video MCU plugin and allowing web developers to only care about people joining and/or leaving rooms. Such a plugin would provide information about media becoming available, without worrying about the specifics of the interaction with the gateway itself.

Both the REST-ful API and the JavaScript API we designed to interact with it are documented on the project website (the documentation can also be built by developers themselves when installing the component). The same site also hosts live examples of their usage within the context of real-world application scenarios.

4. MULTIMEDIA APPLICATIONS

We have so far described how we designed and implemented Janus, in particular how we separated the responsibilities between its core and feature-specific pluggable modules. We also introduced some of the plugins we chose to make available out-of-the-box, and how an application can exploit them by means of an open API.

Rather than digging into the details of such an interaction, we present in this section an overview of some rich multimedia applications that can be built on top of Janus. In particular, we will describe how different plugins, sometimes with more than a single instance per user, can be used in an effective way as “bricks” to realize a much more complex application than the one each individual plugin was designed for. These examples are even more interesting when considering the fact that the features made available by Janus can be easily integrated within a more complex framework and interact with external components taking care of different functionality (e.g., an XMPP library and infrastructure to provide instant messaging associated with a multimedia session powered by Janus).

The following examples are by no means intended to be an exhaustive list of applications. They are rather meant to provide a quick overview of the flexibility made available by Janus and its default plugins (flexibility that can be further improved with third-party plugins). We encourage the reader to think of other applications that may be realized, and figure out the functions they would require from Janus in order to be effectively implemented.

4.1 Screen Sharing with Q/A

Screensharing is a typical scenario commonly deployed nowadays. Such an application proves most useful during webinar sessions, where one or few speakers do a presentation, e.g.,

[1]Discussions that eventually resolved towards the definition of JSEP as a middle ground

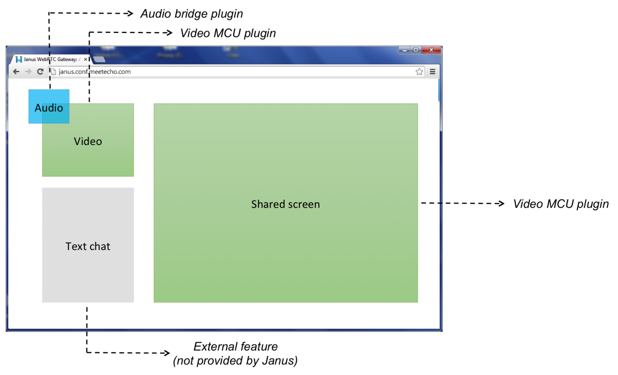


Figure 4: Screen Sharing with Q/A example

by sharing their screen and publishing their audio and video feeds. During the presentation, or immediately thereafter, a questions and answers slot involving any of the attendees can take place to start a discussion.

A simple way to approach such an application would be decomposing it into a subset of individual functionality, such as: i) a way to broadcast/relay the screen being shared by one of the speakers, ii) a way to display the video feed of the main speakers themselves, and iii) a way to allow passive attendees to chime in for a while in order to allow them to ask questions to the speakers (since, for scalability reasons, we may want to limit the level of interaction, especially if several passive attendees are expected). A simple diagram representing this situation is depicted in Fig. 4.

This functional decomposition can be easily matched to features provided by the default plugins in Janus. As to screen broadcasting, the Video MCU can be used, as it allows for an easy way to relay the contribution of a single WebRTC user (in this case, a screen being shared, a feature that Chrome already allows for) to several viewers. Similarly, the same Video MCU plugin can be used for the video feeds coming from the speakers as well: in fact, since the Video MCU plugin allows users to selectively publish and/or subscribe to a feed, a dedicated room can be used by the speakers to publish their own contributions (and get attached to other speakers's feeds, if any), while letting passive attendees just view the feeds being published. Finally, for what concerns audio, the Audio Bridge can be leveraged to let speakers present while at the same time allowing for a time-limited, audio-only interaction with the other attendees. The mentioned plugin would implement an audio mix of the involved parties, thus limiting the bandwidth being used.

It is worth noting that the audio part, including the time-limited interaction for Q/A, could be achieved by means of the same room used for the video of the speakers. Nevertheless, it is sometimes useful to delegate the audio part to a different component, e.g., in case you are interested in an audio recording of the event (that using the Audio Bridge plugin would already be mixed with no need for post-processing later on). The same consideration holds in case you are using a different infrastructure for audio and you want a WebRTC front-end towards it (e.g., you're using a SIP-based conference bridge to allow PSTN users to join

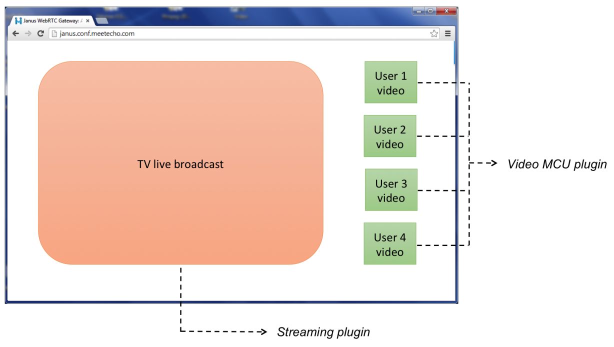


Figure 5: Social TV example

for questions). We do believe such a decomposition clearly illustrates the benefits derived from the separation of responsibilities when building a multimedia application.

4.2 Social TV

Social TV is a challenging application that several researchers and developers are trying to address. If we consider the low-delay nature of WebRTC, together with its easy integration within web applications, it is safe to assume that it definitely represents a useful building block for this kind of scenarios.

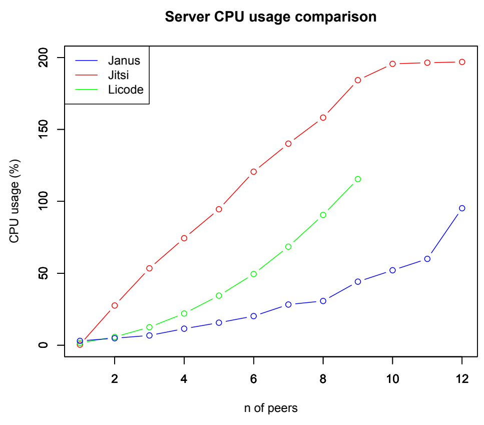
In this section we show how Janus can be effectively leveraged in order to build a social TV. Specifically, we design a web page providing a live broadcast of an event (e.g., a sport match or a TV show), while at the same time allowing a few users watching the same event to interact with each other by means of real-time audio and video.

This application may be implemented via Janus by exploiting two different plugins: i) the Streaming plugin, allowing to seamlessly relay an external source to one or more interested WebRTC users, and ii) the Video MCU plugin, which, as explained in the previous example, can be used to easily implement video rooms involving multiple WebRTC users as publishers and/or listeners. This mapping is depicted in Fig. 5.

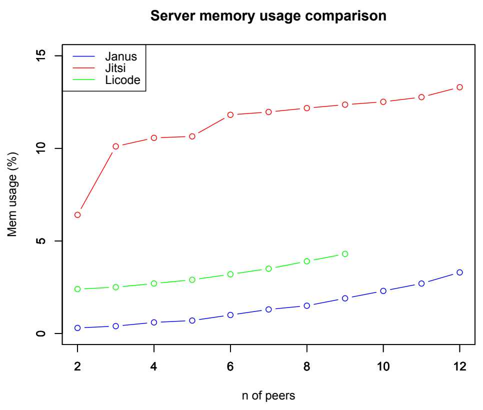
4.3 Audio/Video calls on Social Networks

This third example addresses another common scenario nowadays, that is social networks providing multimedia communication functionality to their users. Several social networks have chosen to rely on plugins for the purpose (e.g., Facebook with Skype, or Google+ with Hangout), while only few of them have switched to WebRTC in the meanwhile (among them, Tiscali's Indoona [10], an effort we recently contributed to). With WebRTC starting to gain pace, more and more such infrastructures will start relying on it for the purpose. The following is an example of how such an effort may be accomplished by means of Janus.

Let us assume this social network already has a SIP infrastructure in place, with each user also associated with a dedicated SIP identity. The Janus SIP plugin can easily do the job of letting these SIP users become WebRTC-enabled, too. In this scenario, a social network may choose to also rely on a couple more functions, i.e.: i) an Echo Test web page, to allow users to make sure they would be currently able to



(a) Server CPU comparison



(b) Server MEM comparison

Figure 6: Experimentation results: server side

setup a WebRTC multimedia session, and ii) a landing page in case the callee is not reachable, allowing us to record a voice message.

Considering the requirements, it is quite simple to identify how Janus may be used to satisfy them. We already anticipated how the SIP plugin may be exploited for such a purpose, as it would allow social network users to transparently register at the SIP infrastructure, and send/receive calls from a page in the social network. At any time, users would be able to check whether or not WebRTC works in their environment by opening a check page. The web page would just need to exploit the features made available by the Echo Test plugin, allowing for an evaluation of the WebRTC-related functionality independently of SIP signaling. Finally, for the landing page to leave voice messages the Voice Mail plugin can be leveraged, as it allows for a simple recording of media frames. It is worth noting that, while both the echo test and the voice message features may be built within the SIP infrastructure itself (e.g., by relying on a SIP PBX for the purpose), the use of a different plugin helps keep the features both separated and more easily controllable.

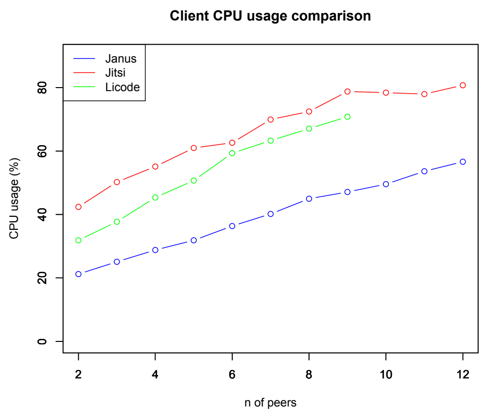
5. EXPERIMENTATION

Our desire to benchmark the performance of Janus, and its ability to properly handle as many users as possible in a scalable way, was immediately faced with a first challenge. In fact, considering its general purpose nature and the fact that different plugins may have completely different requirements and/or impact on the performance of a Janus instance, we soon realized that quantifying this would not be trivial.

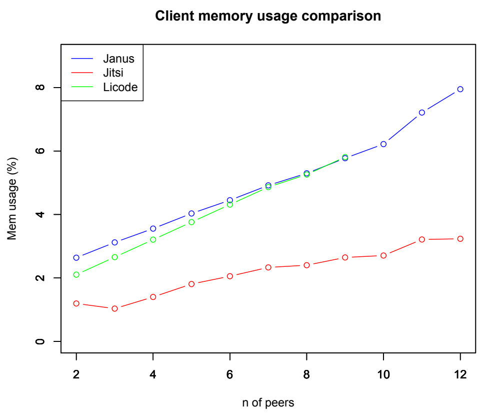
To make things simple, we chose to do our experimentation on a single plugin to be used as a reference, that is the Video MCU plugin. There were several reasons behind this choice: i) the Video MCU plugin exposes some of the most commonly desired features in a WebRTC gateway; ii) it can handle several different PeerConnections at the same time, thus proving a valid testbed for the relaying functionality in the gateway; iii) there are other open source tools available specifically conceived for acting as WebRTC MCUs (see Sec. 6), which makes a comparison of results easier. Of course, we plan to also start further campaigns addressing other plugins as well, which will hopefully help us assess the extensibility and capability of the core itself.

As a first step for the experimental campaign, we chose the tools to use as a comparison for Janus. Specifically, we decided to use Licode and the Jitsi videobridge, two of the most popular WebRTC conferencing platforms available in the open source community. All such systems, including Janus, were installed on the same machine, in order to make sure the results of each test session would be comparable. The purpose of the test was to assess the performance associated with two different factors: CPU and memory usage on both the server and the client (participant) sides. In fact, while the performance of the server is obviously important, the impact of the usage of such a platform on a generic participant cannot be underestimated, and represented in our opinion an important aspect to be taken into account. The server machine was equipped with two 3.2GHz Intel Xeon CPUs and 2GB RAM. It hosted a 32bit Linux Ubuntu Server 12.04 OS. The machine on which we collected client performance data was a 2.4GHz Intel Core 2 Duo with 4GB RAM, hosting a Mac OS X 10.9.3. For all tests, we made use of Google Chrome v. 35.0.1916.153.

We designed some tools to automatically stress test each component to compare, by leveraging the APIs exposed by Licode, Jitsi, and Janus in order to prepare scripts that could be used to programmatically generate users to attach to the platforms. Since we were obviously interested in realistic scenarios, we involved actual WebRTC PeerConnections to handle as well. In order to simulate those without just relying on raw manpower (that is, people behind a laptop physically joining a WebRTC room), we leveraged the Selenium Framework [9] for browser automation. By using the proper "webdriver", this allowed us to remotely control (through Python scripts) Google Chrome instances deployed on some of our servers. Since we did not care about the actual content of the media flows, we made use of the `-use-fake-ui-for-media-stream` and `-use-fake-device-for-media-stream` flags that Chrome exposes for making testing easier. This allowed us to easily increase the number of "partici-



(a) Client CPU comparison



(b) Client MEM comparison

Figure 7: Experimentation results: client side

pants" needed to stress each tool, and just focus on server's performance assessment. In each experiment, we also had a single "real" user joining the WebRTC room, in order to evaluate the impact on the client side.

The several test sessions we carried out on each platform eventually allowed us to get some interesting results, depicted in Fig. 6 and Fig. 7. The figures represent a summary of the CPU and memory usage on both server and client sides when testing each of the target platforms.

For what concerns the server side, if we look at the average performance of each of the tools, we can see that the CPU usage of Janus was quite low, and apparently increasing in a roughly quadratic way with the number of active participants. This was the expected pattern, since every time a new participant joins, Janus has to create a new PeerConnection per each pre-existing participant. Similar yet slightly worse results were achieved by Licode, where we can still recognize a quadratic increase of the CPU load. Jitsi, instead, seemed to be the most CPU-hungry component on the server side. This is probably due to the fact that Jitsi videobridge is executed as a plugin of the Openfire server, which inevitably adds CPU and memory load, while Janus and Licode are stand-alone applications. These results can be seen in Fig. 6a. It is also worth remarking that Jitsi's CPU evolution is roughly linear with the number of participants: this can be explained by the fact that Jitsi is the only tool that makes use of SSRC multiplexing, and as such of a single PeerConnection for all participants, rather than having different PeerConnections for each participant to display.

For what concerns the memory usage on the server side, depicted in Fig. 6b, the results were pretty much the same again, with Janus using less memory than Licode and Jitsi.

The rest of the figures are instead related, as anticipated, to the usage of the same resources on the client side, i.e., they measure the impact each platform has on a single participant joining a conference. For what concerns CPU usage, which is depicted in Fig. 7a, the first results seemed to confirm what we got for the server side, that is a slightly better performance achieved by Janus when compared to Licode and Jitsi. When looking at the memory usage figures reported in Fig. 7b, instead, we can see how Janus and Licode mostly behave the same way, while Jitsi is the clear winner here. This can be explained again by Jitsi's usage of SSRC multiplexing: this technique, while more demanding in terms of CPU, requires less memory since less network resources are needed.

6. RELATED WORK

As soon as the first browser-based WebRTC implementations were made available, work has started on gateways aimed at making them interoperable with existing legacy architectures (SIP infrastructures in the first place). Several implementations of gateways are currently available, many of which even open sourced like our Janus.

That said, most of these implementations usually cover some specific scenarios or use cases. For instance, the webrtc2sip gateway by Doubango Telecom [1] was specifically aimed at implementing a WebRTC-to-SIP gateway, to interact with existing SIP and/or IMS infrastructures, taking care of transcoding if needed. Likewise, a lot of work has been done on well known and widespread platforms like Asterisk or Kamailio to make the interaction with the existing SIP infrastructure easier. Other implementations, like Licode [6] or Jitsi [4], have been specifically conceived as MCUs for video conferencing and collaboration scenarios. Implementations like Medooze [7] and Kurento [5], instead, have been designed as media servers.

To the best of our knowledge, no other effort has been devoted so far to the design of a general purpose WebRTC gateway that can be used in a number of different scenarios and whose features can be selectively activated and/or configured.

7. CONCLUSIONS AND FUTURE WORK

We have presented in this paper our efforts towards the design and implementation of a general purpose WebRTC gateway we called Janus and released as open source. We described our choices in terms of architecture, and presented preliminary results related to tests addressing one of the scenarios it can be used for (namely, video conferencing) and comparing it to some of the existing alternatives.

While in its current form Janus already proved to be effective, especially in terms of flexibility and adaptability to heterogeneous scenarios, work is of course far from completion. Future work definitely includes efforts on making it more stable and reliable, in order to turn it into a production-ready component that can be used in commercial applications. In that respect, we are already working on making it the actual multimedia backend of Meetecho, our web conferencing platform, and are helping several other developers and companies to integrate Janus in their infrastructure. Besides, we plan to make available additional plugins, e.g., to interact with Jingle or RTMP infrastructures. As anticipated before, we also hope that third-party plugins will start to appear as well, to address features that we originally did not envisage, hence pushing Janus beyond its current boundaries.

For what concerns the architecture, we plan to expand the way plugins can be attached and used. For instance, we are interested in allowing for the presence of plugin-to-plugin interaction, e.g., in terms of filters that can be used as a support to other modules (e.g., a recorder or transcoding module that can be used as a support for a more application-oriented plugin).

Finally, we of course aim at keeping the pace of the WebRTC standardization efforts, which are currently quite a moving target. In that respect, we will also work on integrating new functions as they start appearing or become more widespread (e.g., SSRC multiplexing and simulcasting).

Acknowledgments

The authors would like to acknowledge Antonio Bevilacqua and Pasquale Boemio for their significant contribution and help during the experimentation of Janus and its comparison to other tools.

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').

8. REFERENCES

[1] Doubango webrtc2sip. http://webrtc2sip.org/.
[2] Janus on GitHub. https://github.com/meetecho/janus-gateway.
[3] Janus Online Demos. http://janus.conf.meetecho.com.
[4] Jitsi Videobridge. https://jitsi.org/Projects/JitsiVideobridge.
[5] Kurento. http://www.kurento.org/.
[6] Licode. http://lynckia.com/licode/.
[7] Medooze. http://www.medooze.com/.
[8] Real-Time Communication in WEB-browsers (IETF). http://tools.ietf.org/wg/rtcweb/charters.
[9] Selenium Framework. http://docs.seleniumhq.org/.
[10] Tiscali Indoona. http://www.indoona.com/.
[11] Web Real-Time Communications Working Group Charter (W3C). http://www.w3.org/2011/04/webrtc-charter.html.
[12] A. Amirante, T. Castaldi, L. Miniero, and S. P. Romano. Separation of Responsibilities between Application Servers and Media Servers in NGNs: A Practical Approach. In *Next Generation Teletraffic and Wired/Wireless Advanced Networking*, pages 199–211. Springer, 2008.
[13] M. Baugher, D. McGrew, and M. N. et al. The Secure Real-time Transport Protocol (SRTP). RFC 3711, RFC Editor, March 2004.
[14] C. Holmberg, H. Alvestrand, and C. Jennings. Negotiating Media Multiplexing Using the Session Description Protocol (SDP). Internet-Draft draft-ietf-mmusic-sdp-bundle-negotiation-08, IETF Secretariat, Apr. 2014.
[15] E. Ivov, E. Rescorla, and J. Uberti. Trickle ICE: Incremental Provisioning of Candidates for the Interactive Connectivity Establishment (ICE) Protocol. Internet-Draft draft-ietf-mmusic-trickle-ice-01, IETF Secretariat, Feb. 2014.
[16] R. Jesup, S. Loreto, and M. Tuexen. WebRTC Data Channels. Internet-Draft draft-ietf-rtcweb-data-channel-11, IETF Secretariat, June 2014.
[17] S. Ludwig, J. Beda, and P. Saint-Andre. XEP-0166: Jingle. Technical report, XMPP Standards Foundation, December 2009.
[18] D. McGrew and E. Rescorla. Datagram Transport Layer Security (DTLS) Extension to Establish Keys for the Secure Real-time Transport Protocol (SRTP). RFC 5764, RFC Editor, May 2010.
[19] L. Miniero, V. Pascual, and S. G. Murillo. Guidelines to support RTCP end-to-end in Back-to-Back User Agents (B2BUAs). Internet-Draft draft-ietf-straw-b2bua-rtcp-01, IETF Secretariat, Dec. 2013.
[20] C. Perkins, M. Westerlund, and J. Ott. Web Real-Time Communication (WebRTC): Media Transport and Use of RTP. Internet-Draft draft-ietf-rtcweb-rtp-usage-17, IETF Secretariat, Apr. 2014.
[21] J. Rosenberg. Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols. RFC 5245, RFC Editor, April 2010.
[22] J. Rosenberg and H. Schulzrinne. An Offer/Answer Model with the Session Description Protocol (SDP). RFC 3264, RFC Editor, June 2002.
[23] J. Rosenberg, H. Schulzrinne, and G. C. et al. SIP: Session Initiation Protocol. RFC 3261, RFC Editor, June 2002.
[24] H. Schulzrinne, S. Casner, and R. F. et al. RTP: A Transport Protocol for Real-Time Applications. RFC 3550, RFC Editor, July 2003.
[25] J. Uberti and C. Jennings. Javascript Session Establishment Protocol. Internet-Draft draft-ietf-rtcweb-jsep-07, IETF Secretariat, Feb. 2014.