WebRTC:
APIs and RTCWEB Protocols of the HTML5 Real-Time Web

Alan B. Johnston

Daniel C. Burnett

Second Edition
June 2013

C:
Digital Codex LLC

Ø Ø Ø 1

P.O. Box 9131,
St. Louis, MO 63117,
USA

http://digitalcodexllc.com

Smashwords Edition

ISBN-13: 978-0-9859788-5-3
ISBN-10: 0985978856

DEDICATION

For Aidan & Nora, and Craig & Fiona

ACKNOWLEDGMENTS

We would like to thank our technical reviewers Alex Agranovsky, Carol Davids, Emil Ivov, David Kemp, Henry Sinnreich, Harvey Waxman, and Dan York. We would also like to thank Marina Burnett and Chris Comfort for their proofreading and comments. We would also like to thank our families for their encouragement and support.

And finally, we would like to acknowledge our colleagues in W3C and the IETF who are working incredibly hard at creating the WebRTC standards.

PREFACE

WebRTC continues to evolve and grow in the handful of months since we published the first edition of WebRTC: APIs and Protocols of the HTML5 Real-Time Web.

There has been real progress in many areas in the IETF and W3C standards, although much work remains. Eventing, stream representation at the protocol level, and even the syntax for some callbacks are all still very much under discussion, while details such as how multiple video tracks within a single MediaStream work and what should happen when a MediaStream attached to an HTML element has tracks added or removed are only now beginning to be considered. Nevertheless, the core APIs are firming up. On the usage side, conferences, meetups, and startups have sprung up and are growing rapidly in size, with the World seeking to understand WebRTC's impact and opportunities.

Other trends have become clear as well. The disagreement about codecs, especially video codecs is escalating from a fight, to a war, to Mutually Assured Patent Destruction. The authors sincerely hope that the standards and industry come to agreement quickly on the mandatory to implement video codec very soon.

While the basic parts of WebRTC are working well in demos and applications today, many of the concerns and issues with WebRTC revolve around security and the signaling channel, which have complete new chapters in this second edition. Additionally, the sample code out there on the Web today is often either too complex or insufficiently explained, motivating the complete running and completely explained example in this new edition. This demo code, running on both Chrome and Firefox, is also on our website at http://demo.webrtcbook.com .

Note that all code and lists will look best with the smallest font size of your device.

We hope that this new updated edition keeps you excited and informed about WebRTC. Happy reading!

PREFACE TO THE FIRST EDITION

The Internet and the World Wide Web have changed our world. When the history of this period is written, much will be said about the impact of these technologies on life in the late 20th and early 21st centuries. The web has changed the way we receive information, interact with others, work, and play. Now, the web is about to dramatically change the way we communicate using voice and video. This book gives an up-to-the-minute snapshot of the standards and industry effort known as WebRTC, which is short for Web Real-Time Communications. This technology, along with other advances in HTML5 browsers, has the potential to revolutionize the way we all communicate, in both personal and business spheres.

The authors have been involved in the Internet Communications industry for many years, and have seen the advances and impact of the Internet on voice and video communications. We have worked on signaling protocols such as Session Initiation Protocol (SIP), Session Description Protocol (SDP), and security protocols such as ZRTP for voice and video communication systems that will form the basis of what will inevitably replace the telephone system (called the Public Switched Telephone Network or PSTN). These Internet Communications technologies have brought an amazing wave of disruption, but we believe WebRTC has the potential to create even greater disruption.

This book provides information for web developers and telephony developers who want to catch this new wave while it is still building. The standards and protocols needed for WebRTC are still being developed and invented. Browsers are starting to support WebRTC functionality, little by little. However, the authors have seen the need for a book to explain this still-developing technology. This book will explain the technical goals, architectures, protocols, and Application Programming Interfaces (APIs) of WebRTC. In a publishing experiment, we plan to produce frequent editions of this book, perhaps as often as three per year, and focus on digital delivery and on-demand publishing to keep costs down and for maximum hyper linking usefulness. For information on the latest edition and for a list of updates and changes, visit http://webrtcbook.com.

This book begins with an introduction to WebRTC and discusses what is new about it. The unique aspects of WebRTC peer-to-peer media flows are explored, and Network Address Translation (NAT) traversal explained. We then discuss the working documents and finalized documents that together comprise the WebRTC standards-in-

progress in both the World Wide Web Consortium (W3C) and the Internet Engineering Task Force (IETF). Each chapter ends with a references section, listing the various standards documents. References of the form [RFC…] are IETF Request For Comments documents. References of the form [draft-…] are IETF Internet-Drafts, working documents, whose content may have been updated or changed since the publication of this book. The hyperlinks provided will, in most cases, retrieve the most recent version of the document. References to W3C drafts include a link to the latest public working draft, and also a link to the latest editor's draft. For those of you unfamiliar with the standardization processes of W3C and the IETF, we have provided a reference in Appendices A and B. Finally, we discuss the current state of deployment in popular browsers.

If you are a web developer, welcome to the world of Internet Communications! Your users will greatly enjoy the ability to interact with each other using your application's real-time voice and video capabilities. To understand our descriptions of APIs, you will need a working knowledge of HTML and JavaScript, and some experience in web applications. See Appendix D for some useful references for this.

If you are a VoIP or telephony developer, welcome to the web world! Your users will enjoy the capabilities of high quality audio and video communication, and rich, web-powered user interfaces. To understand our descriptions of the on-the-wire protocols for transporting voice, video, and data, you will need a basic understanding of the Internet. Knowledge of another Internet Communication signaling protocol such as SIP or Jingle is also useful. See Appendix D for some additional useful reference reading.

In many ways, WebRTC is a merging of worlds between the web and telephony. To help bridge the gap between the web and telephony world, we have also included a Glossary in Appendix C to briefly explain some common terms and concepts from each world.

The authors look forward to participating in the next wave of disruption and innovation that WebRTC will likely unleash.

We would love to hear from you and interact with you on Twitter (@alanbjohnston and @danielcburnett) or on Google+ (alanbjohnston@gmail.com and danielcburnett@gmail.com).

1 INTRODUCTION TO WEB REAL-TIME COMMUNICATIONS

Web Real-Time Communications (RTC), or WebRTC, adds new functionality to the web browser. For the first time, browsers will interact directly with other browsers, resulting in a number of architectures including a triangle and trapezoid model. The media capabilities of WebRTC are state-of-the-art, with many new features. The underlying standards of WebRTC are being developed by the World Wide Web Consortium (W3C) and the Internet Engineering Task Force (IETF).

1.1 WebRTC Introduction

WebRTC is an industry and standards effort to put real-time communications capabilities into all browsers and make these capabilities accessible to web developers via standard [HTML5] tags and JavaScript APIs (Application Programming Interfaces). For example, consider functionality similar to that offered by Skype™ [SKYPE] but without having to install any software or plug-ins. For a website or web application to work regardless of which browser is used, standards are required. Also, standards are required so that browsers can communicate with non-browsers, including enterprise and service provider telephony and communications equipment.

1.1.1 The Web Browsing Model

The basic model of web applications is shown in Figure 1.1. Transport of information between the browser and the web server is provided by the Hyper-Text Transport Protocol, HTTP (Section 6.2.1), which runs over Transmission Control Protocol, TCP (Section 6.2.9), or in some new implementations, over the WebSocket protocol (see Section 6.2.2). The content or application is carried in Hyper-Text Markup Language, HTML, which typically includes JavaScript and Cascading Style Sheets [CSS]. In the simple case, the browser sends an HTTP request to the web server for content, and the web server sends a response containing the document or image or other information requested. In the more complex case, the server sends JavaScript which runs on the browser, interacting with the browser through APIs and with the user through clicks and selects. The browser exchanges information with the server through an open HTTP or WebSockets channel.

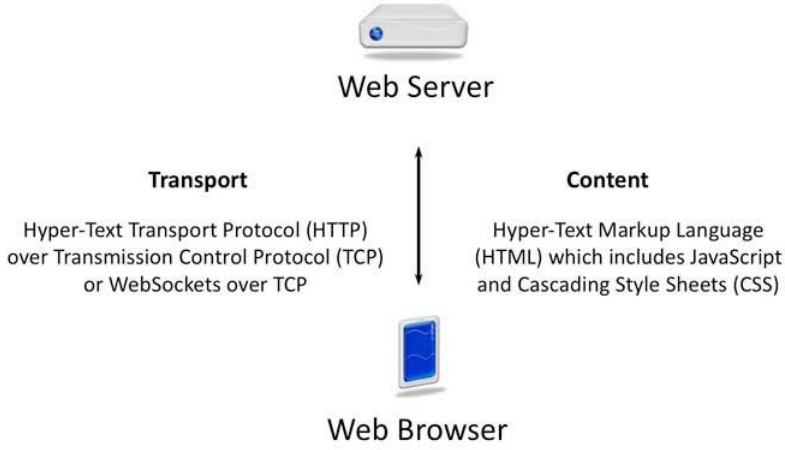


Figure 1.1 Web Browser Model

In figures in this book, we will show an arrow between the web browser and the web server to indicate the web session between them. Since WebRTC can utilize any web transport, the details of this connection, and whether it is HTTP or WebSockets is not discussed.

1.1.2 The Real-Time Communication Function in the Browser

Figure 1.2 shows the browser model and the role of the real-time communication function. The lighter block called "Browser RTC Function" is the focus of this book. The unique nature and requirements of real-time communications means that adding and standardizing this block is non-trivial. The RTC function interacts with the web application using standard APIs. It communicates with the Operating System using the browser. A new aspect of WebRTC is the interaction that occurs browser-to-browser, known as a "Peer Connection", where the RTC Function in one browser communicates using on-the-wire standard protocols (not HTTP) with the RTC Function in another browser or Voice over IP (VoIP) or video application. While web traffic uses TCP for transport, the on-the-wire protocol between browsers can use other transport protocols such as User Datagram Protocol, UDP. Also new is the Signaling Server, which provides the signaling channel between the browser and the other end of the Peer Connection.

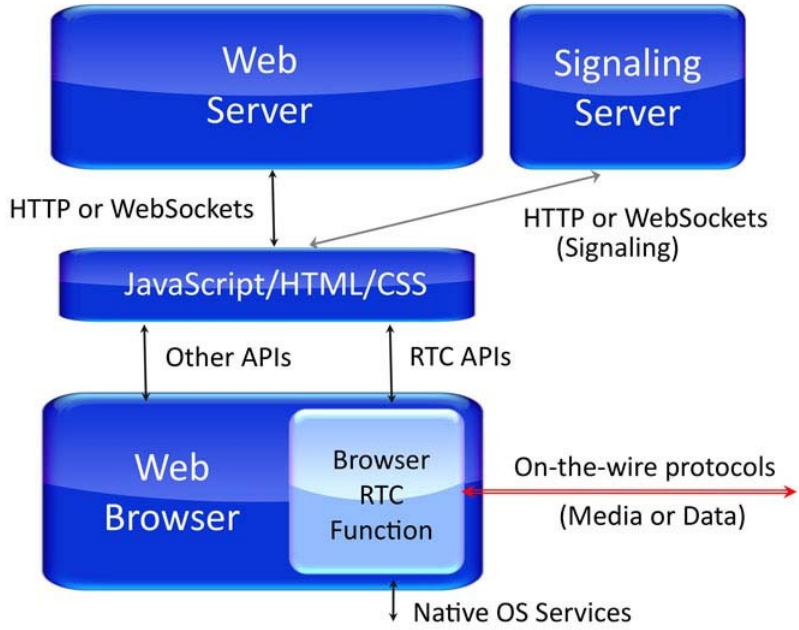


Figure 1.2 Real-Time Communication in the Browser

1.1.3 Elements of a WebRTC System

Figure 1.3 shows a typical set of elements in a WebRTC system. This includes web servers, browsers running various operating systems on various devices including desktop PCs, tablets, and mobile phones, and other servers. Additional elements include gateways to the Public Switched Telephone Network (PSTN) and other Internet communication endpoints such as Session Initiation Protocol (SIP) phones and clients or Jingle clients. WebRTC enables communication among all these devices. The figures in this book will use these icons and elements as examples.

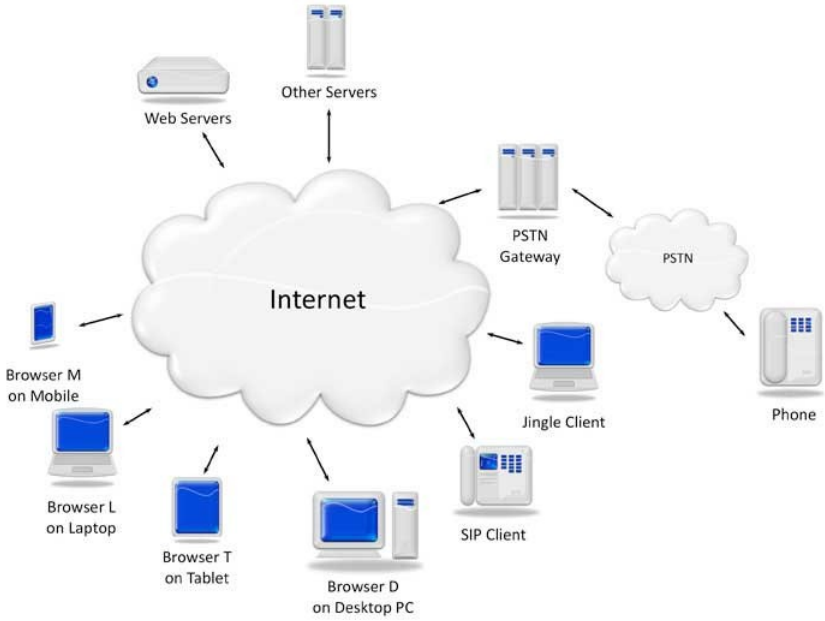


Figure 1.3 Elements in a WebRTC Environment

1.1.4 The WebRTC Triangle

Initially, the most common scenario is likely to be where both browsers are running the same WebRTC web application, downloaded from the same webpage. This produces the WebRTC "Triangle" shown in Figure 1.4. This arrangement is called a triangle due to the shape of the signaling (sides of triangle) and media or data flows (base of triangle) between the three elements. A Peer Connection establishes the transport for voice and video media and data channel flows directly between the browsers.

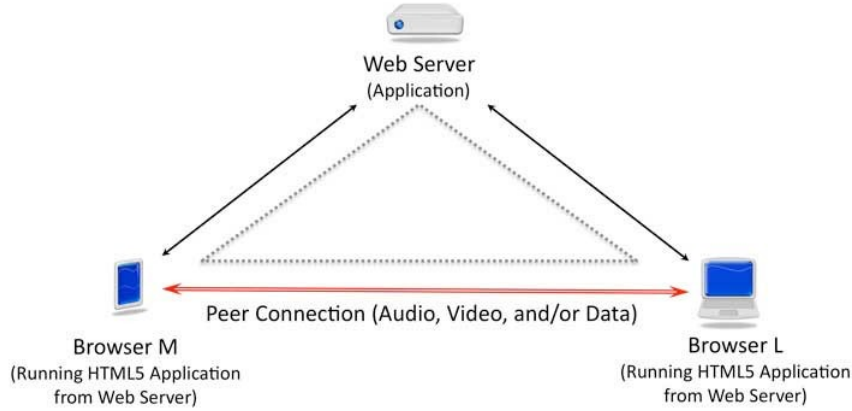


Figure 1.4 The WebRTC Triangle

Note that while we sometimes refer to the connection between the browser and server as signaling, it is not really signaling as used in telephony systems. Signaling is not standardized in WebRTC as it is just considered part of the application. This signaling may run over HTTP or WebSockets to the same web server that serves HTML pages to the browser, or to a completely different web server that just handles the signaling.

1.1.5 The WebRTC Trapezoid

Figure 1.5 shows the WebRTC Trapezoid [draft-ietf-rtcweb-overview], based on the SIP Trapezoid [RFC3261]. The two web servers are shown communicating using a standard signaling protocol such as Session Initiation Protocol (SIP), used by many VoIP and video conferencing systems, or Jingle [XEP-0166], used to add voice and video capability to Jabber [RFC6120] instant messaging and presence systems. Alternatively, a proprietary signaling protocol could be used. Note that in these more complicated cases, the media may not flow directly between the two browsers, but may go through media relays and other elements, as discussed in Chapter 3.

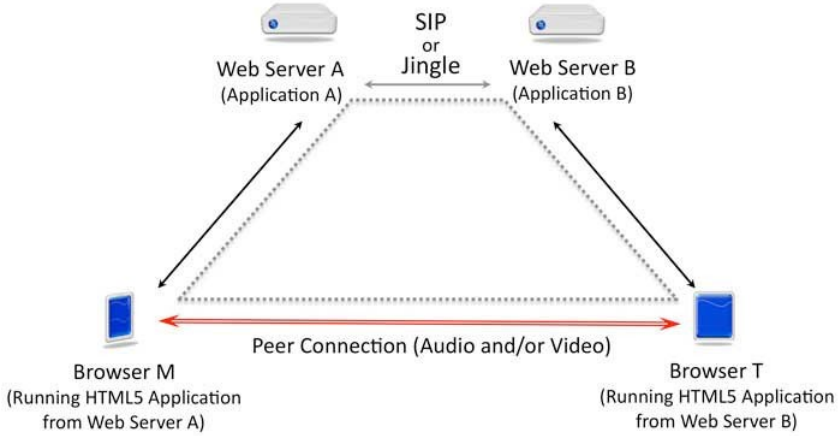


Figure 1.5 The WebRTC Trapezoid

1.1.6 WebRTC and the Session Initiation Protocol (SIP)

Figure 1.6 shows WebRTC interoperating with SIP. The Web Server has a built-in SIP signaling gateway to allow the call setup information to be exchanged between the browser and the SIP client. The resulting media flow is directly between the browser and the SIP client, as the Peer Connection establishes a standard Real-time Transport Protocol (RTP) media session (Section 6.2.3) with the SIP User Agent. Other ways of interoperating with SIP are covered in Section 2.2.6.

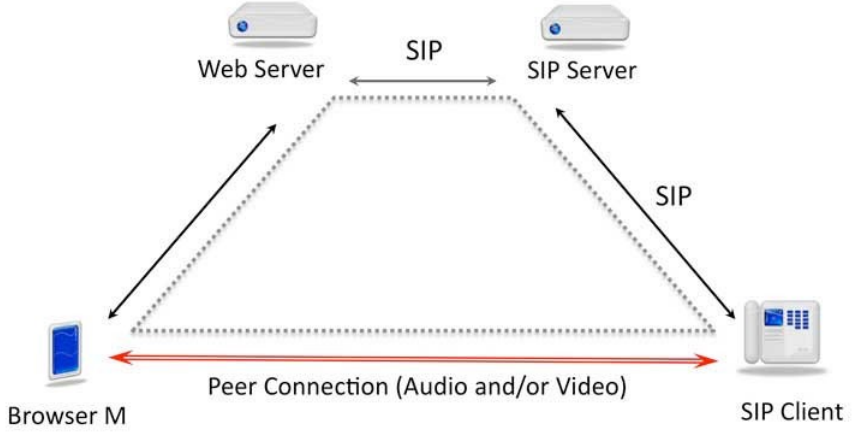


Figure 1.6 WebRTC Interoperating with SIP

1.1.7 WebRTC and Jingle

Figure 1.7 shows how WebRTC can interoperate with Jingle. The Web

Server has a built-in Extensible Messaging and Presence Protocol, XMPP [RFC6120], also known as Jabber, server which talks through another XMPP server to a Jingle client.

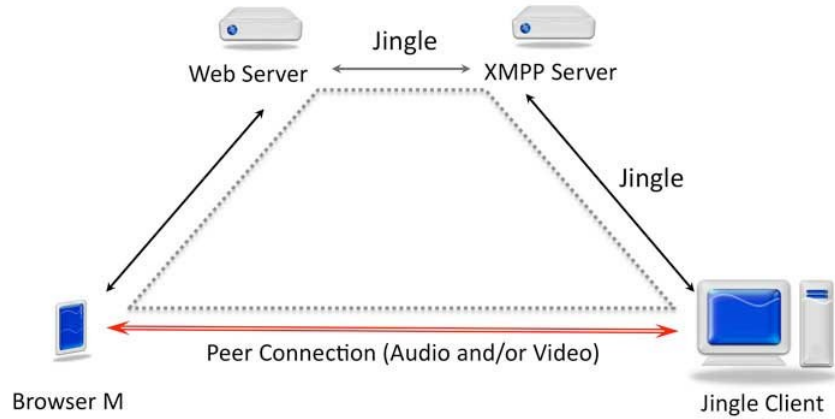


Figure 1.7 WebRTC Interoperating with Jingle

1.1.8 WebRTC and the Public Switched Telephone Network (PSTN)

Figure 1.8 shows how WebRTC can interoperate with the Public Switched Telephone Network (PSTN). The PSTN Gateway terminates the audio-only media stream and connects the PSTN telephone call with the media. Some sort of signaling is needed between the Web Server and the PSTN Gateway. It could be SIP, or a master/slave control protocol.

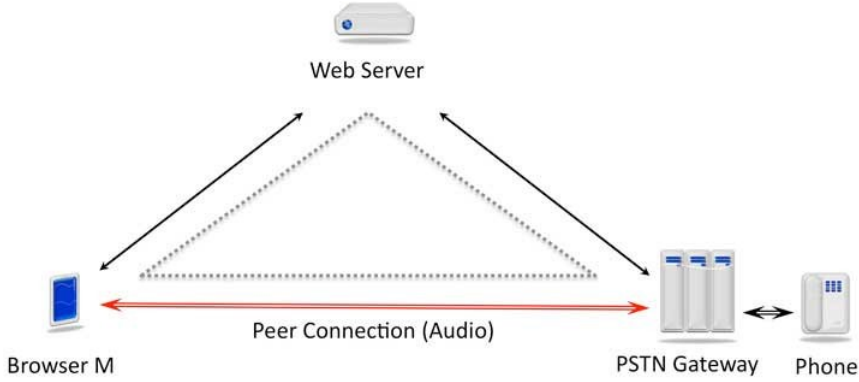


Figure 1.8 WebRTC Interoperating with the PSTN

It is not expected that browsers will be assigned telephone numbers or be part of the PSTN. Instead, an Internet Communication service could assign a telephone number to a user, and that user could use WebRTC to

access the service. As a result, a telephone call to that PSTN number would "ring" the browser and an answered call would result in an audio session across the Internet connected to the PSTN caller. Other services could include the ability to "dial" a telephone number in a WebRTC application which would result in the audio path across the Internet to the PSTN.

Note that the phone in Figure 1.8 could be a normal PSTN phone ("landline" or "black phone") or a mobile phone. The fact that it might be running VoLTE (Voice over Long Term Evolution) or other VoIP (Voice over Internet Protocol) protocol doesn't change this picture, as the Peer Connection will terminate with a VoIP gateway.

Another interesting area is the role of WebRTC in providing emergency services. While a WebRTC service could support emergency calling in the same way as VoIP Internet Communication services, there is the potential that the Public Service Answering Point (PSAP) could become a WebRTC application, and answer emergency "calls" directly from other browsers, completely bypassing the PSTN. Of course, this raises all kinds of interesting security, privacy, and jurisdiction issues.

1.2 Multiple Media Streams in WebRTC

Devices today can generate and consume multiple media types and multiple streams of each type. Even in the simple point-to-point example shown in Figure 1.9, a mobile phone and a desktop PC could generate a total of six media streams. For multiparty sessions, this number will be much higher. As a result, WebRTC has built-in capabilities for dealing with multiple media streams and sources.

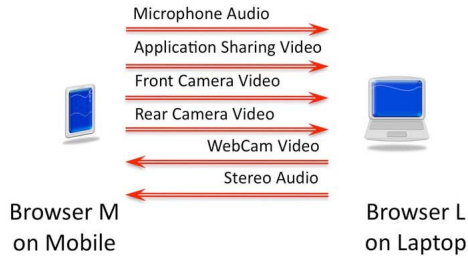


Figure 1.9 Multiple Media Streams in a Point-to-Point WebRTC Session

1.3 Multi-Party Sessions in WebRTC

The preceding examples have been point-to-point sessions between two browsers, or between a browser and another endpoint. WebRTC also supports multi-party or conferencing sessions involving multiple browsers.

One way to do this is to have each browser establish a Peer Connection with the other browsers in the session. This is shown in Figure 1.10. This is sometimes referred to as a "full mesh" or "fully distributed" conferencing architecture. Each browser establishes a full mesh of Peer Connections with the other browsers. For audio media, this might mean mixing the media received from each browser. For video, this might mean rendering the video streams from other browsers to different windows with appropriate labeling. As new browsers join the session, new Peer Connections are established to send and receive the new media streams.

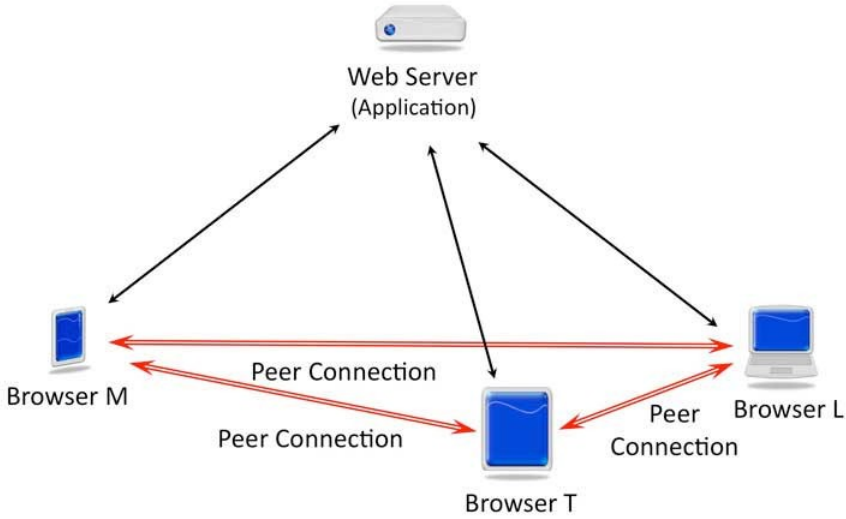


Figure 1.10 Multiple Peer Connections Between Browsers

An alternative architecture to the full-mesh model of Figure 1.10 is also possible with WebRTC. For a multiple browser conference, a centralized media server/mixer/selector can be used; this requires only a single Peer Connection to be established between each browser and the media server. This is shown in Figure 1.11. This is sometimes referred to as a "centrally mixed" conferencing architecture. Each browser sends media to the server, which distributes it to the other browsers, with or without mixing. From the perspective of browser M, media streams from browser L and T are received over a single Peer Connection from the server. As new browsers join the session, no new Peer Connections involving browser M need to be established. Instead, new media streams are received over the existing Peer Connection between browser M and the media server.

The full-mesh architecture of Figure 1.10 has the advantages of no media server infrastructure, and lowest media latency and highest quality. However, this architecture may not be suitable for a large multi-party conference because the bandwidth required at each browser grows with each new participant. The centralized architecture of Figure 1.11 has the advantage of being able to scale to very large sessions while also minimizing the amount of processing needed by each browser when a new participant joins the session, although it is perhaps inefficient when only one or a small number of browsers are involved, such as in peer-to-peer gaming.

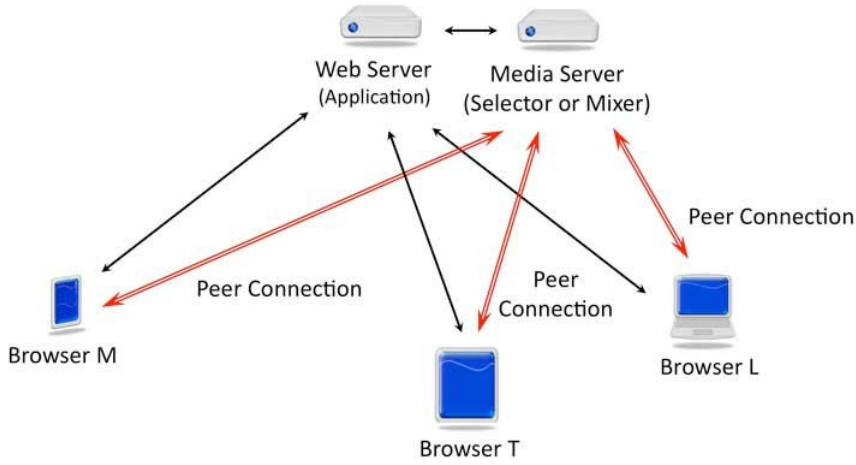


Figure 1.11 Single Peer Connection with Media Server

1.4 WebRTC Standards

The WebRTC standards are currently under joint development by the World Wide Web Consortium (W3C) [W3C] and the Internet Engineering Task Force (IETF) [IETF]. W3C is working on defining the APIs needed for JavaScript web applications to interact with the browser RTC function. These APIs, such as the Peer Connection API, are described in Chapter 5. The IETF is developing the protocols used by the browser RTC function to talk to another browser or Internet Communications endpoint. These protocols, for example, extensions to the Real-time Transport Protocol, are described in Chapter 8.

There are pre-standard implementations of many of the components of WebRTC in some browsers today. See Chapter 8 for details.

Note that there is an important distinction between 'pre-standard' and 'proprietary' implementations. Pre-standard implementations

emerge during the development stage of standards, and are critical to gain experience and information before standards are finalized and locked down. Pre-standard implementations often follow an early or draft version of the standards, or partially implement standards as a 'proof of concept'. Once the standard has been finalized, these pre-standard implementations must move towards the standards, or else they risk becoming a proprietary implementation. Proprietary implementations fragment the user and development base, which in an area such as communications can greatly reduce the value of the services.

The W3C work is centered around the WEBRTC Working Group and the IETF work is centered around the RTCWEB (Real-Time Communications Web) Working Group. The two groups are independent, but closely coordinate together and have many common participants, including the authors.

The projected time frame for publication of the first version of standards in the IETF and W3C is in 2014. However, these dates are most likely overly optimistic. (When do engineers ever realistically estimate level of effort?) Standards-compliant WebRTC browsers are expected to be generally available sometime in late 2014.

1.5 What is New in WebRTC

There are many new and exciting capabilities in WebRTC that are not available even in today's VoIP and video conferencing systems. Some of these features are listed in Table 1.1. The rest of this book will explain how these are achieved using the WebRTC APIs and protocols.

1.6 Important Terminology Notes

In this book, when we refer to the entire effort to add standardized communication capabilities into browsers, we shall use WebRTC. When we are referring to the W3C Working Group, we will use WEBRTC. When we are referring to the IETF Working Group, we will use RTCWEB. Note that WebRTC is also used to describe the Google/Mozilla open source media engine [WEBRTC.ORG], which is an implementation of WebRTC.

In addition, because the main W3C specification is titled "The WebRTC Specification" [WEBRTC 1.0], we use its full title to reference this particular W3C document, which is a key part of WebRTC, but by no means the entire specification.

Also note that the World Wide Consortium refers to itself as "W3C"

and not “the W3C”. We have adopted this convention throughout this book.

Feature	Provided Using	Why Important
Platform and device independence	Standard APIs from W3C, standard protocols from IETF	Developers can write WebRTC HTML5 code that will run across different OS, browsers, and devices, desktop and mobile.
Secure voice and video	Secure RTP Protocol (SRTP) encryption and authentication	Browsers are used in different environments and over unsecured WiFi networks. Encryption means that others can't listen in or record voice or video.
Advanced voice and video quality	Opus audio codec, VP8 video codec, and others	Having built-in standard codecs ensures interoperability and avoids codec downloads, a way malicious sites install spyware and viruses. New codecs can adapt when congestion is detected.
Reliable session establishment	Hole punching through Network Address Translation (NAT)	Direct media between browsers is noticeably more reliable and better quality than server-relayed media. Also, the load on servers is reduced.
Multiple media streams and media types sent over a single transport	Real-time Transport Protocol (RTP) and Session Description Protocol (SDP) extensions	Establishing direct media using hole punching can take time. Sending all media over a single session is also more efficient and reliable.
Adaptive to network conditions	Multiplexed RTP Control Protocol (RTCP), Secure Audio Video Profile with Feedback (SAVPF)	Feedback on network conditions is essential for video, and will be especially important for the high definition, high bandwidth sessions in WebRTC.
Support for multiple media types and multiple sources of media	APIs and signaling to negotiate size/format of each source individually	The ability to negotiate each individually results in most efficient use of bandwidth and other resources.
Interoperability with VoIP and video communication systems using SIP, Jingle, and PSTN	Standard Secure RTP (SRTP) media, Standard SDP and extensions	Existing VoIP and video systems can work with new WebRTC systems using standard protocols.

Table 1.1 New Features of WebRTC

1.7 References

[HTML5] http://www.w3.org/TR/html5

[SKYPE] http://www.skype.com

[CSS] http://www.w3.org/Style/CSS

[draft-ietf-rtcweb-overview] http://tools.ietf.org/html/draft-ietf-rtcweb-overview

[RFC3261] http://tools.ietf.org/html/rfc3261

[XEP-0166] http://xmpp.org/extensions/xep-0166.html

[RFC6120] http://tools.ietf.org/html/rfc6120

[W3C] http://www.w3c.org

[IETF] http://www.ietf.org

[WEBRTC.ORG] http://www.webrtc.org

[WEBRTC 1.0] http://www.w3.org/TR/webrtc

2 HOW TO USE WEBRTC

WebRTC is easy to use, with just a few steps necessary to establish media sessions. A number of messages flow between the browser and the server, while others flow directly between the two browsers, known as peers. WebRTC can even establish sessions with SIP, Jingle, and PSTN endpoints. There are many standards involved in the WebRTC effort – so many that it can be difficult to know where to start when learning about it. Since many readers of this text will likely be developers of WebRTC applications, the following sections give an overview of how to set up a WebRTC session, what can be done while the session is running, and how to close down the session. WebRTC can be used in a number of different architectures. Example pseudo code illustrates the operation of the WebRTC APIs.

2.1 Setting Up a WebRTC Session

As an application developer, the four main actions to take when setting up a WebRTC session are:

1) Obtain local media,
2) Set up a connection between the browser and the peer (other browser or endpoint),
3) Attach media and data channels to the connection, and
4) Exchange session descriptions

These four steps are shown in Figure 2.1. Note that this figure corresponds with Figure 1.4.

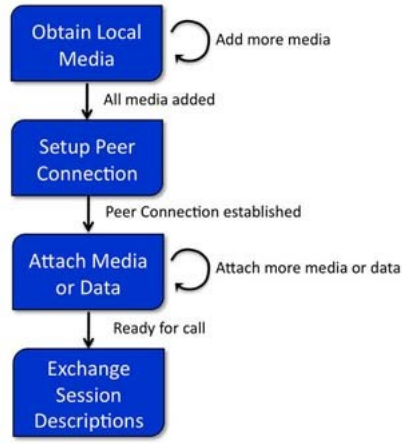


Figure 2.1 WebRTC Session Establishment, API View

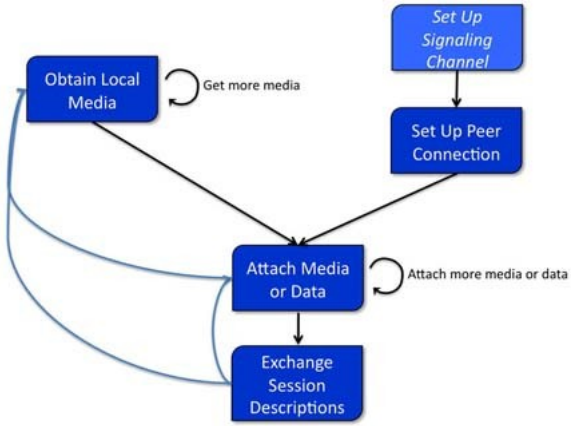


Figure 2.2 WebRTC API View with Signaling

Figure 2.2 shows another view of the steps, this time with signaling steps shown.

The following subsections briefly describe each of these steps, as well as the process for closing down a session when complete.

2.1.1 Obtaining Local Media

There are a variety of ways to obtain media, the complete list of which is out of scope for this book. However, one of the most common ways is defined by the WebRTC effort: *getUserMedia()*(Section 5.3.2). This method can be used to obtain a single local *MediaStream*. Once you have one or more media streams, you can piece them together into the streams you want using the *MediaStream* API (Application Programming Interface). For privacy reasons, a web application's request for access to a user's microphone or camera will only be granted after the browser has obtained permission from the user.

2.1.2 Setting up the Peer Connection

Another important step is to set up the Peer Connection using the API by the same name. The core of WebRTC is the *RTCPeerConnection* API, which, as its name suggests, sets up a connection between two Peers. In this context, "peers" means two communication endpoints on the World Wide Web, as in the phrase "peer-to-peer file sharing". Instead of requiring communication through a server, the communication is direct between the two entities. In the specific case of WebRTC, a Peer Connection is a direct media connection between two web browsers.

This is particularly relevant when a multi-way communication such as a conference call is set up among three or more browsers. Each pair of browsers will require a single Peer Connection to join them, allowing for audio and video media to flow directly between the two, as shown in Figure 1.10. Thus, three browsers communicating would need a total of three connections among them. An application developer will need to set up one Peer Connection per pair of browsers (or a browser and another endpoint such as an existing communications network) being connected. The alternative architecture of Figure 1.11 is also possible.

To establish this connection requires a new *RTCPeerConnection* object. The only input to the *RTCPeerConnection* constructor method is a configuration object containing the information that ICE, Interactive Connectivity Establishment (Section 6.2.7), will use to "punch holes" through intervening Network Address Translation (NAT) devices and firewalls.

2.1.3 Exchanging Media or Data

Once the connection is set up, any number of local media streams may be attached to the Peer Connection for sending across the connection to the remote browser. Similarly, any number of remote media streams may also be sent to the local end of the connection, resulting in new media streams at the local end that may be manipulated just like any other local media stream.

It is important to note that every change in media requires a negotiation (or renegotiation) between browsers of how media will be represented on the channel. When a request is made, locally or remotely, to add or remove media, the browser can be asked to generate an appropriate *RTCSessionDescription* object (a container for a session description – information about how to establish the media session) to represent the complete set of media flowing over the Peer Connection. The *RTCPeerConnection* API provides a means by which the application author can view and edit (if desired) the session description before it is sent to the remote side. This design allows the browser to handle the "heavy lifting" of proposing codecs and writing Session Description Protocol (SDP), Section 6.2.4, used to represent the session description, while still allowing for minor adjustments by the application as needed. However, it is expected that in the majority of cases, the web developer will not need to modify or inspect *RTCSessionDescription* objects.

Once the browsers have exchanged *RTCSessionDescription* objects, the media or data session can be established. Both browsers begin hole punching. Once hole punching completes, key negotiation for the secure media session can begin. Finally, the media or data session can begin.

Note that all of this activity, everything after the *RTCSessionDescription* objects have been exchanged, is done by the browser on behalf of the JavaScript code. The application JavaScript code may add or remove STUN and TURN servers, Sections 6.2.5 and 6.2.6, used for NAT traversal and monitor the process, but the work is done by the browser.

2.1.4 Closing the Connection

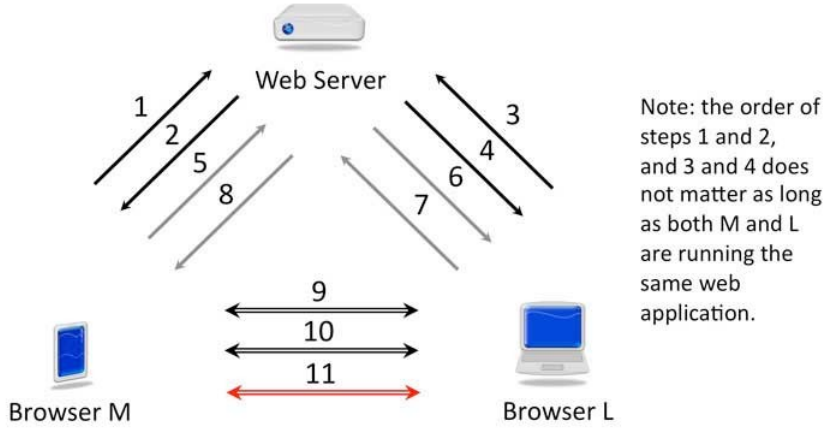
Either browser can close the connection. The application calls *close()*on the *RTCPeerConnection* object to indicate that it is finished using the connection, perhaps in response to a user clicking a button or closing a tab. This causes ICE processing and media streaming to stop. Similarly, should one browser lose Internet connectivity, or crash, the keep-alives sent in the media or data channel will fail, and the other browser will attempt to restart hole punching, and when that fails close the session. Once the session is over, the browser removes any session-granted permissions to access the microphone and camera of the device, so a new session will require new permission(s) from the user.

2.2 WebRTC Example Implementations

The following sections will show some examples of WebRTC implementations. Figure 2.3 shows a protocol view of WebRTC Session Establishment.

The following figures provide more details of the protocol exchanges, using "ladder diagrams", sometimes called call flows.

Protocols such as SRTP, SDP, and ICE are introduced in Chapter 6.



1) Browser M requests web page from web

server
2) Web sever provides web pages to M with WebRTC JavaScript
3) Browser L requests web page from web server
4) Web sever provides web pages to L with WebRTC JavaScript
5) M decides to communicate with L, JavaScript on M causes M's session description object (offer) to be sent to the web server
6) Web server sends M's session description object to the JavaScript on L
7) JavaScript on L causes L's session description object (answer) to be sent to web server
8) Web server sends L's session description object to the JavaScript on M
9) M and L begin hole punching to determine the best way to reach the other browser
10) After hole punching completes, M and L begin key negotiation for secure media
11) M and L begin exchanging voice, video, or data

Figure 2.3 WebRTC Session Establishment, Protocol View

2.2.1 Session Establishment in WebRTC Triangle

Basic session establishment with WebRTC is shown in Figure 2.4. Note that this figure corresponds with Figure 2.3.

Browsers M and L are running the same WebRTC enabled JavaScript downloaded from the web server. When one user wishes to communicate with another user, this begins the media negotiation between the browsers, referred to as an offer/answer exchange. This offer answer exchange occurs between the two browsers over a signaling channel established between them. For the remainder of this chapter, the signaling channel is assumed to be over HTTP through the web server. For the details of how this signaling channel works, and alternative signaling channel designs, see Chapter 4.

Media negotiation is the way in which two parties in a communication session, such as two browsers, communicate and come to agreement on an acceptable media session. Offer/answer is an approach to media negotiation in which one party first sends to the other party what media types and capabilities it supports and would like to establish – this is known as the "offer". The other party then responds indicating

which of the offered media types and capabilities are supported and acceptable for this session – this is known as the "answer". This process can be repeated a number of times to setup and modify a session, for example to add new media streams or to change which streams are to be sent. A common question is why this back-and-forth process is needed, rather than, for example, each side merely stating what it intends to do. The primary reason is to ensure that both sides have agreement before media begins flowing. This is crucial for the lower layers in the browser that may not be able to handle media that comes in before the browser is ready for it.

In addition to offers and answers, there is also a "provisional answer" (pranswer). A provisional answer is an answer to an offer, but it is provisional or tentative. It may not be the final answer given in the actual answer, which comes later in the offer/answer exchange. Provisional answers are optional, and in general will only occur when interoperating with the PSTN or some VoIP systems that emulate the "early media" characteristics of the telephone network.

When the user on browser M decides to communicate with the user on browser L, the JavaScript on browser M provides a constraint-based description of the media it wants, requests the media, and gets user permission. It is important that the permission grant is tied to the domain of the web page, and that this permission does not extend to pop-ups and other frames on the web page. The desired media session information is captured in a session description object. This is the offer, which is sent to browser L through the web server. It is important to note that WebRTC does not standardize how browser M sends this offer to browser L. There are a number of ways in which this could be accomplished, such as XML HTTP Request [XHR]. Browser L receives the session description object offer and generates a session description object answer, which is sent back to browser M using the same method. Once the offer/answer exchange is complete, hole punching (and key negotiation) can begin, and eventually the exchange of media packets.

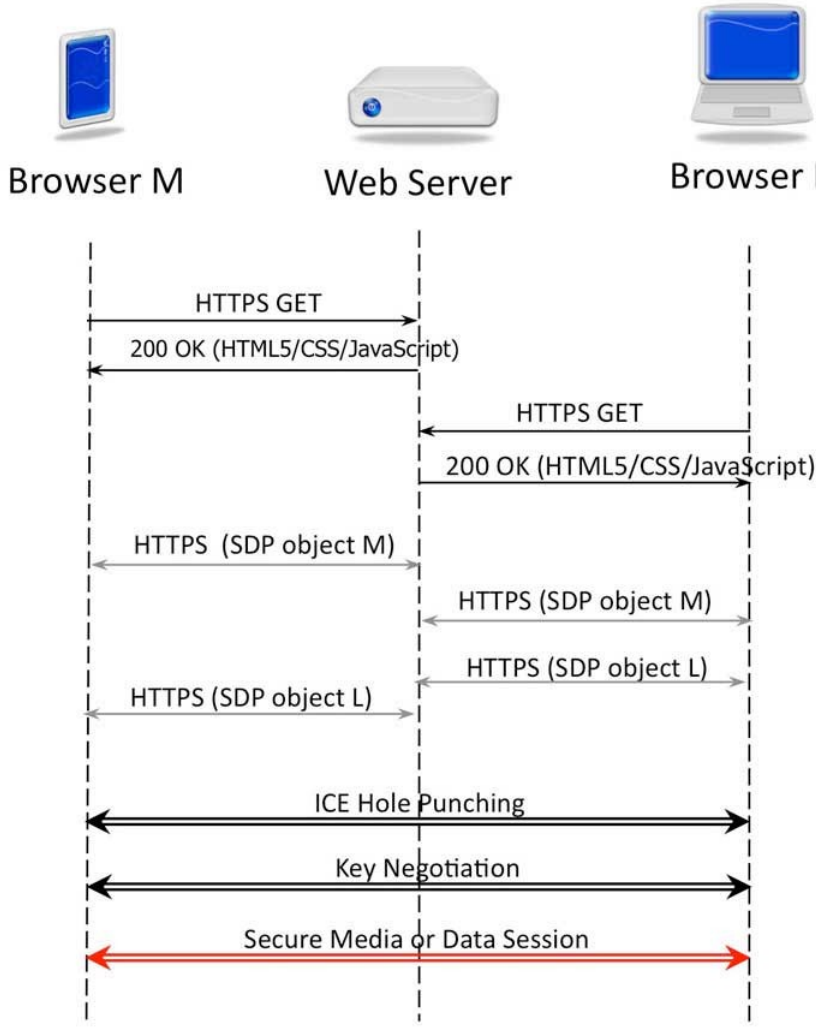


Figure 2.4 WebRTC Triangle Call Flow

Browser L closes the connection, which causes the Peer Connection to be closed and all permissions for microphone and cameras.

2.2.2 Session Establishment in WebRTC Trapezoid

The call flow for the WebRTC Trapezoid of Figure 1.5 is shown in Figure 2.5 below.

In this scenario, browsers M and L exchange media directly, despite running web applications from different web servers. The session

description objects from each browser are mapped to a Jingle [XEP-0166] session-initiate message and session-accept method. Currently, the defined mapping of Jingle to and from Session Description Protocol [XEP-0167] does not include the WebRTC extensions and attributes of SDP, but it could be extended to do so.

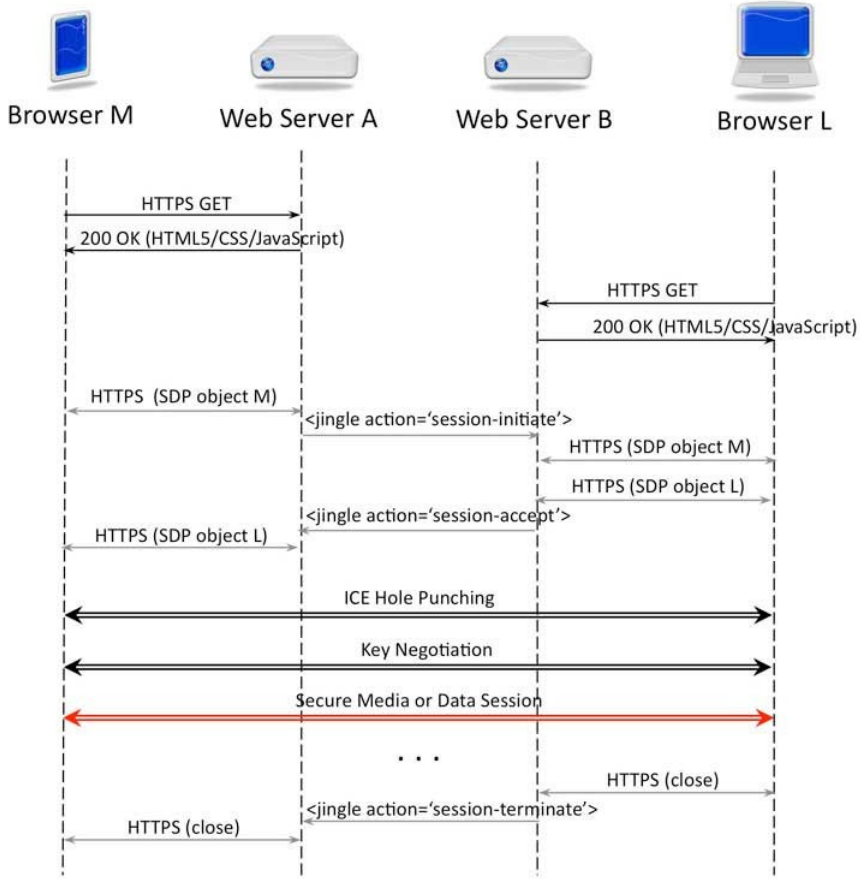


Figure 2.5 WebRTC Trapezoid Call Flow with Jingle

Note that this scenario could alternatively have mapped the session description objects to SDP and used SIP signaling between the browsers. No changes to the SIP protocol are needed to do this.

2.2.3 WebRTC Session Establishment with SIP Endpoint

WebRTC interworking with SIP [RFC3261] is shown in detail in Figure 2.6 below. This corresponds with Figure 1.4. WebRTC defines the mapping of the session description object offers and answers to SIP, which can be carried without modification or extension in normal SIP

INVITE and *200 OK* messages.

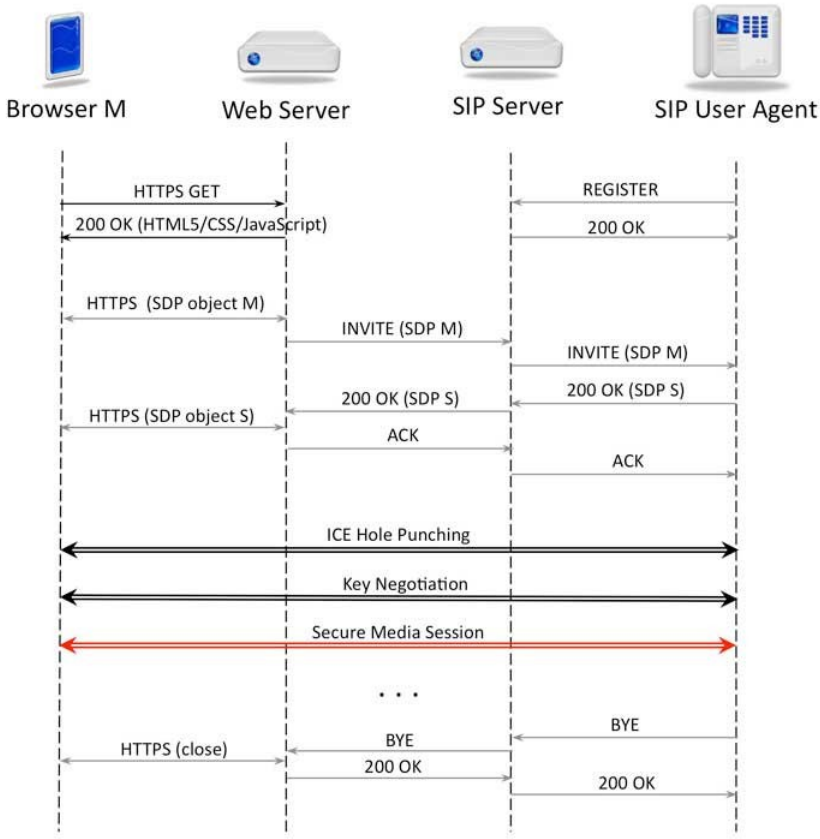


Figure 2.6 WebRTC Interoperating with SIP Call Flow

2.2.4 WebRTC Session Establishment with Jingle Endpoint

Figure 2.7 shows how WebRTC can interoperate with Jingle. This corresponds to Figure 1.4 and works in a similar way to the mapping described in Section 2.2.2. The media can be directly between the browser and the Jingle client provided the Jingle protocol is extended to support WebRTC SDP extensions, and the Jingle client supports the WebRTC media extensions.

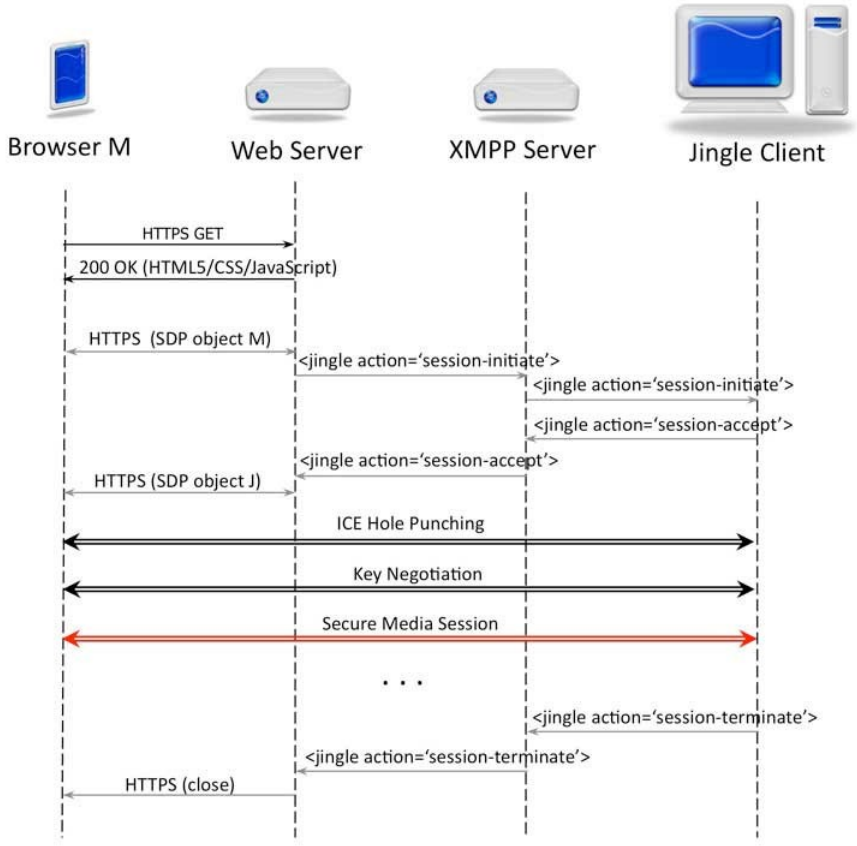


Figure 2.7 WebRTC Interoperating with Jingle

2.2.5 WebRTC Session Establishment with PSTN

Figure 2.8 shows how WebRTC can interoperate with the PSTN. This corresponds to Figure 1.7. The signaling between the Web Server and the PSTN Gateway could utilize any number of signaling or control protocols, or even a SIP trunk [SIP-CONNECT]. The PSTN Gateway terminates the WebRTC media session and connects the audio to a PSTN trunk or line. The G.711 (PCM) codec should be negotiated between the browser and the PSTN Gateway to avoid having to transcode or convert the audio signal.

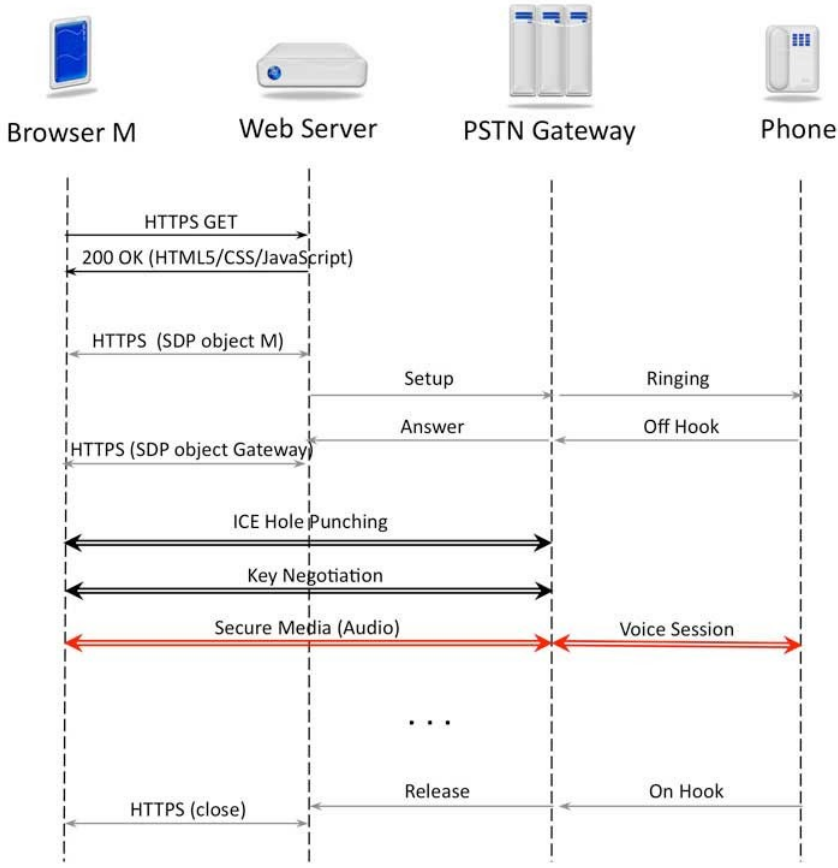


Figure 2.8 WebRTC Session Establishment with the PSTN

2.2.6 WebRTC Session Establishment with SIP and Media Gateway

A slightly different way for WebRTC to interwork with SIP is shown in Figures 2.9 and 2.10. In this example, the media is no longer completely end-to-end. Instead, a Media Gateway is used to terminate the ICE and SRTP, and the media is forwarded to the SIP UA, perhaps even as unencrypted RTP, although this should only be done if the VoIP network utilizes some other security protocol such as IPsec [IPSEC]. This is how VoIP and video endpoints that do not support all the RTP extensions of WebRTC could interwork with WebRTC.

The Media Gateway could also be a border element, known as a Session Border Controller (SBC), used to enable firewall traversal at the edge of a enterprise or service provider network

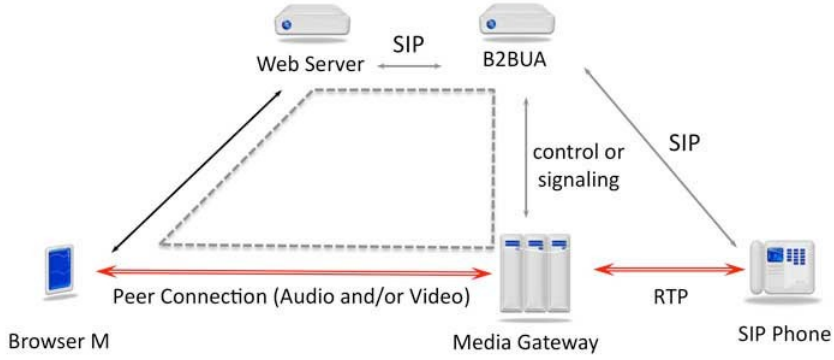


Figure 2.9 WebRTC Alternative SIP Interworking

2.3 WebRTC Pseudo-Code Example

This section contains a simple example corresponding to the media flows in Figures 1.9 and 5.3 through 5.6. The sample code has been made to look much like real code as one might see when using the WebRTC API; however, to keep the control flow simple to follow it is NOT real code. Additionally, in some places in the code we provide only comments rather than actual code, either because that code is irrelevant and out of scope for WebRTC, or because it has not been defined yet. For that reason we refer to this as pseudo code. You should in particular be aware of the following: no HTML code is shown, although the outputs (display, speakers, etc.) are assumed to have been set up; the media constraints shown have not been defined, although constraints similar to them are under discussion; real code would have error handling, but this has none; and finally, real WebRTC JavaScript code makes use of asynchronous callbacks for many function calls – as a result, pseudo code shown here executing sequentially would not execute properly as JavaScript without waiting for each individual callback to complete. The complete running-code example shown in Chapter 7 does not have these limitations. With these caveats, the remaining text of this section describes the examples.

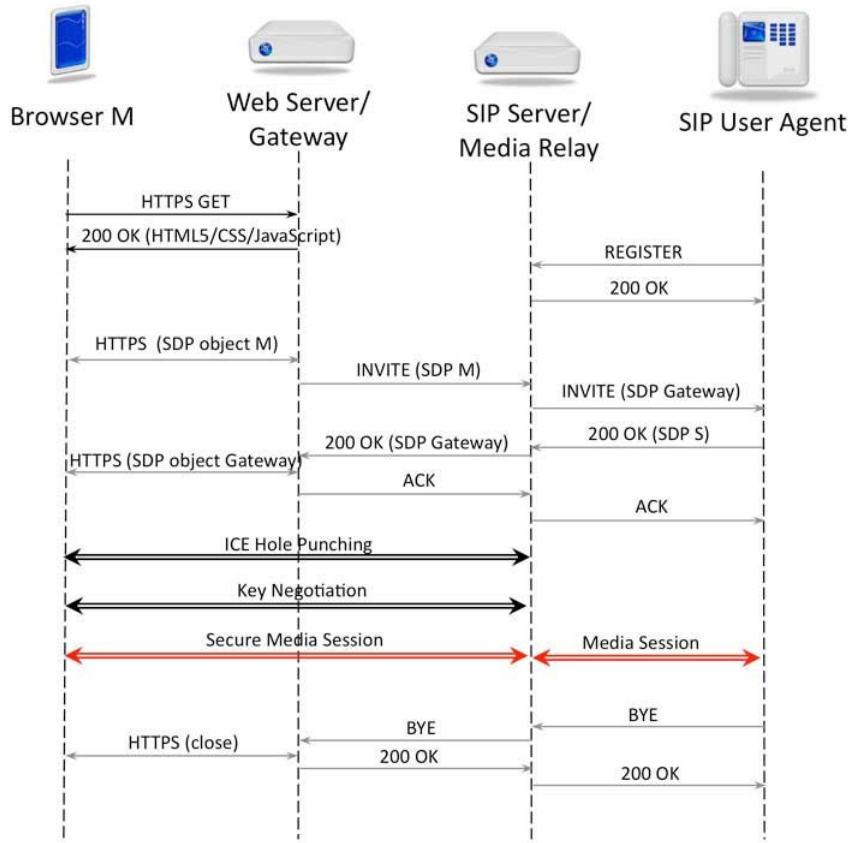


Figure 2.10 Alternative WebRTC Interworking with Media Gateway

The WebRTC specification contains an example in section 7 [EXAMPLE] showing code that is intended to run on both the caller and called sites. For easier understanding, to show different media streams in the two directions, and to show different arrangements of when the local media is obtained, the following example is shown in two separate code segments. The first segment is the code running on the mobile device, and it is hard-coded to place a call (as opposed to receiving one). It obtains local media first, then sets up the connection with the remote peer (laptop). It then creates new media streams from the local ones, attaches them to the new *RTCPeerConnection*, and sends the ids of the media streams to the remote peer so it can distinguish among them. Finally, it starts the call by creating an SDP offer, notifying the browser of the session description via *setLocalDescription(),* and sending it to the remote peer.

The second segment is the code running on the laptop, and it is hard-coded to receive a call (as opposed to placing one). It is similar in many respects. Unlike the first code segment, though, all of the real work of obtaining local media, attaching streams to the peer connection, and creating and sending an SDP answer all take place only after a message is received on the signaling channel. Until then it waits.

Note in this code snippet that the local media is not obtained until *after* the message is received on the signaling channel. If obtaining the user's permission to access the local devices is a slow process, a real-world application might want to obtain the local media in advance of waiting for a call. Notice also that the call to *answer()*is not done until we have actually received the offer from the caller. Finally, the code to process incoming messages on the signaling channel needs to expect the stream ids from the caller.

2.3.1 Pseudo Code for Mobile Browser

```
//////////////////
//  THIS IS PSEUDO CODE.  Yes, it looks just like real code.
//  DON'T BE FOOLED.
//  It is pseudo code because the APIs are still changing.
//  Don't expect this to run anywhere!
//////////////////

var pc;
var configuration =
  {"iceServers":[{"url":"stun:198.51.100.9"},
                 {"url":"turn:198.51.100.2",
                  "credential":"myPassword"}]};
var microphone, application, front, rear;
var presentation, presenter, demonstration;
var remote_av, stereo, mono;
var display, left, right;

var signalingChannel = createSignalingChannel();
/////////////////////////////////////////////////////////////////
//  Step zero is to set up the signaling channel.  There is no
//  requirement on how this is done.  The identity of the peer
//  is determined during the setup of the signaling channel.
//  As a result, the RTCPeerConnection itself does not
//  have any configuration info indicating the identity of
//  the peer. The preliminary code snippets below assume that
//  this signaling channel has a send() method and an onmessage
//  handler.  The former sends its argument to the peer, where
//  it causes the onmessage handler to execute.
/////////////////////////////////////////////////////////////////

////////////////////////////////////
// These are the four main calls
////////////////////////////////////

// First, obtain local media
getMedia();

// Next, create the peer connection
createPC();

// Attach media to the peer connection
```

```
attachMedia();

// Generate and send SDP offer to peer
call();

////////////////////
// Below this point are the function and handler definitions
////////////////////

// Get local media
function getMedia() {
   // get local audio (microphone)
   navigator.getUserMedia({"audio": true }, function (stream) {
     microphone = stream;
   });

   // get local video (application sharing)
   ///// This is outside the scope of this specification.
   ///// Assume that 'application' has been set to this stream.
   //

   // get local video (front camera)
   // note that "videoFacingModeEnum" has not yet been defined as
   // a constraint
   constraint =
              {"video":  {"mandatory":  {"videoFacingModeEnum":
"front"}}};
   navigator.getUserMedia(constraint, function (stream) {
     front = stream;
   });

   // get local video (rear camera)
   // note that "videoFacingModeEnum" has not yet been defined as
   // a constraint
   constraint =
     {"video": {"mandatory": {"videoFacingModeEnum": "rear"}}};
   navigator.getUserMedia(constraint, function (stream) {
     rear = stream;
   });
}

// Create a Peer Connection and set callbacks
```

```
function createPC() {
   pc = new RTCPeerConnection(configuration);

   // send any ice candidates to the other peer
   pc.onicecandidate = function (evt) {
      signalingChannel.send(
       JSON.stringify({ "candidate": evt.candidate }));
   };

   // process addition of remote streams
   pc.onaddstream =
    function (evt) {handleIncomingStream(evt.stream);};
}

// attach media to PC
function attachMedia() {
   // Create streams to send
   presentation =
    new MediaStream(
     [microphone.getAudioTracks()[0],    // Audio
      application.getVideoTracks()[0]]); // Presentation
   presenter =
    new MediaStream(
     [microphone.getAudioTracks()[0],    // Audio
      front.getVideoTracks()[0]]);       // Presenter
   demonstration =
    new MediaStream(
     [microphone.getAudioTracks()[0],    // Audio
      rear.getVideoTracks()[0]]);        // Demonstration

   // Add streams to Peer Connection
   pc.addStream(presentation);
   pc.addStream(presenter);
   pc.addStream(demonstration);

   // Send stream ids to remote peer as a JSON string
   // **before** SDP negotiation, i.e., before media
   // begins flowing.
   signalingChannel.send(
    JSON.stringify({ "presentation": presentation.id,
                     "presenter": presenter.id,
                     "demonstration": demonstration.id
```

```
    }));
}

// initiate a call by creating and sending an SDP offer
function call() {
   // Note at this point that we have not yet begun the media
   // offer/answer process, so no media is flowing.

   // Create an SDP offer based on the current set of streams
   // and ICE candidates.  gotDescription() will be called with
   // this offer.
   pc.createOffer(gotDescription);

   // This function acts based on an SDP offer just
   // created by the browser.
   function gotDescription(desc) {
      // First, tell the browser that this SDP offer is my
      // local session description.
      pc.setLocalDescription(desc);

      // Send the offer to the peer as a JSON string
      signalingChannel.send(JSON.stringify({ "sdp": desc }));
   }
}

// do something with remote streams as they appear
function handleIncomingStream(s) {
  // save handles for all incoming streams.  For the
  // av_stream, present it.
  if (s.getVideoTracks().length == 1) {
    // then this must be the av_stream
    av_stream = s;
    show_av(av_stream);
  } else if (s.getAudioTracks().length == 2) {
    // then this must be the stereo stream
    stereo = s;
  } else {
    // must be the mono stream
    mono = s;
  }
}
```

```
// display/play streams by attaching them to elements
function show_av(s) {
  // display is a video element, while left and right are audio
  // elements
  display.src = URL.createObjectURL(
                new MediaStream(s.getVideoTracks()[0]));
  left.src = URL.createObjectURL(
                new MediaStream(s.getAudioTracks()[0]));
  right.src = URL.createObjectURL(
              new MediaStream(s.getAudioTracks()[1]));
}

// handle incoming messages from the peer.  They will either be
// SDP or they will be ICE candidates.
signalingChannel.onmessage = function (msg) {
    // first parse the JSON event data back into an object
    var signal = JSON.parse(msg.data);

    if (signal.sdp) {
       // If this is SDP from the peer, tell the browser it is
       // the remote's session description.
       pc.setRemoteDescription(
         new RTCSessionDescription(signal.sdp));
    } else {
       // If not, this must be a candidate from the peer.  Tell
       // the browser that this is a candidate IP address
       // through which the media could possibly reach the
       //  peer.  The browser will then use ICE to try to reach
       // this address.
       pc.addIceCandidate(
         new RTCIceCandidate(signal.candidate));
       }
};
```

2.3.2 Pseudo Code for Laptop Browser

```
//////////////////
//  THIS IS PSEUDO CODE.  Yes, it looks just like real code.
//  DON'T BE FOOLED.
//  It is pseudo code because the APIs are still changing.
//  Don't expect this to run anywhere!
//////////////////

var pc;
var configuration =
  {"iceServers":[{"url":"stun:198.51.100.9"},
                 {"url":"turn:198.51.100.2",
                  "credential":"myPassword"}]};
var webcam, left, right;
var av, stereo, mono;
var incoming;
var speaker, win1, win2, win3;

var signalingChannel = createSignalingChannel();
//////////////////////////////////////////////////////////////////
//  Step zero is to set up the signaling channel.  There is no
//  requirement on how this is done.  The identity of the peer
//  is determined during the setup of the signaling channel.
//  As a result, the RTCPeerConnection itself does not
//  have any configuration info indicating the identity of
//  the peer. The preliminary code snippets below assume that
//  this signaling channel has a send() method and an onmessage
//  handler.  The former sends its argument to the peer, where
//  it causes the onmessage handler to execute.
//////////////////////////////////////////////////////////////////

// At this end we basically just wait for an SDP offer to come
// our way before we set up the peer connection, obtain local
// media, and attach it.

// There isn't really anything to answer.  This just creates a
// PC and the handlers to deal with incoming streams.  It also
// sets up media.  It is called by the signaling channel's
// onmessage handler.
function prepareForIncomingCall() {
   // First, create the Peer Connection
   createPC();
```

```
    // Next, obtain local media
    getMedia();

    // Attach media to the peer connection
    attachMedia();
}

////////////////////
// Below this point are the function and handler definitions
////////////////////

// Create a Peer Connection and set callbacks
function createPC() {
    pc = new RTCPeerConnection(configuration);

    // send any ice candidates to the other peer
    pc.onicecandidate = function (evt) {
        signalingChannel.send(
          JSON.stringify({ "candidate": evt.candidate }));
    };

    // process addition of remote streams
    pc.onaddstream =
            function (evt) {handleIncomingStream(evt.stream);};
}

// Get local media
function getMedia() {

    // get local video (webcam)
    navigator.getUserMedia({"video": true }, function (stream) {
      webcam = stream;
    });

    // get local audio (left stereo channel)
    // note that "audioDirectionEnum" has not been defined as
    // a constraint
    constraint =
      {"audio": {"mandatory": {"audioDirectionEnum": "left"}}};
    navigator.getUserMedia(constraint, function (stream) {
      left = stream;
```

```
  });

  // get local audio (right stereo channel)
  // note that "audioDirectionEnum" has not been defined as a
  // constraint
  constraint =
   {"audio": {"mandatory": {"audioDirectionEnum": "right"}}};
  navigator.getUserMedia(constraint, function (stream) {
   right = stream;
  });
}

// attach media to PC
function attachMedia() {
  // Create streams to send
  av = new MediaStream(
       [webcam.getVideoTracks()[0],      // Video
        left.getAudioTracks()[0],        // Left audio
        right.getAudioTracks()[0]]);     // Right audio
  stereo = new MediaStream(
           [left.getAudioTracks()[0],    // Left audio
            right.getAudioTracks()[0]]); // Right audio

  mono = left;     // Treat the left audio as the mono stream

  // Add streams to Peer Connection
  pc.addStream(av);
  pc.addStream(stereo);
  pc.addStream(mono);

  // Note that here we don't need to inform the remote peer of
  // the stream ids because the streams are uniquely identified
  // by what they contain.
}

// answer the call by creating and sending an SDP answer
function answer() {
  // Note at this point that we have not begun the media
  // offer/answer process, so no media is flowing.

  // Create an SDP answer based on the remote session
  // description, which the browser already has, and
```

```
    // the current set of streams and ICE
    // candidates.  gotDescription() will be called with this
    // answer.
    pc.createAnswer(gotDescription);

        // This function acts based on an SDP answer just
    // created by the browser.
    function gotDescription(desc) {
        // First, tell the browser that this SDP answer is my
        // local session description.
        pc.setLocalDescription(desc);

        // Send the answer to the peer as a JSON string
        signalingChannel.send(JSON.stringify({ "sdp": desc }));
    }
}

// do something with remote streams as they appear
function handleIncomingStream(s) {
  // take audio from only one stream but display all video
  // tracks
  if (s.id === incoming.presentation) {
    // use audio and display presentation screen
    speaker.src = URL.createObjectURL(
                  new MediaStream(s.getAudioTracks()[0]));
    win1.src = URL.createObjectURL(
                new MediaStream(s.getVideoTracks()[0]));
  } else if (s.id === incoming.presenter) {
    // display presenter
    win2.src = URL.createObjectURL(
                new MediaStream(s.getVideoTracks()[0]));
  } else {
    // must be demonstration, so display it
    win3.src = URL.createObjectURL(
                new MediaStream(s.getVideoTracks()[0]));
  }
}

// handle incoming messages from the peer.  They will be
// SDP, ICE candidates, or a structure containing remote
// session ids.
signalingChannel.onmessage = function (msg) {
```

```
    // first create a peer connection to answer the call if not
    // already done
    if (!pc) {
      prepareForIncomingCall();
        }
    // now parse the JSON event data back into an object
    var signal = JSON.parse(msg.data);

    if (signal.sdp) {
        // If this is SDP from the peer, tell the browser it is
        // the remote's session description and send an SDP
        // answer.
        pc.setRemoteDescription(
          new RTCSessionDescription(signal.sdp));
        answer();
    } else if (signal.candidate) {
        // If this is a candidate from the peer, tell
        // the browser that this is a candidate IP address
        // through which the media could possibly reach the
        // peer.  The browser will then use ICE to try to reach
        // this address.
        pc.addIceCandidate(new RTCIceCandidate(signal.candidate));
        } else {
        // This must be an object containing the ids of the streams
        // the remote peer will be sending.  Save it for use when
        // the streams appear.
        incoming = signal;
        }
};
```

2.4 References

[XHR] http://www.w3.org/TR/XMLHttpRequest

[XEP-0166] http://xmpp.org/extensions/xep-0166.html

[XEP-0167] http://xmpp.org/extensions/xep-0167.html

[RFC3261] http://tools.ietf.org/html/rfc3261

[SIP-CONNECT] http://www.sipforum.org/sipconnect

[IPSEC] http://tools.ietf.org/html/rfc4301

[EXAMPLE]
http://dev.w3.org/2011/webrtc/editor/webrtc.html#examples-and-call-flows

WebRTC uses unique peer-to-peer media flows, where voice, video, and data connections are established directly between browsers. Unfortunately, Network Address Translation (NAT) and firewalls make this difficult and require special protocols and procedures to work. Peer-to-peer media is also sometimes referred to as end-to-end media.

3.1 WebRTC Media Flows

For the media flows between browsers discussed in this chapter, the four browsers in Figure 3.1 will be used as a reference to illustrate the concepts. The mobile and the tablet access the Internet through the home WiFi router. The laptop connects through a WiFi router at a coffee shop. The PC connects to the Internet through a corporate router.

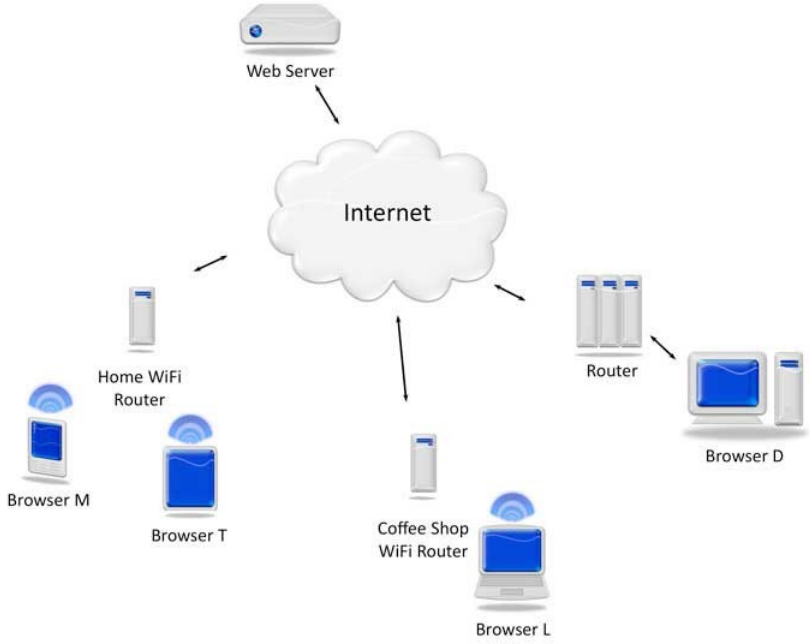


Figure 3.1 WebRTC Browsers Connecting to the Internet

3.1.1 Media Flows without WebRTC

Without WebRTC or a plugin, a browser could establish media flows. However, these media flows must follow the same path as the web browsing traffic. In other words, the media packets will flow from one browser to the web server, then to the other browser. This is shown in

Figure 3.2 below. The web server needs to handle the extra traffic as a result. High definition video streams can use considerable bandwidth. This limits the scalability of this architecture.

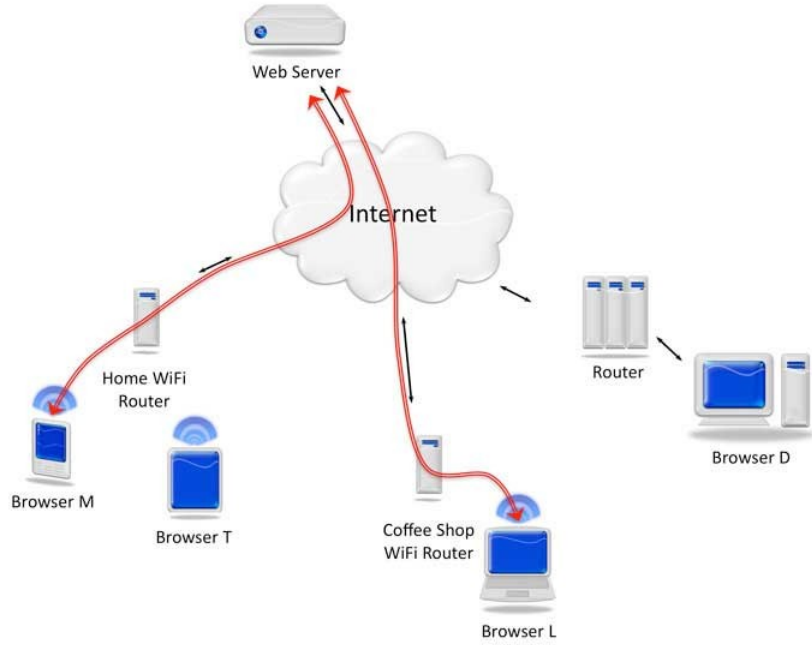


Figure 3.2 Media Flows without WebRTC

3.1.2 Media Flows with WebRTC

The goal of the *RTCPeerConnection* API in WebRTC is to enable the establishment of direct peer-to-peer media connections between browsers. This flow would look like Figure 3.3.

This path for the media can have few Internet hops, take less time (lower latency), and have a lower chance of packet loss. As a result, these types of peer-to-peer media flows can result in much better quality connections. It reduces the bandwidth used by the web server. It also makes the geographic proximity of the web server to the browsers a non-issue. For example, if the two browsers were located, for instance, in Japan, but the web server was located in Europe, the media flow of Figure 3.2 would be very problematic, but the peer-to-peer flow of Figure 3.3 would be much better. These peer-to-peer media flows dramatically reduce the costs of offering a real-time communications service.

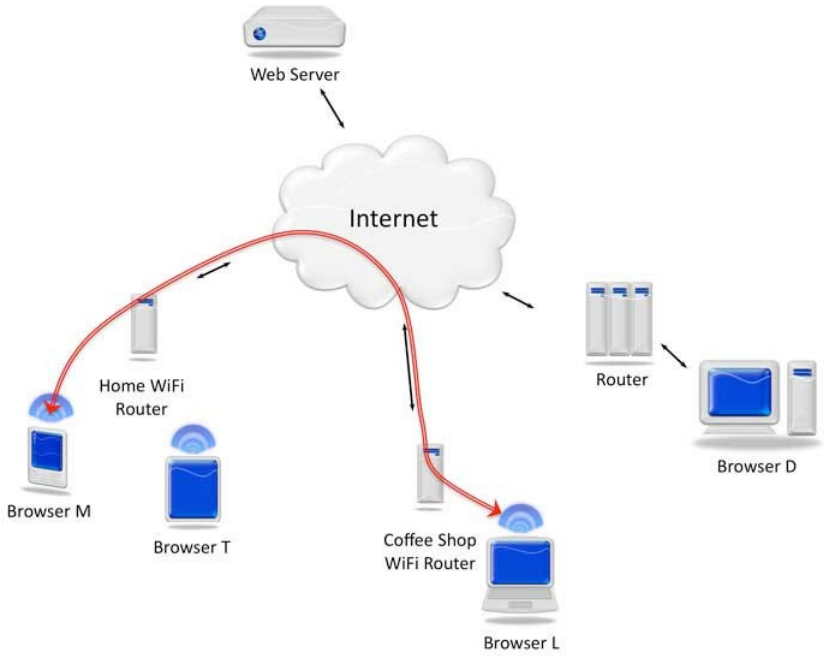


Figure 3.3 Peer-to-Peer Media Flow with WebRTC

However, establishing this media flow is actually quite complicated, as most Internet devices connect to the Internet through a Network Address Translation (NAT) function, as will be discussed in the next section.

3.2 WebRTC and Network Address Translation (NAT)

It is extremely common for browsers to be behind NATs – Network Address Translation devices. A more realistic connection of the browsers is shown in Figure 3.4 with every browser behind a NAT.

NAT is a function often built into Internet routers or hubs that map one IP address space to another space. Usually, NATs are used to allow a number of devices to share an IP address, such as in a residential router or hub. NATs are also used by enterprises or service providers to segment IP networks, simplifying control and administration. Each network behind a NAT is effectively an island, and hosts on that network rely completely on the NAT device providing access. Note that NATs are actually NAPTs – Network Address and Port Translators. Mostly, the term NAT is used even for devices that change transport port numbers in addition to transport addresses.

Many Internet protocols, especially those using a client/server

architecture (for instance normal web browsing, email, etc.), have no difficulty traversing NATs. However, end-to-end or peer-to-peer protocols and services can have major difficulties. Unfortunately, WebRTC is one such service.

In Figure 3.4, the laptop is connected to the Internet through a WiFi router that has built in NAT. The mobile browser and tablet browser connect to the Internet through a WiFi hub that has built in NAT and share a single IP address. The PC connects using an enterprise router with NAT.

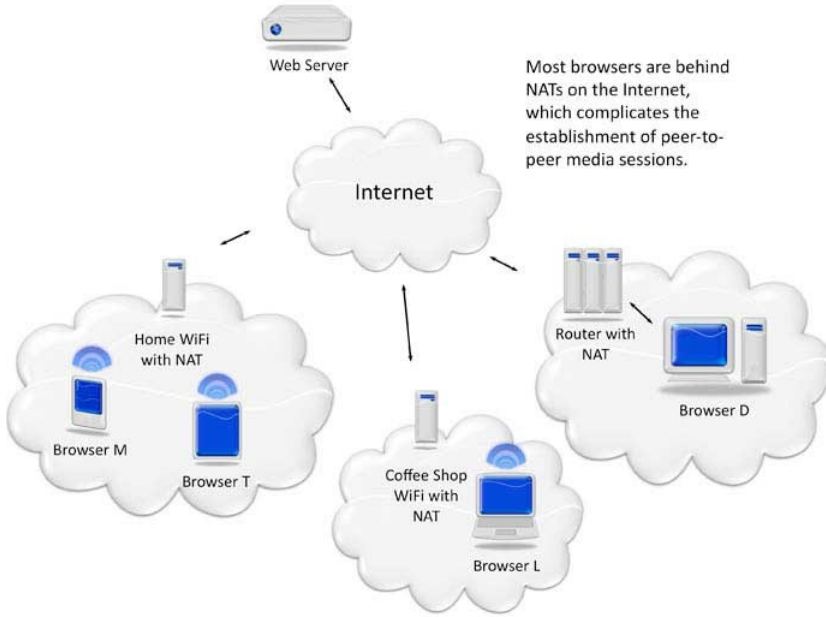


Figure 3.4 WebRTC Browsers Behind NAT

The next sections will discuss the types of media flows that can be established with WebRTC. Some are peer-to-peer across multiple NATs, peer-to-peer behind the same NAT, or relayed through dedicated TURN servers.

3.2.1 Peer-to-Peer Media Flow through Multiple NATs

Figure 3.5 shows a peer to peer media flow that can be established using WebRTC, using the hole punching techniques described in Section 3.3. The media flow can bypass the web server and flow directly between the two browsers, through the NATs.

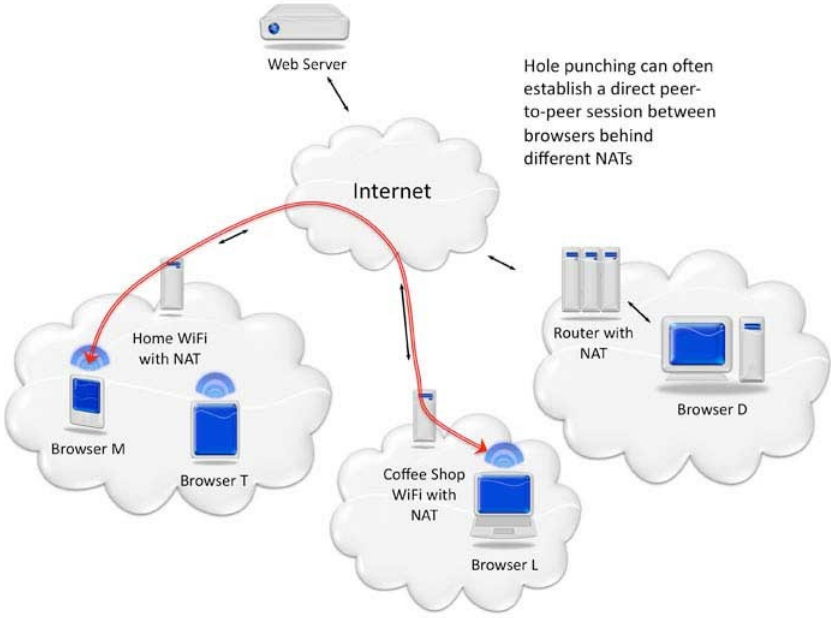


Figure 3.5 Peer-to-Peer Media Flow through NATs with WebRTC

3.2.2 Peer-to-Peer Media Flow through a Common NAT

Figure 3.6 shows the case where a media session is established between two browsers behind the same NAT. In this case, the

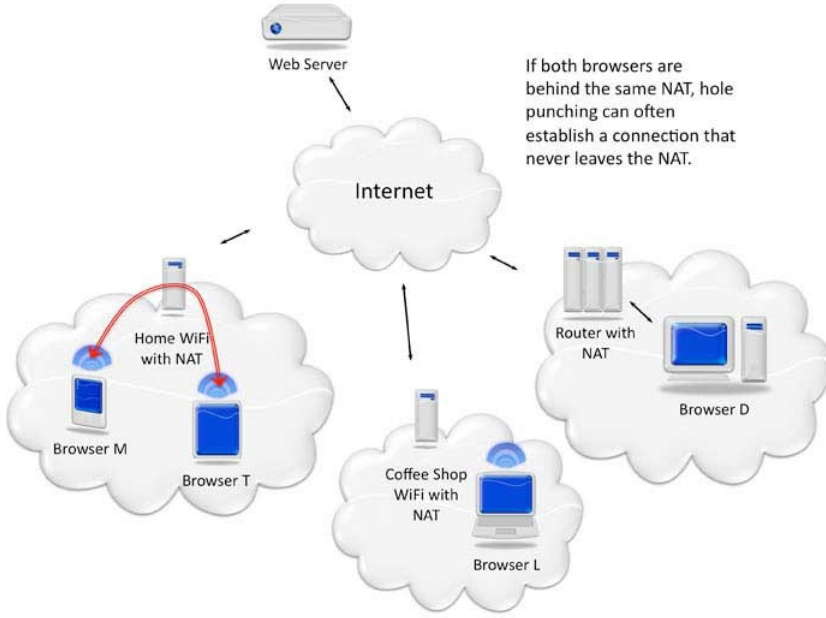


Figure 3.6 Media Flow when Browsers are Behind the Same NAT

optimal media path is to stay on the local area network and never go up to the Internet, as shown. This case also has very desirable quality, bandwidth, and security properties. As in the previous case, hole punching is needed to achieve this media flow.

3.2.3 Private and Public Addresses

Common NAT terminology uses the terms “private address” and “public address”. The IP addresses behind a NAT (or “inside” the NAT) are “private” IP addresses (used inside each of the NAT clouds in the figures). The IP address (or possibly multiple addresses) assigned to the NAT, and used whenever the NAT forwards packets from the inside to the outside, is the “public” IP addresses (used “outside” the NAT). This is shown for the home WiFi network in Figure 3.7.

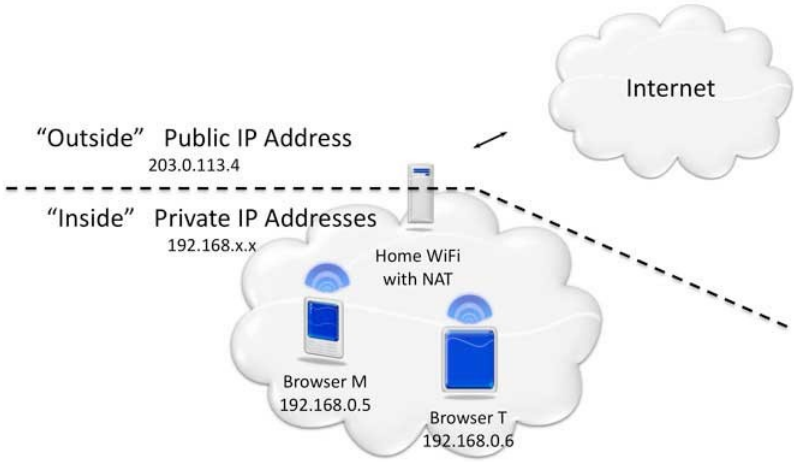


Figure 3.7 Public and Private IP Addresses and NAT

In this example, the home Wifi NAT has been assigned (by the Internet Service Provider of the house) the IP address 203.0.113.4 – this is the outside public IP address of the NAT, and all hosts that connect to the Internet through this NAT will share this IP address. The mobile device and the tablet each have an IP address that has been assigned by the NAT, which is a private IP address.

Note that these addresses are public in the same sense as we call the telephone network the Public Switched Telephone Network. Public IP addresses are usable (routable) anywhere on the Internet. Public IP addresses have to be unique on the Internet, and they are managed in central registries for the Internet and assigned by Internet Service Providers. They are analogous to a full mailing address including the country name, or a telephone number that includes the country code. Private IP addresses, on the other hand, do not have any special privacy features or capabilities. Rather, they are private as in private property – they are only valid within the network hosted by the NAT. There are specific IP address ranges (e.g. 192.168.x.x, 10.x.x.x, and 172.16.x.x-172.31.x.x) that anyone may use inside their own network. They do not need to be unique as they are only valid inside that network. They are analogous to a campus box or intra-company mail address, or a telephone number that is an extension, and only valid inside that building or campus.

A NAT maintains tables of mapping between inside IP (private) addresses and port numbers, and outside (public) IP address and port numbers. In addition to this mapping, NATs also maintain filter rules

about which IP addresses and port numbers in the public Internet are permitted to use the mappings that have been created. There are a number of different categories of NATs depending on the rules they apply in generating the NAT mappings and filter rules.

3.3 Introduction to Hole Punching

The nature of NAT makes establishing direct peer-to-peer sessions difficult. However, using a technique known as "hole punching" [BRYAN], it can be successfully done in many cases, perhaps as much as 85% of the time on average. However, this is an average of of many users across many networks. Certain networks will have a higher success rate, and others will typically have a much lower success rate. For example, mobile data networks in the U.S. reportedly have a lower success rate of about 30%.

There are a number of pre-requisites for hole punching. They are:

1) The two browsers trying to establish a direct connection must both send "hole punching" packets at the same time. As a result, they must both be aware of the to-be-established session and know the addresses to which to send the packets. Note that there is nothing special about a hole punching packet – it is an ordinary IP packet that is sent to test to see if a particular destination address is reachable through the NATs.
2) The two browsers need to know as many possible IP addresses as possible that could be used to reach them. These addresses are often described as "private" (or inside the NAT) addresses, "public" (or on the outside of the NAT) addresses, and relay addresses, depending on privacy settings (see Chapter 10.)
3) As a last resort, a media relay, which has a public IP address (is not behind a NAT) and hence is reachable by both browsers, is needed.
4) Symmetric flows must be used. That is, UDP traffic must appear to operate in a similar manner to a TCP connection.

Requirement 1) is met by using the Web Server to coordinate the hole punching. That is, the Web Server knows that a session is to be established between the browsers, so it ensures that both browsers begin hole punching at approximately the same time.

Requirement 2) is met by using a STUN (Session Traversal Utilities for NAT) Server, described in Section 6.2.5. Each browser queries the STUN Server by sending a STUN packet. The STUN Server responds

indicating the IP address that it observes in the test packet. That is, it responds with the mapped address from the NAT (actually the outermost NAT if there are multiple levels of NAT). This IP address learned from the STUN server is shared with the other browser and becomes a potential "candidate" address. The private IP address is obtained through the operating system from the network interface cards, NICs. These addresses could be IPv4 or IPv6, or a combination of both. This is shown in Figure 3.8. In addition, a Virtual Private Network or VPN connection could provide an additional address. Other NAT traversal protocols such as Universal Plug and Play (UPnP) could be used, although these are uncommon.

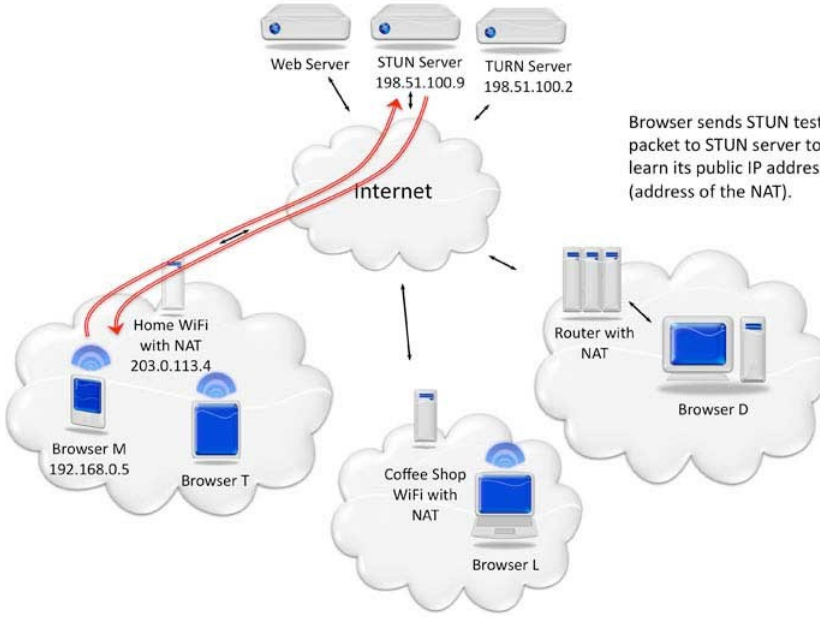


Figure 3.8 Browser use of STUN Server.

Requirement 3) is met using a TURN (Traversal Using Relay around NAT) Server, described in Section 6.2.6. Just as the browsers query the STUN Server prior to initiating hole punching, the browsers query the TURN Server to obtain a media relay address. The media relay address is a public IP address that will forward packets received to and from the browser that setup the relay address. This address is then added to the candidate list.

Requirement 4) is met by the browser sending media from the same UDP port that the browser is using to listen for incoming media. This makes the two one-way RTP sessions over UDP appear to the NAT to be

one bi-directional RTP session. Symmetric RTP is described in Section 9.3.2.

3.3.1 Relayed Media Through TURN Server

In most cases, the hole punching will result in a direct peer-to-peer connection being established. However, in certain cases of a very restrictive NAT or firewall, the direct paths will fail and the only one that succeeds will be the address of the TURN server. This will result it the media being relayed through the TURN server, as shown in Figure 3.9. Although this figure shows the TURN server as a separate server, a TURN server is actually a STUN server with added relaying functionality, and in many cases are combined. While all TURN servers also have STUN functionality, not every STUN server has TURN functionality.

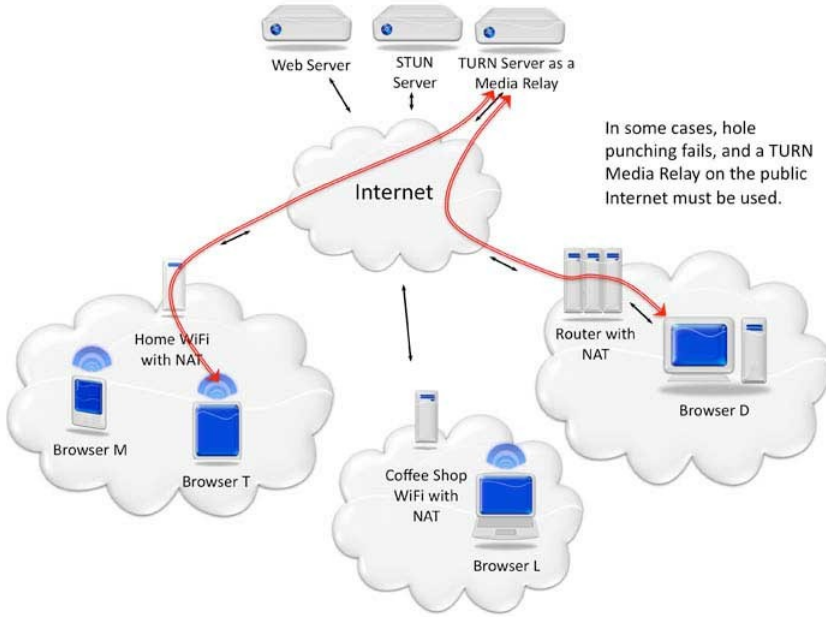


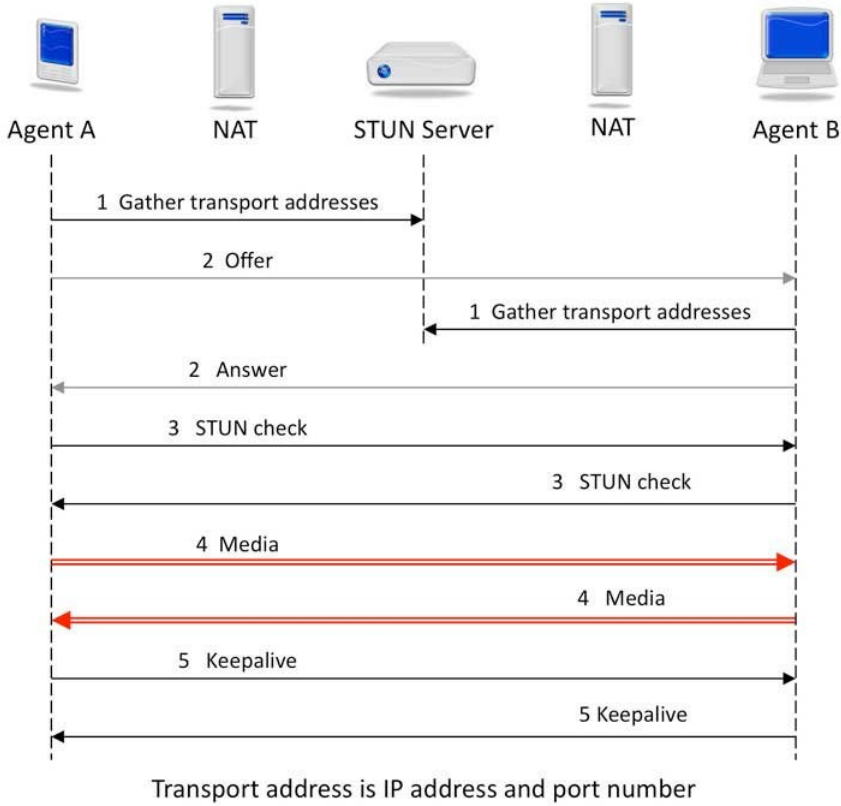
Figure 3.9 Media Relayed Through TURN Server

While this media flow is not ideal, at least the media is not being relayed through the Web Server as it was in Figure 3.2. Also, this is only occurring in a limited number of scenarios because there is no alternative – all direct media paths have failed.

The protocol used to implement hole punching, known as ICE, Interactive Connectivity Establishment, is discussed in the next section.

3.4 Interactive Connectivity Establishment

Interactive Connectivity Establishment, or ICE, is a standardized protocol for hole punching. It uses STUN and TURN to help endpoints establish connectivity. The basic steps in ICE are shown in Figure 3.10.



1) Gather Candidate Transport Addresses
2) Exchange Candidates over Signaling Channel
3) Perform Connectivity Checks
4) Choose Selected Pair and Begin Media
5) Send Keepalives

If either side detects a change in IP address in use,
ICE is restarted (back to Step 1)

Figure 3.10 High Level ICE Call Flow

The following sections will cover these steps.

3.4.1 Gather Candidate Transport Addresses

The first step is to gather candidate transport addresses. Candidates addresses are an IP address and port where media might be able to be received for a Peer Connection. These addresses must be gathered at the time of the call – they cannot be gathered ahead of time in many cases. In the example of Figure 3.10, ICE Agent A begins gathering candidate addresses as soon as the user at A initiates a Peer Connection with B. ICE Agent B begins gathering candidate addresses as soon as the Peer Connection request from A is received in the signaling channel.

There are four types of address candidates, shown in Table 3.1. A host candidate address is an address obtained through the operating system and represents an actual address on a network interface card (NIC). If the ICE Agent is behind a NAT, this address will be a private IP address and not be routable outside the subnet. The next two candidate addresses are known as reflexive addresses since they represent addresses that are reflected back to the ICE Agent by a STUN check, as if the client is looking in a mirror through the STUN server to learn their actual IP addresses. A server reflexive candidate is an address learned from a response to a STUN check sent to a STUN server. If the ICE Agent is behind a NAT, this address will be the outside address of the outermost NAT. That is, there could be multiple layers of NAT between the ICE Agent and the STUN server, but this check only allows discovery of the last NAT before the STUN server.

Candidate Type	Use
Host	Local transport address obtained from the network interface card (NIC). If behind a NAT, this will be a private address
Server Reflexive	Transport address obtained by a STUN check sent to a STUN server. If behind a NAT, this will be the public IP address of the outermost NAT.
Peer Reflexive	Transport address obtained from a STUN connectivity check sent by the other ICE Agent (peer). This is a new candidate which is discovered during the connectivity checks, not sent over the signaling channel
Relayed	Transport address of a media relay server. Usually obtained using a TURN allocation request.

Table 3.1 ICE Candidate Address Types

A peer reflexive candidate is an address learned from a received STUN check sent by the other ICE Agent (a peer). This type of candidate address is not exchanged over the signaling channel but is discovered during the STUN connectivity checks of Step 3.

A relayed candidate is an address of a media relay. Usually this is obtained using the TURN protocol. The transport address obtained using a TURN allocate request is a relayed candidate.

Browsers are configured with the STUN and TURN servers used in this gathering candidates step. This is done using STUN URIs [draft-nandakumar-rtcweb-stun-uri] and TURN URIs [draft-petithuguenin-behave-turn-uris] in the ICE Servers object. Note that access to a TURN server means having STUN server functionality as well.

It is important to note that just learning public IP address candidates using STUN is not enough on its own to traverse the NATs. NATs are complicated and vary widely in operation between networks and service providers. As a result, the full functionality of ICE is needed to ensure NAT traversal.

3.4.2 Exchange of Candidates

The second step is the exchange of candidate addresses over the signaling channel. Candidates are exchanged between the browsers over

the signaling channel, as described in Chapter 4. The candidates are first ordered or prioritized. In general, the highest priority are host candidates, followed by reflexive addresses, followed lastly by relayed candidates. If there is a preference between IPv4 and IPv6, this can be expressed by different priority settings. Candidates are associated with a particular media stream in SDP. The default behavior with WebRTC is to multiplex all media, including voice, video, and data, over the same transport address. As such, a single set of candidates is all that is needed.

3.4.3 STUN Connectivity Checks

The ICE Agents begin connectivity checks as soon as they have sent and received the candidates. In figure 3.10, for Agent A, this is when the SDP answer is received from Agent B. For Agent B, this is when the SDP answer is sent to Agent A. During this phase, the ICE Agents generate STUN responses to any STUN connectivity requests they receive from their peer that pass authentication.

The first step is to pair candidates based on IP address type (IPv4 or IPv6) and other factors. The purpose of pairing and generating foundations for each pair is to reduce the number of connectivity checks performed to minimize the time needed to obtain a working candidate.

Peer reflexive candidate addresses can be discovered during this step and are automatically paired as they are discovered. There are five possible states of connectivity states, as shown in Figure 3.11.

Queued candidate pairs start in the “frozen” state (a joke on the name of the protocol) – a holding state until the checks are ready to be performed. When the ICE connectivity check algorithm determines that a check should be performed, it is “unfrozen” and moves to the “waiting” state. A pair could stay in the waiting state due to pacing considerations for the checks, so that a flood of packets is not sent at once. When the pacing allows for the check to be made, the state moves to “in-progress” when the STUN connectivity check is sent to the other peer. If a response comes back, the state moves to “succeeded”, while if the check times out without a response, it moves to “failed”.

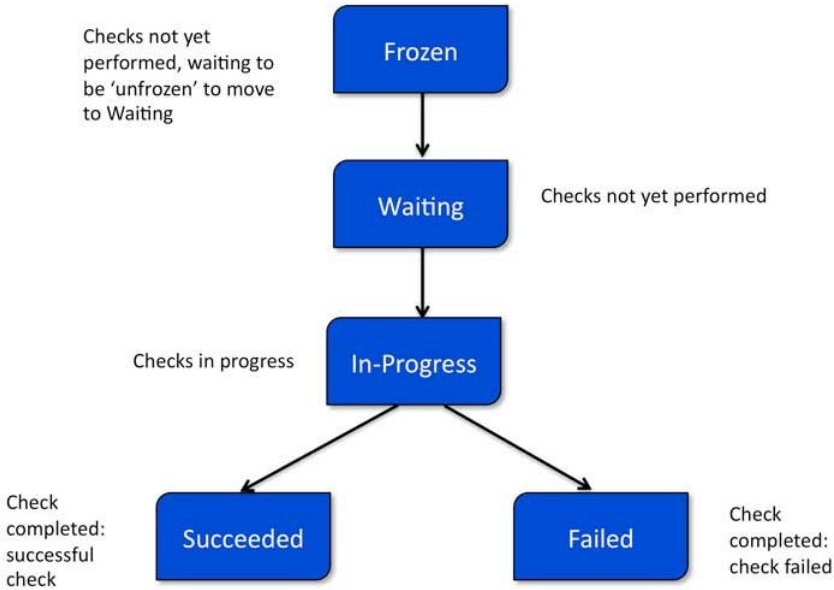


Figure 3.11 ICE Connectivity Check State Machine

There is an optimization of ICE known as Trickle ICE (see Section 8.5.1) where instead of all of the candidates being provided at the start of ICE processing, ICE is started with a minimal set, and additional candidates are added, or trickled in, as processing continues. These new candidates are paired and queued, and go through the same steps as Figure 3.11.

3.4.4 Choose Selected Pair and Begin Media

The connectivity checks continue until either all possible checks have completed (all have moved from the "frozen" state to either "succeeded" or "failed") or one pair has been chosen. Choosing a pair is done by the controlling ICE Agent. The ICE protocol has an algorithm to choose which browser is the Controlling ICE Agent and which is the Controlled ICE Agent. The Controlled ICE Agent learns that the other ICE Agent has chosen a candidate pair when a STUN connectivity check is received with an attribute indicating that this pair is to be used. The Controlled ICE Agent then replies to the connectivity check echoing that the pair will be used. Media is now sent by both browsers using the chosen candidate pair.

3.4.5 Keep-Alives

To ensure that NAT mappings and filter rules do not time out during the media session, ICE continues to send STUN connectivity checks at 15 second intervals over the candidate pairs in use. This ensures that packets are sent, even when media is on hold or otherwise not being sent. If the media session is still active, the other ICE Agent generates a STUN response. The receipt of this STUN response by the other ICE Agent is taken as an indication that media can continue to be sent. If the STUN response is not received, ICE restarts, per the next section. Note that this behavior is not defined in the original ICE protocol specification, but is defined in the ICE extension of Section 8.5.5

3.4.6 ICE Restart

An ICE restart is triggered if either ICE Agent detects a change in the base transport address. Recall that the base address the transport address which was used to generate the candidate pair which is in use. This will cause the ICE Agent to go back to Step 1, gather candidates and send those candidates in an SDP offer to the other ICE Agent. That will cause the other ICE Agent to also go back to Step 1 and the whole process will repeat.

Note that this will occur if a browser page involved in an active Peer Connection is reloaded by a user. This is sometimes described as "rehydration," and is an active area of discussion in the standards on how to best handle this situation.

3.5 WebRTC and Firewalls

Firewalls, named after the fire-proof walls used to stop the spread of fires or heat in buildings and cars, are also frequently found in the Internet. They are used at a network boundary to enforce security policies. They can implement any type of policy and do any type of IP packet filtering or blocking. Most commonly, firewalls implement simple packet filter rules similar to those used in NAT. Firewalls act as a "one way gate", allowing access from "inside" the network to the "outside" (the Internet), but blocking arbitrary traffic from the Internet from entering the network. Essentially, firewalls try to allow normal IP traffic originating from inside the network, but block malicious traffic from the Internet.

One type of policy is to allow only outgoing traffic, with the exception of packets from the outside that are determined to be valid responses to requests from the inside. For example, a firewall might allow TCP connections if they are opened from a host inside the network. Once the connection is open, packets can flow in either direction until the connection is closed. TCP connections from the

outside will be blocked. Handling of UDP traffic is more difficult since the UDP packets are not part of a connection, and there is no signaling indicating the start and stop of a UDP flow. Outgoing UDP packets might also be allowed, and certain incoming UDP packets if they are destined for a host that recently sent out a UDP packet. Note that this firewall behavior is very similar to the filter rules used by most NATs to restrict the usage of mappings between public and private IP addresses and ports. Some firewalls block all UDP traffic, with the exception of DNS lookups. In these circumstances, running media over TCP might be the only way to establish the media flow.

Firewalls are often combined with NAT and implemented in the same box, although they are separate functions. Most home routers and enterprise routers have firewall functionality built in. It is becoming increasingly common for PCs themselves to have firewall functionality, since IP access is handled by the operating system.

3.5.1 WebRTC Firewall Traversal

Hole punching techniques used to traverse NATs often also work for traversal of firewalls. However, some firewalls have stricter rules that cause hole punching to fail. Some firewalls even block all UDP traffic entirely. (For these cases, there have been some discussions of standardizing the transport of SRTP media over TCP or even HTTP.)

There are a number of different approaches used to help today's VoIP and video traffic traverse firewalls. One approach is to build awareness of the VoIP signaling and media protocols into the firewall, so that it can open "pin holes" for the media only for the duration of the media session. This approach is sometimes known as the Application Layer Gateway or ALG approach. This approach is not usable with WebRTC since there is no standardized signaling protocol used – the signaling is just part of the exchange over HTTP between the browser and the web server.

Another approach uses a special-purpose VoIP and video firewall which is trusted by the firewall. These elements are commonly known as Session Border Controllers or SBCs [RFC5853]. They terminate the VoIP and video signaling and media traffic and apply policy. The SBC is connected to the firewall using a trusted link known as a DMZ (for De-Militarized Zone). Again, this approach will not work unless WebRTC is gatewayed to a standard signaling protocol such as SIP.

One approach that could be used is to utilize a media relay which is trusted by the firewall. The media relay would also be connected via the DMZ and would be responsible for authenticating the flows. A TURN Server provides this functionality in a way that is compatible with the ICE hole punching used in WebRTC. Essentially, an enterprise that

wanted to be able to control and monitor WebRTC media flows would implement firewall policies that would cause all the hole punching candidates with the exception of the enterprise TURN server to fail. As a result, all flows would be authenticated and relayed by the TURN server. The TURN server could be configured in a web browser in the same way (and for the same reason) as web proxies are configured today.

Alternatively, a firewall could implement an ICE ALG. This would allow the firewall to use the ICE hole punching as signaling for the UDP flows to come. The pinhole could be kept open as long as the ICE keep-alive packets continue.

Figure 3.12 shows media through a firewall using a TURN Server.

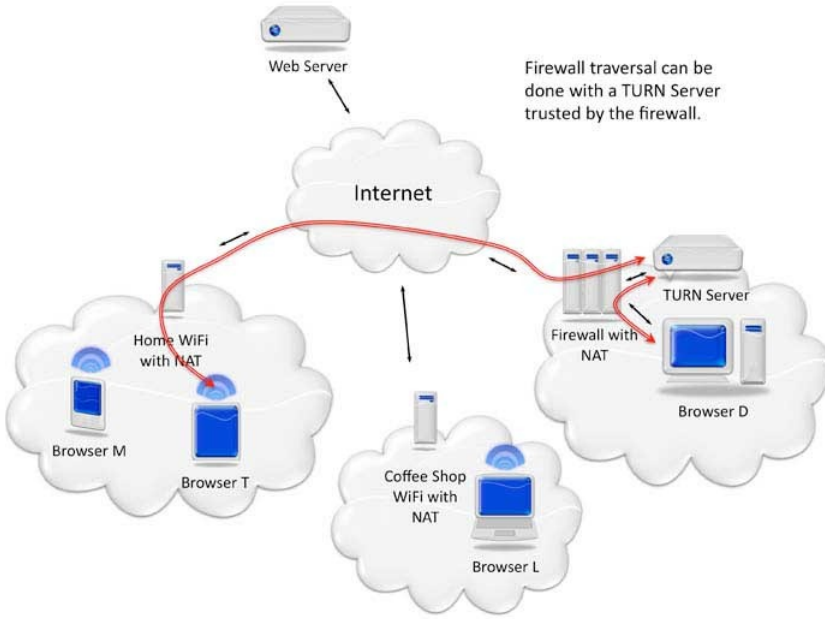


Figure 3.12 Media Relayed through Firewall Traversal TURN Server

For more information about media traversal through firewalls, see [IEEE-COMS] and [draft-hutton-rtcweb-nat-firewall-considerations].

3.6 References

[BRYAN] http://www.brynosaurus.com/pub/net/p2pnat

[draft-nandakumar-rtcweb-stun-uri] http://tools.ietf.org/html/draft-nandakumar-rtcweb-stun-uri

[draft-petithuguenin-behave-turn-uris] http://tools.ietf.org/html/draft-petithuguenin-behave-turn-uris

[RFC5853] http://tools.ietf.org/html/rfc5853

[IEEE-COMS] Alan Johnston, John Yoakum and Kundan Singh, Taking on WebRTC in an Enterprise, IEEE Communications Magazine, Vol. 51, No. 4, April 2013

[RFC 2460] http://tools.ietf.org/html/rfc2460

[draft-hutton-rtcweb-nat-firewall-considerations] http://tools.ietf.org/html/draft-hutton-rtcweb-nat-firewall-considerations

4 WEBRTC SIGNALING

Signaling plays an important role in WebRTC but is not standardized, allowing the developer to choose. This lack of standardization and multiple options has resulted in some confusion. A number of different signaling approaches have been proposed and used, and an understanding of the differences between the approaches is useful in selecting the right one for a given WebRTC application.

4.1 The Role of Signaling

In real-time communications, signaling has four main roles:

1) Negotiation of media capabilities and settings
2) Identification and authentication of participants in a session
3) Controlling the media session, indicating progress, changing and terminating the session
4) Glare resolution, when both sides of a session try to establish or change a session at the same time.

The next sections will examine these functions in detail, showing how 1) is essential and standardized, while 2), 3), and 4) are optional or are just part of the web application in WebRTC.

4.1.1 Why Signaling is Not Standardized

Signaling is not standardized in WebRTC because it does not need to be standardized to enable interoperability between browsers. Signaling is effectively a matter between the web browser and the web server. The web server can ensure that both browsers utilize the same signaling protocol, using downloaded JavaScript code.

In the web model, only the minimum components are standardized, leaving web developers the freedom to choose and design all other aspects of web pages and applications. In practice, this means that only transport (HTTP), markup (HTML), and media (WebRTC) need to be standardized. As shown in Figure 4.1, the server selects the signaling protocol and ensures that users of the web application or site support the same protocol. Web servers A, B, and C do not need to use the same signaling protocol, but in each case the browsers are able to establish media sessions.

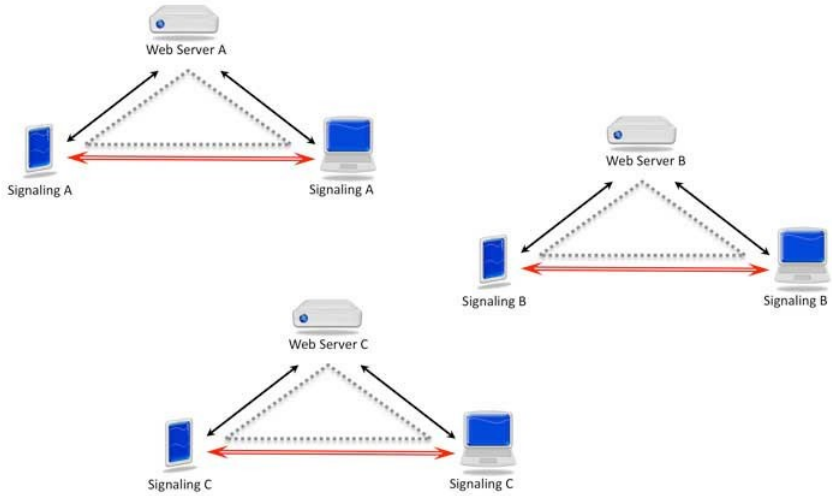


Figure 4.1 Web Server Chooses Signaling Protocol

Compare this situation to the general VoIP or video system where there is no way for a signaling or control server to push signaling code into the end devices. As a result, the only way interoperability can be achieved is for both endpoints to use the same standardized signaling protocol, such as SIP or Jingle, which are introduced in Section 4.3.6 and 4.3.7.

For a federated or trapezoid architecture, such as that shown in Figure 1.5, both web domains need to agree on a signaling protocol in order to interoperate. However, this signaling protocol does not necessarily need to be the same signaling protocol used in each of the browsers. In other words, just because two web domains use SIP to communicate, this doesn't mean that SIP must be used in both browsers.

4.1.2 Media Negotiation

WebRTC specifications includes requirements for the "signaling channel." The most important function of signaling is the exchange of information contained in the Session Description Protocol (SDP) objects between the browsers involved in the Peer Connection. SDP as used by JSEP contains all the information necessary for the RTP media stack on the browser to configure the media session, including the types of media (audio, video, data), codecs used (Opus, G.711, etc), any parameters or settings for the codecs, and information about the bandwidth. Also, the signaling channel is used to exchange candidate addresses for ICE hole

punching. The candidate addresses represent the IP addresses and UDP ports where potentially media packets could be received by the browser. Candidates can also be sent and received outside of SDP in the signaling channel. Keying material for SRTP must also be exchanged in the signaling channel.

ICE hole punching, described in Section 3.4, cannot begin until the candidate addresses have been exchanged over the signaling channel, so without this signaling function, there can be no establishment of a Peer Connection.

4.1.3 Identification and Authentication

When a standard signaling protocol such as SIP or Jingle is used to initiate real-time communication, the signaling channel provides the identity of the participants and also optionally authentication. In WebRTC, there are two non-signaling sources of identity. One is the context provided by the web application. For instance, a user to a WebRTC website might sign on with a particular screen name. When this user wishes to establish a session with another user, the web application presents the screen name to the other user as the identity. A user can only trust the website that this identity is accurate. This is very similar to caller identity in the PSTN (Public Switched Telephone Network). A PSTN user must trust their service provider that the caller ID presented to their telephone is in fact the caller – they have no other way of independently determining it. Or, an identification might be passed in a URL, which could contain a random token. The parties in a WebRTC session established this way would be parties who knew the identification token.

WebRTC defines an alternative approach of identity through the media channel which does not rely on trusting information from the website. The notion of media path identity was first proposed by the ZRTP [RFC6189] media path keying protocol, in which caller identity and authentication was provided in the media path without relying on the signaling channel at all. WebRTC uses DTLS-SRTP [RFC5763] to provide media path identity. This is done using the fingerprint of the public key used during the DTLS handshake. This fingerprint can be authenticated by the use of an Identity Provider, described in Section 10.4. The signaling channel is used to transport the fingerprint and the identity assertion but is not otherwise involved in the generation or validation of the identity assertion.

4.1.4 Controlling the Media Session

A conventional multimedia signaling protocol, such as SIP or Jingle, or some proprietary protocol, can provide call control of the session. The proprietary protocol could be extremely simple, such as the example in the demo application of Chapter 7. However, in WebRTC, while signaling is required to initiate or change a media session, signaling is not needed to indicate status or to terminate a session. Instead, the ICE state machine in the browser can provide this information. For example, as candidate addresses are being checked, this can provide progress information about the session. Once a session is established, if ICE continuing consent checks fail, this is an indication that the session has been terminated.

4.1.5 Glare Resolution

Signaling protocols such as SIP have built-in glare resolution. Glare is when both sides of a communication session attempt to setup or change a session at the same time. It is a race condition that could result in an nondeterministic state for the session. Some of the approaches for using SDP eliminate many glare conditions, and if those approach are incorporated in WebRTC, the requirements on glare could be greatly reduced. For example, if adding a new media source to a session can be done without a new offer/answer exchange, then this common source of glare can be eliminated.

4.2 Signaling Transport

WebRTC requires a bi-directional signaling channel between the two browsers. First, the transport for signaling messages will be discussed. Three transports are commonly used for WebRTC signaling: HTTP, WebSockets, and the data channel.

4.2.1 HTTP Transport

HTTP can also be used to transport WebRTC signaling. A browser can initiate new HTTP requests to send and receive signaling information from a server. The information can be transported using a *GET* or *POST* method, or in responses. If the server used for signaling supports Cross-Origin Resource Sharing (CORS), Section 10.2.3.2, the signaling server can be at a different IP address than the web server.

Sending information to the server is straightforward, using an XML HTTP Request (XHR) API call in JavaScript. In the other direction, receiving information asynchronously from the server is trickier, and a number of techniques have been developed over the years, known as AJAX (Asynchronous JavaScript And XML) [AJAX]. Some of these

approaches are as simple as polling or keeping a GET request continuously open, ready to receive data from the server, such as signaling approach used in the demo application code of Chapter 7. XHR is often used indirectly via JavaScript libraries such as jQuery [JQUERY]. The demo application of Chapter 5 uses HTTP transport for the signaling channel.

The use of HTTP transport is shown in Figure 4.2. Note that the use of HTTP for signaling is often referred to as REST (Representational State Transfer) or RESTful signaling.

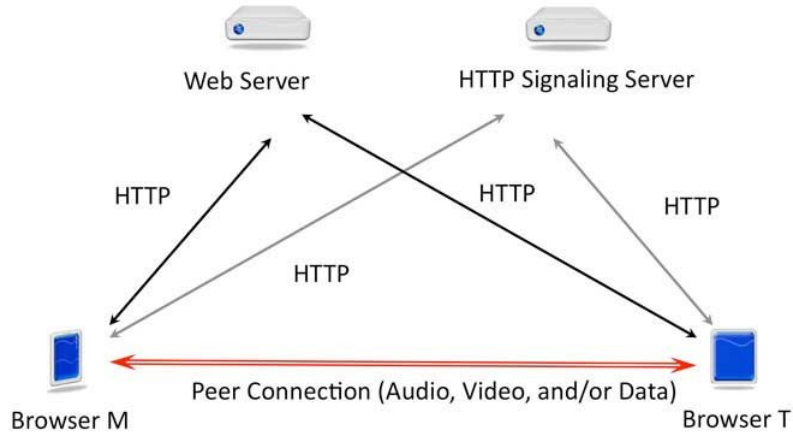


Figure 4.2 HTTP Transport for Signaling

HTTPS is more secure than HTTP, and allows a browser to authenticate the signaling server, as discussed in Section 10.2.1.

4.2.2 WebSocket Transport

WebSocket transport, as introduced in Section 6.2.2, allows a browser to open a bi-directional connection to a server. The connection begins as an HTTP request but then upgrades to a WebSocket. As long as the server supports CORS, the WebSocket server can be at a different IP address than the web server, as shown in Figure 4.3.

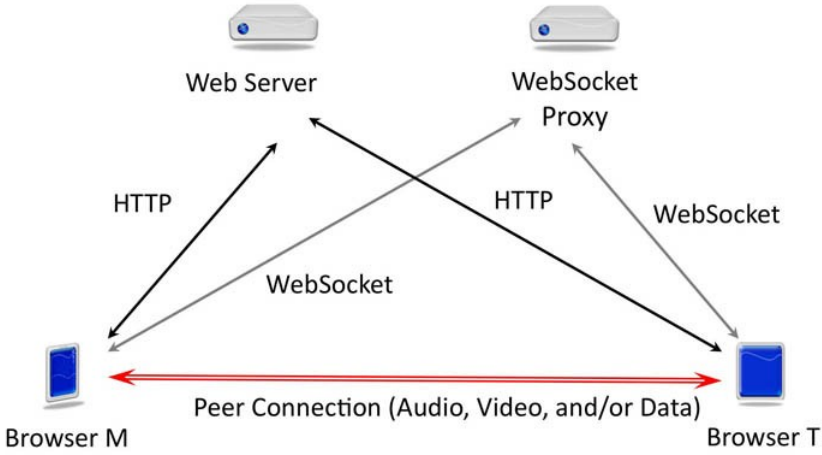


Figure 4.3 Signaling Transport using WebSocket

In order for a WebSocket server to be reachable, it must have a public IP address and be running an HTTP server. Note that this means it is not possible to open a WebSocket directly with another browser since browsers implement HTTP user agent functionality and not HTTP server functionality, so the WebSocket server is still needed to relay between two web clients using WebSockets. Even if a computer ran an HTTP server, NAT traversal would prevent end-to-end WebSocket connections between computers on the Internet. A browser uses WebSockets by utilizing the WebSocket API [WS-API].

WebSockets are supported by all major browser vendors. However, some web proxies and firewalls do not fully support WebSockets and can cause problems, especially with authentication.

4.2.3 Data Channel Transport

The data channel, once established between two browsers, provides a direct, low latency connection which makes it suitable for signaling transport. Since the initial establishment of a data channel requires a separate signaling mechanism, the data channel alone cannot be used for all WebRTC signaling. However, it can be used to handle all signaling after it is set up, including all the signaling for the audio and video media over the Peer Connection. This is shown in Figure 4.4 where data channel signaling goes over HTTP to the web server but all other signaling goes over the data channel.

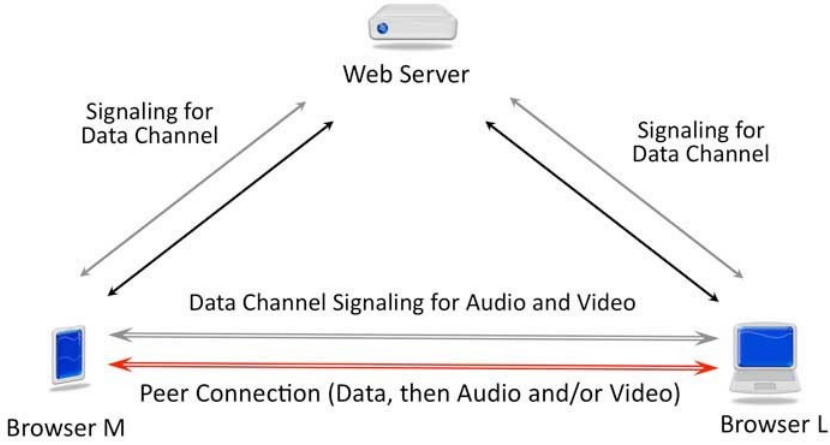


Figure 4.4 Data Channel Signaling Transport

The signaling load on the server to establish just the data channel is much less than handling all signaling for voice and video. The signaling for the data channel could also use a separate server using WebSockets, for example, as shown in Figure 4.5.

An interesting benefit of data channel signaling transport is that it can be used to reduce connection establishment time perceived by users. Before media can be exchanged between browsers, ICE and DTLS-SRTP key management steps must complete. If these steps are started only after the called user accepts the session (i.e. "answers the call"), there can be a perceived delay of perhaps up to a few seconds. With data channel signaling, ICE and DTLS-SRTP key management is performed in order to establish the data channel, and can happen before the user is alerted or asked to accept a session. When the user accepts the session, adding voice and video can be done comparatively quickly, as the signaling messages go directly to the other browser, and the media streams reuse the Peer Connection so no ICE or DTLS-SRTP processing is performed.

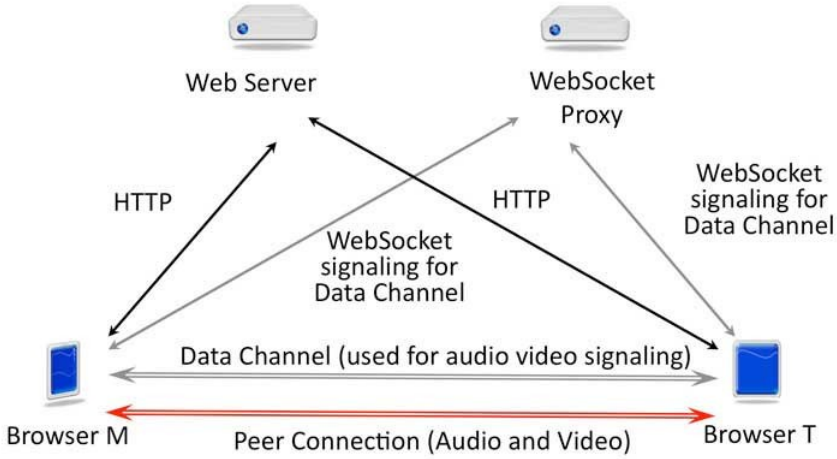


Figure 4.5 Data Channel Signaling Transport with Separate Server

4.3 Signaling Protocol

The choice of signaling protocol for WebRTC is an important one, and not necessarily tied to the choice of signaling transport, discussed in the previous section. A developer may choose to create their own proprietary signaling protocol, use a standard signaling protocol such as SIP or Jingle, or use a library which abstracts away the details of the signaling protocol.

4.3.1 Signaling State Machine

Regardless of the signaling protocol, there will be some sort of "state machine", which could be as simple as Figure 4.6.

Note that not every state in the signaling state machine requires a signaling message be exchanged between the browsers, although some signaling approaches will result in this.

An advantage of using a standard signaling protocol is that the state machine is already fully defined. Also, the signaling message will contain information useful for routing.

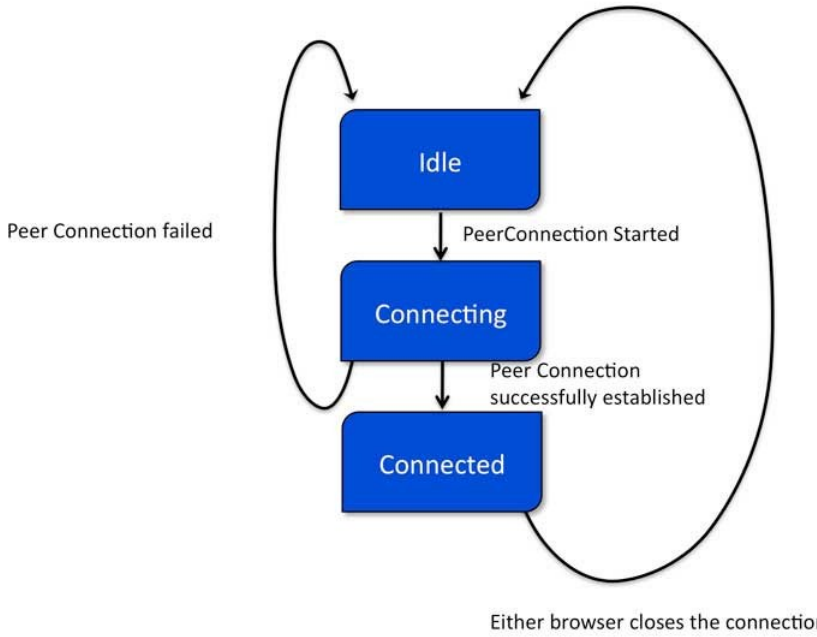


Figure 4.6 A Simple Signaling State Machine.

An advantage of using a proprietary approach is that it can be very simple and only provide the features needed by the application. If the WebRTC Peer Connection is always just between two browsers, and not through an intermediary or to a SIP or Jingle VoIP or video endpoint, this can be a good choice.

An alternative non-signaling approach to identity is to identify the connection rather than the users. For example, a connection identifier could be included in the URL, and knowledge of that identifier could be the only identity and authentication needed to participate in a session.

The following sections will discuss in more detail these different signaling protocol options.

4.3.2 Signaling Identity

In order to implement 2) of Section 4.1, some kind of routing logic will need to be in the server. If a given connection between a browser and the server can be identified by a token, then the signaling message sent to the server could contain the token of the other server to browser connection. The web server code will then proxy or forward the information between the two connections. Two examples of this approach are to use a WebSocket Proxy or use of the Google App Engine Channel API. If a

standard signaling protocol is used, the message will contain the identifier of the other browser (user) as the destination. A standard server for the particular signaling protocol can provide this functionality with minimal configuration. For example, if SIP signaling is used, a SIP Proxy Server will perform this function. If Jingle is used, an XMPP Server provides this function.

4.3.3 HTTP Polling

A simple proprietary signaling approach is to use HTTP polling. An example of this is the XHR-based signaling channel used in the Demo Code of Chapter 7.

XML HTTP Request (XHR) calls within JavaScript or jQuery allows a JavaScript application to generate a new HTTP request and process HTTP responses to a web server. XHR is a W3C standard API being defined as a Working Draft [XHR] on the way to becoming a standard. Various parts of the XHR JavaScript API functionality are supported in browsers. Despite the name, XHR can be used to send more than XML requests: JSON (JavaScript Object Notation) or plain text can also be sent as well. XHR causes the browser to generate a new HTTP or HTTPS request, such as a *GET*, *POST*, *PUT*, etc. The API call specifies the method to be used and the IP address and port number. The response to the request is returned to the JavaScript.

To use XHR as a signaling channel for WebRTC, the web server needs to run an application which receives the HTTP Request and proxies or forwards the information received from one browser to the other browser over another XHR channel, as shown in Figure 4.7.

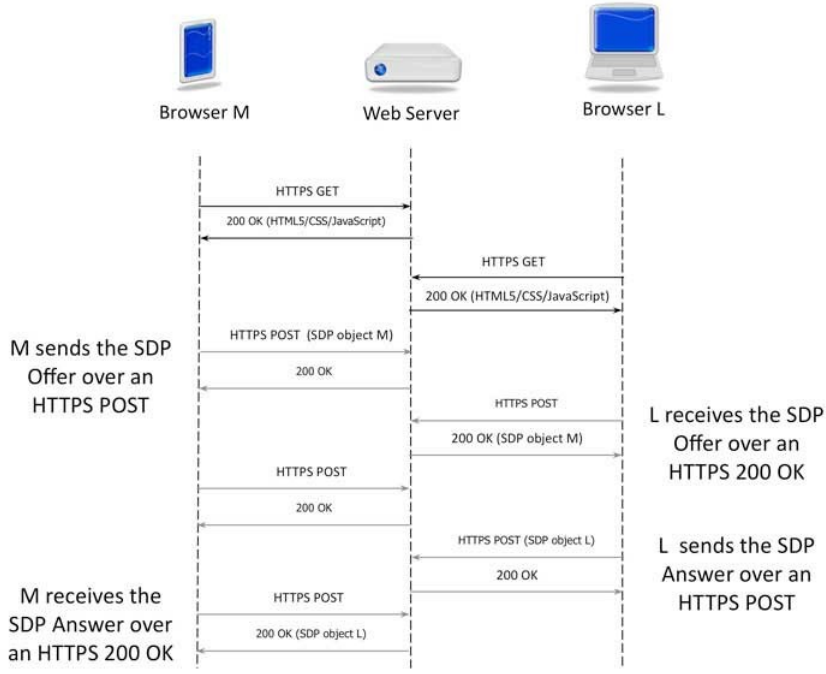


Figure 4.7 Signaling using HTTP Polling of Web Server

To exchange signaling information, the JavaScript running in each browser sends HTTP messages to the signaling server at regular intervals to poll. Signaling information sent by the browser is included in the *POST* method. Signaling information received from the server is included in the *200 OK* response to the *POST*. Note that in this approach, each message is a new request. See Chapter 7 for a complete description of this approach.

4.3.4 WebSocket Proxy

A WebSocket proxy used for WebRTC signaling would use a server which has a public IP address and is reachable by both browsers establishing the Peer Connection. Each browser opens an independent WebSocket connection with the same server, and the server bridges the connections, proxying information from one to another, as shown in Figure 4.8. Since JavaScript does not support DNS lookups, the WebSocket server will need to be provided by the web server as an IP address and port number.

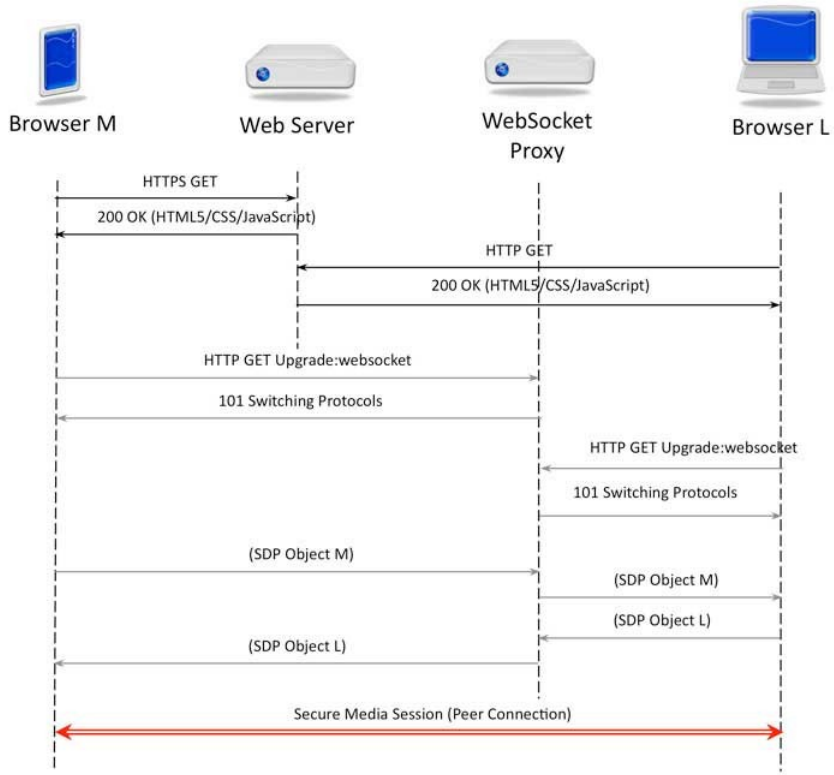


Figure 4.8 Signaling Example: WebSocket Proxy

One example of this is described in [GINGER-TECH]. In this example, a simple WebSocket server is created which listens on a specific port for WebSocket connections. When information is received from a WebSocket, it is broadcast to all open WebSocket connections. For a real WebSocket proxy, the server would not broadcast to all open connections but to one particular connection that has the other browser. Some sort of session ID would be needed for this. Another simple approach would be to allocate one port on the WebSocket proxy for each Peer Connection. The port would be randomly allocated and shared with each browser.

Some signaling pseudo code snippets:

```
// define signaling channel
function createSignalingChannel(msgHandler) {
  var socket = new WebSocket('wss://192.168.0.1:49152/');
  socket.addEventListener("message", msgHandler, false);
```

```
  return {"send": socket.send};

}

// open WebSocket
signalingChannel = createSignalingChannel(onMessage);

// send message over WebSocket
signalingChannel.send(message);

// process incoming message over WebSocket
function onMessage(msg) {
      // handle message
}
```

The use of secure WebSockets transport provides encryption between the browser and the WebSocket server.

4.3.5 Google App Engine Channel API

Another common, albeit proprietary approach is to use the Google App Engine Channel API for the signaling channel. This is described in [APPRTC]. The Google App Engine allows a JavaScript client to establish a channel with a server inside the Google cloud. The Channel API uses XHR to send signaling messages from the browser to the Google server. A technique known by the umbrella term Comet, is used to forward the message to another browser. Comet uses HTTP long polling techniques, and is also known as Reverse AJAX(Asynchronous JavaScript and XML), AJAX Push, or HTTP Server Push. It typically involves having the client send a request to the server, which is kept open until information is to be sent from the server to the client. (The name Comet was chosen as a pun since it is another brand of cleaner, as is Ajax.) Messages are received from the server via the App Engine channel.

Using unique client IDs, information can be sent to the server to be forwarded to another client. An overview of the Channel API for Java is found here [APP-ENGINE-CHANNEL]. The other browser would be identified by the Client ID.

A call to *goog.appengine.Channel(channelToken)* opens the channel. Event handlers for the following events should be setup: *onopen, onmessage, onerror, onclose*. See the example XHR pseudo code snippets below:

```
function createSignalingChannel(channelToken, msgHandler) {
  var channel = new goog.appengine.Channel(channelToken);
  var handler = {
    'onopen': onChannelOpen,
    'onmessage': msgHandler,
    'onerror': onChannelError,
    'onclose': onChannelClose
  };
  var socket = channel.open(handler);

  function onChannelOpen() {
            // channel open
  }
  function onChannelError() {
            // channel error
  }
  function onChannelClosed() {
            // channel closed
  }

  function XHRSend(msg) {
    var req = new XMLHttpRequest();
    req.open('POST', /* path to service */, true);
    req.send(msg);
  }

  return {'send': XHRSend};
}

// open channel
signalingChannel =
  createSignalingChannel(/* some token here */, onMessage);

// send message over XHR
signalingChannel.send(message);

// process incoming message from App Engine Channel
function onMessage(msg) {
  // handle message
}
```

4.3.6 SIP over WebSockets

SIP over WebSockets is another approach. Session Initiation Protocol [RFC3261] is a signaling protocol commonly used in Voice over IP (VoIP) and video conferencing systems. SIP is a key protocol in service provider IP Multimedia Subsystems (IMS) and SIP trunking for PSTN replacement. SIP is also used in enterprise communication systems for Unified Communications (UC) and Instant Messaging (IM) and presence. SIP can use UDP, TCP, SCTP, or TLS as transports. A new Internet-Draft defines a WebSocket transport for SIP [draft-ietf-sipcore-sip-websocket].

In this signaling approach, a browser loads a JavaScript SIP User Agent and then establishes a connection (*REGISTER*) with a SIP Proxy Server/Registrar which supports the WebSocket extension. A browser initiating a WebRTC session would send an *INVITE* containing the SDP offer from the browser to the Proxy Server. The destination browser would be identified by the SIP URI (e.g. *sip:user@webserver.org*). The other browser would receive the *INVITE* and generate appropriate SIP responses, returning the SDP answer in a *200 OK* response which would establish the media session.

Table 4.1 lists some open source SIP stacks that currently support SIP over WebSocket. A number of commercial SIP stacks also support SIP over WebSocket.

SIP Stack	Role	URL
JsSIP	JavaScript SIP User Agent	http://jssip.net
sipML5	JavaScript SIP User Agent	http://sipml5.org
WebRTComm	JavaScript SIP User Agent	https://code.google.com/p/webrtcomm
Asterisk	SIP Server supporting WebSockets	http://asterisk.org
OverSIP	SIP Server supporting WebSockets	http://oversip.net
Kamailio	SIP Server supporting WebSockets	http://kamailio.org
OfficeSIP	SIP Server supporting WebSockets	http://officesip.com

Table 4.1 Open Source SIP over WebSocket Implementations

The message flow for SIP over WebSocket signaling for WebRTC is shown in Figure 4.9.

An example SIP over WebSockets message is shown below:

```
INVITE sip:bob@atlanta.com SIP/2.0
Via: SIP/2.0/WS df7jal23ls0d.invalid;branch=z9hG4bK56sdasks
From: sip:alice@atlanta.com;tag=asdyka899
To: sip:bob@atlanta.com
Call-ID: asidkj3ss
CSeq: 1 INVITE
Max-Forwards: 70
Supported: path, outbound, gruu
Route: <sip:proxy.atlanta.com:443;transport=ws;lr>
Contact: <sip:alice@atlanta.com
 ;gr=urn:uuid:f81-7dec-14a06cf1;ob>
Content-Type: application/sdp

(SDP Offer not shown)
```

Note the host name in the *Via* header field which is *.invalid*, making it unroutable, the *WS* token in the *Via*, and the *transport=ws* parameter in the *Route* header field. SIP over WebSockets requires support of SIP Path header field [RFC3327], SIP Outbound [RFC5626], and GRUU [[RFC5627].

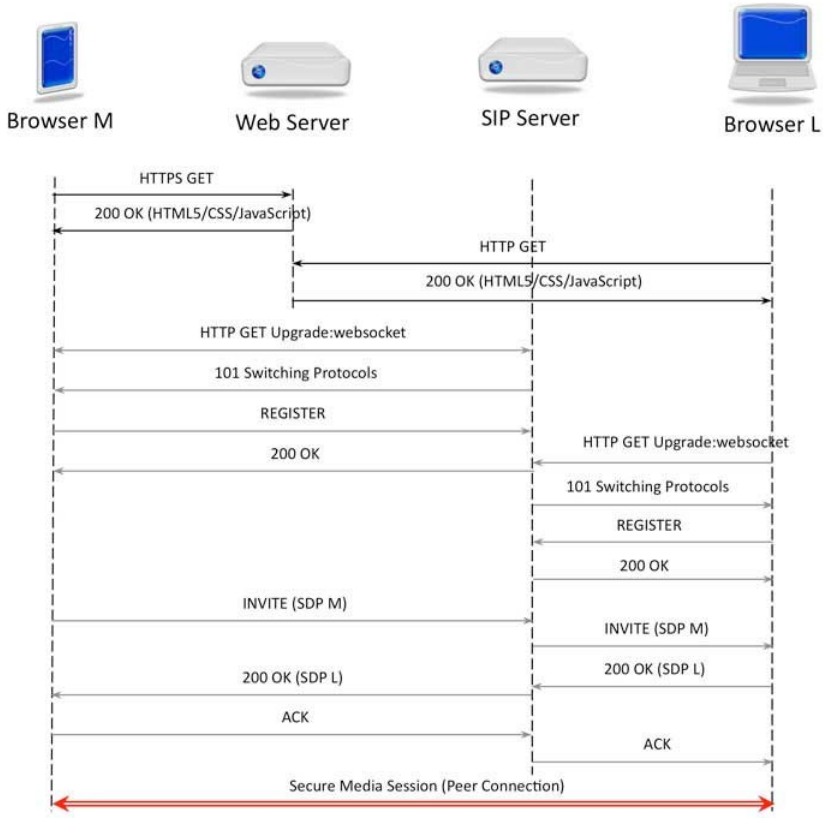


Figure 4.9 SIP over WebSocket Signaling

See the example jsSIP pseudo code snippet below:

```
// initialize SIP UA
var configuration = {
  'outbound_proxy_set': 'ws://sip-ws.example.com',
  'uri': 'sip:alice@example.com',
  'password': '123456'
};

var sipUa = new JsSIP.UA(configuration);

sipUA.call('sip:bob@example.com',useAudio, useVideo,
eventHandlers, views);
```

4.3.7 Jingle over WebSockets

Jingle is an extension of XMPP (extensible Messaging and Presence Protocol), also known as Jabber [RFC6120], that adds media signaling capabilities to XMPP. Jingle provides a way to map SDP session descriptions to an XML format, which can then be transported over TCP or TLS to an XMPP server. Standalone XMPP Jabber clients are commonly used for Instant Messaging and Presence, and is used by GoogleTalk and other enterprise IM services. XMPP has been used embedded in web pages for many years using a variety of techniques including Bidirectional-streams Over Synchronous HTTP (BOSH) [XEP-0124], which can be used as transport for XMPP [XEP-0206]. A new Internet-Draft defines the transport of XMPP over WebSockets [draft-moffitt-xmpp-over-websocket] which promises to offer better performance and interoperability.

To implement a WebRTC signaling channel using Jingle, a Jingle XMPP client written in JavaScript would be downloaded from the web page and run. The client would establish an XMPP connection over WebSockets to a XMPP Server. Clients would then map the SDP offers and answers generated by the browser into Jingle call setup messages and forward them to the other browser. The other browser would be identified by the Jabber ID (JID). The message flows for Jingle signaling in WebRTC are shown in Figure 4.10.

JavaScript XMPP stacks are listed in Table 4.2.

Stack	Role	URL
Strophe	JavaScript XMPP stack using BOSH	http://strophe.im/
node-xmpp	JavaScript XMPP stack using BOSH and WebSocket	https://github.com/astro/node-xmpp
dojox-xmpp	JavaScript XMPP stack using BOSH	http://dojotoolkit.org/api/1.3/dojox
frabjous	JavaScript XMPP stack	https://github.com/theozaurus/frabjous
jQuery-XMPP-plugin	JavaScript XMPP stack	https://github.com/maxpowel/jQuery-XMPP-plugin

Table 4.2 Open Source XMPP Stacks from [XMPP-LIBRARIES].

An example Jingle message is shown below:

```
<iq id="3Tzpd-1650" to=bob@example.com/jitsi-3u1kluu
   from=”alice@example.org/jitsi-2rbmomp" type="set">
```

```
<jingle xmlns='urn:xmpp:jingle:1'
      action='session-initiate'
      initiator='alice@example.org/jitsi-2rbmomp'
      sid='3pd4ihhk9t72q'>
 <content creator='initiator' name='audio'>
  <description xmlns='urn:xmpp:jingle:apps:rtp:1'
               media='audio'>
   <payload-type id='96' name='opus' channels='1'
                 clockrate='16000'/>
   <payload-type id='0' name='PCMU' channels='1'
                 clockrate='8000' />
  </description>
  <transport xmlns='urn:xmpp:jingle:transports:ice-udp:1'
         pwd='asd88fgpdd777uzjYhagZg' ufrag='8hhy'>
   <candidate component='1' foundation='1' generation='0'
               id='el0747fg11' ip='10.0.1.1' network='1'
               port='8998' priority='2130706431'
               protocol='udp' type='host'/>
   <candidate component='1' foundation='2' generation='0'
               id='y3s2b30v3r' ip='192.0.2.3' network='1'
               port='45664' priority='1694498815'
               protocol='udp' rel-addr='10.0.1.1'
               rel-port='8998' type='srflx'/>
  </transport>
 </content>
</jingle>
</iq>
```

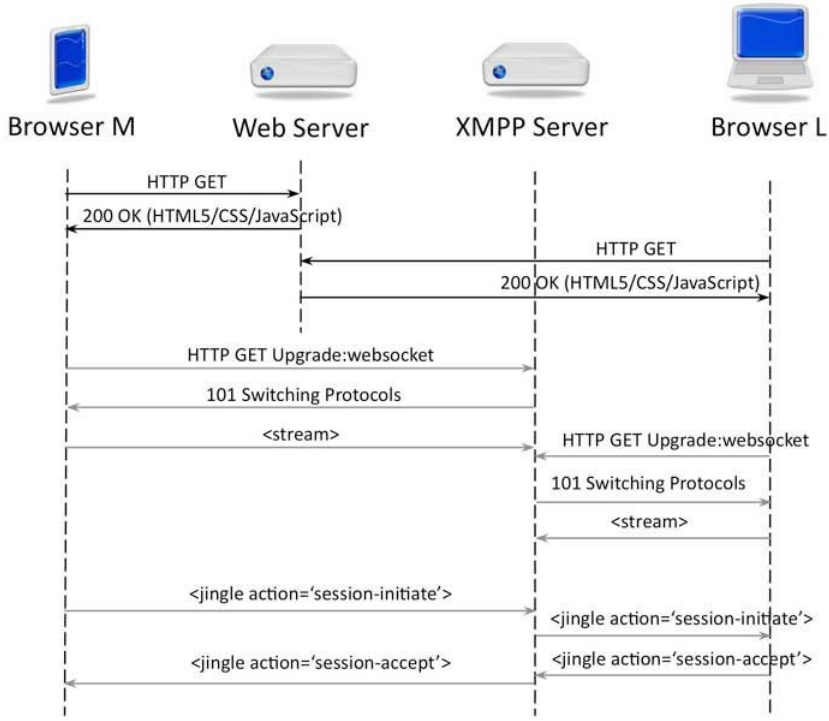


Figure 4.10 Jingle over WebSocket Signaling for WebRTC

4.3.8 Data Channel Proprietary Signaling

Data channel proprietary signaling is shown in Figure 4.11. The nature of the signaling message sent over the data channel can be completely proprietary. Some approaches do not even send SDP objects over the data

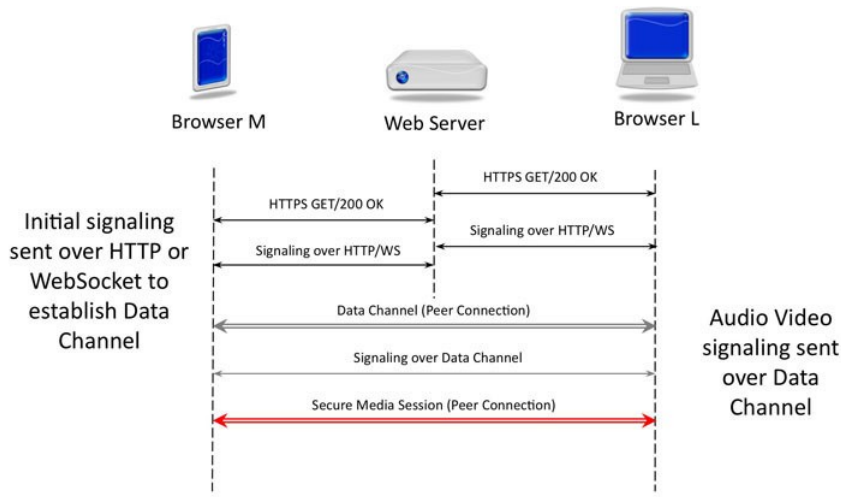


Figure 4.11 Data Channel Proprietary Signaling

channel. Instead, signaling messages are sent over the data channel which are then used to generate locally an SDP object to be passed to the browser.

4.3.9 Data Channel Using an Overlay

An alternative approach to using the data channel for signaling is to build an overlay network using the data channel, and to use that overlay network as the signaling channel. This is a peer-to-peer (P2P) approach that minimizes the need for any types of servers.

An overlay network, as the name suggests, is a network which overlays, or sits on top of another network. The overlay hides the underlying topology and architecture, and provides an alternative way to address and message other members of the overlay. For example, one type of overlay network is a ring. Each member of the overlay keeps track of other neighbor nodes in the ring. An example of technology used to implement and utilize an overlay is a Distributed Hash Table or DHT. A DHT provides a way to map information to an overlay. There are a number of overlay routing protocols, such as Chord [CHORD], which define how information can be stored and retrieved and messages routed across an overlay. Some overlay standards such as RELOAD [RELOAD] have been developed in the P2PSIP Working Group in the IETF [PSPSIP-WG].

The data channel can be used to establish an overlay signaling network for WebRTC. The overlay protocol would be written in

JavaScript and downloaded to the browser. The web server would act as the bootstrap server and assist the browser in joining the overlay, establishing a few data channel connections with other browsers in the overlay. Once the browser joined the overlay, all future messaging and signaling with the overlay would take place over the data channel. The web server would only become involved again if the browser lost connection to the overlay and required bootstrapping again.

With the overlay established, any member of the overlay could exchange a signaling message with any other member of the overlay, and as a result establish a Peer Connection. For example, the RELOAD protocol could be used, or Open Peer [OPEN-PEER].

The main advantages of a peer-to-peer signaling network are minimal server requirements, self-organizing, self-scaling, and privacy. As an overlay grows in size, the additional signaling load is shared among the new members. The usual peer-to-peer disadvantages also apply. P2P systems do utilize resources of members, so additional bandwidth and processing is required. In addition, P2P systems must deal with malicious members which could try to disrupt the operation of the overlay.

4.4 Summary

The various approaches to WebRTC signaling are summarized in Table 4.3.

Approach	Server Requirements	Advantages
WebSocket Proxy	WebSocket server with server code	No signaling infrastructure needed
XML HTTP Request	Web server with server code	No signaling infrastructure needed
SIP	SIP Registrar/Proxy Server which supports SIP WebSocket transport	Easy to interoperate with SIP endpoints or infrastructure, no server code needed
Jingle	XMPP server which supports XMPP WebSocket transport	Easy to interoperate with Jingle endpoints or infrastructure, no server code needed
Data Channel	WebSocket or web server to establish Data Channel	Low latency signaling and signaling privacy

Table 4.3 Comparison of WebRTC Signaling Options

4.5 References

[RFC6189] http://tools.ietf.org/html/rfc6189

[RFC5763] http://tools.ietf.org/html/rfc5763

[RFC6455] http://tools.ietf.org/html/rfc6455

[AJAX] http://en.wikipedia.org/wiki/Ajax_(programming)

[JQUERY] http://jquery.com

[WS-API] http://www.w3.org/TR/websockets/

[XHR] http://www.w3.org/TR/XMLHttpRequest/

[GINGER-TECH] http://blog.gingertech.net/2012/06/04/video-conferencing-in-html5-webrtc-via-web-sockets/

[APPRTC] https://apprtc.appspot.com/

[APP-ENGINE-CHANNEL] https://developers.google.com/appengine/docs/java/channel/overview

[RFC3261] http://tools.ietf.org/html/rfc3261

[draft-ietf-sipcore-sip-websockets] http://tools.ietf.org/html/draft-ietf-sip-websocket

[draft-moffitt-xmpp-over-websocket] http://tools.ietf.org/html/draft-moffitt-xmpp-over-websocket

[RFC3327] http://tools.ietf.org/html/rfc3327

[RFC5627] http://tools.ietf.org/html/rfc5627

[RFC5626] http://tools.ietf.org/html/rfc5626

[RFC6120] http://tools.ietf.org/html/rfc6120

[XEP-0124] http://xmpp.org/extensions/xep-0124.html

[XEP-0206] http://xmpp.org/extensions/xep-0206.html

[[XMPP-LIBRARIES] http://xmpp.org/xmpp-software/libraries/

5 W3C WEBRTC DOCUMENTS

WebRTC is defined by the APIs, many of which are still-being-written. The following sections describe the W3C WebRTC standards documents. Links to both the public working draft and the editor's drafts are provided for reference. Appendix A describes the W3C standards process.

5.1 WebRTC API Reference

Tables 5.1 through 5.11 list the WebRTC APIs and provide a summary of their use. Those documents are described in the next two sections. In the tables, the Reference column (Ref) indicates in which W3C document the API is defined, and contains the value "PC" for Peer Connection [1], described in Section 5.3.1, or "gUM" for getUserMedia [2] described in Section 5.3.2.

Interface and Description	Ref
`AudioMediaStreamTrack` Subclass of `MediaStreamTrack` that can only hold media of `sourceType` "audio".	PC
`MediaStream` Represents a collection of `MediaStreamTrack`s, currently only audio and video.	gUM
`MediaStreamTrack` Represents a single track of a media source. Note that a track can consist of multiple channels, as with a 6-channel surround sound source encoded into a single track. Also, a track may only contain one kind of media regardless of how many channels it has.	gUM
`MediaStreamTrackList` Represents an ordered list of `MediaStreamTrack`s.	gUM
`NavigatorUserMedia` Pre-existing interface in all web browsers.	gUM
`NavigatorUserMediaError` Represents an error returned from a call to `NavigatorUserMedia.getUserMedia()`.	gUM
`RTCDataChannel` Represents a channel over an `RTCPeerConnection` that can carry arbitrary application data.	gUM
`RTCIceCandidate` Container for an ICE candidate.	PC
`RTCPeerConnection` Represents a WebRTC connection between two peers.	PC
`RTCSessionDescription` Container for SDP offer, answer, or pranswer (provisional answer).	PC

Table 5.1 WebRTC API Interface Summary Part 1/2
(PC=Peer Connection [1], gUM=getUserMedia [2])

Interface and Description	Ref
`RTCStatsReport` Contains one or more `RTCStats` objects for the track passed as input to `getStats()`. Since a track may be transmitted over a Peer Connection using one or multiple SSRCs, this report returns an `RTCStats` object for each such SSRC.	PC
`URL` Pre-existing interface in all web browsers.	gUM
`VideoMediaStreamTrack` Subclass of `MediaStreamTrack` that can only hold media of `sourceType` "video".	PC

Table 5.1 WebRTC API Interface Summary
(PC=Peer Connection [1], gUM=getUserMedia [2])

RTCPeerConnection API and Description	Type	Ref
`RTCPeerConnection` Represents a WebRTC connection between two peers.	Interface	PC
`new RTCPeerConnection(configuration)` Creates a new `RTCPeerConnection` object using the given STUN and TURN server information. The configuration parameter is of type `RTCConfiguration`.	Constructor	PC
`RTCPeerConnection.close()` Closes the `RTCPeerConnection`, effectively removing all attached streams and closing all attached `RTCDataChannels`.	Method	PC
`RTCPeerConnectionErrorCallback` Application-settable to a function/method that takes a `DOMString` of error information as a parameter. Used by `RTCPeerConnection.createOffer()`, `RTCPeerConnection.createAnswer()`, `RTCPeerConnection.setLocalDescription()`, `RTCPeerConnection.setRemoteDescription()`, and `RTCPeerConnection.getStats()`.	Callback	PC

Table 5.2 WebRTC RTCPeerConnection APIs

(PC=Peer Connection [1], gUM=getUserMedia [2])

SDP Processing APIs and Description	Type	Ref
`RTCSessionDescription` Container for SDP offer, answer, or pranswer (provisional answer).	Interface	PC
`new RTCSessionDescription(descriptionInitDict)` Creates a new `RTCSessionDescription` object. The `descriptionInitDict` parameter is of type `RTCSessionDescriptionInit`.	Constructor	PC
`RTCSessionDescription.type` Indicates whether the session description is an offer, an answer, or a pranswer.	Attribute	PC
`RTCSessionDescription.sdp` A string representation of the SDP for the session description.	Attribute	PC
`RTCSessionDescriptionInit` Container for a session description to initialize an `RTCSessionDescription` object.	Dictionary	PC
`RTCSessionDescriptionInit.type` Indicates whether the session description is an offer, an answer, or a pranswer.	Attribute	PC
`RTCSessionDescriptionInit.sdp` A string representation of the SDP for the session description.	Attribute	PC
`RTCSessionDescriptionCallback` Application-settable to a function/method that accepts an `RTCSessionDescription` as a parameter. Used by `RTCPeerConnection.createOffer()` and `RTCPeerConnection.createAnswer()`.	Callback	PC
`RTCIceCandidate.sdpMid` Media stream identifier for the m-line associated with this candidate.	Attribute	PC

Table 5.3 WebRTC SDP Processing APIs Part 1/2

(PC=Peer Connection [1], gUM=getUserMedia [2])

SDP Processing APIs and Description	Type	Ref
`RTCIceCandidate.sdpMLineIndex` Zero-based index of the m-line associated with this candidate.	Attribute	PC
`RTCPeerConnection.createOffer()` Creates an `RTCSessionDescription` for an offer with SDP representing the complete set of available local media streams, codec options, ICE candidates, etc.	Method	PC
`RTCPeerConnection.createAnswer()` Creates an `RTCSessionDescription` for an answer with SDP representing an appropriate set of available local media streams, codec options, ICE candidates, etc.	Method	PC
`RTCPeerConnection.setLocalDescription()` Records the given `RTCSessionDescription` object as the current local description. If the object is a final answer, media will then change/begin flowing.	Method	PC
`RTCPeerConnection.setRemoteDescription()` Records the given `RTCSessionDescription` object as the current remote description.	Method	PC
`RTCPeerConnection.localDescription` The `RTCSessionDescription` representing the currently active local description (SDP).	Attribute	PC
`RTCPeerConnection.remoteDescription` The `RTCSessionDescription` representing the currently active remote description (SDP).	Attribute	PC
`RTCPeerConnection.signalingState` Holds the status of SDP exchanges over the RTCPeerConnection: `stable`, `have-local-offer`, `have-remote-offer`, `have-local-pranswer`, `have-remote-pranswer`, `closed`.	Attribute	PC
`RTCPeerConnection.onnegotiationneeded` Application-settable to a function/method that will be called whenever local or remote changes to the `RTCPeerConnection` will result in SDP changes that will require renegotiation.	Attribute	PC
`RTCPeerConnection.onsignalingstatechange` Application-settable to a function/method that will be called whenever `RTCPeerConnection.signalingState` changes.	Attribute	PC

Table 5.3 WebRTC SDP Processing APIs Part 2/2
(PC=Peer Connection [1], gUM=getUserMedia [2])

ICE Processing APIs and Description	Type	Ref
`RTCIceCandidate` Container for an ICE candidate.	Interface	PC
`new RTCIceCandidate(candidateInitDict)` Creates a new `RTCIceCandidate` object from the input parameter. This parameter is of type `RTCIceCandidateInit`.	Constructor	PC
`RTCIceCandidate.candidate` A string representing the ICE candidate.	Attribute	PC
`RTCIceCandidate.sdpMid` Media stream identifier for the m-line associated with this candidate.	Attribute	PC
`RTCIceCandidate.sdpMLineIndex` Zero-based index of the m-line associated with this candidate.	Attribute	PC
`RTCIceCandidateInit` Container for an ICE server URL for initializing an `RTCIceCandidate` object.	Dictionary	PC
`RTCIceCandidateInit.candidate` A string representing the ICE candidate.	Attribute	PC
`RTCIceCandidateInit.sdpMid` Media stream identifier for the m-line associated with this candidate.	Attribute	PC
`RTCIceCandidateInit.sdpMLineIndex` Zero-based index of the m-line associated with this candidate.	Attribute	PC
`RTCIceServer` Container for an ICE server URL.	Dictionary	PC
`RTCIceServer.url` A URL of a STUN or TURN server.	Attribute	PC

Table 5.4 WebRTC ICE Processing APIs Part 1/2
(PC=Peer Connection [1], gUM=getUserMedia [2])

ICE Processing APIs and Description	Type	Ref
`RTCIceServer.credential` The credential (e.g., password) to use if the `RTCIceServer.url` is the URL of a TURN server.	Attribute	PC
`RTCConfiguration` Contains an array of ICE server objects.	Dictionary	PC
`RTCConfiguration.iceServers` An array of `RTCIceServer` objects.	Attribute	PC
`RTCPeerConnection.updateIce()` Causes the browser ICE Agent to restart or update its collection of local candidates and remote candidates, depending on the parameters given.	Method	PC
`RTCPeerConnection.addIceCandidate()` Provides a remote candidate to the browser ICE Agent.	Method	PC
`RTCPeerConnection.RTCIceGatheringState` Holds the ICE Agent's current state with regard to gathering candidate addresses: `new, gathering, complete.`	Attribute	PC
`RTCPeerConnection.RTCIceConnectionState` Holds the ICE Agent's current state with respect to connecting the two peers: `new, checking, connected, completed, failed, disconnected, closed.`	Attribute	PC
`RTCPeerConnection.onicecandidate` Application-settable to a function/method that will be called whenever a new ICE candidate is available to be sent to the remote peer. This is useful for "trickle ICE."	Attribute	PC
`RTCPeerConnection.oniceconnectionstatechange` Application-settable to a function/method that will be called whenever `RTCPeerConnection.RTCIceConnectionState` changes.	Attribute	PC

Table 5.4 WebRTC ICE Processing APIs Part 2/2

(PC=Peer Connection [1], gUM=getUserMedia [2])

Data Channel APIs and Description	Type	Ref
`RTCPeerConnection.createDataChannel()` Creates a new data channel over the `RTCPeerConnection`.	Method	PC
`RTCPeerConnection.ondatachannel` Application-settable to a function/method that will be called whenever a `RTCDataChannel` is created.	Attribute	PC
`RTCDataChannelInit` Contains parameters configuring the creation of a data channel.	Dictionary	PC
`RTCDataChannel` Represents a channel over an `RTCPeerConnection` that can carry arbitrary application data.	Interface	PC
`RTCDataChannel.label` A string label for this data channel set by the application when the data channel is created.	Attribute	PC
`RTCDataChannel.ordered` A Boolean value, set by the application when this data channel was created, indicating whether messages must be delivered in order. Delivering messages in order may require delaying some messages.	Attribute	PC
`RTCDataChannel.maxRetransmitTime` For unreliable data channels, this attribute holds the number of milliseconds for which the browser will continue attempting to retransmit data. This value is configured by the application when the data channel is first created. Note that setting both this and `RTCDataChannel.maxRetransmits` will result in an error.	Attribute	PC

Table 5.5 WebRTC Data Channel APIs Part 1/3
(PC=Peer Connection [1], gUM=getUserMedia [2])

Data Channel APIs and Description	Type	Ref
`RTCDataChannel.`**`maxRetransmits`** For unreliable data channels, this attribute holds the maximum number of times the browser will attempt to retransmit data. This value is configured by the application when the data channel is first created. Note that setting both this and `RTCDataChannel.maxRetransmitTime` will result in an error.	Attribute	PC
`RTCDataChannel.`**`protocol`** The specification is not clear on this. Don't use it.	Attribute	PC
`RTCDataChannel.`**`negotiated`** A Boolean value indicating whether this data channel was automatically negotiated (i.e., not in explicit SDP exchanges). It reflects the value configured by the application when the data channel was created.	Attribute	PC
`RTCDataChannel.`**`id`** A numeric value for the channel established when the channel was created. It is determined automatically by the browser unless the application provides the value as part of the configuration when the channel is created.	Attribute	PC
`RTCDataChannel.`**`readyState`** The state of the `RTCDataChannel`. It will have one of the following values: `connecting`, `open`, `closing`,	Attribute	PC
`RTCDataChannel.`**`bufferedAmount`** The number of bytes queued for sending (by `RTCDataChannel.send()`) that have not yet been transmitted.	Attribute	PC
`RTCDataChannel.`**`onopen`** Application-settable to a function/method that will be called when the data channel is ready to transmit data.	Attribute	PC

Table 5.5 WebRTC Data Channel APIs Part 2/3
(PC=Peer Connection [1], gUM=getUserMedia [2])

Data Channel APIs and Description	Type	Ref
`RTCDataChannel.onerror` Application-settable to a function/method that will be called whenever an error occurs in the functioning of the data channel.	Attribute	PC
`RTCDataChannel.onclose` Application-settable to a function/method that will be called when the data channel is closed.	Attribute	PC
`RTCDataChannel.close()` Closes the data channel.	Method	PC
`RTCDataChannel.onmessage` Application-settable to a function/method that will be called when a message (data) is received on the data channel.	Attribute	PC
`RTCDataChannel.binaryType` A string indicating how binary data is to be exposed to the application. Its default value is "blob".	Attribute	PC
`RTCDataChannel.send()` Sends the argument over the data channel to the remote end. There are a variety of formats the argument can take, but the two most common are a string and a blob.	Method	PC

Table 5.5 WebRTC Data Channel APIs Part 3/3
(PC=Peer Connection [1], gUM=getUserMedia [2])

DTMF Processing APIs and Description	Type	Ref
`RTCPeerConnection.createDTMFSender()` For the `MediaStreamTrack` given as input, this method creates an `RTCDTMFSender` object whose purpose is to inject DTMF tones into the `RTCPeerConnection`'s representation of the track.	Method	PC
`RTCDTMFSender.canInsertDTMF` This Boolean value indicates whether the associated track is able to insert DTMF. Normally `true`, this value can become `false` if there is a problem with the track or the Peer Connection.	Attribute	PC
`RTCDTMFSender.insertDTMF()` Inserts the given DTMF string into the associated track. Note that this call replaces any to-be-played-out tones remaining in the `toneBuffer`.	Method	PC
`RTCDTMFSender.track` The `MediaStreamTrack` associated with this object.	Attribute	PC
`RTCDTMFSender.ontonechange` Application-settable to a function/method that will be called when a message (data) is received on the data channel.	Attribute	PC
`RTCDTMFSender.toneBuffer` Contains the tones remaining to be played out.	Attribute	PC
`RTCDTMFSender.duration` The number of milliseconds for which each tone should be played. The default value is 100 ms.	Attribute	PC
`RTCDTMFSender.interToneGap` The number of milliseconds between the end of a tone and the start of the next one. The default value is 50 ms.	Attribute	PC

Table 5.6 WebRTC DTMF Processing APIs
(PC=Peer Connection [1], gUM=getUserMedia [2])

Statistics Processing APIs and Description	Type	Ref
`RTCPeerConnection.getStats()` For the `MediaStreamTrack` given as input, this method collects and returns transmission statistics about the track via the `RTCStatsCallback` handler.	Method	PC
`RTCStatsCallback` Application-settable to a function/method that takes an `RTCStatsReport` as a parameter. Used by `RTCPeerConnection.getStats()`.	Callback	PC
`RTCStatsReport` Contains one or more `RTCStats` objects for the track passed as input to `getStats()`. Since a track may be transmitted over a Peer Connection using one or multiple SSRCs, this report returns an `RTCStats` object for each such SSRC.	Interface	PC
`RTCStats` Base class for an object containing statistics for a single RTP object type. This base class contains a timestamp, and id, and the type. `RTCRTPStreamStats` inherits from this class.	Dictionary	PC
`RTCRTPStreamStats` Parent class for an object containing statistics for a single RTP stream. This class contains the id of the other end of the stream (so its statistics can be checked) and the SSRC for this RTP stream. This class inherits from `RTCStats`. `RTCInboundRTPStreamStats` and `RTCOutboundRTPStreamStats` inherit from this class.	Dictionary	PC
`RTCInboundRTPStreamStats` Object containing the following statistics for a single inbound RTP stream: the total number of packets received and the total number of bytes received, since transmission began over the Peer Connection. This object inherits from `RTCRTPStreamStats`.	Dictionary	PC
`RTCOutboundRTPStreamStats` Object containing the following statistics for a single outbound RTP stream: the total number of packets sent and the total number of bytes sent, since transmission began over the Peer Connection. This object inherits from `RTCRTPStreamStats`.	Dictionary	PC

Table 5.7 WebRTC Statistics Processing APIs
(PC=Peer Connection [1], gUM=getUserMedia [2])

Identity Processing APIs and Description	Type	Ref
`RTCPeerConnection.setIdentityProvider()` Allows the application to specify an identity provider, protocol, and claimed username (identity) for this Peer Connection. This may be unnecessary if the browser already has this information configured.	Method	PC
`RTCPeerConnection.getIdentityAssertion()` Begins the Identity Provider process (of obtaining an identity assertion). Use of this method is optional but may speed up the application by starting this process before an offer or answer is generated.	Method	PC
`RTCPeerConnection.peerIdentity` This is an `RTCIdentityAssertion` for the remote peer. It is set only if the peer's identity has been verified.	Attribute	PC
`RTCPeerConnection.onidentityresult` Application-settable to a function/method that will be called when an attempt to verify identity either succeeds or fails.	Attribute	PC
`RTCIdentityAssertion` Contains the verified identity provider and identity (name) for the peer.	Dictionary	PC

Table 5.8 WebRTC Identity Processing APIs
(PC=Peer Connection [1], gUM=getUserMedia [2])

Stream Processing APIs and Description	Type	Ref
`MediaStream` Represents a collection of `MediaStreamTrack`s, currently only audio and video.	Interface	gUM
`new MediaStream(MediaStream or MediaStreamTrackSequence)` Creates a new `MediaStream` consisting of tracks of audio and video from other `MediaStream` objects. The input parameter is either a single `MediaStream` or a `MediaStreamTrackSequence`, which is an array of `MediaStreamTrack`s.	Constructor	gUM
`MediaStream.id` A unique, browser-generated identifier string for this `Media Stream` defined in the "Media Capture and Streams" document. The "WebRTC 1.0" document explains how remote-originated stream labels are created.	Attribute	gUM and PC
`MediaStream.getAudioTracks()` Returns an array of all of the `AudioMediaStreamTrack`s in this `MediaStream`.	Method	gUM
`MediaStream.getVideoTracks()` Returns an array of all of the `VideoMediaStreamTrack`s in this `MediaStream`.	Method	gUM
`MediaStream.clone()` Returns a clone of the `MediaStream` that has a different id and clones of all tracks in the `MediaStream`.	Method	gUM
`MediaStream.ended` Set by the browser, this attribute has the value `true` if and only if the stream has finished.	Attribute	gUM
`MediaStream.onended` Application-settable to a function/method that will be called when the `MediaStream` finishes.	Attribute	gUM

Table 5.9 WebRTC Stream Processing APIs Part 1/3
(PC=Peer Connection [1], gUM=getUserMedia [2]

Stream Processing APIs and Description	Type	Ref
`URL` Pre-existing interface in all web browsers.	Interface	gUM
`URL.createObjectURL` Creates and returns a "blob" URL for the `MediaStream` given as the parameter. The URL will be suitable for passing to the `<audio>` element if the stream contains audio and for passing to the `<video>` element if the stream contains video.	Method	gUM
`RTCPeerConnection.addStream()` Adds an existing media stream to an `RTCPeerConnection` for sending to the remote peer.	Method	PC
`RTCPeerConnection.removeStream()` Removes one of the `RTCPeerConnection`'s streams from the `RTCPeerConnection`, which will ultimately result in the stream not being sent anymore.	Method	PC
`RTCPeerConnection.getLocalStreams()` Returns an array containing all of the locally-originated `MediaStream` values.	Method	PC
`RTCPeerConnection.getRemoteStreams()` Returns an array containing all of the remotely-originated `MediaStream` values.	Method	PC
`RTCPeerConnection.getStreamById()` Returns the `MediaStream` in the Peer Connection with the given id, or null if there isn't one.	Method	PC
`RTCPeerConnection.onaddstream` Application-settable to a function/method that will be called whenever a remote stream is added.	Attribute	PC
`RTCPeerConnection.onremovestream` Application-settable to a function/method that will be called whenever a remote stream is removed.	Attribute	PC
`NavigatorUserMedia` Pre-existing interface in all web browsers.	Interface	gUM

Table 5.9 WebRTC Stream Processing APIs Part 2/3
(PC=Peer Connection [1], gUM=getUserMedia [2]

Stream Processing APIs and Description	Type	Ref
`NavigatorUserMedia.getUserMedia()` Returns a `MediaStream` containing one or more media tracks that satisfy the constraints (see `MediaStreamConstraints`) given as input.	Method	gUM
`NavigatorUserMediaSuccessCallback` Application-settable to a function/method that accepts a `MediaStream` as a parameter. Used by `NavigatorUserMedia.getUserMedia()`.	Callback	gUM
`NavigatorUserMediaError` Represents an error returned from a call to `NavigatorUserMedia.getUserMedia()`.	Interface	gUM
`NavigatorUserMediaError.name` The error that occurred in calling `NavigatorUserMedia.getUserMedia()`. Must be either "PERMISSION_DENIED", indicating that the user denied permission for the page to use the local device(s), or "CONSTRAINT_NOT_SATISFIED", indicating that a mandatory constraint could not be satisfied.	Attribute	gUM
`NavigatorUserMediaError.message` A human-readable description of the error that occurred.	Attribute	gUM
`NavigatorUserMediaError.constraintName` If the error name was "CONSTRAINT_NOT_SATISFIED", this attribute has as its value the constraint that caused the error.	Attribute	gUM
`NavigatorUserMediaErrorCallback` Application-settable to a function/method that accepts a `NavigatorUserMediaError` as a parameter. Used by `NavigatorUserMedia.getUserMedia()`.	Callback	gUM

Table 5.9 WebRTC Stream Processing APIs Part 3/3
(PC=Peer Connection [1], gUM=getUserMedia [2]

Track Processing APIs and Description	Type	Ref
`MediaStreamTrack` Represents a single track of a media source. Note that a track can consist of multiple channels, as with a 6-channel surround sound source encoded into a single track. Also, a track may only contain one kind of media regardless of how many channels it has.	Interface	gUM
`MediaStreamTrack.kind` Has the value `audio` or `video`.	Attribute	gUM
`MediaStreamTrack.id` A globally-unique identifier generated by the browser.	Attribute	gUM
`MediaStreamTrack.label` A browser-generated label string for this `MediaStreamTrack`, e.g., "Built-in microphone". It is optional for the browser to provide anything other than the empty string as the label.	Attribute	gUM
`MediaStreamTrack.enabled` Application-settable Boolean to disable and re-enable the output of the track.	Attribute	gUM
`MediaStreamTrack.muted` A Boolean indicating whether or not the track is muted.	Attribute	gUM
`MediaStreamTrackState.live` A possible value for `MediaStreamTrack.readyState` that indicates the track is active, i.e., capable of producing output. Note that even if active, the `MediaStreamTrack` may be disabled (see `MediaStreamTrack.enabled`) or muted (see `MediaStreamTrack.muted`) and thus not producing output.	Enum	gUM
`MediaStreamTrackState.new` A possible value for `MediaStreamTrack.readyState` that indicates the track is not yet connected to a source.	Enum	gUM

Table 5.10 WebRTC Track Processing APIs Part 1/3
(PC=Peer Connection [1], gUM=getUserMedia [2])

Track Processing APIs and Description	Type	Ref
`MediaStreamTrackState.ended` A possible value for `MediaStreamTrack.readyState` that indicates the track has ended, i.e., that it is no longer capable of producing output and never will be.	Enm	gUM
`MediaStreamTrack.onmute` Application-settable to a function/method that will be called whenever the `MediaStreamTrack` is muted.	Attribute	gUM
`MediaStreamTrack.onunmute` Application-settable to a function/method that will be called whenever the `MediaStreamTrack` is unmuted.	Attribute	gUM
`MediaStreamTrack.onstarted` Application-settable to a function/method that will be called when the `MediaStreamTrack` becomes active.	Attribute	gUM
`MediaStreamTrack.onended` Application-settable to a function/method that will be called when the `MediaStreamTrack` finishes.	Attribute	gUM
`MediaStreamTrack.readonly` A Boolean indicating whether or not the track is unconstrainable. Examples are a file or some tracks received over a Peer Connection.	Attribute	gUM
`MediaStreamTrack.remote` A Boolean indicating whether or not the track was originated remotely over a Peer Connection.	Attribute	gUM
`MediaStreamTrack.readyState` Indicates the state of the track: `new`, `live`, or `ended`. Defined in the "Media Capture and Streams" document. The "WebRTC 1.0" document explains how remote-originated track attributes must be set.	Attribute	gUM and PC
`MediaStreamTrack.clone()` Creates a duplicate of the track with its own id.	Method	gUM

Table 5.10 WebRTC Track Processing APIs Part 2/3
(PC=Peer Connection [1], gUM=getUserMedia [2])

Track Processing APIs and Description	Type	Ref
`MediaStreamTrack.stop()` Ends the track. If there are no other tracks referencing the media source, the relevant source is stopped and any device-in-use notifications are turned off.	Method	gUM
`AudioMediaStreamTrack` Subclass of MediaStreamTrack that can only hold media of `sourceType` "audio".	Interface	gUM
`AudioMediaStreamTrack.getSourceIds()` Static method that returns all available audio source ids.	Method	gUM
`VideoMediaStreamTrack` Subclass of MediaStreamTrack that can only hold media of `sourceType` "video".	Interface	gUM
`VideoMediaStreamTrack.getSourceIds()` Static method that returns all available video source ids.	Method	gUM
`MediaStream.getTrackById()` Returns the `MediaStreamTrack` in this `MediaStream` with the given id, or null if there is no such track.	Method	gUM
`MediaStream.addTrack()` Adds the track given as the parameter to the `MediaStream` if it doesn't already exist.	Method	gUM
`MediaStream.removeTrack()` Removes the track given as the parameter from the `MediaStream`.	Method	gUM
`MediaStream.onaddtrack` Application-settable to a function/method that will be called whenever a track is added to this `MediaStream`.	Attribute	gUM
`MediaStream.onremovetrack` Application-settable to a function/method that will be called whenever a track is removed from this `MediaStream`.	Attribute	gUM

Table 5.10 WebRTC Track Processing APIs Part 3/3
(PC=Peer Connection [1], gUM=getUserMedia [2])

Constraint and Capability APIs and Description	Type	Ref
`MediaStreamTrack.getSourceInfos()` Returns source-distinguishing information an application can use to create functional audio/video input chooser dialogs.	Method	gUM
`MediaStreamTrack.constraints()` Returns the current constraints, if any, applied to this track.	Method	gUM
`MediaStreamTrack.states()` Returns the current values of all constraint states.	Method	gUM
`MediaStreamTrack.capabilities()` Returns the supported values for all constraints possible on this track.	Method	gUM
`MediaStreamTrack.applyConstraints()` Attempts to replace the track's current constraints with those given as argument.	Method	gUM
`MediaStreamTrack.onoverconstrained` Application-settable to a function/method that will be called whenever this track becomes *overconstrained*.	Attribute	gUM
`SourceInfo` Contains simple info about a source useful for disambiguation.	Dictionary	gUM
`SourceInfo.sourceId` A unique id for this source. See `MediaStreamTrack.id`.	Attribute	gUM
`SourceInfo.kind` Either "audio" or "video".	Attribute	gUM
`SourceInfo.label` A browser-provided label for the track. See `MediaStreamTrack.label`.	Attribute	gUM

Table 5.11 WebRTC Constraint and Capability APIs Part 1/2 (PC=Peer Connection [1], gUM=getUserMedia [2])

Constraint and Capability APIs and Description	Type	Ref
`MediaStreamConstraints` Contains constraints for selection of audio tracks, video tracks, both, or neither.	Dictionary	gUM
`MediaTrackConstraints` Contains mandatory and/or optional constraints for use when selecting tracks of a single media type (audio or video).	Dictionary	gUM
`MediaTrackConstraintSet` Contains one or more key-value pairs, where each key is a constraint to be satisfied. Used to represent a set of mandatory constraints.	Dictionary	gUM
`MediaTrackConstraint` Contains one key-value pair, where the key is a constraint to be satisfied. Used to represent a single optional constraint.	Dictionary	gUM
`MediaSourceStates` Contains all state names and their current values for the track.	Dictionary	gUM
`CapabilityRange` Contains minimum and maximum allowed values for a capability, along with a Boolean flag indicating whether or not the capability is supported.	Dictionary	gUM
`AllVideoCapabilities` Contains all possible capabilities for a video track, with their permitted values.	Dictionary	gUM
`AllAudioCapabilities` Contains all possible capabilities for an audio track, with their permitted values.	Dictionary	gUM
`MinMaxConstraint` Contains a min/max setting for a constraint that is of that type.	Dictionary	gUM

Table 5.11 WebRTC Constraint and Capability APIs Part 2/2 (PC=Peer Connection [1], gUM=getUserMedia [2])

5.2 WEBRTC Recommendations

None of the WEBRTC specifications have reached Recommendation status yet.

5.3 WEBRTC Drafts

All the WEBRTC documents are in the Working Draft stage. They are described in the following subsections. Note that the descriptions below are intended to be read in conjunction with the specification documents themselves – they do not repeat everything given in the specifications.

5.3.1 "WebRTC 1.0: Real-time Communication Between Browsers"

This document [1] is the primary document for the WebRTC work, informally known as the Peer Connection draft. It defines the *RTCPeerConnection* interface and extensions to the *MediaStream* interface defined in the "Media Capture and Streams" (getusermedia) specification. There are many bits of functionality defined in the RTCPeerConnection interface. To simplify life for the reader of the specification, related functionality has been grouped into separate sections, giving the naïve reader the impression that there are only a small number of methods and attributes in the RTCPeerConnection definition. In this section, we explain each of these grouped functionality sets. They include: the core RTCPeerConnection itself, the Data API, the DTMF API, the Statistics API, and the Identity API.

5.3.1.1 RTCPeerConnection Interface

The *RTCPeerConnection* interface is the primary API of the WebRTC effort. The function of this API is to set up a media connection path between two browsers. Although the API is strongly tied to the JavaScript Session Establishment Protocol (JSEP, described in Section 8.3.7) for media negotiation, most of the details for JSEP are handled by the browser.

The constructor for the *RTCPeerConnection* object takes a configuration object containing the addresses of servers that assist in establishing sessions through NATs and firewalls (STUN and TURN servers, described in Section 6.2.5 and 6.2.6). Optionally, it also takes a *MediaConstraints* object. The format of this object is described in the section on the "Media Capture and Streams" specification (Section 5.3.2). Currently the specification defines the following constraints: *OfferToReceiveVideo*, *OfferToReceiveAudio*, *VoiceActivityDetection*, *IceTransports*, *IceRestart,* and *RequestIdentity*. It is important to note

that these are constraints on the *RTCPeerConnection* and not on the selection of media, despite the name of the object.

The *RTCPeerConnection* object can have associated media streams, of course. These are added and removed using the *addStream()* and *removeStream()* methods, respectively. Note that the media streams are not created by these methods; these methods add and remove existing local streams from the set of streams being sent to the remote peer. The *getLocalStreams()*and *getRemoteStreams()*methods can be used to fetch arrays tracking the complete set of local and remote streams, respectively. A not very obvious but significant aspect of the Peer Connection API is that it is the responsibility of the web application to manage SDP session negotiation. In other words, *addStream()* and *removeStream()* do not cause media to flow or stop flowing. They change the internal state of the local *RTCPeerConnection* object, but an explicit session negotiation is needed to coordinate the media change with the remote end. To trigger the application code to do the negotiation, the *addStream()* and *removeStream()* methods cause a *negotiationneeded* event to be thrown. When the application catches this event (or sets the *onnegotiationneeded* callback), the application must then negotiate media by:

1) Calling *createOffer()* - the user agent will examine the internal state of the Peer Connection and generate an appropriate *RTCSessionDescription* object (an offer).
2) Calling *setLocalDescription()* with the *RTCSessionDescription* object.
3) Sending the generated SDP session description to the remote peer. Note that the specification does not define or mandate the mechanism to send SDP session descriptions to and from the remote peer. The specification refers to this undefined channel as the “signaling channel”, described in more detail in Chapter 4.

Of course, if the remote peer were the one to send the offer, the application would need to call *createAnswer()* instead and send the generated *RTCSessionDescription* object back. The call to *setRemoteDescription()* must be done by the application when the remote offer (or answer) is received on the signaling channel in order for the offer/answer negotiation to be completed. In either case, local or remote, the actual media state change occurs when a final answer is successfully applied in the browser.

An obvious question at this point is why the setup and parsing of the

signaling channel, as well as all of the negotiation, were left to the application. The primary reason is flexibility. With respect to signaling, many browser-to-browser communication applications will have both browsers using exactly the same source code from the same web server, making it logical to have the signaling done through the server. Others may want to signal through gateways. With respect to the SDP session negotiation, the two biggest advantages of leaving it to the application are the ability to do "trickle" ICE, where media can begin flowing even before all ICE candidates have been checked (see Section 8.5.1), and the ability to modify the SDP if necessary. Since SIP/SDP interoperability is still not at 100%, it is not uncommon to need to adjust the SDP. It is likely that libraries will be developed to support the most common use cases for both signaling and negotiation.

A complexity in the Peer Connection mechanism is that it has two processes, an ICE process and an offer/answer media negotiation process, each of which has its own state machine. The offer/answer state machine is controlled by the JavaScript process, while the ICE state machine is controlled by the browser. The session description reflected in the SDP carries the media offered or answered, as well as the candidates for ICE "hole punching". However, the ICE process is not dependent upon the offer/answer process, which allows for the sometimes slow-moving ICE process to continue checking additional candidates after the media negotiation has completed, i.e., when no more SDP needs to be exchanged in order to agree upon the media. Although in reality there may be more than one underlying ICE state machine, depending on the way the media transport is specified, the WebRTC API exposes a combined state machine. In the specification, the progress of ICE processing is actually split into two pieces – candidate gathering, and checking and connection – because they can proceed somewhat in parallel. The *iceGatheringState* values are simply *new*, *gathering* and *complete*. The *iceConnectionState* values are *new, checking, connected, completed, failed, disconnected, closed*. Again, progression through these states occurs largely without the involvement of the JavaScript code. However, the ability to check these state variables can be useful when building robust applications. The *signalingState* variable tracks the offer/answer exchange status, with values of *stable*, *have-local-offer*, *have-remote-offer*, *have-local-pranswer*, *have-remote-pranswer*, and *closed*. The specification contains state transition diagrams for both *signalingState* and *iceConnectionState*. The JSEP signaling state machine diagram in Figure 8.6 is very similar to the transition diagram for *signalingState*.

5.3.1.2 DataChannel Interface

A key piece of functionality defined in the WebRTC specification is the *RTCDataChannel* interface, an API for a bi-directional data channel for use over a Peer Connection. To create the *RTCDataChannel*, the aptly-named *RTCPeerConnection.createDataChannel()* method is provided. In addition to an application-settable *label*, the method also takes an optional *dataChannelDict* configuration object. The different properties of this object allow for control of two primary characteristics: reliability of the channel and whether ordering matters. In an ideal world, there would be no cost to having reliable, ordered delivery. In practice, however, perfect reliability would require infinite retransmissions, and ordered delivery would require an infinite buffer to collect out-of-order messages. The two attributes *maxRetransmitTime* and *maxRetransmits* are used to limit the reliability. The first allows the application to specify the maximum amount of time, in milliseconds, that the browser will continue retransmitting. The second allows the application to specify the maximum total number of retransmission attempts. Only one may be specified (without causing an error). The Boolean ordered attribute allows the application to specify, as you might have guessed, whether or not the data channel is to deliver the messages in order. The configuration object also allows for the application to set an *id* (or have it chosen automatically by the browser) and to specify that negotiation of new data channels will be handled by the browser (the default for *negotiated*) rather than by the application.

The progress of the creation of a new *RTCDataChannel* is tracked by its *readyState* property, with values of *connecting* (the data channel is being established), *open* (the data channel is ready to be used), *closing* (the data channel is being shut down), and *closed* (the data channel has been shut down or was never established in the first place). Clearly there is a *close()* method on *RTCDataChannel*, as well as handlers that can be set for *onopen*, *onclose*, and *onerror*.

Once established, the data channel behaves similarly to a WebSocket, with a *send()* method for sending data and an ability to set an *onmessage* handler for incoming messages. Some of the details of data channels are still being finalized, but the general structure is fairly stable now. Future editions of the book will likely be able to include sample code.

5.3.1.3 DTMF API

Audio communication channels, when connected to today's phone networks, need to be able to convey DTMF (Dual Tone Multi-

Frequency) tones. In particular, a JavaScript application sending audio into a SIP infrastructure will need a way to generate DTMF tones, since the audio may come from a generic microphone or other device without a DTMF keypad. However, it is during the negotiation of media for a Peer Connection that an ability to also send RFC 4733 DTMF packets is arranged. The specification thus provides an interface on *RTCPeerConnection* to associate DTMF capability with a designated *MediaStreamTrack* for that *RTCPeerConnection*. This section adds a method *createDTMFSender()* to the *RTCPeerConnection* interface that takes a *MediaStreamTrack* as argument. Thereafter, DTMF tones can be sent on that *MediaStreamTrack* over the Peer Connection via the *insertDTMF()* method on the new *RTCDTMFSender* object. Note that DTMF is inserted only on an *RTCDTMFSender* and not on the original *MediaStreamTrack* itself.

5.3.1.4 Statistics API

The WebRTC specification defines an API for collecting statistics from the tracks being transported over a Peer Connection. This can be important for determining how many packets are getting through, for example. The API works as follows: *getStats()* takes a *MediaStreamTrack* as input, along with a callback to be executed when the statistics are available. The callback is given an *RTCStatsReport* containing *RTCStats* objects. Statistics are grouped into these objects by type, but all types contain id, type, and timestamp properties. Currently the only defined types are *inbound-rtp* and *outbound-rtp*, both of which are instances of the *RTCRTPStreamStats* subclass that additionally provides *remoteId* and *ssrc* properties. The *outbound-rtp* object type is represented by subclass *RTCOutboundRTPStreamStats*, which provides *packetsSent* and *bytesSent* properties. The *inbound-rtp* object type is, unsurprisingly, represented by the subclass *RTCInboundRTPStreamStats* that provides analogous *packetsReceived* and *packetsSent* properties. Together, these properties can be used to track both how much data is being sent on the track and how much is being received. Note that the *remoteId* property can be used directly as a key into the *RTCStatsReport* to pull out the report for the remote end of the current track. Regardless of type, objects in the report can easily be processed using a for loop to access each one.

5.3.1.5 Identity API

The Identity API in the WebRTC specification is slightly different from

the other APIs in that the underlying use of a web Identity Provider is not supposed to require any action on the part of the web application, assuming that the browser has already been configured with a claimed identity and an identity provider. In theory, all the application needs to do is to set the *onidentityresult* handler that will be called whenever an identity has been verified since on one end the browser can automatically generate a signature for the configured identity and on the other end the browser can automatically pull up the Identity Provider's verification JavaScript to verify the signature. In practice, however, the IETF draft (see Section 10.4) describing the underlying protocol requirements suggests that the signature itself needs to be transported using the signaling channel. Since the signaling channel is only used by the application JavaScript code directly, the application code would at least need to be involved in this step. In short, the specification is still vague in several respects. However the signing process is initiated, the signature conveyed, and the signature verified, the Identity API does provide a way for the application code to override the browser's configured identity and Identity Provider – the *setIdentityProvider()* method on the Peer Connection. The API also provides the *getIdentityAssertion()* method for applications that wish to start the identity checking process in advance of the offer/answer exchange (when it would occur automatically on its own). This may improve the user interface responsiveness if the time to verify an identity is long due to user login time needs.

5.3.1.6 MediaStream Interface Extensions

The WebRTC specification also defines extensions to the *MediaStream* interface defined in the "Media Capture and Streams" specification (Section 5.3.2). First, each *MediaStream* has an *id* attribute to distinguish it from others sent through the *RTCPeerConnection* API. Second, the remote addition or removal of a *MediaStreamTrack* to an existing *MediaStream* will generate local *addtrack* and *removetrack* events. Similarly, remote track muting and unmuting is duplicated locally as well.

5.3.2 "Media Capture and Streams"

This document [2] is under the control of the Media Capture Task Force, which is composed of members of both the WEBRTC Working Group and the Device APIs and Policy (DAP) Working Group. Formerly known as the getusermedia draft, it defines the *getUserMedia()*method, the API for requesting and obtaining a local media stream from a device

such as a camera or microphone.

5.3.2.1 getUserMedia() method

This API is intended to be the primary API used by all web application developers to obtain access to local device media. As such, it requires the browser to obtain user permission before accessing the device. However, the mechanism for obtaining this permission, the specificity of the permission, the duration of the permission, and all other details regarding permission are left up to the user agent. User agents are encouraged to indicate in a prominent manner when local devices are recording. There are two separate callbacks, *successCallback* and *errorCallback*, for the cases of a successful setup and a failure, respectively. The latter also returns an error code, but at the moment the only valid error defined is the constant *PERMISSION_DENIED*, which has the value 1.

5.3.2.2 States, Capabilities, and Constraints

The format and processing of the constraints argument to *getUserMedia()* warrants a separate explanation. This parameter consists of a JavaScript object with one property for each media type (currently only *audio* and *video*). The value for each type can be a Boolean value (explained in a moment) or an object. The object has two optional properties: *mandatory* and *optional*. The former contains a set of constraint keys and values that must be satisfied – if not, an error will be returned. The second contains a priority-ordered list of constraint keys and values that should be satisfied. Failure to satisfy one or more of these optional constraints does not result in an error, but there is a requirement that when conflicting constraints exist, the one occurring earlier in the sequence is to be satisfied. If, instead of an object value, a Boolean value of *true* is given for a media type, e.g., *video:true*, then either a video track must be obtained or an error returned. The list of possible constraints is yet to be defined for media, although some likely ones are currently given as examples in the specification.

The ability to select and control track and other properties via constraints has two advantages over an earlier proposal for a simpler API that used optional hints: an ability for the application to indicate in advance whether certain tracks would be unacceptable, and an ability to indicate whether some constraints are more important than others. For these reasons, and to allow for extensibility going forward, *getUserMedia()* accepts the *constraints* parameter described in the

preceding paragraph.

Each constraint has an associated capability and an associated state, both described fairly extensively in the Media Capture and Streams specification. The capability is the valid range for the constraint. The state is the current setting/value of the source for that constraint. Although capabilities, states, and constraints are all accessed via the methods and attributes of a *MediaStreamTrack*, the first two are rightly properties of the media source feeding the track. In other words, two tracks that share the same source will have the same state value. Let's look at video width as an example.

Used in a constraint specification:

```
{
 mandatory: {
  width: { max: 1280 }
 },
 optional: [
  { width: 1000 },
  { width: { min: 640 }}
 ]
}
```

In this example, the structure is requesting that the video track have a width of exactly 1000 pixels if possible (meaning min of 1000 and max of 1000). If that fails, it should at least have a width greater than or equal to 640. Most importantly, the width must not exceed 1280 pixels. If the source cannot provide a width of no more than 1280 pixels the constraint request must fail.

As a state value:

mytrack.states().width returns the track's video width

As a capability:

mytrack.capabilities().width might return a structure such as *{ max: 1920, min: 640, supported: true }*

The capability and state functionality are fairly new to the specification and will probably change somewhat. In particular, the *getSourceInfos()* method provides similar information to that available through the *capabilities()* method, and the level of information available in either may depend on what permissions have been granted.

Another new track method in the specification is *applyConstraints*(), which allows for constraints to be changed on the fly. Since tracks impose constraints on sources and these constraints can be changed dynamically, it is possible for a track to become *overconstrained*. This is

even more likely in the case where multiple tracks are associated with the same source, since each track has the opportunity to impose constraints on the source that may conflict. The specification discusses this topic fairly extensively. The main thing to realize is that a track can become *overconstrained* at any time, in which case an *overconstrained* event will be generated for the track and the track will be muted. This event can of course be caught and handled.

5.3.2.3 MediaStream Interface

This document also defines the *MediaStream* interface, an API for creating objects representing streams of media data. The following subsections describe this interface.

5.3.2.4 MediaStreamTrack Interface

The foundational unit of the API is a *MediaStreamTrack*. This track represents the media of a single type that a single device or recording could return. A single stereo source or a 6-channel surround sound audio signal could be treated as a single track, even though both consist of multiple channels of audio. Note that the specification does not define a means to access or manipulate media at the channel level, although it does roughly define channels as having a "well known relationship to each other". From a practical standpoint, the contents of a track are defined in the WebRTC document as "intended to be encoded together for transmission as, for instance, an RTP payload type." In other words, the channels of a track are treated together as a single unit when being transported using a Peer Connection, and even locally with respect to being enabled/disabled or muted.

Until very recently there has been no way to create a *MediaStreamTrack* directly; there was no constructor for this object. Although no browser yet implements this, the specification does now define the *AudioMediaStreamTrack* and *VideoMediaStreamTrack* subclasses, both of which have constructors. At some point soon it is likely that the *getUserMedia()* method will accept as an input parameter one or more newly-created tracks and/or streams that will be attached to the media being acquired. The other (currently-implemented) means of creating tracks will be covered in the following section describing media streams. An interesting aspect of a *MediaStreamTrack* is that it is essentially just a handle to an underlying browser-managed media source. As such, it is possible for a *MediaStreamTrack* object to become disassociated from its underlying source. Additionally, different

MediaStreamTrack objects can represent the same media source, which will be explained more fully in the following section describing *MediaStream* objects. There are two ways in which tracks can have their media suspended – via muting and via disabling. The muting/unmuting of a track is something done by the user and/or browser, indicating that the track's underlying media source is temporarily unable to provide media. This can happen, for example, if the end user has suspended permission to use the media source by clicking a mute button in the browser chrome or toggling a switch on the side of their phone. In general, the application does not have control over when a track is muted. It can, however, check the value of the track's *muted* attribute. When muted, an audio track will have silence and a video track will show blackness. Separately, tracks can be disabled individually by setting the track object's *enabled* attribute to *false*. Both of these attributes are separate from the *readyState* attribute of a track that indicates its status – *new*, *live* or *ended*. A new track is one which has not yet been connected to media, while an ended track is one whose source is not providing and can never again provide more data. This could occur, for example, if a camera in use is then unplugged. A live track is one that could produce media. Since these attributes are independent, as an example one could perhaps have a track that is *live*, *unmuted*, and *disabled*.

5.3.2.5 Creation of Media Streams

A *MediaStream*, then, is a collection of *MediaStreamTrack* objects. Aside from the too-new-to-be-implemented-yet explicit track creation described in the section above, there are three ways to create these *MediaStream* objects – by requesting access to local media by copying tracks from an existing *MediaStream*, or by receiving a new stream using a Peer Connection. Currently the only way to request and access local media is through a *getUserMedia()* call, although in the future there may be other methods, say, to stream from a local file. One can "copy" the tracks of an existing *MediaStream* object into a new one via the *MediaStream* constructor, which takes as an argument either an existing *MediaStream* object or a list of *MediaStreamTrack* objects. A somewhat recent addition is the ability to specify no arguments at all, along with the ability to add tracks to (*addTrack*()) and remove tracks from (*removeTrack*()) an existing *MediaStream*, but this is not yet widely supported. The *clone*() method may also be used to duplicate a stream (and all its tracks) for those who find passing a *MediaStream* to a constructor offensive. Note that in a derived media stream (i.e., created

from other *MediaStream*s) the elements of the array argument to the *MediaStream* constructor need not all come from the same existing *MediaStream* object; mixing and matching is allowed. Tracks may also be of different types, with both audio and video allowed within the same *MediaStream* object. Regardless of how the *MediaStream* object is created, a key characteristic is that all of the tracks within the *MediaStream* object will be synchronized when rendered. However, tracks are not ordered within a stream, and any attempt to add a duplicate track will be silently ignored. Since each track has an *id* which is preserved across Peer Connections, it is possible to match up tracks in a *MediaStream* which is sent over a Peer Connection by checking *id*s after calling *getAudioTracks*() or *getVideoTracks*() or even by directly requesting a track via *getTrackById*(). Similar to the *ended* status of tracks, a *MediaStream* has a Boolean *ended* attribute indicating whether the *MediaStream* is finished. A *MediaStream* is considered finished if all of its tracks are *ended*.

Since the first edition of this book was published, there has been a key change in thought about how local media should be treated. Originally a call to *getUserMedia*() returned a special kind of *MediaStream,* called a *LocalMediaStream*, that had some unique characteristics. Most importantly, the tracks of a *LocalMediaStream* were tied tightly to user permissions. The user agent was required to obtain/confirm user permission before returning a *LocalMediaStream* (see Figure 8.1), and any user revocation of permission to use a particular media track set the status of the track to *ended* not only for the *LocalMediaStream*, but for all tracks derived from it in other *MediaStream*s. The currently-described behavior is similar, but a little clearer. The specification now talks about all live tracks being associated with a logical source provided by the browser. A source is only a theoretical entity in that the browser makes a source available in whatever way it wishes to based on devices that are available, although in practice there is likely to be a close correlation between physical devices and browser-provided sources. In this new theoretical model, permissions are tied to sources and each track has its own association to a specific source, although multiple tracks can be associated with the same source. The consequence of this model is that there is no longer a need for a special *LocalMediaStream*. Of course, the new ability to apply constraints to existing tracks can result in a source being *overconstrained*. Please see section 5.3.2.2 for more details on track constraint modifications. Whereas before there was a *stop*() method on the *LocalMediaStream*, now there is only a *stop*() method on a *MediaStreamTrack*.

5.3.2.6 Media Streams Example

For the media sources of Figure 1.9, one arrangement of tracks and streams is shown in Figures 5.1 through 5.4. In this example, Figures 5.1 and 5.3 show the media flowing from browser M to browser L, and Figures 5.2 and 5.4 show the reverse. The first thing to notice is that the media flow is not symmetrical, with completely different streams flowing in one direction than in the other.

Beginning with Figure 5.1, four different media sources are available for use on this device: microphone audio, application sharing video, front camera video, and rear camera video. Through four separate calls to *getUserMedia()*, four separate local *MediaStream*s are created. The JavaScript application code then creates three new *MediaStream* objects by mixing and matching. Specifically, the first *MediaStream* contains new audio and video *MediaStreamTrack*s pulled from the *MediaStreamTrack*s in the local Audio (microphone audio source) and Presentation Video (application sharing video source) *MediaStream*s.

Similarly, the second *MediaStream* contains new *MediaStreamTrack*s for the microphone audio and the front camera video derived from the originals in the local *MediaStream*s. Finally, the third *MediaStream* contains new *MediaStreamTrack*s for the microphone audio and the rear camera video derived from the originals in the local *MediaStream*s.

Streams from Browser M to Browser L

At this point the application code now has four local *MediaStream* objects and three derived *MediaStream* objects. Notice that the microphone audio track has been duplicated three times. The application then decides to send the three derived *MediaStream* objects to the remote peer (on browser L). Figure 5.3 shows how these streams appear after coming across the Peer Connection from browser M. Notice what the application code on browser L then does with these. It pulls the audio track from the Presentation *MediaStream* and sends it to the speaker, it pulls the video track from the same *MediaStream* and sends it to one window on the display, it pulls the video track from the Presenter *MediaStream* and sends it to another window, and it pulls the video track from the final *MediaStream* and sends it to yet another window. Although not shown, the application can figure out which stream is which because each stream has a unique id that is

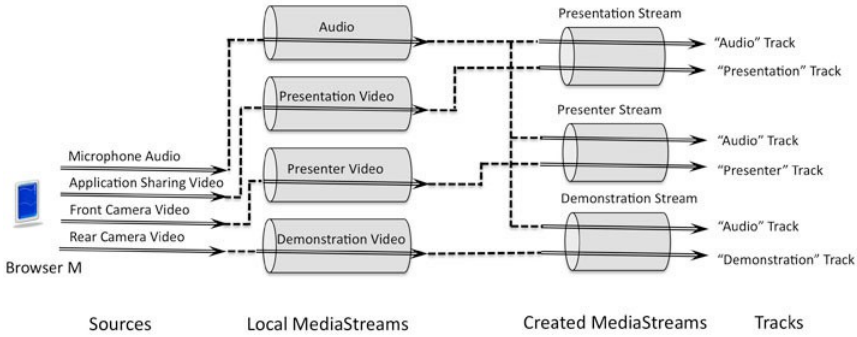


Figure 5.1 Browser M Sending Sources, Streams, and Tracks

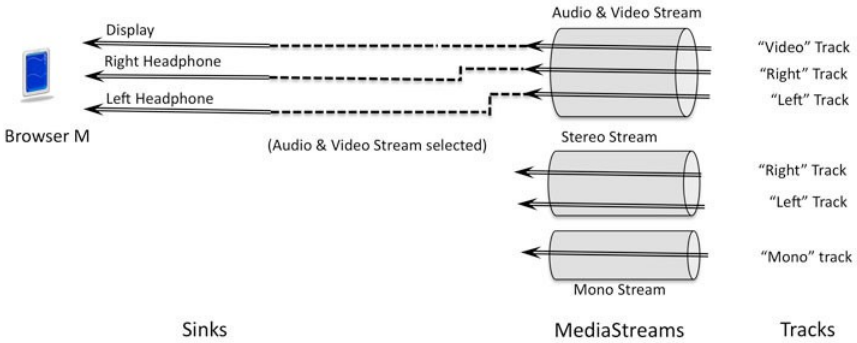


Figure 5.2 Browser M Receiving Tracks, Streams, and Sinks

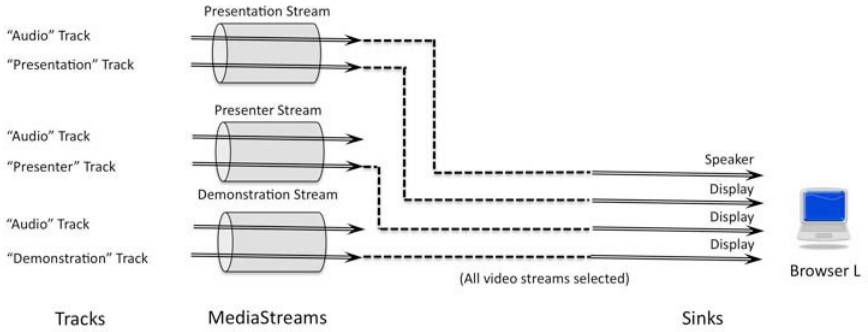


Figure 5.3 Browser L Receiving Tracks, Streams, and Sinks

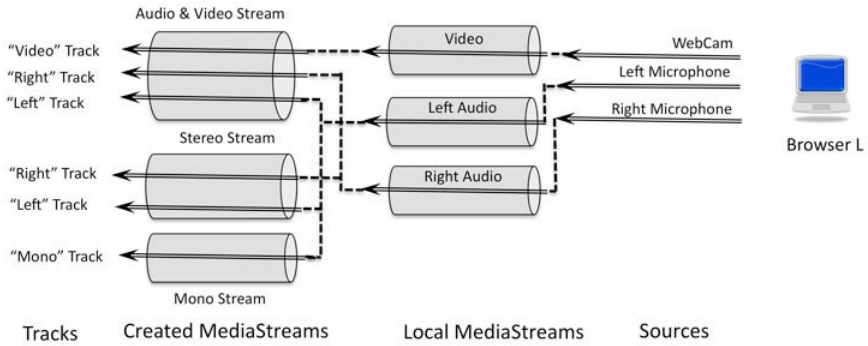


Figure 5.4 Browser L Sending Sources, Streams, and Tracks

preserved across the Peer Connection. Of course, Browser L would need to be told which id is the Presentation stream, which the Presenter, etc. The code on Browser M would likely send this information over the signaling channel.

Streams from Browser L to Browser M

A similar process occurs for the streams originating on browser L that are sent to browser M. In Figure 5.4, on the right-hand side, we can see that there are three media sources being used – a webcam video, a left microphone, and a right microphone. As in the other direction, each is obtained through a separate call to *getUserMedia()* resulting in three separate local *MediaStream* objects. These are then mixed into three new derived *MediaStream* objects. The first contains all three tracks, the second contains both audio tracks to make a stereo audio stream, and the third contains only the right audio track to make a mono audio stream. These new *MediaStream* objects are the ones sent over the Peer

Connection to browser M.

Figure 5.2 shows how these streams appear after coming across the Peer Connection from browser L. The JavaScript application code in browser M decides to create two new *MediaStream*s – one containing the video from the only *MediaStream* containing a video track, and one containing the audio from one of the three streams, the choice of which is controlled by a button on the display.

5.3.3 "MediaStream Capture Scenarios"

This document [3] is the requirements and use cases document for media capture and media streams. It contains requirements in the following four categories: permissions, local media, remote media, and media capture. The requirements are still under review and will likely soon include requirements from the IETF RTCWEB use cases and requirements draft document (see Section 8.3.2) as well.

5.4 Related Work

There are areas in W3C that are under active discussion, either for inclusion into one of the WebRTC specifications or but have not yet been included in either an editor's draft or public draft of a specification. For more information on the W3C standards process, see Appendix A.

5.4.1 MediaStream Recording API Specification

The WebRTC APIs provide a handle to a device track but no access to the content (data) itself. It is of course possible for the contents of the track to be sent over a Peer Connection to an entity that can record the content, but there are valid use cases for having direct access to the data. One such use case is for Interactive Voice Response systems, the systems that answer the phone and play a voice menu. Many such IVR systems allow callers to use their voices to speak their menu choices. A recording API would allow the JavaScript code access to the content for saving, additional manipulation, or posting to an automatic speech recognition system. Early Working Drafts of this document are now available [4].

5.4.2 Image Capture API

During the discussion of what constraints should be defined in the Media Capture and Streams specification, it became clear that there are many kinds of processing and control that application developers might want over video *MediaStream*s. A proposal for controlling cameras and

capturing still images from them [5] is likely to become an official Working Draft at some point.

5.4.3 Futures

There have been requests recently to revise all of the API methods that use callbacks to be futures instead. Roughly speaking, in computer science a *future* is a variable that does not yet have a value but will at some unspecified point in the future. The futures being defined for the HTML DOM [6] have a non-blocking semantics in which handlers can be defined to take action when the future obtains a value. As an example, instead of calling

createOffer(gotSDP(), didntGetSDP())

you would call

createOffer().done(gotSDP(), didntGetSDP())

This minor syntactic difference, along with a few other features, could allow for combinations of asynchronous results to be checked for in a convenient manner. For example, if both *getUserMedia()* and *createSignalingChannel()* were defined as futures,

```
Future.every(getUserMedia({video:true, audio:true}),
             createSignalingChannel()
            ).done(executeMe())
```

would wait until both had completed before calling *executeMe()*. Although this programming paradigm has been in use in a variety of languages for many years, it is new to HTML standards. For this reason, and because a strong effort is being made to finalize the technical content of the WebRTC specifications at this time, this relatively new feature may not be incorporated into the first versions of the WebRTC standards. Note that no official W3C Working Drafts yet exist for DOM Futures, so syntax, content, and even the name can change. An alternate name being considered, for example, is Promises (rather than Futures).

5.4.4 Media privacy

Very shortly some early draft text is likely to appear in the Media Capture and Streams specification to address privacy of captured media. There is discussion about having controls to restrict media content from being used in local elements, Peer Connections, or specific Peer Connections. The syntax and, in fact, all the details are still being worked on.

5.4.5 MediaStream inactivity

The behavior of a *MediaStream* when attached to an HTML element

such as <*video*> is still under discussion. In particular, since it is possible to add a new track to a *MediaStream* even after its existing tracks have ended, the behavior in this case needs to be specified. One proposal under consideration is to remove the ability for a *MediaStream* to end and replace it with the notion of being *active* or *inactive*. In this case, when all the tracks in a *MediaStream* are *ended* the *MediaStream* would become inactive. Adding new *live* tracks would make it active again.

5.4.6 Risks, Critiques, and Disagreements

There are a couple miscellaneous, but important, considerations not discussed elsewhere in this book. Here is a very brief summary:

1) There is a risk that not all of the major web browser vendors will support WebRTC. In particular, Apple (creator of the Safari web browser) has not been very actively involved in the creation of the standard, making it difficult to predict whether or how completely the Safari web browser will support the eventual standard. Microsoft (creator of the Internet Explorer web browser), on the other hand, has expressed continued strong interest in the WebRTC work. However, representatives from Microsoft have conveyed significant concerns with the current direction of the standard. It is thus difficult to predict the level of support we will see for the eventual standard in the Internet Explorer web browser.

2) There is continued disagreement about which codecs should be mandatory-to-implement. To guarantee at least basic audio and video interoperability between any pair of compliant browser implementations, the two implementations must support at least one audio codec and one video codec in common. To guarantee this, the working groups are considering mandating support for one or more audio and one or more video codecs. However, the specific choice of codecs is heavily disputed. We will provide more details on this as the various working groups come to better agreement. See Section 8.4.2 for a summary of the codecs under consideration.

5.5 References

[1] Public Working Draft: http://www.w3.org/TR/webrtc, Editors' Draft: http://dev.w3.org/2011/webrtc/editor/webrtc.html

[2] Public Working Draft: http://www.w3.org/TR/mediacapture-streams, Editors' Draft: http://dev.w3.org/2011/webrtc/editor/getusermedia.html

[3] Public Working Draft: http://www.w3.org/TR/capture-scenarios, Editor's Draft: http://dvcs.w3.org/hg/dap/raw-file/tip/media-stream-capture/scenarios.html

[4] http://www.w3.org/TR/mediastream-recording

[5] http://gmandyam.github.io/image-capture

[6] http://dom.spec.whatwg.org/#futures

6 WEBRTC PROTOCOLS

There are a number of protocols related to WebRTC. The most important ones are listed in Table 6.1 below. Their usage is discussed in this chapter. They are shown in the protocol stack of Figure 6.1.

Protocol	Use	Specification
HTTP	Hyper-Text Transport Protocol	RFC 2616
WebSocket	Socket between Web Browser and Server	RFC 6455
SRTP	Secure Real-time Transport Protocol	RFC 3711
SDP	Session Description Protocol	RFC 4566
STUN	Session Traversal Utilities for NAT	RFC 5389
TURN	Traversal Using Relays around NAT	RFC 5766
ICE	Interactive Connectivity Establishment	RFC 5245
TLS	Transport Layer Security	RFC 5246
TCP	Transmission Control Protocol	RFC 793
DTLS	Datagram Transport Layer Security	RFC 4347
UDP	User Datagram Protocol	RFC 768
SCTP	Stream Control Transport Protocol	RFC 2960
IP	Internet Protocol, version 4 and version 6	RFC 791, RFC 2460

Table 6.1 WebRTC Protocols

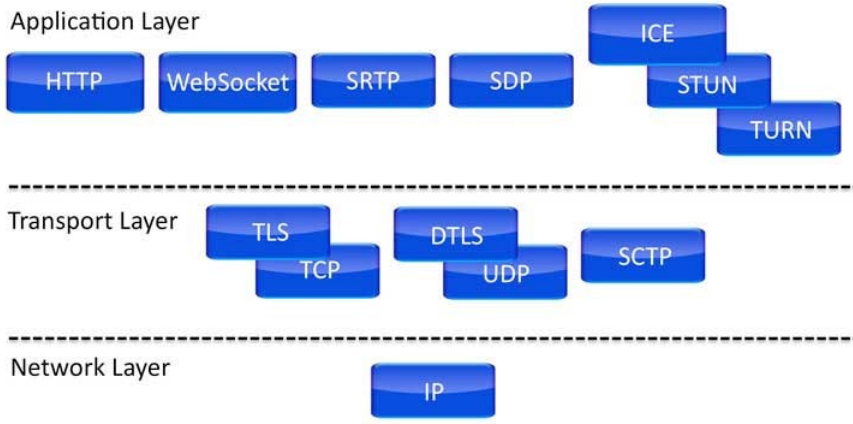


Figure 6.1 Protocols in WebRTC

6.1 Protocols

The preceding sections have discussed the WebRTC APIs as standardized by W3C. Web developers will interact directly with these APIs in their applications and web sites to add communication capabilities. The following sections will discuss the protocols utilized by WebRTC. These protocols are the "bits on the wire" that allow two browsers to communicate, or a browser and a server to communicate. A web developer in general will never directly interact with protocols, as the default settings and configurations of the protocols will usually meet their needs. However, in some cases, especially when a WebRTC client is communicating with a non-WebRTC client, some configuration and knowledge of the protocols used by WebRTC is necessary. Additionally, if there arises a need to adjust the *RTCSessionDescription* object in the WEBRTC API, such as may occur if there are SDP interoperability problems between user agents, the application author will need to have a deeper understanding of how the negotiation works. In any case, a basic understanding of the protocols used by WebRTC is useful for the developer – this is provided in the first section. A telephony developer who wants to utilize WebRTC, on the other hand, will need to have a detailed understanding of the protocols used. For them, subsequent sections of this book contain a detailed description of how they work together.

6.2 WebRTC Protocol Overview

6.2.1 HTTP (Hyper-Text Transport Protocol)

Of course, WebRTC uses HTTP, Hyper-Text Transport Protocol [RFC2616]. This is the protocol of the World Wide Web and is used between a web browser and a web server. WebRTC uses HTTP the same as any web application. As such, no specific knowledge of HTTP is needed. The current version of HTTP is 1.1. There is work in the IETF to define the next version of HTTP, known as 2.0. This protocol will likely increase the speed and efficiency of web downloads and applications. WebRTC will be able to use this and any other future version of HTTP. See Section 4.3.3 for examples of how to use HTTP for the WebRTC signaling channel.

6.2.2 WebSocket Protocol

The WebSocket protocol [RFC6455] allows a browser to open additional bi-directional TCP connections to a web server. The connection opening is signaled using HTTP and has similar security properties to the HTTP

web session, and can reuse existing HTTP infrastructure. This avoids HTTP polling and the opening of multiple HTTP connections between a browser and web server. The browser indicates the application using the WebSocket in the opening. This is known as the WebSocket sub-protocol. The SIP WebSocket sub-protocol or XMPP WebSocket sub-protocol are examples, described in Sections 4.3.6 and 4.3.7.

6.2.3 RTP (Real-time Transport Protocol) and SRTP (Secure RTP)

The most important protocol used by WebRTC is the Real-time Transport Protocol, or RTP [RFC3550]. WebRTC uses only the secure profile of RTP or Secure RTP, SRTP [RFC3711]. SRTP is the protocol used to transport and carry the audio and video media packets between WebRTC clients. The media packets contain the digitized audio samples or digitized video frames generated by a microphone or camera or application, and are rendered using a speaker or display. A successful setup of a Peer Connection, along with a complete offer/answer exchange, will result in an SRTP connection being established between the browsers or a browser and a server, and an exchange of media information.

SRTP provides information essential for successfully transporting and rendering media information: the codec (coder/decoder used to sample and compress the audio or video, the source of the media (the synchronization source or SSRC), a timestamp (for correctly timed play-out), sequence number (to detect lost packets), and other information needed for playback. For non-audio or video data, SRTP is not used. Instead, a call to the *RTCDataChannel* API will result in a data channel being opened between the browsers allowing any arbitrarily formatted data to be exchanged.

The RTP header is shown in Figure 6.2.

- **Version (V)** - (2 bits) Set to 2
- **Padding (P)** - (1 bit) Padding octets are present in payload
- **Extension (X)** - (1 bit) Header extension present
- **CSRC Count (CC)** - (4 bits) Number of CSRC identifiers present – 0 for non-mixed media
- **Marker (M)** – (1 bit) Profile defined marker
- **Payload Type (PT)** – (7 bits) Codec in payload
- **Sequence Number** - (16 bits) Count incremented for each packet sent
- **Timestamp** - (32 bits) Sampling instant for first octet in payload
- **Synchronization Source (SSRC)** - (32 bits) Identifies the sender of the packet, unique for each session
- **Contributing SSRC (CSRC)** - (32 bits) Optional contributing sources in payload due to mixing

Figure 6.2 RTP Header

The set of extensions to RTP used by WebRTC is shown in Table 6.2.

Extension	Description	Reference
SAVPF Profile	Extended Secure Audio and Video Profile with Feedback uses Secure RTP and early RTCP feedback	Section 9.1.4
Multiplexing RTP and RTCP	RTCP received on same UDP port as RTP	Section 9.1.5
Multiplexing Audio & Video	All media received on same UDP port, media type (audio or video) identified by SSRC and Payload Type	Section 8.5.6
Symmetric RTP	RTP media sent and received from same UDP port to help NAT traversal	Section 9.3.2
Congestion Avoidance & Control	Avoiding Congestion by stopping sending RTP or adjusting bandwidth usage	Section 8.5.10

Table 6.2 WebRTC RTP Extension Summary

6.2.4 SDP (Session Description Protocol)

WebRTC session descriptions are described using the Session Description Protocol, SDP [RFC4566]. An SDP session description (encoded as an *RTCSessionDescription* object) is used to describe the media characteristics of the Peer Connection. There is a long and

complicated list of information that must be exchanged between the two ends of the SRTP session so that they can communicate. API calls to *RTCPeerConnection* will result in an SDP session description, a set of data formatted in a particular way, being generated by the browser and accessed using JavaScript by the web application. An application that wants to have tight control over the media may make changes to the session description before sharing it with the other browser. When changes are made to a Peer Connection, this will result in changes to the session description which the two peers will exchange. This is known as an offer/answer exchange (see Section 2.2.1. Any developer wishing to have fine-grained control over the media sessions needs to understand SDP.

Both SRTP and SDP are protocols standardized by the IETF and widely used by Internet Communications devices and services on the Internet, such as Voice over IP (VoIP) phones, clients, and gateways, and video conferencing and collaboration devices. As a result, communication between one of these devices or clients and a WebRTC client is possible. However, few VoIP or video devices or clients today support the full set of capabilities and protocols of WebRTC. These devices will need to be upgraded to support these new protocols, or a gateway function used between the WebRTC client and the VoIP or video client to do the conversion. Telephony and Internet Communication developers can use the descriptions of these protocols in this book to guide their development efforts with WebRTC compatible clients or gateways.

6.2.5 STUN (Session Traversal Utilities for NAT)

Session Traversal Utilities for NAT, STUN [RFC5389], is a protocol used to help with NAT traversal. In WebRTC, a STUN client will be built into the browser user agent, and web servers will run a STUN server. STUN test packets are sent prior to session establishment to allow a browser to learn if it is behind a NAT and to discover the mapped addresses and port. This information is then used to construct candidate addresses in ICE “hole punching”. STUN can be transported over UDP, TCP, or TLS. The port number for STUN can be determined using a DNS SRV lookup; the default UDP port for STUN is 3478.

The format of the STUN header is shown in Figure 6.3. The first two bits are set to zero to help distinguish a STUN packet from an RTP packet (which will have the first two bits set to zero and one). The next 14 bits are the message type, which includes the class and method. The next 16 bits contain the message length, which has the entire size of the STUN packet including the header and any padding. The next 32 bits are

a “magic cookie” which is a unique string to aid in the identification of STUN packets. The value of the magic cookie is 0x2112A442 in hexadecimal. The last field in the header is the Transaction ID, which is a cryptographic random number used to correlate a response to a request.

- **First two bits are 0's** – (2 bits) Differentiate from RTP
- **Message Type** - (14 bits) set to indicate end of event
  - Identifies the Class and Method of a message
- **Message Length** - (16 bits) Total length of the STUN packet including the 20 octet header.
- **Magic Cookie** – (32 bits) Unique string to aid in identification of STUN packets:
  - Value of 0x2112A442
- **Transaction ID** – (96 bits) Cryptographically random number used to correlate responses to requests

Figure 6.3 STUN Header Format

STUN is a client/server protocol. There are two types of STUN requests: request/response and indication. In a request/response request, the STUN client originating the request expects a response from the STUN server that received the request. When used over UDP, request/response request message are retransmitted using an exponential back-off timer known as RTO (Retransmission Time Out). RTO is configurable, but must be greater than 500ms. The first retransmission is sent after RTO expires, the second after twice RTO, the third after three times RTO, etc. Retransmissions end after a total of seven requests have been sent. In an indication request, no response is expected.

In the core STUN protocol, there is only one method called Binding. Binding is used by a STUN client to create and discover NAT mappings. If a STUN client is behind a NAT, and the STUN server is outside the NAT, sending a STUN Binding request will cause the NAT to create a mapping and assign a public IP address and port. In addition, this creates a filter rule in the NAT about who is permitted to use this mapping to send packets to the private network. When the STUN server receives a STUN Binding request, it records the IP address and port number that it

received the STUN Binding request from. This address and port number is then returned to the client in the STUN Binding response. The STUN client receiving the STUN Binding response compares the IP address and port received in the response to the IP address and port that it sent the request from. If they are the same, there is no NAT between the client and the server. If they are different, there is at least one NAT, and the client is able to learn the IP address and port assigned by the outer most NAT. Note that there could be multiple NATs between the STUN client and server, but only information about the outer most NAT is learned.

A STUN Binding request sent as an indication can used to prevent NAT mappings from timing out if sent at regular intervals. The indication request will cause the NAT to reset its UDP timer. As long as the interval of sending out the STUN Binding indications is less than the shortest NAT UDP timer between the client and server, the NAT mapping will be maintained.

STUN has two authentication mechanisms: short term authentication and long term authentication. Short term authentication uses a username/ password which is valid for a single session. This approach is used by ICE to make authentication unique for each set of connectivity checks. Long term authentication uses a challenge/response mechanism which allows a long term credential to be used. For example, a SIP authentication credential can be reused.

The format of STUN attributes is shown in Figure 6.4. Attributes are optional after the STUN header. Attributes are encoded as TLV (Type, Length, and Value). Type and Length are two octets long, but the Value is variable length. The Value is always padded out so that the length is a multiple of four octets.

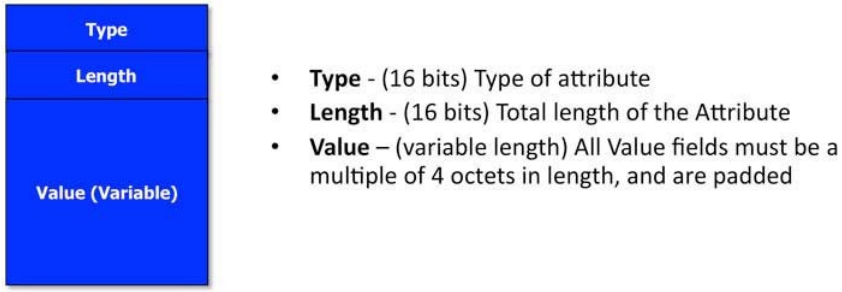


Figure 6.4 STUN Attribute Format

The list of STUN attributes in the core protocol is shown in Table 6.3. In Binding responses, the IP address and port number seen by the STUN server will be returned in a *XOR-MAPPED-ADDRESS* attribute. The

XOR-MAPPED-ADDRESS attribute obfuscates the IP address and port number by performing an exclusive OR operation on it. This is because some NATs have functions known as general application layer gateways (ALGs) which look for IP addresses and ports associated with NAT mappings and rewrite them on the fly. (These ALGs can cause major problems with many protocols and should be disabled or removed whenever possible from a network). The presence of these ALGs can be determined if the server includes both attributes and the information as seen by the client is different in the two attributes. The *MAPPED-ADDRESS* attribute is only used if backwards compatibility with the original STUN protocol is needed.

Attribute	Use
XOR-MAPPED-ADDRESS	Exclusive ORed reflexive transport address of client
MAPPED-ADDRESS	Reflexive transport address of client seen by server
USERNAME	Username as used in message integrity
MESSAGE-INTEGRITY	Keyed message authentication code for STUN message
FINGERPRINT	Cyclic Redundancy Code (CRC) of STUN message
ERROR-CODE	Error code and reason phrase for error responses
REALM	Realm to be used for long-term authentication
NONCE	Nonce to be used for long-term authentication
UNKNOWN-ATTRIBUTES	Unknown attribute in 420 error response
SOFTWARE	Manufacturer and version of STUN for debugging
ALTERNATE-SERVER	Alternative STUN server in 300 error response

Table 6.3 STUN Attributes

Failure responses are carried in a *ERROR-CODE* attribute in a response. The set of STUN error codes is shown in Table 6.4. If a *420 Unknown Attribute* code is used, the *UNKNOWN-ATTRIBUTES* attribute will list the attributes that were not understood. If a *300 Try Alternative* code is used, the *ALTERNATIVE-SERVER* attribute will contain the IP address and port of the other STUN server to be used.

STUN servers do not implement any authentication mechanism. This is because the effort needed to authenticate requests is greater than the level of effort needed to simply respond to the requests.

Error Code	Reason Phrase	Usage
300	Try Alternative	STUN
400	Bad Request	STUN
401	Unauthorized	STUN
403	Forbidden	TURN
420	Unknown Attribute	STUN
437	Allocation Mismatch	TURN
438	Stale Nonce	STUN
441	Wrong Credentials	TURN
442	Unsupported Transport Protocol	TURN
486	Allocation Quota Reached	TURN
487	Role Conflict	ICE
500	Server Error	STUN
508	Insufficient Capacity	TURN

Table 6.4 STUN Error Codes

6.2.6 TURN (Traversal Using Relays around NAT)

Traversal Using Relays around NAT, TURN [RFC5766], is an extension to the STUN protocol that provides a media relay for situations where ICE "hole punching" fails. In WebRTC, the browser user agent will include a TURN client, and a web server, service provider, or enterprise will provide a TURN server. The browser requests a public IP address and port number as a transport relay address from the TURN server. This address is then included as a candidate address in the ICE "hole punching". TURN can also be used for firewall traversal, as described in Section 3.4. The port number for TURN can be determined using a DNS SRV lookup; the default UDP port for TURN is 3478.

TURN can be used to establish relayed transport addresses that use UDP, TCP, or TLS transport. However, the communication between the TURN server and the TURN client (through the NAT) is always UDP.

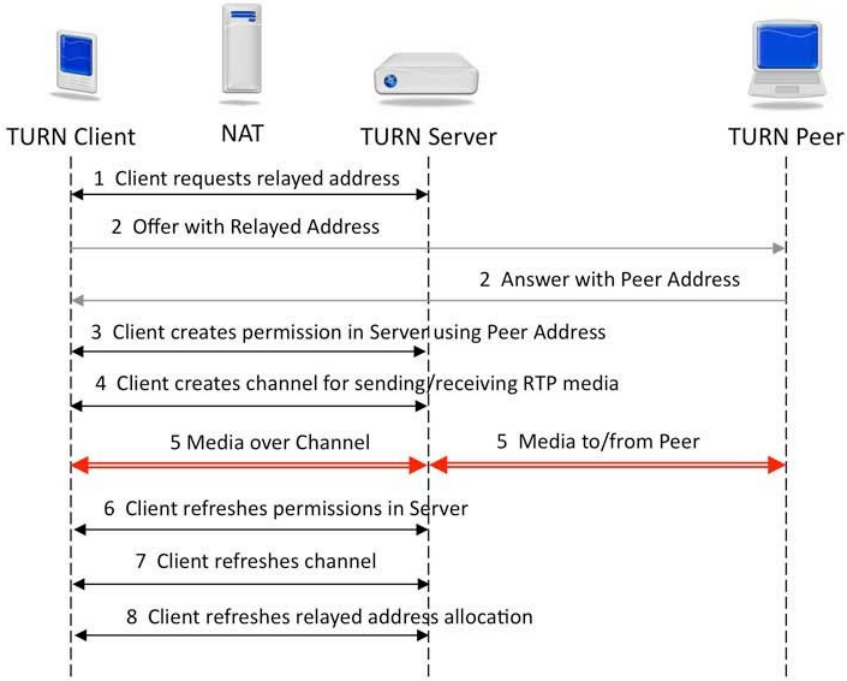
The set of STUN attributes defined for TURN are shown in Table 6.5.

The TURN error codes for STUN are listed in Table 6.4.

Attribute	Use
CHANNEL-NUMBER	Channel number for Channel binding or refreshing
LIFETIME	Allocation, Channel, or Permission must be refreshed within this time or it will expire in server
XOR-PEER-ADDRESS	Peer's Server Reflexive Address which has permission to use the Relayed Address
DATA	Contains relayed application data in Send and Data
XOR-RELAYED-ADDRESS	Allocated relay address in Allocate response
EVEN-PORT	Request even port for RTP and next higher for RTCP
REQUESTED-TRANSPORT	Requested transport between relay and peer, usually UDP
DONT-FRAGMENT	Set don't fragment bit for application data sent to peer
RESERVATION-TOKEN	Uniquely identifies a relayed transport address

Table 6.5 TURN STUN Attributes

TURN defines a whole new set of STUN methods, in addition to the Binding, shown in Table 6.6. The *Allocate* method is used to request a relayed transport address from the STUN server. The *Refresh* method is used to refresh and keep alive an existing transport allocation. The *CreatePermission* method is used to set filtering rules on the relayed address, similar to those used by NATs. TURN has two ways in which data can be relayed through the TURN server. One is to use the *Send* and *Data* methods, where the relayed data is carried in STUN message. Another method is to establish a channel between the TURN client and the TURN server, and the data is sent using a *ChannelData* message which does not use the STUN header and the 36 octets of overhead. For audio or video media, channel is usually used. The *ChannelBind* method is used to establish the channel.



1) Client requests relayed address allocation from TURN Server and receives in response
2) In Offer, Client sends Relayed Address to Peer. In Answer, Peer sends Peer Address to Client
3) Client creates permission in Server using Peer Address to Server
4) Client creates channel for sending and receiving RTP media
5) Client sends and receives media over channel. Peer sends and receives media with TURN Server
6) Client refreshes permissions in Server
7) Client refreshes channel with Server
8) Client refreshes relayed address allocation with Server

Figure 6.5 High Level TURN Call Flow

- **Channel Number** - (16 bits) `CHANNEL-NUMBER` attribute from CreateChannel request
- **Length** - (16 bits) Total length in octets of the Application Data field
- **Application Data** – (variable length) The application data sent to or received from the Peer

Figure 6.6 TURN ChannelData Format

STUN Method	Use	Spec
Binding	Create and maintain NAT mapping	STUN
Allocate	Client receives allocated relayed transport address	TURN
Refresh	Regular keep alives for allocation from client	TURN
Send	Application data from client to server (not used for RTP)	TURN
Data	Application data from server (peer) to client (not used for RTP)	TURN
CreatePermission	Sets permissions in the server to allow a particular IP address to use a relayed address	TURN
ChannelBind	Create or refresh a channel for exchange of RTP data	TURN

Table 6.6 STUN Methods

TURN servers commonly use the STUN long-term authentication. A TURN request can be challenged with a *401 Unauthorized* response containing a *REALM* and *NONCE* attribute. These are used by the TURN client to resend the request with the *USERNAME* attribute containing the username of the user, and the *MESSAGE-INTEGRITY* attribute, which is keyed using an MD5 hash of the concatenation of the username, realm, and password. This is similar to (but not identical to) the challenge/response authentication of SIP Digest and HTTP Digest.

6.2.7 ICE (Interactive Connectivity Establishment)

Another key protocol used in WebRTC is Interactive Communication Establishment, or ICE [RFC5245] ICE has two important functions:

1) ICE allows WebRTC clients to exchange media across devices that perform Network Address Translation or NAT.
2) ICE provides a verification of communication consent. This means that media packets will only be sent to a browser that is expecting the traffic. A malicious web application might try to trick a browser into sending media to an Internet host that is not a party to communication. This type of attack is known as a Denial of Service or DOS flooding attack. ICE will prevent this from succeeding since media will never be sent unless the ICE exchange completes successfully.

ICE uses a technique known as "hole punching" (see Section 3.3), which was pioneered by online gamers who needed to exchange packets directly between PCs playing multiplayer games despite the presence of NATs between. ICE is run at the start of a session prior to establishing the SRTP session between the browsers. It is also used for the non-media data channel establishment as well.

The operation of ICE is described in Chapter 3 in detail. This section will discuss the way ICE utilizes the STUN and TURN protocols.

ICE utilizes the short term authentication mechanism of STUN. A username and password are randomly generated for each session, and exchanged in the offer/answer exchange. Each side contributes half of the session username (carried in the *a=ice-ufrag* SDP attribute, and a password) and one password. The session username is the concatenation of each of the username fragments, and the key used is the password sent by the other ICE Agent. The *MESSAGE-INTEGRITY* attribute contains a keyed message authentication code (keyed HMAC or keyed hash function) over the entire STUN message.

Gathered candidate addresses are categorized in two ways. All non-host candidates have a base. The base is the IP address and port that was used to generate the candidate. For a reflexive candidate, it is the host address from which the STUN check was sent. For a relayed candidate, the base is the IP address and port where the relayed traffic will be forwarded. Each candidate also has a foundation associated with it. A foundation is a unique string used to group candidates that are the same connection address (IPv4 or IPv6), base IP address, transport protocol (usually UDP), and were derived using the same STUN or TURN server. For example, candidates that include both IPv4 and IPv6 address, but all use UDP transport and utilize the same STUN and TURN server, would represent two foundations.

ICE provides a keep-alive functionality by sending packets at periodic intervals. There is a proposal (see Section 8.5.5) to turn this into a continuing consent functionality by requiring a response and restarting ICE if the response does not come.

ICE peer-to-peer connectivity checks are sent over the same ports as the RTP media session. To help de-multiplex the protocols, ICE requires the use of the STUN *FINGERPRINT* attribute. The *FINGERPRINT* attribute contains a Cyclic Redundancy Code across the entire STUN message. ICE also requires the use of the *PRIORITY* STUN attribute. The *PRIORITY* attribute carries the chosen priority of the candidate, and is used by the other ICE Agent to determine the priority of any peer reflexive candidates generated from this connectivity check.

Table 6.7 shows the new STUN attributes defined for ICE. The *USE-CANDIDATE* attribute is used by the Controlling ICE Agent to select a candidate pair for use in a connectivity check. The Controlled ICE Agent includes the attribute in a response to confirm the chosen candidate pair. The *ICE-CONTROLLED* attribute is included in all connectivity checks by the Controlled ICE Agent. The *ICE-CONTROLLING* attribute is included in all connectivity checks by the Controlling ICE Agent. ICE includes mechanisms for choosing which ICE Agent is Controlling and which is Controlled. Should both Agents choose the same role, they will discover this using these attributes, send a *487 Role Conflict* response, and choose again.

Attribute	Use
USE-CANDIDATE	Controlling ICE Agent indicates candidate is to be used
ICE-CONTROLLED	Attribute included by Controlled ICE Agent
ICE-CONTROLLING	Attribute included by Controlling ICE Agent

Table 6.7 STUN Attributes defined for ICE

To select a particular candidate pair to be used for media, the Controlling ICE Agent sends a connectivity check containing the *USE-CANDIDATE* attribute. The Controlled ICE Agent responds to the connectivity check echoing the *USE-CANDIDATE* attribute. If the Controlling ICE Agent wants to choose the best candidate pair to use, the Controlling ICE Agent would complete all connectivity checks, then select the highest priority successful pair. If the Controlling ICE Agent wants to use the first successful candidate pair, the Controlling ICE Agent would include the *USE-CANDIDATE* attribute in every connectivity check sent. The first successful connectivity check received

by the Controlled ICE Agent will contain the *USE-CANDIDATE* attribute and the first successful response received by the Controlling ICE Agent will also contain the attribute.

ICE used in WebRTC will support the Trickle ICE extension discussed in 8.5.1.

6.2.8 TLS (Transport Layer Security)

Transport Layer Security, TLS [RFC5246], whose older versions were known as Secure Sockets Layer or SSL, is a shim layer between TCP and the application that provides confidentiality and authentication services. Confidentiality is provided by encrypting the "over the wire" packets. Authentication is provided using digital certificates. Secure web browsing today (HTTPS) utilizes only TLS transport. WebRTC can take advantage of TLS for signaling and user interface security. There is also a version of TLS that runs over UDP, called Datagram TLS (DTLS, see Section 6.2.10), and a version that can be used to generate keys for SRTP known as DTLS-SRTP [RFC5764].

6.2.9 TCP (Transmission Control Protocol)

Transmission Control Protocol, TCP [RFC793], is a transport layer protocol in the Internet Protocol stack that providers reliable transport with congestion control and flow control. TCP is used to transport web (HTTP) traffic, but is not suitable for carrying real-time communications traffic such as RTP, as retransmissions used to implement reliability introduce unacceptably long delays. Like UDP, TCP uses a concept of ports, a 16-bit integer number, to separate flows and protocols. TCP is provided by the operating system under the browser.

6.2.10 DTLS (Datagram TLS)

Datagram TLS [RFC6347] is a version of TLS that runs over UDP. The same confidentiality and authentication properties are provided. UDP is easier to get through NAT and can be better suited to peer-to-peer applications.

6.2.11 UDP (User Datagram Protocol)

User Datagram Protocol, UDP [RFC768], is a transport layer protocol in the Internet Protocol stack that provides an unreliable datagram service for upper layers. UDP is commonly used to transport small, short packet exchanges (for example DNS, Domain Name Service packets) or to transport real-time media such as RTP. UDP provides for very fast and

efficient exchange of information; however, users of UDP must deal with possible packet loss. In addition, UDP has no congestion control, so users must be sensitive to packet loss and congestion to avoid overloading Internet connections. Like TCP, UDP uses a concept of ports, a 16-bit integer number, to separate flows and protocols.

Most Internet applications use a reliable transport, such as TCP, Transmission Control Protocol, in which lost packets are automatically retransmitted. Web browsing, email, and streaming audio and video use reliable transport. Received packets are acknowledged, and a lack of an acknowledgement after a certain amount of time triggers a retransmission of the packets until an acknowledgement is received. Real-time communication cannot take advantage of this type of reliable transport due to the time delay involved in detecting packet loss and receiving retransmitted packets. Lost packets in a web page load could result in a page taking an extra second or two to fully load. A real-time communication session cannot pause for a second or two in the middle of a voice conversation, or freeze playing back video for a second while awaiting the retransmission of the missing information. Instead, real-time communications systems just have to do the best they can when information is lost. Techniques to cover up loss or minimize the effects are known as packet loss concealment or PLC.

The average packet loss in general over the Internet is extremely low, on the order of fractions of a percent. Although occurring infrequently, packet loss occurs in a burst, resulting in high packet loss over short intervals. The ability to handle these short-duration loss events has a major impact on the perceived quality of a communication system. Advanced codecs, especially the Opus audio codec (see Section 8.5.8), are designed to provide a good user experience, even during high packet loss. In addition, real-time feedback from the receiver of media also provides the ability to reduce bandwidth or resolution during packet congestion, providing a better user experience and sharing bandwidth fairly with other Internet users.

UDP is provided by the operating system under the browser.

6.2.12 SCTP (Stream Control Transport Protocol)

Stream Control Transport Protocol, SCTP [RFC4960], is a transport layer that provides reliable or unreliable transport across IP along with congestion control and multiple streams in a session. Congestion control is the ability of a protocol to sense when Internet packet loss and delay is starting to build, and dynamically adjust its sending rate to minimize the effects. Multiple sessions allows a single session to be split into a number of streams, each of which can share the available

bandwidth of the session equally.

SCTP is not commonly supported in operating systems, so browsers will have their own SCTP protocol stack built-in for the data channel. See Section 5.3.1.2 for information on how WebRTC applications can configure SCTP parameters.

6.2.13 IP (Internet Protocol)

Internet Protocol, IP, is the network layer protocol that underlies the Internet. IP version 4, IPv4 [RFC791], the current version, is running out of unique address identifiers, known as IP addresses. IP version 6, IPv6 [RFC2460], was defined to greatly extend the address space to allow the Internet to continue its phenomenal growth into the 21st century. Unfortunately, support and deployment of IPv6 continues to proceed slowly, although many Internet backbones and services and web sites currently support it. Not all Internet Service Providers (ISPs) support it today, unfortunately. Negotiating media and data transport over the different versions of IP can be done using ICE. It is perfectly possible for a dual-stack WebRTC browser to run HTTP over IPv4 and the media over IPv6, or vice versa.

6.3 References

[RFC2616] http://tools.ietf.org/html/rfc2616

[RFC6455] http://tools.ietf.org/html/rfc6455

[RFC3550] http://tools.ietf.org/html/rfc3550

[RFC3711] http://tools.ietf.org/html/rfc3711

[RFC4566] http://tools.ietf.org/html/rfc4566

[RFC5245] http://tools.ietf.org/html/rfc5245

[RFC5389] http://tools.ietf.org/html/rfc5389

[RFC5766] http://tools.ietf.org/html/rfc5766

[RFC5246] http://tools.ietf.org/html/rfc5246

[RFC5764] http://tools.ietf.org/html/rfc5764

[RFC793] http://tools.ietf.org/html/rfc793

[RFC6347] http://tools.ietf.org/html/rfc6347

[RFC768] http://tools.ietf.org/html/rfc768

[RFC4960] http://tools.ietf.org/html/rfc4960

[RFC791] http://tools.ietf.org/html/rfc791

[RFC2460] http://tools.ietf.org/html/rfc2460

7 DEMO APPLICATION CODE

This chapter discusses a sample WebRTC application [DEMO].

7.1 Overview of Basic WebRTC Demo Code

This simple WebRTC demo includes three functional components: a web server, a two-party-per-key signaling channel using the web server, and the WebRTC API calls in the main app. This demo allows for two browsers to connect such that their media takes the most direct path between them that ICE can find. To set up the call, each browser contacts the web server and provides a key. The first browser to provide that specific key initiates a new two-browser signaling channel, waiting until the next browser connects with the same key, at which point the two now have a signaling channel via the web server. The next browser to connect with that specific key resets the signaling channel for that key and is now again waiting for another browser to connect with that key. Media connections are negotiated between the two browsers by sending SDP offers and answers across the signaling channel between the two parties. Then, the browsers send media as directly as possible between themselves.

The next three sections describe the web server code, the signaling channel code, and the WebRTC application. The first two are generic and independent of the WebRTC APIs.

7.2 Web Server

In their simplest form, WebRTC applications are web applications, meaning that they are accessed from web servers using web browsers. In general the web server provides two useful functions for WebRTC applications: serving the web application itself and acting as a relay (providing a "signaling channel") between two or more browsers that can be used to negotiate the destination, type, and format of media to be communicated between the browsers. The HTTP polling signaling channel used here is described in Section 4.3.3 and Figure 4.7.

Any modern web server is capable of providing these two functions, although there is an important property of WebRTC application requirements that must be considered. WebRTC applications are *real-time*. Since human beings are waiting on the completion of connection setups, and since they are accustomed to rapid call connections on the PSTN, if the web server is used to provide the signaling channel, then efficient synchronization across requests from browsers intending to communicate is mandatory.

Broadly speaking, web servers fall into one of two categories: a multi-threaded, separate process per request server that makes file retrieval simple but inter-request synchronization complex, or a single-threaded process that makes inter-request synchronization simple but input/output such as file retrieval more complex. The Apache web server is a quintessential example of the former, and Node.js of the latter.

This demo makes use of Node.js (called "Node" from here on) as the server framework because

1) it dramatically simplifies the communication between separate browser requests, and
2) it uses the asynchronous JavaScript programming model with which most web programmers are already familiar.

For more information on Node, please see the Glossary. This demo makes use only of standard built-in Node packages.

We begin by looking at index.js.

7.2.1 index.js

```
// Copyright 2013 Digital Codex LLC
// You may use this code for your own education.  If you use it
// largely intact, or develop something from it, don't claim
// that your code came first.  You are using this code completely
// at your own risk.  If you rely on it to work in any particular
// way, you're an idiot and we won't be held responsible.

var server = require("./server");
var requestHandlers = require("./serverXHRSignalingChannel");
var log = require("./log").log;
var port = process.argv[2] || 5000;

// returns 404
function fourohfour(info) {
  var res = info.res;
  log("Request handler fourohfour was called.");
  res.writeHead(404, {"Content-Type": "text/plain"});
  res.write("404 Page Not Found");
  res.end();
}

var handle = {};
handle["/"] = fourohfour;
handle["/connect"] = requestHandlers.connect;
handle["/send"] = requestHandlers.send;
```

```
handle["/get"] = requestHandlers.get;

server.serveFilePath("static");
server.start(handle, port);
```

Assuming you already have Node installed, the demo can be run by calling "node index.js".

The code in this file does the following:

1) it loads the server code in "server.js", the signaling channel code in "serverXHRSignalingChannel.js", and the logging code in "log.js",
2) it specifies how certain custom URI paths are to be handled,
3) it specifies the directory containing static files that can be served, and
4) it starts the web server.

Let's next look at the server code in "server.js".

7.2.2 server.js

```
// Copyright 2013 Digital Codex LLC
// You may use this code for your own education.  If you use it
// largely intact, or develop something from it, don't claim
// that your code came first.  You are using this code completely
// at your own risk.  If you rely on it to work in any particular
// way, you're an idiot and we won't be held responsible.

var http = require("http");
var url = require("url");
var fs = require('fs');

var log = require("./log").log;
var serveFileDir = "";

// Sets the path to the static files (HTML, JS, etc.)
function setServeFilePath(p) {
  serveFilePath = p;
}
exports.serveFilePath = setServeFilePath;

// Creates a handler to collect POSTed data and to route the
// request based on the path name
function start(handle, port) {
  function onRequest(req, res) {
```

```
    var urldata = url.parse(req.url,true),
        pathname = urldata.pathname,
        info = {"res": res,
                "query": urldata.query,
                "postData":""};

    log("Request for " + pathname + " received");
    req.setEncoding("utf8");
    req.addListener("data", function(postDataChunk) {
      info.postData += postDataChunk;
      log("Received POST data chunk '"+ postDataChunk + "'.");
    });
    req.addListener("end", function() {
      route(handle, pathname, info);
    });
  }

  http.createServer(onRequest).listen(port);

  log("Server started on port " + port);
}

exports.start = start;

// Determines whether requested path is a static file or a custom
// path with its own handler
function route(handle, pathname, info) {
  log("About to route a request for " + pathname);
  // Check if path after leading slash is an existing file that
  // can be served
  var filepath = createFilePath(pathname);
  log("Attempting to locate " + filepath);
  fs.stat(filepath, function(err, stats) {
    if (!err && stats.isFile()) {  // serve file
      serveFile(filepath, info);
    } else {  // must be custom path
      handleCustom(handle, pathname, info);
    }
  });
}

// This function adds the serveFilePath to the beginning of the
```

```
// given pathname after removing .., ~, and other such
// problematic syntax bits from a security perspective.
// ** There is no claim that this is now secure **
function createFilePath(pathname) {
  var components = pathname.substr(1).split('/');
  var filtered = new Array(),
      temp;

  for(var i=0, len = components.length; i < len; i++) {
    temp = components[i];
    if (temp == "..") continue;    // no updir
    if (temp == "") continue;      // no root
    temp = temp.replace(/~/g,'');   // no userdir
    filtered.push(temp);
  }
  return (serveFilePath + "/" + filtered.join("/"));
}

// Opens, reads, and sends to the client the contents of the
// named file
function serveFile(filepath, info) {
  var res = info.res,
      query = info.query;

  log("Serving file " + filepath);
  fs.open(filepath, 'r', function(err, fd) {
    if (err) {log(err.message);
              noHandlerErr(filepath, res);
              return;}
    var readBuffer = new Buffer(20480);
    fs.read(fd, readBuffer, 0, 20480, 0,
      function(err, readBytes) {
        if (err) {log(err.message);
                  fs.close(fd);
                  noHandlerErr(filepath, res);
                  return;}
        log('just read ' + readBytes + ' bytes');
        if (readBytes > 0) {
          res.writeHead(200,
                    {"Content-Type": contentType(filepath)});
          res.write(
            addQuery(readBuffer.toString('utf8', 0, readBytes),
```

```
                query));
      }
      res.end();
    });
  });
}

// Determine content type of fetched file
function contentType(filepath) {
  var index = filepath.lastIndexOf('.');

  if (index >= 0) {
    switch (filepath.substr(index+1)) {
      case "html":  return ("text/html");
      case "js":  return ("application/javascript");
      case "css":  return ("text/css");
      case "txt":  return ("text/plain");
      default:  return ("text/html");
    }
  }
  return ("text/html");
}

// Intended to be used on an HTML file, this function replaces
// the first empty script block in the file with an object
// representing all query parameters included on the URI of the
// request
function addQuery(str, q) {
  if (q) {
    return str.replace('<script></script>',
                  '<script>var queryparams = ' +
                    JSON.stringify(q) + ';</script>');
  } else {
    return str;
  }
}

// Confirm handler for non-file path, then execute it
function handleCustom(handle, pathname, info) {
  if (typeof handle[pathname] == 'function') {
    handle[pathname](info);
  } else {
```

```
    noHandlerErr(pathname, info.res);
  }
}

// If no handler is defined for the request, return 404
function noHandlerErr(pathname, res) {
  log("No request handler found for " + pathname);
  res.writeHead(404, {"Content-Type": "text/plain"});
  res.write("404 Page Not Found");
  res.end();
}
```

After loading the Node packages it needs and setting some variables, the code defines the two commands it exports – *serveFilePath* and *start*. The first one merely sets an internal variable that will be explained later.

The main method is *start()*. It uses Node's built-in http server module (*http.createServer(onRequest).listen(port)*) to start a server that listens on port 5000. The interesting part, though, is the *onRequest* handler. First, note that Node calls the *onRequest* handler for every request, sending it the request (the URI) and the *res* object to hold the response to send.

onRequest() first uses Node's built-in url parsing module to parse the relevant pieces of the request URI, logs receipt of the request, and then finally sets up asynchronous handlers to parse and save any *POST* data and call *route()* when done.

The primary task of *route()* is to figure out whether the URI refers to a static file to be served or a custom path that needs its own special handler. It will call *serveFile()* for the former and *handleCustom* for the latter. One interesting complication is that file paths can be unsafe unless checked. *route()* first calls *createFilePath* before checking for the existence of a file with that (path)name.

createFilePath() checks for some common pathnames that would give access to files outside the official static file directory and removes them. Note that this code is not guaranteed to provide safety of file access. Writing such code, and explaining it, is beyond the scope of this book. However, there are other Node modules such as Express that have had more extensive work to provide security features. You should consider using it.

serveFile(), as mentioned before, asynchronously opens, reads, and responds with the requested file. It uses Node's built-in *fs* module along with two helper functions: *contentType()* and *addQuery()*.

contentType() examines the file's extension to determine an appropriate value for the *Content-Type* header in the HTTP response.

addQuery() is a minor hack to dynamically insert content into a (HTML) file before it is returned. This particular function is intended to take an object representing the query parameters given in the request URI and insert them into the file. For this WebRTC example, this allows a key specified as a query parameter in the request URI to be inserted into the fetched file so it can be used by the browser. The query parameters are converted to a JSON (JavaScript Object Notation) string and inserted into the first empty script block in the file.

handleCustom(), the final piece of code in server.js, simply attempts to locate the custom path in the *handle[]* array passed into the *start()* function. If found, it executes the code stored there. Otherwise, it uses *noHandlerErr()* to return an HTTP *404*.

7.3 Signaling channel

WebRTC allows for browsers to use any communication method they wish to transmit signaling information, from smoke signals, to tin cans with string, to the faster and more common approaches such as HTTP (XML HTTP Request), WebSockets, and Google App Engine channels. These various approaches are discussed and compared in Chapter 4. Note in each case, though, that the browser is communicating with the web server and not the other browser. Although there is no restriction against browsers communicating directly, the web model itself assumes that browsers (web clients) only communicate with servers and not directly with other web clients. In fact, unless a web server provides IP addresses to web clients, the clients would have no idea how to reach each other. Thus, in most cases there is a server in the cloud that acts as a relay for the messages from one browser to another.

In general, web server-based signaling approaches fall into one of two categories: polling and session-oriented. With polling approaches such as XML HTTP Request (XHR), all communication is initiated by the browser, and the browser's only means of receiving information from the web server is in the response to its messages. To receive messages *from* the server at all, then, the browser needs to regularly send messages to (poll) the server in case there are messages waiting. With session-oriented approaches such as WebSockets, a virtual connection is established between the browser and the web server, allowing for messages to be sent in either direction.

To simplify application programming, it is convenient to define a signaling channel interface that exposes connect and send methods with an ability to specify handlers for when messages are received. If the application intends to use WebRTC in a peer-to-peer mode (the "triangle" model), the purpose of the virtual signaling channel is to

provide a simple connection-oriented model for communicating signaling information to the peer. There are several tricky aspects to creating this illusory connection. The first, and most important, is figuring out how to indicate to the web server what connection to make. Will this truly be a connection to just one peer, or is it intended to be a group "call"? Assuming it is a one-on-one connection, how will the other party be identified? One reasonable approach is to have each browser register with the server and have the server send back the list of registered browsers such that the user can select which one to communicate with. Another approach is to use a key that the other browser also knows and provides. The latter is the approach used in this demo. In this particular demo, the first browser to connect with a given key is registered as the first party and told to wait. When a second client connects with the same key, the signaling channel is established between the two. When another connection attempt is then made using the same key, the existing connection is destroyed and the new party is told to wait. Then, when another browser connections with the same key, the signaling channel is established between these two parties, and so on, toggling between *waiting* and *connected*. Although this approach has the potential disadvantages of requiring both parties to know their keys and a new party destroying an existing connection, it has two very nice properties as well: the key can be included as a parameter in the URI, allowing for the "connection" to be saved merely by bookmarking the URI, and there is very little state to manage when a connection drops because of the browser or network going down. If that happens, both parties just reload their pages and start over.

With that introduction, let's take a look at the code. The signaling channel code in the demo is split across two files – one on the server and one on the client. However, before we look at the signaling code itself on the server, let's look briefly again at where it is used – in index.js.

The lines to consider look like "*handle[...] = ...;*". These lines use methods defined in the server signaling channel code as the handlers for specific URIs. For example, *handle["/connect"]* means that a URI of the form "*http://webrtcserver.example.com:5000/connect*" will be handled via the *connect()* handler defined in serverXHRSignalingChannel.js.

Let's look at the server code in serverXHRSignalingChannel.js.

7.3.1 serverXHRSignalingChannel.js

```
// Copyright 2013 Digital Codex LLC
// You may use this code for your own education.  If you use it
// largely intact, or develop something from it, don't claim
// that your code came first.  You are using this code completely
```

```
// at your own risk.  If you rely on it to work in any particular
// way, you're an idiot and we won't be held responsible.

var log = require("./log").log;

var connections = {},
    partner = {},
    messagesFor = {};

// queue the sending of a json response
function webrtcResponse(response, res) {
  log("replying with webrtc response " +
      JSON.stringify(response));
  res.writeHead(200, {"Content-Type":"application/json"});
  res.write(JSON.stringify(response));
  res.end();
}

// send an error as the json WebRTC response
function webrtcError(err, res) {
  log("replying with webrtc error:  " + err);
  webrtcResponse({"err": err}, res);
}

// handle XML HTTP Request to connect using a given key
function connect(info) {
  var res = info.res,
      query = info.query,
      thisconnection,
      newID = function() {
        // create large random number unlikely to be repeated
        // soon in server's lifetime
        return Math.floor(Math.random()*1000000000);
      },
      connectFirstParty = function() {
        if (thisconnection.status == "connected") {
          // delete pairing and any stored messages
          delete partner[thisconnection.ids[0]];
          delete partner[thisconnection.ids[1]];
          delete messagesFor[thisconnection.ids[0]];
          delete messagesFor[thisconnection.ids[1]];
        }
```

```
      connections[query.key] = {};
      thisconnection = connections[query.key];
      thisconnection.status = "waiting";
      thisconnection.ids = [newID()];
      webrtcResponse({"id":thisconnection.ids[0],
                  "status":thisconnection.status}, res);
    },
    connectSecondParty = function() {
      thisconnection.ids[1] = newID();
      partner[thisconnection.ids[0]] = thisconnection.ids[1];
      partner[thisconnection.ids[1]] = thisconnection.ids[0];
      messagesFor[thisconnection.ids[0]] = [];
      messagesFor[thisconnection.ids[1]] = [];
      thisconnection.status = "connected";
      webrtcResponse({"id":thisconnection.ids[1],
                  "status":thisconnection.status}, res);
    };

  log("Request handler 'connect' was called.");
  if (query && query.key) {
    var thisconnection = connections[query.key] ||
                  {status:"new"};
    if (thisconnection.status == "waiting") { // first half ready
      connectSecondParty(); return;
    } else { // must be new or status of "connected"
      connectFirstParty(); return;
    }
  } else {
    webrtcError("No recognizable query key", res);
  }
}
exports.connect = connect;

// Queues message in info.postData.message for sending to the
// partner of the id in info.postData.id
function sendMessage(info) {
  log("postData received is ***" + info.postData + "***");
  var postData = JSON.parse(info.postData),
      res = info.res;

  if (typeof postData === "undefined") {
    webrtcError("No posted data in JSON format!", res);
```

```
    return;
  }
  if (typeof (postData.message) === "undefined") {
    webrtcError("No message received", res);
    return;
  }
  if (typeof (postData.id) === "undefined") {
    webrtcError("No id received with message", res);
    return;
  }
  if (typeof (partner[postData.id]) === "undefined") {
    webrtcError("Invalid id " + postData.id, res);
    return;
  }
  if (typeof (messagesFor[partner[postData.id]]) ===
          "undefined") {
    webrtcError("Invalid id " + postData.id, res);
    return;
  }
  messagesFor[partner[postData.id]].push(postData.message);
  log("Saving message ***" + postData.message +
     "*** for delivery to id " + partner[postData.id]);
  webrtcResponse("Saving message ***" + postData.message +
              "*** for delivery to id " +
              partner[postData.id], res);
}
exports.send = sendMessage;

// Returns all messages queued for info.postData.id
function getMessages(info) {
  var postData = JSON.parse(info.postData),
      res = info.res;

  if (typeof postData === "undefined") {
    webrtcError("No posted data in JSON format!", res);
    return;
  }
  if (typeof (postData.id) === "undefined") {
    webrtcError("No id received on get", res);
    return;
  }
  if (typeof (messagesFor[postData.id]) === "undefined") {
```

```
    webrtcError("Invalid id " + postData.id, res);
    return;
  }

  log("Sending messages ***" +
      JSON.stringify(messagesFor[postData.id]) + "*** to id " +
      postData.id);
  webrtcResponse({'msgs':messagesFor[postData.id]}, res);
  messagesFor[postData.id] = [];
}
exports.get = getMessages;
```

The code begins by requiring the logging framework and establishing variables to hold the list of connections, a simple array to map between peers, and an array to hold the messages intended for a client. Before the meat of the code (defining how to connect, send messages, and retrieve messages), we define two heavily-used functions for handling replies to the HTTP request: *webrtcResponse()*, which turns any passed in object into a JSON string and returns it, and *webrtcError()*, which returns a given object (possibly a string) as the value of the "*err*" property using *webrtcResponse()*.

The three main handlers are *connect()*, *sendMessage()*, and *getMessages()*. Aside from fetching files, *connect()* is the first interaction a browser will have with the WebRTC server. The original request and any query parameters are passed in as properties of the info object. It's easiest to follow the code if we first look at the end of this function, after all the function definitions. After logging that the *connect* handler was called, the code confirms that the URI contained a key, e.g., "*http://webrtc.example.org:5000/connect&key=1234*". It finds the connection if it already exists in the *connections[]* array or creates a new one if not. It then calls either *connectFirstParty()* if the connection is new or already connected, or *connectSecondParty()* if the status is *waiting*. Now take a look at *connectFirstParty()*, and you'll see that the first thing we do is to delete any existing connection using that key, along with the corresponding entries in the *partner[]* and *messagesFor[]* arrays that will be explained later. Yes, if this occurs during a live connected call this will remove the "signaling channel" for that call, preventing any further signaling. Of course, media flowing directly between the parties will keep flowing because that's what peer-to-peer media means! Although this code sample doesn't, it could check for the loss of the signaling channel and kill the media. Getting back to the code, after deleting any existing connection using that key, a new

connections[] entry is created for the key. Its status is set to *waiting*, a new *ids[]* array property is created containing a newly-generated id for this browser client. Finally, a response to the connect request is generated that contains the new id and the status.

connectSecondParty() behaves similarly to *connectFirstParty()*, but in this case, we

1) don't delete the existing (*waiting*) connection,
2) create a second id to add to the connection, and
3) set up the ancillary arrays that link the two partners (peers) by id and that store the messages from each side for the other.

This shows one of the key benefits of using Node – there is a shared memory space used by all requests to the server, and there is only one thread of control, so it is safe for all requests to access the same connection arrays. As we'll see shortly, all messages are simply stored as values in these shared arrays (in a single memory space).

sendMessage() is the handler for a URI of the form "*http://webrtc.example.org:5000/send*". Note that all information from the client is assumed to be encoded as *POST* data. After the various error checks for missing *POST* data, no message, no id, or lack of a connection (meaning no entry in the *partner[]* or *messagesFor[]* arrays), we push the message onto the end of the *messagesFor* entry for the partner of this client. Then we send a status message back to the client.

Finally, *getMessages()* is the handler for a URI of the form "*http://webrtc.example.org:5000/get*". As with *sendMessage()*, all info from the client is assumed to be encoded as *POST* data. Again, we make sure that there *is POST* data, an id, and an entry in the *messagesFor[]* array for this client id, and then we send back the messages array as the value of the *msgs* property. Oh, and we reset the *messagesFor[]* entry for this id so we don't get the same messages over and over again.

There is one subtlety about when the response is sent back that is due to using Node and asynchronous JavaScript. Because there is only one thread of control, the response is only sent after our code finishes. To keep this clear, the code largely sets the response as the last thing it does before returning, for every path through the code.

Now let's look at the signaling code on the client side, in clientXHRSignalingChannel.js.

7.3.2 clientXHRSignalingChannel.js

```
// Copyright 2013 Digital Codex LLC
// You may use this code for your own education.  If you use it
// largely intact, or develop something from it, don't claim
```

```
// that your code came first.  You are using this code completely
// at your own risk.  If you rely on it to work in any particular
// way, you're an idiot and we won't be held responsible.

// This code creates the client-side commands for an XML HTTP
// Request-based signaling channel for WebRTC.

// The signaling channel assumes a 2-person connection via a
// shared key.  Every connection attempt toggles the state
// between "waiting" and "connected", meaning that if 2 browsers
// are connected and another tries to connect the existing
// connection will be severed and the new browser will be
// "waiting".

var createSignalingChannel = function(key, handlers) {

var id, status, doNothing = function(){},
  handlers = handlers || {},
  initHandler = function(h) {
    return ((typeof h === 'function') && h) || doNothing;
  },
  waitingHandler = initHandler(handlers.onWaiting),
  connectedHandler = initHandler(handlers.onConnected),
  messageHandler = initHandler(handlers.onMessage);

// Set up connection with signaling server
function connect(failureCB) {
  var failureCB = (typeof failureCB === 'function') ||
                  function() {};

  // Handle connection response, which should be error or status
  //  of "connected" or "waiting"
  function handler() {
    if(this.readyState == this.DONE) {
      if(this.status == 200 && this.response != null) {
        var res = JSON.parse(this.response);
        if (res.err) {
          failureCB("error:  " + res.err);
          return;
        }

        // if no error, save status and server-generated id,
```

```
      // then start asynchronous polling for messages
      id = res.id;
      status = res.status;
      poll();

      // run user-provided handlers for waiting and connected
      // states
      if (status === "waiting") {
        waitingHandler();
      } else {
        connectedHandler();
      }
      return;
    } else {
      failureCB("HTTP error:  " + this.status);
      return;
    }
  }
 }

 // open XHR and send the connection request with the key
 var client = new XMLHttpRequest();
 client.onreadystatechange = handler;
 client.open("GET", "/connect?key=" + key);
 client.send();
}

// poll() waits n ms between gets to the server.  n is at 10 ms
// for 10 tries, then 100 ms for 10 tries, then 1000 ms from then
// on. n is reset to 10 ms if a message is actually received.
function poll() {
 var msgs;
 var pollWaitDelay = (function() {
  var delay = 10, counter = 1;

  function reset() {
   delay = 10;
   counter = 1;
  }

  function increase() {
   counter += 1;
```

```
    if (counter > 20) {
      delay = 1000;
    } else if (counter > 10) {
      delay = 100;
    }                       // else leave delay at 10
  }

  function value() {
    return delay;
  }

  return {reset: reset, increase: increase, value: value};
}());

// getLoop is defined and used immediately here.  It retrieves
// messages from the server and then schedules itself to run
// again after pollWaitDelay.value() milliseconds.
(function getLoop() {
  get(function (response) {
    var i, msgs = (response && response.msgs) || [];

    // if messages property exists, then we are connected
    if (response.msgs && (status !== "connected")) {
      // switch status to connected since it is now!
      status = "connected";
      connectedHandler();
    }
    if (msgs.length > 0) {          // we got messages
      pollWaitDelay.reset();
      for (i=0; i<msgs.length; i+=1) {
        handleMessage(msgs[i]);
      }
    } else {                        // didn't get any messages
      pollWaitDelay.increase();
    }

    // now set timer to check again
    setTimeout(getLoop, pollWaitDelay.value());
  });
}());
}
```

```
// This function is part of the polling setup to check for
// messages from the other browser.  It is called by getLoop()
// inside poll().
function get(getResponseHandler) {

  // response should either be error or a JSON object.  If the
  // latter, send it to the user-provided handler.
  function handler() {
    if(this.readyState == this.DONE) {
      if(this.status == 200 && this.response != null) {
        var res = JSON.parse(this.response);
        if (res.err) {
          getResponseHandler("error:  " + res.err);
          return;
        }
        getResponseHandler(res);
        return res;
      } else {
        getResponseHandler("HTTP error:  " + this.status);
        return;
      }
    }
  }

  // open XHR and request messages for my id
  var client = new XMLHttpRequest();
  client.onreadystatechange = handler;
  client.open("POST", "/get");
  client.send(JSON.stringify({"id":id}));
}

// Schedule incoming messages for asynchronous handling.
// This is used by getLoop() in poll().
function handleMessage(msg) {   // process message asynchronously
  setTimeout(function () {messageHandler(msg);}, 0);
}

// Send a message to the other browser on the signaling channel
function send(msg, responseHandler) {
  var reponseHandler = responseHandler || function() {};

  // parse response and send to handler
```

```
  function handler() {
    if(this.readyState == this.DONE) {
      if(this.status == 200 && this.response != null) {
        var res = JSON.parse(this.response);
        if (res.err) {
          responseHandler("error:  " + res.err);
          return;
        }
        responseHandler(res);
        return;
      } else {
        responseHandler("HTTP error:  " + this.status);
        return;
      }
    }
  }

  // open XHR and send my id and message as JSON string
  var client = new XMLHttpRequest();
  client.onreadystatechange = handler;
  client.open("POST", "/send");
  var sendData = {"id":id, "message":msg};
  client.send(JSON.stringify(sendData));
}

return {
  connect:  connect,
  send:  send
};

};
```

createSignalingChannel() takes the shared key and a set of handlers and returns (at the end of the file) an object containing *connect()* and *send()* methods. The code first initializes some variables, including handlers for a status of *waiting*, a status of *connected*, and receipt of a message.

connect() takes a callback for the case of a problem with connection request. Let's look at the end of the routine first. A new *XMLHttpRequest()* is created for the connection request. After setting the handler that will be discussed shortly, the *GET* (as opposed to *POST*) request is made to the '*/connect*' URI at the web server, with the key attached as a query parameter. In looking at the handler, first note that

there are a variety of events from *XMLHttpRequest()* that can trigger execution of the handler. In this code the only case we care about is that the result is complete and parsed, i.e., has a *readyState* of *DONE*. The code then confirms that there was a successful HTTP response (status of *200*), calling the *failureCB()* if not. The next line of the handler points out an interesting fact about *XMLHttpRequest()* – it actually can receive arbitrary data in any format, not just XML. In our case, we parse the response as JSON and first check to see whether the returned object contains an error response. If it does, we call the *failureCB()* callback with the error. Otherwise, we parse out the id generated by the server for this browser client and the status (*waiting* or *connected*), and then start asynchronously polling for messages (discussed in a moment). Then, we finish up handling the response by calling the appropriate handler for the status returned.

poll() is the most complex section of code in this file, and its sole purpose is to check frequently for new messages from the peer (via the server). Code such as this is necessary when using a signaling channel approach that is not session-oriented. The first part in this routine is the definition of a counter object, *pollWaitDelay*. When *reset()* it has a value of 10, but otherwise each call to *increase()* increments a counter. At a count of 11 after the reset the value becomes 100, and at a count of 21 after the reset the value becomes 1000. This value will be used as the number of milliseconds until the next get request. The goal of this is to rapidly check for messages after receiving some and then fall back to less frequent checks if no new messages are forthcoming, in an attempt to be responsive when messages are sent but not flood the server when messages aren't being exchanged.

getLoop() is the routine that schedules get requests. It calls *get()*, which requests any waiting messages from the server, and passes the response from the server to the handler given as the argument. The handler, defined inline in *getLoop()*, first creates a *msgs* array and fills it with messages from the server, if any. Since the server only sends back a response with a *msgs* property if the client browser is connected to another, if there is such a property and the browser didn't already know it was connected, it can now be marked as connected and the *connectedHandler()* can be run. This will happen if the browser is the first to connect, meaning that it is waiting for another to connect. Receipt of the response with a *msgs* property is the indication that the connection has been completed, even if no actual messages were available. What happens next depends upon whether any messages were returned. If there were, two things would happen: first, the timing counter would be reset, and second, the messages would be sent to the

message handler for asynchronous handling. The timing counter is reset to ensure that any rapidly-following messages from the other browser will be picked up quickly. In either case, *getLoop()* is scheduled to run again after *pollWaitDelay.value()* milliseconds.

get() actually does the message fetch, sending the response to the handler given as input. The main part of this code is at the end of the routine and looks much like *connect()* does – a new XMLHttpRequest to the server. In this case, though, the URI is to "*/get*" and we send along this client browser's id (assigned to the browser in the *connect()* call), encoded as a JSON string. As with *connect()*, the response handler

1) confirms a *readyState* of *DONE*, meaning we have the final response,
2) confirms that the HTTP request succeeded (status of *200* and non-null response), and
3) confirms that the response does not contain an *err* property.

Assuming nothing went wrong, the entire response is sent to the response handler passed in as argument to the *get()* method.

handleMessage(), the next piece of code, is called from within *getLoop()* inside *poll()* for each received message. All it does is to schedule the signaling channel's *messageHandler()* callback to be run asynchronously on the message.

The final piece of client signaling code is the definition of *send()*, the routine that sends a message on the signaling channel to the peer. The main code at the end of this method looks much like *connect()* and *get()* did – a new XMLHttpRequest to the server. In this case, though, the URI is to "*/send*" and we send along this client browser's id (assigned to the browser in the *connect()* call) and message (provided as a parameter to *send()*). Again, these two are encoded as a JSON string, and as with *connect()* and *get()*, the routine's response handler

1) confirms a *readyState* of *DONE*, meaning we have the final response,
2) confirms that the HTTP request succeeded (status of *200* and non-null response), and
3) confirms that the response did not contain an *err* property.

As with *get()*, we merely pass the entire response to the response handler passed in as argument to the *send()* method. Unlike for *connect()* and *get()*, note that for *send()* we don't expect any real info in the response – it's there mainly for passing errors and, perhaps, status messages that can be logged.

7.4 Client WebRTC application

We are finally ready to look at the client application, located in the

file start.html.

7.4.1 start.html

```
<!--
// Copyright 2013 Digital Codex LLC
// You may use this code for your own education.  If you use it
// largely intact, or develop something from it, don't claim that
// your code came first.  You are using this code completely at
// your own risk.  If you rely on it to work in any particular
// way, you're an idiot and we won't be held responsible.
-->

<html>
<head>
  <meta http-equiv="Content-Type"
      content="text/html; charset=UTF-8" />
  <style>
    video {
      width:  320px;
      height:  240px;
      border:  1px solid black;
    }
    div {
      display:  inline-block;
    }
  </style>
</head>
<body>

<!-- blank script section is placeholder for query params -->
<script></script>

<!-- load polyfill, local copy first for local testing -->
<script src="adapter.js" type="text/javascript"></script>
<script
  src="https://
webrtc.googlecode.com/svn/trunk/samples/js/base/adapter.js"
  type="text/javascript"></script>

<!-- load XHR-based signaling channel that direct connects based
     on a key -->
<script src="clientXHRSignalingChannel.js"
```

```
      type="text/javascript"></script>

<script>
var signalingChannel, key, id,
    haveLocalMedia = false,
    weWaited = false,
    myVideoStream, myVideo,
    yourVideoStream, yourVideo,
    doNothing = function() {},
    pc,
    constraints = {mandatory: {
                OfferToReceiveAudio: true,
                OfferToReceiveVideo: true}};

////////////////////////////
// This is the main routine.
////////////////////////////

// This kicks off acquisition of local media.  Also, it can
// automatically start the signaling channel.
window.onload = function () {

  // auto-connect signaling channel if key provided in URI
  if (queryparams && queryparams['key']) {
    document.getElementById("key").value = queryparams['key'];
    connect();
  }

  myVideo = document.getElementById("myVideo");
  yourVideo = document.getElementById("yourVideo");

  getMedia();

  //  connect() calls createPC() when connected.
  //  attachMedia() is called when both createPC() and getMedia()
  //  have succeeded.
};

////////////////////
// This next section is for setting up the signaling channel.
////////////////////
```

```
// This routine connects to the web server and sets up the
// signaling channel.  It is called either automatically on doc
// load or when the user clicks on the "Connect" button.
function connect() {
  var errorCB, scHandlers, handleMsg;

  // First, get the key used to connect
  key = document.getElementById("key").value;

  // This is the handler for all messages received on the
  // signaling channel.
  handleMsg = function (msg) {
    // First, we post the message on-screen
    var msgE = document.getElementById("inmessages");
    msgE.innerHTML = JSON.stringify(msg) + "<br/>" +
                     msgE.innerHTML;

    // Then, we take action based on the kind of message
    if (msg.type === "offer") {
      pc.setRemoteDescription(new RTCSessionDescription(msg));
      answer();
    } else if (msg.type === "answer") {
      pc.setRemoteDescription(new RTCSessionDescription(msg));
    } else if (msg.type === "candidate") {
      pc.addIceCandidate(
        new RTCIceCandidate({sdpMLineIndex:msg.mlineindex,
                             candidate:msg.candidate}));
    }
  };

  // handlers for signaling channel
  scHandlers = {
    'onWaiting' : function () {
      setStatus("Waiting");
      // weWaited will be used later for auto-call
      weWaited = true;
    },
    'onConnected': function () {
      setStatus("Connected");
      // set up the RTC Peer Connection since we're connected
      createPC();
    },
```

```
    'onMessage': handleMsg
  };

  // Finally, create signaling channel
  signalingChannel = createSignalingChannel(key, scHandlers);
  errorCB = function (msg) {
    document.getElementById("response").innerHTML = msg;
  };

  // and connect.
  signalingChannel.connect(errorCB);
}

// This routine sends a message on the signaling channel, either
// by explicit call or by the user clicking on the Send button.
function send(msg) {
  var handler = function (res) {
    document.getElementById("response").innerHTML = res;
    return;
  },

  // Get message if not passed in
  msg = msg || document.getElementById("message").value;

  // post it on-screen
  msgE = document.getElementById("outmessages");
  msgE.innerHTML = JSON.stringify(msg) + "<br/>" +
                   msgE.innerHTML;

  // and send on signaling channel
  signalingChannel.send(msg, handler);
}

////////////////////////////
// This next section is for getting local media
////////////////////////////

function getMedia() {
  getUserMedia({"audio":true, "video":true},
               gotUserMedia, didntGetUserMedia);
}
```

```
function gotUserMedia(stream) {
  myVideoStream = stream;
  haveLocalMedia = true;

  // display my local video to me
  attachMediaStream(myVideo, myVideoStream);
  // wait for RTCPeerConnection to be created
  attachMediaIfReady();
}

function didntGetUserMedia() {
  console.log("couldn't get video");
}

//////////////////////////////
// This next section is for setting up the RTC Peer Connection
//////////////////////////////

function createPC() {
  pc = new RTCPeerConnection(
       {iceServers:[{url:"stun:stun.l.google.com:19302"}]});
  pc.onicecandidate = onIceCandidate;
  pc.onaddstream = onRemoteStreamAdded;
  pc.onremovestream = onRemoteStreamRemoved;

  // wait for local media to be ready
  attachMediaIfReady();
}

// When our browser has another candidate, send it to the peer
function onIceCandidate(e) {
  if (e.candidate) {
    send({type:  'candidate',
          mlineindex:  e.candidate.sdpMLineIndex,
          candidate:  e.candidate.candidate});
  }
}

// When our browser detects that the other side has added the
// media stream, show it on screen
function onRemoteStreamAdded(e) {
  yourVideoStream = e.stream;
```

```
  attachMediaStream(yourVideo, yourVideoStream);
  setStatus("On call");
}

// Yes, we do nothing if the remote side removes the stream.
// This is a *simple* demo, after all.
function onRemoteStreamRemoved(e) {}

////////////////////////////////////
// This next section is for attaching local media to the Peer
// Connection.
////////////////////////////////////

// This guard routine effectively synchronizes completion of two
// async activities:  the creation of the Peer Connection and
// acquisition of local media.
function attachMediaIfReady() {
  // If RTCPeerConnection is ready and we have local media,
  // proceed.
  if (pc && haveLocalMedia) {attachMedia();}
}

// This routine adds our local media stream to the Peer
// Connection.  Note that this does not cause any media to flow.
// All it does is to let the browser know to include this stream
// in its next SDP description.
function attachMedia() {
  pc.addStream(myVideoStream);
  setStatus("Ready for call");

  // auto-call if truthy value for call param in URI
  // but also make sure we were the last to connect (to increase
  // chances that everything is set up properly at both ends)
  if (queryparams && queryparams['call'] && !weWaited) {
    call();
  }

}

////////////////////////////
// This next section is for calling and answering
////////////////////////////
```

```
// This generates the session description for an offer
function call() {
  pc.createOffer(gotDescription, doNothing, constraints);
}

// and this generates it for an answer.
function answer() {
  pc.createAnswer(gotDescription, doNothing, constraints);
}

// In either case, once we get the session description we tell
// our browser to use it as our local description and then send
// it to the other browser.  It is the setting of the local
// description that allows the browser to send media and prepare
// to receive from the other side.
function gotDescription(localDesc) {
  pc.setLocalDescription(localDesc);
  send(localDesc);
}

////////////////////////////////////
// This section is for changing the UI based on application
// progress.
////////////////////////////////////

// This function hides, displays, and fills various UI elements
// to give the user some idea of how the browser is progressing
// at setting up the signaling channel, getting local media,
// creating the peer connection, and actually connecting
// media (calling).
function setStatus(str) {
  var statuslineE = document.getElementById("statusline"),
      statusE = document.getElementById("status"),
      sendE = document.getElementById("send"),
      connectE = document.getElementById("connect"),
      callE = document.getElementById("call"),
      scMessageE = document.getElementById("scMessage");

  switch (str) {
    case 'Waiting':
      statuslineE.style.display = "inline";
```

```
    statusE.innerHTML =
      "Waiting for peer signaling connection";
    sendE.style.display = "none";
    connectE.style.display = "none";
    break;
   case 'Connected':
    statuslineE.style.display = "inline";
    statusE.innerHTML =
      "Peer signaling connected, waiting for local media";
    sendE.style.display = "inline";
    connectE.style.display = "none";
    scMessageE.style.display = "inline-block";
    break;
   case 'Ready for call':
    statusE.innerHTML = "Ready for call";
    callE.style.display = "inline";
    break;
   case 'On call':
    statusE.innerHTML = "On call";
    callE.style.display = "none";
    break;
   default:
  }
}

</script>

<div id="setup">
  <p>WebRTC Demo</p>
  <p>Key:
    <input type="text" name="key" id="key"/>
    <button id="connect" onclick="connect()">Connect</button>
    <span id="statusline" style="display:none">Status:
      <span id="status">Disconnected</span>
    </span>
    <button id="call" style="display:none"
          onclick = "call()">Call</button>
  </p>
</div>

<div id="scMessage" style="float:right;display:none">
  <p>Message:
```

```
   <input type="text" width="100%" name="message"
id="message"/>
   <button id="send" style="display:none"
         onclick="send()">Send</button>
  </p>

  <p>Response:  <span id="response"></span></p>
</div>

<br/>

<div style="width:45%;vertical-align:top">
  <div>
   <video id="myVideo" autoplay="autoplay" controls
         muted="true"/>
  </div>
  <p><b>Outgoing Messages</b>
    <br/>
    <span id="outmessages"></span>
  </p>
</div>

<div style="width:45%;vertical-align:top">
  <div>
   <video id="yourVideo" autoplay="autoplay" controls />
  </div>
  <p><b>Incoming Messages</b>
    <br/>
    <span id="inmessages"></span>
  </p>
</div>

</body>
</html>
```

Before we dive into the JavaScript code, let's skip to the end of the file where the HTML markup lives. There are three main sections: the upper left app control area, the upper right status and signaling message section (hidden by default via "*display:none*"), and the lower video/messages section.

The upper left app control area initially shows a field for entering a key and a Connect button that, when clicked, calls the *connect()* function

that contacts the server with the key to set up the signaling channel. We'll get to that code in a moment. It also contains a status section that is initially hidden ("*display:none*") and a Call button that is also hidden until everything is ready for the call to take place.

The upper right status and signaling message area will be explained when we get to the code that activates it, but for now know that it will contain an information area to hold responses to the various XMLHttpRequests, with an ability to manually send a message on the signaling channel. This section is largely for testing purposes, but it can be used as a primitive chat capability over the signaling channel.

The final section consists of two side-by-side video elements, one for this browser's video and one for the peer's, with a section under each showing the messages sent by that browser over the signaling channel. This is where the SDP messages will appear, most recent at the top. Note that we begin with the audio from our side muted in the video element. It's not necessary to play our own audio, and muting it reduces problems with feedback. Even with our own audio muted it is still helpful to have headphones of some sort so the audio from the peer doesn't get fed back into the microphone on the device, something that happens easily on laptops.

Let's look next at the *setStatus()* function right above the HTML. It simplifies display of the various HTML components into a single status setting. Screen shots of the various states are shown in Figures 7.1 through 7.5.

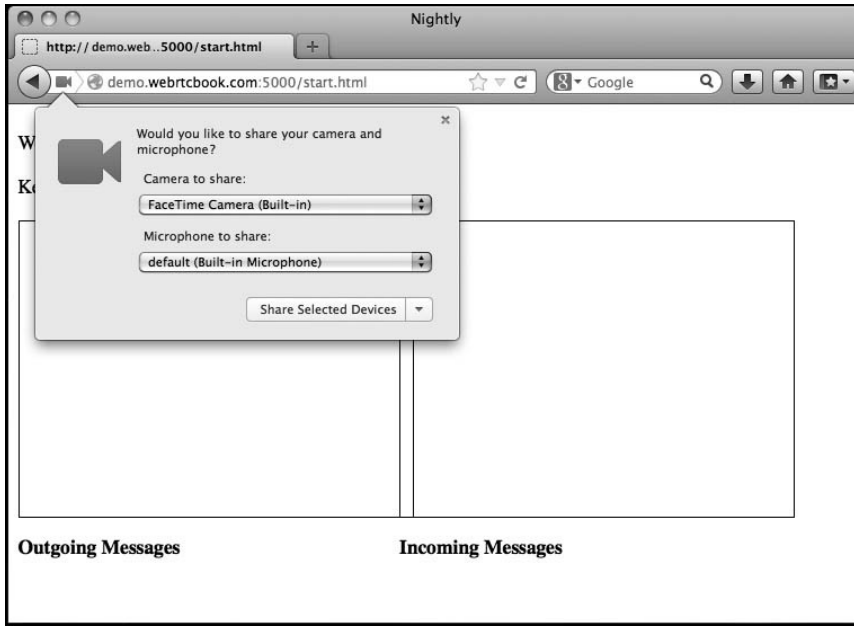


Figure 7.1 Demo Code Permissions Grant

Figure 7.2 Demo Code Waiting State

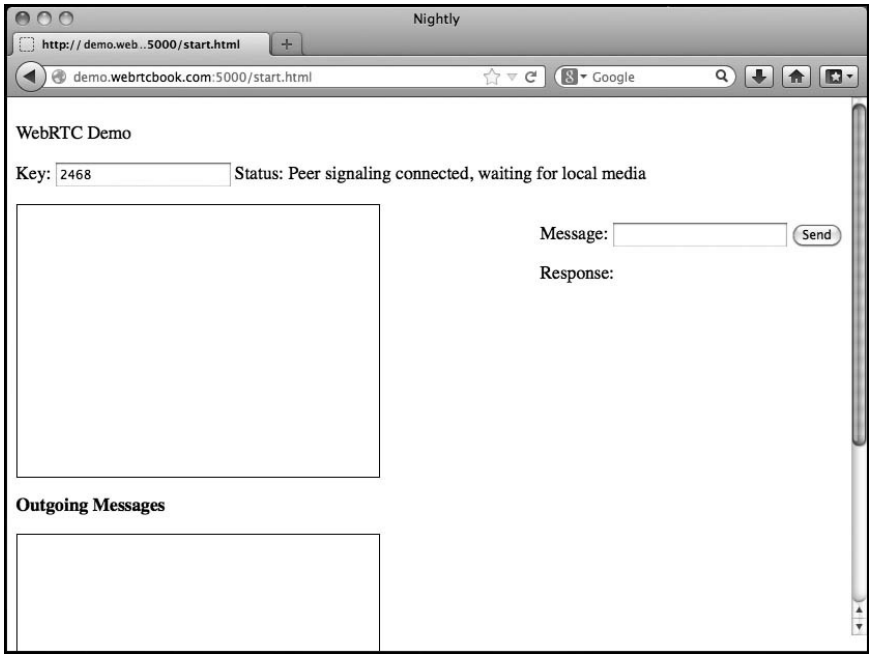


Figure 7.3 Demo Code Connected State

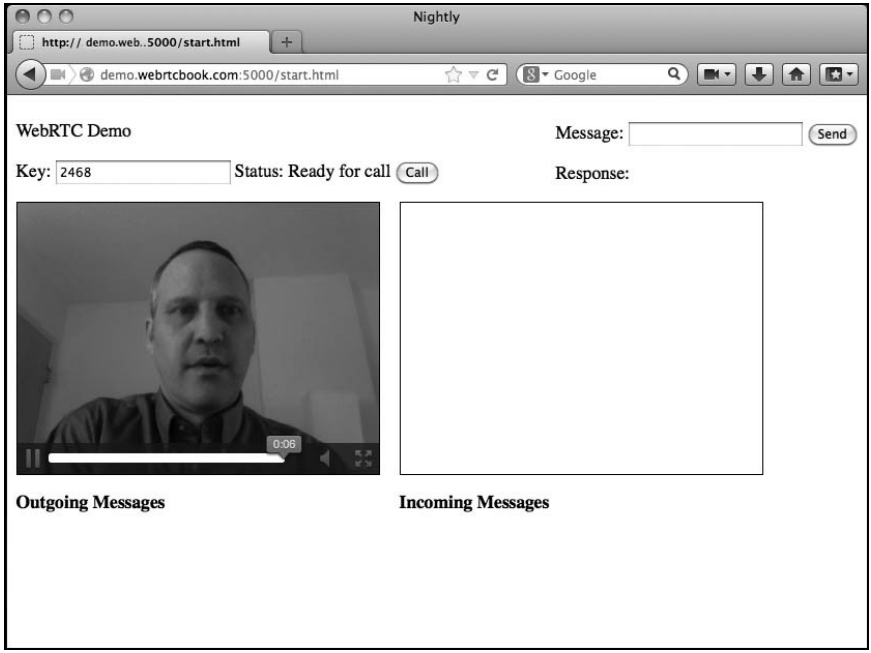


Figure 7.4 Demo Code Ready For Call State

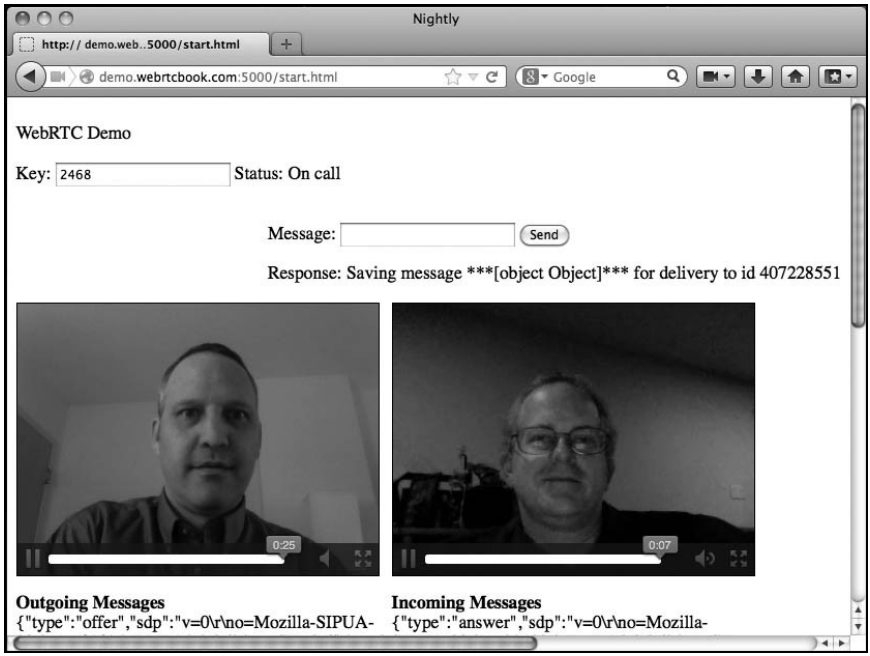


Figure 7.5 Demo Code On Call State

Now let's go back up to the top. Skipping the CSS (Cascading Style Sheets) styling, which just sets how the video elements look and the div elements lay out on the screen, we come to the first *<script></script>* block. Yes, it's empty! Remember our server code? *addQuery()* in server.js will insert into this block any query parameters included in the URI requesting this file. They will be inserted as properties on a new local *queryparams* variable.

Next, there are two *<script>* lines loading the adapter.js file that maps the standard API calls to the current non-standard names used by Google Chrome and Mozilla Firefox. Only the second line is necessary, since it refers to the online location for this file. This file was created by Mozilla and Google. However, for offline testing it is convenient to copy this file locally. If you do, then the first line will try to load the local copy first.

The third *<script>* section loads the client *XHRSignalingChannel* code that we just reviewed. Finally, the fourth section contains all the JavaScript code that is unique to this demo application.

In this section, we first define all of our common variables. After this the core routine is defined first, as a function that is executed when the page finishes loading. If the key was given as a query parameter on the URI, for example "*http://www.example.org:5000/start.html?key=1234*",

the key field on the screen is filled in with this value and then *connect()* is called automatically to connect to the server for setting up the signaling channel. After setting local variables to the two video elements, the code then calls *getMedia()* to request access to local media. In the pseudo code we called *getMedia()* and *createPC()*, then *attachMedia()* to attach the local media to the newly-created Peer Connection. In this demo, *createPC()* is called automatically by *connect()* once the signaling connection is established between the two browsers (still via the web server, of course). Also, because the creation of the signaling channel and acquisition of local media are both asynchronous and running in parallel, *attachMedia()* is only called once both have completed. This will be clearer when we reach the code that does that.

connect() is next, the routine that sets up the signaling channel. It can be activated by the user clicking the Connect button or automatically (in *window.onload* that we just reviewed) if the key is provided as a query parameter in the URI for this file request. The first action is to get the key from the key input field on the screen. We then define the handlers that will be used by the signaling channel. The *onWaiting* and *onConnected* handlers are defined in-line later, but the code for the *onMessage* handler is large enough that we define it first as the *handleMsg()* function. First, we prepend the message to the messages list under the remote video (since it was sent by the remote browser). Then, we take appropriate WebRTC action based on the type of message we received. This is very similar to what's in the pseudo code, where for either an offer or an answer we set the remote session description to the value received. If what we received was an offer, then we need to generate and send an answer by calling *answer()*. The only other type of WebRTC message we send in this demo is an ICE candidate, so if we receive one from the other side we need to tell our browser about it by calling *addIceCandidate*. Note that by the time we handle a message we have already set up the Peer Connection, so all of these methods should be defined. How do we know that's the case? If you look at the handlers defined next for *onWaiting* and *onConnected*, you'll see that when connected we set our status to *connected* and call *createPC()* to set up the Peer Connection. Since we can't process a message until we are connected, and since messages are scheduled to be processed after the *onConnected* handler completes, we should be safe. If we happen to be the first browser to connect with the given key, the *onWaiting* handler will be called, in which case we set the status to *waiting* so the user can be informed, and we record that we were the one who waited. We'll see later how this is used to determine which browser sends the offer if the

"*call=1*" query parameter and value are set on the URI to indicate that the call should happen automatically.

Now that the handlers for the signaling channel have all been set, we can finally call *createSignalingChannel()* with the key and handlers and then call its *connect()* method to connect.

send() is used to send a message on the signaling channel to the other browser. If the message is passed as a parameter it will be sent; otherwise, the message will be pulled from the message field on-screen. The user can call *send()* explicitly by clicking on the Send button in the upper right portion of the screen. In either case, the message is prepended to the message list under this browser's video, since it's an outgoing message. The final step is to send it on the signaling channel.

The *getMedia()* function is quite simple, calling *getUserMedia* and requesting both audio and video. When it gets the media stream, which requires the user's permission, it calls the *gotUserMedia()* callback with the stream.

gotUserMedia() saves the media stream in the *myVideoStream* variable and records that we actually have received local media (*haveLocalMedia = true*). Then it calls *attachMediaStream()* to display the new stream in the *myVideo* element. *attachMediaStream()* is a function in adapter.js that displays a stream in a video element. It is needed because Google Chrome and Mozilla Firefox use different syntaxes to accomplish this. The last step in *gotUserMedia()*, an extremely important step, is to call *attachMediaIfReady()*. The pseudo code immediately called *attachMedia()* to attach the media to the Peer Connection, but in our case we need to make sure that we both have local media and have created the Peer Connection. Since both are asynchronous and can take significant time, *attachMediaIfReady()* is a check function that is called both after getting local media and after the Peer Connection is set up, so that once both are ready we can call *attachMedia()*.

In a full application with good user handling, the *didntGetUserMedia()* callback would notify the user of a problem and offer options to resolve them. In this simple demo, it just logs the inability to get video. In this simple application it's not that big a deal, because the application can be restarted simply by reloading the page in the browser. Of course, a second restart might be needed if this browser was still waiting for another to connect since the server doesn't know this isn't a different browser.

createPC() here is very similar to the pseudo code. We create a new connection and set the various handlers for it. As with *getMedia()*, we finish by calling the *attachMediaIfReady()* check function.

Next are the three handlers. First, *onIceCandidate()* will be called when our browser determines it has a new candidate address at which it might be reachable. So, to inform the other browser the code calls *send()* with the new candidate in the format the other browser expects for *addIceCandidate()*.

onRemoteStreamAdded() is called by the Peer Connection when the remote browser adds a new stream (via an offer or answer). The handler saves the media stream, displays it in the *yourVideo* element, and sets the status to "*On call*" since our media streams are now set up.

Finally, *onRemoteStreamRemoved()* does absolutely nothing. It should do something, but that's a topic for a more advanced demo.

We have finally reached *attachMediaIfReady()*, the guard function that confirms that we have both a Peer Connection and local media, at which point it calls *attachMedia()*. *attachMedia()* is similar to how it appears in the pseudo code, calling *addStream()* with our local video so it can be included in the next offer or answer. At this point our Peer Connection is ready to negotiate media, so we are ready for the call. We set the status accordingly. At this point there is one more automatic capability that will trigger if the *call* query parameter is given on the URI for this page with a value that will be interpreted by JavaScript as true – initiation of the *call()*, the offer/answer exchange the Peer Connection will do to negotiate media. Of course, there is a risk that both browsers will reach this point at the same time and both send offers. We can avoid this by picking one of the browsers to be the one that initiates the offer/answer exchange. Since all requests to our Node server are naturally serialized, one browser will have been the first to connect, receiving a status of *waiting* from the server. Hopefully that browser will already have set up both the Peer Connection and local media, so when the second browser, the one that wasn't waiting, has attached its media to the Peer Connection it should be ready to call. In short, we don't guarantee but do increase the chances that everything is set up on both sides by having the browser that didn't wait be the one to *call()*. The neat thing now about the key and call query parameters is that two users can just load a URI like "*http://www.example.org:5000/start.html?key=1234&call=1*", the same URI for both, and the two apps should just connect up automatically – first the signaling channel and then the Peer Connection/media. A URI such as this could even be bookmarked with the person's name, or stored in an address book entry for that person's name.

Moving on, the final section of the code handles the calling and answering (offer and answer). *call()* calls *createOffer()* and *answer()* calls *createAnswer()*. In both cases, when the session description is

ready it is given to the *gotDescription* callback, which sets it as our local description and then sends it to the other browser.

And that's it!

So, what's next beyond this sample application? Obviously some extra error handling can and should be added. Also, there is no authentication or authorization in the use of the server's signaling channel capabilities. The server code is very simplistic and potentially insecure. From a WebRTC perspective, this code only provides a single offer and a single answer. A real application should make use of *negotiationneeded* to handle the case where media needs to be renegotiated.

7.5 References

[DEMO] http://demo.webrtcbook.com

8 IETF WEBRTC DOCUMENTS

There are a number of standards documents that define the protocols used in WebRTC. Some are Internet-Drafts – working documents in the IETF that will continue to be refined and developed before final publication. Others have already been published as RFCs (Request for Comments), the standards documents of the IETF. The draft documents are grouped according to the working group currently discussing and editing each document. Other RFCs that are related to WebRTC such as RTP and SDP are covered in the next chapter.

8.1 Request For Comments

IETF Request For Comments or RFCs are referenced by their RFC number, and do not change with time. There are numerous sources for RFCs including the RFC Editor's Page [RFC-EDITOR]. The RFC links provided in this book are to a conveniently hyperlinked version stored at the IETF website.

Currently, no IETF WebRTC documents have been published as RFCs.

8.2 Internet-Drafts

IETF Internet-Drafts are the work-in-progress documents in the IETF. They change frequently before being finalized as RFCs. Internet-Drafts can be working group documents or individual submissions. Individual documents are likely to undergo the largest changes and will very likely change document names before being published as an RFC. For more information on the IETF standards process, see Appendix B.

8.3 RTCWEB Working Group Internet-Drafts

The main documents of the IETF RTCWEB Working Group are listed in Table 8.1. The documents are discussed in the following sections.

Document	Title	Section
Overview	"Overview: Real Time Protocols for Browser-based Applications"	8.3.1
Use Cases and Requirements	"Web Real-Time Communication Use-cases and Requirements"	8.3.2
RTP Usage	"Web Real-Time Communication (WebRTC): Media Transport and Use of RTP"	8.3.3
Security Architecture	"RTCWEB Security Architecture"	8.3.4
Threat Model	"Security Considerations for RTC-Web"	8.3.5
Data Channel	"RTCWeb Data Channels"	8.3.6
JSEP	"JavaScript Session Establishment Protocol"	8.3.7
Audio	"WebRTC Audio Codec and Processing Requirements"	8.3.8
Quality of Service	"DSCP and other packet markings for RTCWeb QoS"	8.3.9

Table 8.1 IETF RTCWEB Working Group Documents

8.3.1 "Overview: Real Time Protocols for Brower-based Applications" [draft-ietf-rtcweb-overview]

The Overview working group Internet-Draft [draft-ietf-rtcweb-overview] provides an overview of the protocols and architecture used by WebRTC. The high level goal is to build into a standard HTML5 browser the capabilities for real-time communication with audio, video, and data communications. Codecs for encoding and decoding media streams will be built-in, as will media processing such as echo cancellation (allowing for hands-free or speakerphone operation without a headset or push-to-talk button) and packet loss concealment. A key goal is the establishment of multimedia session between two browsers with the media packets being sent directly between the browsers ("peer-to-peer"). This reduces load, processing, and bandwidth requirements on servers, and minimizes latency (delay) and packet loss on the media path. APIs (Application Programming Interfaces) will be used to expose

the browser RTC functions to JavaScript web applications downloaded as part of a web page. This document provides an overall view of the architecture and approach to the WebRTC problem.

This document is currently listed as Standards Track. However, it mainly is a discussion of the WebRTC architecture and philosophy, so it may eventually be published as an Informational RFC.

8.3.2 "Web Real-Time Communication Use-cases and Requirements" [draft-ietf-rtcweb-use-cases-and-requirements]

This working group Internet-Draft [draft-ietf-rtcweb-use-cases-and-requirements] details the requirements and use cases for WebRTC. The requirements include the ability to traverse NAT (Network Address Translation), work with IPv4 and IPv6 and dual stack browsers, utilize wideband and narrowband Internet connections and deal with congestion and packet loss. Use cases include audio and video with multiple sources and streams. Multiparty communication is also described. Applications include conventional telephony calling, meet-me video chat, gaming with peer-to-peer exchange of information, and distributed music making. Interworking with the PSTN (Public Switched Telephone Network) and existing VoIP (Voice over IP) and multimedia systems using SIP and other signaling protocols are also discussed.

This document will be published as an Informational RFC, which will document the thinking and logic behind the design of the actual protocol documents.

8.3.3 "Web Real-Time Communication (WebRTC): Media Transport and Use of RTP" [draft-ietf-rtcweb-rtp-usage]

This working group Internet-Draft [draft-ietf-rtcweb-rtp-usage] describes the usage of Real-time Transport Protocol (RTP) in WebRTC. Browsers will have a full RTP stack built-in as part of the RTC function, as shown in Figure 1.2. The use of the RTP Control Protocol (RTCP) is also specified for the exchange of session information and sender and receiver reports on quality and congestion. Besides the core RTP specification described in Section 6.2.3, WebRTC implements a number of extensions and additions to RTP. Some of these extensions are common, while others are uncommon. This document does not define any new RTP extensions, but references other RFCs and Internet-Drafts that do. The most important difference between regular RTP and RTP as used by WebRTC relates to multiplexing. Normally, each RTP media stream uses a unique UDP port number, and the RTCP session associated with a given RTP stream uses another unique port number. So a multimedia

session involving audio and video and associated RTCP sessions would normally require four separate UDP ports. In WebRTC, only one UDP port will be used: all media, voice and video, and the corresponding RTCP sessions will be multiplexed over the same port. This greatly reduces the effort needed to traverse Network Address Translation (NAT) boxes. The details on how to accomplish this are described in Section 8.4.1 and Section 8.5.1. The multiplexing of RTP and RTCP packets on a single port is described in Section 9.1.5.

For backwards compatibility with non-WebRTC endpoints (such as SIP or Jingle clients), browsers will be required to fall back to using multiple UDP ports, as part of normal media negotiation.

A number of conferencing and header extensions are also referenced in this document.

This document will be published as a standards track RFC as it documents the required usage of RTP and RTCP for WebRTC.

8.3.4 "RTCWEB Security Architecture" [draft-ietf-rtcweb-security-arch]

This working group Internet-Draft [draft-ietf-rtcweb-security-arch] describes the security architecture for WebRTC. The basic security model of web browsing is applied to real-time communications. In its simplest form, the human user must trust their web browser. The user relies on their web browser to protect them against potentially malicious sites they might visit. Before a site is given access to a microphone or camera, the browser must get permission from the user. Figure 8.1 shows an example of an actual WebRTC browser (Google Chrome Canary on Mac OS) requesting user consent when a WebRTC application (Meetecho collaboration [MEETECHO]) requesting permission from the user to use the microphone and camera. The permission request is located under the URL bar.

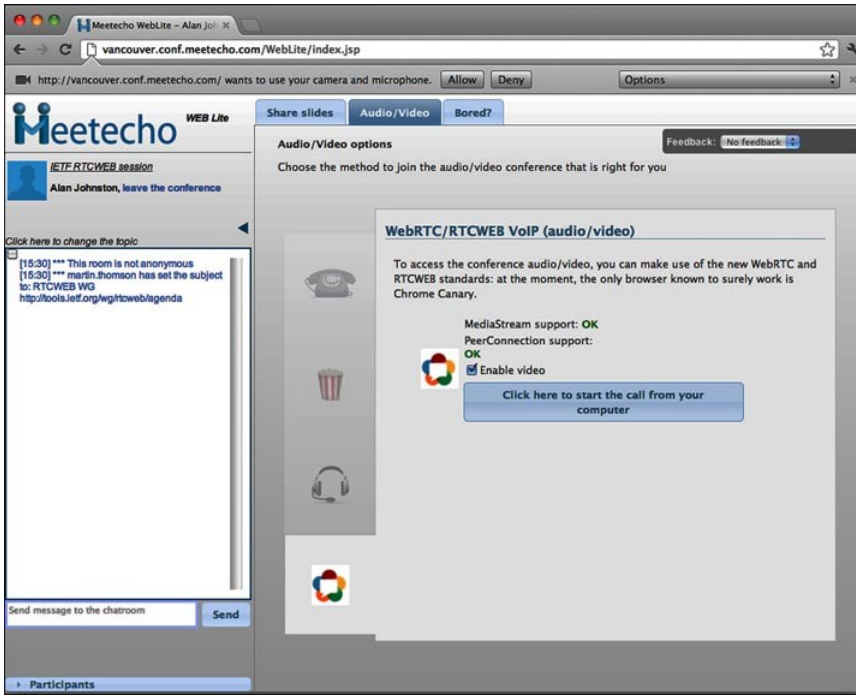


Figure 8.1 WebRTC Browser Asking Permission of User

In addition, the fact that a microphone or camera is being used by a web site must also be indicated to the user. This is shown in Figure 8.2 for the Meetecho WebRTC application.

The document also discusses the ways in which protocols such as TLS (Section 6.2.8), SRTP (Section 6.2.3), and DTLS-SRTP [RFC5764] can be used to provide security in WebRTC. For example, the use of security provided by HTTPS is discussed, as shown in Figure 8.3.

The use of an Identity Provider is also discussed, as described in Section 10.4.

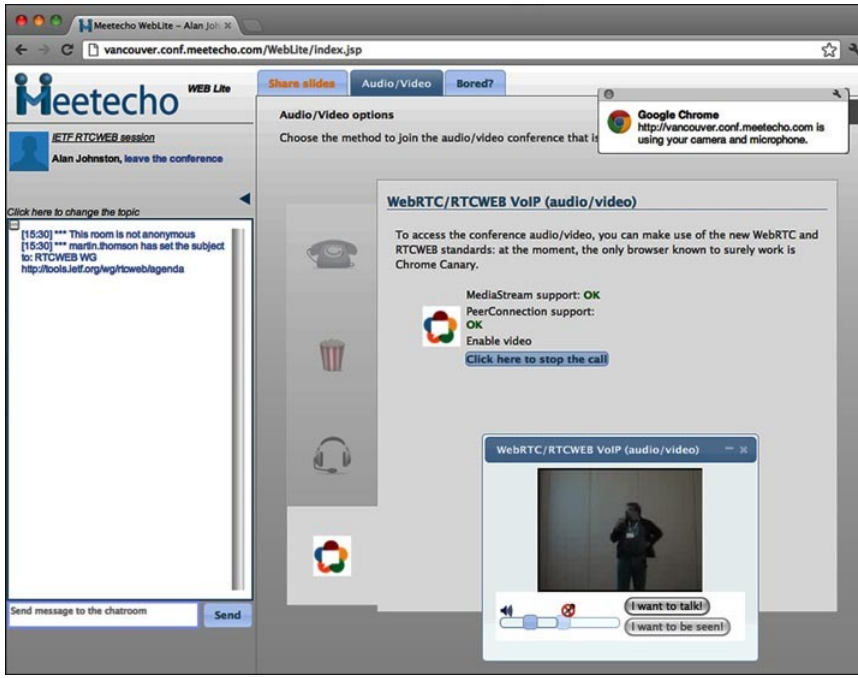


Figure 8.2 WebRTC Browser Providing Indication to User that Microphone and Camera are in Use

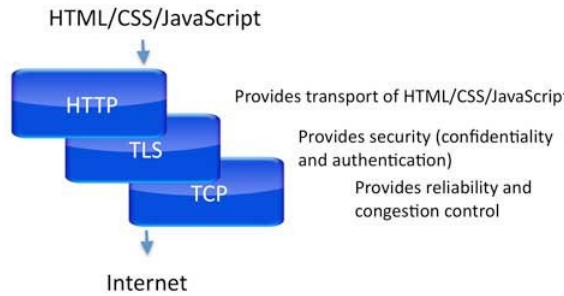


Figure 8.3 HTTPS Security Layers in WebRTC

8.3.5 "Security Considerations for RTC-Web" [draft-ietf-rtcweb-security]

This working group Internet-Draft [draft-ietf-rtcweb-security] describes the threat model for WebRTC in support of the RTCWEB Security Architecture document. This threat model will be used to evaluate the security mechanisms in the protocol specifications.

8.3.6 "RTCWeb Data Channels" [draft-ietf-rtcweb-data-channel]

This working group Internet-Draft [draft-ietf-rtcweb-data-channel] discusses the requirements and protocols used for non-RTP, non-media data exchanged between browsers. The proposal is to use Stream Control Transport Protocol, SCTP (see Section 6.2.12) over Datagram Transport Layer Protocol, DTLS (see Section 6.2.10), over User Datagram Protocol (UDP) as shown in Figure 8.4. This somewhat complicated protocol stack is designed to provide NAT traversal, authentication, confidentiality, and reliable transport of multiple streams. While SCTP is a transport layer protocol, it cannot be used directly on top of IP (Internet Protocol) due to the presence of NAT. Instead, the whole stack is tunneled over UDP so that NAT will not drop packets. ICE is used to establish the data channel to provide communication consent and hole punching to enable NAT and firewall traversal. Note that SCTP will be implemented in the browser itself (so called "user-land") as opposed to relying on an operating system (kernel) implementation.

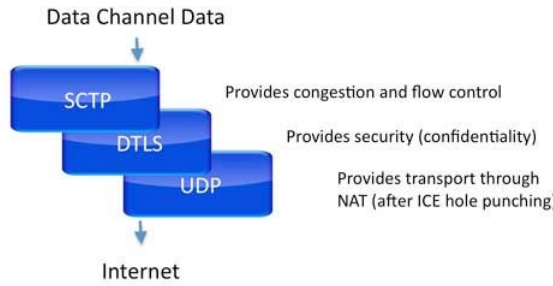


Figure 8.4 Protocol Layers of the WebRTC Data Channel

8.3.7 "JavaScript Session Establishment Protocol" [draft-ietf-rtcweb-jsep]

The JavaScript Session Establishment Protocol (JSEP) [draft-ietf-rtcweb-jsep] is a new 'signaling protocol' developed for WebRTC. It is not really a signaling protocol in the way that SIP and Jingle are signaling protocols. Instead, JSEP describes how a JavaScript application running on a browser interacts with the built-in RTC function. It defines how the JavaScript can get information about the capabilities of the browser, including supported media types and codecs – the SDP *RTCSessionDescription* object. It also describes how JavaScript can manage the offer/answer negotiation of media between the browsers, and the ICE hole punching process running in the browser. It is very

important to note that JSEP does not define any on-the-wire protocol – how the SDP objects are sent to and from the web server is not described. (This is expected to be done using a standard web approach such as XHR).

Those experienced with signaling protocols may wonder where the 'state machine' is located. The state machine in WebRTC is not standardized, but is instead just part of the JavaScript code that uses JSEP.

APIs are used to get candidate and capability information from the browser. The ICE state machine runs natively in the browser, and is decoupled from the JSEP state machine. The division of state between the browser and the server is shown in Figure 8.5.

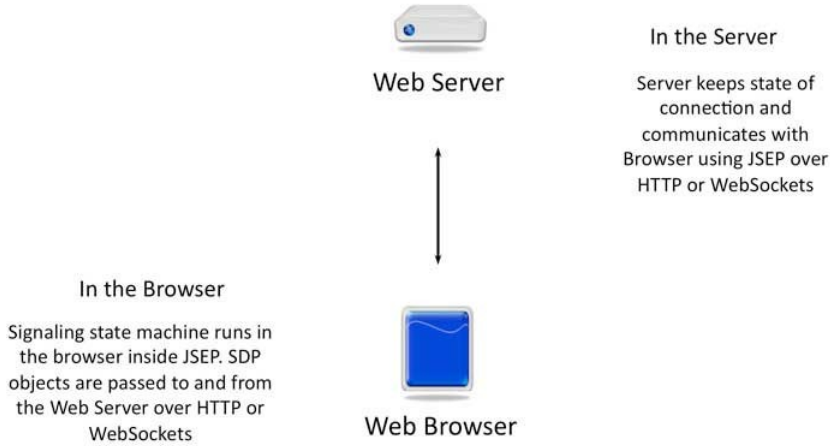


Figure 8.5 Division of State Between Browser and Server

JSEP currently uses SDP session descriptions, Section 6.2.4, as the syntax for offers and answers, although a JavaScript Object Notation (JSON) format could be defined in the future. Within the application JavaScript code, some form of an offer/answer state machine is needed. Some initial work was to use the existing SIP offer/answer model, defined in [RFC3264], but with a different encoding. (This was known as ROAP, RTCWEB Offer Answer Protocol, and was described in a number of individual Internet-Drafts.) However, this was viewed as too restrictive, and this current proposal tries to relax the offer/answer requirements as much as possible. In addition, this proposal tries to deal with the "refresh" or "rehydration" problem where a user manually reloads a web page during an RTC session, although this is still under discussion in the working group. This book does not cover the details of

SDP offer/answer. To understand SDP offer/answer, see [SDP-OA].

The JSEP state machine is shown in Figure 8.6.

Consider the case where a browser is initiating a new Peer Connection. Starting from the Idle state, the browser generates and sends the offer to the other browser. This is the Local-Offer state. If an answer is received,

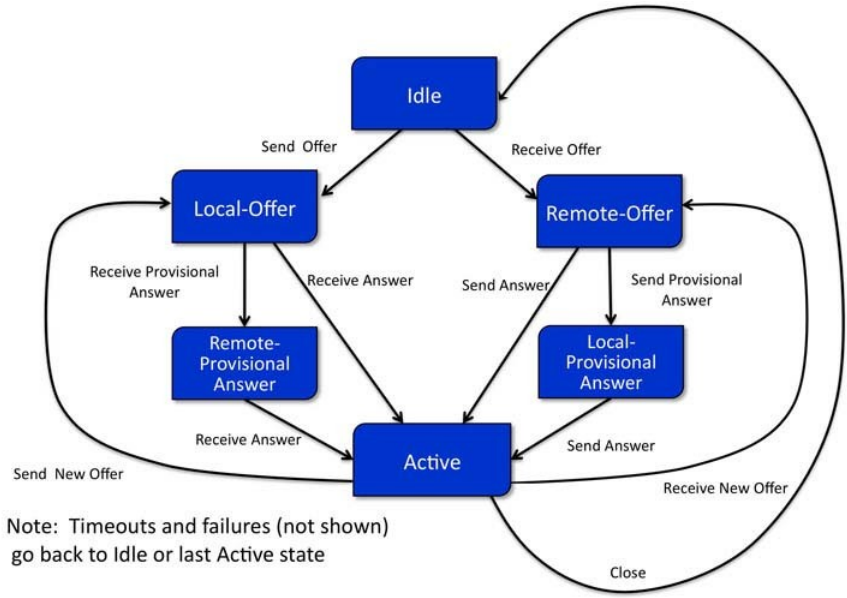


Figure 8.6 JSEP State Machine

the Peer Connection is established and the resulting state is Active. Alternatively, a provisional answer could be received. A provisional answer is identical to an answer except that resources associated with the offer should not be freed up. More than one provisional offer could be received, but the receipt of an answer establishes the Peer Connection, moving into the Active state. In the case where a browser receives an offer, the state machine is similar, except a browser should not normally send a provisional answer, but instead a full answer. Provisional answers are normally sent by gateways to other protocols such as SIP in order to handle special cases such as forking. Once in the Active state, a Peer Connection can be changed by either browser initiating a new offer, repeating the process.

8.3.8 "WebRTC Audio Codec and Media Processing Requirements" [draft-ietf-rtcweb-audio]

This Internet-Draft [draft-ietf-rtcweb-audio] discusses requirements for

audio codecs and media processing. For example the Opus codec [RFC6176] is the default audio codec between browsers, with PCM u-law/A-law (G.711) [RFC3551] and telephone events [RFC4733] (for DTMF – dual tone multi frequency) for interoperation with the PSTN and SIP and Jingle clients. The recommended audio level is also discussed. Requirements for both near-end and far-end echo cancellation are also discussed.

The default video codec is currently under discussion with H.264 [draft-burman-rtcweb-h264-proposal], VP8 [draft-alvestrand-rtcweb-vp8], and others being considered by the working group.

Codec information is summarized in Table 8.2.

Codec	Use	Specification
Opus	Narrowband to wideband Internet audio codec for speech and music	RFC 6176
G.711	PCM audio encoding for PSTN interworking and backwards compatibility with VoIP systems	RFC 3551
Telephone Events	Transport of Dual Tone Multi Frequency (DTMF) tones	RFC 4733
H.264	Proposed video codec requiring licensing	RFC 6184
VP8	Proposed open source video codec	RFC 6386

Table 8.2 WebRTC Codec Summary

8.3.9 "DSCP and other packet markings for RTCWeb QoS" [draft-ietf-rtcweb-qos]

This draft [draft-ietf-rtcweb-qos] provides suggested quality of service (QoS) settings for WebRTC. This includes DiffServ Code Points (DSCP), QoS Class Indicator (QCI) mapping for LTE (Long Term Evolution), and WiFi mapping priorities for IEEE 802.11e.

8.4 Individual Internet-Drafts

These documents have not yet been adopted as working group items but have had discussion and seem to have some level of support and interest in the IETF.

8.4.1 "Multiplexing Multiple Media Types In a Single Real-Time Transport Protocol (RTP) Session" [draft-lennox-rtcweb-rtp-media-type-mux]

This individual Internet-Draft [draft-lennox-rtcweb-rtp-media-type-mux] discusses the RTP issues associated with multiplexing different media types (e.g. audio and video) over the same ports. When multiple RTP streams are multiplexed, payload types and SSRCs must not overlap across all the media streams. The SDP issues with multiplexing are discussed in [draft-ietf-mmusic-sdp-bundle-negotiation].

8.4.2 "WebRTC Data Channel Protocol" [draft-jesup-rtcweb-data-protocol]

This individual Internet-Draft [draft-jesup-rtcweb-data-protocol] defines the operation of the WebRTC data channel over SCTP transport. A *DATA_CHANNEL_OPEN* message is defined to open a new data channel. Three new SCTP Payload Protocol Identifiers (PPIDs) are also defined.

8.4.3 "Cross Stream Identification in the Session Description Protocol" [draft-alvestrand-rtcweb-msid]

This individual Internet-Draft document [draft-alvestrand-rtcweb-msid] defines the *msid* attribute extension to the *a=ssrc* source specific attribute extension (see Section 9.2.3). This extension is used to associate a *MediaStream* to an SSRC. As discussed in the WebRTC (Peer Connection) draft, a *MediaStream* is made up of multiple *MediaStreamTrack*s. An example of this is shown in the example SDP of Section 9.2.1. More work is needed to finalize this extension.

8.4.4 "IANA Registry for RTCWeb Media Constraints" [draft-burnett-rtcweb-constraints-registry]

This individual Internet-Draft [draft-burnett-rtcweb-constraints-registry] defines an Internet Assigned Numbers Authority (IANA) registry to track all media constraints and capabilities used in both the *RTCPeerConnection* and *getUserMedia* interfaces.

8.4.5 "A Google Congestion Control for Real-Time Communication on the World Wide Web" [draft-alvestrand-rtcweb-congestion]

This individual Internet-Draft [draft-alvestrand-rtcweb-congestion] describes Google's current implementation of congestion control used in their WebRTC open source project.

Congestion control is about the behavior of protocols and applications when packet loss occurs on the Internet. TCP has very advanced congestion control algorithms that have been developed over many

decades of Internet experience. In very simple terms, TCP slowly ramps up its throughput. When packet loss occurs, TCP reduces its throughput and backs off, and slowly ramps up again. This allows a number of different TCP flows to share available bandwidth. UDP does not have any built-in congestion control, and WebRTC media flows utilize UDP transport.

This has not really been much of a problem so far in Internet Communication for two reasons. First, these deployments have mainly been VoIP, where the audio bandwidth is quite small. Also, most VoIP systems have used fixed-rate telephony codecs which cannot adapt even if they are aware of congestion.

WebRTC is likely to see a significant deployment of high definition video communications, which uses a significant bandwidth and is also bursty (due to being a mixture of infrequent but large I-frames or key frames, with more frequent smaller P-frames and B-frames). In addition, WebRTC uses Internet codecs such as Opus that can operate over a wide range of bandwidth and can adapt without signaling. The main feedback mechanism is RTCP messages received from the remote peer giving information about packet loss, delay, and delay variation.

There is significant work in the IETF to develop new congestion control algorithms suitable for RTP over UDP flows. These algorithms will use network statistics derived from the RTCP feedback to estimate conditions and detect congestion before packet loss occurs. This information will then be used to rate-limit the bandwidth used by the SRTP media flows and the data channel flows.

The approach described in this Internet-Draft uses RTCP feedback based on Temporary Maximum Media Stream Bit Rate Request (TMMBR, pronounced “timber”, described in Section 9.1.11) . The approach uses signal processing to detect congestion before packet loss occurs. The approach uses the TCP Friendly Rate Control (TFRC) described in Section 9.1.12.

The IETF will attempt to standardize a congestion control algorithm for RTP in the future suitable for use in WebRTC browsers. In July 2012, the Internet Architecture Board (IAB) held a workshop on “Congestion Control for Interactive Real-Time Communication”. Information about the workshop including a link to the papers discussed is available on the IAB website [IAB-CCIRTC]. The RMCAT Working Group has been formed to work on congenstion control for RTP media.

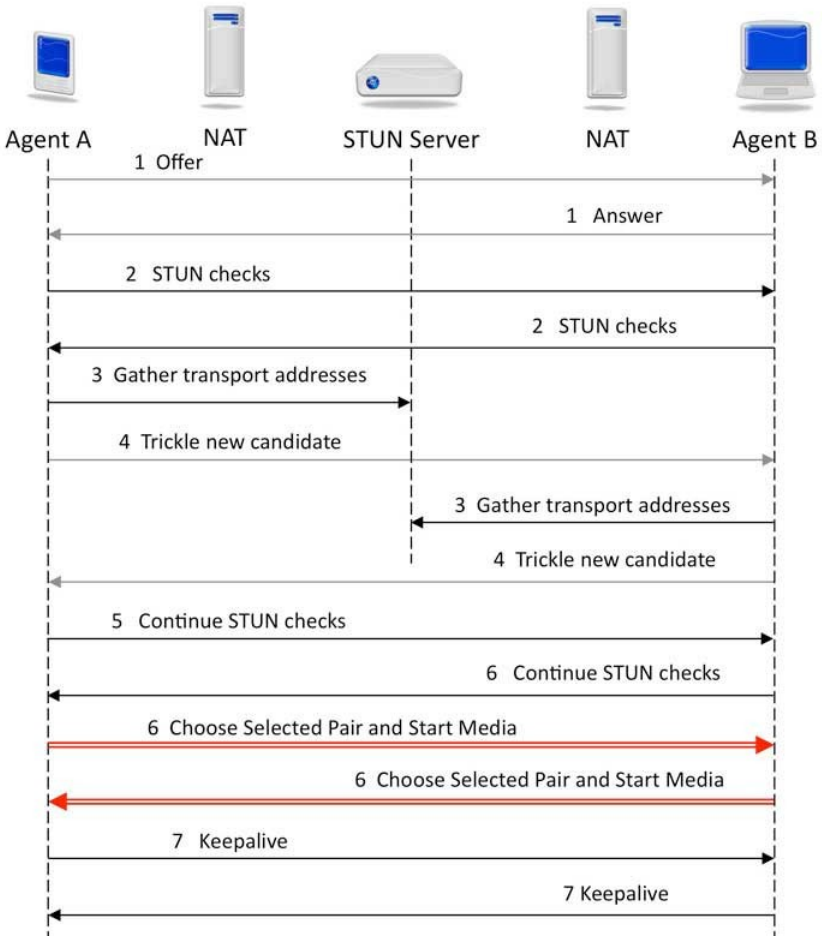
8.5 RTCWEB Documents in Other Working Groups

Some of the protocols being developed for use in WebRTC are being worked on in working groups other than RTCWEB, as discussed in

Section B.3. The Internet-Drafts are listed in this section.

8.5.1 "Trickle ICE: Incremental Provisioning of Candidates for the Interactive Connectivity Establishment (ICE) Protocol" [draft-ivov-mmusic-trickle-ice]

Trickle ICE [draft-ivov-mmusic-trickle-ice] is an optimization for ICE designed to shorten the time taken for ICE to complete. Trickle ICE is shown in Figure 8.7.



1) Exchange Host Candidate over Signaling Channel
2) Start Connectivity Checks
3) Gather More Transport Addresses

4) Trickle In New Candidate(s)
5) Continue Connectivity Checks
6) Choose Selected Pair and Begin Media
7) Send Keepalives

Figure 8.7 Trickle ICE

Instead of waiting until all candidates are gathered before beginning ICE, as shown in Figure 3.10, Trickle ICE begins ICE processing as soon as at least one candidate is available. A host candidate can be gathered the fastest, since it doesn't involve sending any STUN or TURN messages. The host candidates can be exchanged in the offer/answer exchange and ICE processing begun right away. In parallel, other candidates, such as server reflexive addresses from a STUN server, can be added later, or "trickled" in. If it appears that neither the host or reflexive candidate pairs will work, a relay candidate can be obtained and also trickled in. This has an additional advantage of not reserving relay candidates, except when it is likely that they will be needed.

8.5.2 "Multiplexing Negotiation Using Session Description Protocol (SDP) Port Numbers" [draft-ietf-mmusic-sdp-bundle-negotiation]

This MMUSIC Working Group Internet-Draft [draft-ietf-mmusic-sdp-bundle-negotiation], discussed in the MMUSIC Working Group, defines a new SDP grouping framework extension called BUNDLE (*a=group:BUNDLE*). SDP grouping is discussed in Section 9.2.4. This grouping allows the grouped *m=* media lines to share the same port number. WebRTC will use this to signal the multiplexing of multiple media types on the same ports, as described in Section 8.4.1. Backwards compatibility has not been fully designed yet, so this mechanism may change.

8.5.3 "Mechanisms for Media Source Selection in the Session Description Protocol (SDP)" [draft-lennox-mmusic-sdp-source-selection]

This individual Internet-Draft [draft-lennox-mmusic-sdp-source-selection] extends the Source Specific Media Attributes described in Section 9.2.3 to allow selection of a media stream by source. The *a=remote-ssrc* SDP attribute is used to select a particular stream and set a particular attribute. This specification also defines a number of media attributes that can be set using this mechanism, including *recv*, *framerate*, *imageattr*, and *priority*. The *recv* attribute is used to enable

(*recv:on*) or disable (*recv:off*) a source. If the *recv* attribute is not present, then *recv:on* is assumed. The *framerate* attribute is used to request a particular frame rate for a video stream. The *imageattr* attribute is used to set image resolution of a media source. The *priority* attribute is used to set the relative priory among the media sources. If bandwidth or other limitations prevent receiving all the requested sources, the priority is used to decide which sources should be omitted or scaled back.

This specification also defines two new parameters to be used for Source Specific Media attributes: *information* and *sending*. The *information* attribute is used to provide human readable text about a media source, in a similar way to the *i=* field in SDP. The *sending* attribute indicates the sending state of a particular source, either *on* or *off*. A source may be off due to it being disabled by the receiver, or the sender may just no longer wish to send the source.

In offer/answer SDP exchanges, all parties should list all available sources. Note that sources can be discovered via other mechanisms such as receipt of SSRC in RTP or through other conferencing notifications.

8.5.4 "The WebSocket Protocol as a Transport for the Session Initiation Protocol (SIP)" [draft-ietf-sipcore-sip-websocket]

This SIPCORE Working Group Internet-Draft [draft-ietf-sipcore-sip-websocket] defines a WebSocket transport for SIP. WebSocket, described in Section 6.2.2, allows a Web browser to open a new connection to the web server. While this specification is not required for WebRTC, it is related in that it allows Session Initiation Protocol (SIP) to be used as the signaling protocol. The SIP User Agent (UA) stack would be written, for example, in JavaScript and downloaded by the web server. The SIP UA would then use WebSockets to open a new connection to the SIP Proxy Server. The media from the SIP signaling would use the normal WebRTC methods, e.g. a Peer Connection to establish media sessions. This specification defines a new Via transport token *WS* (WebSocket) and new SIP URI transport parameters *ws* (WebSocket) and *wss* (Secure WebSocket) which uses TLS transport.

Note that SIP signaling between Web Servers, as shown in the WebRTC Trapezoid of Figure 1.4, would most likely not use WebSocket transport, and instead would use normal SIP transport such as TCP or UDP.

8.5.5 "STUN Usage for Consent Freshness and Session Liveness" [draft-muthu-behave-consent-freshness]

This individual Internet-Draft [draft-muthu-behave-consent-freshness] discusses some important potential changes to the way in which a browser determines if a multimedia session is still alive, and if the other party wishes to continue to receive the negotiated media. The document extensions to ICE to require that ICE keep-alive Binding responses be processed, and a failure to receive the response will result in an ICE restart. This work is being discussed in both the BEHAVE Working Group (which works on NAT issues) and in the RTCWEB Working Group.

8.5.6 "Multiple Media Types in an RTP Session" [draft-ietf-avtcore-multi-media-rtp-session]

This Internet-Draft [draft-ietf-avtcore-multi-media-rtp-session] explains how RTP can be multiplexed, the issues discussed in RFC 3550 against multiplexing, and possible approaches to overcome them. It is being discussed in both the AVTCORE Working Group and the RTCWEB Working Group.

8.5.7 "Multiple RTP Sessions on a Single Lower-Layer Transport" [draft-westerlund-avtcore-transport-multiplexing]

This individual Internet-Draft [draft-westerlund-avtcore-transport-multiplexing] defines a shim layer ("wedged" between two layers) for multiplexing multiple RTP sessions on the same transport address. A one (or possibly four) octet Session ID is inserted (shimmed) between the transport header (UDP) and the start of the SRTP header. Unfortunately, this makes the on-the-wire format incompatible with RTP, and it is virtually a new protocol. The Session ID is signaled in SDP using an *a=session-mux-id* attribute and *a=group:SHIM* attribute. An alternative approach using the RTP SSRC to demultiplex multiple streams is also under consideration in WebRTC. So far, no clear consensus has emerged in the working group. This draft is being discussed in both the AVTCORE Working Group and the RTCWEB Working Group.

8.5.8 "Random algorithm for RTP CNAME generation" [draft-rescorla-random-cname]

This individual Internet-Draft [draft-rescorla-random-cname] defines a new algorithm for generating random CNAMEs, canonical names which are sent over RTCP and used to identify RTP endpoints in a RTP session. This is important for media privacy in WebRTC where CNAMEs could be used to identify senders across sessions and across applications.

8.5.9 "Multimedia Congestion Control: Circuit Breakers for Unicast RTP Sessions" [draft-ietf-avtcore-rtp-circuit-breakers]

This Internet-Draft [draft-ietf-avtcore-rtp-circuit-breakers] discusses the congestion conditions under which RTP senders should stop sending to avoid making congestion worse. This protection is analogous to the use of circuit breakers to interrupt the flow of excessive current in an electrical circuit. WebRTC will also eventually have congestion control algorithms to reduce traffic before this condition is reached.

8.5.10 "Support for Multiple Clock Rates in an RTP Session" [draft-ietf-avtext-multiple-clock-rates]

This AVTEXT Working Group Internet-Draft [draft-ietf-avtext-multiple-clock-rates] provides guidance in the event that the clock rate changes for an SSRC in a media session. This could happen in WebRTC when switching between Opus and PCM codecs, for example, which use different clock rates.

8.6 References

[RFC-EDITOR] http://www.rfc-editor.org

[draft-ietf-rtcweb-overview] http://tools.ietf.org/html/draft-ietf-rtcweb-overview

[draft-ietf-rtcweb-use-cases-and-requirements] http://tools.ietf.org/html/draft-ietf-rtcweb-use-cases-and-requirements

[draft-ietf-rtcweb-rtp-usage] http://tools.ietf.org/html/draft-ietf-rtcweb-rtp-usage

[draft-ietf-rtcweb-security-arch] http://tools.ietf.org/html/draft-ietf-rtcweb-security-arch

[MEETECHO] http://www.meetecho.com

[RFC5764] http://tools.ietf.org/html/rfc5764

[draft-ietf-rtcweb-security] http://tools.ietf.org/html/draft-ietf-rtcweb-security

[draft-ietf-rtcweb-data-channel] http://tools.ietf.org/html/draft-ietf-rtcweb-data-channel

[draft-ietf-rtcweb-jsep] http://tools.ietf.org/html/draft-ietf-rtcweb-jsep

[RFC3264] http://tools.ietf.org/html/rfc3264

[SDP-OA] Chapter 13 of SIP: Understanding the Session Initiation Protocol, 3rd Edition.

[draft-lennox-rtcweb-rtp-media-type-mux] http://tools.ietf.org/html/draft-lennox-rtcweb-rtp-media-type-mux

[draft-ietf-rtcweb-audio] http://tools.ietf.org/html/draft-ietf-rtcweb-audio

[RFC6176] http://tools.ietf.org/html/rfc6176

[RFC3551] http://tools.ietf.org/html/rfc3551

[RFC4733] http://tools.ietf.org/html/rfc4733

[H.264] http://www.itu.int/rec/T-REC-H.264

[draft-burman-rtcweb-h264-proposal] http://tools.ietf.org/html/draft-burman-rtcweb-h264-proposal

[VP8] http://tools.ietf.org/html/rfc6386 Note that although this is an RFC, it is not an IETF document, instead it is an Independent Stream Submission

[draft-alvestrand-rtcweb-vp8] http://tools.ietf.org/html/draft-alvestrand-rtcweb-vp8

[draft-ietf-rtcweb-qos] http://tools.ietf.org/html/draft-ietf-rtcweb-qos

[draft-jesup-rtcweb-data-protocol]http://tools.ietf.org/html/draft-alvestrand-rtcweb-msid

[draft-alvestrand-rtcweb-msid] http://tools.ietf.org/html/draft-jesup-rtcweb-data-protocol

[draft-burnett-rtcweb-constraints-registry]
http://tools.ietf.org/html/draft-burnett-rtcweb-constraints-registry

[draft-alvestrand-rtcweb-congestion] http://tools.ietf.org/html/draft-alvestrand-rtcweb-congestion

[IAB-CCIRTC] http://www.iab.org/activities/workshops/cc-workshop/

[draft-ivov-mmusic-trickle-ice] http://tools.ietf.org/html/draft-ivov-mmusic-trickle-ice

[draft-ietf-mmusic-sdp-bundle-negotiation]
http://tools.ietf.org/html/draft-ietf-mmusic-sdp-bundle-negotiation

[draft-lennox-mmusic-sdp-source-selection]
http://tools.ietf.org/html/draft-lennox-mmusic-sdp-source-selection

[draft-ietf-sipcore-sip-websocket] http://tools.ietf.org/id/draft-ietf-sipcore-sip-websocket

[draft-ietf-avtcore-srtp-encrypted-header-ext]
http://tools.ietf.org/html/draft-ietf-avtcore-srtp-encrypted-header-ext

[draft-muthu-behave-consent-freshness]
http://tools.ietf.org/html/draft-muthu-behave-consent-freshness

[draft-ietf-avtcore-multiplex-guidelines]
http://tools.ietf.org/html/draft-ietf-avtcore-multiplex-guidelines

[draft-westerlund-avtcore-transport-multiplexing]
http://tools.ietf.org/html/draft-westerlund-avtcore-transport-multiplexing

[SILK] http://developer.skype.com/silk

[CELT] http://www.celt-codec.org

[draft-spittka-payload-rtp-opus] http://tools.ietf.org/html/draft-spittka-payload-rtp-opus

[draft-rescorla-random-cname] http://tools.ietf.org/html/draft-rescorla-random-cname

[draft-perkins-avtcore-rtp-circuit-breakers]
http://tools.ietf.org/html/draft-ietf-avtcore-rtp-circuit-breakers

[draft-ietf-avtext-multiple-clock-rates] http://tools.ietf.org/html/draft-ietf-avtext-multiple-clock-rates

9 IETF RELATED RFC DOCUMENTS

WebRTC uses a number of IETF standards and protocols documented in Request For Comments (RFCs). These RFCs were not specifically developed for or used by WebRTC.

9.1 Real-time Transport Protocol RFCs

9.1.1 "RTP: A Transport Protocol for Real-Time Applications" [RFC3550]

RFC 3550 [RFC3550] defines version 2 of the Real-time Transport Protocol, RTP, and the RTP Control Protocol, RTCP. RTP includes a bit-oriented header field which carries information such as the payload type (codec), timestamp, sequence number, and the synchronization source (SSRC). RTCP messages include Sender Reports (SR), Receiver Reports (RRs), and Source Description (SDES). (Note that the term SDES is an informal name for SDP Security Descriptions, defined in [RFC4568] – the two concepts are unrelated.) The SDES messages carry the Canonical Name (CNAME) which identifies the user in an RTP session. An RTP mixer can provide information about the senders whose media is included in the packet using the contributing source field (CSRC).

9.1.2 "RTP Profile for Audio and Video Conferences" [RFC3551]

RFC 3551 [RFC3551] defines the basic RTP Audio and Video Profile, known as AVP. Formats for a number of common audio and video codecs are defined, along with static payload types (values 0-95). Note that static payload types are no longer allocated – instead dynamic payload types (values 96-127) must be used. This document includes the definition of PCM G.711 audio codec, both A-law and μ-law companding (a portmanteau of compressing and expanding, which results in level compression). This document will need to be updated for use by WebRTC as it recommends DVI4 as an audio codec in addition to G.711.

9.1.3 "The Secure Real-time Transport Protocol (SRTP)" [RFC3711]

RFC 3711 [RFC3711] defines the Secure Audio Video Profile for RTP, known as SAVP. This includes the use of Secure RTP (SRTP) and Secure RTCP (SRTCP). SRTP provides confidentiality and authentication to RTP, using symmetric keys to encrypt and decrypt the media and control messages. SRTP uses the Advanced Encryption

Standard in Counter Mode, AES-CM. SRTP uses 128 bit keys, although it has been extended to allow 192 and 256 bit keys in [RFC6188]. SRTP requires a key management protocol to ensure the sender and receiver have the same symmetric key. WebRTC is considering SDP Security Descriptions (SDES) [RFC4568] and DTLS-SRTP [RFC5764] as potential key management protocols. SRTP generates an encrypted keystream, which is then Exclusive ORed with the media or control packets to encrypt them. This allows the keystream to be generated in parallel with the media or control packets, resulting in minimum added latency. The encryption is applied to the RTP body, leaving the RTP header in the clear, including RTP header extensions. Authentication is provided by an added authentication tag, which can be 0 to 10 octets in length. Each media stream needs to have a unique session key. If a browser sends two video and two audio streams in a session, there will be four unique session keys used to encrypt them.

9.1.4 "Extended Secure RTP Profile for RTCP-Based Feedback (RTP/SAVPF)" [RFC5124]

WebRTC uses the Extended Secure RTP Profile for RTCP-Based Feedback [RFC5124], known as SAVPF. In comparison, most Internet Communications VoIP and video systems today either use the normal Audio Video Profile, AVP, or the Secure Audio Video Profile, SAVP. The SAVPF profile combines the security of SRTP from SAVP [RFC3711], and the timely feedback of the AVPF profile [RFC4585]. The basic AVP profile defined in [RFC3551] includes RTCP feedback messages, and has a mechanism to ensure that excessive bandwidth is not used for these control messages, even for large conferences. One drawback of this is that feedback messages cannot always be sent by the receiver when they would be most useful to the sender. To improve the timeliness of this feedback, AVPF introduces the concept of early RTCP packets and an additional RTCP message known as a feedback message (*FB*) which can be useful for codecs. The *a=rtcp-fb* SDP attribute is used to signal which *FB* messages are to be used. The AVPF profile can interoperate with AVP profiles. However, SAVP cannot interoperate with AVP profiles, due to the lack of support for best effort encryption.

9.1.5 "Multiplexing RTP Data and Control Packets on a Single Port" [RFC5761]

This document [RFC5761] describes how to multiplex RTP and RTCP on the same port. This is done for NAT traversal reasons, minimizing the number times "hole punching" needs to be done. A method to

negotiate this in SDP using the attribute *a=rtcp-mux* is described.

9.1.6 "A Real-time Transport Protocol (RTP) Header Extension for Mixer-to-Client Audio Level Indication" [RFC6465]

RFC 6465 [RFC6465] can be used by a mixer in WebRTC to indicate to a WebRTC user agent the audio levels in a mixed audio conference. An audio media mixer receives RTP streams and combines them into a single stream. The mixer usually implements a mixing policy such as the three loudest active speakers. The RTP packet can contain the CSRC (Contributing Source identifiers) indicating which participants contributed to the mixed packet. This RTP header extension adds to this information by providing the audio level of each participant included in the mix. The level is encoded in 7 bits as dBov, which is the level, in decibels, relative to the overload point of the system (the maximum loudness). The presence of this RTP header extension is negotiated using the approach described in [RFC5285]. This information can be rendered against the conference participant roster, for example, for active speaker identification. This information could have been determined from the participant RTP packet using [RFC6464].

9.1.7 "A Real-time Transport Protocol (RTP) Header Extension for Client-to-Mixer Audio Level Indication" [RFC6464]

This specification [RFC6464] can be used to simplify the operation of a mixer in a WebRTC conference. This RTP extension provides a way for a participant in a conference to indicate the audio level of the packet sent to the mixer. This information is useful for a mixer to quickly select which streams to include in the mix, or a media selector to quickly choose which streams will be selected without having to decode the media packet. The level is encoded in 7 bits as dBov, which is the level, in decibels, relative to the overload point of the system (the maximum loudness). The presence of the RTP header extension is negotiated using the approach described in [RFC5285]. This information could be copied into a mixed packet using the approach of [RFC6465].

9.1.8 "Rapid Synchronization of RTP Flows" [RFC6051]

At the start of an RTP session, there is a period of synchronization between the RTP senders and RTP receivers. For a simple two-party session, this occurs rapidly. However, for multiparty sessions, this can take longer. This document [RFC6051] discusses the issues in synchronization and redefines RTCP timing and new *FB* (FeedBack) messages to speed up this process. This could be useful in large

multiparty WebRTC sessions.

9.1.9 "RTP Retransmission Payload Format" [RFC4588]

In cases where the latency requirements of a media stream are not strict, this technique for requesting retransmission of lost RTP packets can be used. The SAVPF profile is necessary, which allows for rapid RTCP *FB* (FeedBack) packets to be sent. Since WebRTC is about real-time communications, where low latency is necessary, it is far from obvious how this approach could be useful.

9.1.10 "Codec Control Messages in the RTP Audio-Visual Profile with Feedback RTP/AVPF" [RFC5104]

This document [RFC5104] describes how to send Codec Control Messages (CCM) using the AVPF profile. These codec control messages can be used for H.271 Video Back Channel, Full Intra Request (FIR), Temporary Maximum Media Stream Bit Rate (TMMBR), and Temporal-Spatial Trade-off. TMMBR can used for congestion control, as discussed in Section 8.4.5. The FIR is used by a video receiver to request that a video sender send an I-frame when video switching has taken place. The *ccm* parameter is defined for use in the *a=rtcp-fb* attribute. For example *a=rtcp-fb:98 ccm fir* would be used to indicate support of FIR CCM for payload 98.

9.1.11 "TCP Friendly Rate Control (TFRC): Protocol Specification" [RFC5348]

This document [RFC5348] describes a congestion control mechanism for real-time UDP traffic that shares bandwidth fairly with TCP flows. This mechanism relies on feedback from the receiver to the sender about the packet loss rate and round trip time (RTT). The sender then computes the TCP throughput using the TCP Throughput Equation and adjusts its transmit rate to match this. WebRTC approaches incorporating congestion control will likely make use of this specification.

9.1.12 "A General Mechanism for RTP Header Extensions" [RFC5285]

The RTP specification [RFC3550] allows RTP header extensions, but does not specify how to signal or negotiate them or allow more than one extension per RTP packet. This document [RFC5285] partitions the RTP header extension, allowing for more than one, and describes how they

can be signaled using the SDP *a=extmap* attribute. If RTP header extensions such as [RFC6464] and [RFC6465] are used, then this header extension mechanism will be required in WebRTC.

9.1.13 "Guidelines for the Use of Variable Bit Rate Audio with Secure RTP" [RFC6562]

This document [RFC6562] discusses issues with variable bit rate encoding (VBR) and encrypted media. Variations in the rate and size of variable bit rate audio packet streams can leak information about the information content, even when encrypted. The use of RTP padding to protect against this is discussed. PCM (G.711) is a constant bit rate (CBR) codec, while Opus can operate in VBR or CBR mode.

9.1.14 "Support for Reduced-Size Real-Time Transport Control Protocol (RTCP): Opportunities and Consequences" [RFC5506]

This document [RFC5506] discusses the conditions under which reduced-size RTCP packets (i.e. non-compound packets) can be sent. The use of the *a=rtcp-rsize* SDP attribute is used to indicate support for reduced-sized RTCP packets. This allows more RTCP packets to be exchanged using the same bandwidth percentage as full-sized RTCP packets. WebRTC uses RTCP feedback for a number of purposes.

9.1.15 "Encryption of Header Extensions in the Secure Real-Time Transport Protocol (SRTP)" [RFC6904]

Normal SRTP, described in Section 6.2.3, does not encrypt RTP header extensions, although header extensions are authenticated. This document [RFC6904] defines how to encrypt RTP header extensions. If RTP header extensions, such as those in Section 9.1.6 and Section 9.1.7, are used in WebRTC and need privacy, then this extension may be used by WebRTC. Encryption of the RTP header is specified using the *a=extmap* SDP extension described in Section 9.1.14.

9.2 Session Description Protocol RFCs

9.2.1 "SDP: Session Description Protocol" [RFC4566]

This specification [RFC4566] defines version 0 of the Session Description Protocol, SDP. SDP session descriptions are used in WebRTC to represent a media stream offer or answer, and is transported and manipulated using JSEP (Section 8.3.7). SDP provides a way to describe media sessions in terms of connection IP address and port, media types, codecs, and configuration information. However, a number

of SDP extensions are needed in WebRTC. The SIP usage of SDP to negotiate sessions is known as the offer/answer protocol. The syntax of SDP and SDP extensions is defined using ABNF.

A minimal SDP session description is shown below, which includes an IPv6 address:

```
v=0
o=alice 2890844526 2890844526 IN IP4 client.digitalcodexllc.com
s=-
c=IN IP6 FF1E:AD32::72EF:8D21:B866
t=0 0
m=audio 49178 RTP/AVP 98
a=rtpmap:98 OPUS/48000
```

Here is an SDP session description generated by a WebRTC browser:

```
v=0
o=mz23.0a1 12536 0 IN IP4 0.0.0.0
s=WebRTC Call
t=0 0
a=ice-ufrag:670fffde
a=ice-pwd:179059f3d3fecb206c0124cafb1c602c
a=fingerprint:sha-256
F8:7A:A9:78:33:3F:18:85:C0:A3:6E:A8:2D:09:30:20:D2:CD:2E:B7:76:
02:78:E0:30:57:E6:5D:89:D2:71:E0
m=audio 52314 RTP/SAVPF 109 101
c=IN IP4 77.100.43.84
a=rtpmap:109 opus/48000/2
a=ptime:20
a=rtpmap:101 telephone-event/8000
a=fmtp:101 0-15
a=sendrecv
a=candidate:0 1 UDP 2111832319 192.168.1.39 49818 typ host
a=candidate:1 1 UDP 1692467199 77.100.43.84 52314 typ srflx raddr
192.168.1.39 rport 49818
a=candidate:0 2 UDP 2111832318 192.168.1.39 49819 typ host
a=candidate:1 2 UDP 1692467198 77.100.43.84 52315 typ srflx raddr
192.168.1.39 rport 49819
m=video 52316 RTP/SAVPF 120
c=IN IP4 77.100.43.84
a=rtpmap:120 VP8/90000
a=sendrecv
```

```
a=candidate:0 1 UDP 2111832319 192.168.1.39 49820 typ host
a=candidate:1 1 UDP 1692467199 77.100.43.84 52316 typ srflx raddr
192.168.1.39 rport 49820
a=candidate:0 2 UDP 2111832318 192.168.1.39 49821 typ host
a=candidate:1 2 UDP 1692467198 77.100.43.84 52317 typ srflx raddr
192.168.1.39 rport 49821
```

A more complicated WebRTC example is shown below, which corresponds to the streams and tracks of Figure 5.1 and the pseudo code of Sections 2.3.1 and 2.3.2. Note that the *a=candidate* lines would not be line wrapped in real SDP.

```
v=0
o=alice 2890844526 2890844526 IN IP4 browserm.example.com
s=-
c=IN IP4 203.0.113.4
t=0 0
a=ice-pwd:asd88fgpdd777uzjYhagZg
a=ice-ufrag:8hhY
a=group:BUNDLE 1 2
m=audio 49178 RTP/SAVPF 99
a=rtpmap:99 opus/48000
a=mid:1
a=ssrc:43218 cname:GWdiw91ksCqSDw
a=ssrc:43218 msid:F8kdls a2
a=ssrc:43218 msid:3dfdf2 a1
a=ssrc:43218 msid:8dFlf a0
a=ssrc:43218 information: microphone
a=rtcp-mux
a=candidate:1 1 UDP 2130706431 192.168.0.5 7381 typ host
a=candidate:2 1 UDP 1694498815 203.0.113.4 49178 typ srflx raddr
192.168.0.5 rport 7381
m=video 49178 RTP/SAVPF 98
a=rtpmap:98 VP8 90000
a=mid:2
a=ssrc:39322 cname:dkJdiElNw+7dCUqL
a=ssrc:39322 msid:8dFlf v0
a=ssrc:39322 information: front camera
a=ssrc:93847 cname:8dBtKjd2sqqPnzRd
a=ssrc:93847 msid:3dfdf2 v1
a=ssrc:93847 information: back camera
a=ssrc:17339 cname: dkdk+3dkwNC31e
```

a=ssrc:17339 msid:F8kdls v2
a=ssrc:17339 information: presentation
a=rtcp-mux
a=candidate:1 1 UDP 2130706431 192.168.0.5 17632 typ host
a=candidate:2 1 UDP 1694498815 203.0.113.4 56197 typ srflx raddr 192.168.0.5 rport 17632

9.2.2 "Session Description Protocol (SDP) Bandwidth Modifiers for RTP Control Protocol (RTCP) Bandwidth" [RFC3556]

This specification [RFC3556] defines new SDP bandwidth modifiers useful for specifying the bandwidth for RTCP. Normally, RTCP bandwidth is capped at 5% of total bandwidth. The *b=RS* and *b=RR* fields defined in this specification allow for direct specification of RTCP senders and RTCP receivers, respectively. Note that *b=CT* and *b=AS* are defined in [RFC4566] and represent the conference total and application specific bandwidth.

9.2.3 "Source-Specific Media Attributes in the Session Description Protocol (SDP)" [RFC5576]

This specification [RFC5576] allows for the properties of individual media sources in a stream to be specified in SDP. Note that the term "media stream" is slightly confusing. In some contexts, it means a media object defined in SDP by an *m=* line. In other contexts, it means a source of RTP packets. In WebRTC, a number of media sources may be associated with a single *m=* line – this could be multiple streams from the same user, or multiple users contributing streams. This specification defines a media source as an SSRC in RTP. This specification defines the *a=ssrc* attribute which allows the properties of an SSRC to be declared. Properties such as CNAME (*cname*), previous SSRC (*previous-ssrc*), and format-specific parameters (*fmtp*) are defined. This is being extended, as described in Section 8.5.2, with additional parameters of interest in WebRTC.

9.2.4 "Negotiation of Generic Image Attributes in SDP" [RFC6236]

This specification [RFC6236] defines the *a=image-attr* SDP attribute used to negotiate image attributes. For example, consider:

a=imageattr:97 send [x=800,y=640,sar=1.1,q=0.6] [x=480,y=320] recv [x=330,y=250]

This attribute sets for payload 97 the send and receive image sizes, in

pixels. For sending, two possible image sizes are offered. The first has a storage aspect ration (*sar*) of 1.1 and a preference value of 0.6. The second one has the default *sar* (1.0 for square pixels) and a default preference of 0.5. In addition, the range of acceptable picture aspect ratio (*par*) can also be set. This is being extended, as described in Section 8.5.2, to allow this to be source specific, which may be used in WebRTC to modify the attributes of each media source.

9.3 NAT Traversal RFCs

9.3.1 "Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols" [RFC5245]

As described in Chapter 4, WebRTC uses Interactive Connectivity Establishment (ICE) [RFC5245] for NAT traversal and media authorization. ICE is a standardized protocol for "hole punching" – a technique developed in the gaming world to establish a peer-to-peer connection between two hosts behind NAT. Each host gathers potential address candidates: local addresses (read from its NIC, network interface card, interfaces), reflexive addresses (determined from a STUN server), and relay addresses (obtained from a TURN server or other media relay), as shown in Figure 3.8. These candidates are prioritized, encoded as *a=candidate* lines in SDP, then exchanged using a server located in the public Internet known as a rendezvous server. Both hosts then begin sending test packets, sometimes referred to as "hole punching packets", at roughly the same time. As they attempt to send test packets to the other host's candidate addresses, the packets create NAT mappings and filter rules. In many cases, after a few test packets, an end-to-end path through the NATs is obtained, and this connection is then used for the duration of the session. In some cases, due to strict NAT or firewalls, there is no peer-to-peer connection possible. In this case, the TURN media relay address will be used instead. Non-published statistics from service providers who have used ICE or other similar hole punching approaches report that a direct connection can be obtained up to 85% of the time. A TURN server address could be configured in a web browser in a similar way (and for a similar reason) as a web proxy can be configured in browsers today to enable firewall traversal, as described in Section 3.4.

Since a candidate address in ICE will only be used if an authenticated reply hole punching packet is received from the other host, this provides the media authorization needed. Only if the candidate address is expecting and actively trying to establish a session will the candidate

succeed and be used for the session. This prevents the "voice hammer" attack where a candidate address of another host is provided in an attempt to have that host flooded with unwanted traffic.

Connection address candidates are carried in an SDP attribute *a=candidate* along with the type of address. For example:

a=candidate:2 1 UDP 1694498815 192.0.2.3 45664 typ srflx raddr 10.0.1.1 rport 8998

This specification defines the values *host*, *srflx*, *prflx*, and *relay* for host, server reflexive, peer reflexive, and relayed candidates, respectively.

9.3.2 "Symmetric RTP / RTP Control Protocol (RTCP)" [RFC4961]

This document [RFC4916] defines symmetric RTP and RTCP, and provides guidance on when it should be used. RTP is symmetric if packets are sent from the same UDP port to send and receive in a bi-directional RTP session. This is important for traversal of NAT and traversal through TURN and other media relays, such as those provided by SBCs. All media in WebRTC is symmetric as described in Section 3.3.

9.4 Codecs

9.4.1 "Definition of the Opus Audio Codec" [RFC6176]

Opus [RFC6176] is the Internet low latency codec for audio and music known as Opus. Opus incorporates elements and technology from Skype's SILK [SILK] codec and the open source CELT (Constrained Energy Lapped Transform) [CELT] codec. Opus is extremely flexible, supporting bit rates from 6 – 510 kb/s, constant or variable bit rate, sampling rates from 8 – 48 kHz, support for speech and music, mono and stereo, frame sizes from 2.5ms to 60ms, and floating point or fixed-point implementation. Opus also has very good packet loss concealment (PLC) and good quality even during packet loss. The RTP payload for Opus is defined in [draft-ietf-payload-rtp-opus].

9.4.2 "VP8 Data Format and Decoding Guide" [RFC6386]

VP8 [RFC6386] is an open source video codec used in WebRTC. The RTP Payload for VP8 is defined in [draft-ietf-payload-vp8].

9.5 References

[RFC3550] http://tools.ietf.org/html/rfc3550

[RFC4568] http://tools.ietf.org/html/rfc4568

[RFC3551] http://tools.ietf.org/html/rfc3551

[RFC3711] http://tools.ietf.org/html/rfc3711

[RFC6188] http://tools.ietf.org/html/rfc6188

[RFC5764] http://tools.ietf.org/html/rfc5764

[RFC5124] http://tools.ietf.org/html/rfc5124

[RFC4585] http://tools.ietf.org/html/rfc4585

[RFC5761] http://tools.ietf.org/html/rfc5761

[RFC4588] http://tools.ietf.org/html/rfc4588

[RFC6465] http://tools.ietf.org/html/rfc6465

[RFC5285] http://tools.ietf.org/html/rfc5285

[RFC6464] http://tools.ietf.org/html/rfc6464

[RFC6051] http://tools.ietf.org/html/rfc6051

[RFC4588] http://tools.ietf.org/html/rfc4588

[RFC5104] http://tools.ietf.org/html/rfc5104

[RFC5104] http://tools.ietf.org/html/rfc5348

[RFC5888] http://tools.ietf.org/html/rfc5888

[RFC5285] http://tools.ietf.org/html/rfc5285

[RFC6562] http://tools.ietf.org/html/rfc6562

[RFC5506] http://tools.ietf.org/html/rfc5506

[RFC4566] http://tools.ietf.org/html/rfc4566

[RFC3556] http://tools.ietf.org/html/rfc3556

[RFC5576] http://tools.ietf.org/html/rfc5576

[RFC6236] http://tools.ietf.org/html/rfc6236

[RFC5245] http://tools.ietf.org/html/rfc5245

[RFC4961] http://tools.ietf.org/html/rfc4961

[RFC6176] http://tools.ietf.org/html/rfc6176

[RFC6386] http://tools.ietf.org/html/rfc6386

[draft-ietf-payload-vp8] http://tools.ietf.org/html/draft-ietf-payload-vp8

10 SECURITY AND PRIVACY

WebRTC security has a number of aspects that will be introduced and discussed in this chapter. They include:

1) New attacks and compromises that can be launched against a browser by a malicious website.
2) Whether sessions established by WebRTC are secure, and what types of attacks can be launched against them.
3) Whether WebRTC compromises a user's privacy when browsing.
4) Whether allowing WebRTC sessions to cross an enterprise border is secure.

For a complete discussion of the security threats and concepts relating to WebRTC, see [draft-ietf-rtcweb-security] and [draft-ietf-rtcweb-security-arch].

This chapter assumes basic familiarity with security concepts such as privacy, authentication, and security approaches such as encryption, digital signatures, and certificates. Refer to [APPLIED-CRYPTO] for an in depth introduction to these topics.

10.1 Browser Security Model

Any discussion of WebRTC security must begin with the basic web browser security model. In this model, the user must trust their web browser. A malicious browser could redirect a user to web pages they do not wish to visit, log browsing history and data and share that with unauthorized third parties, or gain access to the computer microphone and webcam without telling the user. In short, no security measures provided by protocols or APIs are of any value if they are ignored or implemented incorrectly. Note that this applies to any software installed by users.

Web sites, however, are not necessarily trusted. A user must be able to browse malicious sites and not suffer any harm. While users avoid visiting sites they believe may potentially be malicious, it cannot always be prevented. For example, a user might follow a shortened URL link that results in that user accessing a malicious web site. Or, a link or a website can connect or redirect a user to a malicious site. While a user is visiting a website, that site has control over the screen and can display arbitrary images or text, and can also initiate video or audio streaming. Of course, the user can simply close the browser tab or window to end everything. If a website attempts to download a file, the browser will not allow it unless a user authorizes it by clicking a dialog box. Some operating systems will also warn a user before executing software that

has been downloaded using the browser. All of these features are designed to help protect the user against malicious sites.

10.1.1 WebRTC Permissions

WebRTC provides some similar protections. In particular, a browser is required to verify user consent before providing access to the user's microphone or camera. For many browsers, this means that a WebRTC application wishing to access a user's microphone or camera will not be given access until the user has been prompted and authorization is given. This authorization dialog is handled by the browser itself, and is not under the control of the JavaScript. An example browser confirmation is shown in Figures 10.1 and 8.2. This permission is asked for each session – an application is not given the ability to access the microphone or camera at any time, but just when the user wants to participate in a real-time session. Also, this permission is granted on a per-domain basis. This is important since it is common for a given web window view to have content rendered from many different sites, including additional frames, ads, and tracking software. If multiple domains on a given page wish to have access to the media, then individual permission will need to be obtained for each. This prevents domains piggy-backing on another domain's permission grant.

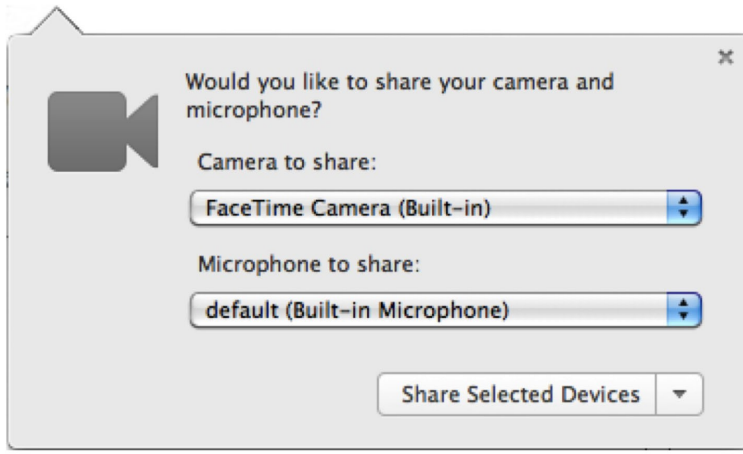


Figure 10.1 Browser Prompting User for Permission

Interestingly, WebRTC JavaScript calls to establish only a data channel do not require user permission. The logic behind this is that by visiting a web site, a user has given the site permission to send arbitrary data to the browser, which is what the data channel allows. A permissions check is needed only to acquire user media, not to transmit it

once it has been acquired.

Enforcing permission grants is only useful if a browser can securely determine which web site is requesting permission. This is discussed in the next section.

10.1.2 Web Site Identity

Early in the history of the World Wide Web, a need was recognized to be able to determine the identity of a web site, and to have encrypted, authenticated sessions with a web site. This led to the development of Secure Sockets Layer or SSL, which has evolved into the TLS protocol, Section 6.2.8. The use of secure web browsing, when Secure HTTP or HTTPS is used between the browser and a web site, is increasingly common today. In the past it was mainly used just during login and for sensitive financial transactions. Today, however, it is becoming the default, and browser plug-ins such as the Electronic Frontier Foundation's HTTPS Everywhere [HTTPS-EVERY] make it easy to ensure the highest level of privacy in web browsing.

The identity of a website providing a WebRTC service or application can be determined if secure browsing (Secure HTTP which utilizes TLS transport) is used rather than regular browsing (plain HTTP which just uses TCP transport). Through the TLS handshake performed when the connection is opened, the browser can learn the identity of the site. In this approach, the web site provides a digital certificate which has been issued by a Certificate Authority (CA), a trust anchor in the browser. Browsers come pre-configured with a set of trust anchor certificate authorities. These certificate authorities issue digital certificates to the holder of a specific domain name, upon proof of ownership and control of that domain name. For example, the owner of the example.com domain can purchase a digital certificate (X.509 certificate) from a certificate authority. A web server in the example.com domain is then able to use the certificate as part of the TLS handshake to prove that it is in the example.com domain. The browser can validate the certificate with the certificate authority to make sure it is authentic. This is an example of a Public Key Infrastructure or PKI.

If the identity validates, the browser displays the padlock or other user interface hint to indicate that this is a secure and authenticated web session. If not, the browser displays a warning and recommends the user disconnect. These types of warnings are most often encountered during login WiFi redirects.

One major problem with this web site authentication model is that a browser will trust any certificate for that website that passes validation. There is no way to know that this is, in fact, the right certificate for the

web site. DANE, DNS-Based Authentication of Named Entities [DANE], is a new security mechanism that uses DNSSEC, DNS Security [DNSSEC], to verify that a digital certificate received in a TLS connection is the correct certificate used by that site.

WebRTC benefits from these web security mechanisms when the JavaScript is provided over an HTTPS connection and the signaling channel is provided over HTTPS or Secure WebSockets (a WSS URI). For example, a long term permission for use of the camera or microphone can be stored if the WebRTC site uses HTTPS. For a non-HTTPS WebRTC site, only a short term, per use, permission grant is possible, possibly requiring the browser to prompt the user for each session in order to verify consent.

10.1.3 Browser User Identity

Unfortunately, the same techniques for web site authentication described in the previous section cannot be used for browser user identity. While it is possible for a browser to present a digital certificate during the TLS handshake, browsers in general do not have installed digital certificates. This is due to cost, ease of use, and the fact that a user has multiple identities, each of which would require a separate certificate. For example, an inexpensive certificate from a trusted certificate authority can cost between $60 and $100 per year – this would be a major expense for most service and application providers.

Instead, other approaches are typically used on the web such as prompting a user for a username/password combination, or the use of a "cookie" from a previously authenticated session. Sometimes a third party identification service is also used. WebRTC can reuse these authentication techniques, as will be discussed later in this chapter.

10.2 New WebRTC Browser Attacks

WebRTC does introduce many new potential attacks on browsers, due to the new APIs, protocols, and the signaling channel. The attacks on each will be described.

10.2.1 API Attacks

WebRTC introduces a number of new JavaScript APIs. Each of these is a potential new attack vector into a browser.

We have already discussed the user consent requirements in order for JavaScript code to request access to the user's camera and microphone. The media capture APIs do not actually give access to the data flowing in a *MediaStreamTrack*, only a handle to the track that can be passed to

other page elements. However, it is relatively simple to access the data flowing on the track. By creating a <*canvas*> containing the user's video, it is then possible to sample the canvas at regular intervals. This allows for some interesting JavaScript-coded effects to be applied to a user's video stream, but it does mean that user consent to access the camera implicitly provides access to the data from the camera as well.

It is also possible to take any stream, whether from camera, microphone, local file, or other source, and stream it over a Peer Connection to another endpoint which can do anything with it that it wants. The Recording API provides even more direct access, allowing for easy saving of streams.

The Media Capture and Streams APIs around capabilities determination are still in flux, but it is likely that an application will be able to get device names in advance of user consent and details about any device for which the user gives consent.

Users must be cautious about the permissions they give. It is likely that, over time, browsers will have settings enabling users to give longer-term permissions for certain sites to access their cameras and microphones. Keep in mind that any permission lasting beyond the time a user is actively operating the device could result in an application using the device at an inopportune time for the user, such as at night or while in a private conversation near the device.

Aside from the general concerns described above, the APIs should not provide new major threats beyond those of any use of JavaScript in a web application. However, these APIs are new, and there are always the possibility of implementation bugs that could turn these APIs into potential attacks on browsers and WebRTC sessions.

10.2.2 Protocol Attacks

Figure 6.1 shows the new protocols present in WebRTC. Each of these protocols provides a new opportunity for attackers. For example, since WebRTC uses SDP, a malformed SDP object could be used to try to crash a browser. WebRTC uses RTP for media transport, and so any attack based on RTP could be utilized. In addition, during an established session, the browser will utilize a codec to decode received audio or video samples. Malformed media samples could be used to try to compromise a browser.

The new architecture of WebRTC also makes some of these attacks even more severe. Before WebRTC, the browser only communicated with the web server. With WebRTC, browsers can establish peer-to-peer connections directly with other browsers, or other devices. This means that the browser needs to not only protect against malicious web sites but

malicious browsers. A perfectly safe WebRTC application could enable an attack by another browser through the Peer Connection.

Another aspect is the potential for a malicious web site to use WebRTC to launch denial of service attacks on other hosts. On possible attack is shown in Figure 10.2.

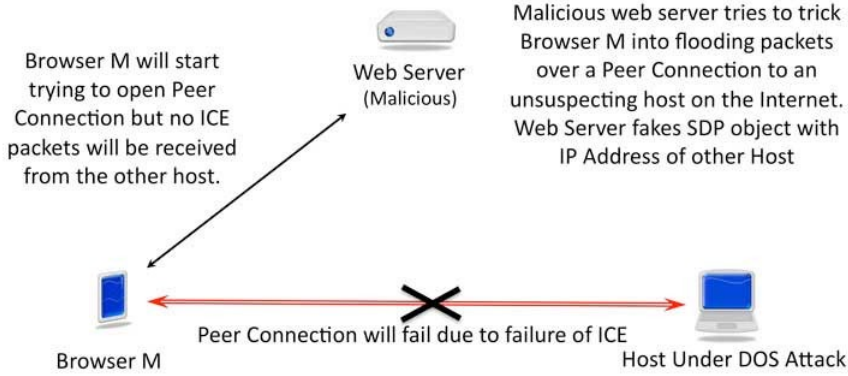


Figure 10.2 Malicious Web Server DOS Attack using WebRTC

A malicious web server could attempt to trick a browser into sending a high bandwidth stream (such as high definition video) to another host. Fortunately, WebRTC requires ICE to indicate consent by the remote site before starting any media stream. A web server cannot spoof those ICE packets, preventing this kind of DOS attack. It is still possible for WebRTC to generate considerable Internet traffic, producing a type of DOS attack on the browser. A user might, for example, authorize a WebRTC session, not realizing how much bandwidth would be consumed. For example, a site could cause a browser to initiate ten HD video streams to a server which would just drop the traffic, leaving the user wondering why their Internet connection is so slow. This could interfere with other Internet activity on that host or other hosts that share a common infrastructure.

ICE is yet another protocol implemented in the browser, so attacks using ICE are also possible, such as malformed ICE packets. Also, the nature of ICE hole punching is such that during this phase of peer connection establishment, the browser will accept and process incoming packets from any destination. Fortunately, once the ICE handshake is complete, the session is locked down to a particular IP address. However, malicious STUN or TURN servers used with ICE could still attempt to disrupt sessions.

10.2.3 Signaling Channel Attacks

WebRTC relies on the signaling channel to establish media and data channels If the signaling channel or signaling server is compromised, an attacker can do a number of things. For example, the attacker could prevent a media or data channel session from being established. Or, the attacker could downgrade the security on the connection. Or, the attacker could redirect the connection to another user, or through a man-in-the-middle device which could record, inject or modify media or data, or otherwise interfere with the session. As such, securing the WebRTC signaling channel is very important.

The way in which the signaling channel for WebRTC is implemented affects what attacks are possible. For example, if WebSockets are used, then potential attacks on WebSockets are possible. If SIP or Jingle is used, then known and new vulnerabilities in these protocol implementations are possible. For a full security analysis of SIP, refer to [VOIP-SEC].

The same origin security model is used in web servers to protect against handling unwanted web traffic. However, having signaling forced through the Web Server is not ideal for bandwidth and performance reasons. As a result, the same origin policy can be relaxed for WebRTC signaling using Cross-Origin Resource Sharing (CORS) [CORS]. WebSockets (Section 6.2.2) allow for a relaxation of the same origin policy. This allows a SIP Proxy or XMPP Server or WebSocket Proxy to be run independently from the Web Server.

For HTTP requests generated by XHR [XHR], CORS can also be used to allow HTTP signaling messages to be routed to a different server than the Web Server. If HTTPS is used, then this server can be authenticated by the browser prior to sending signaling information.

10.3 Communication Security

Communication security represents the security of the real-time communications session established using WebRTC. Privacy and authentication are two important security properties that are enabled with WebRTC.

10.3.1 Communication Privacy

Privacy prevents a third party from eavesdropping on a voice or video communication session. Privacy can be provided by end-to-end encryption. For audio and video sessions, this is provided by default in WebRTC using Secure RTP [RFC3711]. SRTP uses symmetric secret keys to encrypt and decrypt media packets, and commonly uses the AES

encryption cipher in Counter Mode. For the data channel, DTLS provides the encryption. Data channel data can also be encrypted and decrypted in the JavaScript, providing an additional layer of security.

The encrypted and authenticated parts of the SRTP header are shown in Figure 10.3. Note that the RTP header is authenticated, but not encrypted. This means that SRTP packets can be identified as RTP but the media cannot be played or recorded without the key. Allowing SRTP traffic to be detected allows networks to apply policy, and provide Quality of Serivce (QoS) and monitoring services.

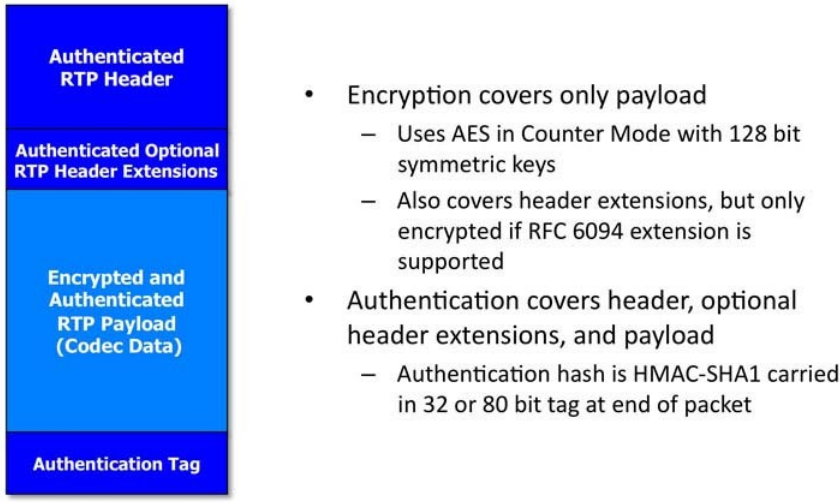


Figure 10.3 Encrypted and Authenticated parts of SRTP Packet

The security of SRTP depends strongly upon the keying method. There are two approaches to keying SRTP: sending the key over the signaling channel, or generating the key in the media path. SDP Security Descriptions is an example of the former, while DTLS-SRTP is an example of the latter.

10.3.2 Key Transport over the Signaling Channel

Sending the key over the signaling channel places security requirements on the signaling channel. If the signaling channel is not encrypted, then the entire SRTP media session will not have privacy. This encryption could be provided by Secure HTTP (HTTPS) or Secure WebSockets (WSS) transport for the signaling channel. In addition, since signaling usually goes through a signaling server, that signaling server must be trusted and must be protected against compromise. In particular, if

logging is performed on a signaling server used to transport SRTP keys, those logs must be secured and encrypted or else SRTP keys could be revealed.

SDP Security Descriptions [RFC4568], informally known as SDES, is the most commonly used signaling protocol for VoIP and video systems that use SRTP. SDP Security Descriptions carries the SRTP master key and salt in the *a=crypto* SDP attribute. For example:

a=crypto:1 AES_CM_128_HMAC_SHA1_32
inline:a0RgdmcmVCspeWpVLFJhfHAQ

This configures SRTP to use AES Counter Mode with 128 bit keys and a 32 bit HMAC SHA-1 authentication tag. Inline contains the key and salt encoded in base 64.

10.3.3 Key Agreement in the Media Path

Key agreement in the media path does not rely on the security of the signaling channel or trust of the signaling server. Instead, keys are generated using a public key operation in the media path (over the same transport IP address and port number as the resulting media session). One example of this public key operation is a Diffie-Hellman (DH) key agreement. DTLS-SRTP [RFC5764] is a media path key agreement that performs this operation during the DTLS handshake. The DTLS handshake occurs after ICE connectivity checks have completed and the candidate pair chosen for the media session. The operation of DTLS-SRTP is shown in Figure 10.4.

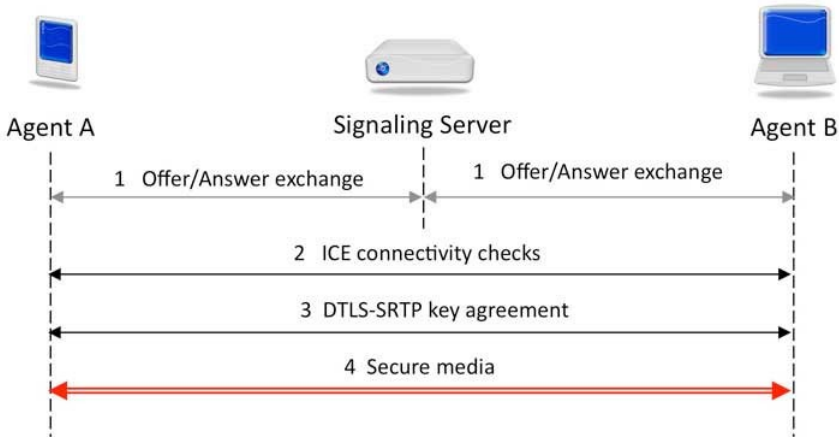


Figure 10.4 Operation of DTLS-SRTP Key Agreement

10.3.4 Authentication

Authentication for the communication session is the ability to determine that a media packet comes from the other party in the session. The HMAC carried inside the authentication tag are part of SRTP's media authentication, as shown in Figure 10.3.

10.3.5 Identity

There are a few ways that the identity of a participant in a WebRTC session can be determined. Both approaches rely on the DTLS-SRTP keying approach. If the browser has a client certificate which is part of a PKI (Public Key Infrastructure), then this certificate can be passed in the DTLS handshake and validated. Having a PKI browser certificate is not common.

If a PKI certificate is not used, a browser uses a so-called self-signed certificate. This is a digital certificate which is not issued by a certificate authority, and is not part of chain of trust. Essentially, it is just a container for a public key which is associated with the private key which will be used for public key operations during the SRTP key negotiation. A finger print, the output of a hash function, of that certificate is included in the SDP offer or answer message. If the signaling channel has end-to-end integrity protection, this can provide authentication, but this is not very common.

10.4 Identity in WebRTC

There are two types of identity that apply in WebRTC. The first is the identity of the website that is providing the WebRTC application. This is applicable for any type of web site, and WebRTC can simply make use of existing browser-based approaches described in Section 10.1.2. The other is the identity of the other end of the Peer Connection. With the addition of WebRTC, this is new to browsers, as most websites only involve interaction directly with a web server, not another browser as WebRTC allows.

WebRTC defines a new identity scheme that is an extension of existing cross-web site ID and single sign on approaches. Some common web-based approaches are listed in Table 10.1. The Identity Provider (IdP) defined in [draft-ietf-rtcweb-security-arch] is a framework for a generalized web identity scheme to be used to provide identity in WebRTC. In theory, each of these approaches could be used with Identity Provider.

Approach	Used By	URL
OpenID	OpenID Foundation	http://openid.net/foundation
BrowserID	Mozilla, part of Persona	https://login.persona.org/
Facebook Login	Facebook	http://developers.facebook.com/docs/facebook-login

Table 10.1 Common Web Identity Schemes

An Identity Provider fulfills a similar role as a trust anchor in server identity. Figure 10.5 shows a simple case of Identity Provider in the WebRTC triangle. Both browsers are connected to the same Web Server and also use the same Identity Provider. Note that there does not need to be any interaction or cooperation between the web site and the identity service – they can be completely independent. However, the signaling channel must pass the identity assertion between the browsers. This allows a WebRTC provider to completely stay out of providing any identity.

A more general case of Identity Provider in the WebRTC trapezoid is shown in Figure 10.6. In this case, there are two separate Web Servers, with a signaling connection between them to allow the establishment of a Peer Connection between browsers in each of their web pages. There are also two separate Identity Providers, each unassociated with the web domain.

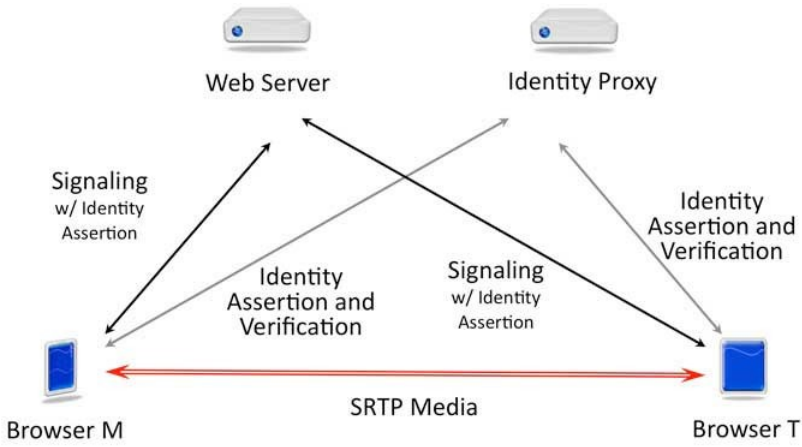


Figure 10.5 Identity Provider in WebRTC Triangle

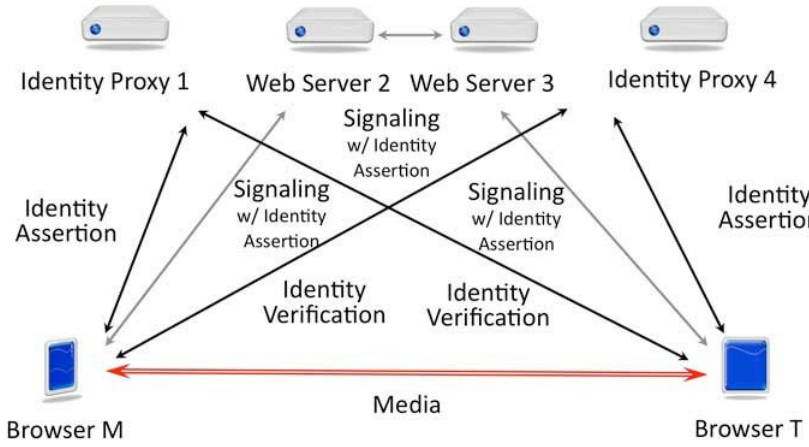


Figure 10.6 WebRTC Identity Provider in WebRTC Trapezoid

Figure 10.7 shows another possible architecture with the WebRTC triangle but separate Identity Providers. This architecture will be used to explain how Identity Provider works. This figure shows a single web domain with two Identity Providers.

In this example, the user at browser M has an account with Identity Provider 1, and has configured his browser to use Identity Provider 1 as his identity provider. The user signs in to Identity Provider 1. When the user at browser M initiates a WebRTC session, the browser contacts Identity Provider 1 for a short-term certificate which is used to generate the identity signature. When the certificate is received, Browser M generates a digital signature over the SRTP keying material to be used to establish the Peer Connection. This signature is sent over the signaling channel with the SDP object to the Web Server. The Web Server forwards the SDP object and the identity signature to Browser T.

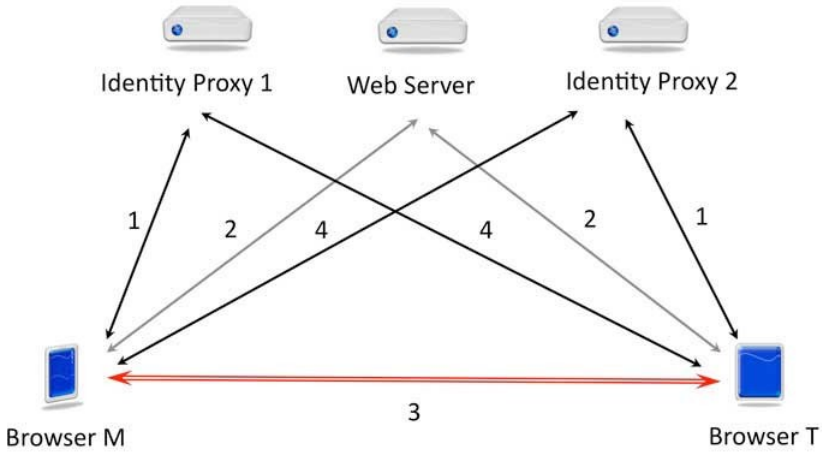


Figure 10.7 Steps in using an Identity Provider in WebRTC

The user at browser T has an account with Identity Provider 2, and has configured her browser to use Identity Provider 2 as her identity provider. The user signs in to Identity Provider 2. Upon receipt of the SDP object, Browser T contacts Identity Provider 2 for a short-term certificate for generating the identity signature. The SDP object is generated, and a signature generated of the SRTP keying material to be used for the session. Both are sent over the signaling channel which forwards them to Browser M. Browser M and Browser T begin ICE hole punching, then SRTP key negotiation. When the keying is complete, each browser checks the keying material used to negotiate the SRTP keys with the signature. The certificate used to sign the identity signature is verified with the other's Identity Provider. If all checks out, the Peer Connection is authenticated and the identity asserted by the Identity Provider can be displayed as an authenticated identity.

Note that this identity assertion is generated and checked by the

browser. The Web server plays no role, on either the downloaded local JavaScript or on the server code itself. The only role the Web server provides is to transport the signature exchange over the signaling channel. Note that the Web Server might also provide an identity indication, which may or may not match the one provided by the Identity Provider.

Needless to say, if a user wants privacy in a session, or to not to have their identities linked across multiple WebRTC sites, they should not enable this identity service.

So far, Mozilla has shown the most interest in Identity Provider and will likely have the first browser supporting it. Depending on its utility and adoption by WebRTC developers, other browsers may add support for it in the future.

See Section 5.3.1.5 for information about Identity Provider APIs.

10.5 Enterprise Issues

Enterprises typically use border security to control what Internet traffic flows across their boundaries. For general IP traffic, firewalls are often utilized. Firewalls are points of policy enforcement for IP access. The simplest access rules are one-way filtering. In general, incoming packets from the Internet are blocked unless they are responses to requests sent from inside the enterprise. Other rules can be set up for particular IP addresses, ports, and transports. The combination of local and remote IP addresses, local and remote ports, and transport protocol (e.g. UDP or TCP) is often referred to as a 5-tuple for the five data points that can characterize a particular protocol. Some protocols always use a particular port, allowing for 5-tuple rules to be used for a particular protocol. For example, web browsing typically uses TCP port 80, so opening port 80 in a firewall can be used to enable web browsing, and blocking port 80 can restrict web browsing. Media flows such as those established by WebRTC, or SIP, or Jingle do not use well-known or registered ports. As such, they cannot be allowed or blocked simply by 5-tuple filtering rules.

For particular application traversal through firewalls, Application Layer Gateways or ALGs are used. For Internet Communications, specialized ALGs known as Session Border Controllers or SBCs are used. Typically, they use a signaling channel, such as SIP or Jingle, to authorize the resulting media flows, such as RTP or SRTP. The SBC is typically placed next to the firewall in a trusted connection known as a DMZ (for De-Militarized Zone, a buffer region used around national borders to prevent direct interaction between opposing military forces.)

An SBC uses the signaling channel to apply policy. For example,

based on the destination IP address or URL, a particular type of media communication can be allowed or blocked. In addition, other policies can be applied, such as whether the media is recorded and archived.

With WebRTC, there is no standardized signaling channel, so a device such as an SBC is limited in what it can do in general, unless a standardized signaling protocol is used. Even if a WebRTC application uses a signaling protocol within the JavaScript, this will not necessarily be accessible to an SBC, as it could be transported over a Secure HTTP or Secure WebSocket connection and mixed in with all the other web traffic going in and out of the enterprise.

For a more in-depth discussion of enterprise WebRTC issues, see [IEEE-COMS].

10.6 Privacy

There are a number of areas in WebRTC that relate to privacy. Identity privacy, IP address privacy, and browser fingerprinting are all important. Useful guidelines for Internet protocols are in [draft-iab-privacy-considerations].

10.6.1 Identity Privacy

Identity privacy in WebRTC means that a user can visit a website and use WebRTC and not have their identity compromised. Normally web sites using WebRTC will use and share identity information as part of the service. However, a web site might promise anonymous or private communication using WebRTC. Of course, a web site needs to know one's identity in order to share it, although even IP addresses can provide some level of identity information. Once a user has signed in to a web site, there is no protocol or API way to enforce an anonymous or private WebRTC session. This is analogous to the PSTN, where the telephone network always knows a caller's identity. While the user can select to block delivery of their identity, this won't always be the case, such as a call to a toll free (8xx) number in which the caller identity is always delivered regardless of the caller's indication.

It is also somewhat obvious that if a private or anonymous WebRTC session is established, that an Identity Provider should not be used.

Identity information could also be leaked in the signaling channel. For example, a SIP *INVITE* request might contain the non-anonymized SIP URI of a user.

10.6.2 IP Address Privacy

Privacy on the Internet is a very complicated subject, and even privacy in

web browsing is also very involved. This book does not discuss topics such as web cookies. However, there is one aspect of privacy that WebRTC affects which we will discuss here: IP address privacy.

Whenever a browser connects to a web sever, the web server knows the IP address of the browser. In most cases, it is actually just the public IP address of the outer-most NAT that the browser is connecting through. For example, in Figure 3.7, if either the mobile or the tablet access a web page, the web server will get the IP address 203.0.113.4 and not the individual IP address that would distinguish the mobile from the tablet. However, this IP address can reveal a lot about the user, such as location information which is commonly used for location specific ads and search results. There are some services which allow a browser to connect to a server without revealing this information. One such example is The Onion Router [TOR], an anonymizing service for browsing. In general, though, a web server always knows the IP addresses of the browsers that connect to it, and it can do anything with this information including log it, store it, or share it with anyone. For example, there are sites that display this information for troubleshooting purposes [WHATSMYIP] or to correlate anonymous postings. WebRTC does not change any of these well-known web properties.

However, if WebRTC is used to establish direct browser-to-browser media flows, then each browser can learn the IP address of the other browser. This can be true even if the session is not established or is refused by the other party. This information can also include the inside IP address, the private IP address, if this address was shared as a candidate address for hole punching. This is a new privacy exposure introduced by WebRTC and could potentially be used to obtain new information about other users.

If privacy is important in the WebRTC application, there are a number of possible ways to improve the situation:

1) A browser could only send one IP address candidate, that of a TURN server, discussed in Section 3.4. This would allow the other browser to learn where the TURN server is located but not the actual browser using the TURN server.
2) A virtual private network, VPN, service could be used and only the VPN IP address shared as a candidate.
3) A browser might choose not to share the private IP address as a candidate. The browser would instead share the public IP address, which might be that of a service provider or enterprise, but not reveal the individual user or computer behind it.

4) A web server could have a policy of intentionally relaying all media traffic, using a TURN server for example. However, users must trust the web site to do this for them.

In the signaling channel, IP address information can be leaked in the content of the signaling message or in the transport of the signaling message. For example, a WebSocket Proxy will learn the IP address of browsers that use it for WebRTC. Since the SDP object will contain ICE IP address candidates, this also provides IP address information. This also provides private IP address information, something not normally known by a web server which normally only sees the IP address of the outside NAT.

It is very important to note that in all web browsing cases, some IP address must be shared with both the web server and the other web browser. The only question is which IP address and how much information this reveals about the user.

10.6.3 Browser Fingerprinting

Browser fingerprinting is the process of trying to identify a particular browser by its capabilities and features. The more information about a browser that a web site can learn, the easier fingerprinting can be. Fingerprinting can be used for tracking that circumvents a browser's cookie policy, for example. The EFF performed a survey of the uniqueness of web browsers using an online test tool [PANOPTICLICK]. In their published report [EFF-FINGER], they list the top sources of browser fingerprinting as plug-ins, fonts, browser, accepted methods, screen resolution, time-zone, and cookies enabled. A discussion from a W3C plenary considers this topic [FINGERPRINT].

WebRTC provides much more information about a browser that could be used for fingerprinting, such as supported media types, codecs, network interfaces, etc. This is an area of active research and standardization work.

10.7 Summary

WebRTC security is a very important topic that will receive much attention as this new technology rolls out. WebRTC can make use of existing web security features such as secure HTTP. It also has built-in security in the media path. It can also make use of browser identity approaches through the Identity Provider mechanism.

It is important that WebRTC browsers support auto updates so that users have the latest patches and updates.

10.8 References

[draft-ietf-rtcweb-security] http://tools.ietf.org/html/draft-ietf-rtcweb-security

[draft-ietf-rtcweb-security-arch] http://tools.ietf.org/html/draft-ietf-rtcweb-security-arch

[APPLIED-CRYPTO] http://www.amazon.com/Applied-Cryptography-Protocols-Algorithms-Source/dp/0471117099

[HTTPS-EVERY] https://www.eff.org/https-everywhere

[DANE] http://www.internetsociety.org/deploy360/resources/dane/

[DNSSEC] http://tools.ietf.org/html/rfc4033

[VOIP-SEC] http://www.amazon.com/Understanding-Security-Artech-Telecommunications-Library/dp/1596930500

[CORS] http://www.w3.org/TR/cors/

[XHR] http://www.w3.org/TR/XMLHttpRequest/

[RFC3711] http://tools.ietf.org/html/rfc3711

[RFC4568] http://tools.ietf.org/html/rfc4568

[RFC5764] http://tools.ietf.org/html/rfc5764

[IEEE-COMS] Alan Johnston, John Yoakum and Kundan Singh, Taking on WebRTC in an Enterprise, IEEE Communications Magazine, Vol. 51, No. 4, April 2013

[draft-iab-privacy-considerations] http://tools.ietf.org/html/draft-iab-privacy-considerations

[TOR] http://theonionrouter.com

[WHATSMYIP] http://whatismyipaddress.com

[PANOPTICLICK] https://panopticlick.eff.org/

[EFF-FINGER] https://panopticlick.eff.org/browser-uniqueness.pdf

[FINGERPRINT] http://www.w3.org/wiki/Fingerprinting

11 WEBRTC IMPLEMENTATIONS

Web browsers are at various stages of supporting WebRTC APIs and protocols. This information changes rapidly, so always do some searching of official documentation to determine the exact support of WebRTC. This information is presented in alphabetical order.

11.1 Apple Safari

No information yet. However, projects such as webrtc4all [WEBRTC4ALL] aim to provide a temporary solution.

11.2 Google Chrome

Support for *RTCPeerConnection()*, *getUserMedia()*, and *MediaStreams* is available in standard Chrome [CHROME] today (as of version 23). As of version 24 Opus and TURN are supported. As of version 25 a limited version of data channels is supported. For details on this, as well as the most up-to-date status on Chrome support of WebRTC, please visit http://www.webrtc.org/chrome. Chrome Canary [CANARY] is the developers preview of new features that will be in future versions of Chrome and often has early WebRTC functionality available for testing.

11.3 Mozilla Firefox

Support for *RTCPeerConnection()*, *getUserMedia()*, *MediaStreams*, and *DataChannels*, is available in standard Firefox [FIREFOX] today (as of version 22). For the most recent documentation on WebRTC support in Firefox, see https://developer.mozilla.org/en-US/docs/WebRTC. Firefox Nightly [NIGHTLY] is the developers preview of new features that will be in future versions of Firefox and often has early WebRTC functionality available for testing.

11.4 Microsoft Internet Explorer

Microsoft has published several versions of an alternative API document [CU-RTC-WEB]. Discussions about integrating this into the existing standards work are ongoing. As a result, it is not yet clear whether or how much of the existing WebRTC APIs will be supported in Internet Explorer. The Google Chrome Frame extension [CHROME-FRAME] can also be used for WebRTC support in Internet Explorer.

11.5 Opera

Opera Mobile 12 [OPERA] has *getUserMedia()*support.

11.6 References

[WEBRTC4ALL] http://code.google.com/p/webrtc4all

[CHROME] http://www.google.com/chrome

[CANARY] http://www.google.com/intl/en/chrome/browser/canary.html

[FIREFOX] http://www.firefox.com

[NIGHTLY] http://nightly.mozilla.org

[CU-RTC-WEB] http://lists.w3.org/Archives/Public/public-webrtc/2012Oct/att-0076/realtime-media.html

[CHROME-FRAME] http://www.google.com/chromeframe

[OPERA] http://www.opera.com/mobile

APPENDIX A – THE W3C STANDARDS PROCESS

The World Wide Web Consortium is developing the standard APIs for WebRTC. Besides the WEBRTC Working Group, there are a number of other W3C working groups and task groups working on WebRTC.

A.1 Introduction to the World Wide Web Consortium

The World Wide Web Consortium (W3C) [W3C] was established by Tim Berners-Lee, the creator of HTML, to promote the development of HTML and other web technologies. Over time, W3C has built up processes that make it a proper Standards Development Organization (SDO), including a focus on consensus, processes for dispute resolution, handling of non-member comments, implementability testing, permanent storage of and access to discussions, resolutions, and official development drafts of specifications. W3C is a membership organization – only members can create Working Groups, and some Working Groups are restricted such that only members can attend meetings and calls. However, all official documents and final discussions are publicly available on the web without charge. All discussions in the WEBRTC Working Group are public.

Although W3C has a variety of group structures for discussing specifications, only a Working Group leads to an officially sanctioned W3C standard, known as a W3C Recommendation. Recommendations are published in (what else?) HTML format. W3C Working Groups are loosely organized into Activity Domains, but that is primarily for internal W3C administration reasons and has little impact on the direction of the group. In addition, there are Task Forces which are informal collaborations between two or more Working Groups. The Media Task Force that is developing the "Media Capture and Streams" document is one such task force.

The two most important factors in W3C standard development are the specification development stages and the focus on consensus. Specifications in W3C, referred to as Technical Reports, progress through the following stages of maturity:

1) First Public Working Draft
2) Working Draft
3) Last Call Working Draft (LCWD)
4) Candidate Recommendation
5) Proposed Recommendation
6) Recommendation

The first three stages together indicate that a Technical Report is not yet complete technically and does not yet represent broad consensus. The last three stages together indicate that the specification is believed to be technically complete and has had broad review.

More specifically, a specification begins its public life as a First Public Working Draft. At this stage there is no consensus, only an Intellectual Property commitment. There are then any number of Working Drafts released regularly by the Working Group as it hashes out the technical contents of the specification. Although an efficient and effective group will plan to build consensus as it develops these Working Drafts, officially, a Working Draft does not represent consensus and must be cited as a Work in Progress.

Once the Working Group believes a specification is technically complete, it publishes the document as a Last Call Working Draft (LCWD). Interestingly, this is often the first time that other W3C groups will review the draft, so it is not unusual to have more than one LCWD before moving forward. It is also the first time that a record must be kept of the disposition of every public comment – the specification may not move forward until every comment has been addressed to the satisfaction of the commenter, except in unusual circumstances.

After passing the LCWD stage, the next step is to produce a Candidate Recommendation document containing a detailed Implementation Report Plan, which is typically a list of assertions and tests along with a plan for how many implementations of each feature are required in order to progress. The goal at this stage is not to test implementations, but to ensure that there have been enough implementations to confirm that the specification is implementable and, ideally, that implementations will be interoperable. Any specification changes beyond clarifications and editorial changes will cause the document to move back to the Working Draft stage to again ensure sufficient review and consensus. After receiving sufficient implementations according to the plan, and with no major changes needed, the Technical Report can move to the penultimate stage, the Proposed Recommendation. This stage is largely a formality and is, in fact, out of the hands of the Working Group. Assuming there are no public objections within this stage (typically a month), the Report moves automatically to the Recommendation stage.

Any discussion of W3C process would be insufficient if it did not include a discussion of consensus. W3C takes the consensus requirement very seriously, requiring that every objection be addressed if at all possible. Although the IETF only requires rough consensus, W3C requires full consensus. While this can increase the time needed to

produce a Recommendation, it reduces the likelihood that the document will knowingly alienates a sizeable fraction of the intended users.

A.2 The W3C WEBRTC Working Group

The main working group in W3C for WebRTC is the WEBRTC Working Group [WEBRTCWG]. Although efforts to create the IETF RTCWEB and the W3C WEBRTC groups began at the same time, it took significantly longer to get the WEBRTC group going. The issue, in this case, was to agree on what to use as a starting document. The WHATWG [WHATWG] is an independent organization that makes suggestions for the direction of HTML development by creating a modification of HTML itself. One such modification was an extension to set up media connections (known as a "Peer Connection") between two browsers. It took a substantial amount of time to work out the copyright issues, but eventually W3C developed a copyright statement that allowed the WEBRTC Working Group to use the WHATWG's PeerConnection text as a starting point for the group's work.

A.3 How WEBRTC relates to other W3C Working Groups

WebRTC's goal is to define APIs for setting up direct browser-to-browser media connections. It is not the goal of WebRTC to define what media is, how media will or could be used by the near or far end, or how it relates to the existing capabilities of HTML. While there are aspects of media synchronization that do need to be defined by WebRTC because of the need to choose synchronized or asynchronous transports, the other aspects listed above are handled by other groups within W3C. Some related groups are listed below.

Media Capture Task Force [MEDIAWG] - The Media Capture Task Force is comprised of members of two W3C Working Groups: WEBRTC and Device APIs. The goal of this group is to jointly define *getUserMedia()*, the API call used to request local media (access to a camera, microphone, speaker, etc.). Additionally, this group is defining the core of the *MediaStream* interface since it is of relevance to both groups.

HTML [HTMLWG] - Clearly the HTML Working Group focuses on the development of the Hyper-Text Markup Language, the language that is the foundation of the World Wide Web. Although there is no direct working

relationship between the WEBRTC and HTML WGs, many of the participants in the WEBRTC group are active participants in or followers of the HTML WG. More importantly, WebRTC participants understand that the WebRTC APIs must be consistent with and integrate well with existing HTML APIs and markup. The current version of HTML is HTML5.

Audio [AUDIOWG] - The Audio Working Group develops APIs for more advanced audio manipulation within HTML. Although there is no direct connection between the WEBRTC and Audio WGs, the Audio Working Group has use cases that affect the *MediaStream* and *getUserMedia* interfaces being defined by the Media Capture Task Force.

A.4 References

[W3C] http://www.w3c.org

[WEBRTCWG] http://www.w3.org/2011/04/webrtc

[WHATWG] http://www.whatwg.org

[MEDIAWG] http://www.w3.org/wiki/Media_Capture

[HTMLWG] http://www.w3.org/html/wg

[AUDIOWG] http://www.w3.org/2011/audio

APPENDIX B – THE IETF STANDARDS PROCESS

The Internet Engineering Task Force is developing standard protocols for WebRTC. Besides the RTCWEB Working Group, there are a number of other IETF working groups working on WebRTC.

B.1 Introduction to the Internet Engineering Task Force

The Internet Engineering Task Force (IETF) [IETF] is the international standards body responsible for protocol standardization on the Internet. The IETF has standardized protocols such as IP, TCP, UDP, DNS, SIP, RTP, HTTP, and SMTP to name some popular ones. The IETF publishes its standards documents as the numbered series known as "Request for Comments" or RFCs. Note that not all RFCs are IETF documents. Also, not all IETF RFCs are standards documents. Before they are finalized and approved as RFCs, working drafts of standards documents are known as "Internet-Drafts".

Work in the IETF is primarily done over email using mailing lists. There are separate mailing lists for each Working Group. Much of the work on WebRTC in the IETF is discussed in the RTCWEB Working Group, although related work is also happening in other working groups.

There are no membership fees, and anyone can contribute to the work by subscribing to a mailing list, sending comments, writing Internet-Drafts, or attending face-to-face IETF meetings. Work is organized into areas known as Working Groups.

The normal process steps for an IETF document are listed below:

1) Submission of individual Internet-Draft
2) Adoption of a Working Group document
3) Working Group Last Call (WGLC)
4) IETF Last Call
5) Approval by IESG as an RFC.

Internet-Drafts are working documents submitted to the IETF via email or the online form on the IETF website. Internet-Drafts must meet specific formatting requirements and have intellectual property and copyright declarations. Internet-Drafts are frequently updated, and automatically expire after six months if they are not updated or finalized as an RFC. The initial version is -00 (counting from zero) and is incremented for each update. As an individual draft, the authors can include any content they like in their draft. Internet-Drafts are identified by their filename, which always begin "draft-lastname-wgname" where "lastname" is the last name of the principle author or editor, and

"wgname" is the name of the working group where the work is likely to be discussed. The rest of the filename is a hyphenated version of the title or content. For example, draft-burnett-rtcweb-constraints-registry is an individual Internet-Draft, written by Daniel C. Burnett for the RTCWEB working group about a constraints registry.

Since WebRTC is a work in progress, many of the documents discussed in this book are Internet-Drafts, and as such their content may have changed. The hyperlinks in this book will automatically take you to the latest version. However, the document name may have changed or documents may have been merged together or split into multiple documents.

Working Groups in the IETF are chartered to produce documents to meet specific protocol milestones. Working Groups "adopt" a draft as a starting point towards producing a consensus document to meet a particular milestone. The authors or editors of the draft are expected to try to reflect working group consensus in the draft from this point on. When the draft is revised, the filename will change to "draft-ietf-wgname", dropping the author name. Since the filename has changed, the version resets to -00. Working group documents tend to get wider review, agenda time at face-to-face IETF meetings, and listings on Working Group pages and summaries.

Once Working Group chairs believe an Internet-Draft is complete and represents the consensus of the group, they will call for a Working Group Last Call (WGLC) for final reviews and comments. If there are significant changes or updates as a result, there may be additional WGLCs for the document. After this process is completed and the chairs believe the draft has "rough consensus" they will move the document towards IETF-wide final review in an IETF Last Call. Upon completion, the members of the Internet Engineering Task Force Steering Committee (IESG) vote. If the vote is successful, the Internet-Draft will be approved and put into the RFC Editor's queue. After a few months, the draft will be assigned an RFC number and published as an RFC.

There are a number of types of RFCs published by the IETF. The most common are Proposed Standards (PS) and Informational documents. Proposed Standards are actual IETF protocol standards. Informational RFCs do not define protocols or standards but instead document requirements, issues, or the motivation behind protocols. Some WebRTC documents will be published as Informational RFCs, although most will be published as Proposed Standards.

B.2 The IETF RTCWEB Working Group

The main Working Group for WebRTC in the IETF is the RTCWEB

Working Group [RTCWEB WG], short for Real-Time Communications Web. However, the WebRTC work encompasses a number of areas, and as such the work is spread across a number of working groups. In addition to Internet-Drafts (working standards documents), WebRTC references other IETF RFCs (Request for Comments, the finished standards documents). Both are also listed and explained in this book.

B.3 How RTCWEB relates to other IETF Working Groups

Besides the work in the RTCWEB Working Group, there is active work relating to WebRTC being done in other working groups, which are listed below.

AVTCORE [AVTCOREWG] - The Audio Video Transport Core Working Group (AVTCORE) standardizes extensions to the Real-time Transport Protocol (RTP), which is used by WebRTC. This is the group responsible for defining how different types of media can be synchronized and sent together.

MMUSIC [MMUSICWG] - The Multiparty Multimedia Session Control Working Group (MMUSIC) standardizes extensions of the Session Description Protocol (SDP), which is used by WebRTC.

SIPCORE [SIPCOREWG] - The Session Initiation Protocol Core Working Group is involved in the ongoing maintenance and extensions to the core SIP protocol. One chartered item is a WebSocket transport for SIP, which would allow SIP to be used with WebRTC systems. SIP can also be used as a signaling protocol between web servers in the WebRTC trapezoid shown in Figure 1.5.

RMCAT [RMCATWG] – The RTP Media Congestion Avoidance Techniques Working Group is developing congestion control for RTP media over UDP transport. Requirements, feedback information in RTP or RTCP packets, and congestion control algorithms are being developed. These approaches will be used by WebRTC to ensure that widespread use of WebRTC will not cause excessive congestion on the Internet, and so that WebRTC media sessions can avoid congestion for the best user

experience.

B.4 References

[IETF] http://www.ietf.org

[RTCWEBWG] http://tools.ietf.org/wg/rtcweb

[AVTCOREWG] http://tools.ietf.org/wg/avtcore

[MMUSICWG] http://tools.ietf.org/wg/mmusic

[SIPCOREWG] http://tools.ietf.org/wg/sipcore

[RMCATWG] http://tools.ietf.org/wg/rmcat

APPENDIX C – GLOSSARY

ABNF – Augmented Backus-Naur Format. This is the meta-language used to define the syntax of text-based Internet protocols such as SDP and URLs. Originally defined in RFC 822, the most recent specification is RFC 5234.

API – Application Programming Interface. APIs are interfaces used by software components to communicate with each other.

HTML5 – The latest version of Hyper-Text Markup Language, the markup language used on the World Wide Web for web pages and applications. HTML originally defined simple markup tags in XML. Today, HTML5 supports Cascading Style Sheets (CSS) and scripting such as JavaScript. WebRTC is the part of HTML5 that deals with real-time voice, video, and data streams in browsers.

JavaScript – An interpreted scripting programming language used on web pages. Despite the name, it is quite different from Java. Today, most advanced web pages and applications use JavaScript. Technically, JavaScript is an implementation of the ECMA-262 (ECMAScript) standard. In practice, the terms JavaScript and ECMAScript are used interchangeably.

Jingle – A multimedia signaling protocol, which is an extension of XMPP (Extensible Messaging and Presence Protocol, RFC 6120, also known as Jabber). Jingle is defined by XEP-0166. Jingle uses RTP/SRTP for media, ICE NAT traversal, and supports mapping of media information to SDP.

NAT - Network Address Translation. NAT is a function often built into Internet routers or hubs that map one IP address space to another space. Usually, NATs are used to allow a number of devices to share an IP address, such as in a residential router or hub. NATs are also used by enterprises or service providers to segment IP networks, simplifying control and administration. Many Internet protocols, especially those using TCP transport or a client/server

architecture, have no difficulty traversing NATs. However, peer-to-peer protocols and protocols using UDP transport can have major difficulties. NAT traversal in WebRTC uses the ICE protocol. For details of how NAT and hole punching works, see Chapter 10 of SIP: Understanding the Session Initiation Protocol, 3rd Edition. NAT is also sometimes used to refer to Network Address Translator.

Offer/Answer – Media negotiation is the way in which two parties in a communication session, such as two browsers, communicate and come to agreement on an acceptable media session. Offer/answer is an approach to media negotiation in which one party first sends to the other party what media types and capabilities it supports and would like to establish – this is known as the "offer". The other party then responds indicating which of the offered media types and capabilities are supported and acceptable for this session – this is known as the "answer". This process can be repeated a number of times to setup and modify a session. While the term offer/answer is general, when used in WebRTC, it usually refers to RFC 3264 which defined the Offer/Answer Protocol, a usage of SDP by SIP. Offer/Answer must be studied in order to understand how exchanging SDP session descriptions can be used to negotiate a media session with WebRTC. For examples of SDP offer/answer, see Chapter 13 of SIP: Understanding the Session Initiation Protocol, 3rd Edition.

Node.js – The WebRTC APIs operate on the client side, in the web browser. They are JavaScript APIs that make use of the event loop built into the browser for handling the asynchronous input and output that is the hallmark of a Web GUI. WebRTC has quite a number of methods that take time to execute and are thus structured to execute application developer-provided callbacks when complete. Node.js, often referred to as just "Node", is a JavaScript interpreter and an event loop rolled together, along with modules for networking and file access and a convenient module/package management system. Traditional web servers spawn a new process for each request that is handled by PHP, Perl, or other languages. For isolated pages this is fine, but for services where the requests are

related (or even connected) and still asynchronous, coordination among the different running request handlers can be tricky, requiring a shared database, a shared file system, or explicit interprocess communication. Node's http server, on the other hand, uses a single process thread to handle all incoming requests. Each request is queued and handled via the event loop. Thus, programming a web server in Node is similar to programming a dynamic web page in a browser – every code snippet is initiated by etiher an event or a callback, and all code must ensure that it does not block. In return for this programming style requirement, all requests are handled within the same memory and process space, allowing for much easier control of and synchronization across multiple requests. When used with web pages containing WebRTC APIs, Node is convenient for handling both page requests and XHR pushes/requests, able to maintain state about the various browsers using it as the signaling gateway between the browsers.

Peer Connection – This term is used to refer to a direct connection set up between two "peers", two web browsers in the context of WebRTC, for the purpose of transporting audio, video, and data. Such a connection is established using the *RTCPeerConnection* and related APIs.

SIP – Session Initiation Protocol. SIP is an application level signaling protocol used for Internet Communications, Voice over IP, and video. SIP is defined by RFC 3261 and uses SDP session descriptions as defined by the offer/answer protocol.

WebSocket – The WebSocket protocol establishes a long-lived bi-directional TCP connection between a web browser and a web server, opened by the browser using HTTP.

APPENDIX D – SUPPLEMENTARY READING AND SOURCES

For online training and certification relating to WebRTC, we recommend WebRTC School:

http://www.webrtcschool.com

For background on HTML5 and JavaScipt, we recommend the easy-to-follow tutorials at:

http://www.w3schools.com

For background on an Internet communication signaling protocol such as Session Initiation Protocol, we recommend:

Johnston, Alan B, SIP: Understanding the Session Initiation Protocol, Artech House, Boston, 2009, 283 pages, 3rd Edition. ISBN-13:978-1607839958

This book also discusses NAT traversal and hole punching, SDP session descriptions, and SDP offer/answer.

Sinnreich, Henry and Alan B. Johnston, Internet Communications using SIP: Delivering VoIP and Multimedia Services with Session Initiation Protocol (Networking Council Series), John Wiley and Sons, New York, 2005, 298 pages, 2nd Edition. ISBN-13:978-0471776574

For background on RTP and media transport, we recommend:

Perkins, Colin, RTP: Audio and Video for the Internet, Addison-Wesley Professional, New York, 2003, 432 pages. ISBN-13:978-0672322495

For Internet Communications security for VoIP and video, we recommend:

Johnston, Alan B. and D. Piscitello, Understanding Voice over IP Security , Artech House, Boston, 2006, 276 pages, ISBN-13: 978-1596930506

For an entertaining fictional account of cybercrime and hacking that also happens to teach the basics of computer and Internet security, we recommend:

Johnston, Alan B, Counting from Zero, 2011, 281 pages, paperback ISBN-13:978-1461064886 or Kindle eBook

ABOUT THE AUTHORS

Dr. Alan B. Johnston has over thirteen years of experience in SIP, VoIP (Voice over IP), and Internet Communications, having been a co-author of the SIP specification and a dozen other IETF RFCs, including the ZRTP media security protocol. He is the author of four best selling technical books on Internet Communications, SIP, and security, and a techno thriller novel "Counting from Zero" that teaches the basics of Internet and computer security. He is on the board of directors of the SIP Forum. He holds Bachelors and Ph.D. degrees in electrical engineering. Alan is an active participant in the IETF RTCWEB working group. He is currently a Distinguished Engineer at Avaya, Inc. and an Adjunct Instructor at Washington University in St Louis. He owns and rides a number of motorcycles, and enjoys mentoring a robotics team.

Dr. Daniel C. Burnett has more than a dozen years of experience in computer standards work, having been author and editor of the W3C standards underlying the majority of today's automated Interactive Voice Response (IVR) systems. He has twice received the prestigious "Speech Luminary" award from Speech Tech Magazine for his contributions to standards in the Automated Speech Recognition (Voice Recognition) field. As an editor of the PeerConnection and getUserMedia W3C WEBRTC specifications and a participant in the IETF, Dan has been involved from the beginning in this exciting new field. He is currently the Chief Scientist at Voxeo Labs and Director of Standards at Voxeo. When he can get away, Dan loves camping both with his family and with his son's Boy Scout Troop.

Follow Alan and Dan on Twitter as @alanbjohnston and @danielcburnett and on Google+ as alanbjohnston@gmail.com danielcburnett@gmail.com.

For information on future editions along with updates and changes since publication, visit http://webrtcbook.com

Facebook http://www.facebook.com/webrtcbook
Google+ http://plus.google.com/102459027898040609362

Also by Alan B. Johnston:
Counting from Zero

Can a security expert save the Internet from a catastrophic zero day cyber attack by a network of zombie computers, known as a botnet? At what cost? Unfolding across three continents, this novel gives a realistic insider's view of the thrust and parry world of computer security and cryptography, and the very real threat of botnets.

"Credible and believable, this story is told by a subject matter expert. I could not wait to find out what happened next."
- Vint Cerf, Internet pioneer

"The threat to the Internet from worms, viruses, botnets, and zombie computers is real, and growing. **Counting from Zero** is a great way to come up to speed on the alarming state of affairs, and Johnston draws you in with his story and believable cast of characters."
- Phil Zimmermann, creator of Pretty Good Privacy (PGP) the most widely used email encryption program

"**Counting from Zero** brings Dashiell Hammet and Raymond Chandler into the computer age."
- Diana Lutz