



Real-Time Communications in the Web

Issues, Achievements, and Ongoing Standardization Efforts

Salvatore Loreto • *Ericsson Research*

Simon Pietro Romano • *University of Napoli Federico II*

Web Real-Time Communication (WebRTC) is an upcoming standard that aims to enable real-time communication among Web browsers in a peer-to-peer fashion. The IETF RTCWeb and W3C WebRTC working groups are jointly defining both the APIs and the underlying communication protocols for setting up and managing a reliable communication channel between any pair of next-generation Web browsers.

Support for enabling real-time communication (RTC) in the Web is currently gaining momentum with the two main Internet standardization bodies – the IETF and W3C.

Standardization activities in this area aim to define a W3C API that enables a Web application running on any device – through secure access to input peripherals (such as webcams and microphones) – to send and receive real-time media and data in a peer-to-peer (P2P) fashion between browsers. The API's design must allow Web developers to implement functionality for finding and connecting participants in a communication session. The W3C API will rely on existing protocols the IETF community has identified as the most appropriate for addressing network-related aspects (control protocols, connection establishment and management, connectionless transport, selection of the most suitable encoders and decoders, and so on). However, no clean separation exists between the two standardization activities, which clearly intersect at the interface between the application-level responsibilities residing in a single node and the intercommunication activities among remote nodes.

This migration toward browser-enabled RTC represents a major breakthrough and has motivated

many industry and academic researchers' recent work. Here, we discuss the growing interest in integrating interactive multimedia features into Web applications.

How Did We Get Here?

A few documents have tried to foster browser-enabled RTC in place of traditional communication services. In one notable example, researchers compared the complexity of traditional telecommunications systems with that of Web architectures,¹ concluding that the former should move toward the latter to make this kind of communications available to as many users as possible. This paper also paved the way for a (now expired) Internet draft² describing a RESTful interface to the Session Initiation Protocol (SIP). Researchers and implementors have devoted many other efforts to a similar approach, but often in a nonstandard way (for example, by using proprietary plug-ins such as Adobe Flash or Microsoft ActiveX) and without documentation. A notable exception is recent work from Ericsson Labs in which researchers have tried to add native support for the Real-Time Transport Protocol (RTP) and media devices within browsers themselves by exploiting an ad hoc JavaScript API and a properly modified version

of WebKit that uses gstreamer (see <https://labs.ericsson.com/developer-community/blog/beyond-html5-peerpeer-conversational-video>).

That said, this migration has led the main standardization bodies to devote fresh energy to this problem, eventually leading to two different yet interrelated working groups: RTCWeb (<http://tools.ietf.org/wg/rtcweb/charters>) within the IETF and WebRTC (www.w3.org/2011/04/webrtc-charter.html) within the W3C. RTCWeb has focused on the protocols and interactions that the IETF must address, including interoperability with legacy systems (such as existing telecommunications systems). WebRTC is working to define an API allowing browsers and scripting languages to interact with media devices (microphones, webcams, and speakers), processing devices (encoders/decoders), and transmission functions. Such efforts will likely expand and enhance the HTML5 specification, which already provides a standard way to stream multimedia content from servers to browsers.

Both working groups will have to consider any security issues that arise from the features that will be addressed. We expect (and hope to see) several prototype implementations during these working groups' lifetime.

Open Web Platform

HTML5 is generally used as an umbrella term for the advances taking place on the so-called Open Web Platform, although HTML is itself just one part of the various features used for developing Web applications and commonly referred to as part of that platform. The complete set of features also includes Cascading Style Sheets (CSS), the Document Object Model (DOM) convention, JavaScript, and several scripting APIs.

HTML represents an application and its data in a structured way, and lets developers style the application

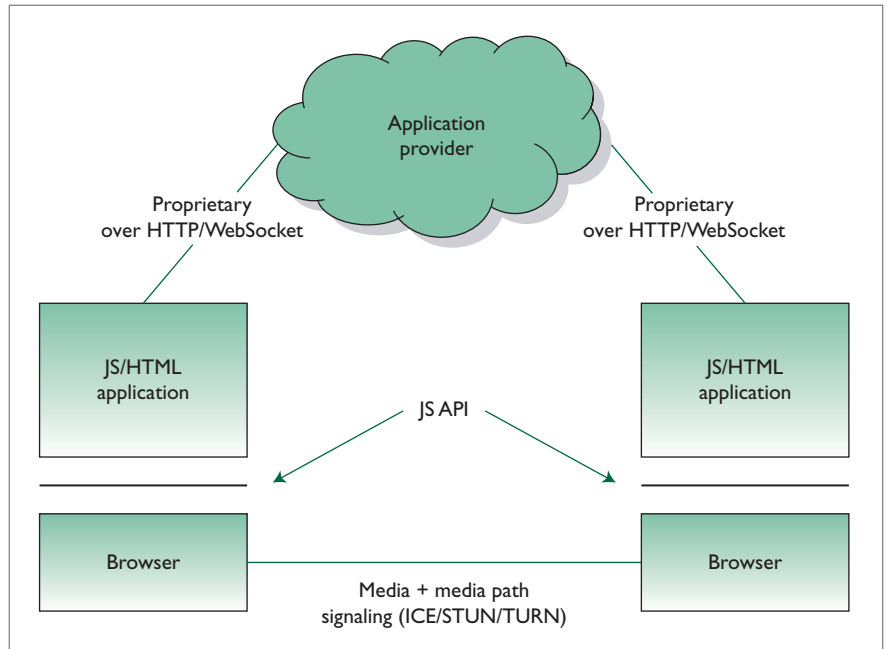


Figure 1. The RTCWeb architecture. This typical voice-over-IP communication trapezoid has a server-mediated signaling path and a direct (browser-to-browser) media path.

with CSS and control it via JavaScript. All these technologies are delivered over the Web infrastructure (via browsers, proxies, and Web servers) using either HTTP or WebSocket (<http://tools.ietf.org/html/rfc6455>).

Scripting APIs let programmers properly control and exploit browser capabilities through JavaScript. Indeed, as soon as new functions are added to a browser, the W3C also devises novel APIs to expose those functions to developers, letting the browser's capabilities grow closer to those of native application environments.

Architecture

RTC's architectural model is the browser RTC trapezoid (see Figure 1), which lets the media path flow directly between browsers without any intervening servers. The signaling path crosses servers that can modify, translate, or manage signals as needed.

The idea is to have client-side Web applications (typically written in a mix of HTML and JavaScript)

interact with Web browsers through the WebRTC API, letting them properly exploit and control browser functions and interact with browsers themselves in both a proactive (for instance, to query browser capabilities) and reactive (to receive browser-generated notifications) way. The aforementioned application-browser API must thus provide a wide set of functions, such as connection management (in a P2P fashion), encoding/decoding capabilities, negotiation, selection and control, media control, firewall, and network address translation (NAT) traversal. From a technical perspective, the API is being implemented in JavaScript, which has demonstrated its effectiveness as a powerful scripting language on the client side of a Web application.

The application-browser API's design represents a challenging issue, but doesn't solve the overall problem at hand. The complete picture, in fact, envisages a continuous, real-time flow of data streams across the network to put into direct communication

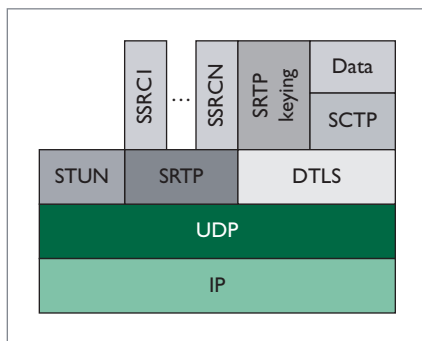


Figure 2. The RTCWeb protocol stack. RTCWeb allows for secure transmission of multiplexed real-time flows over the Internet.

two (or even more) browsers at a time, with no further intermediaries along the path. We're thus talking about browser-to-browser communication, which is a revolutionary approach to Web-based communication because it allows P2P (in which each peer is a browser) data communication to enter the Web application arena for the first time.

Imagine a real-time audio and video call between two browsers. Communication in such a scenario might involve direct media streams between the two browsers, with the media path negotiated and instantiated through a complex sequence of interactions involving the following entities:

- the caller browser and the caller JavaScript application (for example, through the JavaScript API);
- the caller JavaScript application and the application provider (typically, a Web server);
- the application provider and the callee JavaScript application; and
- the callee JavaScript application and the callee browser (again through the application-browser JavaScript API).

Given this overall picture, we next dig into the details of the most relevant features of RTCWeb.

Signaling

Since its inception, the general idea behind WebRTC's design has been to fully specify how to control the media plane while leaving the signaling plane to the application layer as much as possible. The rationale is that different applications might prefer to use different standardized signaling protocols (such as SIP or XMPP) or even something custom. In this approach, the important information that browsers must exchange is the multimedia session description, which specifies the transport (and Interactive Connectivity Establishment [ICE]) information as well as the media type, format, and all associated media configuration parameters necessary to establish the media path.

The original idea to exchange session description information in the form of Session Description Protocol (SDP) "blobs" had several shortcomings, some of which would be difficult to address. Thus, the IETF is standardizing the JavaScript Session Establishment Protocol.³ JSEP provides the interface an application needs to deal with the negotiated local and remote session descriptions (with the negotiation carried out via any desired signaling mechanism), along with a standardized way for the application to interact with the ICE state machine. The JSEP approach leaves the responsibility for driving the signaling state machine entirely to the application. Rather than simply forwarding the messages the browser emits to the remote side, the application must call the right APIs at the right times and convert the session descriptions and related ICE information into the defined messages of its chosen signaling protocol.

API

The W3C WebRTC 1.0 API⁴ lets a JavaScript application exploit the novel browser's real-time capabilities.

It requires the browser core to provide the functionality needed to establish the necessary audio, video, and data channels. However, ongoing standardization work hasn't yet resulted in a decision regarding which audio (G.711, opus, vorbis, and so on) and video (H.264, VP8, and so on) codecs the browser core will use. The current assumption is that all media and data streams will always be encrypted.

The API is being designed around three main concepts: *PeerConnection*, *MediaStreams*, and *DataChannel*.

PeerConnection

A *PeerConnection* lets two users communicate directly, browser-to-browser. It then represents an association with a remote peer, which is usually another instance of the same JavaScript application running at the remote end. Communications are coordinated via a signaling channel provided by scripting code in the page via the Web server – for instance, using XMLHttpRequest or WebSocket. Once the calling browser establishes a peer connection, it can send *MediaStream* objects directly to the remote browser.

As Figure 2 illustrates, the peer-connection mechanism uses the ICE protocol along with the Session Traversal Utilities for NAT (STUN) and Traversal Using Relays around NAT (TURN) servers to let UDP-based media streams traverse NAT boxes and firewalls. ICE lets browsers discover enough information about the topology of the network on which they're deployed to find the best exploitable communication path. Using ICE also provides a security measure because it prevents untrusted webpages and applications from sending data to hosts that aren't expecting to receive it.

The remote host feeds each signaling message into the receiving *PeerConnection* on arrival. The APIs send signaling messages that most

applications will treat as opaque blobs, but which the Web application must transfer securely and efficiently to the other peer via the Web server.

MediaStream

A *MediaStream* is an abstract representation of an actual data stream of audio or video. It serves as a handle for managing actions on the media stream – such as displaying the stream’s content, recording it, or sending it to a remote peer. A *MediaStream* can be extended to represent a stream that either comes from (remote stream) or is sent to (local stream) a remote node. A *LocalMediaStream* represents a media stream from a local media-capture device (such as a webcam or microphone).

To create and use a *LocalMediaStream*, the Web application must request access from the user via the `getUserMedia()` function. The application specifies the type of media – audio or video – to which it requires access. The devices selector in the browser interface grants or denies access. Once the application is done, it can revoke its own access by calling the `stop()` function on the *LocalMediaStream*.

Media-plane signaling is carried out-of-band between peers. Figure 2 shows the protocol stack needed for media transmission. Secure Real-time Transport Protocol (SRTP) carries the media data and the RTP Control Protocol (RTCP) information used to monitor transmission statistics associated with data streams. Datagram Transport Layer Security (DTLS) is used as an SRTP key and for association management. The IETF is still discussing the option of using SDP Security Descriptions for Media Streams (SDS) as an alternative key and for association management.

In a multimedia session, each medium is typically carried in a separate RTP session with its own RTCP packets. However, to overcome

the issue of opening a new “hole” for each stream used, the IETF is working on possibly reducing the number of transport layer ports that RTP-based real-time applications consume by combining (that is, multiplexing) multimedia traffic in a single RTP session.⁵

DataChannel

The *DataChannel* provides a generic transport service that lets Web browsers exchange generic data in a bidirectional, P2P fashion. Within the IETF, standardization work has reached a general consensus on using the Stream Control Transmission Protocol (SCTP) encapsulated in DTLS to handle nonmedia data types.⁶ Encapsulating “SCTP over DTLS over ICE over UDP” provides a NAT traversal solution together with confidentiality, source authentication, and integrity-protected transfers. Moreover, this solution lets the data transport interwork smoothly with parallel media transports, and both can share a single transport-layer port number. The IETF chose SCTP because it natively supports multiple streams with reliable, unreliable, and partially reliable delivery modes. It allows applications to open several independent streams (up to 65,536 in each direction) within an SCTP association toward a peering SCTP endpoint. Each stream actually represents a unidirectional logical channel, providing in-sequence delivery.

An application can send a message sequence either ordered or unordered. The message delivery order is preserved only for all ordered messages sent on the same stream. However, the *DataChannel* API is bidirectional, which means that each *DataChannel* bundles an incoming and an outgoing SCTP stream.

An application sets up a data channel (that is, creates the SCTP association) when the `CreateDataChannel()` function is called for the first time on

an instantiated *PeerConnection* object. Each subsequent call to the `CreateDataChannel()` function just creates a new *DataChannel* within the existing SCTP association.

A Simple Example

Alice and Bob use a common calling service. To communicate, they must be simultaneously connected to the Web server that implements that service. Indeed, when Bob (or Alice) points his browser to the calling service webpage, he will download both the HTML page and a JavaScript that keeps the browser connected via a secure HTTP or WebSocket connection.

Figure 3 illustrates a typical call flow associated with setting up a real-time, browser-enabled communication channel. When Alice clicks on the webpage button to start a call with Bob, the JavaScript instantiates a *PeerConnection* object. Once the *PeerConnection* is created, the JavaScript code on the calling service side must set up media, which it does via the *MediaStream* function. Alice must also grant a permission to let the calling service access both her camera and her microphone.

In the current W3C API, once an application has added some streams, Alice’s browser, enriched with JavaScript code, generates a signaling message. The IETF hasn’t yet defined the exact format of this message. It must contain media channel information and ICE candidates, as well as a fingerprint attribute binding the communication to Alice’s public key. Alice’s browser then sends this message to the signaling server (for example, via `XMLHttpRequest` or `WebSocket`). The signaling server processes the message from Alice’s browser, determines that this is a call to Bob, and sends a signaling message to Bob’s browser. The JavaScript on Bob’s browser processes the incoming message and alerts Bob. Should Bob decide to answer the

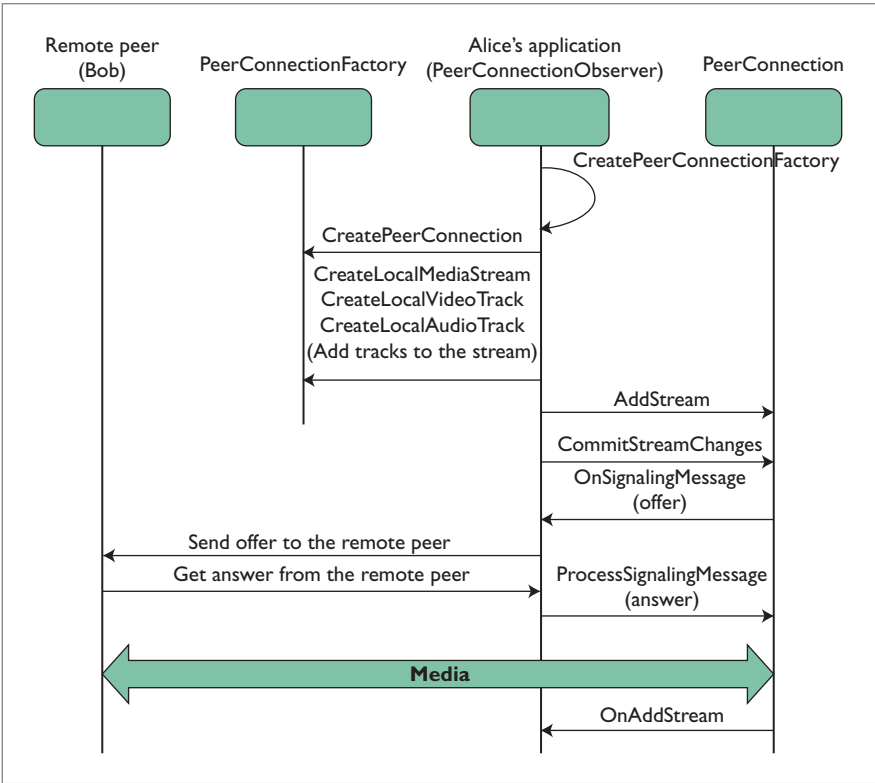


Figure 3. Call setup from Alice's perspective. We can use the WebRTC API to create a direct media connection between Alice's and Bob's browsers.

call, the JavaScript running in his browser would then instantiate a *PeerConnection* related to the message coming from Alice's side. Then, a process similar to the one on Alice's browser would occur. Bob's browser verifies that the calling service is approved and the media streams are created; afterward, it creates a signaling message containing media information and ICE candidates, and returns a fingerprint to Alice via the signaling service.

Congestion Control

The discussion of a congestion control mechanism specifically conceived for interactive media and data transmissions is at an early stage – the IETF hasn't even decided whether to begin working on it. The idea is to tightly couple the congestion control of streams, ideally using only a single congestion control instance for all the WebRTC transfers (that is, audio, video, or data). At the same

time, prioritizing different parts of the WebRTC transport bundle should be possible.

Security Considerations

As we described, the RTCWeb approach and related architecture definitely represent a new challenge in the world of telecommunications because they allow Web browsers to directly communicate with little or no intervention from the server side. This is extremely challenging because it requires that we consider several issues, among which trust and security play a fundamental role. With respect to this last point, a new set of potential security threats comes to the fore when we allow direct browser-to-browser communication. The basic Web security policy currently in place is based on the principle of isolation (also known as *sandboxing*), which lets users protect their computers from malicious scripts and cross-site content references.

With this model, security policies mainly relate to JavaScript code running in the browser, which actually acts as a trusted computing base for the visited sites. Widening the scope to browser-to-browser communications unveils new facets of the general security issue.

First and foremost, we must consider communications security in much the same way as we do with other network protocols (such as SIP) that allow for direct communication between any two endpoints – unless we envisage the presence of relays acting as transparent intermediaries.

Second, if we let browsers directly communicate between each other, mechanisms must exist that let users verify consent before the actual data exchange phase starts. Consent verification shouldn't rely on the presence of a trusted server and should hence be directly enforced by the browser aiming at initiating communication with a potential peer. Last but not least, RTC scenarios through the Web clearly call for the browser to interact on a deep level with the node on which it resides – for example, to access local audio and video devices before making a multimedia call with a target peer. Access policies must be defined that potentially involve some form of user consent, thus opening up several new challenges.

The RTCWeb vision is to standardize an open framework enabling browser-to-browser multimedia applications along the most straightforward path, with no need to install plug-ins. Two major browser vendors have already made available an early implementation of the framework in their developers' release. However, neither is fully compliant to the standard as of yet, even if they are expected to become so soon.

In the near future, standardization activities will focus on congestion control, audio and video codec


selection, telepresence, and enhanced usage of data channels. □

References

1. H. Sinnreich and W. Wimmreuter, "Communications on the Web," *Elektrotechnik und Informationstechnik*, vol. 127, 2010, pp. 187–194; <http://dx.doi.org/10.1007/s00502-010-0742-1>.
2. H. Sinnreich and A. Johnston, "Sip APIs for Communications on the Web," IETF Internet draft, June 2010.
3. J. Uberti and C. Jennings, "Javascript Session Establishment Protocol" IETF Internet draft, June 2012.
4. A. Bergkvist et al., "WebRTC 1.0: Real-Time Communication between Browsers," W3C working draft 09, Feb. 2012; www.w3.org/TR/webrtc/.
5. C. Holmberg and H. Alvestrand, "Multiplexing Negotiation Using Session Description Protocol (SDP) Port Numbers," IETF Internet draft, Feb. 2012.
6. R. Jesup, S. Loreto, and M. Tuexen, "RTC-Web Datagram Connection," IETF Internet draft, Mar. 2012.

Salvatore Loreto is a senior researcher in the MultiMedia Technology branch of the NomadicLab at Ericsson Research Finland. He's made contributions in Internet transport protocols, signal protocols, VoIP, IP-telephony convergence, conferencing over IP, 3GPP IP Multimedia Subsystem, HTTP, and Web technologies. Loreto has a PhD in computer networking from Napoli University. He serves within the IETF as cochair of the SIP Overload Control (SOC), BiDirectional or Server-Initiated HTTP (HyBi), and Application Area (Appsawg) working groups. Contact him at salvatore.loreto@ieee.org.

Simon Pietro Romano is an assistant professor in the Computer Engineering Department at the University of Napoli, and cofounder of Meetecho, a startup (and spin-off of the university) focusing on Internet-based conferencing and collaboration. His research interests primarily fall in the field of networking, with special regard to real-time multimedia applications, network security, and autonomic network management. Romano has a PhD in computer networks from the University of Napoli. He actively participates in IETF standardization activities in the Real-time Applications and Infrastructure area.

 Selected CS articles and columns are also available for free at <http://ComputingNow.computer.org>.

NEW

**ESSENTIAL INDUSTRIAL
IMPLEMENTATIONS OF
FLOATING-POINT UNITS
DURING THE LAST DECADE:**

VOLUMES 1 & 2

Transactions on Computers {EssentialSets} Available:

Edited by TC AE Elisardo Antelo, this EssentialSet surveys the industrial design of floating-point units during the last decade. This EssentialSet is broken into two volumes, sold separately.

PDF edition • \$15 each (\$9 members)

Order Online: computer.org/store.



IEEE  computer society

 CPress