Università degli Studi di Napoli Federico II

Facoltà di Ingegneria

Dottorato di Ricerca in Ingegneria Informatica ed Automatica
XXVII Ciclo
Dipartimento di Ingegneria Elettrica e delle Tecnologie dell'Informazione

SOLEIL: Streaming Of Large scale Events over Internet cLouds

Lorenzo Miniero

Ph.D. Thesis

Tutor
Prof. Simon Pietro Romano

Coordinator
Prof. Francesco Garofalo

March 2015

Abstract

This Ph.D. Thesis deals with the design and realization of a streaming platform specifically targeted at large-scale and live, low delay, events. The streaming platform, called SOLEIL (Streaming Of Large scale Events over Internet cLouds), is conceived to be compliant, at its edge nodes, with the WebRTC specification, meaning users can take advantage of its capabilities using a simple browser. Details on both the design and realization processes are provided, and a thorough experimental campaign will prove the validity of the achieved results.

Acknowledgments

This thesis wouldn't have been possible without the help and support by excellent people.

I cannot start this acknowledgments section without thanking my family. My parents have always been more than supportive, no matter which path I decided to take, and deserve this thanks more than anyone else. A huge hug to my little sisters as well: despite what a doctoral thesis may suggest, they're way smarter than me!

Thanks to my tutor, Prof. Simon Pietro Romano: several years have passed since I first met him to discuss the possibility of starting a Master Degree thesis with him, and I've never stopped learning from him since then. What he saw in that *hippie* that made him think I could achieve good results in my thesis and in research after that I'll never know: I do know that he always supported and encouraged me in all my activities, and in all these years he did make me a better researcher and a better person.

Thanks to the excellent partners I have in the Meetecho adventure: Tobia and Alessandro, together with Simon, believed in this as much as I did, and for several years have been the companions of a long trip that is still going on, and will hopefully keep on going for a long time. All this years confirmed my certainty that I've not only found incredible colleagues, but lifetime friends.

And, last but not least, thank you for taking the time to read this work: I hope you'll enjoy going through it just as much as I did in these past three years, which have been a challenging and exciting journey.

"If I could put it into a very few words, dear sir, I should say that our prevalent belief is in moderation. We inculcate the virtue of avoiding excesses of all kinds – even including, if you will pardon the paradox, excess of virtue itself."

Contents

Chapter 1

Introduction

Streaming has become a more and more pervasive application scenario in our lives, whether it's about on-demand media or live events. Services like YouTube, Vimeo and others have made streaming easily accessible to the masses, by providing web interfaces and ease of access to media that were previously available only through ad-hoc player applications.

This was made possible by the evolution in the technology and protocols used to implement such a functionality. 20 years have passed since the first streaming applications made available by RealNetworks, and just a few less since that effort eventually became the first standard protocol for implementing media streaming, the Real Time Streaming Protocol (RTSP). At the time, the protocols used to implement this functionality required users to install an ad-hoc application for the purpose of receiving and reproducing the feeds.

Such a requirement has proven, year after year, harder and harder to justify. Nowadays, we are all used to use our web browser for pretty much anything, including things we used desktop application for in the past. While a few years back we would have relied on a mail client for sending and receiving e-mails, WebMails have become more and more widespread. The same applies to document editing, which has started to shift from desktop applications like the Office Suite to web environments as Google Drive. This evolution from desktop applications to web interfaces has impacted many more

applications as well, including those that envisage some form of interaction with other people: nowadays it's not at all unusual, for instance, to chat with friends and colleagues using a website, rather than an instant messaging application. As a result of this change in perspective, users have become increasingly reluctant to install applications on their systems. Whether they trust the implementor or not (another factor that has contributed to this reluctance), users often expect the applications they need to be available on some website, one way or another.

While this is often true for most applications, this is not always the case for multimedia applications, especially for those that have more or less stringent requirements in terms of real-time functionality. In fact, despite the fact that such applications are indeed currently available to web users, they most of the time rely on proprietary plugins for the purpose. Almost all of the applications that provide collaboration features, for instance, have for a long time relied on plugins like Adobe Macromedia Flash, Microsoft Silverlight or Java Applets. That's the case, to name a few, of Google Hangout, Adobe Connect, the video calls in Facebook made available by the Skype plugin, and so on. The same also partly applies to multimedia streaming scenarios as well. Most of the streaming providers still rely on plugins like the ones mentioned above, although purely web-based alternatives have become available, e.g., HTML5 and HTTP-based streaming (HLS and MPEG-DASH), often involving content delivery networks (CDNs) in order to to ensure an efficient delivery of popular content to large audiences.

While functional, a plugin-based approach is far but optimal. In fact, a plugin is, by definition, something that was not natively available in the browser, and so needs to be implemented by someone else as a third-party addition and integrated within a web page somehow. This often results in proprietary libraries that are not able to interact with other components. Besides, in order to target users on different systems and using different browsers, such libraries need to have different implementations for the same functionality: to make a simple example, should a developer be interested

in making a plugin available to all potential users, there needs to be a Windows, a Mac OS and a Linux version of such a plugin, while at the same time targeting both x32 and x64 architectures too; besides, different versions of the plugin may be needed on different browsers, e.g., Internet Explorer, Chrome, Firefox, Safari, etc. This is a very complex and demanding task, one that as a matter of fact several implementors have chose not to pursue. As a result, it's not unusual to encounter plugins that are only available on Windows or on specific browsers, which means that relying on such solutions for implementing a multimedia streaming application is very likely to lead to platform-dependency and a complete lack of interoperability. Platform dependency becomes even more of an issue when we think about mobile devices like smartphones and tablets. In fact, even envisaging a plugin that covers all desktop environments, there currently is no way to write plugins for mobile browsers, with the result of having a large basin of potential users uncovered, unless ad-hoc applications are written for the purpose.

It's within this framework that I started, together with my colleague Tobia Castaldi, to investigate some possible alternatives to address the several shortcomings that have been just identified. More specifically, we decided to study the WebRTC standardisation effort as a technology enabler to provide native browser-based multimedia streaming and, more in general, communication, on a more or less large scale. This effort lead to two doctoral thesis, carried in parallel by me and my colleague, in order to design a large-scale streaming architecture that would be standards-compliant, scalable, dynamically deployable and web based. We would address different aspects of the architecture, in order to eventually merge the efforts into a joint framework.

Chapter 2 introduces this architecture, which we called *Streaming Of Large scale Events over Internet cLouds* (SOLEIL), by analyzing the requirements we identified and the components implementing the different roles. Particular focus will be given on the low-latency requirements, while keeping the quality acceptable from a user's perspective.

Chapter 3 will instead delve in the details related to the ongoing WebRTC

standardisation efforts, by describing the several different requirements that were targeted and how they were eventually addressed. The chapter will highlight my contributions in that respect, and describe the preliminary implementation efforts I took care of in a web conferencing environment.

The implementation aspects of SOLEIL will be addressed in Chapter 4, where the component that will be responsible for the implementation of the SOLEIL "last mile" towards end-users, Janus, will be described. In particular, the general purpose and modular architecture of Janus will be introduced, together with the extensible API.

Experimentation results and enhancements to the platform that came as a result will eventually be addressed in Chapter 5.

Some final remarks, together with hints on possible next steps, will be provided in Chapter 6.

Chapter 2

SOLEIL: design and architecture

Streaming as an application scenario has been around for many years already. The easiest way to describe it is the transmission of a multimedia source of information to one or more destination across a network. Such a multimedia source is usually encoded in order to make the delivery through a network more efficient, and sometimes encrypted when security and privacy represent a concern. That said, this very simple definition is often not enough to describe streaming as a technology. In fact, several different approaches can be taken to actually implement such a functionality, usually depending on the requirements of the applications, the constraints of the technology or other aspects. For instance, a first distinction can be made at the very transport level: should a stream be transferred to its recipients using unicast or multicast? Is a simple file transfer enough for the purpose, or would a progressive delivery be more suit to the purpose?

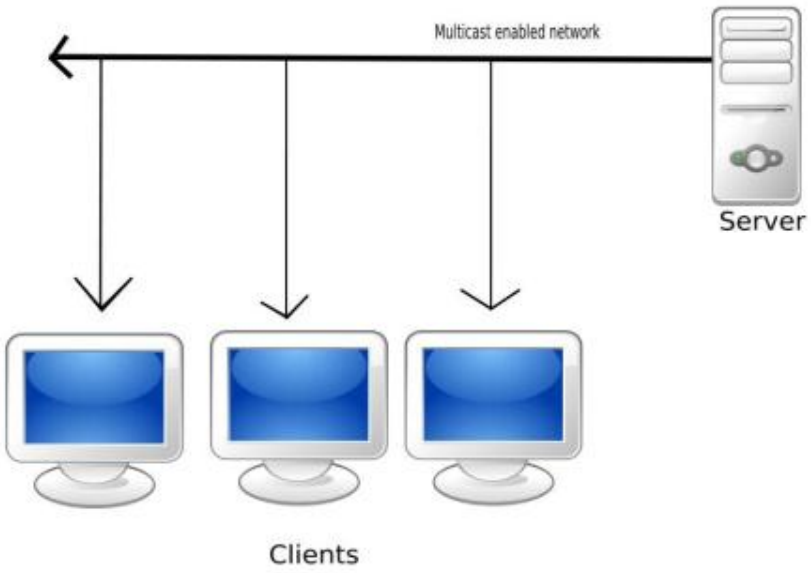
2.1 Streaming as an application

These and other questions have paved the way to what streaming is today and how it can be achieved. As it often happens, there's not a best answer that fits everything, but it all depends on the kind of scenario the streaming application wants to tackle. For instance, a multicast delivery of a stream to

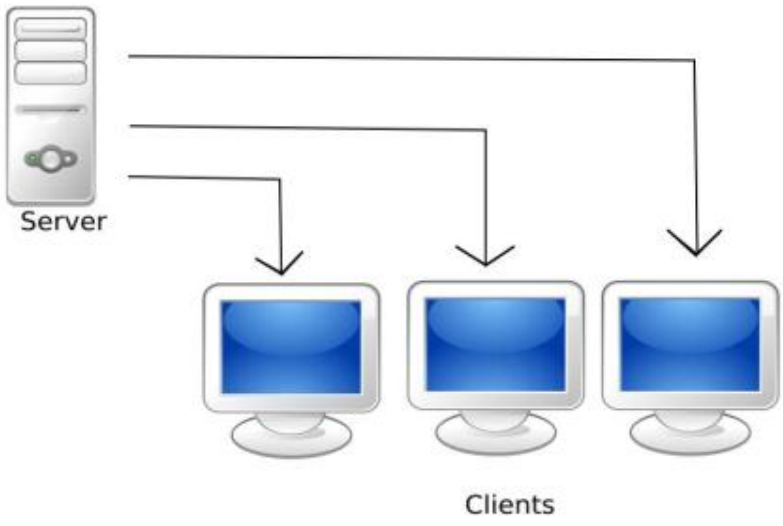
multiple recipients as depicted in Figure 2.1a is a very efficient way to transfer the same pieces of information from a single source to several destinations, but has several constraints too: it requires support for a multicast network, and does not allow for content adaptation or streaming on demand as the stream is shared among all viewers. At the same time, unicast as depicted in Figure 2.1b works perfectly when streaming from a single source to a single destination, but proves more challenging when multiple destinations need to be reached. Anyway, in recent evolutions of the network multicast-based streaming has become a less and less widespread solution, usually limited to LAN-based streaming. The reason for this is basically the above mentioned limited availability of pure multicast networks, especially outside the boundaries of private networks. As a consequence of this, all streaming technologies, both proprietary and standard, make use of unicast streams to implement streaming to both single and multiple recipients, employing the required optimization steps at different levels.

A more distinctive difference in streaming application can be highlighted at a different level. Specifically, one can identify two separate streaming scenarios that, as a consequence, have completely different requirements and constraints that different streaming technology usually handle accordingly: (i) *on-demand* streams and (ii) *live* streams. A good overview on the typical approaches used for video streaming is provided in [1], where the authors analyze the common requirements in terms of video compression, Quality of Service (QoS) control, media synchronization mechanisms and distribution.

When talking about *on-demand* streams, we usually envision pre-recorded (and possibly pre-encoded) streams that viewers can request and watch on their own and on their schedule. This is what popular services like YouTube, Vimeo and others have been making available for some time already, and what some TV channels like RAI, Mediaset, Sky and others have started to provide as an additional service too, e.g., to allow viewers to re-watch a TV show or an event that already occurred on TV. Since we're talking about pre-recorded streams that users ask for specifically, the streaming technolo-



(a) Multicast Streaming



(b) Unicast Streaming

Figure 2.1: Multicast vs. Unicast Streaming

gies suited to cover these scenarios are usually optimised for a personalised delivery of those flows. Each viewer has their own streaming context, which means it has complete control over the playout process, e.g., in terms of pausing, resuming, rewinding the stream and so on. Besides, since the source is pre-recorded, it is usually pre-encoded as well, in order to furtherly optimise the delivery process: in fact, the streaming server is this way not required to also encode the same stream over and over for all interested viewers, but can just rely on an already encoded stream that only needs to be properly packetised and transferred in an optimised way. Several different technologies have been implemented across the years to cover this scenario, and most of them nowadays allow for a web-based access to such streams, whether using proprietary technology (Adobe Flash, Microsoft Silverlight) or standard ones (Real Time Streaming Protocol, HTML5).

A completely different scenario is the one referring to *live* streams instead. In this case, the source cannot possibly be pre-recorded, as the event it is capturing is happening in that moment, which means that pre-encoding the stream is not a viable solution for optimizing either. In fact, as depicted in Figure 2.2, the approach is to usually have a live source of media, e.g., a camera, capture a stream to have it encoded on the fly and restreamed by a streaming server. Besides, a *live* stream imposes more constraints over the level of control viewers can enforce on it: you definitely cannot fast-forward a *live* event, for instance, and even pausing/resuming has less sense in such a scenario. That said, *live* streams represent an application more and more people are starting to get interested in for several different reasons. Whether it's about attending a live conference remotely or simply watching a football match from the comfort of a couch, the ability to watch a live event using web technologies has become increasingly important and desirable. Just as for the *on-demand* scenario, that are services that have focused on making *live* streaming a working application, especially focusing on scalability concerns to allow for more people to attend the same event. Specifically, several efforts have been devoted across the years on standard and proprietary

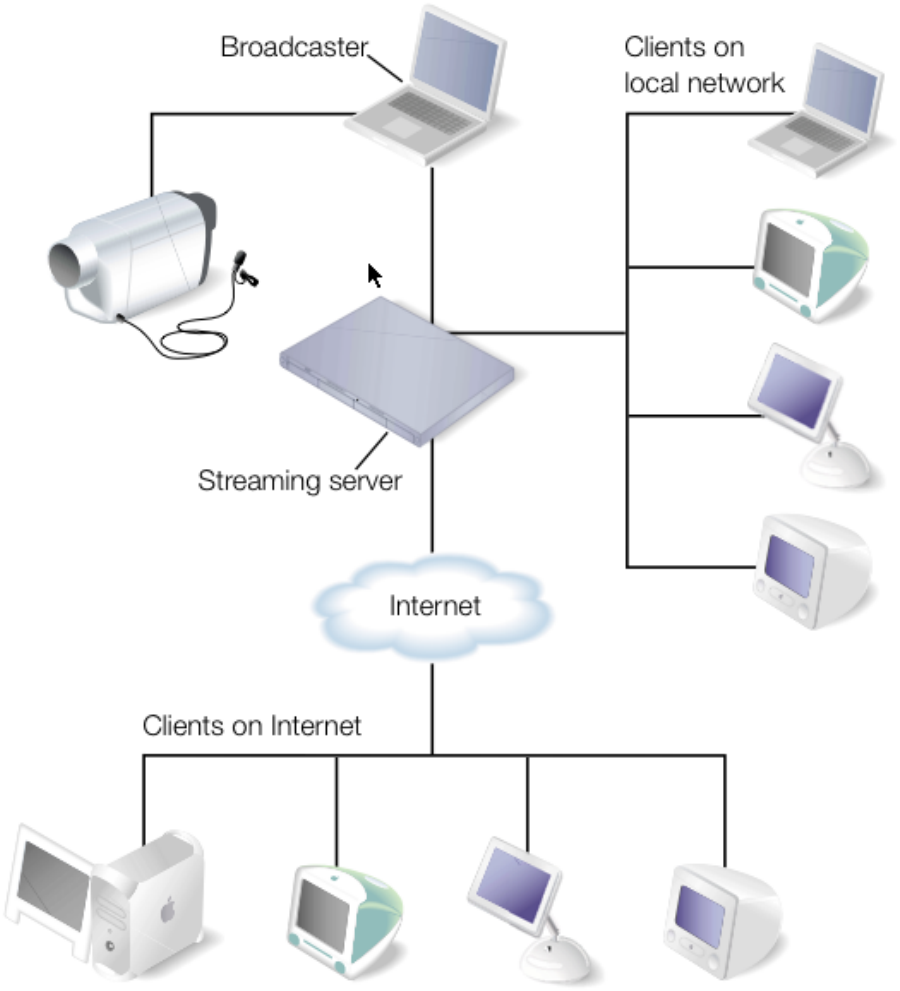


Figure 2.2: Streaming live events: broadcaster and streaming server

solutions to allow for such scenarios to take place in a web environment. Such solutions include proprietary protocols like Adobe's Real Time Messaging Protocol (RTMP) [2] or Microsoft's Silverlight [3], both of which require specific plugins to work, and standard technologies like Dynamic Adaptive Streaming over HTTP (DASH) [4].

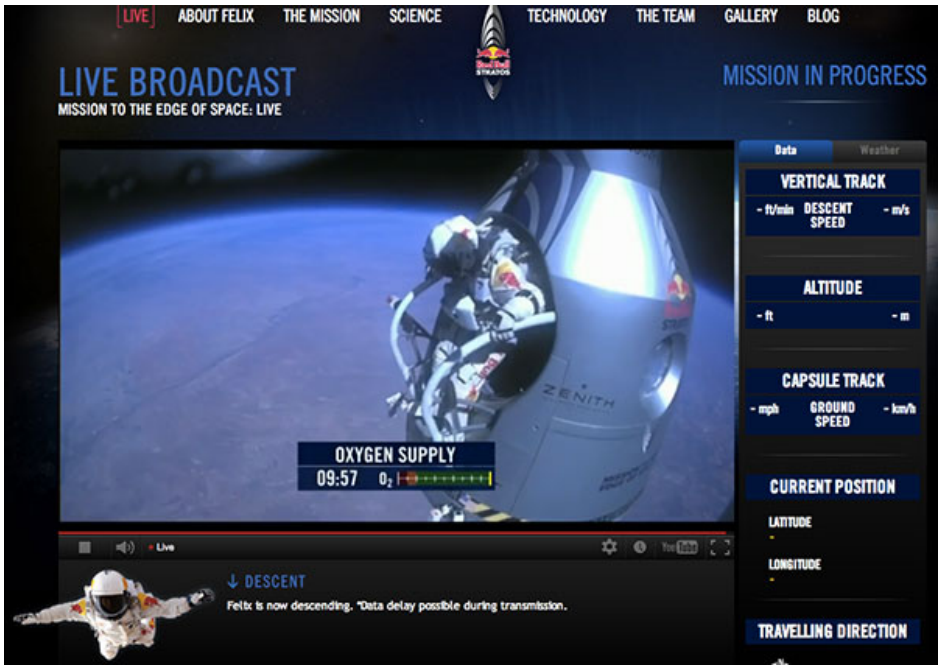
That said, the above solutions, while more or less effective, all suffer from the same issue: they tend to favor quality over delay, which means you'll usually get a good quality stream, but with a sometimes considerable delay, ranging from 10 seconds to more than a minute. While this may not look as a serious issue, its impact can be more than noticeable if we consider some specific contexts. For instance, having a delay in getting a feed from a webinar or conference may not be very important, but it would be if you're watching a live football game instead, especially if the pub down the road is transmitting the same match on live TV and supporters there exult for a goal you haven't seen yet. Besides, a truly *live* feed allows for a much more interactive fruition of the media: different users watching the same event may interact with each other or even with the source of the media itself using out-of-band mechanisms (e.g., instant messaging or a separate audio/video feed) exactly while the event is happening, rather than contenting themselves of commenting whatever happened minutes ago.

These and similar scenarios are among what motivated us in investigating better solutions for *live* streaming, and in particular for streaming of events with a potentially very large audience.

2.2 Large-scale Streaming as a new use case

As anticipated in the previous section, *live* streaming represents an open field for research, especially as far as delay is concerned. This scenario becomes even more relevant when considering a potentially very large audience, since the amount of viewers for an event implicitly indicate the level of interest for the event itself, and as a result their level of tolerance for an excessive delay in the transmission.

(a) TV feed



(b) YouTube stream

Figure 2.3: A very large scale event: Felix Baumgartner at the Red Bull Stratos

Events like President Obama's speech or the Superbowl, for instance, are watched at the same time by millions of people at the same time, all interested in what's happening in that specific moment. The same can be said about similar very popular events, like the famous 39 kilometers jump Felix Baumgartner made in 2012 as part of a Red Bull Stratos event, a snapshot of which is depicted in Figure 2.3. This event specifically had a very large media coverage, and was transmitted live on both regular television and in streaming, by means of the YouTube Live platform. The live stream, in particular, which was followed by millions of viewers across the works, was of very good quality and provided additional interactive pieces of information to viewers, but also suffered from a considerabile delay, which was in the range of 60-90 seconds. This means that whatever happened in real-time was only perceived by streaming viewers more than a minute later, thus affecting in a very noticeable way the emotional response.

As anticipated in the previous section, scenarios like these strongly motivated us to look into better solutions, specifically targeted at large-scale live events. Our feeling was that, while delay is in part hardly avoidable when it comes to events with such a high requirement in scalability, it should not be excessively sacrificed on the altar of quality, and that the two can proceed hand-in-hand using specifically tailored architectural choices. In particular, recent and innovative technologies like WebRTC [5] [6] allow for issues like delay to be greatly reduced, thanks to protocols that are specifically conceived for a live transmission, rather than adapting to unfit technologies like HTTP for the purpose.

More precisely, starting from the previous considerations we decided to work on a framework that woult take into account a set of specific requirements, that is:

- an architecture that had scalability in mind;
- compliance with the existing and new standards;
- support for dynamic "deployability";

- web-based access for end users.

The following section will introduce the architecture we conceived to cover these requirements, called *Streaming Of Large scale Events over Internet cLouds* (SOLEIL). More details about the WebRTC technology we based it upon will instead be provided in Chapter 3.

2.3 Streaming Of Large scale Events over Internet cLouds

In the previous sections we explained how the excessive delay introduced by current technologies in *live* streaming greatly affects the expectations viewers have of such applications. Besides, we hightlighted the requirements we identified for a new architecture that could address the target scenario in a scalable and standard way. This motivated us to look into solutions more apt to the purpose, and eventually lead to the design of a completely new architecture called *Streaming Of Large scale Events over Internet cLouds* (SOLEIL).

The first assumption we started from is that all current web-based streaming technologies are using, in our opinion, the wrong foundation: that is, considering the constraints typically imposed by what's made available by browsers (the main target for implementing viewer applications nowadays), all of these solutions either recur to ad-hoc plugins with proprietary protocols for the purpose, or fall back to unfit protocols like HTTP for transferring the media. Both of these approaches, while functional, have several issues caused by the fact they're based on reliable protocols originally conceived for completely different purposes. For instance, while HTTP is an excellent general purpose protocol for transferring more or less static information between a client and a server, it definitely falls short as soon as it is involved in the streaming of information that needs to be as timely as possible, due to the overhead it imposes and it being based on the Transmission Control Protocol (TCP) for reliability. While optimizations like chunked-based transfer help

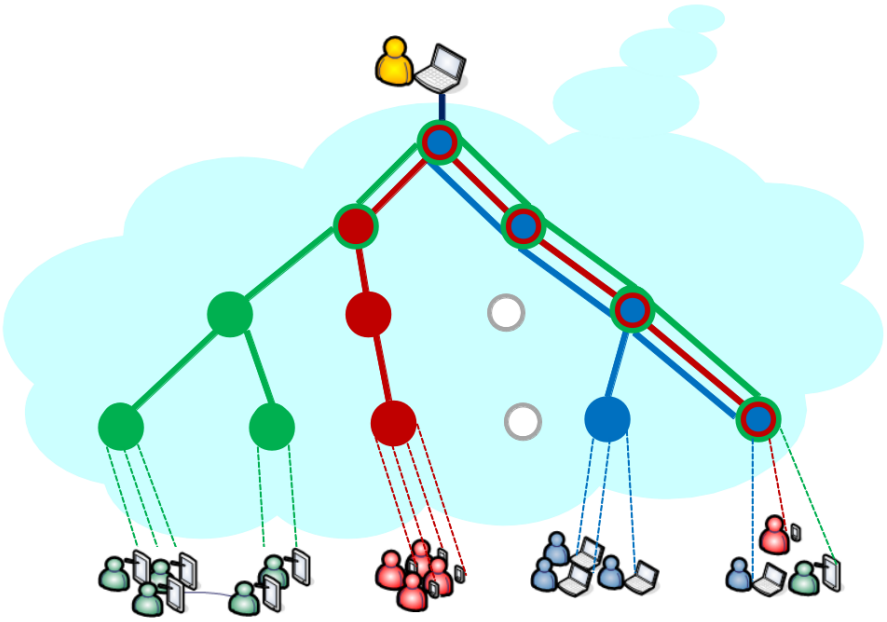


Figure 2.4: SOLEIL architecture: a tree topology

in coping with some of these issues, they do not manage to solve the issue at its core, that is the protocol is unsuited for the purpose when actual *live* streaming is required. Similar issues affect plugin-based solutions as well: while forcing a user to make use of a proprietary plugin for an application is already a wrong approach for several reasons (including strict dependencies on the operating system), such solutions typically try to make use of reliable protocols as well, since they're conceived to work in pretty much the same setups where regular web-based communications work.

For these reasons, the first step we made in conceiving our architecture was to employ a different technology as the foundation for the streaming part. One of the requirements identified at the beginning was standards compliance, which is why we eventually decided to make use of the standard Real-Time Transport Protocol (RTP) [7] for the purpose. Such a protocol has been used for years within the context of standards-based Voice over

IP (VoIP) technologies, and was the foundation of the legacy Real Time Streaming Protocol (RTSP) [8] as well. Besides, it currently is the foundation of the new WebRTC standard technology as well, which allowed us to cover another of the key requirements, that is a web-based access to the platform that would make it easier for end users to use it. All this convinced us that RTP would be the perfect candidate for the purpose: in fact, since it was conceived to allow for real-time communications among two or more peers, it would definitely be suited for monodirectional live delivery of streams as well, with the minimum possible impact on latency and as such delay. Besides, RTP is by definition coupled with the Real-Time Transport Control Protocol (RTCP), which is a very effective tool for the monitoring and management of the status of the multimedia delivery. This provided us with an excellent means for gathering statistics on the transmission at different levels and, as it will be explained in the following chapters, also for implementing pro-active feedback to increase the overall quality of experience.

From this first step, we eventually had to focus on the most suited model for employing this protocol in the delivery from a single source to a vast audience of destinations, especially considering the large-scale target we conceived for the platform. After reasoning on this a bit, we decided to partly re-use some approaches that are well known in literature, even though not necessarily within the same field. Specifically, since one important requirement for the architecture was a way to scale the platform, we decided to envisage a tree-based topology to increase the scalability of the framework. A similar topology was for instance introduced in [9], where the authors envisaged an architecture to overcome the limiting factor for large-scale media streaming services based on Source-Specific Multicast (SSM), that is the excessive RTCP bandwidth being shared among all the receivers.

Unlike the mentioned effort, we conceived a tree-based topology where each node could have different roles: (i) a *root*, that is the root of the tree that originates the stream itself; (ii) several *relay* nodes at different levels, whose main purpose is to make the stream they receive available at several different

nodes at the same time; (iii) *leaf* nodes, who are responsible for making the stream available to the actual viewers. A high level perspective on the proposed tree topology is depicted in Figure 2.4. Such a tree-based topology, with nodes that could be added and removed dynamically in order to properly scale the platform, also allowed us to address the final requirement that we identified, that is some kind of dynamic "deployability" of the platform.

Apart from this tree topology, which is responsible for the actual delivery of the media, we can actually envisage an additional logical component, the so-called *master* node. This component is responsible for the dynamic organization of the tree, e.g., to allocate new *relay* nodes when needed, or to discipline the injection of a new media flow. It is important to point out that, due to the nature of such a component, it might or might not be colocated with the *root*, and might besides be physically partitioned in different machines in order to have it scale properly in an orthogonal way with respect to the SOLEIL media distribution network itself.

2.3.1 The *root* node

As to the *root*, it's important to point out that this does not necessarily have to be the actual source of the multimedia streaming. This is just the root of the tree, and as such is the node that is either actually originating or just receiving the live stream that needs to be made available to the audience. As such, this *root* constitutes the first point of access to the SOLEIL architecture, and is responsible for starting the delivery of the stream by injecting it in the first level of *relay* nodes. This role is depicted in Figure 2.5, in which an external device, e.g., a camera or a laptop, is capturing and encoding a stream; this stream is then injected in the SOLEIL infrastructure through the *root* node, which takes care of relaying it to the lower branches of the tree as configured.

Another important aspect to clarify is that a stream will only be injected once, no matter how many viewers will eventually be attached to it: in fact, since the source is the same and shared for everyone, the relevant nodes only

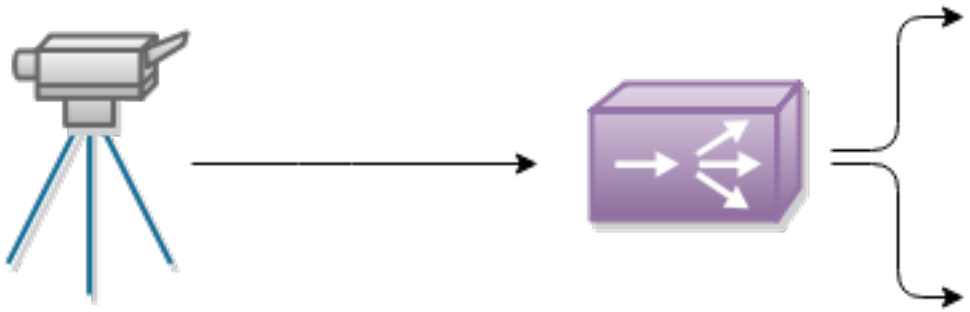


Figure 2.5: SOLEIL architecture: the root/broadcaster

need to have access to the stream once, in order to be able to properly make it available to all interested viewers. This allows for a much more optimised delivery of the streams, using a pseudo-multicast distribution.

We envisaged this approach as some kind of "waterfall" controlled by "valves": the "waterfall" source starts at the root and, assuming "valves" are open at some of the children nodes, as a spring it flows through those as well, down to the point where one of the "valves" is closed or eventually to endpoints.

2.3.2 The *relay* node

The *relay* nodes, instead, as their name suggests are only meant to relay the media they receive from their parent node (be it the *root* or another *relay* node) to all their children nodes (which may be another level of *relay* nodes or *leaf* nodes, if the end of the tree has been reached). Their main purpose is to effectively increase the width of the tree and, as a result, the very scalability of the stream. In fact, considering each node has limits in terms of how many destinations it can serve, by adding further levels of *relay* nodes you're actually increasing the number of viewers that can be reached at the same time, by relying on their additional potential basin of destinations. Of course, each level of *relay* nodes adds a small amount of latency to the distribution, which means this approach should not be taken lightly and should be properly weighted instead. Figure 2.6 depicts a simplified zoomed

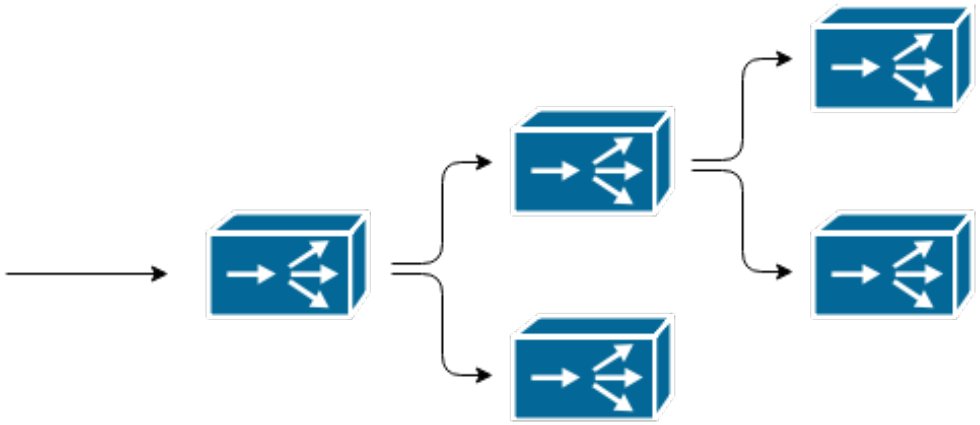


Figure 2.6: SOLEIL architecture: the relays

view on the *relay* node components in the SOLEIL architecture: as the figure illustrates, the stream coming from the parent (be it another *relay* node at an upper level or the *root* itself) is relayed at the lower levels to other nodes that are part of the architecture.

An important aspect to consider is that, considering the "waterfall" and "valves" approach described before, a *relay* node is only going to relay a stream if any of its children (be it a *relay* or *leaf* node) is interested in receiving it, that is if there's any viewer attached to that subtree that is going to need that stream. This allows for a much more lightweight approach for what concerns the distribution of the flows, as streams will only be transferred where they're needed; besides, this also allows for further optimizations that may be enforced on the topology, e.g., to aggregate viewers of the same kind in the same subtree in order not to waste bandwidth. Just as the *root*, a *relay* node only transmits the same stream once per children, no matter how many recipients are meant to receive it beyond that level. Considering the usage of RTP for the actual relaying of media streams across node levels, *relay* nodes are finally also responsible for gathering hop-by-hop statistics using RTCP, in order to identify potential issues in the distribution tree at either a local or more global scale.

These and other aspects will be described in better detail in Section 2.5, which will introduce the different responsibilities a relay node will have.

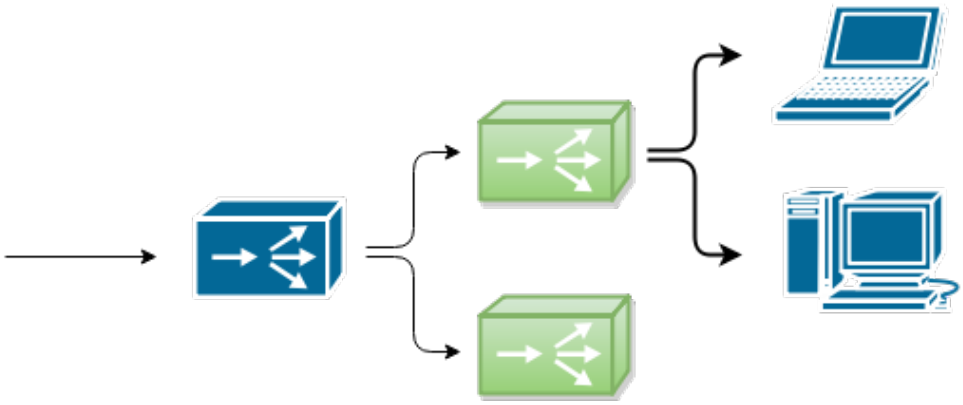


Figure 2.7: SOLEIL architecture: the leaves

2.3.3 The *leaf* node

Finally, at the edge of the tree there are the so-called *leaf* nodes. These nodes differ from the generic *relay* nodes in the sense that, while *relay* nodes simply relay media to other nodes of the same kind while monitoring statistics along the way, *leaf* nodes are instead responsible for making the feeds available to the actual viewers, and as such using a different technology. In fact, while the streams are internally relayed using plain RTP/RTCP, the viewers will need to receive those in another format, especially if they're supposed to be able to display them using a regular web browser. Figure 2.7 depicts the different nature of a *leaf* node when compared to the responsibilities of a generic *relay* node. Specifically, while *relay* nodes, colored in blue, take care of relaying the stream originated by the broadcaster using RTP/RTCP, the *leaf* nodes, colored in green, have to employ a different technology to reach their viewers (depicted as generic laptops or computers in the picture, but those could be heterogeneous devices like smart TVs, phones or tablets as well), which explains why the related arrows have a different width.

As anticipated before, the technology we identified for the purpose is the recent standard called WebRTC, which was partly based on legacy VoIP protocols, properly extended in order to account for the additional requirements in terms of effective connectivity establishment, security and privacy. Considering this peculiar role for *leaf* nodes, they constitute what may be

considered as the SOLEIL "last mile", as, no matter what happened up to that point, this is where viewers will physically attach and whence they'll expect to receive the actual streams from. This "last mile" will be addressed in the Section 2.6 more in detail, and will be the main subject of this doctoral thesis due to its specific requirements and challenges.

2.3.4 The *master* node

When first introducing the SOLEIL architecture, we mentioned an additional component that, while not directly involved in the media delivery and as such not part of the tree topology that takes care of the flows distribution, is indeed a very important piece of the framework. This component, called the *master* node, is indeed what might be better described as the *manager* or *handler* of the SOLEIL distribution architecture. More specifically, it's the component responsible for the dynamic management of the tree and possible recovery actions.

Section 2.4 will introduce the SOLEIL protocol and how it can be used to dynamically add and remove nodes and flows. For what concerns recovery, instead, the mechanisms used in SOLEIL for the purpose will be addressed in Section 2.5.

2.4 Managing the tree: the SOLEIL protocol

One of the requirements we identified when first conceiving the SOLEIL architecture was some way to dynamically deploy the components involved in the process. Considering the tree based topology and the roles identified within the architecture, this mainly referred to the possibility of dynamically adding and removing *relay* and *leaf* nodes, by starting ad-hoc machine on the fly for the purpose if needed, and handling subscriptions to the available streams. This was an important aspect to take into account as, while a scalable architecture clearly demands an on-demand availability of new resources when required, an over-provisioning with more resources than are actually

needed for a flow can be problematic as well. For this reason, we conceived a simple protocol that would allow us to programmatically allocate resources and update the tree nodes accordingly, in function of incoming requests for a flow or to react to events on the SOLEIL network. Considering the focus of this doctoral thesis was not on the architectural aspects of the framework, we won't delve into the details of this protocol, that was implemented as a eXtensible Markup Language (XML) based one. We will, though, address some common use cases we wanted to tackle, and the way the SOLEIL protocol was conceived to handle them.

The main use cases we wanted to address were two: (i) how a device capturing a stream could inject the flow in SOLEIL, and (ii) how another node in the SOLEIL network or a viewer could make sure this could be received. These two use cases have slightly different requirements. The former, which is addressed in Section 2.4.1, can mostly be seen as a discovery process: the broadcaster needs a reference *root* node to send the media frames to. On the other hand, the latter, described in Section 2.4.2, could be seen as some form or "reservation": a node or viewer needs a reference "father" to get the stream, and in case this node is not receiving the stream as part of the "waterfall" approach we conceived in SOLEIL, the parent nodes need to be informed in order to turn the "valves" in the *relay* nodes and make this stream available.

2.4.1 Injecting new flows in SOLEIL

Figure 2.8 sketches the way the first use case is handled within the SOLEIL protocol. More specifically, the figure depicts three different parties in the communication: (i) a device capturing some media that need to be broadcast, pictured as a camera; (ii) a *master* node that the device refers to in order to get information, depicted in the blue box on the right; (iii) a *root* node that can inject flows in the SOLEIL distribution network, sketched as in Figure 2.5. In this figure, the device is interested in broadcasting the stream it is capturing. In order to so so, it contacts (1.) the *master* node, by means

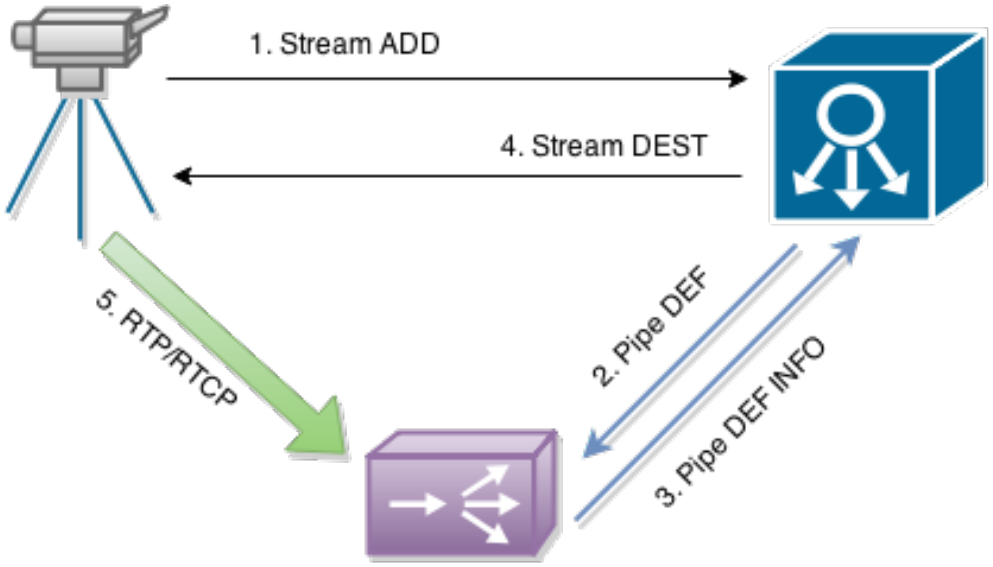


Figure 2.8: SOLEIL protocol: injecting a new flow

of an HTTP request, to get the address of a node it can send this stream to. The *master* node handles the request and, according to the parameters that were passed, chooses a *root* node among the available ones, or spawns a new instance just for the purpose. The chosen *root* node is then contacted by the *master* node (2.) in order to allocate the required resources: in particular, the *root* node starts listening on a couple of ports to receive the RTP stream and exchange RTCP messages. Once done, a response (3.) is sent to the *master* node, which realises the allocation has been successful. At this point, it can provide the broadcasting device with the final info it requested (4.), including the network addresses needed to establish the multimedia connectivity. At this stage, the broadcasting device has all the information it needs to send the flow as an RTP stream to the chosen *root* node and to exchange RTCP feedback and statistics with it, confident that from that moment on the flow will be injected in the SOLEIL framework and potentially be available to as many viewers will be interested in it.

As it can be seen in the figure, the communication between device and *master* node, and between *master* and *root* nodes, are sketched in different colors. This is done on purpose as, while both steps are actually based on a

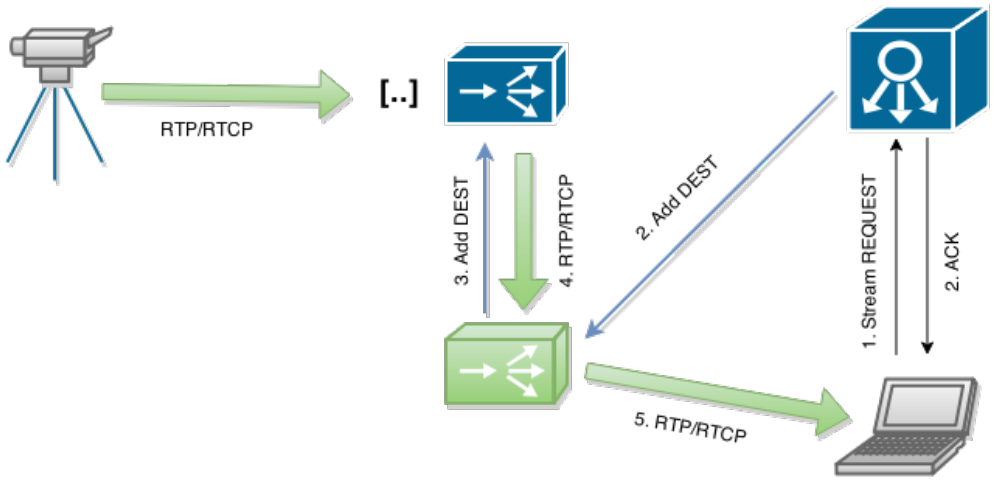


Figure 2.9: SOLEIL protocol: subscription of a viewer

usage of the SOLEIL protocol, the related semantics and transport protocol are different. In fact, the device is expected in most cases to be a web-based component: as such, it makes sense for it to use (1. and 4.) HTTP or WebSockets as a transport for its requests. On the other hand, the two inner nodes communicate using a different, still reliable, protocol: this is the XML-based SOLEIL protocol that was introduced in this section.

2.4.2 Handling new subscriptions

The second case introduced before, instead, is sketched in Figure 2.9. In this use case, we assume that a flow is already being injected in a SOLEIL network through a *root* node, e.g., after a sequence like the one described above. At this point, a new user decides to watch this stream. The *master*, just as before, must choose a node to provide the user with in order to allow it to receive the stream. In case a *leaf* node suited for the purpose is already available (e.g., one that is already receiving the stream and that is not overloaded) its address can be returned right away. For the sake of this example, though, we assume that the *master* node doesn't have any candidate available, e.g., because all nodes have too many viewers already or because none of the available ones is receiving that specific stream as of yet.

In order to do so, the *master* node may decide to spawn some *relay* and/or *leaf* node instances on the fly. What's important to highlight is that the request by the user (1.) is in this case handled by the *master* by contacting a *leaf* node (2.) in order to make sure it subscribes to the requested stream on behalf of the user. As anticipated in the previous section, this is only needed the first time a viewer shows interest for a specific stream, as after that the stream becomes available for all viewers that may follow. Since in this example the chosen *leaf* node is not receiving the requested stream, it forwards the same request to its parent, that is the *relay* node it is going to receive the media from. This may proceed up some more levels, until the request reaches a *relay* node that is already receiving the stream: for the sake of simplicity, we left out the rest of the tree topology that may be in place, and just focused on these last nodes. Once done that, the "valves" are open and the involved nodes can establish the required hop-by-hop RTP/RTCP trunks (4. and 5.) needed to forward the media and exchange statistics, which eventually results in the flow being made available to the reference *leaf node*.

As in the previous example, different protocols are involved in this sequence. Again, the SOLEIL components interact with each other by means of the custom SOLEIL protocol we devised for the purpose. The user interested in watching the stream, instead, resorts to different technologies for requesting access to the media, e.g., HTTP or WebSockets. More details on this interactions with end users will be sketched in Section 2.6.

2.5 Within SOLEIL: relaying the media

As introduced in Section 2.3.2, *relay* nodes play an important role within the SOLEIL architecture. It is their responsibility, in fact, to make sure a stream injected at a *root* node is made available at the lower levels, and down to the *leaf* nodes which will eventually provide the wide audience of end-users with the stream itself. While it's true that the main purpose of these *relay* nodes is to widen the width of the tree and increase scalability by simply

relaying media hop-by-hop across the levels, though, that's not their only responsibility.

It is important to recall, in fact, that the multimedia flows being forwarded by means of RTP can be associated with a control channel by means of RTCP. This protocol provides invaluable information associated with the media delivery, specifically with respect to statistics related to the effectiveness of the transmission, packet loss, jitter and so on. Considering that all the connections *relay* nodes establish are hop-by-hop, so that they receive a flow from their father and then forward it to their children, the availability of such a control channel becomes of paramount importance in order to evaluate the performance of each hop, and possibly intervene in case a broken or faulty media leg is affecting the performance of the platform as a whole. In fact, such information may even be forwarded, along or using out-of-band mechanisms, to the other nodes of the architecture, in order to isolate issues and allow for a dynamic recovery.

That said, when envisaging a topology with several different relay nodes communicating with each other, a blind forwarding of such feedback and statistics messages can be counterproductive, due to the potential waste of bandwidth and resources that may result from the increasing RTCP traffic. Besides, RTCP statistics mostly make sense only on the leg they refer to, and so would be useless on other hops if not to make other nodes aware of an ongoing issue. As such, we decided to involve, as part of the *relay* nodes logic, some form of aggregation of these statistics. This would allow all nodes to still be aware of potential issues in other areas of the SOLEIL network, but by means of synthetic information: drill-down information would still be available by interrogating the affected nodes directly.

The way we designed this was by intercepting the so-called Receiver Reports (RR) in the RTCP messaging. These messages, as specified in [7], allow a media recipient to provide the media source with information related to the reception of the multimedia flows, including packet loss, jitter, delay and others. This means that a *node* relay can, this way, inform its father of po-

tential issues with the reception of a specific flow. Since each *relay* node is both a child of a father node, and in turn father of other children nodes, such information can then be associated to feedback coming from adiacent nodes, and more in particular by the creation of aggregated statistics that take into account a weighted average over the node's descendants. Finally, these aggregated statistics can be sent through the SOLEIL protocol to a separate component, e.g., a round-robin database to only store values for a well-defined period of time, that can be assigned the task of evaluating them and, in case, react accordingly. This database can then in turn be exploited by the *master* node in order to assess whether or not any recovery action is needed.

2.6 At the edge of the topology: the "last mile"

The previous sections introduced the SOLEIL architecture and the several different roles that can be envisaged within its framework. Specifically, a key role is placed in the hands of the so-called *leaf* nodes of the tree, which on one side are responsible for receiving the streams to broadcast from the upper levels and layers (the *relay* nodes) and on the other have to take into account the different technology required to actually distribute the streams in a way the end-users will be able to exploit. For this reason, this last level of the distribution tree can be seen as the "last mile" of the SOLEIL architecture, with very specific targets and requirements that substantially differ from what is needed within the SOLEIL distribution network. As anticipated in this chapter, this is what I focused my doctoral activity upon, as the dynamic deployability of the architecture has been the target of my colleague Tobia Castaldi.

Considering the two different technologies employed at the two sides of a *leaf* node, the first requirement that can be identified is the ability to act as a *gateway*: in fact, on one side (the SOLEIL one) the node will need to be able

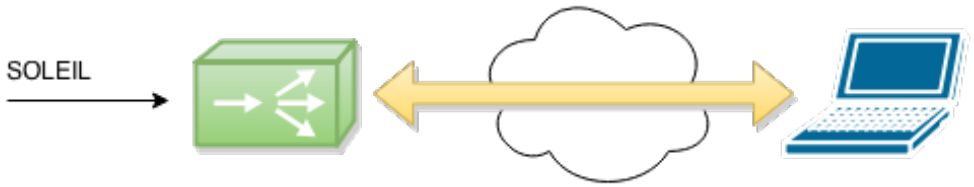


Figure 2.10: SOLEIL architecture: the "last mile"

to interact using plain RTP/RTCP and, more in general, the semantics expected of *relay* nodes; on the other side, instead, this node needs to be able to interact with viewers using the web technologies they'll have available. As anticipated, we identified as a key technology for the purpose the WebRTC standard, which, although still under definition, is already in a more than usable state and currently exploited in several commercial frameworks and applications. The WebRTC suite, which will be described in detail in Chapter 3, allows for the setup of secure real-time multimedia channels in a dynamic way among two peers: this means that a *leaf* node will need to be able to act as one of these peers, in order to negotiate and establish a communication channel with interested viewers, make sure connectivity can be established in a reliable way, take care of the creation of a secure channel and, last but not least, feed the viewer with the streams they're interested in in a timely way, by properly "translating" from the technology employed on the SOLEIL side, if needed.

As it will be explained in further detail in the following chapters, WebRTC does make use of an extended set of the RTP/RTCP suite, which means that in part the translation process from the SOLEIL side is indeed made easier and more lightweight. That said, not everything is that easy, and Chapter 3 will idenfity the key issues we had to face with respect to the WebRTC suite.

Figure 2.10 presents a zoomed-in overview of a single *leaf* node within a SOLEIL architecture, which in turn highlights the role of the translator such a node needs to employ and its requirement to implement a final distribution tree for the interested viewers. Specifically, a stream is receiced by the SOLEIL infrastructure: in order to make this stream available to a viewer,

an ad-hoc communication channel needs to be established through a network that may make this hard to accomplish (e.g., in the presence of Network Address Translators or Firewalls), and specific protocols need to be involved in order to setup and properly mantain the channel and the transmission.

Chapter 3

WebRTC media components

What is commonly called "WebRTC" is actually the joint effort of two of the largest standardisation bodies of the Internet ecosystem: the Real-time Communication Between Browsers (WebRTC) [5] in the World Wide Web Consortium (W3C), and the Real-Time Communication in WEB-browsers (RTCWEB) [6] within the Internet Engineering Task Force (IETF). These two standardisation bodies, which have worked so far on pretty much every standard technology that is currently used on the Internet, are working together on separate aspects of "WebRTC". Specifically, while the W3C is working on aspects related to the presentation and application sides of the technology (multimedia tracks, JavaScript APIs, hooks in the browsers, etc.), the IETF is instead responsible for everything that goes on the wire (protocols, signalling and negotiation, security, etc.). The technology as a whole and the related suite is what is synthetically WebRTC.

The main purpose of the standardisation efforts was to design a standard suite of protocols and APIs to allow for the realization of real-time multimedia scenarios within the context of web browsers, and without any need for proprietary or ad-hoc plugins. This included a programmatic and secure access to media devices (microphones, webcams, screens, etc.) that could be used for establishing a peer-to-peer communication among parties interested in interacting with each other. In order to do so, the W3C and IETF have agreed that a partial re-use of the legacy standard Voice over IP infrastruc-

ture would be beneficial, especially considering the ongoing efforts that are still taking place in that respect in the IETF. Specifically, it was agreed that the foundation for the multimedia delivery would be the Real-Time Transport Protocol (RTP) [7] in its Secure version (SRTP) [10]. To take into account the several different requirements in terms of quality of experience, security, privacy and effective connectivity establishment, several extensions have been proposed among pre-existing solutions and new efforts. Most of these requirements, which are specified together with typical use cases in an ad-hoc standard document [11], the related proposed extensions will be introduced in the next sections.

Considering the network-oriented nature of this doctoral thesis, more relevance will be given to the efforts devoted by the IETF on this, especially considering my involvement as an active contributor at the IETF in several Working Groups.

3.1 The WebRTC Protocol Suite

Several different requirements were identified when working on a first draft of the specification. Specifically, several different aspects were considered, namely:

- Signalling and Negotiation;
- Connection Establishment and NAT Traversal;
- Media Transport and Control;
- Quality codecs;

and so on. Most of the above mentioned challenges are typical of any VoIP infrastructure. Signalling, for instance, allows peers to indicate their willingness to setup a media communication with someone else, while the negotiation process allows them to agree on a shared set of features to actually setup the communication channel (e.g., codecs to be used, network-related

information, etc.). At the same time, once negotiated the communication channel needs to be physically established: this can at times be a troublesome process, especially whenever Network Address Translators (NAT), Firewalls or other network filters are in the path between the two parties, although we'll see there are standard approaches that can help cope with those. Once a communication channel is ready, there needs to be an effective and secure/private way to transfer media packets in a timely way: this is what the media transport requirement is for. Besides, since the transport used will be unreliable to minimise the impact of latency and congestion over the communication, there also needs to be some way to control the delivery, in terms of both statistics and pro-active feedback the peers can make use of to evaluate the quality of the transmission. Finally, these media packets need to be encoded someway, possibly using codecs that have been specifically conceived for a usage in the Internet, and as such account for a dynamic management of the available bandwidth.

As anticipated, most of the above mentioned requirements are normally part of any VoIP specification, be it proprietary or not. From a standardisation point of view, a lot of efforts have been devoted to such issues within the Real-Time Applications and Infrastructures (RAI) area in the IETF. This is where, for instance, the Session Initiation Protocol (SIP) [12] protocol was born, and where all the related specifications including the above mentioned RTP/RTCP came from. In order to re-use part of these efforts, the RTCWEB Working Group decided to base the WebRTC specification on top of some of them, in order to have a valid foundation they could expand as needed.

The following subsections will go through all of the mentioned requirements and how they were addressed from a standardisation/specification point of view, highlighting the related challenges when it comes to actually try and implement them within a WebRTC compliant implementation, which as discussed was the main target of the "last mile" of the SOLEIL infrastructure.

3.2 Signalling and Negotiation

As anticipated, among the main operations a technology like WebRTC has to take into account are signalling and negotiation. While the former, as the name suggests, basically "signals" the intention by one of the involved peers to initiate a multimedia communication with somebody else, the latter takes care of allowing the involved parties to agree on whatever is needed to get the multimedia connection to work.

Within the IETF, there are two protocols that have been defined with exactly those target in mind: the Session Initiation Protocol (SIP) [12] handles the signalling requirement, while the Session Description Protocol (SDP) [13] takes care of the negotiation process. The two protocols, while independent of each other, are nonetheless quite intertwined when used within the context of standard VoIP. Specifically, whenever a multimedia session needs to be established after a SIP session has started, SDP payloads are exchanged within the context of SIP messaging. This usually happens within the framework of the so-called "three-way handshake" that is part of the SIP/SDP specification: in its simplest form, a user initiates a SIP session by sending a SIP INVITE to a different user, and attaches an SDP payload to provide its own part of the negotiation; the user receives the SIP INVITE, and decides to accept the setup of the session by replying with a SIP 200 OK and providing its own SDP payload; the SDP payloads are then matched by the two users as part of what is called the "offer/answer" pattern, in order to allow both users to make sure there is a shared set of codecs that can be used, that both parties can actually establish a network communication with each other, and so on. This eventually leads to the setup of a multimedia channel that the involved parties can use to exchange media packets with each other. Such an approach is depicted in Figure 3.1. For the sake of simplicity, the typical SIP trapezoid is omitted, which means the caller and callee are assumed to be in direct contact with each other: in a more general scenario, each user would refer to its own SIP proxy, and their respective proxies would relay the messages among each other while keeping the media trans-

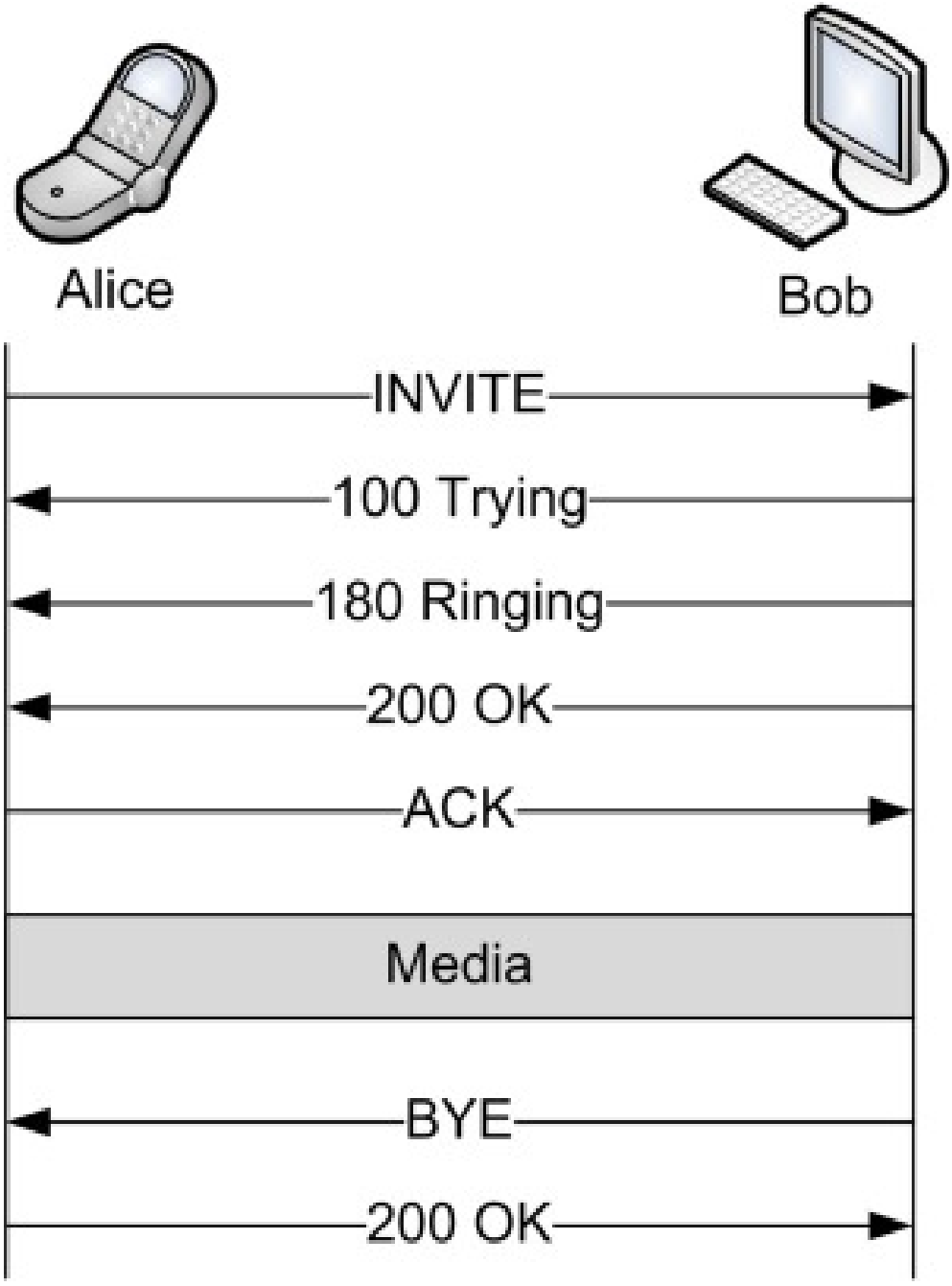


Figure 3.1: SIP call flow diagram

fer peer-to-peer. As the figure suggests, the caller, Alice, decides to start a multimedia session with Bob, the callee. To do so, she makes use of a SIP INVITE message, which is the method defined in the SIP signalling to originate a session. This eventually results in Bob accepting the call by means of a SIP 200 OK message, and to a media channel being established between the two. Eventually, Bob decides to hangup the communication, and makes use of the related method made available by the signalling protocol, that is a SIP BYE request.

3.2.1 Session Description Protocol (SDP)

The steps analyzed in the previous section describe a quite standard behaviour, and more or less all VoIP technologies, standard or not, follow a similar approach. WebRTC does not make a difference, here, and in fact, in the early stages of the standardisation process, it was decided to keep on relying on SDP as the foundation for the negotiation part. As to signalling, instead, a different approach was chosen. Specifically, while choosing SIP could have seemed a no-brainer solution, considering how widespread it is and how it would have allowed for an easier backwards compatibility management, it was eventually decided not to mandate it or, actually, any specific signalling protocol whatsover. In fact, while initially some WebRTC authors tried to specify a very simple JSON-based protocol called RTCWeb Offer/Answer Protocol (ROAP) [14] for the purpose, and others tried to foster the adoption of an existing protocol like SIP instead, it soon became clear that the adoption of a specific signalling protocol would have been quite limiting to WebRTC. The reason for this was a desire not to put excessive contraints on what had been conceived since the beginning as a web technology and, as such, inherently dynamic and flexible enough to cover heterogeneous scenarios: in fact, while SIP is quite effective for VoIP-based scenarios, it has some issues whenever more innovative scenarios need to be tackled, something that WebRTC advocates had in mind since day one. Since other protocols like Jingle, the Inter Asterisk eXchange (IAX), H.323, etc., or ROAP itself, all

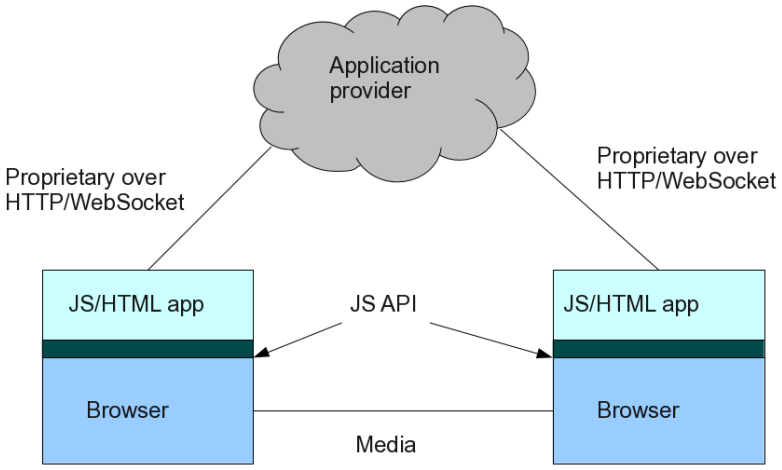


Figure 3.2: The WebRTC trapezoid/triangle: no specific signalling

had pros and cons, it was eventually decided to keep the WebRTC specification completely agnostic with respect to signalling, leaving the choice for a signalling protocol to use to web developers, and only mandate a common protocol for negotiating instead, that is SDP.

3.2.2 Javascript Session Establishment Protocol (JSEP)

That said, some form of signalling is still required, which means that it's up to the web application to take that requirement into account, e.g., to exploit existing signalling protocols for the purpose or to design new ones. One way or the other, signalling will always travel across one or more web servers in order to allow two or more parties to setup a WebRTC connection: this means that WebRTC is still based on one of the base concepts of the SIP specification, that is the so-called *trapezoid*. Figure 3.2 depicts a WebRTC trapezoid where the servers involved for the communication setup

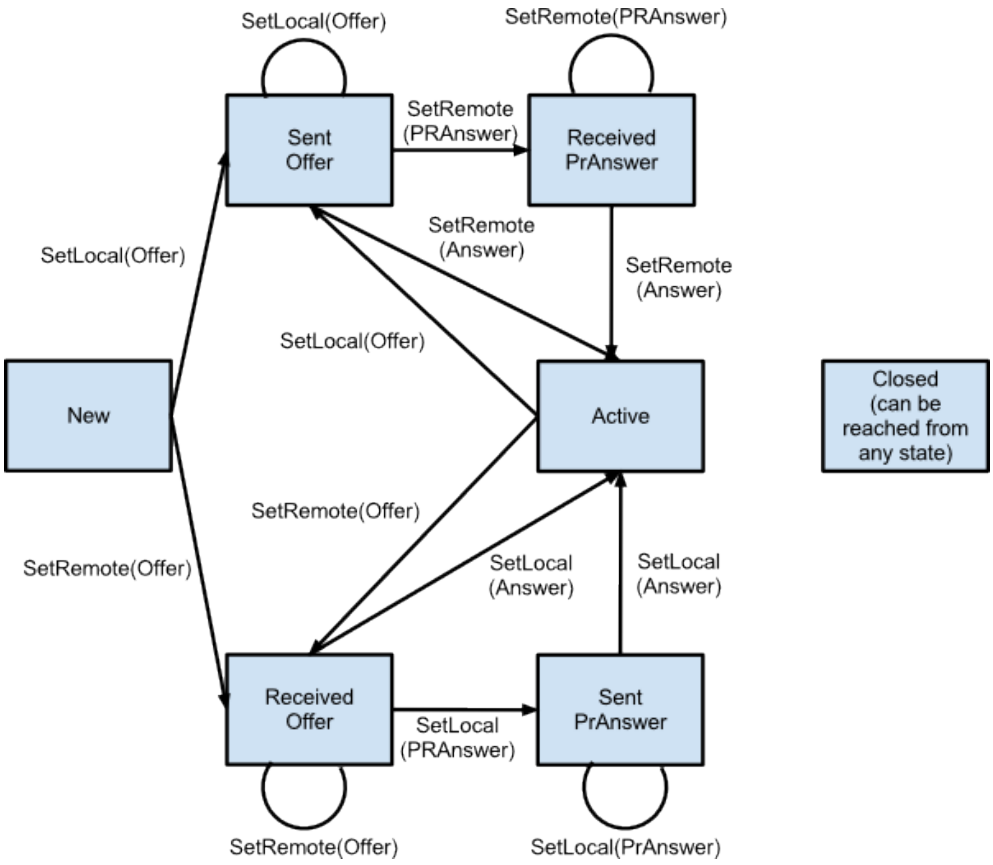


Figure 3.3: JSEP: peer state transitions

are actually colocated, which means the trapezoid becomes a triangle. Both parties who want to communicate refer to the same web server (e.g., the one hosting the WebRTC application), and it's through that web server that the signalling, whatever protocol is exploited, will be exchanged. What's important to point out is that SDP payloads will be used to negotiate the WebRTC media channel, which will eventually result, as it will be clearer in the next sections, in a peer-to-peer multimedia connection, meaning that the web server which facilitated the signalling and negotiation process will not be involved anymore.

Of course, in order to allow web developers to implement their own preferred approach to signalling on top of SDP, it soon became clear that some standardised way to keep the state would have been needed, and the choice

came to rely on a standardised JavaScript API for the purpose. The main debates over this API revolved around the level of depth it should provide: specifically, whether or not this should be a complex, low-level API allowing web developers to go very deep in the setup and management of media communications, or whether it should be a much simpler high-level API instead, with basic hooks and wrappers that would masque the complexities of the media management. While the former was clearly advocated by developers and researchers with a pre-existing background in VoIP and multimedia applications, the latter was fostered by the web world instead, which didn't have the same familiarity with such a complex ecosystem. In order to keep both sides happy, the decision was to eventually design a middle-ground API, something that could be simple enough to use, while at the same time still be able to allow for more complex scenarios to be covered. Such an API was called Javascript Session Establishment Protocol (JSEP) [15], although it is not a real protocol but, as anticipated, just a standardised mechanism for keeping and managing the signalling state.

JSEP was specifically conceived to allow web developers to create and manage local and remote SDP payloads, in order to allow for an automated creation and management of the so-called PeerConnections, that is the abstractions of the multimedia channels that take care of the media transfer. Since JSEP works on top of SDP, it is very easy to make use of it as the foundation for any kind of signalling protocol, which means it can be used in conjunction with existing protocols like SIP, ROAP, Jingle and others, or new proprietary protocols developed just for the purpose. In case the signalling protocol does not make use of SDP directly, it is up to the developer to take care of the required translation to and from SDP in order to comply with JSEP.

Figure 3.3 describes what the states JSEP handles could be and how they could transition from one to another according to the events occurring over the state machine. For instance, a new PeerConnection may be handed a local SDP, representing the user's side of the negotiation, and transition to a

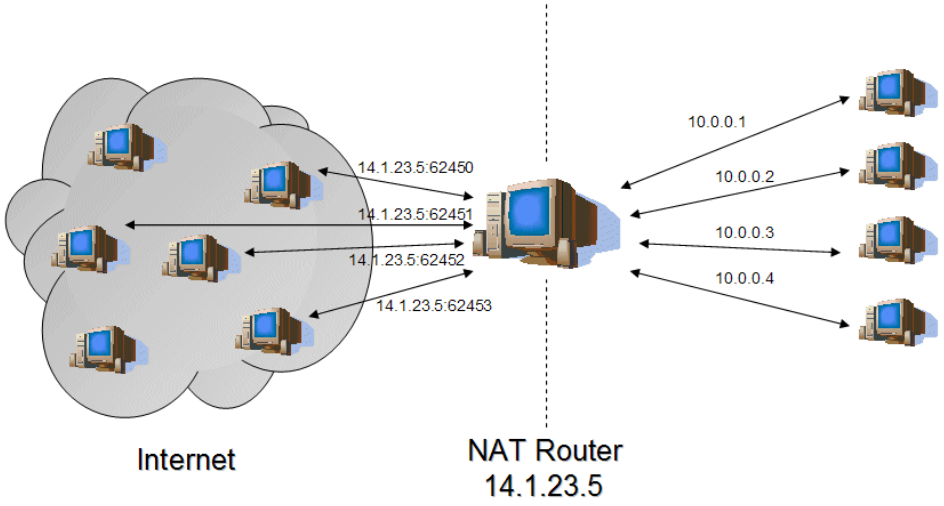


Figure 3.4: Network Address Translators

state that is expecting the remote SDP to be available as well. As soon as it is and it is passed to the stack, the state machine can transition to a different state, e.g., active in case everything was properly set up and it's time to try and actually establish a communication channel.

3.3 Connection Establishment and NAT/Firewall Traversal

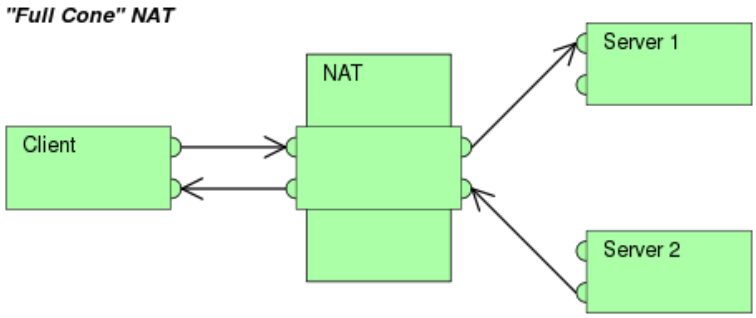
Section 3.2.1 described how the signalling and negotiation requirements were addressed, partially or not, within the WebRTC specification. As anticipated, though, these steps only refer to the preliminary setup of a multimedia session: specifically, they allow two or more parties to decide to start a multimedia communication with each other, and provides them with the means to come to an agreement about how this communication should take place.

A very important step, though, is actually establishing such a media communication. While this step may be given for granted, as it is often assumed that two endpoints in the Internet should be able to always be able to in-

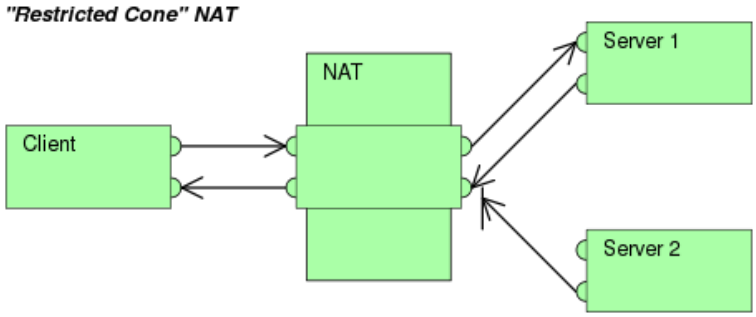
teract with each other somehow, this is often not that easy, as there can be several components across a network that may make this step much harder to accomplish. It is the case, for instance, of Network Address Translators (NAT) [16] or Firewalls, network components that, willingly or not, can interfere with the normal means for communicating and, in some circumstances, even prevent such a communication entirely. NATs in particular, although they don't filter any packet and as such should in principle not be an obstacle to communication, can actually be troublesome whenever multimedia interactions are involved. In fact, their purpose is to allow a series of machines on a private network to all share the same public address: an example of this is depicted in Figure 3.4, where machines in the Internet can actually interact with different machines in a private network through a NAT. In order to do so, a NAT has to translate the addresses for every incoming and outgoing request, so that to make sure, for instance, that a response to a request originated by a specific machine in the private network can reach the exact destination although the same public address is used.

While this normally works fine in normal web-based interactions, it is particularly problematic within the context of multimedia communications, which are dynamic by nature and as such cannot be easily provisioned by a network administrator. Specifically, the approaches described in Section 3.2.1 often refer, within the context of the respective protocols, to IP addresses and ports that may only be valid in a private network, and mean nothing outside of the context of those "walled gardens". This would result in broken information made available to the involved parties, as either peer in the communication may be provided with addresses to set up a communication that would actually not be reachable at all, e.g., because they're private addresses of a LAN or because a restrictive firewall is filtering the communication.

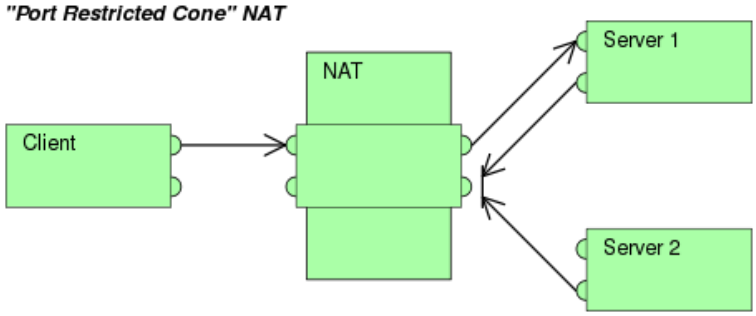
Besides, there are actually different type of NATs deployed in the network, each of which can have a slightly different behaviour when it comes to translate addresses, and which can result in different problems that need different care. Figure 3.5 highlights the different type of NATs that could be



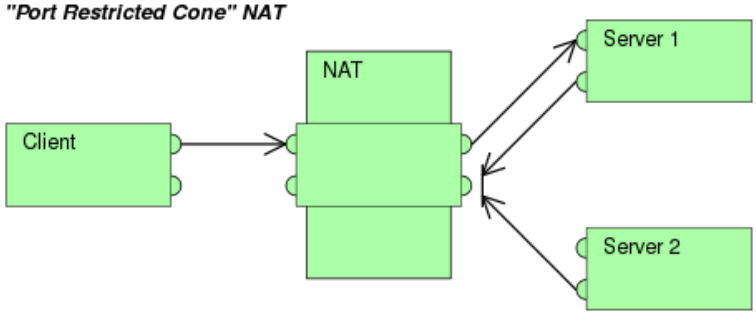
(a) Full Cone NAT



(b) Restricted Cone NAT



(c) Port-Restricted Cone NAT



(d) Symmetric NAT

Figure 3.5: Different type of NATs

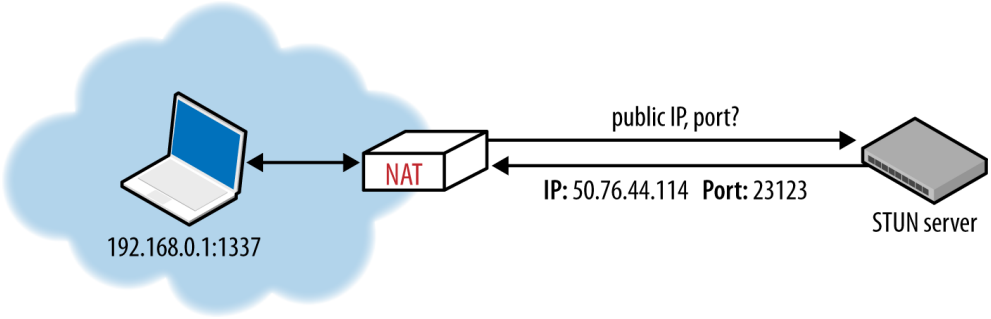


Figure 3.6: Session Traversal Utilities for NAT (STUN)

deployed in a network, from the least problematic one, the Full Cone NAT, to the most troublesome of them all, the Symmetric NAT. This is not a new issue within the context of multimedia communications, and the IETF itself has for a long time worked on solutions for these problems. In particular, a suite of protocols have been designed to address these and other issues in an incremental way, in order to maximise the chances of eventually getting a working communication channel.

3.3.1 Session Traversal Utilities for NAT (STUN)

The first attempt to solve the issue of NATs in real-time communications was a protocol called Session Traversal Utilities for NAT (STUN) [17], which was designed exactly to tackle this kind of scenario. Specifically, as depicted in Figure 3.6, this protocol allows a user to get information about what address is seen outside of the context of a LAN: this information can then be involved within the signalling and negotiation mechanism in order to provide the peers with correct and usable information.

This protocol works under a simple assumption. All NATs implement a mapping for outgoing requests, which means that, for a request originated from a specific private address and port, the NAT will map the address to another address in the NAT public interface before relaying the request to the actual recipient. This is needed in order to allow the recipient to reply to the actual sender: in fact, all the recipient will see is a public address, the one of the NAT mapped, and has no information about the sender actually

being in a private network, just as it has no access to the private address the sender has, which would have no meaning outside the boundaries of the private network itself. Since the NAT mapped the two addresses, the recipient can then simply send a response or a new request to the original sender by sending it to the public address it received the first message from. It will be the NAT's responsibility, then, to inspect its mapping table, and forward the request to the correct machine in the private network, at the right private address and port.

This means that, if a user willing to setup a multimedia session wants to be reachable at a specific private address, it needs to know what mapping the NAT will create for it. To do so, it can ask an external component what the public address that has been assigned is: this means that all the user has to do is to send a request from a specific private address to the external component, in this case the STUN server. The NAT will receive the request, create a mapping, and forward the request to the STUN server. The STUN server, in turn, will see a request coming from a public address, the one the NAT mapped, and will inform the user in a response about it. This way, the user will be able to update all the required connectivity information in both the negotiation and, if needed, signalling protocols in order to allow the other party in the communication that will need to be established to know how to reach it. This kind of approach works fine in almost all the NAT topologies, and more precisely in the presence of Full Cone NATs, Restricted Cone NATs and Port-Restricted Cone NATs.

3.3.2 Traversal Using Relays around NAT (TURN)

While STUN is indeed effective and works fine most of the times, it does not work in the presence of symmetric NATs. In fact a Symmetric NAT maps different ports when contacting different servers from the same private port: this means that, while a request to a STUN server originated from a specific port would result in a specific mapping, a different machine trying to contact that address (e.g., the actual peer the user wants to communicate with) would

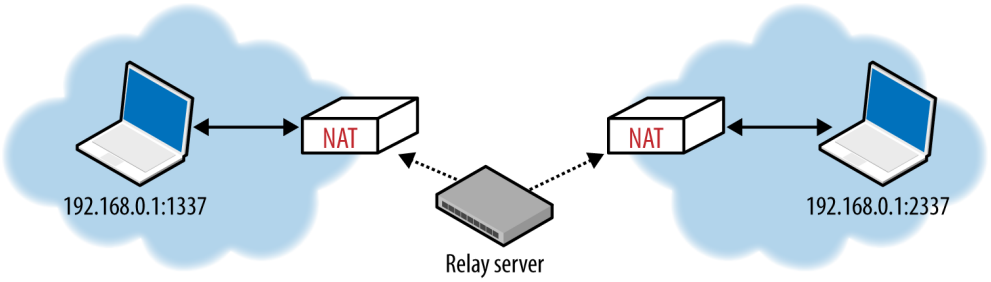


Figure 3.7: Traversal Using Relays around NAT (TURN)

be blocked. As a result, the information collected via STUN in the presence of a Symmetric NAT would be useless for the purpose of providing connectivity information. The only way to get a working connection out of a Symmetric NAT is to make sure that the same network endpoint used to check the connectivity is also used to actually handle the communication process. This is exactly what another protocol defined within the IETF standardisation efforts, the Traversal Using Relays around NAT (TURN) [18], was designed for. TURN is a set of extensions to STUN which allow media frames to be relayed through an external component, that is the TURN server. This means that, as depicted in Figure 3.7, a user can make use of the TURN server not only to get connectivity information, but also to actually act as an intermediary for the media: since the same address is used for both actions, the Symmetric NAT does not interfece with its operations. That said, since relaying media is not a cheap operation, both in terms of bandwidth that needs to be made available and of computational resources, relying on a TURN relay is usually not advised unless strictly necessary. This is not only the case of Symmetric NATs, but also of particularly restrictive firewalls. If a user, for instance, is behind a firewall that filters some specific traffic like UDP or VoIP-related, a TURN relay may help in allowing the user to circumvent the filters and still be able to communicate through it as a tunnel.

3.3.3 Interactive Connectivity Establishment (ICE)

The previous subsections described how different protocols can help in getting a multimedia session up and running, according to the different network topologies that may be in place and, willing or not, make this harder to accomplish. Specifically, STUN and TURN are key in this process, as they both allow to get things working in some specific scenarios. That said, it's in general quite hard to programmatically figure out the scenario one is in, and as such it's not always clear to understand what the right steps to follow are.

In order to properly orchestrate the STUN/TURN processes, the IETF also devised a standard mechanism called Interactive Connectivity Establishment (ICE) [19] which defines the required, incremental steps needed in order to allow two or more peers to gather and exchange information about their connectivity details, and then try all of those in a sequential way to eventually come to a working pair that allows for a communication. Since ICE is quite effective and commonly deployed in existing VoIP infrastructures, it was also chosen as the preferred and mandatory way to establish connectivity within the contect of the WebRTC specification.

Section 3.7 will provide some details about an extension to the base ICE mechanism, called *Trickle ICE*, which allows for a much faster and more reliable approach to connectivity establishment.

3.3.4 An open area for research: TURN over WebSockets

As anticipated in the previous section, a TURN relay can be helpful in traversing restrictive components like firewalls. In fact, especially if a firewall is filtering ports or protocols that may be vital to setup a multimedia session, the only way to get a working connection to transport the media would be some sort of tunnel that is not affected by the filters.

The way this is usually accomplished within deployments is to indeed rely on a TURN implementation, but properly configured in order to look like a

different protocol. For instance, a TURN server may be configured to listen on ports 80 and/or 443: this means that, in the eyes of a filtering component, a request addressed to a TURN server may actually look like a simple HTTP request addressed to a web server, that is something that almost no firewall tends to block. Other deployments may try and configure a TURN server to listen on port 53 instead, to trick network filters into thinking the media traffic is actually DNS traffic instead.

While in principle these approaches tend to work fine, they're not always one hundred percent reliable. Specifically, network filters implementing some form of Deep Packet Inspection (DPI) may still recognise the media traffic for what it really is, despite the unusual ports being exploited for the purpose, with the communication attempts being shut down as a result.

Starting from these assumptions, and from an effort I worked upon on a related subject [20], I recently contributed to a proposal within the IETF community to define a new transport for TURN. Specifically, the document my co-authors and I wrote tries to specify a way to transport TURN messages over a WebSocket connection [21], rather than using a transport protocol like UDP or TCP directly. This approach would have several benefits: (i) it would start as a WebSocket upgrade from an HTTP session, meaning the context of the communication would indeed be an HTTP one, and no masquing of the intentions would be needed; (ii) it would be simple to implement, as apart from the initial handshake a WebSocket connection acts exactly as regular sockets, although being message-orianted which is actually a plus; (iii) it would be network-administrator friendly, since the WebSocket subprotocol mechanism would allow media communications to be easily identified, if needed, and as such specific rules to allow them while keeping other filters in place would be easy to implement.

That said, the proposal is still in its early stages, and has so far not gathered a complete consensus within the context of the IETF discussions. Work on this will procede nevertheless, as we do believe this would be an invaluable instrument in making WebRTC more easily accessible even in

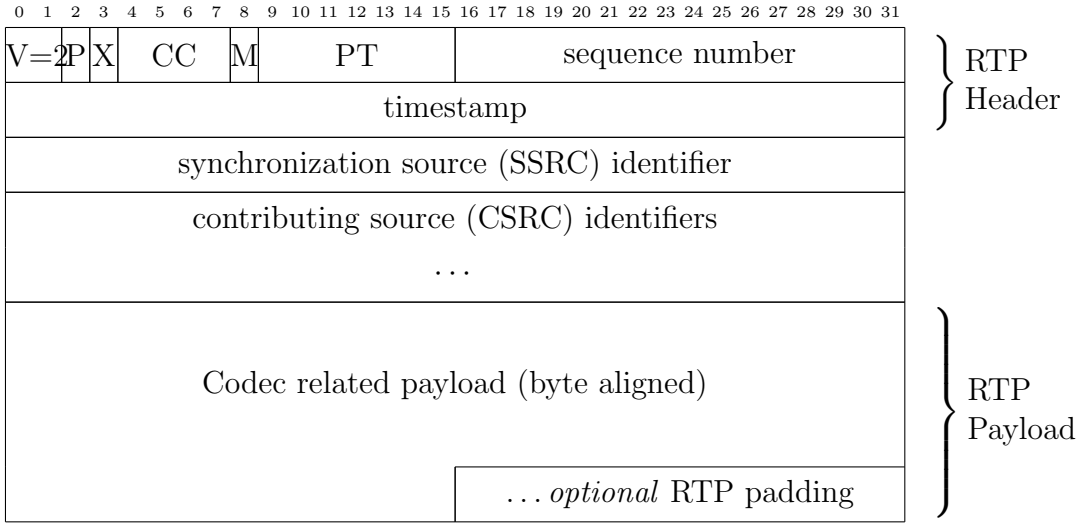


Figure 3.8: Real-Time Transport Protocol (RTP)

edge-case scenarios and topologies where it doesn't work today.

3.4 Media Transport and Control

Once a media channel has been negotiated and, thanks to ICE, physically established among two or more peers, media is ready to be exchanged. In order to do so, a proper transport protocol that takes into account the real-time requirements of the media exchange is needed, possibly with the aid of something that also monitors the effectiveness of the delivery and ideally provides feedback that may be used to improve it.

3.4.1 Real-Time Transport Protocol (RTP)

In previous sections we already anticipated that the WebRTC specification relies on the well-known RTP/RTCP suite for the purpose. RTP and RTCP are commonly used within all SIP/SDP and other standard infrastructures, which means that they constituted the obvious choice for covering the same requirement in WebRTC as well. The common format of an RTP packet is depicted in Figure 3.8, which highlights some key pieces of information that allow for a synchronised delivery and rendering of multiple media streams, in-

cluding the *timestamp*, the *sequence number* and the *synchronization source identifier* (SSRC). While the first two values provide timing related information, respectively related to when a media packet should be reproduced and to the actual ordering of the incoming media packets, the last value is very important to identify a specific RTP session. As it will be clearer in the next subsection, this value in particular needs special care whenever different RTP sessions are bridged, as will be the case of the SOLEIL architecture, especially at the *leaf* nodes.

That said, the WebRTC specification identified several additional and important requirements, not all of which were covered by the base RTP/RTCP documents. Specifically, two of the key requirements that were identified were *security* and *privacy*, that is the ability to establish a secure RTP/RTCP channel that would prevent eavesdropping or manipulations on the exchanged media. This requirement will be dealt with in Section 3.5. On the other hand, a great relevance was given to other key aspects like the ability to provide timely feedback on the communication experience.

3.4.2 Real-Time Transport Control Protocol (RTCP)

RTCP was at first designed as a "sister" protocol to RTP with exactly those targets in mind: a protocol that could be used in conjunction with RTP to monitor the streaming sessions and provide statistics related to the media transfer. Figure 3.9 presents a view on the common header of RTCP messages, which can indeed be quite specialised as it will be clearer in the following sections and chapters.

The base mechanisms for providing statistics are the so-called Sender Reports (SR) and Receiver Reports (RR), which both provide the respective parties with information on the media transmission, e.g., in terms of lost packets, jitter, and so on. Nevertheless, RTCP was conceived as an extensible protocol, meaning several different additional messages have been defined as well, both in terms of feedback and pro-active requests (e.g., actively requesting key frames in a video session).

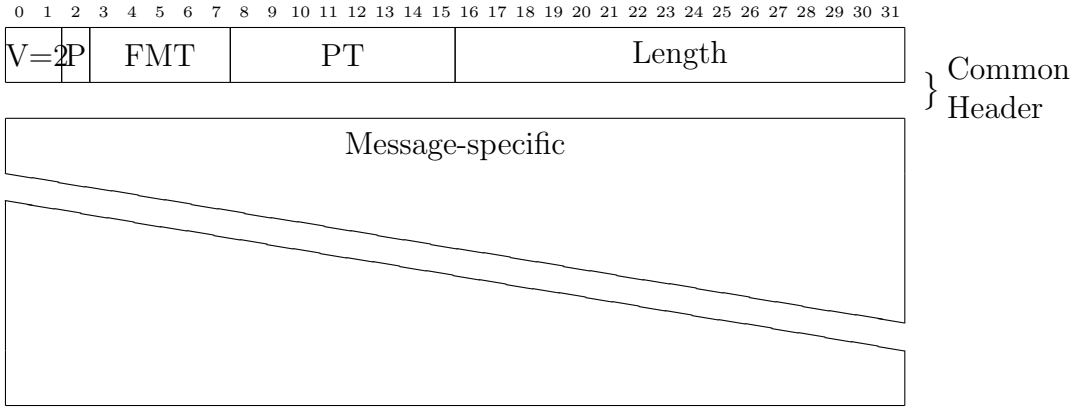


Figure 3.9: Real-Time Transport Control Protocol (RTCP)

For WebRTC, specifically, some specific requirements were identified, in order to allow for a high quality and nice experience for end users. While there already existed solutions, within the IETF, to cover those requirements, not all of those are currently very widespread, while some others were completely missing. In order to properly discipline the RTP/RTCP-related requirements for WebRTC, the IETF worked on a document to specify what aspects of the RTP/RTCP specification were mandated and how in the *rtp-usage* draft [22]. This document, in particular, highlighted the importance of RTCP feedback [23], e.g., NACK, PLI, FIR [24], REMB [25], which are often ignored within the context of traditional VoIP.

Anyway, rather than going through all these requirements, it's worthwile to highlight a different aspect that needs special care, especially within a bridging context as the one envisioned for the SOLEIL "last mile". In fact, as anticipated in Section 2.3.3 and Section 2.6, a *leaf* node in the SOLEIL architecture is assumed to act as an intermediary between heterogeneous technologies, specifically between a plain RTP/RTCP feed coming from the *relay* nodes and the related WebRTC-based communication channels on the viewers' side. As explained, this is only partially an issue, as WebRTC makes use of RTP/RTCP as well, although in an extended and enhanced fashion. Nevertheless, considering that even in the simplest approach different RTP sessions will be bridged, special care must be given to the related SSRC

information in each of them. Specifically, whenever an SSRC or RTP-related value is modified across two separate RTP sessions, both sessions need to have both the RTP and RTCP related values updated accordingly in order to make sure no part of the communication or control is broken. This is exactly the main target of an IETF specification I co-authored in the STRAW Working Group, called "Guidelines to support RTCP end-to-end in Back-to-Back User Agents (B2BUAs)" [26]. This specification, as the title suggests, provides guidelines for components that act as intermediaries in an RTP-based communication, and is especially useful within the context of WebRTC-based implementations as the one we're interested to design. At the time of writing, it's close to becoming an official standard specification as an RFC.

Section 3.7 will provide additional details on some more advanced enhancements that were added to the WebRTC specification, namely the usage of *BUNDLE* and *rtcp-mux* as ways to reduce the usage of network resources and increase the chances of getting a successful connection establishment.

3.5 Security Extensions

The previous section hightlighted the key role of RTP/RTCP and its extensions to cover the media transport and control requirements in WebRTC. It was also introduced, though, that a key requirement in WebRTC was to also provide the proper mechanisms to not only establish a functional mechanism for exchanging media packets in a timely way, but a secure and private one as well, in order to prevent attacks on the media like eavesdropping, media manipulation, man-in-the-middle attacks and so on.

3.5.1 Datagram Transport Layer Security (DTLS)

Although the requirement of *security* and *privacy* by itself was never really under discussion, debates on the best way to provide them proved much more challenging. In general, RTP/RTCP can be easily secured by making use of its secure extensions, the Secure Real-time Transport Protocol (SRTP) [10].

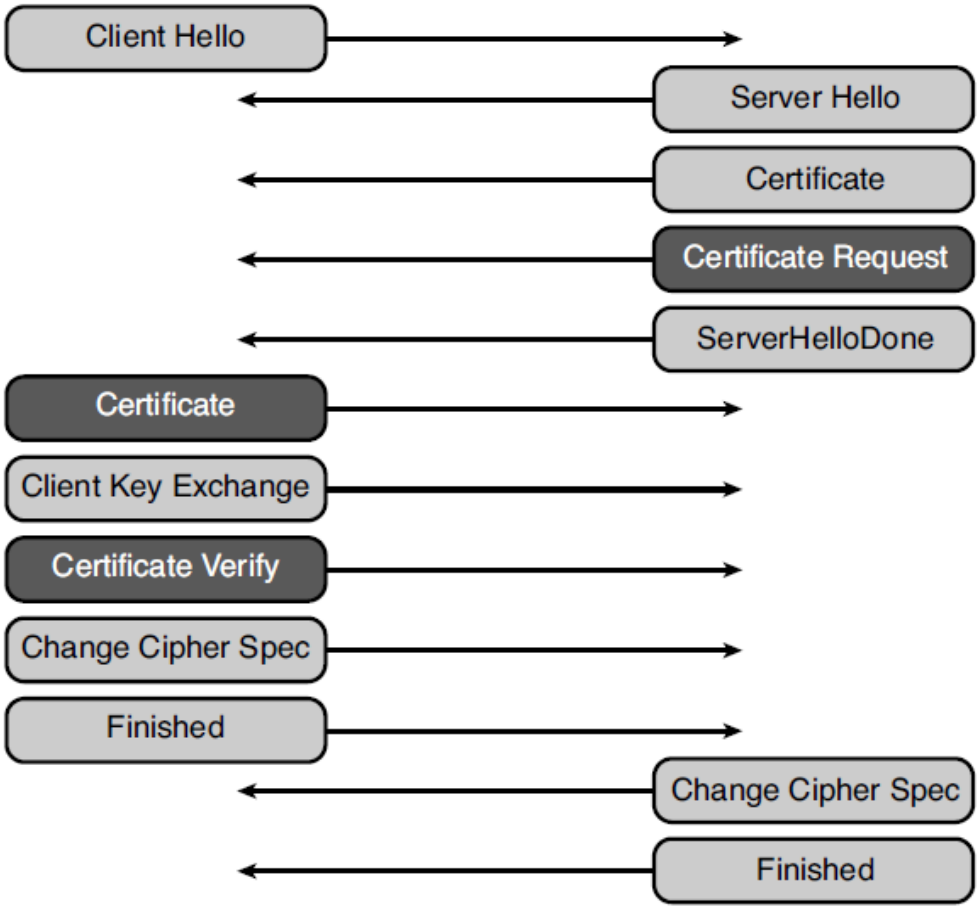


Figure 3.10: Datagram Transport Layer Security (DTLS)

Considering the cryptographic nature of these extensions, though, there are different approaches that can be used to exchange the related information to actually secure a session. Specifically, there are two main approaches to this: (i) the Session Description Protocol Security Descriptions (SDES) [27], which exchanges the key-related information within the SDP in the negotiation process, and (ii) Datagram Transport Layer Security (DTLS) [28], which instead exchanges those keys end-to-end through the media communication channel, and only makes use of the SDP to advertise the certificate fingerprints. Since day one, a strong debate was born on which of the two approaches should be mandated in the WebRTC specification, giving for granted that media security in general had to be mandated anyway. Advocates of SDES insisted on the much wider deployment in existing VoIP frameworks of this approach, explaining how DTLS-SRTP had basically never been used in VoIP infrastructures before. On the other hand, DTLS advocates instead pushed on the much weaker security that SDES provides with respect to DTLS, since an out-of-band exchange of cryptographic information is much more prone to manipulation and attacks. Eventually, DTLS won the fight and SDES was explicitly ruled out of the specification, which means that all WebRTC-compliant implementation now must only implement DTLS-SRTP to provide media security. While an agreeable solution in terms of effective protection, this proved a challenging requirement on WebRTC implementations, since DTLS-SRTP is a very recent specification and as such almost never deployed in the field.

3.6 Codecs

An important aspect in a multimedia application, apart from the protocols to allow for a transfer of media packets, is how these media are actually going to be encoded. In fact, in order to allow two or more parties to communicate with each other, they all need to have at least one codec in common to exchange media information, or otherwise they'd be talking different "languages". This applies to both audio and video.

While SIP/SDP and other signalling/negotiation protocols easily allow for a dynamic negotiation of such codecs, meaning peers can usually advertise support for multiple codecs and eventually agree on one they all share support for, it was decided within the WebRTC specification to also mandate support for some specific codecs. This was done for a specific reason, that is avoid scenarios where peers could potentially end up with no shared set of codecs, thus preventing any chance of communicating with each other. On the other hand, having at least one or a couple of codecs that all WebRTC compliant endpoints need to implement would ensure this to never happen.

As it will be explained in the next subsections, while an agreement has been reached with respect to the codecs to mandate for audio, there's still some debate regarding the related choice for video.

3.6.1 Audio codecs: G.711 and Opus

For what concerns audio, two codecs were almost immediately chosen as the candidates to mandatorily implement in WebRTC: G.711 and Opus [29].

The choice of G.711 was driven by a need to properly interact with legacy infrastructures, in particular the PSTN (Public Switched Telephone Network). In fact, G.711 is the most widely deployed audio codec in VoIP and standard telephony infrastructures, since its license expired and it's very lightweight and trivial to implement. That said, the also are some cons, as G.711 is a narrowband (8kHz) with a low compression factor, which means it has a bad quality from a user experience perspective, while at the same time consuming a considerable bandwidth.

For this reason, Opus was also mandated as an additional choice for audio. Opus is a standard codec designed and specified within the contect of the IETF standardisation efforts, and represents an evolution on pre-existing codecs like CELT and SILK, from which it derives optimizations in terms of speech compression and low latency. More specifically, Opus was explicitly conceived as a high quality Internet codec, which means it can easily adapt to the available bandwidth and update the resulting quality as a consequence.

3.6.2 Video codecs: VP8 and/or H.264

As anticipated, for video the debate was not quite as easy as the one for audio. In fact, different interests played a consistent role in the discussion, and two main camps eventually ended up advocating two different protocols: (i) VP8 [30], an open source and license free codec developed by Google, and (ii), H.264, a patent encumbered but widespread codec developed within the International Telecommunication Union (ITU) standardisation body. The discussion has so far mostly addressed the licensing concerns each camp has on the other proposal, as from a technical standpoint both codecs provide a comparable high quality experience and performance in terms of both bandwidth and resources.

Without delving into the details of a debate that prolonged for many years since WebRTC was first specified, it is worth pointing out that a temporary agreement has been recently reached with that respect within the IETF discussions: specifically, it was decided that all endpoints will have to implement both VP8 and H.264 in the future, unless one of them is explicitly and unequivocably identified as a license free solution in the forecoming future.

That said, VP8 is the only codec currently available in all WebRTC-compliant implementations, which means it has been, so far, a *de-facto* standard for developers and researchers interested in experimenting with the technology.

3.7 Advanced Functionality

The previous sections introduced some of the key requirements that the WebRTC specification identified in order to provide all the required functionality. From signalling and negotiation to the establishment of a connection and then a secure channel, several different protocols are involved in the setup and management of the lifecycle of a WebRTC PeerConnection. That said, as anticipated in some of the previous sections, not all of the available

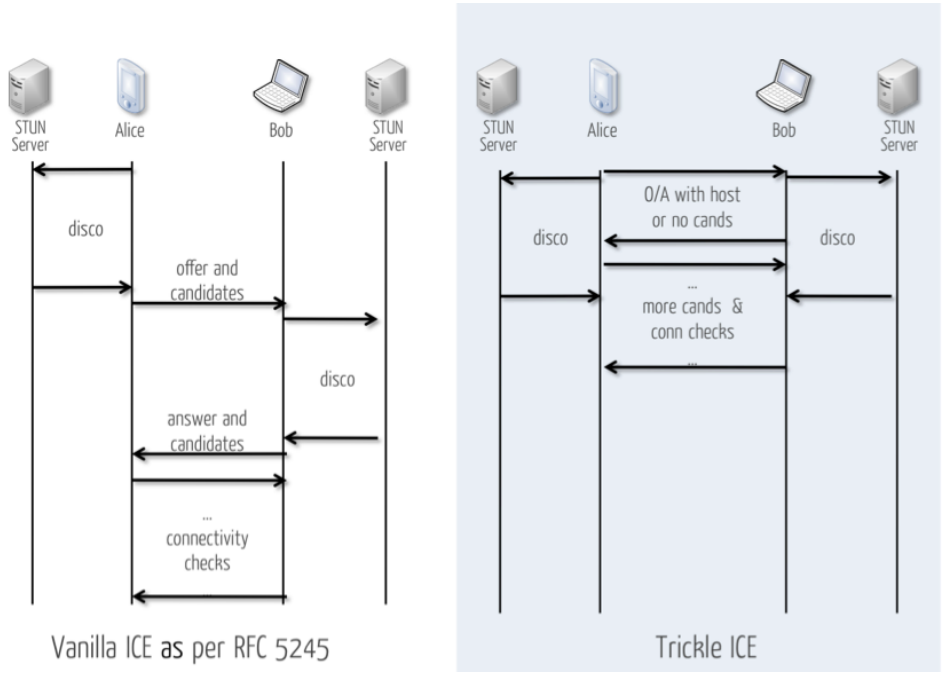


Figure 3.11: Trickle ICE compared to "Vanilla" ICE

specifications within the IETF proved to be enough to cover all the required steps, which means that some additional work needed to be done in order to address the missing "bricks".

The main enhancements the IETF introduced in the WebRTC specification were *Trickle ICE*, *rtcp-mux*, *BUNDLE*, and *Data Channels*.

3.7.1 Trickle ICE

Trickle ICE [31] is an important update to the base ICE mechanism for establishing connectivity. As explained in Section 3.3.3, ICE specifies the required steps that are needed in order to evaluate the best way to put two or more peers in contact with each other: a *gathering* of the so-called *candidates* where the peer can be reached, their *prioritizing* and *encoding* within the SDP to have them advertised on the other side, the *offering* and *answering* to have the peers negotiate the available candidates with each other and finally a *checking* phase to try and find the right candidate pairs that effectively allow for a connection establishment between the involved par-

ties. While ICE is quite effective and almost always results in a successful connection establishment, it can also be a very slow process, especially during the *gathering* phase: in fact, before the candidates can be encoded and advertised in the SDP, they need to be all gathered, which means that, if any candidate is taking more time than usual to be collected (e.g., a *srflx* candidate on a slow interface), the whole process will be slowed down as a consequence.

In order to solve this serious issue, which can result in severe delays in the setup of a communication, *Trickle ICE* was suggested as a solution. This approach, which is illustrated in Figure 3.11, starts from the assumption that you don't need all candidates to be available to start the checking phase, but that you can start with what you have and, if that doesn't work, wait for more information to arrive later. As such, the idea is that, as soon as a candidate has been gathered, it is immediately sent on the other side outside of the context of the SDP negotiation: as a result, the SDP is only used to negotiate other aspects related to the communication (e.g., ICE credentials, DTLS fingerprints, codecs and so on), while remote candidates can be received at any time and paired with the candidates that have been gathered locally in the meantime. This approach, while not one hundred percent compliant with the "vanilla" ICE specification, definitely helps in reducing the setup times, and has as a result been mandated within the WebRTC specification for all compliant implementations.

3.7.2 RTCP Muxing

rtcp-mux [32] is, instead, a different optimization that can be applied over the media transport and control channel. Namely, in general RTP and RTCP are transported over different transport protocols: this means that, in its default form, different ports and, as a consequence, connections need to be established for an RTP session and its related RTCP control channel. This makes management of the related protocols easier, as it's trivial to handle RTP packets and RTCP messages separately, since they're muxed in completely

dfferent transports. That said, this approach also introduces an unneeded waste of network resources. In fact, since two different transports are required for the two protocols, this also means that two different ports need to be allocated and negotiated within the context of a SDP offer/answer. Considering the usage of ICE for establishing connectivity, this translates to a duplicate management of candidates for handling the protocols in a separate way. While this may not sound like a big issue for a single endpoint, it definitely can be a problem when envisaging a multimedia session in a system where different applications are consuming network resources. For this reason, the IETF recently proposed a solution to this issue, by specifying a way to actually mux RTP packets and RTCP messages over the same transport, in order to have the two transports share the same "connection". This is what the *rtcp-mux* specification mandates, which are basically recommendations to properly mux and demux RTP and RTCP in order to avoid conflicts, e.g., as in incorrectly interpreting an RTCP message as a RTP packet or viceversa, which could easily lead to serious problems in the lifetime of a multimedia session. Considering the noticeable benefit in terms of network usage (half of the connections can be avoided, thus improving the setup times and resources waste), the WebRTC specification mandates support for *rtcp-mux*.

3.7.3 BUNDLE

The *BUNDLE* [33] specification tries to cover a similar problem, but related to different RTP sessions rather than to the muxing of heterogeneous protocols over the same transport. Specifically, the normal approach when negotiating multimedia sessions using the SDP standard is to negotiate separately all the media channels that need to be involved. This means that if, for instance, the multimedia session will include an audio and a video channel because the target is a video call, the audio and video channels will be negotiated separately within the context of the same multimedia session. This is normal and in principle to be expected: in fact, audio and video streaming

have typically quite different requirements in terms of required functionality or encodings, which means they should indeed be negotiated separately in order to avoid ambiguities. Nevertheless, this normally also means that a separate connection, and as such different transport-related information like ports to be used for the transmission, would be negotiated. Just as in the *rtcp-mux* case, this can easily lead to too many connections being established, especially in case several different media are going to be involved in a session. The consequence is the same, that is a potential waste of network resources and an increase of the setup times, as a different connection establishment and secure context setup is required for each involved medium, while the same connection could be re-used for the purpose, considering that all the media actually belong to the same session. This is where the *BUNDLE* specification tries to help. The specification describes a way to signal the fact that more media of different types (e.g., audio, video and/or data channels) will actually share the same transport. This is not a trivial specification to implement, as it requires implementations to be able to properly multiplex and demultiplex not only different protocols like RTP and RTCP, but also different sessions that use the same protocol. That said, the benefits in terms of used reources, especially within the context of a WebRTC implementation that has to mantain several different sessions at the same time, greatly overcome the concerns related to the *BUNDLE* complexity, which is why it has been chosen as one of the mandatory enhancements to implement for WebRTC-compliant devices.

3.7.4 Data Channels

Finally, one of the most interesting and challenging concepts introduced in the WebRTC specification are definitely *Data Channels* [34]. In fact, while WebRTC was originally conceived to allow for an easy realization of audio/video multimedia sessions natively in a browser, without the need of plugins of any sort, the specification was soon enriched with a new requirement, that is being able to also exchange generic data in real-time in a

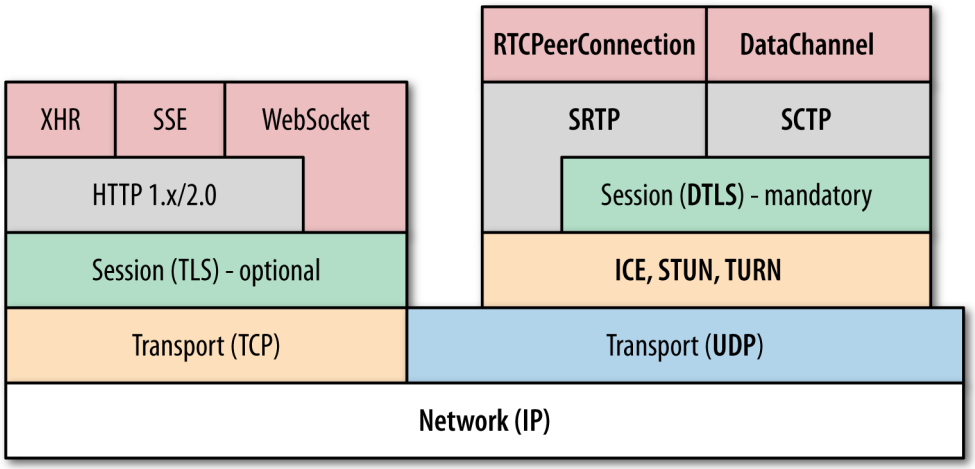


Figure 3.12: Protocols stack for Web, Media and Data Channels

peer-to-peer fashion.

This may not look like a great or disruptive change, especially considering that browsers have already been able to exchange generic data with each other for a long time, using technologies like XHR or WebSockets. That said, these approaches all rely on the presence of a server component to act as an intermediary for the data delivery, as one peer sends data to the server and the server pushes it to the desired recipient. This introduces considerable delays and is also problematic whenever privacy is a concern or a user would rather not have the server have access to the data it's sending for any reason (e.g., because the service is not trusted). *Data Channels* were thus conceived to cover this exact requirement, that is a purely peer-to-peer mechanism for exchanging data in real-time: the availability of a direct connection between the involved parties solves both the aforementioned issues, that is the excessive latency, as no intermediary is involved anymore, and the privacy concerns, as the same secure context used for the media transport can be re-used for data as well. In fact, all the exchanged data is encrypted end-to-end, despite the fact that the channel was originally negotiated through a server. *Data Channels* are a completely new concept that was introduced solely for WebRTC, but despite this it is already being used in several differ-

ent contexts as well.

Figure 3.12 illustrates what the protocols stack looks like for, respectively, legacy web-based technologies, media channels in WebRTC and Data Channels. As it can be seen in the picture, the above mentioned technologies like XHR or WebSockets all work on top of a reliable transport protocol like TCP. Besides, while they can implement security, those secure channels are terminated at the web server they refer to, meaning there's no end-to-end privacy or security involved.

On the other end, all the new WebRTC technologies make use of an unreliable protocol as UDP for the transport: this allows for a much more timely and fast delivery of the messages. The connectivity is established in all cases through ICE, as explained in Section 3.3.3, and security and privacy are ensured, end-to-end, by the usage of the DTLS protocol, introduced in Section 3.5.1. That said, media and data channels in WebRTC are, at this point, handled differently: specifically, media packets are exchanged by means of SRTP over a so-called PeerConnection, while the generic data channel messages are transported over an additional transport protocol called Stream Control Transmission Protocol (SCTP) [35], and encrypted via the existing DTLS channel. SCTP optionally allows for features typically provided by reliable transport protocols, like reliability, ordered delivery and so on. This means that, while the underlying transport is unreliable, other features can be envisaged as well at the SCTP level if needed.

3.8 First WebRTC integration: Meetecho as a "real" use case

As part of the efforts devoted to the development of my doctoral thesis, I've not only studied and contributed to the WebRTC specification from a theoretical perspective, but I've also tried to gain some first hand experience with WebRTC-related implementations. In particular, I started looking into existing and well known multimedia applications, in order to evaluate if and

how they could be made compliant with the WebRTC enhancements. Without delving too much into the details of these efforts, as they were mostly meant to improve my knowledge of the state of the art and start "playing" with WebRTC before working on the "last mile" as planned, this section will briefly go through my achievements in that sense.

In order to give a broader purpose to these efforts, I focused my activities on implementing a WebRTC-based access to the Meetecho [36] web conferencing and collaboration platform. Meetecho is a standards-based conferencing architecture which makes use of standard protocols (like XMPP, SIP, BFCP, among the others) to provide collaboration features. This architecture was at first devised [37] and developed within the framework of the Network Group of our Computer Science department as an implementation of the Centralized Conferencing (XCON) IETF Working Group. I was one of the authors and implementors of the prototype architecture, and together with some colleagues and professors we co-founded Meetecho, which eventually became an academic spin-off of the University of Napoli Federico II.

The motivation for implementing a WebRTC-based access to Meetecho was simple. In fact, due to the standard nature of the Meetecho architecture, and especially for its use of SIP/SDP and RTP to provide audio and video, it could be seen as one of the "legacy" systems a WebRTC endpoint might want to interact with. For what concerns Meetecho and browsers, we had already devised a fully web based interface to the functionality provided, called *WebLite*. Meetecho WebLite allowed participants to make use of all features by just making use of a browser: this means that access to all of the above mentioned standard protocols and functionality was provided by means of a gateway allowing users to interact with native clients in a seamless and transparent way. While this was relatively simple to accomplish with just HTML and JavaScript for most of the envisaged features, for what concerns audio and video we had so far had to rely on a different approach, namely a Java Applet that, guided by the application logic in JavaScript, would take

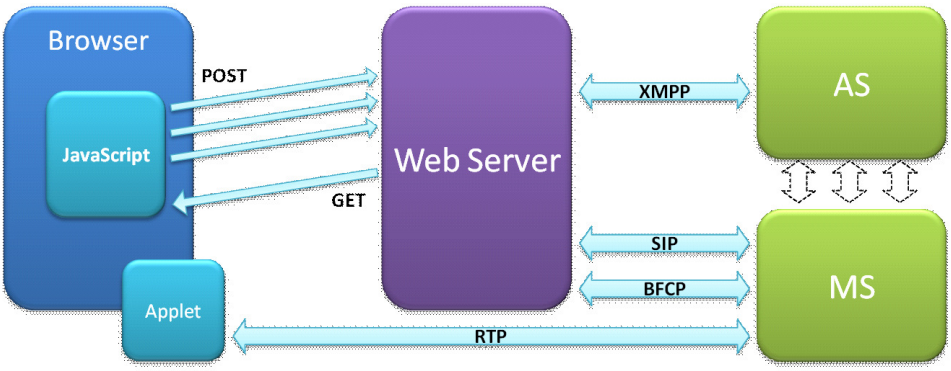


Figure 3.13: The Meetecho WebLite "legacy" architecture

care of the user devices (microphone, webcam), as well as of encoding and decoding functionality and management of the RTP streams exchanged with the conferencing server, as sketched in Figure 3.13. This kind of plugin-based approach is exactly what the WebRTC has been trying to replace since day one, and as such Meetecho proved an excellent testbed for experimenting with both WebRTC and "legacy" applications.

Following the requirements identified in this chapter, the first step was of course to take care of the signalling. As anticipated, Meetecho makes use of SIP as its signalling protocol to negotiate media streams. SIP is also used to negotiate a BFCP channel with participants, in order to provide floor control functionality. Considering the protocol-agnostic nature of WebRTC with respect to signalling, we chose to make use of the above mentioned ROAP protocol as a simple and quick solution to provide signalling in the web interface. We then provided a simple UI to let users decide whether to negotiate both audio and video or just audio, in order to trigger the signalling accordingly. For the server side, we implemented the signalling gateway as a simple web application taking care of signalling messages originated by the client, and in turn generating SIP INVITEs to our conferencing system as a consequence. The signalling state was then handled accordingly, e.g., by providing ROAP answers when a 200 OK to the INVITE would be received or hanging up either side of the call when the other party closed the session. The

task accomplished by the gateway was of course not only limited to signalling, as described when going through the several requirements WebRTC imposed.

One of the features WebRTC demands and we were lacking in Meetecho was SRTP support. As anticipated, this is an absolutely mandatory feature in WebRTC. In order to fix this requirement, we decided to rely on a more recent version of Asterisk[1], the open source PBX (Private Branch eXchange) implementation our audio/video features were built upon, which provided us with native SRTP support, and more specifically its SDES counterpart. While this worked fine for a while, we soon had to face the mandatory support for DTLS as a requirement. The support for DTLS-SRTP, though, was sketchy, so we worked on improving it in order to make it comply with what browser implementations expected. Another mandatory feature we found ourselves lacking was ICE. When we first started working on this WebRTC integration, this was not an issue we could solve by just upgrading Asterisk, which didn't provide support for it at the time. As such, we implemented support for it ourselves. We did so in two different parts of our system. The negotiation of candidates was taken care of in the above introduced signalling gateway: in fact, considering our conferencing server was supposed to be always reachable at a public address, the gathering of candidates could be definitely simplified, and allow for the gateway to act as an ICE-Lite peer. The connectivity checks were instead implemented on the Asterisk side, by modifying the pre-existing STUN protocol implementation in order to take care of all the required additional attributes like USE-CANDIDATE, XOR-MAPPED-ADDRESS, MESSAGE-INTEGRITY and so on, and be able to both respond to connectivity checks and generate checks on its own. This choice was made in order to keep things as simple as possible for signalling, and focus on the media challenges instead, while also keeping signalling- and media-related issues well separated and independently addressable. At this point, we started looking at possible issues in the negotiation process itself. We first of all noticed that Asterisk would reject the "RTP/SAVPF" profile

[1] http://www.asterisk.org

as negotiated by the WebRTC endpoint. Hence, we looked after rewriting of the SDP descriptions on both directions, in order to make sure that Asterisk would find RTP/SAVP in the SDP received from the gateway, and WebRTC would get back the RTP/SAVPF it negotiated in the first place instead. When done with that, we looked after taking care of SDP attributes that might confuse either side. As anticipated in Section 3.7.2, WebRTC endpoints try to multiplex RTP and RTCP streams, which is something Asterisk does not support at the moment. Specifically, Asterisk still assumed the port used for RTCP to be equal to the RTP port, incremented by one. In order to make sure the WebRTC endpoint could be aware of this, we implemented the gateway so that, before sending the conferencing server SDP to the WebRTC endpoint, the SDP was modified with a new media-level attribute (a=rtcp:...) for each involved media with explicit indication of the port to be used for RTCP. Moving further, as anticipated we chose to demand the ICE-related negotiation to the gateway rather than to the PBX itself. To this purpose, we made sure that, before providing the WebRTC endpoint with any reply or update from the conferencing server, the SDP would be extended with the ICE additional attributes, namely a session level attribute to report an ICE-Lite implementation (a=ice-lite) and media level attributes to convey authentication information (a=ice-ufrag and ice-pwd) and candidates (a=candidate) out of what the PBX negotiated. This of course obliged us to provide, for each negotiated medium, two different candidates instead of just one: in fact, as explained before, we would negotiate RTCP explicitly on a different port, which had to be reported as a candidate accordingly, as connectivity checks would need to be achieved for RTCP as well and not just RTP.

When done with the above mentioned steps, we started looking into interoperability at the encoding level. For audio, this was quite easy at first. In fact we found out that both WebRTC endpoints and our conferencing server had a codec in common, specifically G.711 μ-law. That said, we immediately verified G.711 not to be the best choice when it comes to audio:

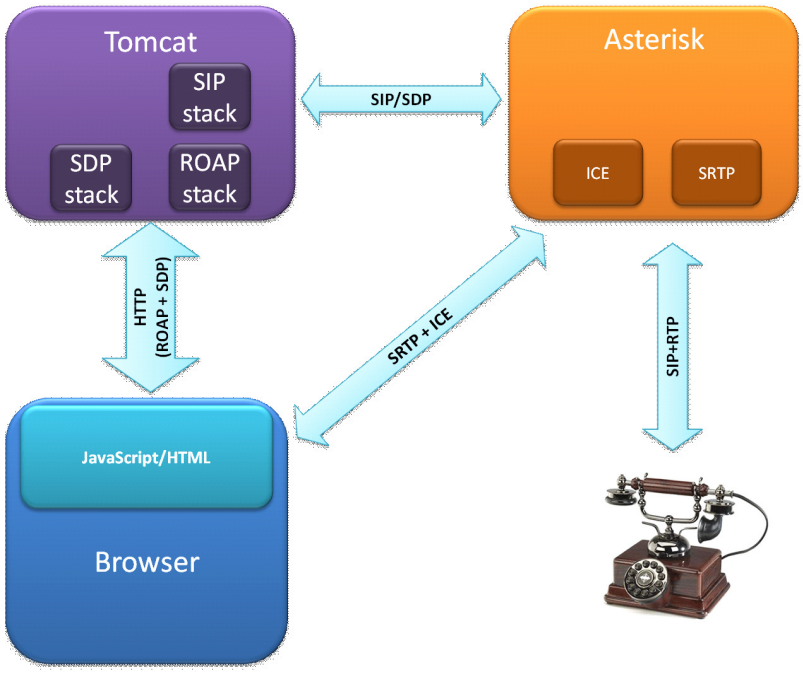


Figure 3.14: The Meetecho RTCWebLite architecture

in fact, while it did indeed greatly simplify the task of creating a bridge between WebRTC and our legacy system, the resulting quality experience left much to be desired. As such, we worked on an integration of the high quality and standard Opus codec within Asterisk, an effort we released as an open source patch[2] to the community. This effort was subsequently integrated in the mainstream distribution of Asterisk, and is now part of the official package. Besides, I had the opportunity of presenting the results of my efforts during the Technical Plenary[3] at the IETF 87 meeting in Berlin, an event we also streamed via Meetecho using a superwideband (48kHz) audio quality Opus stream for the benefit of all remote attendees.

For what concerns video, instead, the work to be done was quite different. In fact, it is important to notice that while Asterisk provides audio mixing natively, it does nothing in that sense when it comes to video, which

[2]https://github.com/meetecho/asterisk-opus
[3]http://trac.tools.ietf.org/group/iab/trac/wiki/IETF-87

it can only handle with a passthrough approach. To take care of this, we had long time before designed and implemented a videomixer that could take care of both transcoding and mixing heterogeneous video streams of different formats. Our videomixer, though, only supported "legacy" encodings like H.261, H.263 (and its extensions) and optionally H.264, but not VP8, which, waiting for a clear consensus on the MTI video codec in WebRTC, was and still is the only video codec the reference WebRTC implementations support. As a consequence, we had to accomplish two different steps: (i) implement a simple passthrough mechanism for VP8 frames in Asterisk, and (ii) implement full VP8 transcoding features in our videomixer, in order to allow VP8-compliant endpoints to take advantage of the video composition our videomixer provided. This process was successful but had a small drawback. In fact, our video mixer only supported QCIF and CIF resolutions with respect to video, while the reference WebRTC endpoint always sent 640x480 video streams. While this would not be a problem on the client side (the browser showed no issue when presented with a lower resolution incoming video stream) this could indeed be problematic for the videomixer, as it had to continuously downscale incoming frames from WebRTC endpoints in order to mix them. To complete the integration, we also implemented support for the RTCP FIR feedback message in Asterisk: this was needed in order to allow the videomixer to receive a full frame for a participant whenever the participant was to be included in the mix (e.g., when the participant was granted the video floor by means of BFCP), in order to avoid annoying artifacts or ghosting effects.

The resulting integrated architecture, which is is depicted in Figure 3.14, was eventually addressed in a journal article [38] we wrote for the benefit of the community of researchers that were interested in achieving similar efforts.

Chapter 4

Janus: a general purpose WebRTC gateway

In Chapter 2 we've gone through the SOLEIL architecture, thus identifying its many challenges and requirements. Among those, we hightlighted the need for a component that could bridge two very different worlds: on one side, relays talking plain RTP and RTCP, while on the other side, an audience of WebRTC-empowered attendees. At the same time, we explained how a similar approach could be used on the broadcaster side as well, in order to allow a user to just make use of their browser to inject one or more WebRTC originated streams into a SOLEIL instance, and have it relayed internally in the format SOLEIL expects.

Chapter 3 described our efforts in studying and identifying the several different technologies WebRTC embodies, and the challenges they provide when having to deal with them. More specifically, we addressed each of those technologies separately, describing our efforts in getting them to work in pre-existing, or "legacy", multimedia frameworks like Meetecho, in order to foster the realization of simple WebRTC "gateways", that is tools that would allow WebRTC users to interact with a non-WebRTC world.

All those considerations and efforts eventually confluted in the design and realization of a brand new, written from scratch, WebRTC gateway, that could be used to bridge WebRTC to and from pretty much any pre-existing technology. The following sections will start from why such a component is

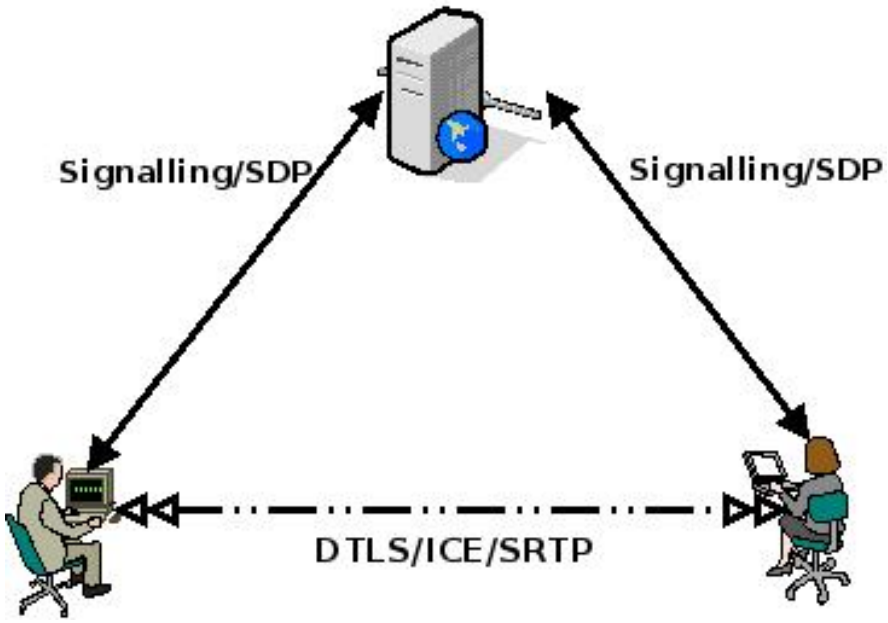


Figure 4.1: WebRTC native peer-to-peer communication

needed in the first place, and then move to describe the design process for this component in detail, to eventually introduce its integration within the SOLEIL architecture to satisfy the "last mile" requirement.

4.1 A gateway? Why?

Since day one, WebRTC has been seen as a great opportunity by two different worlds: those who envisaged the chance to create innovative and new applications based on a new paradigm, and those who basically just envisioned a new client to legacy services and applications. Actually, as it often happens, the actual deployment of WebRTC eventually happened somewhere in the middle, and it's not at all unusual to have WebRTC-compliant implementations that break the peer-to-peer approach and reside on the media path between them.

As anticipated in Chapter 3, and depicted in Figure 4.1, WebRTC has been conceived as a peer-to-peer solution: that is, while signalling goes

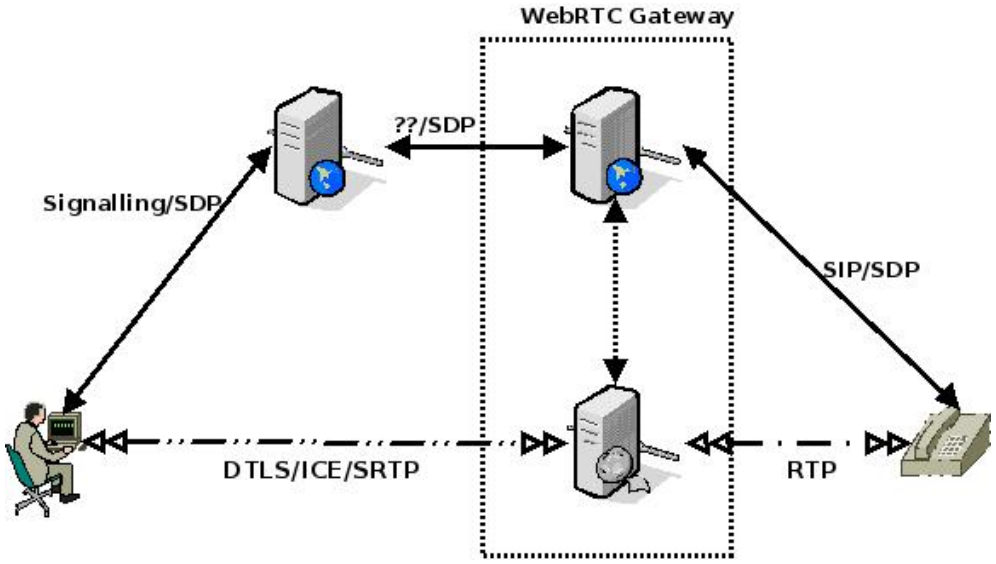


Figure 4.2: One of the peers as a logically decomposed WebRTC gateway (SIP example)

through a web server/application, the media flow is peer-to-peer. Even in a simple peer-to-peer scenario, though, one of the two involved parties (or maybe even both) doesn't need to be a browser, but may very well be an application, as depicted in Figure 4.2. The reasons for having such an application may be several: it may be acting as a Multipoint Control Unit (MCU) or Selective Forwarding Unit (SFU), a media recorder, an IVR application, a bridge towards a more or less different technology (e.g., SIP, RTMP, or any legacy streaming platform) or something else. Such an application, which should implement most, if not all, the WebRTC protocols and technologies, is what is usually called a *WebRTC Gateway*: one side talks WebRTC, while the other still WebRTC or something entirely different (e.g., translating signalling protocols and/or transcoding media packets).

That said, there are several reasons why a gateway can be useful. Technically speaking, MCUs, SFUs and server-side stacks can be seen as gateways as well, which means that, even when you don't 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 according to the scenario you want to achieve.

Nevertheless, the main motivation comes from the tons of existing and so-called legacy infrastructures out there, that may benefit from a WebRTC-enabled kind of access. In fact, one would assume that the re-use of existing protocols like SDP, RTP and others in WebRTC would make this trivial. Unfortunately, most of the times that is not the case, as confirmed by our analisis of the WebRTC media components in Chapter 3 and the several challenges they posed to a successful integration and deployment. If, for instance, we just refer to existing SIP infrastructures, even by making use of SIP as a signalling protocol in WebRTC there are too many differences between the standards WebRTC endpoints implement and those available in the currently widely available deployments.

Just to make a simple example, most legacy components don't support media encryption, and when they do they usually only support SDES. On the other end, for security reasons WebRTC mandates the use of DTLS as the only way to establish a secure media connection, a mechanism that has been around for a while but that has seen little or no 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 for NAT traversal, RTCP feedback messages for managing the status of a connection or RTP/RTCP muxing, whereas existing infrastructures usually rely on simpler approaches like Hosted Nat Traversal (HNT) [39] in SBCs, separate even/odd ports for RTP and RTCP, and more or less basic RTCP statistics and messages. Things get even wilder when we think of the additional stuff, mandatory or not, that is being added to WebRTC right now, as BUNDLE, Trickle ICE, new codecs the existing media servers will most likely not support and so on, not to mention Data Channels and WebSockets and the way they could be used in a WebRTC environment to transport protocols like BFCP or MSRP, that SBCs or other legacy components would usually expect on TCP and/or UDP and negotiated the old fashioned way.

This brings us back to SOLEIL and the obvious need for a component able

to cover the requirements we just identified being exactly those of some kind of WebRTC gateway. In fact, all the relays internally work using plain, unencrypted, RTP/RTCP traffic, and don't make use of any specific signalling protocol besides the synchronization messages SOLEIL endpoints exchange with each other. On the other end, WebRTC attendees obviously expect to be able to access those streams somehow, which means we needed a way to make the SOLEIL world with WebRTC. Besides, considering the dynamic nature of such a scenario, the WebRTC gateway component needed to also be "smart" enough, meaning it needed to be programmable and controllable in order to dynamically handle new users and scenarios without requiring a complete rewrite of the component.

In order to cope with this requirements, we designed a brand new WebRTC gateway called Janus. We explicitly conceived it as general purpose, meaning we were interested in designing a framework that could be used with whatever technology (legacy or not) we needed to interact with, without needing to write a new component from scratch later on when the issue presented itself. Janus, which was released as completely open source software [1] less than a year ago, will be introduced in the following sections.

4.2 A programmable approach: MEDIACTRL

Work on the design of Janus actually started much earlier than it saw the light about a year ago. In particular, we have been working for years on programmable and controllable Media Server instances within the framework of the IETF Media Server Control (MEDIACTRL) Working Group. This WG worked on the specification of a standard, SIP-based, communication channel between Media Servers (as in the entities taking care of the actual media manipulation and delivery, the "arm") and Application Servers (the entities hosting the application logic, that is what should be done with media and when/where, the "brain").

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

The motivation for such a work was simple. In fact, within the real-time multimedia applications ecosystem, users can either interact with others users directly (e.g., for a video call) or with an application (e.g., an Interactive Voice Response system, a conference bridge, a call center, etc.), that might or might not in turn handle other users that may be put in contact with each other somehow. This means that this application needs to implement the whole protocols specification (e.g., SIP/SDP, RTP) in order to be able to interact with users from a technological perspective, while at the same time be controllable in order to allow the application to decide what media to provide its users with, or how to handle the media users themselves may be sending. This already highlights how internally such an application may be structured, that is, a part that handles the application logic (something that decides what to do when a user calls in) and a part that handles the media instead (negotiating the media channels, sending and receiving RTP packets, and so on).

While from a functional perspective these operations may be seen as achieved by the same component, as a user calling a conference bridge, for instance, will think the media mix is coming from there, this is not how such applications are usually implemented. In fact, most of the times the components implementing the application logic and those handling the media are actually not co-located but distributed across a network, although often close enough to each other. This allows for a proper separation of responsibilities among different implementations, and besides also allows for scaling such components separately and as such in a more controllable way.

This was exactly the purpose of the IETF MEDIACTRL Working Group efforts, to which we participated actively during the latest years by contributing to specifications and providing proof-of-concept implementations as a support for the design process. The idea was to basically design a communication channel Application Servers and Media Servers could exploit for talking to each other in order to implement dynamic and heterogeneous multimedia applications based on SIP. This would allow Media Server im-

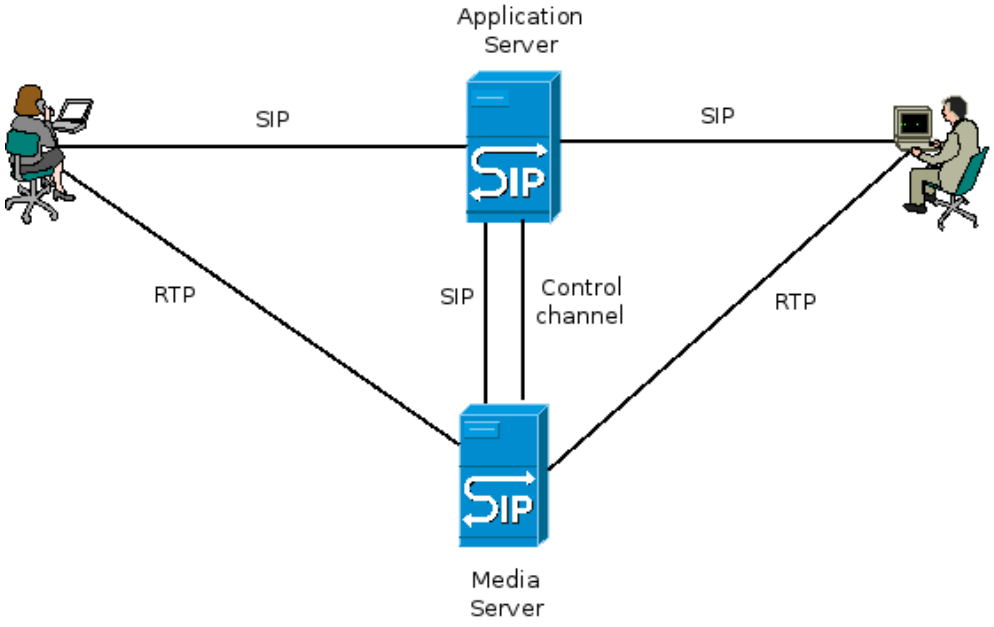


Figure 4.3: Media Server Control architecture

plementors to just focus on the media aspects and provide their services to third-party Application Server developers, who could in turn just care about how the media components made available by a Media Server could be combined together in order to implement a new, rich multimedia application. Considering the dynamic nature of this interaction among components written by unrelated entities, it's quite evident how the choice to work on a standard framework for the purpose proved quite important.

4.2.1 The MEDIACTRL architecture

The MEDIACTRL Working Group, as anticipated, worked on the definition of a SIP-based modular framework for media server control. Specifically, the assumption was that the components handling the application logic and the components actually manipulating/serving the media would be distributed and not co-located, which meant that some kind of communication channel among them was needed in order to allow for the realization of rich multimedia applications.

In order to achieve the desired result, a generic architecture [40] was first devised. Figure 4.3 describes the overall architecture that was envisaged for MEDIACTRL, which involved the following components: (i) an Application Server implementing the application logic; (ii) a Media Server responsible for manipulating and handling the media channels; (iii) some User Agents interested in taking advantage of real-time multimedia applications. Within this framework, the key functionality provided by the MEDIACTRL architecture was to be the communication channel between the Application Server and the Media Server. Specifically, since the Application Server is responsible for the application logic, it's there that the User Agents interested in the application (e.g., a multimedia conference, a contact center, an Interactive Voice Response application and so on) would send their SIP requests. The Application Server would then, in turn, forward the media session requests to the Media Server, in order to have the media channels negotiated and set up between the Media Server itself and the User Agents; at the same time, a programmable and extensible control channel would be used to instruct the Media Server to implement some specific actions on the related media channels (e.g., play a pre-recorded media file, mute/unmute a user in a conference, etc.).

As anticipated in the previous section, this would allow for a proper separation of responsibilities, as Application Servers could just focus on handling the interactions with User Agents, while delegating all the media processing to the Media Servers they'd control. In order to do so, a text-based Control Channel was designed within the MEDIACTRL framework [41] to allow Application Servers and Media Servers to interact with each other. Specifically, this Control Channel can be negotiated by means of SIP as other media: this means that an Application Server can just place a SIP call to a Media Server to negotiate the creation and setup of a new Control Channel, and after the channel has been created the Application Server can start sending requests and receiving responses/events from the Media Server.

Of course, since the main purpose of a Media Server is to actually provide

configurable and controllable media processing on SIP dialogs, the availability of a Control Channel was only the first step. In order to allow a Media Server to expose some processing functionality and an Application Server to exploit them, this Control Channel also needed to make available programmable APIs to control these features. That said, considering the dynamic nature of multimedia applications, it was decided that these features should not be "baked in" the MEDIACTRL architecture itself, but delegated to external packages that could be negotiated and exploited separately.

4.2.2 Control Packages and Extensibility

These external components were called Control Packages, and were specifically conceived to allow for an easy extensibility of the MEDIACTRL framework. In fact, while there are some well known multimedia application patterns, new application scenarios may appear at any time, meaning a Media Server should be flexible enough to account for applications it wasn't designed for initially.

To handle this requirement, the MEDIACTRL specification was designed to expose an extensible interface these Control Packages could implement in order to register at a Media Server and expose their services. In order to make this dynamic and flexible, the Control Channel protocol was designed to allow for a negotiation of Control Packages after creation (e.g., to allow an Application Server to check for the presence of a specific package at a Media Server) and for an opaque transfer of messages between applications and packages. In fact, it was quite clear that, while each Control Package could implement its own, package-specific API, this would need to be transported over a shared communication channel. This is what Figure 4.4 illustrates: each Control Package message is composed of a generic header and, if present, an opaque payload. This opaque payload is addressed to a specific package according to the info made available in the headers, which means the MEDIACTRL core can simply act as a dispatcher for these messages between packages and interested applications, while still being able to expose generic

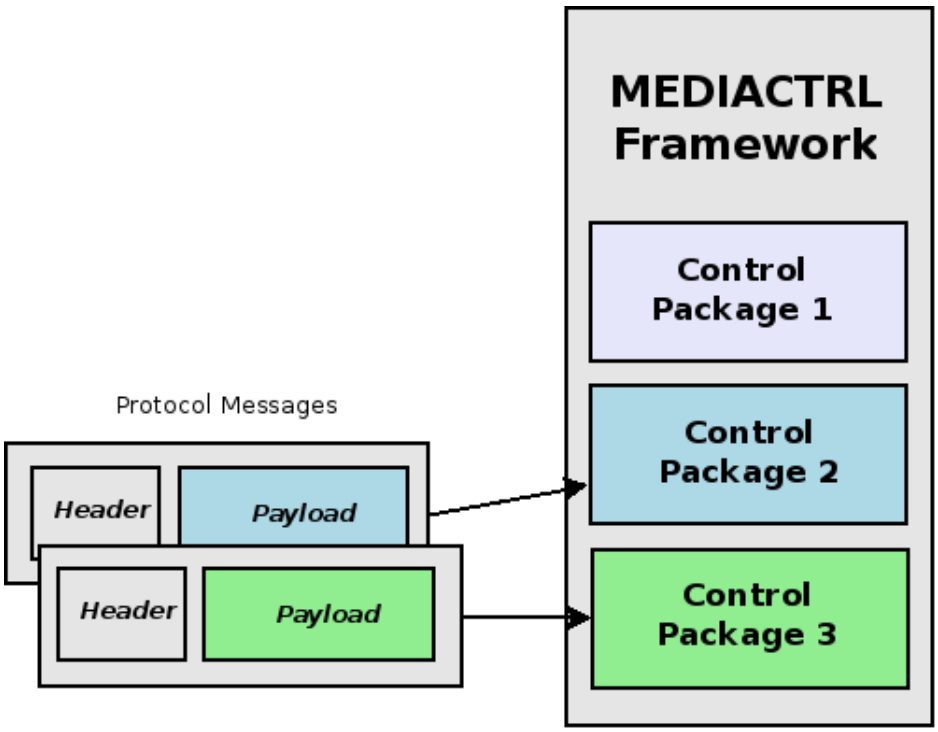


Figure 4.4: Control Packages and the MEDIACTRL Protocol

media processing on behalf of packages.

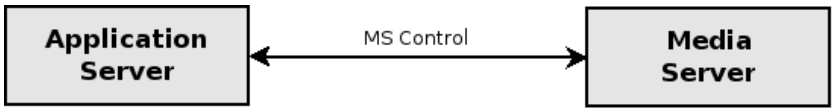
During the works on the MEDIACTRL specifications, two different Control Packages were specified, an effort on which we contributed as part of the design team. Specifically, an IVR [42] and a Mixer Control Package [43] were designed. While the IVR package allowed for generic interaction patterns as the playback of pre-recorded media, recording of media sent by users and interaction with users by means of Dual-tone multi-frequency (DTMF) tones, the Mixer package was instead focused on bridging media connections, by involving mixing if required. These two packages, combined with each other, provided as a consequence all the required bricks to implement very different and rich multimedia applications. In fact, while a simple IVR system could be implemented by just relying on features made available by the IVR Control Package, more complex applications like contact centers, conference bridges and others could instead be implemented by using features made available by both. For instance, a conference bridge application

typically starts with the conference system asking for your name, followed by a global announcement that tells all participants about the new user joining in and then mixing all the contributions, muting/unmuting if needed. Such a complex application can indeed be decomposed in several different simpler operations: (i) the conference system asking for your name could be a playout followed by a brief recording, features made available by the IVR package; (ii) the global announcement can in turn be implemented as a playout of the just recorded message on the conference mix; (iii) finally, the bridging, mixing and muting features are all available in the Mixer package.

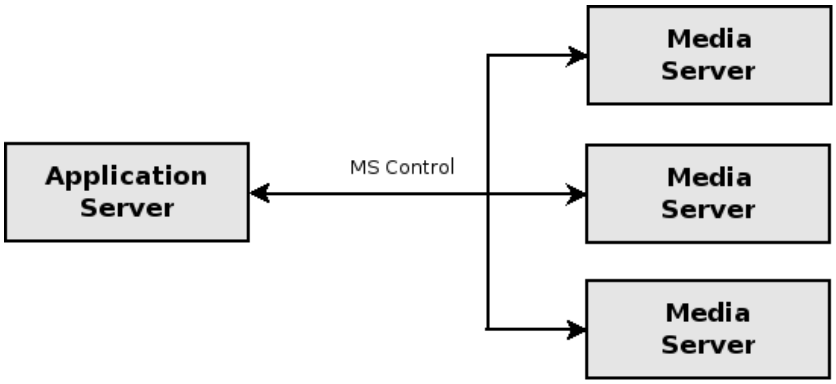
This is a very simple example of the complex interactions that can be envisaged using the Control Packages as features rather than applications by themselves. This and other scenarios were covered in a document, called "Media Control Channel Framework (CFW) Call Flow Examples", we wrote as part of the MEDIACTRL standardisation efforts, and that eventually became an official standard as RFC 7058 [44] a few months ago. Our efforts on the MEDIACTRL design and implementations were also documented in a paper [45] that was subsequently extended to become a book chapter [46].

4.2.3 Media Resource Brokering

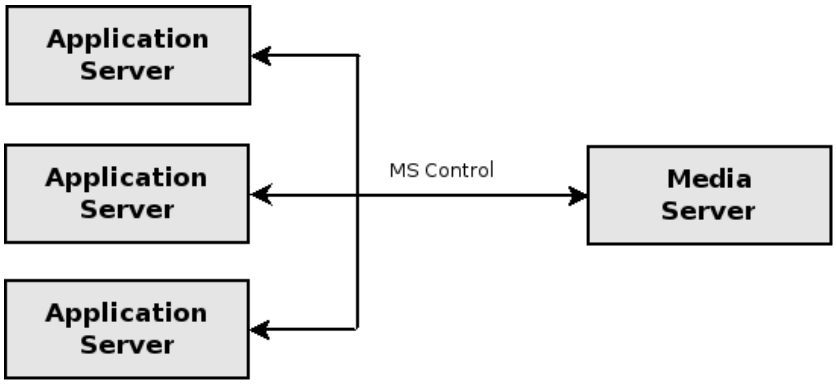
All the discussions made so far on the MEDIACTRL architecture always envisaged an interaction between a single Application Server and a Single Media Server. While this is the most simple and common topology that can occur within a multimedia deployment, this is not the only way the MEDIACTRL specification can be used. In fact, different Application Servers may be interested in the services of the same, shared, Media Server, or the same Application Server may actually refer to multiple Media Servers rather than a single one. The different topologies one could encounter are depicted in Figure 4.5. As it can be seen in the figure, in general one can assume that the most generic topology envisages a battery of Application Servers requiring the services of a pool of Media Servers, thus describing a M:N kind of relationship.



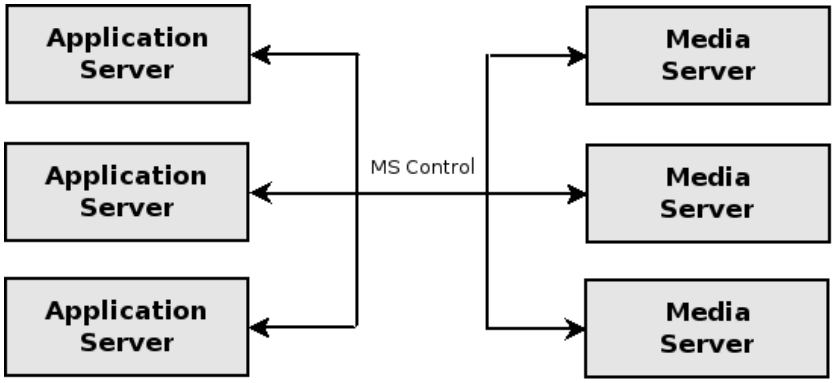
(a) 1:1 Topology



(b) 1:N Topology



(c) M:1 Topology



(d) M:N Topology

Figure 4.5: Different possible topologies in MEDIACTRL

In order to allow for these kind of interactions to happen, the MEDIACTRL Working Group identified the need for some kind of mechanism for Application Servers to dynamically discover Media Servers, and for Media Servers to advertise their presence and the available functionality. The information from Media Servers needed to also include a more or less detailed overview on the available resources, in order to allow for a proper distribution of those across different interested applications.

These requirements were eventually addressed in a separate specification we worked on, which was called "Media Resource Brokering" [47] and became an official standard in RFC 6917 a few months ago. This specification defined a new component, called Media Resource Broker (MRB) that could act as an intermediary between multiple Application Servers and Media Servers. This component was designed to allow Media Servers to register at the MRB, advertise the available functionality in terms of Control Packages, supported codecs and so on, and keep it up-to-date with respect to the resources still available (e.g., how many calls are up and how many can still be handled). The information coming from multiple Media Servers, then, could in turn be used by interested Application Servers in order to reserve resources they might need for a multimedia application, e.g., in order to have the MRB return the addresses of one or more Media Servers that can support a specific volume of users for a certain application.

Figure 4.6 illustrates an example of how a MRB can indeed serve a request coming from an Application Server. In this case, the MRB is aware of three different Media Servers, two of which only implement the IVR package, while the third one implements the Mixer Package instead. All of those Media Servers have some "slots" available to handle a few users (e.g., 60 new user agents). According to the requirements provided by an Application Server in its request (e.g., a Media Server able to support an IVR system for 100 callers), the MRB can then inspect the available resources and provide the Application Server with what it needs, in this case the address of a Media Server it can contact, directly or through the MRB itself, in order to

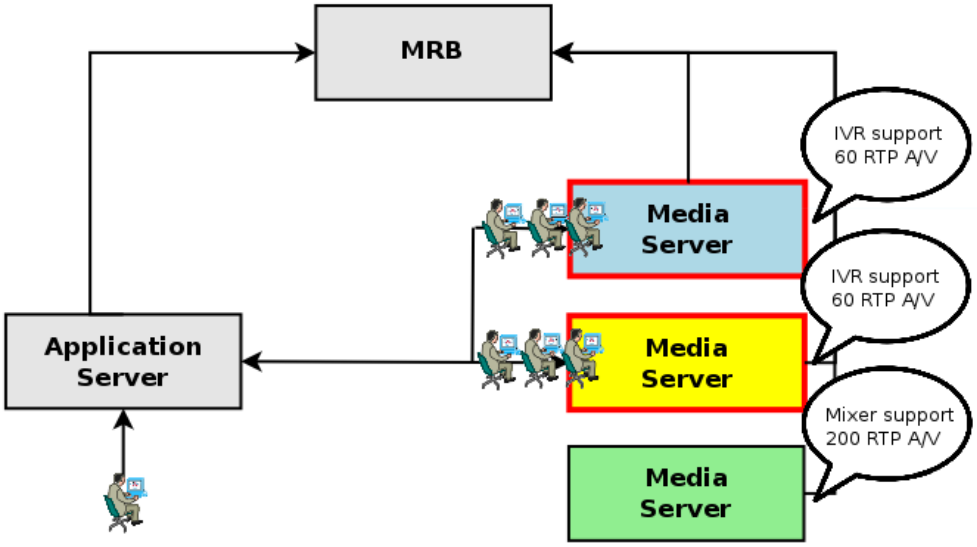


Figure 4.6: Media Resource Brokering

implement the multimedia application.

The interested reader can refer to [48] for a more detailed overview on how the brokering of media resources can effectively improve the scalability of real-time multimedia applications.

4.3 Janus: a general purpose WebRTC gateway

As anticipated at the beginning of this chapter, the main purpose was to design and implement a component that could take care of the "last mile" requirements of a SOLEIL *leaf* node. Specifically, we highlighted how such component should be able to act as a WebRTC gateway in Section 4.1, in order to transparently interact with both the SOLEIL distribution network and WebRTC viewers. Besides, we also noticed how a programmable and controllable implementation could help cover heterogeneous scenarios in a flexible and dynamic way, which is why in Section 4.2 we introduced the efforts we made on the MEDIACTRL specification within the IETF standardisation work.

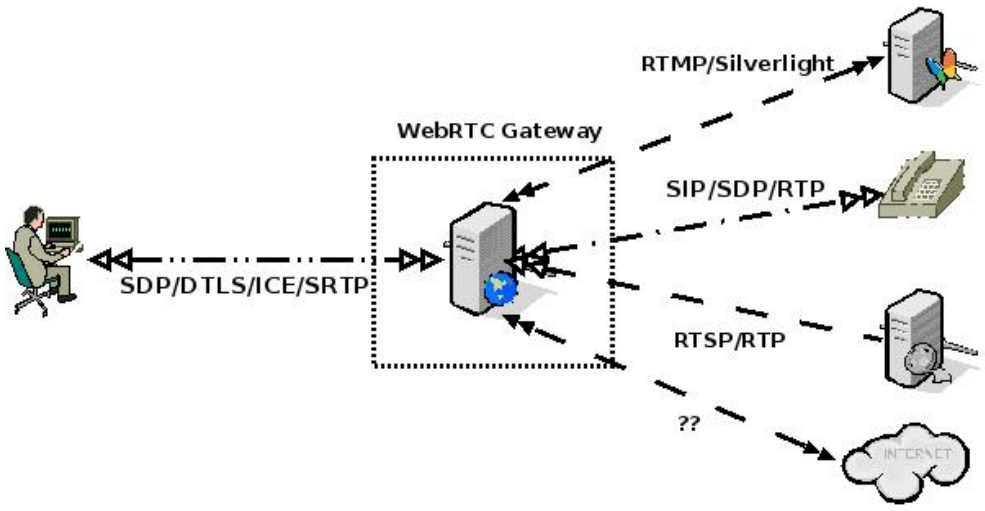


Figure 4.7: Bridging to different technologies

Starting from these assumptions, we eventually designed an implemented a brand new WebRTC gateway called Janus, which we then released as open source for the benefit of the whole researchers community. As it will be clearer in the next sections, our contributions in the MEDIACTRL works proved fundamental for designing a modular architecture for Janus as well, that is something that could actually turn a specific tool like a WebRTC gateway into a general purpose multimedia service.

Janus was also introduced to the research community in a paper that described its architecture and features [49].

4.3.1 A modular architecture

Just as the well known Ancient Rome god had two faces, one looking at the past and one at the future, our own Janus always has two faces as well: one is WebRTC (the future), and the other is whatever technology it needs to interact with. As such, it's conceived to be able to allow for the implementation of WebRTC-based media streaming, conferencing, recording, gatewaying to legacy technologies and so on, as depicted in Figure 4.7.

In fact, the Janus gateway was explicitly conceived to be a general pur-

pose one. As such, it doesn't provide any functionality per se other than implementing the means to set up a WebRTC media communication with a browser, exchanging JSON messages with it, and relaying RTP/RTCP and messages between browsers and the server-side application logic they're attached to. Any specific feature/application is provided by server side plugins, that browsers can then contact via the gateway to take advantage of the functionality they provide. Example of such plugins can be implementations of applications like echo tests, conference bridges, media recorders, SIP gateways and the like.

The reason for such a modular architecture is simple. We wanted something that would have a small footprint (hence a C implementation) and that we could only equip with what was really needed (hence pluggable modules). That is, something that would allow us to deploy anything ranging from a full-fledged WebRTC gateway on the cloud to the small nettop/box you build to handle a specific use case.

The reference architecture is depicted in Figure 4.8. As anticipated, 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. As explained in Section 4.2, we were motivated in following this approach from our former experiences within the IETF MEDIACTRL Working Group. As introduced before, 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. Considering the effective approach fostered by MEDIACTRL, we chose to follow a similar path for Janus as well, with a core handling the high level communication with users (management of sessions and handles, concepts that will be introduced in the following sections, and WebRTC-related protocols) and server-side plugins to provide specific

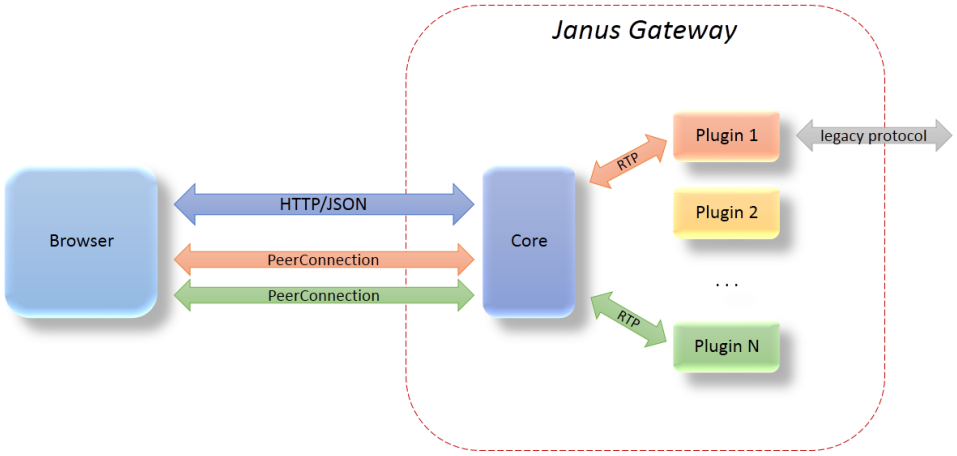


Figure 4.8: Janus modular architecture

functionality in a way that is transparent to WebRTC itself, and as such independent of the web application.

Since this core was meant to implement the whole WebRTC stack in order to be able to interact with WebRTC-compliant endpoints, we had to take care of all the related media components, as highlighted in Chapter 3. As reinventing the wheel is never wise, and considering WebRTC itself is partly based on top of pre-existing technologies that were opportunely modified or enhanced to better suit their requirements, we chose to rely on some well known open source libraries to help us address the media requirements. Specifically, we chose the *libnice*[2] library to implement the whole ICE process, the widely deployed *openssl*[3] to take care of DTLS, *libsrtp*[4] to implement the secure SRTP cryptographic contexts, *usrsctp*[5] as the SCTP stack needed for Data Channels and finally *Sofia-SIP*[6] as an SDP parser.

As anticipated, we had to tweak their usage in order to satisfy the addi-

[2]http://nice.freedesktop.org/wiki/
[3]http://www.openssl.org/
[4]http://srtp.sourceforge.net/srtp.html
[5]http://code.google.com/p/sctp-refimpl/
[6]http://sofia-sip.sourceforge.net/

tional requirements imposed by the WebRTC specification, namely Trickle ICE, BUNDLE and Data Channels as introduced in Section 3.7. This allowed us to focus on the more challenging steps of the design, that is how to glue these components together and build a core that could be safely shared among completely heterogeneous modules.

4.3.2 The Plugins interface

Since we were interested in making Janus a general purpose application, we decided to devise the modular architecture introduced in the previous section. Specifically, we wanted to have a core that could implement all the WebRTC suite, and pluggable modules that could implement the application logic instead, without having to worry about the complexities of the WebRTC interactions. To do so, we designed a core architecture that recalled what we had already done in MEDIACTRL with the Control Packages, that is some kind of interface exposed by the core that external plugins could implement in order to provide additional features to Janus.

This was exactly what we did when designing the Plugin API in the Janus core. More precisely, we specified an interface composed of both methods plugins could expose or callbacks by which they could contact the core. This included basic methods to get generic versioning information about plugins and more advanced method and callbacks to receive and relay RTP/RTCP and data, handle plugin-specific messaging with the applications, and so on.

That said, the modular approach defined in MEDIACTRL and in Janus are not identical. In particular, while the Control Packages as defined in Section 4.2.2 could be shared on the same multimedia connection, e.g., to first play an announcement and then attach the connection to a media conference, in Janus each plugin has complete control over a specific PeerConnection. While at a first glance this may seem limiting, this was actually done by design and for some specific reasons. In fact, considering the heterogeneous and possibly very different nature of the plugins implemented in Janus, each of those could have some very specific requirements in terms of

what they require of a media connection. For instance, a plugin implementing audio mixing and conferencing functionality would only negotiate audio in a PeerConnection and require some specific codecs to be supported; on the other hand, a streaming plugin would require different characteristics of a PeerConnection, which would need to be monodirectional and possibly depending on the capabilities provided by an external encoder. As such, plugins need to have an important voice as far as negotiation is involved, to ensure the media channels they'll be handling are compliant with the functionality they expose. This means that sharing a PeerConnection among different plugins, while possible, would indeed strongly limit the flexibility on what's negotiated, thus constraining it to something that is generic enough to be shared. We decided not to add any limitations in this aspect, especially considering our purpose was to provide a modular architecture that could easily be expanded and enhanced to cover scenarios that were not thought of when specifying the interface.

This approach also proved actually more flexible that the one conceived for MEDIACTRL for a different reason. In fact, when MEDIACTRL was first designed the target was allowing an Application Server to control the media being served on a single media channel: this meant that a user would negotiate, by means of SIP, a single audio and/or video channel, and what would travel across those channels was decided as part of the communication between the Application Server and the Media Server. This explains why the modular architecture conceived for Control Packages allowed them to share the channel, as the same media connection would need to be used for all the features provided by the packages. On the other hand, this single-channel approach is quite limiting when envisaged within the context of WebRTC applications: on the contrary, many applications actually make use of multiple PeerConnections at the same time to provide rich multimedia scenarios, e.g., in video conferences or similar applications. This multi-PeerConnection approach convinced us that the plugin approach we chose for Janus could actually allow for much more flexibility in deploying applications. In fact,

different plugins using different PeerConnections could be exploited together as "bricks" for a much richer user experience: for instance, to implement a Social TV application, where friends watch and discuss together a TV show, a streaming plugin used for the TV broadcast could be exploited in conjunction with a video conferencing plugin to address the video-based users interaction. At the same time, the same plugins could be combined to implement a completely different application scenario, e.g., an e-learning platform.

For this reason, we explicitly conceived the plugin API not as a way to design plugins that would implement a specific application, but only some specific features or "bricks". This way, the same plugin can be actually reused in several different contexts, and composed, at an application level, with different plugins as well.

4.3.3 Interacting with the gateway: multi-transport API

Of course, since Janus is supposed to be a controllable component, it's essential that it can be interacted with in a programmable way. To do so, we derived yet another lesson from MEDIACTRL, that is we designed and implemented an API that would take into account the pluggable nature of the server side modules. As such, the Janus API, which is JSON-based, has a generic header container, which can be used to interact with the core, while at need plugin-specific payloads can be attached to interact with each module's application logic. The same applies to asynchronous event notifications coming from Janus and originated by plugins.

As depicted in Figure 4.9, the API works in a pretty straightforward way: whenever a user needs to send a message to a specific plugin, it can do so by specifying the recipient using its handle identifier, and then attach a JSON-based body that will be delivered by the Janus core to the plugin itself. This JSON-based body is completely plugin-specific, which means each plugin is absolutely open to design and implement its own messaging scheme, as long as it doesn't break the JSON constraint. Since it's completely opaque to the

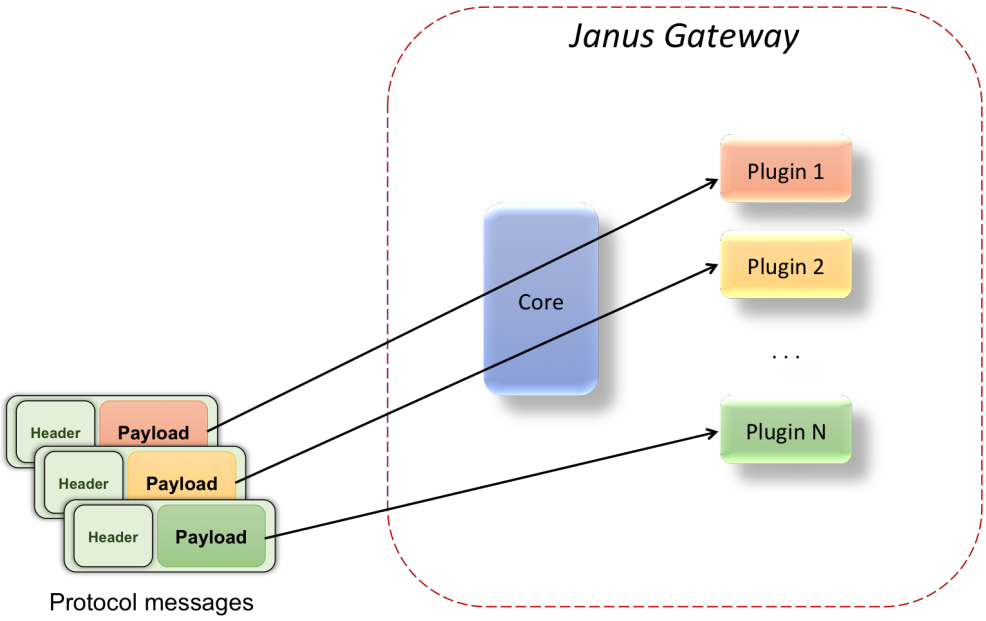


Figure 4.9: Janus API: REST, WebSockets and RabbitMQ

Janus core, there's no other constraint imposed on these messages.

In order to take into account both the extensibility of the protocol and the need to address the associations among application contexts, PeerConnections and specific plugins, we introduced at the core level the concepts of *sessions* and *handles*. A *session* is basically an applicative session between a user/application and the core; a *handle* instead is the abstraction of a connection between a user/application and a specific plugin, which means, as explained before, that each *handle* is actually also the abstraction of a PeerConnection between the user/application and the plugin itself. From the core and API perspective, each user/application usually mantains a single *session* with Janus, while the *session* itself can be used to establish and handle one or more *handles* with the same or different plugins. More *handles*, in fact, can be established with the same plugin by the same user: to make a simple example, in a video forwarding plugin, for instance, a user may use a *handle* to send their own video in a video conference, and other *handles* to receive the video coming from other participants. Besides, the ability to create more *handles* with different plugins allows for the plugins-as-features composition

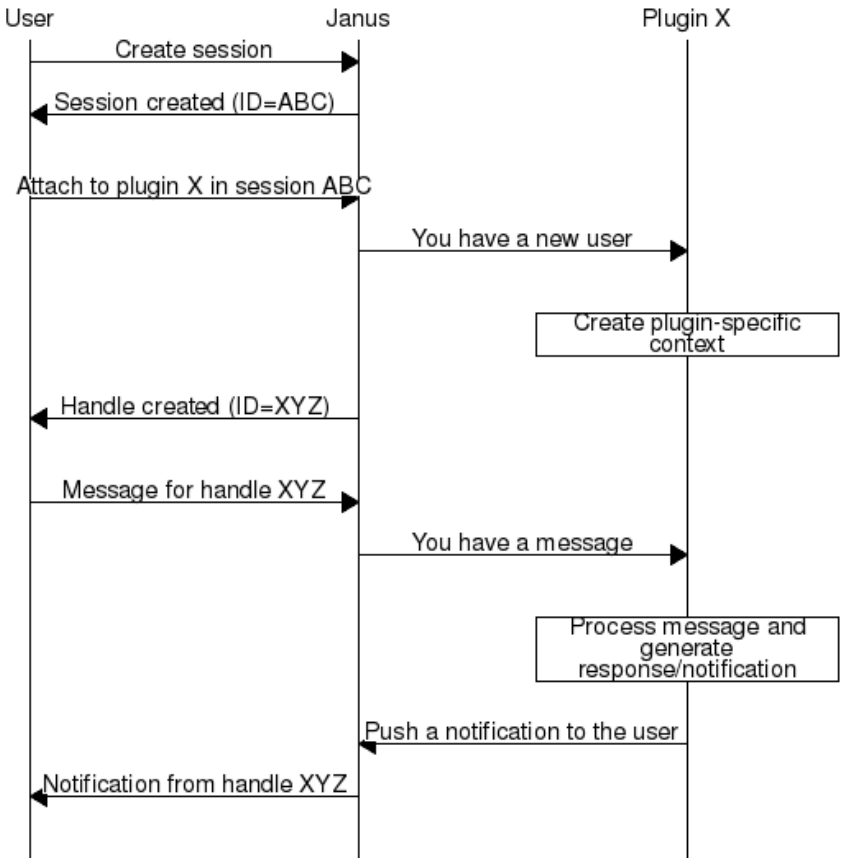


Figure 4.10: Janus API example: creating a handle to a plugin

in applications that was previously introduced. All users/applications can then interact with the specific plugin instance by sending and receiving messages through the Janus API on the *handle* itself. An example of how this approach can be used for interacting with a plugin is depicted in Figure 4.10.

As for the transport to be used to exchange these API messages, we added support to three of those: (i) an HTTP-based REST API, a (ii) WebSocket API, and (iii) a RabbitMQ API. The first two can safely be used directly in a browser, meaning web users can interact with and control a Janus instance directly from the browser. The third one, instead, was explicitly conceived to allow server-side applications to control one or more Janus instance, and handle the web based interaction their own way. All the three transport exchange exactly the same kind of messages, which means they're just different trans-

port for the same protocol. The only difference is in how the REST-based API handles asynchronous notifications, due to the well known limitations in push-based mechanisms in HTTP. As with the WebRTC-related components, we chose to re-use some reliable libraries to support these transports, namely *libmicrohttpd*[7] for the HTTP REST interface, *libwebsockets*[8] for WebSockets and *rabbitmq-c*[9] for RabbitMQ interaction.

WebRTC-related interactions instead work a bit differently. Specifically, although the media will be made available to and from plugins for them to act upon, the actual WebRTC negotiation process is handled directly by the core. This can be explained by the fact that it will be Janus to be responsible for verifying if any connectivity can be established by means of ICE, of creating a secure media channel by means of DTLS and SRTP, and eventually exchanging media and data with WebRTC-compliant endpoints. This process is always the same no matter which plugin or application a WebRTC PeerConnection will be used for, and as such it made sense to have this taken care of at the core level. Nevertheless, plugins are not completely ruled out of the process: specifically, since plugins will be responsible for actually feeding the PeerConnections with data and/or receiving them from browsers and other WebRTC-compliant endpoints, it's very important that they can speak for what is needed at a negotiation level, in order to make sure the media they're going to send and/or receive is compliant with what they can do. As such, whenever a negotiation offer or answer is received by the core to setup a new PeerConnection to attach to a plugin, its SDP gets stripped of all the connectivity and WebRTC-related attributes (see IP addresses, ICE information, DTLS fingerprints, and so on). This "anonymised" SDP will eventually only contain media-related information, e.g., what kind of RTCP feedback is supported, which codecs are being negotiated, their format and so on, that is, exactly everything plugins actually need in order to figure out whether they can indeed establish a communication with the WebRTC

[7] http://www.gnu.org/software/libmicrohttpd/
[8] https://libwebsockets.org/
[9] https://github.com/alanxz/rabbitmq-c

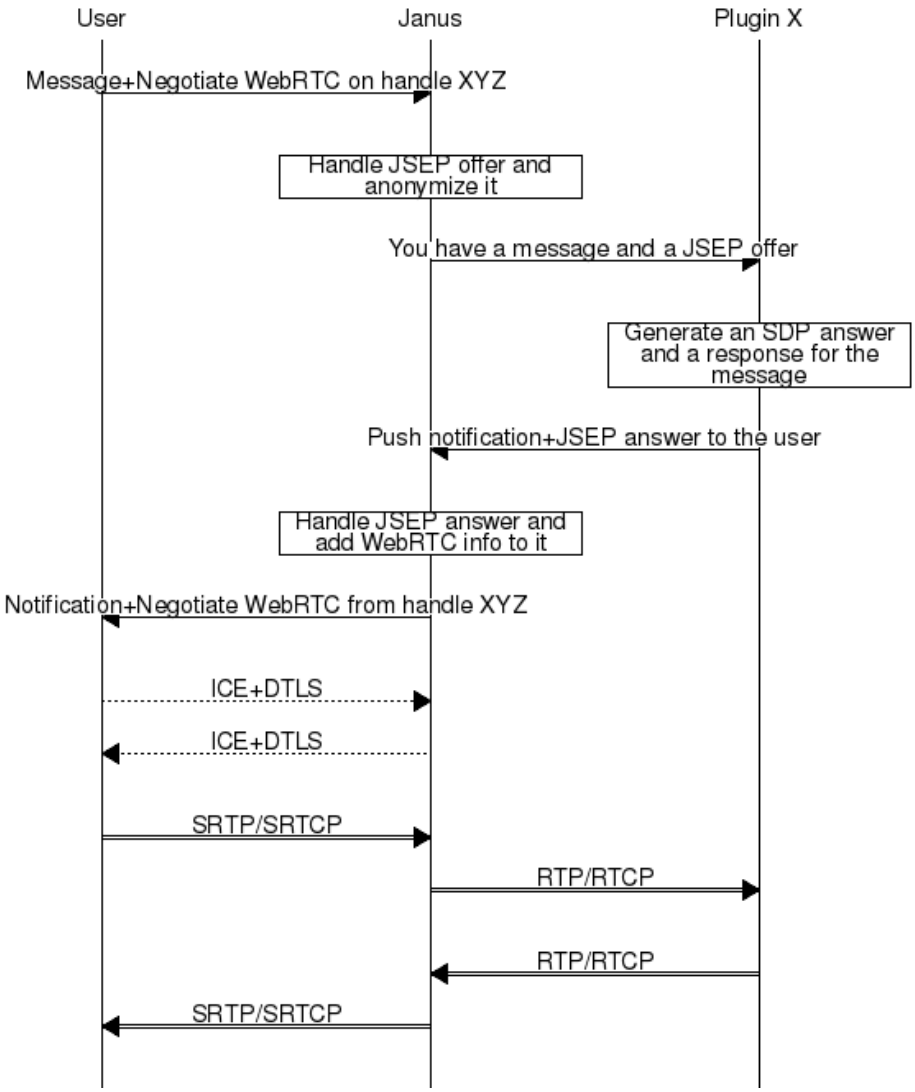


Figure 4.11: Janus API example: negotiating a PeerConnection with a plugin

peer. In the same way, plugins can craft an anonymised answer (if they received an offer previously) or offer (if they're going to initiate the media communication) that will then be "enriched" by Janus with all the SDP attributes needed for a successful setup of a media channel, that is ICE candidates, credentials, DTLS finterprints, SSRC-retaled attributes and so on. The whole process that has just been described is depicted in Figure 4.11, which assumes a *session* and *handle* to a specific plugin have already been setup as per Figure 4.10.

While apparently complicated, as it forces plugins to have to deal with SDP that is notoriously not a very friendly protocol, this approach allows for the maximum flexibility in writing new plugins or covering new scenarios, independently of new codecs or formats that may appear in the future. In fact, since Janus is completely opaque to the media encoding process (no transcoding is done at the core, just relaying of media packets and control messages), pretty much any codec can be successfully negotiated if both the plugin and the WebRTC-compliant endpoint support it, whether or not this codec was known at the time Janus was first implemented.

4.4 Janus as the "last mile": the Streaming plugin

The last sections hightlighted how this new component we designed and implemented, Janus, could be used to bridge two heterogeneous multimedia technologies with each other in an efficient and transparent way. Although due to its flexibility we eventually ended up integrating Janus in several multimedia scenarios we didn't think of initially (more details about this will be provided in Chapter 6), SOLEIL was of course the first and most important scenario we wanted to tackle thanks to this new instrument at our hand.

Specifically, as explained in the previous sections, the target was an implementation that could satisfy the "last mile" requirement, and as such

make the raw RTP streams provided by the relays in the tree topology above available to any interested WebRTC attendee. As such, we implemented an ad-hoc plugin that would take care of multimedia streaming functionality, using an external component as a SOLEIL relay node as the actual source of the media. This plugin was conceived to be able to broadcast this incoming feed to many WebRTC attendees at the same time and without any delays. As such, a single Janus instance acting as "last mile" for a relay node can actually be seen as an additional, and last, level of the tree topology that constitutes the SOLEIL architecture.

From an implementation point of view, the plugin was relatively simple to realise. In fact, as anticipated in Chapter 3, WebRTC does make use of RTP and RTCP as its foundations for the media delivery and control, although in a form that is enriched with additional feedback mechanisms. For what concerns encoding, though, everything works pretty much the same way as with legacy RTP/RTCP-based technologies like SIP. This means that the media coming from SOLEIL was already in a format that was intellegible by WebRTC peers: all they needed was to be enriched with the WebRTC-related layers like ICE and DTLS to be exploitable by browsers. Since the Janus core takes care of this step for us, we just had to make sure every WebRTC peer attaching to a specific streaming session in the Janus streaming plugin could receive the frames: this meant taking care of the "anonymised" SDP negotiation discussed in the previous sections, and exploit the callbacks and methods made available by the Janus plugin API, which allow plugins to ask for the delivery of media frames to a specific peer, besides making media sent by them available to plugins as well. Once done that, all we had to do was properly taking care of the RTCP messaging on both sides of the communication: that is, make sure we interacted with a SOLEIL relay node the way it expects a peer node to do RTCP-wise, while at the same time taking into account feedback coming from browsers to improve their experience.

Figure 4.12 illustrates a typical sequence diagram for the above mentioned interaction: assuming a *session* has been created with Janus and a *handle*

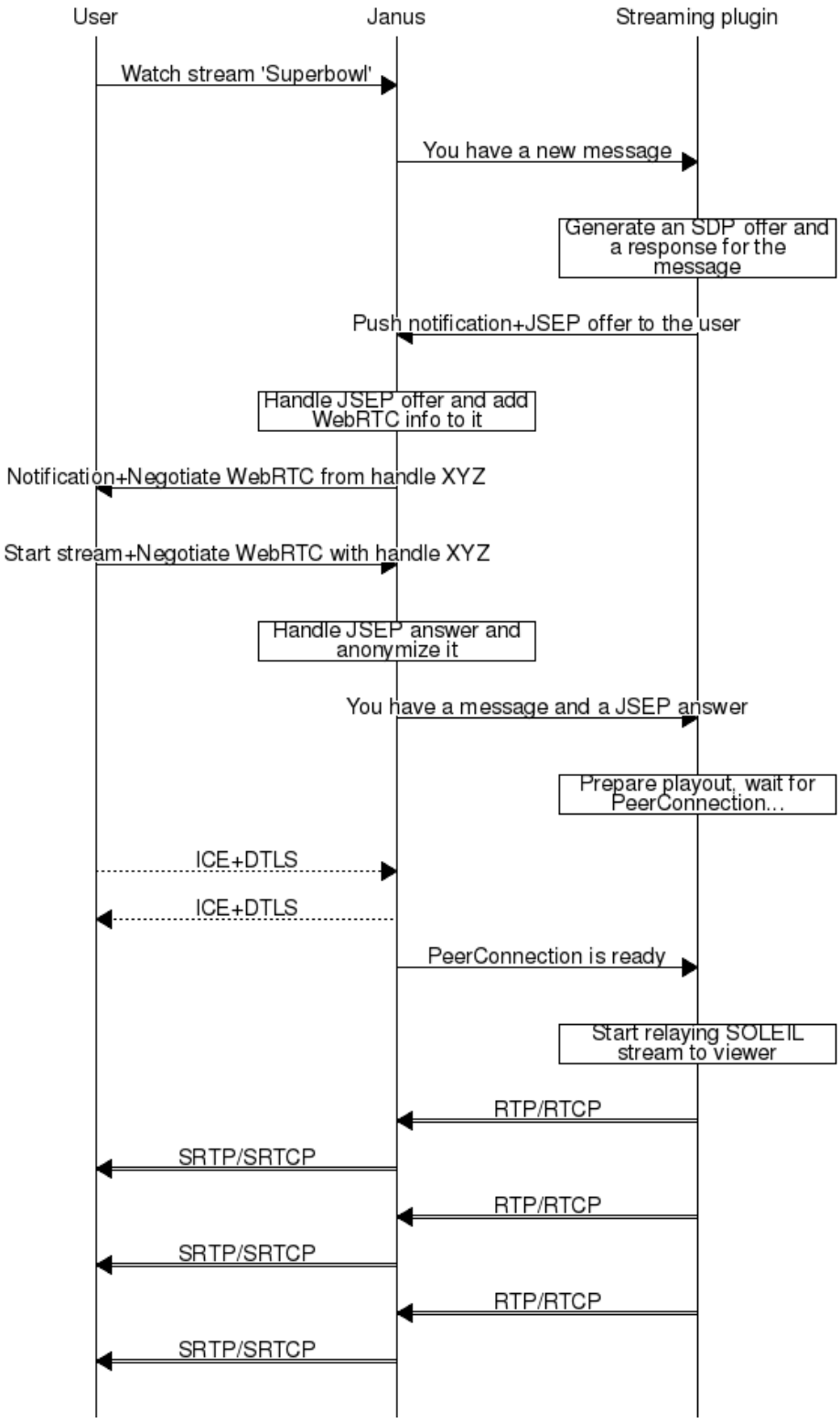


Figure 4.12: Janus Streaming plugin example

attached to the Streaming plugin, the user can, through the Janus API, ask the plugin to reproduce one of the available streams that is being originated by the SOLEIL archutecture in the background. This results in the Streaming plugin forging an SDP that describes the stream, e.g., in terms of codecs and other media-related configuration. This SDP, which is "anonymised" as it doesn't provide any connectivity information, is handled by the core in order to enrich it with WebRTC-related configuration data, i.e., ICE candidates, DTLS fingerprint, etc. Besides, the Janus core also initialises the WebRTC stack, in order to be ready when it's time to set it up with the user. As soon as the complete, enriched SDP is ready, it is sent to the user through the Janus API as a notification, together with any message the plugin may have provided. At this point, the user has a JSEP offer available, meaning it can prepare a JSEP answer to complete the negotiation process. This JSEP answer is sent back to Janus together with a message for the plugin. The Janus core handles the JSEP answer it just received and uses it to start the setup of the WebRTC PeerConnection: specifically, the ICE process for establishing connectivity is started, to which the DTLS handshake will follow. Both the JSEP answer and the message sent by the user are forwarded to the plugin, which will complete the preparation of the playout, waiting for the PeerConnection between the user and Janus to be ready. As soon as it is, the Janus core notifies the plugin about it, which in turn realises the user is ready to receive media. As a result of this final callback, the plugin starts relaying the media it is receiving from the SOLEIL *relay* node to the Janus core, addressing it to new user it is meant for, besides all those who subscribed before. The Janus core, using the secure context and ICE connectivity information it established before, can at this point enrich the plain RTP/RTCP packet with the WebRTC layer, and send it over the PeerConnection to the user, who will in turn decode and render it in its web page.

An overview of the integration of Janus as the *leaf* node in the SOLEIL architecture is depicted in Figure 4.13. As it can be seen in the picture, it

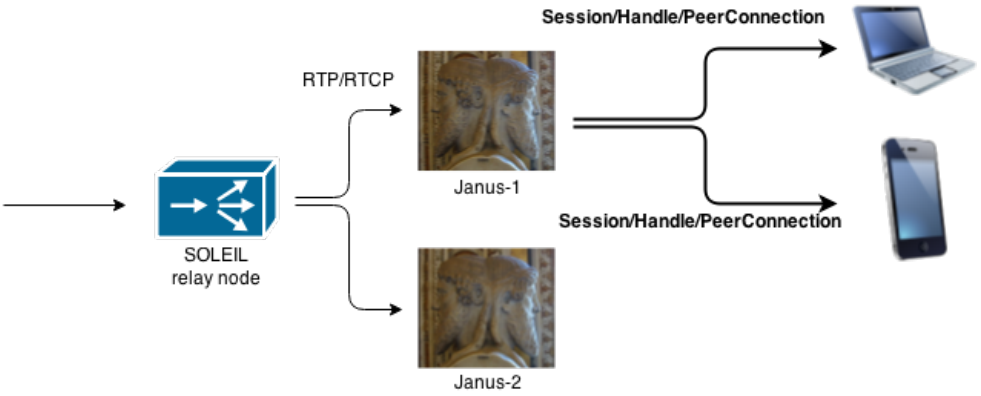


Figure 4.13: Janus as the *leaf* node in SOLEIL: "last mile"

refers to the previously introduced Figure 2.10: with respect to that picture, the green box that acted as a placeholder for the *leaf* node is replaced by a Janus instance; besides, the interactions on both sides are hightlighted as well, with the plain RTP/RTCP coming from the SOLEIL side (a *relay* node) and the *session*, *handle* and PeerConnection management that is instead required on the viewers' side.

As it will be clearer in Chapter 5, this last bit proved very important to improve the quality of experience for end users, especially in the presence of considerable packet loss.

Chapter 5

Experimentation and Measurements

This chapter deals with the experimental campaigns we devised and implemented, in order to evaluate the achieved results from both a functional and performance perspective. Considering the focus of this doctoral thesis on the "last mile" of the SOLEIL infrastructure, all of the campaigns were conceived starting from the assumption that a *relay* node was more or less successfully feeding a Janus instance acting as one of the *leaf* nodes. This allowed us to focus on the effectiveness of Janus as the "last mile" in the architecture, which would in turn allow us to properly dimension an actual deployment in function of the specified target, e.g., in terms of how many viewers should be supported for a live event.

Considering the web-based nature of the experimental campaign, this immediately presented us with the first challenge, that is how we could effectively simulate/emulate several different viewers being interested in receiving a WebRTC-based live event. In fact, the usual tools that are commonly employed for simulating or generating a high volume of media requests cannot be exploited for WebRTC scenarios too: most of these tools only focus on purely web-based technologies, e.g., scenarios that only involve a HTTP- or WebSockets-based interaction that needs to be stressed out somehow; other tools, which are more oriented to the evaluation of real-time multimedia infrastructures, are not viable either, since they're mostly targeted at very

specific technologies as SIP, and do not take into account the several additional requirememements and functionality introduced in WebRTC. For this reason, it was immediately obvious to us that we could only partly simulate the client side of the tests, and that we would probably need to rely on actual browser instances to act as the viewers in our scenarios. Of course, assuming a person would be physically controlling these browser instances was out of the question, as this would have substantially affected our capability of scaling the testing campaigns to a high volume of requests we envisioned.

In order to take this requirement into account, we eventually decided to rely on an approach that would allow us to remotely control a pool of headless browser instances. We found in the Selenium Framework [50] the perfect tool for the purpose, as it allowed us to do exactly what we needed, that is deploy multiple browser instances on server machines, and control them remotely and in a programmable way in order to have them execute the web-based interactions required to setup a multimedia session with Janus and subscribe to a stream originated via SOLEIL. While quite flexible, this approach also allowed to actually have very realistic results, as all streams would be actually transported as if real users were actually requesting the streams.

To take into account the Quality of Experience (QoE) too, we also included a subset of real users to the simulated ones, in order to evaluate the perceived quality in a browser used in a regular way. As it will be clearer in the next sections, and more specifically in Section 5.2 and Section 5.3, this allowed us to identify a few areas where some fixes and enhancements were due, especially with respect to the management of lost packets on either side of the communication.

At the time of writing, the results presented in this section were also the basis for a paper, called “Performance analysis of the Janus WebRTC gateway” [51], that has been recently accepted at the “All-Web real-time Systems” (AWeS 2015) workshop and is pending publication.

5.1 Design of the experimentation campaign

Before moving to the results of the experimentation, and especially to what we identified as possible improvements on the platform, it is worthwhile to describe the setup we prepared in order to assess the Janus performance.

The experiments we ran were focused on the server-side CPU, memory and bandwidth consumption. The testbed we set-up envisaged 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, as anticipated, we leveraged the Selenium 2.0 framework[1] 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, as depicted in Figure 5.1. 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 what at the time of testing was the latest stable version of Firefox (35.0.1) since, unlike Chrome, it did not implement the audio-video BUNDLE techniques described in Section 3.7.3: this means that audio and video streams would not share the same ICE component, 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, as no actual microphone and webcam access would be needed, while still resulting in real audio and video streams being sent and received.

All tests were conducted over a dedicated gigabit LAN.

The following subsections present the performance figures devised by exploiting four of the Janus plugins, namely the *videoroom* plugin in Sec-

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

Figure 5.1: Selenium grid

tion 5.1.1, the *audiobridge* plugin in Section 5.1.2, the *SIP* plugin in Section 5.1.3 and eventually the *streaming* plugin, the one of the most interest for the SOLEIL platform, in Section 5.1.4. The reason for not analyzing just the *streaming* plugin was simple: we wanted to assess the performance of Janus in general, and consider whether or not further optimizations to the role of "last mile" within the SOLEIL architecture could be envisaged by involving other plugins. Not all plugins were tested, as we were mostly interested in evaluating the performance of different aspects of the platform, especially with respect to relaying vs. mixing in the Janus framework. Besides, most of the other plugins could actually be considered as sub-cases of the ones we studied: for instance, the *videocall* plugin can be seen as two-participants video rooms when looking at performance, just as the *SIP* plugin, from a media relay perspective, is not unlike the *streaming* plugin we analyzed.

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 streams, or the rooms in case conferencing scenarios were addressed; (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 com-

pare different possible approaches to provide the same service, as explained in better detail in Section 5.1.5.

5.1.1 Testing the *videoroom* plugin

In this subsection we present the performance figures derived from stress testing of the Janus *videoroom* plugin, which can be seen as some sort of audio/video *router*, forwarding media streams from one or more users to other participants in a programmable way. Two different roles were envisaged: *subscribers* only received remote streams, while *publishers* both sent and received Opus-encoded audio and VP8-encoded video. This is not unlike what a Janus instance has to take care of when involved in a SOLEIL scenario, with the difference being that in this case both broadcasters and viewers were WebRTC-based, and were involved in a communication scenario (e.g., web conferencing, e-learning, etc.). During our tests, we first made 10 publishers join the videoroom, then we let 140 subscribers join as well. As already anticipated, the *videoroom* plugin implements the SFU logic, hence the 150 participants envisaged by this scenario corresponded to 1500 PeerConnnections maintained by Janus. Figure 5.2 shows the CPU utilization level and memory occupation in such a scenario. We noticed how memory load increased 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 increased in a roughly linear manner with the number of subscribers. We noticed 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, did not follow this pattern and always kept the same

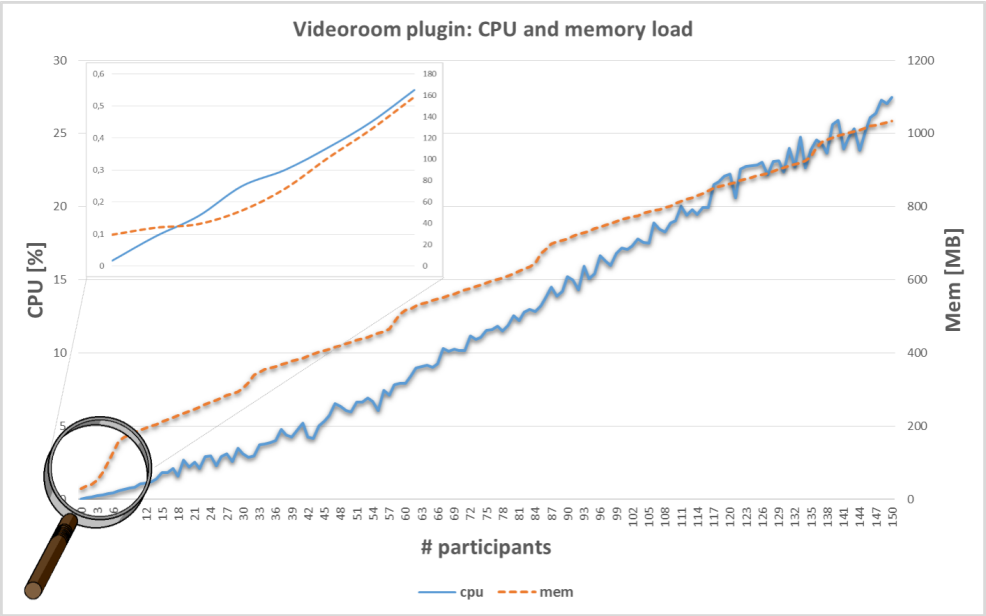


Figure 5.2: Videoroom plugin: CPU and memory

growth trend.

Figure 5.3 plots the bandwidth usage. Each publisher sent both Opus audio and a 640x480 VP8 video, generating around 180kbit/s towards the server. Hence, the downlink traffic generated by publishers growed linearly to 1.8Mbit/s. This value slightly increased toward 3Mbit/s as more and more subscribers joined, accounting for signalling traffic, RTCP feedback and possible retransmissions on each PeerConnection, as it will be discussed in Section 5.2. On the other hand, uplink traffic had a quadratic evolution with the number of publishers, for the same reasons already discussed for memory load. Then, it increased almost linearly with the number of subscribers.

5.1.2 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. In fact, such operations are in general quite demanding in terms of CPU resources: 200 participants in a wideband (16kHz) audio mix took up around 73% of CPU as depicted

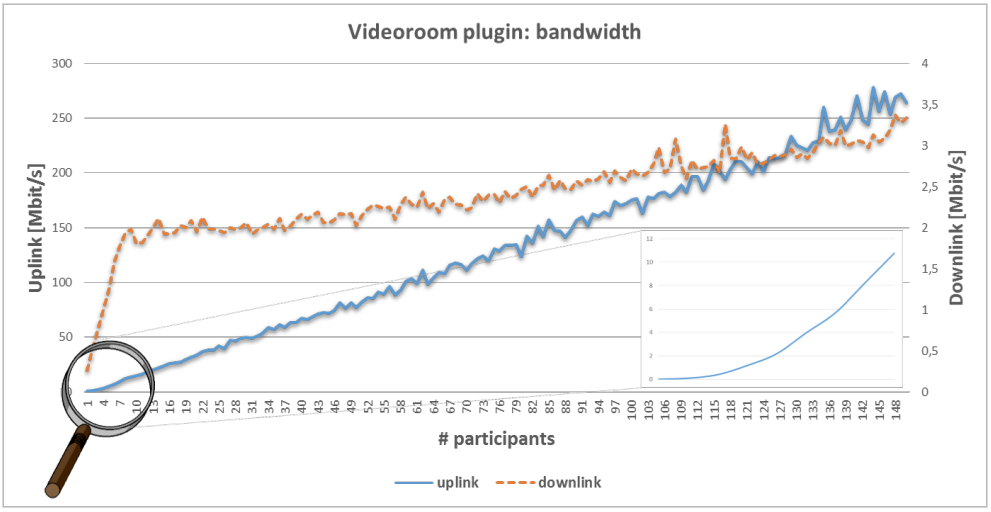


Figure 5.3: Videoroom plugin: bandwidth

in Figure 5.4. Memory occupation, instead, kept relatively small, as it did not exceed 150 MB.

Finally, Figure 5.5 shows how uplink and downlink traffic followed the same trend, as expected.

5.1.3 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 Figure 5.6 shows, the CPU level increased linearly. 330 participants took up around 22% of the CPU. The memory consumption, instead, presented the same behavior already discussed in Section 5.1.1.

For what concerns bandwidth, uplink and downlink levels were exactly the same, as expected and sketched in Figure 5.7.

5.1.4 Testing the *streaming* plugin

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

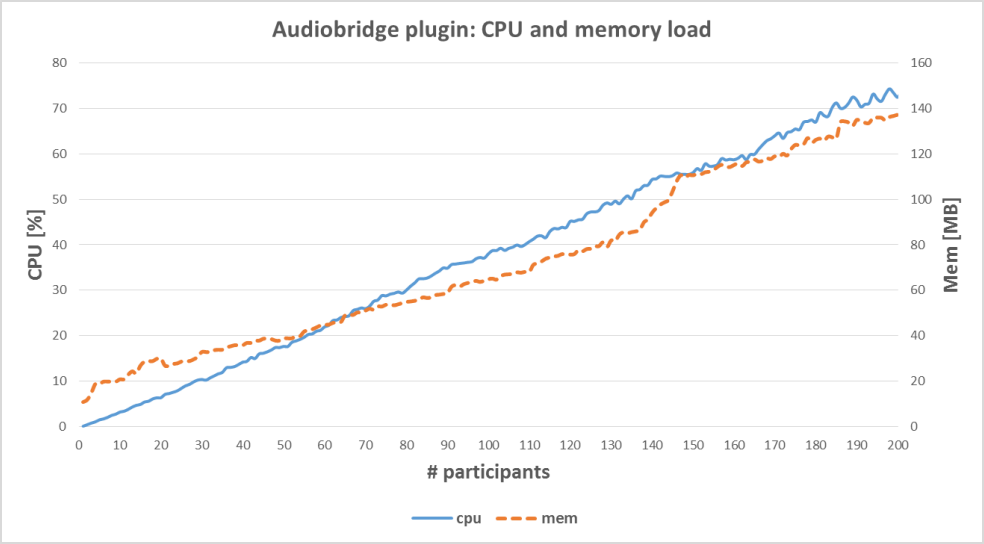


Figure 5.4: Audiobridge plugin: CPU and memory

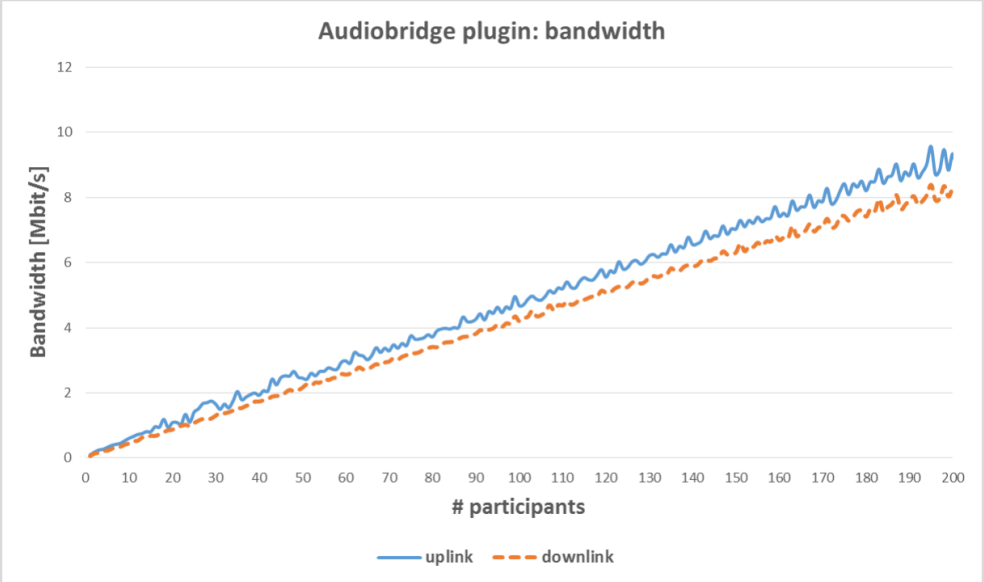


Figure 5.5: Audiobridge plugins: bandwidth

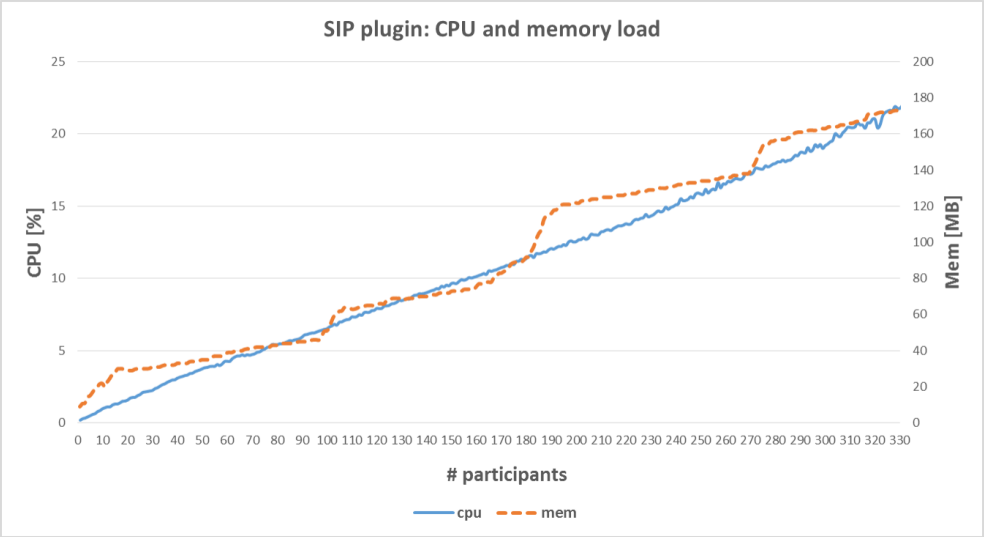


Figure 5.6: SIP plugin: CPU and memory

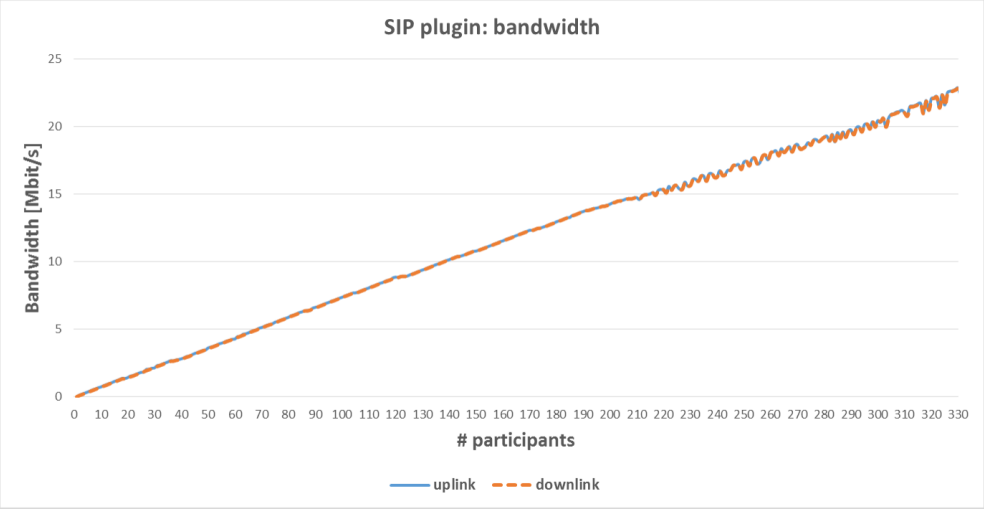


Figure 5.7: SIP plugin: bandwidth

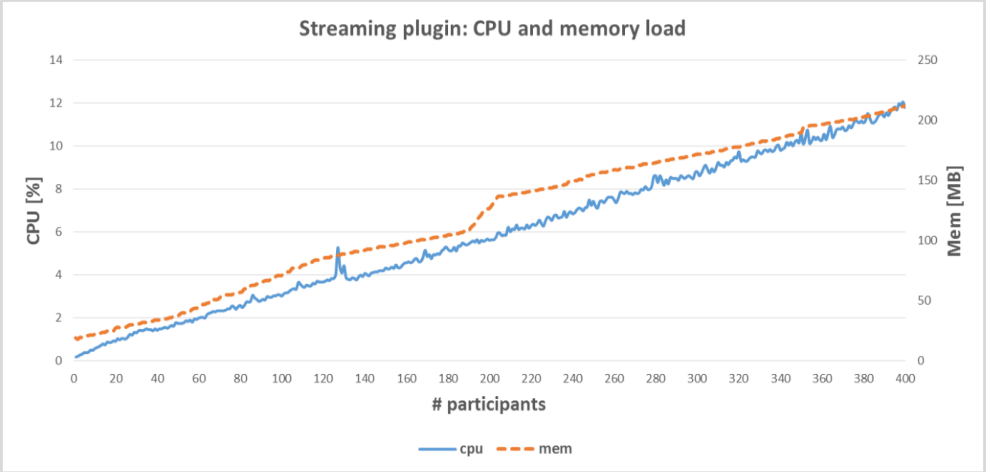


Figure 5.8: Streaming plugin: CPU and memory

Figure 5.8.

5.1.5 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 realised by exploiting either the videoroom, or the audiobridge or the SIP plugin, with different performance attained as we show in the following. As it will be described in the next subsections, our aim was mostly to compare the MCU and SFU approach. While both approaches try and put in contact more participants at the same time, the way they do this differ. Specifically, while an MCU usually mixes the contributions it receives in order to provide participants with a single stream that is actually a composition of them all, an SFU instead simply relays incoming streams according to a programmable logic (e.g., who's allowed to contribute and who's allowed to receive what).

MCU vs. SFU

In this first comparative example, we implemented the aforementioned audio conference service by leveraging the videoroom and audiobridge plugins. In

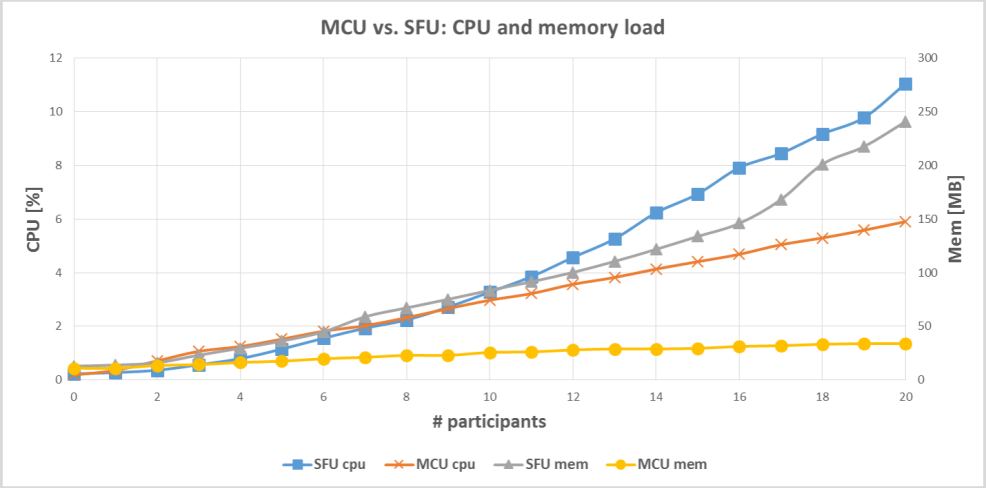


Figure 5.9: MCU vs. SFU: CPU and memory

the former case, of course, we disabled video functionality to obtain an audio-only SFU, i.e., a component that would just relay audio packets to all the intended recipients instead of mixing them. The latter, as the name suggests, implements an audio-only MCU instead. Figure 5.9 shows CPU and memory loads of the two approaches. CPU evolution over time is also depicted in Figure 5.10, while bandwidth consumption is shown in Figure 5.11.

The results demonstrate that, using a wideband (16kHz) mixer in the audiobridge, MCU won over SFU whenever the number of participants went beyond 8, as it ostensibly required less CPU, memory, and most importantly bandwidth, as much less PeerConnections were needed. 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 enough operation that could be more or less easily performed without taking too many CPU cycles. It is worth remarking that the results provided do not take into account video flows. Video mixing, in fact, is a quite heavier task, which would probably lead to completely different outcomes.

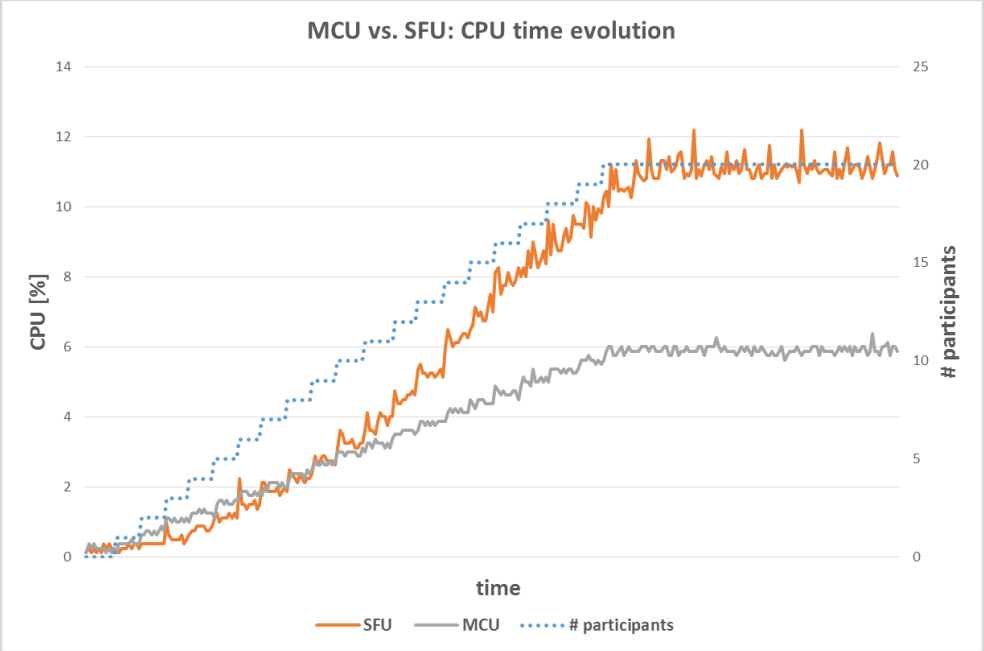


Figure 5.10: MCU vs. SFU: CPU time evolution

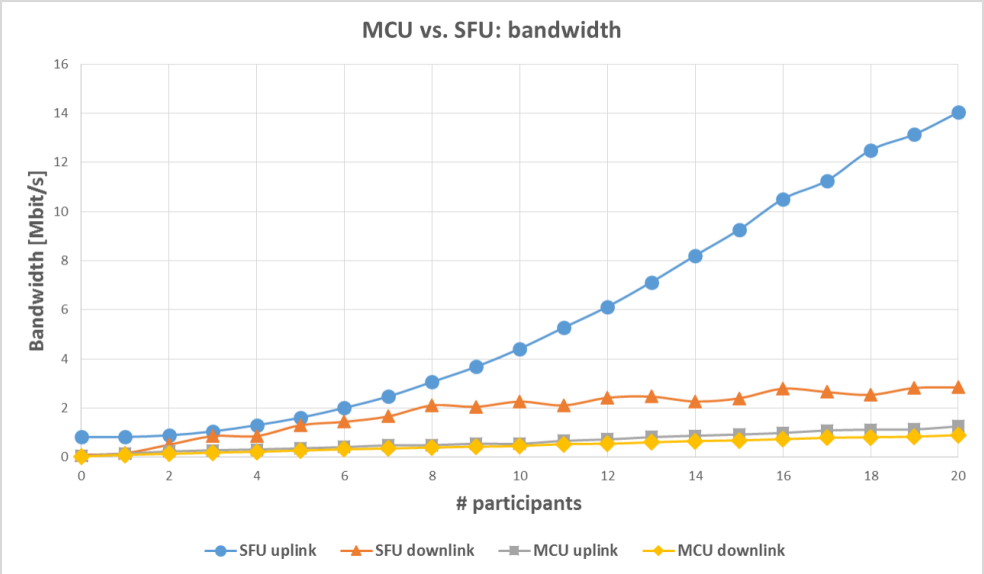


Figure 5.11: MCU vs. SFU: bandwidth

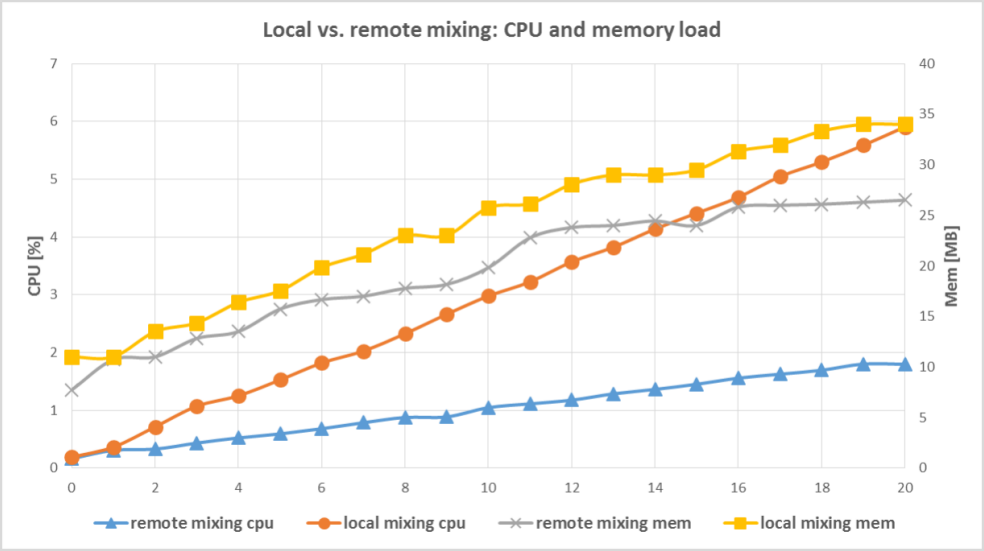


Figure 5.12: Local vs. remote mixing: CPU and memory

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 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 Figure 5.12, remote mixing proved 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 Figure 5.13.

5.2 Improving the QoE: NACKs and retransmissions

The figures and evaluations presented in the previous sections were the ones we collected after several iterations. Specifically, after each test we tried to

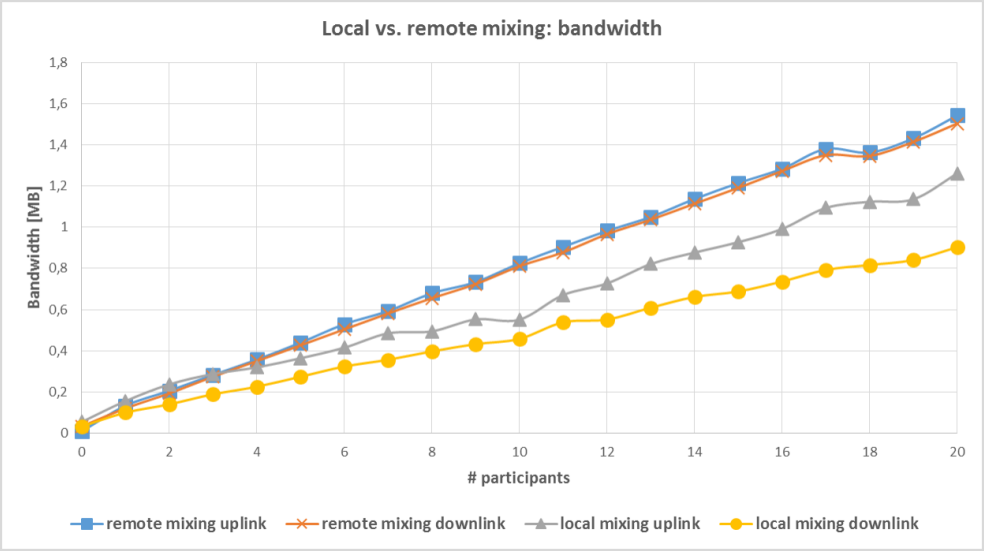


Figure 5.13: Local vs. remote mixing: bandwidth

identify potential issues on both the client and server sides, in order to verify whether or not any improvement could be done on the platform.

The first experimental campaigns, while effective from both a functional and performance point of view, immediately highlighted a potentially serious issue in terms of Quality of Experience. From time to time, in fact, real users would experience issues in how the video they requested was presented to them as part of the WebRTC session. Specifically, they would sometimes experience heavy artifacts on the video, and at times the video would freeze completely. This meant that, while we could be reassured by the ability by Janus to effectively broadcast the incoming SOLEIL stream to a lot of users at the same time, there also were issues that prevented this stream from being actually experienced the way it was supposed to.

By studying the results of the campaigns and some captured data, we identified the main cause of issue in some packets being lost between the Janus instance and the affected browser instance. This was indeed to be expected, since RTP makes use of an unreliable protocol as UDP as its transport. This is a choice made by design, as UDP allows for a much more timely and effective delivery of media packets, no congestion or flow control being

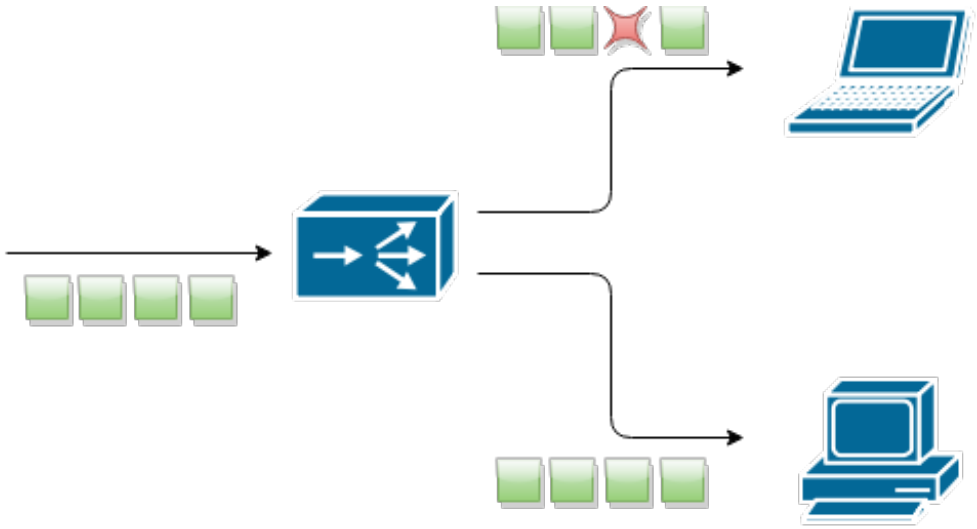


Figure 5.14: Packet loss on a viewer: local impact

involved; as a result, though, this can also lead to RTP packets being lost in transmission, especially in case of congestion, and this packet loss, happening on an unreliable channel, is by default unaccounted for.

This is not entirely an undesirable feature, especially when the main target is making sure the media packets get to the other side as fast as possible. In particular, this is less of an issue as far as audio is concerned, since in that case the audio artifacts or holes are in general more acceptable from a user experience point of view. Things get more complicated when video is involved, though, since video is a visual medium which makes issues that may arise much more evident. Specifically, packet loss on a video stream can easily lead to more or less serious problems in the rendering, ranging from simple artifacts in the video playout to complete freezes of the video image, e.g., whenever the packet loss affects important packets like those carrying key frames. Since the main purpose of Janus as a *leaf* node in SOLEIL is providing a good quality monodirectional stream in real-time, such issues are less tolerable than when they happen, for instance, in a regular video call. As such, some mechanism to cope with this packet loss in a timely way, that is without introducing too much latency, and possibly without affecting the experience for other users, can be desireable.

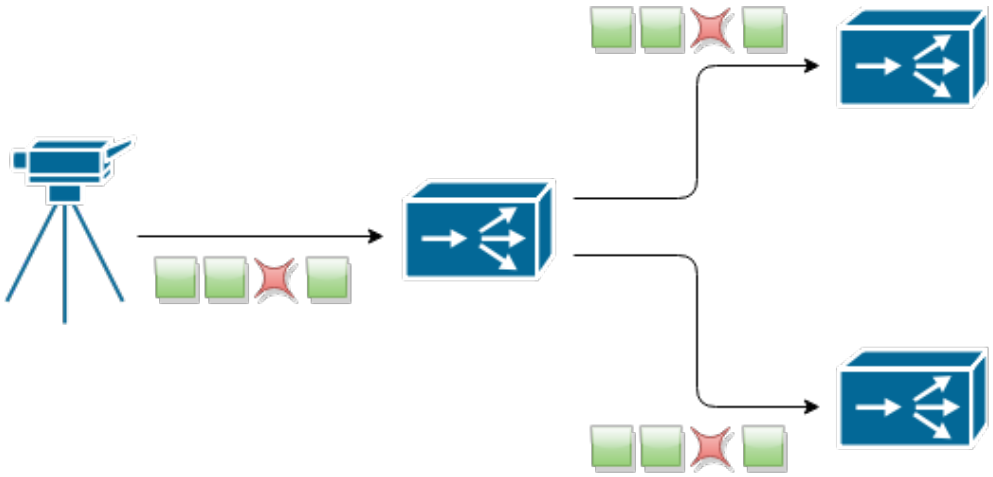


Figure 5.15: Packet loss on a broadcaster: global impact

Besides, the same issue may have either a local or a global impact, depending on where this is happening. In fact, if everything's fine within the SOLEIL infrastructures and all packets are available down to the *leaf* node, a lost packet between such a node and one of the viewers indicates a problem that may affect a specific path, and as such is probably only limited to that users and a few more. Other viewers may or may not be affected by packet loss, which means that in this case the impact of the issue is local and more manageable, as depicted in Figure 5.14. On the other hand, if the *leaf* node is being used to allow a WebRTC user to actively broadcast a stream through SOLEIL, packet loss may be happening there as well. In this case, a packet being lost between the broadcaster and the *leaf* node would have a much greater impact on the overall experience, as the same loss would be propagated within the whole SOLEIL infrastructure, as depicted in Figure 5.15, and as thus affect the user experience of all viewers, no matter where they're attached.

That said, this problem is indeed known by the IETF community, and was addressed within the framework of the standardisation efforts. Specifically, the IETF community worked on some feedback mechanism that would allow a user to notify their peer about specific packets being lost in transmission: this way, the peer could decide whether or not to retransmit some

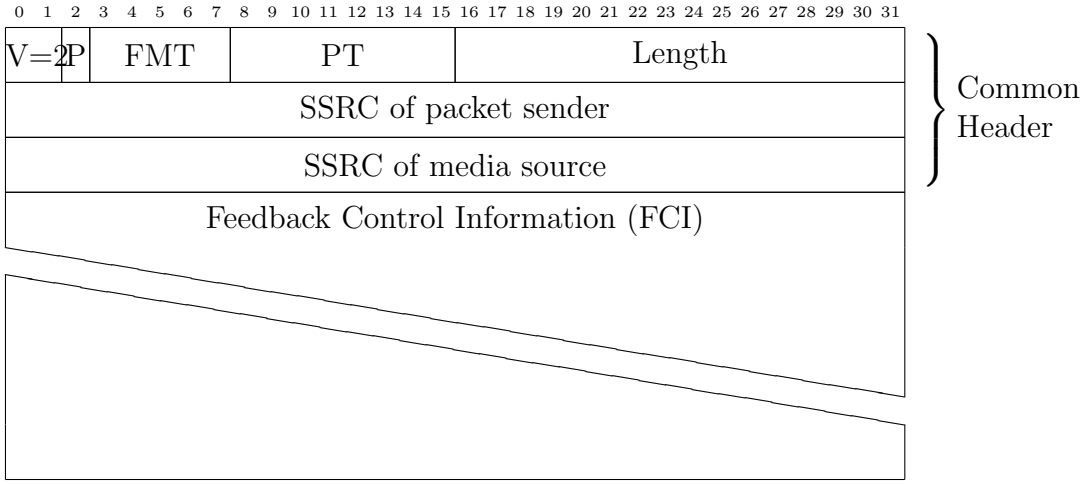


Figure 5.16: Common Packet Format for RTCP Feedback Messages

or all of the lost packets, in order to allow the affected user to have access to the missing pieces of information before rendering the streams. Considering the existence of an RTCP control channel coupled with RTP sessions, it made perfect sense to design this feedback mechanism as part of the RTCP messaging. In particular, a new feedback message called Negative Acknowledgment (NACK) [23] was specified for the purpose as an extension to the generic feedback mechanics defined in RTCP. More in detail, the generic semantics for RTCP feedback messages are depicted in Figure 5.16, while how this is customised with respect to the Feedback Control Information (FCI) for NACKs is illustrated in Figure 5.17 instead.

As depicted in Figure 5.16, All RTCP feecback messages basically share the same pieces of information: apart from the common RTCP header, they need to provide the synchronization source identifiers (SSRC) of both the originator of the packet (who in the session is sending this feedback) and the media source that this piece of feedback information is related to. This allows the recipient of the feedback to be aware the message is meant for them, and that they should act on it if needed. Of course, considering SSRCs are involved, they should be handled accordingly in case a manipulation of those is involved, as specified in 3.4.1. These feedack messages are then specialised

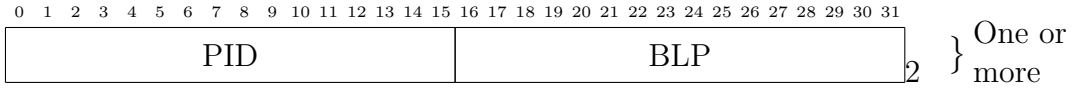


Figure 5.17: Syntax for the Generic NACK message

in function of the *PT* and *FMT* information in the header: according to their values, the FCI assumes a different meaning and can have a different syntax. This allows for an easy and effective way to extend feedback messages by envisaging an opaque payload.

This is exactly what is done with respect to the specialization of feedback messages for NACKs. Specifically, whenever *PT* equals to 205 (the value of *RTCPFB*, which stands for generic feedback on the transport layer) and *FMT* equals to 1 (the unique identifier for NACKs), then the FCI assumes a specific meaning and the NACK syntax is used to interpret the payload. The specific syntax of NACKs is depicted in Figure 5.17. Each word is composed of two different parts: (i) a Packet ID (PID) that is used to specify a lost packet by its sequence number, and (ii) a bitmask of the following lost packets (BLP) using the PID as a base. This allows for a synthetic report of multiple lost packets in an easy way. Considering that more packets than those a single *PID* and *BLP* couple may represent, more words can be part of the same FCI: the length in the common header would be updated accordingly.

This feedback mechanism was exactly what was needed within Janus to handle the packet losses affecting the user experience, especially considering NACKs are part of the recommendations within the WebRTC specification. As such, we implemented support for the reception of NACK feedback from clients, in order to retransmit lost packets for them as needed. Such retransmissions were implemented following the related specification [52], which describes an RTP Retransmission Payload specifically designed for this purpose. At the same time, we implemented support for the generation of NACK feedback as well: this allowed us to detect packet loss from a broadcaster, and promptly react by asking for a retransmission of the missed packets. This way, the retransmitted packets could be injected in the SOLEIL archi-

tecture, and as a consequence prevent the global impact they might have on users.

5.3 A further instrument for the QoE: Simulcast

The enhancements implemented as described in Section 5.2 did indeed improve the performance and user experiences, as we could verify in our updated experimental campaign. At the expenses of a slightly increased bandwidth consumption, that is to be expected as a result of some retransmissions taking place, the experience for users in terms of video artifacts or freezes was greatly improved thanks to the redundant information made available by the *leaf* node.

That said, we did indeed notice that, while NACKs did indeed help in coping with the packet loss issue, there were a few cases were they actually made things worse, rather than better. This was the case for viewers on particularly slow or ill-performing networks. The experience for these viewers, who were trying to access a stream whose bandwidth exceeded the throughput they could handle on their networks, inevitably resulted in a considerable packet loss. In this case, trying to cope with packet loss with NACKs and retransmissions actually made things worse, as the insufficient bandwidth for the stream was further clogged by the retransmitted packets, which most of the times were lost as well. Eventually, this resulted in a completely broken experience for those viewers, as it was a situation from where they could not recover without doing something other than NACK the lost packets. This scenario is depicted in Figure 5.18 where a simplified view of a SOLEIL architecture hightlights the difficulties of one of the viewers in getting a high resolution stream being originated by a broadcaster.

There are, in principle, different ways to cope with this situation. One of those could be to inform the broadcaster of such difficulties for one or more viewers, and instruct it to lower the bitrate of the encoding in order to limit

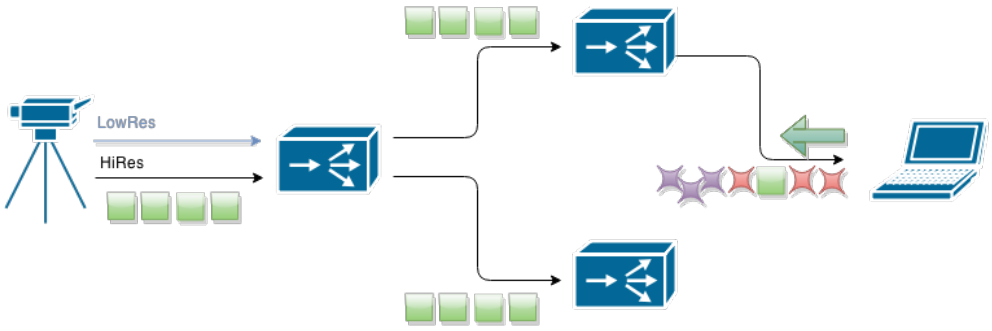


Figure 5.18: Too many retransmissions: making things worse

it to the point of it being more easily accessible to users in problematic networks. While this could be an effective solution, this is often not a desireable approach for several reasons. In fact, as discussed when first introducing the SOLEIL architecture, the stream made available by a brooadcaster is actually shared among all of the interested viewers, according to its descent as a "waterfall" through all the *relay* nodes and down to the *leaf* nodes making it eventually available to viewers themselves. This means that reducing the bitrate on the encoder reduces the perceived quality not only for those viewers who could not cope with the previous bitrate, and would as such benefit from such an operation, but also for those who were not experiencing issues at all, who may very well be the majority of the audience. As such, this approach is usually not taken into account, or when it is, it is actually enforced only up to a certain threshold that is considered the minimum acceptable quality for a stream.

A different solution that is typically considered is to dedicate a separate encoder for the flawed viewer, e.g., to reduce the bitrate just for the viewer that is experiencing issues. This does indeed address the concern previously stated about the shared nature of the encoding in the broadcaster, as in this case the viewers who were not experiencing issues will not be affected and will keep on receiving the higher quality stream, while the user in a problematic network will receive an ad-hoc, lower quality and bitrate stream instead, e.g., be something they'll be able to receive and display without issues. That said, such an approach assumes that transcoding can take place somewhere in the

network, e.g., in the SOLEIL architecture in one of the nodes. Anyway, this was explicitly ruled out by design when the SOLEIL architecture was first devised for a couple of important reasons. In fact, while transcoding allows for a per-viewer content adaptation which can in some cases be desirable, it also adds a considerable latency, as the stream can not just be relayed but needs to be processed instead. Besides, transcoding is a quite heavy operation, CPU-wise, and can definitely affect scalability whenever multiple viewers need the same kind of care from the platform. As such, this solution is not viable either, and we chose not to pursue it within our framework.

After studying the possible alternatives and evaluating pros and cons of each, we eventually came to the conclusion that we'd have to rely on a solution that basically constituted a middle ground among those described above. Specifically, we decided to rely on an approach called Simulcasting to address the issue in a scalable way. This approach still works under the assumption that different versions of the same stream are available, e.g., a higher and a lower quality version of the same feed, which just as in the previous solutions allows viewers which are experiencing problems to switch to a more apt stream than the one they're receiving. The difference is that these different versions are all originated at the broadcaster side, and no transcoding or shared decrease of quality is envisioned in any part of the platform: it's the broadcaster that, in the capture and encoding process, generates different versions of the same stream at different bitrates and qualities, and injects them all within the SOLEIL infrastructure. This may or may not be done in response to problems in a specific viewer instance: most of the times it's not, and is provisioned in advance, which means much less versions of the same streams are needed with respect to the pure transcoding approach. From a SOLEIL perspective everything works exactly as before: in fact, these different versions of the stream are basically treated in a completely opaque way as different streams, which means they can be propagated among the relaying infrastructure separately from each other. From an application perspective, instead, these streams can be related to each other, meaning that each viewer

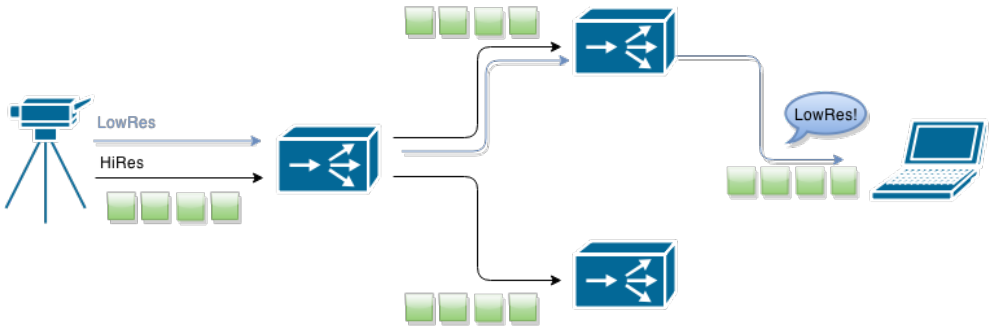


Figure 5.19: Simulcast: multiple versions of the same stream

can dynamically choose to switch among the available streams, whether voluntarily (e.g., because the user specifically requested a higher/lower quality version) or not (e.g., it's the *leaf* node that, in function of the feedback received, decides that a viewer must be switched to a different version of the stream automatically). This approach is depicted in Figure 5.19, which is basically the following step to the issue identified in Figure 5.18: the broadcaster makes available an alternative, lower bitrate version of the same live event; this alternative version is notified to both the *leaf* nodes and the viewers, and the viewer on the problematic network is switched to the lower quality version of the stream, which does not result in the unrecoverable packet loss anymore.

Chapter 6

Conclusions and next steps

The work carried out during the last three years as Ph.D. Student has been focused on large-scale multimedia streaming over next-generation IP networks. The ultimate goal was to design an architecture having high scalability requirements, dynamic "deployability" and web-access in mind, while taking into account the outcome of the standardisation efforts conducted by the World Wide Web Consortium and the Internet Engineering Task Force.

This lead my Ph.D. Student colleague Tobia Castaldi and I to design an architecture that would address all those requirements, which we called SOLEIL, as in *Streaming Of Large scale Events over Internet cLouds* and introduced in Chapter 2. Then, as part of our respective doctoral activities, we started working on separate aspects of the architecture, in order to eventually merge those efforts in a unified platform that would implement the scenarios we envisaged during the design process.

In particular, while Tobia Castaldi devoted himself to the cloud-based challenges of the platform, and in particular on the deployability and dynamic reorganization of a SOLEIL network, I focused on the real-time multimedia aspects, that is the distribution of flows among the involved nodes and, more specifically, the delivery of these streams to end-users. As for the former, I identified the required separation of responsibilities among the different nodes that could constitute a SOLEIL network. The latter, instead, saw the design and realization of a gateway component, called Janus, that could act

both as a "translator" between different technologies, namely SOLEIL and WebRTC, and as a "last mile" that end-users could refer to in order to access the flows made available in a SOLEIL instance. Both these activities lead me to study and contribute to the standardisation efforts within the IETF, in particular within the context of the Media Server Control (MEDIACTRL), Sip Traversal Required for Applications to Work (STRAW) and Real-Time Communication in WEB-browsers (RTCWEB) working groups. While the standardisation activities, with special attention to WebRTC and to their impact on legacy infrastructures, were discussed in Chapter 3, the actual implementation of the "last mile" and the Janus framework was described in detail in Chapter 4.

All the design and implementation efforts were eventually evaluated through a comprehensive experimental campaign, aimed to assess both the functional requirements and the performance of the Janus component. This experimental campaign, described in Chapter 5, convinced us that the efforts we had devoted to this process had been well addressed, and that the milestones we wanted to tackle had been successfully reached.

At this stage, only one last step remains within this research effort, that is actually merging the successful results both my colleague Tobia Castaldi and I reached within our respective responsibility areas as part of a wider research activity. This will eventually allow us to focus on an important aspect of this activity, that is the industrialization of these efforts.

As to this last aspect, it's worthwile to mention that part of the activities I conducted during my doctoral efforts has already been subject to industrialization, and has proven quite fruitful also from a strictly commercial point of view. In fact, as anticipated in Chapter 4, while the main target of the design process was indeed to provide the means for a successful WebRTC translator that could implement multimedia streaming, Janus was actually conceived as a general-purpose WebRTC gateway. This lead us to the definition of a completely modular architecture, composed of a core and multiple, extensible, plugins implementing the actual application logic. The motivation for this

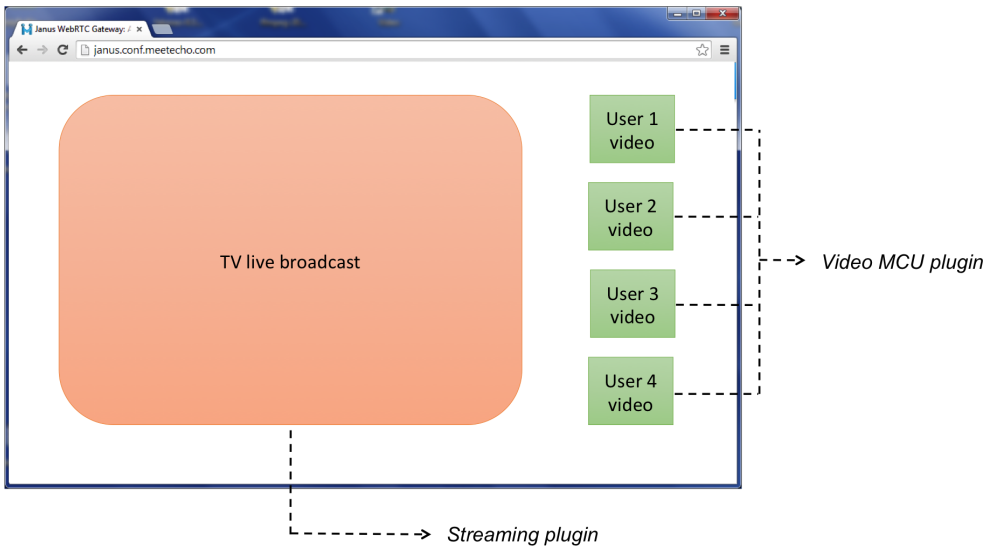


Figure 6.1: Combining different Janus plugins: Social TV

came from our willingness to design something that could be easily extended and enhanced in a modular way in the future, without having to re-design or re-write any part from scratch, thus separating the aspects strictly related to the WebRTC interactions from those related to legacy technologies or applicative considerations.

This choice proved much more successful than we initially thought, as even during my doctoral activities on SOLEIL several opportunities presented to start working on related multimedia research efforts that were not strictly related to multimedia streaming. Figure 6.1. and Figure 6.2 sketch a couple of scenarios that can, for instance, be implemented by means of Janus and some of its plugins available out-of-the-box, combined in a creative way.

As a matter of fact, these research efforts also turned into actual consulting activities within the framework of my involvement in the Meetecho spinoff. The huge flexibility provided by Janus and its modular architecture, in fact, allowed us to address multimedia scenarios and applications we didn't even think of when first working on the framework. For instance, as part of our consulting activities we have already helped several companies and developers implement, with the help of Janus and existing or new

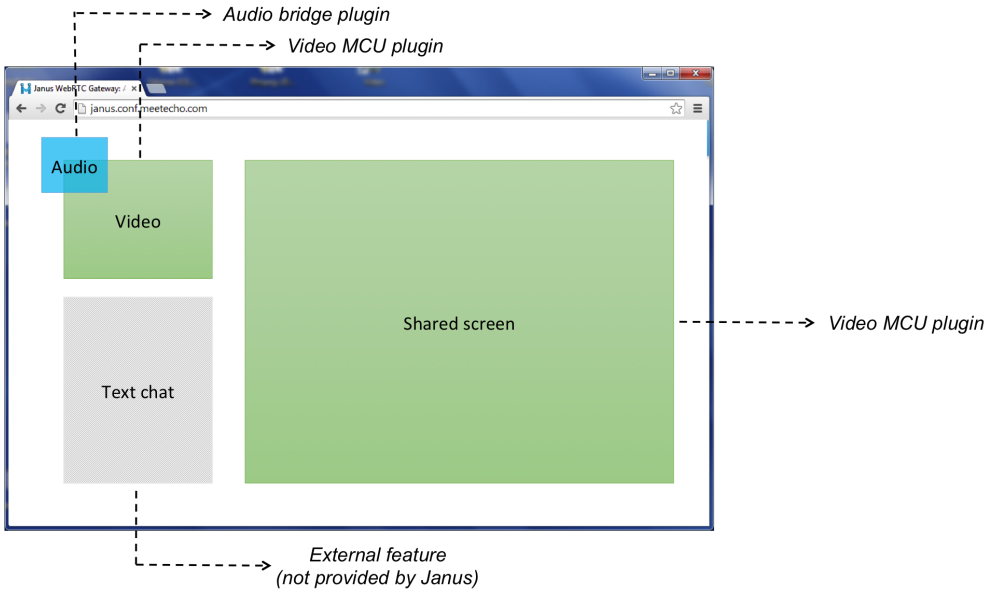


Figure 6.2: Combining different Janus plugins: Screensharing with Q&A

plugins, web conferencing scenarios, collaboration platforms in the broader sense, e-learning and webinar frameworks, Social TV applications and even scenarios that belong to the Internet of Things (IoT) and Home Automation ecosystem.

At the same time, Janus has also proved a quite valuable instrument to third party developers and researchers. As anticipated, in fact, we released Janus as a completely open source component, which allowed implementors from all over the world to take advantage of its functionality and sometimes even contribute back to the project with feedback, fixes and new ideas.

An interesting example of this can be found in the "Jumping Janus" effort. As part of a WebRTC contest called *WebRTCfest*[1] and sponsored by Parrot, a company specialised in building drones, a challenge was issued among implementors to write an application that could take control over a Parrot drone over WebRTC. Tim Panton and Neil Stratford, two well known implementors within the real-time multimedia world, chose Janus as the foun-

[1] http://webrtcfest.com/

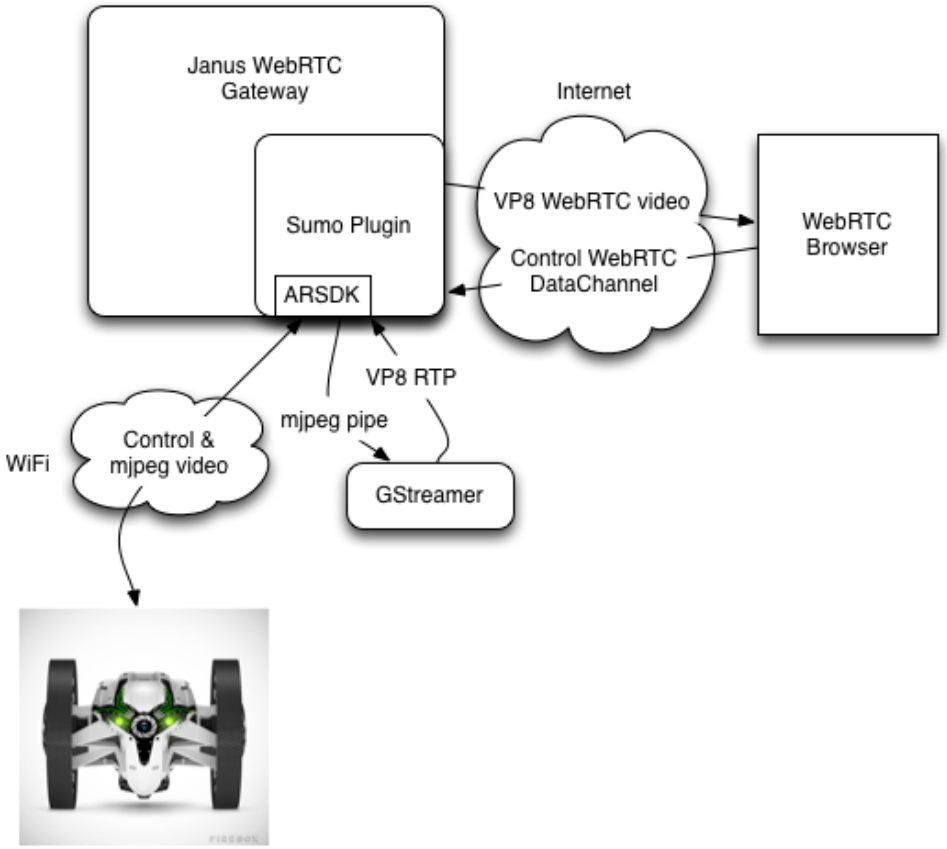


Figure 6.3: The Jumping Janus: overall architecture

dation to build their implementation[2]. The resulting architecture, depicted in Figure 6.3, saw Janus as an intermediary between WebRTC endpoints (in this case, a browser) and the non-WebRTC framework the Parrot drone was based upon. In order to do so, they implemented a new plugin that would implement the *ARSDK* API to interact with the drone programmatically, and took advantage of the data channel and streaming functionality made available by Janus itself.

The efforts eventually lead to a working prototype, a demo of which was publicly made available as a video on YouTube (see Figure 6.4). This prototype effectively allowed browser users to watch a live video as captured by

[2]https://babyis60.wordpress.com/2015/02/04/the-jumping-janus/

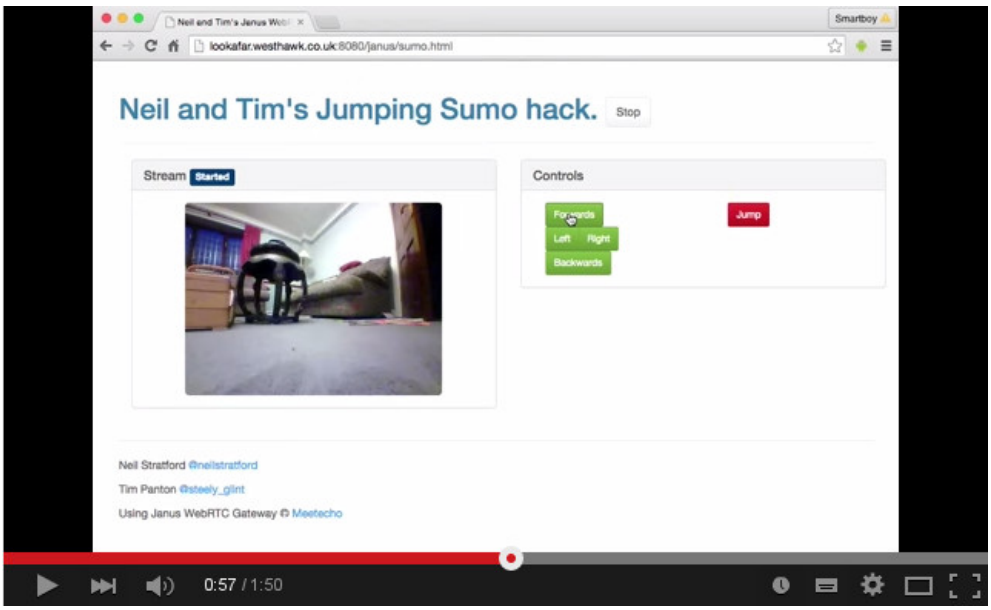


Figure 6.4: The Jumping Janus: live demo

the drone itself, and to control the drone using virtual commands to make it move around a room or even jump. The result was so popular and successful that the authors did indeed win the contest, something that, besides making us proud for the part Janus had taken in it, furtherly convinced us of the validity of our efforts.

Such an application, while not really useful per se if not to showcase the possibilities of WebRTC-based drone scenarios, is a perfect example of the incredible flexibility made available by Janus, a flexibility we aim to take advantage of in even more challenging research activities in the near future.

Bibliography

[1] D. Wu, S. Member, Y. T. Hou, W. Zhu, Y. qin Zhang, J. M. Peha, and S. Member, "Streaming video over the internet: approaches and directions," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 11, pp. 282–300, 2001.

[2] H. Parmar and H. Thornburgh, "Adobe's Real Time Messaging Protocol," tech. rep., December 2012.

[3] Microsoft, "Smooth Streaming Protocol," tech. rep., May 2014.

[4] I. Sodagar, "The mpeg-dash standard for multimedia streaming over the internet," *IEEE Multimedia*, 2001.

[5] "Web Real-Time Communications Working Group Charter (W3C)." http://www.w3.org/2011/04/webrtc-charter.html.

[6] "Real-Time Communication in WEB-browsers (IETF)." http://tools.ietf.org/wg/rtcweb/charters.

[7] H. Schulzrinne, S. Casner, and R. F. et al., "RTP: A Transport Protocol for Real-Time Applications," RFC 3550, RFC Editor, July 2003.

[8] H. Schulzrinne, A. Rao, and R. Lanphier, "Real Time Streaming Protocol (RTSP)," RFC 2326, RFC Editor, April 1998.

[9] V. Novotny and D. Komosny, "Large-scale rtcp feedback optimization," *Journal of Networks,Vol. 3, No. 3*, March 2008.

[10] M. Baugher, D. McGrew, and M. N. et al., "The Secure Real-time Transport Protocol (SRTP)," RFC 3711, RFC Editor, March 2004.

[11] C. Holmberg, S. Hakansson, and G. Eriksson, "Web Real-Time Communication Use Cases and Requirements," RFC 7478, RFC Editor, March 2015.

[12] J. Rosenberg, H. Schulzrinne, and G. C. et al., "SIP: Session Initiation Protocol," RFC 3261, RFC Editor, June 2002.

[13] J. Rosenberg and H. Schulzrinne, "An Offer/Answer Model with the Session Description Protocol (SDP)," RFC 3264, RFC Editor, June 2002.

[14] J. Rosenberg and C. Jennings, "RTCWeb Offer/Answer Protocol (ROAP)," Internet-Draft draft-jennings-rtcweb-signalling-00, IETF Secretariat, Oct. 2011.

[15] J. Uberti and C. Jennings, "Javascript Session Establishment Protocol," Internet-Draft draft-ietf-rtcweb-jsep-07, IETF Secretariat, Feb. 2014.

[16] P. Srisuresh and K. Evegang, "Traditional IP Network Address Translator (Traditional NAT)," RFC 3002, RFC Editor, January 2001.

[17] J. Rosenberg, R. Mahy, and P. M. et al, "Session Traversal Utilities for NAT (STUN)," RFC 5389, RFC Editor, October 2008.

[18] J. Rosenberg, R. Mahy, and P. Matthews, "Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN)," RFC 5766, RFC Editor, April 2010.

[19] J. Rosenberg, "Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols," RFC 5245, RFC Editor, April 2010.

[20] A. Amirante, T. Castaldi, L. Miniero, and S. P. Romano, "NTRULO: A Tunneling Architecture for Multimedia Conferencing over IP," in *Lecture Notes in Computer Science - 10th International Conference on Next Generation Teletraffic and Wired/Wireless Advanced Networking (NEW2AN 2010)*, Springer-Verlag, August 2010.

[21] X. Chen, S. G. Murillo, and L. M. et al., "WebSocket Protocol as a Transport for Traversal Using Relays around NAT (TURN)," Internet-Draft draft-chenxin-behave-turn-websocket-01, IETF Secretariat, Sept. 2013.

[22] 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.

[23] J. Ott, S. Wenger, and N. S. et al., "Extended RTP Profile for Real-time Transport Control Protocol (RTCP)-Based Feedback (RTP/AVPF)," RFC 4585, RFC Editor, July 2006.

[24] S. Wenger, U. Changra, and M. W. et al., "Codec Control Messages in the RTP Audio-Visual Profile with Feedback (AVPF)," RFC 5104, RFC Editor, February 2008.

[25] H. Alvestrand, "RTCP message for Receiver Estimated Maximum Bitrate," Internet-Draft draft-alvestrand-rmcat-remb-03, IETF Secretariat, Oct. 2013.

[26] 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-05, IETF Secretariat, Mar. 2015.

[27] F. Andreasen, M. Baugher, and D. Wing, "Session Description Protocol (SDP) Security Descriptions for Media Streams," RFC 4568, RFC Editor, July 2006.

[28] 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.

[29] J. Valin, K. Vos, and T. Terriberry, "Definition of the Opus Audio Codec," RFC 6716, RFC Editor, September 2012.

[30] J. Bankoski, J. Koleszar, and L. Q. et al., "VP8 Data Format and Decoding Guide," RFC 6386, RFC Editor, November 2011.

[31] 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.

[32] C. Perkins and M. Westerlund, "Multiplexing RTP Data and Control Packets on a Single Port," RFC 5761, RFC Editor, April 2010.

[33] 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.

[34] R. Jesup, S. Loreto, and M. Tuexen, "WebRTC Data Channels," Internet-Draft draft-ietf-rtcweb-data-channel-11, IETF Secretariat, June 2014.

[35] R. Stewart, "Stream Control Transmission Protocol," RFC 4960, RFC Editor, September 2007.

[36] A. Amirante, T. Castaldi, L. Miniero, R. Presta, and S. P. Romano, "Standard multimedia conferencing in the wild: the meetecho architecture," *Multimedia Tools and Applications, Springer*, September 2011.

[37] A. Amirante, A. Buono, T. Castaldi, L. Miniero, and S. P. Romano, "Centralized Conferencing in the IP Multimedia Subsystem: from theory to practice," *Journal of Communications Software and Systems (JCOMSS)*, vol. 4, pp. 80–90, March 2008.

[38] A. Amirante, T. Castaldi, L. Miniero, and S. Romano, "On the seamless interaction between webrtc browsers and sip-based conferencing systems," *Communications Magazine, IEEE*, vol. 51, no. 4, pp. 42–47, 2013.

[39] E. Ivov, H. Kaplan, and D. Wing, "Latching: Hosted NAT Traversal (HNT) for Media in Real-Time Communication," RFC 7362, RFC Editor, September 2014.

[40] T. Melanchuk, "An Architectural Framework for Media Server Control," RFC 5567, RFC Editor, June 2009.

[41] C. Boulton, T. Melanchuk, and S. McGlashan, "Media Control Channel Framework," RFC 6230, RFC Editor, May 2011.

[42] C. Boulton, T. Melanchuk, and S. McGlashan, "An Interactive Voice Response (IVR) Control Package for the Media Control Channel Framework," RFC 6231, RFC Editor, May 2011.

[43] C. Boulton, T. Melanchuk, and S. McGlashan, "A Mixer Control Package for the Media Control Channel Framework," RFC 6505, RFC Editor, March 2012.

[44] A. Amirante, T. Castaldi, L. Miniero, and S. Romano, "Media Control Channel Framework (CFW) Call Flow Examples," RFC 7058, RFC Editor, November 2013.

[45] 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 *Lecture Notes in Computer Science - 8th International Conference on Next Generation Teletraffic and Wired/Wireless Advanced Networking (NEW2AN 2008)*, vol. 5174, pp. 199–211, Springer-Verlag, 2008.

[46] A. Amirante, T. Castaldi, L. Miniero, and S. P. Romano, *Protocol interactions among User Agents, Application Servers and Media Servers:*

standardization efforts and open issues, vol. Intelligent Multimedia Technologies for Networking Applications: Techniques and Tools. IGI Global, 2012.

[47] C. Boulton, L. Miniero, and G. Munson, "Media Resource Brokering," RFC 6917, RFC Editor, April 2013.

[48] L. Miniero, "Improving the scalability of real-time multimedia applications using brokering of media resources." InfQ 2013, June 13-14, 2013, Sorrento, Italy, June 2013.

[49] 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.

[50] "Selenium Framework." http://docs.seleniumhq.org/.

[51] A. Amirante, T. Castaldi, L. Miniero, and S. P. Romano, "Performance analysis of the Janus WebRTC gateway," in *All-Web real-time Systems (AWeS 2015), Bordeaux, France*, April 2015.

[52] J. Rey, D. Leon, and A. M. et al., "RTP Retransmission Payload Format," RFC 4588, RFC Editor, July 2006.