How Far Are We from WebRTC-1.0? An Update on Standards and a Look at What's Next

Salvatore Loreto and Simon Pietro Romano

Real-time communication between browsers has represented an unprecedented standardization effort involving both the IETF and the W3C. These activities have involved both the real-time protocol suite and the application-level JavaScript APIs to be offered to developers in order to allow them to easily implement interoperable real-time multimedia applications in the web. The authors shed light on the current status of standardization, with special focus on the upcoming final release of the so-called WebRTC-1.0 standard ecosystem.

Abstract

Real-time communication between browsers has represented an unprecedented standardization effort involving both the IETF and the W3C. These activities have involved both the real-time protocol suite and the application-level JavaScript APIs to be offered to developers in order to allow them to easily implement interoperable real-time multimedia applications in the web. This article sheds light on the current status of standardization, with special focus on the upcoming final release of the so-called WebRTC-1.0 standard ecosystem. It takes stock of the situation with respect to hot topics such as codecs, session description and stream multiplexing. It also briefly discusses how standard bodies are dealing with seamless integration of the initially competing effort known as "Object Real Time Communications."

Background, Rationale and Motivation

Real-time communication in the web has been the subject of a challenging standardization process for the last five years or so. Back in 2011, the Internet Engineering Task Force (IETF) chartered the "Real-Time Communication in WEB-browsers" (RTCWEB) Working Group, with the aim of defining an architecture and a complete suite of protocols for the support of real-time multimedia communications directly between browsers. The RTCWEB WG has since worked on key aspects like the overall communication infrastructure, the protocols and API (application programming interface) requirements, the security model, the media formats (and related media codecs), as well as advanced functionality like congestion/flow control and interworking with legacy VoIP equipment.

In parallel, the World Wide Web Consortium (W3C) has conducted an activity defining a set of APIs exposing functions like exploration and access to device capabilities, capture of media from local devices, encoding/processing of "media streams", establishment of peer-to-peer connections between browsers (and web-enabled devices in general), decoding/processing of incoming media streams and delivery of such streams to the end-user in an HTML5-compliant fashion.

To date, the two mentioned working groups have achieved a major milestone in the field of real-time multimedia communications: the so-called *WebRTC*-1.0 standards suite. The idea behind WebRTC-1.0 is to allow all of the involved stakeholders (browser vendors, telecommunication providers, application providers, web developers, and so on) to converge on a well-defined set of protocols and APIs to be leveraged in order to allow widespread deployment on the market of interoperable products offering end-users a media-rich, web-enabled, real-time experience. To achieve this goal, the standardization process has necessarily had to face a number of obstacles while trying to strike a balance among diverging interests and/or viewpoints.

This article will briefly survey the current state of the art with respect to WebRTC-1.0 completion and introduce the envisioned work program for the second generation of the standard. In doing so, it will touch upon debated topics and illustrate how the community has successfully coped with them.

State of the Art

Related Work

In our previous work on the subject [1] we discussed the evolution of real-time communication in the web, by highlighting the main steps that brought the IETF and the W3C to the launch of the joint standardization initiatives known, respectively, as *RTCWEB* and *WebRTC*. At the time of that writing the standards process had already reached a good level of maturity, even though a number of issues were still open (e.g., congestion control, audio and video codec selection, enhanced use of data channels).

In a subsequent work [2], Jennings *et al.*, focused on security challenges and transport issues, while presenting the solutions and mechanisms proposed within both the IETF and the W3C. They also identified congestion control as an open research question.

Other authors have focused on specific aspects of WebRTC, with special reference to security. Barnes and Thomson [3] provide a thorough description of the security threats associated with peer-to-peer web-based communications, and identify the WebRTC security architecture as a good candidate for the implementation of appli-

Digital Object Identifier:
10.1109/MCOM.2017.1600283

Salvatore Loreto is with Ericsson; Simon Pietro Romano is with University of Napoli Federico II.

cations that can be secured from tampering by intermediaries. Similarly, Johnston *et al.* [4] discuss issues specific to WebRTC enterprise adoption by focusing on security, compliance, and interoperation.

The objective of this article is to provide an up-to-date view of the current status of standardization, while also identifying challenges that the standardization community will have to tackle once the first release of the WebRTC standards suite has been finalized. The WebRTC standard has in fact had to confront itself with both inner disputes and alternative views. Among the inner disputes we can cite the so-called "codec battle" between the supporters of two prominent candidates for the *Mandatory To Implement* (MTI) WebRTC video codec, namely H.264 and VP8. After an unsuccessful consensus call at IETF 88 (held in Vancouver in November 2013), such a battle ended up with the compromise decision of indicating both codecs as MTI for WebRTC. A further significant issue concerns WebRTC support within browsers. With respect to this particular topic, the current situation is that several browser vendors (Chrome, Firefox, Opera, Edge and Bowser) with differing completion scores,[1] are WebRTC-enabled. An important exception is currently represented by Safari. Apple, in fact, while closely following the standardization activities, has played no active role until now and their browser has no WebRTC capabilities.

Coming to the alternative views, since the beginning of 2014, a brand new initiative has seen the light in the W3C, the ORTC (Object Real-time Communications) Community Group. ORTC has indeed taken over from a previous initiative launched in mid 2013 and called ORCA (OBJECT-RTC API). Both ORCA and ORTC have initially been identified as alternatives to WebRTC. ORCA's explicit goal was to provide an alternative to the existing WebRTC API, aimed at allowing finer grained control to web developers willing to leverage real-time functionality within browsers. The same holds true for its successor ORTC, whose mission is to "define object-centric APIs to enable real-time communications in Web browsers, mobile endpoints, and servers".

Lately, the standardization community has agreed to converge to an agreed-upon solution for the first version of the standard by allowing the ORTC community to contribute to its finalization. At the same time, a common decision has been taken to adopt key concepts proposed with ORTC's low-level object API in the 'Next Version' of the standard, which nonetheless has backward compatibility with the 1.0 release among its foundational requirements.

This is exactly where the community stands now. A step away from completing WebRTC-1.0, with all minds already looking at the emerging initiative informally known as *WebRTC Next Version* (WebRTC-NV).

The WebRTC Architecture

WebRTC extends the classic web architecture semantics by introducing a peer-to-peer communication paradigm between browsers. The WebRTC architectural model draws its inspiration from the so-called SIP (Session Initiation Protocol) [5] Trapezoid. The most common WebRTC scenario is indeed one where both browsers are running the same web application, downloaded from the same application server. In this case, the Trapezoid becomes a Triangle, as shown in Fig. 1. Signaling messages are used to set up and terminate communications. They are transported by the HTTP or WebSocket protocol via the web server, which can modify, translate, or manage them as needed. It is worth noting that the signaling between browser and server is not standardized in WebRTC, as it is considered to be part of the application. As to the data path, the *PeerConnection* abstraction allows media to flow directly between browsers without any intervening servers.

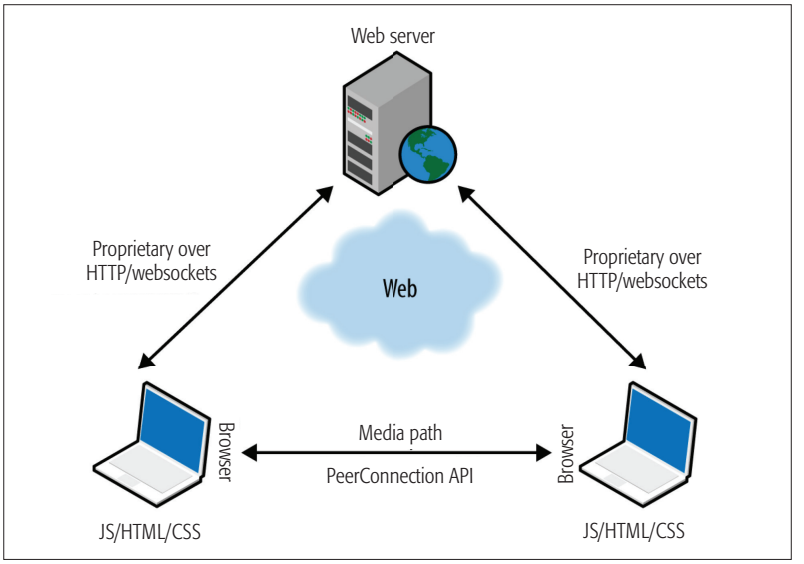

Figure 1. The WebRTC architecture.

A WebRTC web application is typically written as a mix of HTML and JavaScript. It interacts with web browsers through the standardized WebRTC API, as well as other standard APIs, allowing it to properly exploit and control the real-time browser function, both proactively (e.g., to query browser capabilities) and reactively (e.g., to receive browser-generated notifications). The WebRTC API must hence provide a wide set of functions, like connection management (in a peer-to-peer fashion), encoding/decoding capabilities negotiation, selection and control, media control, firewall and NAT element traversal.

Session description represents an important piece of information that needs to be exchanged. It specifies the transport information, as well as the media type, format, and all associated media configuration parameters needed to establish the media path. The IETF is now standardizing the JavaScript Session Establishment Protocol (JSEP) [6]. JSEP provides the interface needed by an application to deal with the negotiated local and remote session descriptions (with the negotiation carried out through whatever signaling mechanism might be desired), together with a standardized way of interacting with the ICE (interactive connectivity establishment) [7] state machine. The JSEP approach delegates entirely to the application the responsibility for driving the signaling state machine: the application must call the right APIs at the right times, and convert the session

[1] See http://iswebrtcreadyyet.com/ for an interesting summarizing table.

From the programmer's perspective, an important update to the WebRTC specification has been the introduction of Promises. Promises currently represent an advanced way for allowing asynchronous communication when using JavaScript. In a nutshell, they are similar to event listeners, but with a couple of fundamental improvements.

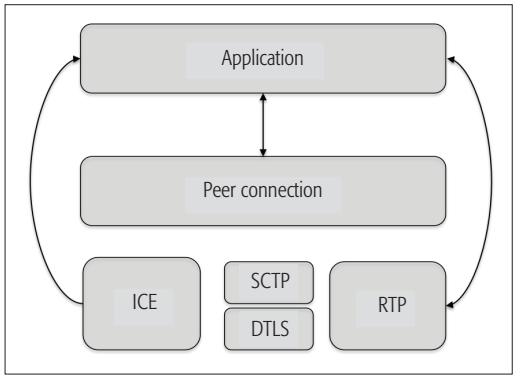

Figure 2. WebRTC: coarse-grained logical decomposition.

descriptions and related ICE information into the defined messages of its chosen signaling protocol.

It is worth mentioning that JSEP offers the possibility of manipulating session descriptions contained inside SDP (Session Description Protocol) messages. This happens within some limits (since browsers try to limit SDP "munging" to avoid disrupting communications) and at the developer's risk.

The W3C WebRTC-1.0 API allows a JavaScript application to take advantage of the novel browser's real-time capabilities. The real-time browser function implemented in the browser core provides the functionality needed to establish the necessary audio, video, and data channels. All media and data streams are encrypted using DTLS [8] (Datagram Transport Layer Security). DTLS is actually used for key derivation, while SRTP [9] (Secure Real-time Transport Protocol) is used on the wire. So, the audio and video packets on the wire are sent using SRTP. Data channel packets are handled by using SCTP [10] encapsulated in DTLS.

Figure 2 sketches, at a very high level, the current structure of the object oriented WebRTC framework. As anticipated, low-level components are for the most part indirectly controlled through the *PeerConnection* structure. Only a restricted form of direct control is allowed for ICE-related and RTP-related functionality. As shown in the figure, RTP allows for some form of control over the behavior of the protocol itself (e.g., for what concerns bandwidth capping). Coming to ICE, with the advent of WebRTC we have assisted to a renewed interest in such a protocol (as well as in its companion protocols STUN and TURN), as witnessed by the creation of the *tram* (TURN Revised and Modernized) working group within the IETF.

Identity Management in WebRTC

The WebRTC API also offers methods to enable verifying user identities. The solution decouples identity provision from communication providers via a third-party *identity provider* (IdP) (supporting a protocol such as OpenID or BrowserID) that can be used to demonstrate their identity to other parties. With this approach, trust between users is built by relying on an external entity [11].

This separation between identity provision and signaling is particularly important in federated scenarios (calls from one domain to another) and when calling via untrusted sites such as when two users who have a relationship via a given social network want to call each other via another, untrusted, site. The solution decouples the browser from any particular identity provider. The browser only needs to know how to load the IdP's JavaScript. Thus, a single browser can support any number of identity protocols. WebRTC offers and answers can in this way be authenticated by using the IdP. The entity sending an offer or answer acts as the *Authenticating Party* (AP) and obtains an identity assertion from the IdP, which it then attaches to the session description. The consumer of the session description acts as the *relying party* (RP) and verifies the assertion.

Toward a First Release of the Standard: WebRTC-1.0

In this section we will briefly discuss some relevant features that are going to be part of the WebRTC-1.0 specification. A non-exhaustive list of such features is reported in Table 1. For each item in the table, we provide a short description, as well as our estimation of its maturity level in terms of inclusion into the standard specifications. The following sections delve into some of the details associated with each of the reported features.

From Legacy JavaScript to ECMAScript Promises

From the programmer's perspective, an important update to the WebRTC specification has been the introduction of *Promises*. Promises currently represent an advanced way for allowing asynchronous communication when using JavaScript. In a nutshell, they are similar to event listeners, but with a couple of fundamental improvements. First, Promises can succeed or fail only once and they can never switch between success and failure states. Second, Promises can be associated with success and failure callbacks that are triggered independently from the exact time when the success/failure event has been raised. This allows applications to react to the outcome of an event rather than focusing on the exact time such an event took place.

All WebRTC-related APIs have lately been modified in order to move from the callback-based approach to the Promise-based approach, with the exception of the well known `navigator.getUserMedia()` method, which has been left unchanged for backward compatibility reasons.

From Streams to Tracks

The W3C *MediaStream* API specified by the "Media Capture and Streams" WG (and used within the WebRTC WG as one of its foundational blocks) has recently been modified in order to increase the level of granularity associated with the various media managed from within the browser. Namely, it has moved from streams to tracks. Streams have initially been interpreted as the most atomic data structure being transmitted over a *PeerConnection*. With the evolution of the specification, they have now been further described as collections of tracks. In summary, the current *MediaStream* objects represent synchronized streams of media that can be recorded or rendered in a media element. For example, a stream taken from camera and microphone

Feature	Function	Expected timeline
Promises	Use of ECMAScript promises in the API. No more callback-based methods exist	Certainly part of the WebRTC-1.0 spec
MediaStreamTrack objects	Allow developers to differentiate stream processing on a per-track basis	Certainly part of the WebRTC-1.0 spec
SDP bundling	Transmission of multiple media flows using a single 5-tuple	Details still under discussion, but most probably part of WebRTC-1.0
Codec priority reordering	Allow codecs to be reordered at the API level	Certainly part of the WebRTC-1.0 spec
RTCP multiplexing	Send both RTP and related RTCP data over a single port	Certainly part of the WebRTC-1.0 spec
Simulcasting	Send the same video stream at multiple resolutions and/or rates	Details still under discussion, but most probably part of WebRTC-1.0
Forward error correction (FEC)	Add redundancy to the encoded information and allow the receiver to compensate for partial data losses	Preliminary discussions ongoing (requirements draft under evaluation in RTCWEB)
Early media	Send media to the remote party before emitting an answer to an already received SDP offer	Certainly part of the WebRTC-1.0 spec
Screen sharing	Capture a user's screen and send it to a remote side in the form of a video stream	Details still under discussion, but certainly part of WebRTC-1.0

Table 1. WebRTC-1.0 Features and timeline.

Session negotiation is an important part of WebRTC. This calls into play the well-known Session Description Protocol (SDP). SDP provides multimedia applications with a standard means to describe a session, in terms of connectivity (i.e., IP addresses and ports), codecs, media attributes, and so on.

inputs has synchronized video and audio tracks representing synchronized streams of media. Each track is represented by a *MediaStreamTrack*. The main reason behind this increased granularity resides in the consideration that developers want to be capable of differentiating stream processing on a per-track basis, for example, to specify which codecs must be adopted, as well as the specific parameters used to configure such codecs. Some key transport properties can now also be set on a per track basis. To name just a few examples, we cite forward error correction (FEC), retransmission policy and bandwidth capping. All of the mentioned configuration actions are actually carried out by leveraging the brand new *RTCRtpSender* and *RTCRtpReceiver* interfaces, which allow applications to control how a given MediaStreamTrack is encoded/decoded and transmitted/received to/from a remote peer.

SDP "Bundling"

Session negotiation is an important part of WebRTC. This calls into play the well known *Session Description Protocol* (SDP). SDP provides multimedia applications with a standard means to describe a session, in terms of connectivity (i.e., IP addresses and ports), codecs, media attributes, and so on. As part of the SDP specification, it is possible to leverage a quite recent feature called *BUNDLE* [12], which refers to the transmission of multiple media flows (i.e., a 'bundle') using a single 5-tuple, that is to say, a single combination of a sending "IP address/port" pair, a receiving "IP address/port" pair, and a specific transport protocol (e.g., RTP). Within the context of WebRTC, the use of this technique has since the outset been encouraged, since it makes it possible to both save port numbers and reduce the number of ICE (Interactive Connectivity Establishment) protocol candidates. The latter point is particularly important since it dramatically reduces session setup time.

Bundling can be properly configured, at the API level, through an ad hoc defined parameter called *RTCConfiguration*, which contains, among other things, a property called `bundlePolicy`. Such a property can assume one of the following values: "Max-bundle", "Max-compat", or "Balanced."

The basic idea is that a WebRTC device will always try to use the bundle mechanism when negotiating a session with another peer. If the remote peer does not support bundle, then the aforementioned policy property comes into play. More precisely, "max-bundle" will instruct the WebRTC device to select a single media flow (among those that had to be bundled) and negotiate such a flow via SDP. If "max-compat" is selected, it will instead negotiate all of the flows separately, just as if bundle had never been introduced. This second approach is indeed the optimal one in case of backward compatibility with legacy (i.e., not aware of the bundle feature) devices. Finally, "balanced" refers to the intermediate approach of choosing two tracks (one audio track and one video track) to be negotiated separately via SDP.

Somehow related to the bundling mechanism is a further feature called "streams multiplexing," which is the possibility of adding multiple streams of the same type (either audio or video) to a single PeerConnection. BUNDLE indeed describes how to transmit/receive audio and video together, but does not explicitly deal with multiple instances of the same media type. This has been the subject of long discussions within RTCWEB, often referred

Simulcast is a relatively new function that draws inspiration from stream multiplexing, that is, a technique whereby a media source simultaneously sends multiple different encoded streams towards a specific destination, for example, the same video source encoded with different video encoder types or image resolutions.

to as the "Plan B vs. Unified Plan" debate, which eventually saw Unified Plan prevailing and being merged in the JSEP specification. The so called MSID (Media Stream Identification) draft [13] in the MMUSIC WG is targeted at allowing this to work, by specifying an SDP grouping mechanism for RTP media streams that can be used to indicate relations between media streams.

Playing with Codec Priority at the API Level

SDP makes it possible, among other things, to specify, for each media stream, the list of supported codecs. Upon session negotiation, the two peers agree on a set of codecs that is computed as the largest subset of common codecs signaled by the two parties. Such a subset is ordered as a list, and the first element is selected as the default codec to be used during the session. All other elements in the subset have to be supported by both parties (since they were advertised in the respective SDPs upon session setup time). Hence, the SDP specification allows for a peer to change codec during the session (provided that the new one belongs in the agreed-upon list of supported options) with no need to renegotiate the session itself.

Given this assumption, the WebRTC specification now makes it possible to programmatically select the desired codec for a PeerConnection with no need to edit the original SDP. More precisely, the API currently makes it possible to:

- Gain access to the bundle of parameters associated with an RTP sender (through the `RTCRtpSender.getParameters()` method).
- Select, within such a structure, the "codecs" property, which is basically an array of supported codecs related to that sender.
- Reorder (or even remove) information contained in the codecs list.
- Commit changes to the RTP sender object (through the `setParameters()` method).

RTCP Multiplexing

The standard way of streaming real-time media across the Internet envisages the use of RTP (Real-time Transport Protocol) for application-level framing of media samples, in conjunction with the companion RTCP protocol used to carry both feedback and minimal session control information back and forth between the two peers. Usually, RTP and RTCP are associated with different ports (e.g., if $2n$ is an even port used for RTP, then $2n + 1$ will be an odd port associated with RTCP control information). With RTCP multiplexing (also known as RTCP MUX), we refer to a way of sending both RTP and related RTCP data over a single port. The idea of leveraging such a function is, once again, to both save allocated port numbers and reduce ICE setup time.

After a good deal of discussions on whether or not to specify RTCP MUX support as optional for WebRTC, there currently seems to be consensus around making it mandatory at least in those cases in which the peers are also using SDP bundle. At recent IETF meetings, a further step was done along the same lines and two major WebRTC browser vendors (namely, Google and Firefox) have clearly stated their will to allow WebRTC endpoints to simply reject legacy (i.e., non multiplexed) RTCP sessions. This resolution, while simplifying things a lot for WebRTC-capable devices, clearly calls for the introduction of a proxying function (provided by some sort of WebRTC gateway intervening along the data path) if the need arises to interact with any legacy application still relying on two different ports for RTP and RTCP.

Simulcasting

Simulcast is a relatively new function that draws inspiration from stream multiplexing, that is, a technique whereby a media source simultaneously sends multiple different encoded streams toward a specific destination, for example, the same video source encoded with different video encoder types or image resolutions. It can be somehow associated with Scalable Video Coding (SVC), namely the mechanisms by which a single encoded video stream can be organized in layers and each participant is allowed to receive (and decode) only the layers that they are able to process. The WebRTC community has long since identified a number of use cases for simulcast. One interesting example is represented by conferencing scenarios involving the presence of a so-called *selective forwarding unit* (SFU). In the mentioned scenario, the clients send to the SFU (which is acting as a conference focus) multiple video streams, each associated with exactly the same scene, but at different resolutions. The SFU can hence properly select the specific incoming stream that has to be forwarded to the other participants. As an example, the SFU might forward a high resolution version of the stream only when the client in question is playing an active role in the conference (e.g., they are currently holding the floor), while relying on the lower resolution version while they are not actively participating in the discussion. Other, more complex, forwarding choices can obviously be applied once the general mechanism described above is available. Just to cite one, the SFU might let the choice depend on considerations associated with optimizing overall bandwidth consumption, while at the same time offering a good-enough service to the end-users in terms of quality of experience (QoE).

Coming to the technical details, until recently, there has been a lack of uniformity in the way simulcasting has been deployed in the wild. The basic mechanism leveraged by all implementations is represented by the insertion of multiple *m* (i.e., media) lines of the same media type (e.g., audio, video, and so on) inside the SDP body. What was lacking in this case was a means to signal to the other party that those m-lines were indeed all associated with a single source. A recent proposal from Google seems to have filled exactly this gap and has gained consensus within the IETF community. In a nutshell, the idea is to add a new identifier in SDP, namely a source stream identifier, that can be leveraged to differentiate sets of media attribute lines.

As a result of this approach, the W3C has allowed some form of manipulation of simulcast streams at the API level. More precisely, within the context of the newly defined *RTCRtpTransceiver* interface (which is basically a combination of an *RtpSender* and an *RtpReceiver* associated with the same SDP media identifier) it is possible

to refer to a property called "rid," which is nothing but a copy of the above mentioned source stream identifier. This structure, combined with a new feature called "scaleDownResolutionBy" indicating a scaling down factor relative to the maximum resolution available for the stream, allows the developer to explicitly choose the desired quality of a signaled simulcast stream.

Forward Error Correction

One interesting topic of discussion at recent IETF meetings has been the introduction (and configuration) of *forward error correction* (FEC) [14] capabilities inside WebRTC endpoints. Opus, which is one of the "MTI" (mandatory to implement) codecs for audio, does provide in-band support for it.

FEC is a generic mechanism for the protection of media streams against packet corruption due, for example, to the presence of one or more lossy links along the end-to-end communication path. It adds some level of redundancy inside the encoded information, so to allow the receiving peer to properly compensate for partial data loss with no need for retransmissions.

As it always happens when redundant encoding is introduced, the advanced reconstruction capabilities at the receiving side are paid in terms of increased network overhead. Hence, the challenge in these cases is to try to strike an optimal balance between robustness to packet corruptions and increased bandwidth consumption. This holds particularly true in all those cases in which the network does not provide any form of congestion control. In such cases, indeed, the issue is *congestion* rather than *lossy communication*, and the use of FEC can only make things worse as it contributes to increasing congestion due to the overhead it unavoidably introduces.

Within the standardization community, work is currently in progress in order to allow WebRTC implementations to fine-tune the configuration of FEC parameters (as allowed by the RTP specification), to enforce a fair behavior on the side of the applications. At recent meetings there has also been some preliminary discussion on whether or not to allow such tuning knobs to surface at the JavaScript API level.

With reference to congestion control, it is instead worth mentioning the ongoing work within both the AVTCORE and RMCAT Working Groups within the IETF, with special regard to the so called Circuit Breakers [15] document, which is soon to become an RFC.

Allowing Early Media

Early media is a well-known term in VoIP networks, referring to the capability of sending some media to the other party before emitting an answer to an already received SDP offer. While this might seem awkward, it is a very useful mechanism that real-time applications are used to leverage in order to provide an enriched end-user experience through, for example, playing music while the user is waiting for a call to be connected.

WebRTC has since long looked at early media as a desired functionality, both to seamlessly interact with legacy VoIP applications that already rely on it and to bring its benefits to the WebRTC ecosystem itself. Recently, this function has been standardized. More precisely, it has been specified that an end-point that receives media before getting the answer to its own offer can accept such media provided that:

- It is consistent with the emitted SDP offer (in terms of codecs and other media attributes).
- The end-point in question (i.e., the emitter of the SDP offer) has already created an instance of the *RTCRtpReceiver* object that is to be associated, upon successful completion of the session setup procedures, with the incoming media stream.

The mentioned requirements have been provided through minor modifications to the WebRTC-1.0 specification. Fundamentally, a change was made as to when tracks are created for the offerer. This can now happen either as a result of a call to the `setLocalDescription` method, or as soon as media packets are received. The mentioned modifications ensure that these objects can be created and connected to media elements for play-out when needed. Without digging in too much detail, we just mention as a side note that, in order to prevent potential security breaches, early media cannot happen 'earlier' than the remote DTLS (Datagram Transport Layer Security) fingerprint has been received.

Screen Sharing

Within the context of WebRTC, screen sharing refers to the capability of capturing a user's screen (all or in part) and sending it to a remote side (across a PeerConnection) in the form of a video stream. Such a function leverages an ad hoc defined extension to the *Media Capture API*, which defines a new method called `getDisplayMedia`. Such a method allows for the acquisition of different types of captures, in terms of both the "portion" of the screen one is interested in sharing and the type of display "surface." With respect to this last term, a distinction is made between a *logical* surface and a *visible* one. The former refers to an entire application window, independently from the fact that part of such a window might be covered by another application's window; the latter is instead associated with the part of the window that is visible on the user's side, that is, that is not covered by any other window that is not being shared. As to the portion of the screen that is going to be shared, the following choices are available:

- *Monitor:* one or more physical displays (connected to a user's computer).
- *Window:* a single application window.
- Application: all of the windows associated with a specific application.
- *Browser:* a single browser window (or *Tab*).

Inherently, screen sharing poses a number of security and privacy concerns. The most intuitive risk is related to the fact that users might inadvertently share content that they did not wish to share. A less obvious risk is also associated with display capture. Namely, this new function might weaken the cross site request forgery protections that should be guaranteed by the browser sandbox. As an example, sharing of a window containing a canvas might circumvent standard controls on such an object that do not allow sampling or even conversion to any accessible form if it is not "origin-clean."

Within the standardization community, work is currently in progress in order to allow WebRTC implementations to fine-tune the configuration of FEC parameters (as allowed by the RTP specification), so to enforce a fair behavior on the side of the applications. There has also been preliminary discussion on whether or not to allow such tuning knobs to surface at the JavaScript API level.

A further reason why ORTC supporters proposed a lower-level API concerns the implementation of advanced functionality like simulcasting and Scalable Video Coding (SVC), which both benefit from the possibility of gaining direct access to the basic building blocks of the media pipeline.

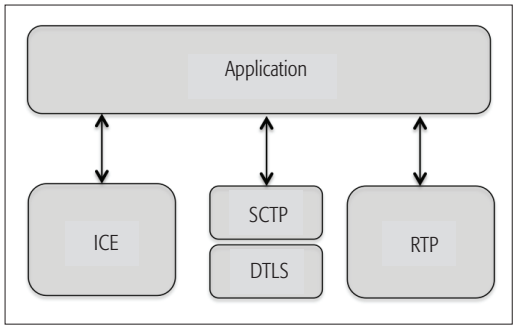

Figure 3. The ORTC architecture.

This and other related issues are currently under discussion within the RTCWEB working group, which has taken at the outset the responsibility of defining the overall security architecture for web real-time communications. With respect to the aforementioned cross-origin protection capabilities, it is strongly advised that users are asked to exhibit elevated permissions before being allowed to access any available display surface.

Bringing ORTC Concepts into WebRTC

Seminal work behind *Object Real-Time Communications* (ORTC) stemmed from the consideration that the SDP-based offer/answer paradigm embraced by the WebRTC API did not fit well the emerging real-time communication models (with special reference to peer-to-peer systems). The core of ORTC is represented by a JavaScript API designed within the ORTC W3C Community Group. Such an API aims at offering finer-grained control over how a real-time web application is implemented, by exposing to the surface most of the objects that the standard WebRTC API typically controls as a single pipelined unit of elaboration through a higher-level configuration interface. Since the outset, the idea has been to allow the coexistence between the SDP-based Offer/Answer approach proposed by WebRTC and the low-level ORTC API. This is achieved thanks to the superposition, on top of the ORTC API, of a WebRTC-compliant *shim* library. With this approach, programmers can choose between ORTC-style raw control of the real-time communications engine on one side and WebRTC-style SDP-based negotiation on the other.

A rough comparison between Figs. 2 and 3 allows us to highlight the major difference between the WebRTC and the ORTC approach. Namely, the two models work, at the lowest layer, with the same set of objects. WebRTC-1.0 relies on the *PeerConnection* abstraction as a glueing component that somehow orchestrates the overall behavior of a peer. ORTC, on the other hand, allows the programmer to gain full direct control over the set of available objects and optionally enables the use of the PeerConnection as an API facility that is provided through the above mentioned shim adapter library.

Based on the considerations above, it is fair to claim that ORTC is not to be considered as a competitor to WebRTC. Full compatibility with the WebRTC-1.0 API is guaranteed by the development of the aforementioned SDP-based JavaScript shim on top of ORTC. Such a library takes on the responsibility of ensuring that SDP parsing and negotiation features are identical and work on top of the ORTC primitives. Compatibility is to be thoroughly checked via unit testing procedures. This is expected to foster interoperability among heterogeneous implementations. A further reason why ORTC supporters proposed a lower-level API concerns the implementation of advanced functionality like *simulcasting* and *scalable video coding* (SVC), which both benefit from the possibility of gaining direct access to the basic building blocks of the media pipeline.

It is important to stress the consideration that, since its foundation as a W3C community group, ORTC has never been really conceived as a competitor to WebRTC. As already anticipated, it has rather been seen as an alternative, yet compliant, approach that can be leveraged by those developers who are targeting scenarios different than the "standard" Offer/Answer based ones. The efforts that have been devoted to the design of the shim library allowing for the seamless operation of a WebRTC application on top of the pipeline-based ORTC framework can indeed be seen as a real added value to the overall WebRTC ecosystem.

The above statement is so true that during a recent WebRTC charter renewal process, key representatives of the ORTC community group have been formally invited to join the WebRTC effort. More precisely, one of the founders of the ORTC initiative has joined the WebRTC chairs, while another ORTC representative has become a member of the WebRTC-1.0 editing team. It has also been decided that all future standardization work in WebRTC will take place within the WebRTC Working Group, while the ORTC community group will fade away and its contributors will join the WebRTC effort. Finally, once done with the WebRTC-1.0 milestone, all energies will be devoted to a brand new initiative called WebRTC-NV, as discussed in the next section.

Discussion and Directions of Future Work

In this article we presented the current state of the art in the field of standardization of web-based real-time communications. We focused on the upcoming new standard known as WebRTC-1.0, by briefly describing both the genesis of this challenging initiative and its evolution toward an agreed upon final specification. We also discussed in some detail the relationship between WebRTC-1.0 and the companion initiative known as *Object Real Time Communications* (ORTC), which has brought a new perspective on how to properly look at and manage the entire media pipeline associated with real-time interaction among web-based devices. Finally, we have highlighted how the two initiatives have eventually converged into a unified effort that has contributed to finalizing the WebRTC-1.0 specification.

The term *WebRTC-NV* refers to the upcoming 'next version' of the WebRTC standard, which has been on purpose called neither WebRTC-1.1 (as proposed by those who are in favor of applying only minor changes to the current spec) nor WebRTC-2.0 (indicating a major departure from the agreed-upon 1.0 version). At the time of this writing, there is indeed no official decision about

the direction that will be followed for this new initiative. Unofficial rumors state that the NV initiative will continue to work on ORTC-style low-level controls while maintaining interoperability with WebRTC-1.0. This means that the most important building blocks of the WebRTC-1.0 architecture (SRTP, RTCP, SCTP over DTLS, and so on) will be supported. Similarly to ORTC, SDP support will not be mandatory at all, and the proposed API will offer direct control over the various components of the media pipeline. Apart from this basic set of requirements, discussions are still ongoing as to whether or not the scope of the working group should be expanded in order to cover all or some of the hot topics we mentioned in the article, for example, simulcast, Scalable Video Coding, Forward Error Correction. Finally, contributors will continue to focus on security and privacy as key areas of interest for the working group.

References

[1] S. Loreto and S. P. Romano, "Real-Time Communications in the Web: Issues, Achievements, and Ongoing Standardization Efforts," *IEEE Internet Computing*, vol. 16, no. 5, Sept.-Oct. 2012, pp. 68–73, DOI: 10.1109/MIC.2012.115

[2] C. Jennings, T. Hardie, and M. Westerlund, "Real-time Communications for the Web," *IEEE Commun. Mag.*, vol. 51, no. 4, April 2013, pp. 20–26, DOI: 10.1109/MCOM.2013.6495756

[3] R. L. Barnes and M. Thomson, "Browser-to-Browser Security Assurances for WebRTC," *IEEE Internet Computing*, vol. 18, no. 6, Nov.-Dec. 2014, pp. 11–17, DOI: 10.1109/MIC.2014.106.

[4] A. Johnston, J. Yoakum, and K. Singh, "Taking on WebRTC in an Enterprise," *IEEE Commun. Mag.*, vol. 51, no. 4, April 2013, pp. 48–54, DOI: 10.1109/MCOM.2013.6495760

[5] J. Rosenberg *et al.*, "SIP: Session Initiation Protocol," Request for Comments (RFC) 3261, Internet Engineering Task Force (IETF).

[6] J. Uberti, C. Jennings, and E. Rescorla, "Javascript Session Establishment Protocol," Internet Draft (work in progress), draft-ietf-rtcweb-jsep-17.txt, expires: April 24, 2017, Internet Engineering Task Force (IETF).

[7] J. Rosenberg, "Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols," Request for Comments (RFC) 5245, Internet Engineering Task Force (IETF).

[8] E. Rescorla and N. Modadugu, "Datagram Transport Layer Security Version 1.2," Request for Comments (RFC) 6347, Internet Engineering Task Force (IETF).

[9] M. Baugher *et al.*, "The Secure Real-time Transport Protocol (SRTP)," Request for Comments (RFC) 3711, Internet Engineering Task Force (IETF).

[10] R. Stewart (Ed.), "Stream Control Transmission Protocol," Request for Comments (RFC) 4960, Internet Engineering Task Force (IETF).

[11] E. Rescorla, "WebRTC Security Architecture," Internet Draft (work in progress), draft-ietf-rtcweb-security-arch-12.txt, expires: Dec. 10, 2016, Internet Engineering Task Force (IETF).

[12] C. Holmberg, H. Alvestrand, and C. Jennings, "Negotiating Media Multiplexing Using the Session Description Protocol (SDP)," Internet Draft (work in progress), draft-ietf-mmusic-sdp-bundle-negotiation-36.txt, expires: Apr. 30, 2017, Internet Engineering Task Force (IETF)

[13] H. Alvestrand, "WebRTC MediaStream Identification in the Session Description Protocol," Internet Draft (work in progress), draft-ietf-mmusic-msid-15.txt, expires: Jan. 8, 2017, Internet Engineering Task Force (IETF)

[14] S. Holmer, M. Shemer, and M. Paniconi, "Handling Packet Loss in WebRTC," *2013 IEEE Int'l. Conf. Image Processing*, Melbourne, VIC, 2013, pp. 1860–64, DOI: 10.1109/ICIP.2013.6738383

[15] C. Perkins and V. Singh, "Multimedia Congestion Control: Circuit Breakers for Unicast RTP Sessions," Internet Draft (work in progress), draft-ietf-avtcore-rtp-circuit-breakers-18.txt, expires: February 19, 2017, Internet Engineering Task Force (IETF).

Biographies

Simon Pietro Romano is an associate professor in the Department of Electrical Engineering and Information Technology (DIETI) at the University of Napoli. He teaches computer networks, network security, and telematics applications. He is also the co-founder of Meetecho, a startup and University spin-off dealing with WebRTC-based unified collaboration. He actively participates in IETF standardization activities, mainly in the applications and real time (ART) area.

Salvatore Loreto works for Ericsson Research in Sweden. He is product manager for the MediaFirst Video Delivery end to end solution, the Operator holistic CDN solution including cloud services like transcoding, repackaging and storage. He is also driving and executing the strategy evolution and technology roadmap toward the 5G networks of Media Delivery business line. He works on standardization, both as an IETF working group chair and as an active participant, mainly in the applications and real time (ART) area.

It has been decided that all future standardization work in WebRTC will take place within the WebRTC Working Group, while the ORTC community group will fade away and its contributors will join the WebRTC effort. Finally, once done with the WebRTC-1.0 milestone, all energies will be devoted to a brand new initiative called WebRTC-NV.