# A More Technical Introduction to Web-based Real-Time Communication

This chapter introduces you to the technical concepts behind the new **Web-based Real-Time Communication** (**WebRTC**) standards. After reading this chapter, you will have a clear understanding of the following topics:

* How to set up peer-to-peer communication
* The signaling options available
* How the key APIs relate to each other

## Setting up communication

Although the basis of WebRTC communication is peer-to-peer, the initial step of setting up this communication requires some sort of coordination. This is most commonly provided by a web server and/or a signaling server. This enables two or more WebRTC capable devices or peers to find each other, exchange contact details, negotiate a session that defines how they will communicate, and then finally establish the direct peer-to-peer streams of media that flows between them.

### The general flow

There are a wide range of scenarios, ranging from single web page demos running on a single device to complex distributed multi-party conferencing with a combination of media relays and archiving services. To get started, we will focus on the most common flow, which covers two web browsers using WebRTC to set up a simple video call between them.

*A More Technical Introduction to Web-based Real-time Communication*

Following is the summary of this flow:

* Connect users
* Start signals
* Find candidates
* Negotiate media sessions
* Start RTCPeerConnection streams

#### Connect users

The very first step in this process is for the two users to connect in some way. The simplest option is that both the users visit the same website. This page can then identify each browser and connect both of them to a shared signaling server, using something like the WebSocket API. This type of web page, often, assigns a unique token that can be used to link the communication between these two browsers.

You can think of this token as a room or conversation ID. In the [http://apprtc.](http://apprtc/) appspot.com demo described previously, the first user visits [http://apprtc.](http://apprtc/) appspot.com, and is then provided with a unique URL that includes a new unique token. This first user then sends this unique URL to the second user, and once they both have this page open at the same time the first step is complete.

#### Start signals

Now that both users have a shared token, they can now exchange signaling messages to negotiate the setup of their WebRTC connection. In this context, "signaling messages" are simply any form of communication that helps these two browsers establish and control their WebRTC communication. The WebRTC standards don't define exactly how this has to be completed. This is a benefit, because it leaves

this part of the process open for innovation and evolution. It is also a challenge as this uncertainty often confuses developers who are new to RTC communication in general. The apprtc demo described previously uses a combination of XHR and the Google AppEngine Channel API (https://developers.google.com/appengine/ docs/python/channel/overview). This could, just as easily, be any other approach such as XHR polling, Server-Sent Events (<http://www.html5rocks.com/en/> tutorials/eventsource/basics/), WebSockets (http://www.html5rocks. com/en/tutorials/websockets/basics/), or any combination of these, you feel comfortable with.

*Chapter 2*

#### Find candidates

The next step is for the two browsers to exchange information about their networks, and how they can be contacted. This process is commonly described as "finding candidates", and at the end each browser should be mapped to a directly accessible network interface and port. Each browser is likely to be sitting behind a router that may be using Network Address Translation (NAT) to connect the local network to the internet. Their routers may also impose firewall restrictions that block certain ports and incoming connections. Finding a way to connect through these types

of routers is commonly known as NAT Traversal (<http://en.wikipedia.org/> wiki/NAT\_traversal), and is critical for establishing a WebRTC communication. A common way to achieve this is to use a Session Traversal Utilities for NAT (STUN) server (<http://en.wikipedia.org/wiki/Session_Traversal_Utilities_for_> NAT), which simply helps to identify how you can be contacted from the public internet and then returns this information in a useful form. There are a range of people that provide public STUN servers. The apprtc demo previously described uses one provided by Google.

If the STUN server cannot find a way to connect to your browser from the public internet, you are left with no other option than to fall back to using a solution that relays your media, such as a Traversal Using Relay NAT (TURN) server (http:// en.wikipedia.org/wiki/Traversal\_Using\_Relay\_NAT). This effectively takes you back to a non peer-to-peer architecture, but in some cases, where you are inside a particularly strict private network, this may be your only option.

Within WebRTC, this whole process is usually bound into a single Interactive Connectivity Establishment (ICE) framework (<http://en.wikipedia.org/wiki/> Interactive\_Connectivity\_Establishment) that handles connecting to a STUN server and then falling back to a TURN server where required.

#### Negotiate media sessions

Now that both the browsers know how to talk to each other, they must also agree on the type and format of media (for example, audio and video) they will exchange including codec, resolution, bitrate, and so on. This is usually negotiated using

an offer/answer based model, built upon the Session Description Protocol (**SDP**) ([http://en.wikipedia.org/wiki/Session\_Description\_Protocol).](http://en.wikipedia.org/wiki/Session_Description_Protocol)) This has been defined as the JavaScript Session Establishment Protocol (**JSEP**); for more information visit <http://tools.ietf.org/html/draft-ietf-rtcweb-jsep-00)> by the IETF.

*A More Technical Introduction to Web-based Real-time Communication*

#### Start RTCPeerConnection streams

Once this has all been completed, the browsers can finally start streaming media to each other, either directly through their peer-to-peer connections or via any media relay gateway they have fallen back to using.

At this stage, the browsers can continue to use the same signaling server solution for sharing communication to control this WebRTC communication. They can also use a specific type of WebRTC data channel to do this directly with each other.

### Using WebSockets

The WebSocket API makes it easy for web developers to utilize bidirectional communication within their web applications. You simply create a new connection using the var connection = new WebSocket(url); constructor, and then create your own functions to handle when messages and errors are received. And sending a message is as simple as using the connection.send(message); method.

The key benefit here is that the messaging is truly bidirectional, fast, and lightweight. This means the WebSocket API server can send messages directly to your browser whenever it wants, and you receive them as soon as they happen. There are no delays or constant network traffic as it is in the XHR polling or long-polling model, which makes this ideal for the sort of offer/answer signaling dance that's required to set up WebRTC communication.

The WebSocket API server can then use the unique room or conversation token, previously described, to work out which of the WebSocket API clients messages should be relayed to. In this manner, a single WebSocket API server can support a very large number of clients. And since the network connection setup happens very rarely, and the messages themselves tend to be small, the server resources required are very modest.

There are WebSocket API libraries available in almost all major programming languages, and since Node.js is based on JavaScript, it has become a popular choice for this type of implementation. Libraries such as socket.io (<http://socket.io/)> provide a great example of just how easy this approach can really be.

### Other signaling options

Any approach that allows browsers to send and receive messages via a server can be used for WebRTC signaling.

*Chapter 2*

The simplest model is to use the XHR API to send messages and to poll the server periodically to collect any new messages. This can be easily implemented by any web developer without any additional tools. However, it has a number of drawbacks. It has a built-in delay based on the frequency of each polling cycle. It is also a waste of bandwidth, as the polling cycle is repeated even when no messages are ready to be sent or received. But if you're looking for a good old-fashioned solution, then this is the one.

A slightly more refined approach based on polling is called long-polling. In this model, if the server doesn't have any new messages yet, the network connection is kept alive until it does, using the HTTP 1.1 keep-alive mechanisms. When the server has some new information, it just sends it down the wire to complete the request. In this case, the network overhead of the polling is reduced. But it is still an outdated and inefficient approach compared to more modern solutions such as WebSockets.

Server-Sent Events are another option. You establish a connection to the server using the var source = new EventSource(url); constructor, and then add listeners to that source object to handle messages sent by the server. This allows servers to send you messages directly, and you receive them as soon as they happen. But you are still left using a separate channel, such as XHR, to send your messages to the server, which means you are forced to manage and synchronize two separate channels.

This combination does provide a useful solution that has been used in a number of WebRTC demonstration apps, but it does not have the same elegance as a truly bidirectional channel, such as WebSockets.

There are all kinds of other creative ideas that could be used to facilitate the required signaling as well. But what we have covered are the most common options you will find being used.

## MediaStream API

The **MediaStream** API is designed to allow you to access streams of media from local input devices, such as cameras and microphones. It was initially focused upon the **getUserMedia** API or **gUM** for short, but has now been formalized

as the broader media capture and streams API, or MediaStream API for short. However, the getUserMedia() method is still the primary way to initiate access to local input devices.

Each MediaStream object can contain a number of different MediaStreamTrack objects that each represents different input media, such as video or audio from different input sources.

*A More Technical Introduction to Web-based Real-time Communication*

Each MediaStreamTrack can then contain multiple channels (for example, the left and right audio channels). These channels are the smallest units that are defined by the MediaStream API.

MediaStream objects can then be output in two key ways. First, they can be used to render output into a **MediaElement** such as a <video> or <audio> element

(although the latter may require pre-processing). Secondly, they can be used to send to an **RTCPeerConnection**, which can then send this media stream to a remote peer.

Each MediaStreamTrack can be represented in a number of states described by the MediaSourceStates object returned by the states() method. Each

MediaStreamTrack can also provide a range of capabilities, which can be accessed through the capabilities() method.

At the top level, a MediaStream object can fire a range of events such as addtrack, removetrack, or ended. And below that a MediaStreamTrack can fire a range of events such as started, mute, unmute, overconstrainted, and ended.

## RTCPeerConnection API

The RTCPeerConnection API is the heart of the peer-to-peer connection between each of the WebRTC enabled browsers or peers. To create an RTCPeerConnection object, you use the var peerconnection = RTCPeerConnection(configuration); constructor. The configuration variable contains at least one key named iceServers, which is an array of URL objects that contain information about STUN, and possibly TURN servers, used during the finding candidates phase.

The peerconnection object is then used in slightly different ways on each client, depending upon whether you are the caller or the callee.

### The caller's flow

Here's a summary of the caller's flow after the peerconnection object is created:

* Register the onicecandidate handler
* Register the onaddstream handler
* Register the message handler
* Use getUserMedia to access the local camera
* The JSEP offer/answer process

*Chapter 2*

#### Register the onicecandidate handler

First, you register an onicecandidate handler that sends any ICE candidates to the other peer, as they are received using one of the signaling channels described previously.

#### Register the onaddstream handler

Then, you register an onaddstream handler that displays the video stream once it is received from the remote peer.

#### Register the message handler

Your signaling channel should also have a handler registered that responds to messages received from the other peer. If the message contains an RTCIceCandidate object, it should add those to the peerconnection object using the addIceCandidate() method. And if the message contains an RTCSessionDescription object, it should add those to the peerconnection object using the setRemoteDescription() method.

#### Use getUserMedia to access the local camera

Then, you can utilize getUserMedia() to set up your local media stream and display that on your local page, and also add it to the peerconnection object using the addStream() method.

#### The JSEP offer/answer process

Now, you are ready to start the negotiation using the createOffer() method and registering a callback handler that receives an RTCSessionDescription object. This callback handler should then add this RTCSessionDescription to your peerconnection object using setLocalDescription(). And then finally, it should also send this RTCSessionDescription to the remote peer through your signaling channel.

### The callee's flow

The following is a summary of the callee's flow, which is very similar in a lot of ways to the caller's flow, except that it responds to the offer with an answer:

* Register the onicecandidate handler
* Register the onaddstream handler

*A More Technical Introduction to Web-based Real-time Communication*

* Register the message handler
* Use getUserMedia to access the local camera
* The JSEP offer/answer process

#### Register the onicecandidate handler

Just like the caller, you start by registering an onicecandidate handler that sends any ICE candidates to the other peer as they are received, using one of the signaling channels described previously.

#### Register the onaddstream handler

Then, like the caller, you register an onaddstream handler that displays the video stream once it is received from the remote peer.

#### Register the message handler

Like the caller, your signaling channel should also have a handler registered that responds to messages received from the other peer. If the message contains an RTCIceCandidate object, it should add those to the peerconnection

object using the addIceCandidate() method. And if the message contains an RTCSessionDescription object, it should add those to the peerconnection object using the setRemoteDescription() method.

#### Use getUserMedia to access the local camera

Then, like the caller, you can utilize getUserMedia() to set up your local media stream and display that on your local page, and also add it to the peerconnection object using the addStream() method.

#### The JSEP offer/answer process

Here you differ from the caller and you play your part in the negotiation by passing remoteDescription to the createAnswer() method and registering a callback handler that receives an RTCSessionDescription object. This callback handler should then add this RTCSessionDescription to your peerconnection object using setLocalDescription(). And then finally, it should also send this RTCSessionDescription to the remote peer through your signaling channel. It is also important to note that this callee flow is all initiated after the offer is received from the caller.

*Chapter 2*

### Where does RTCPeerConnection sit?

The following diagram shows the overall WebRTC architecture from the [www.WebRTC.org](http://www.WebRTC.org/) site. It shows you the level of complexity that is hidden below the RTCPeerConnection API.

Your web app #3

Your web app #2

Your web app #1

Overall architecture diagram from [www.WebRTC.org](http://www.WebRTC.org/)

...

The web

Web API (Edited by W3C WG)

**WebRTC**

WebRTC C C++ API (PeerConnection)

Session management / Abstract signaling (Session)

**Voice Engine**

iSAC / iLBC Codec

**Video Engine**

VP8 Codec

**Transport**

SRTP

Your browser

NetEQ for voice

Video jitter buffer

Multiplexing

Echo Canceler / Noise Reduction

Image enhancements

P2P

STUN +TURN+ ICE

**Audio Capture/Render**

**Video Capture**

**Network I/O**

API for web developers

API for browser makers

Overrideable by browser makers

## RTCDataChannel API

As well as sending media streams between peers using WebRTC, it is also possible to use the DataChannel API to send arbitrary streams of data. Although many people commonly refer to this as the RTCDataChannel API, it is more accurately defined as just the WebRTC DataChannel API and is created by using the var datachannel = peerconnection.createDataChannel(label); constructor. It is a very flexible and powerful solution that has been specifically designed to be similar to the WebSocket API through the send() method and the onmessage event.

At the time of writing this chapter, this API is still in a state of flux with the varying

browser implementations still struggling with standardization.

*A More Technical Introduction to Web-based Real-time Communication*

## Summary

You should now have a clear overview of the various APIs and protocols that combine to make WebRTC work.

Throughout the rest of the book, we will explore the MediaStream, RTCPeerConnection, and RTCDataChannel APIs in more detail as we work to apply these concepts to real world examples.

First, we will start by fleshing out the simple peer-to-peer video call scenario into

a fully working application.

Then, we will explore how this can be simplified down to just an audio only call or extended with text-based chat and file sharing.

And then, we will explore two real-world application scenarios based upon e-learning and team communication.

# Creating a Real-time

Video Call

This chapter shows you how to use the MediaStream and RTCPeerConnection APIs to create a working peer-to-peer video chat application between two people. After reading this chapter, you will have a clear understanding of:

* Using a web server to connect two users
* Setting up a signaling server for a peer-to-peer call
* How the caller's browser creates an offer
* How the callee's browser responds with an answer
* Previewing local video streams
* Establishing and presenting remote video streams
* The types of stream processing available
* Extending this into a Chatroulette application

## Setting up a simple WebRTC video call

The most common WebRTC example application involves setting up a video call between two separate users. Within a few seconds, you can easily see and talk to anyone, anywhere in the world who has one of the one billion or more WebRTC- enabled browsers. Let's take a detailed look at how this can be achieved and create the code we need as we go.

Throughout this book, some simple coding conventions will be used to aid communication and readability.

*Creating a Real-time Video Call*

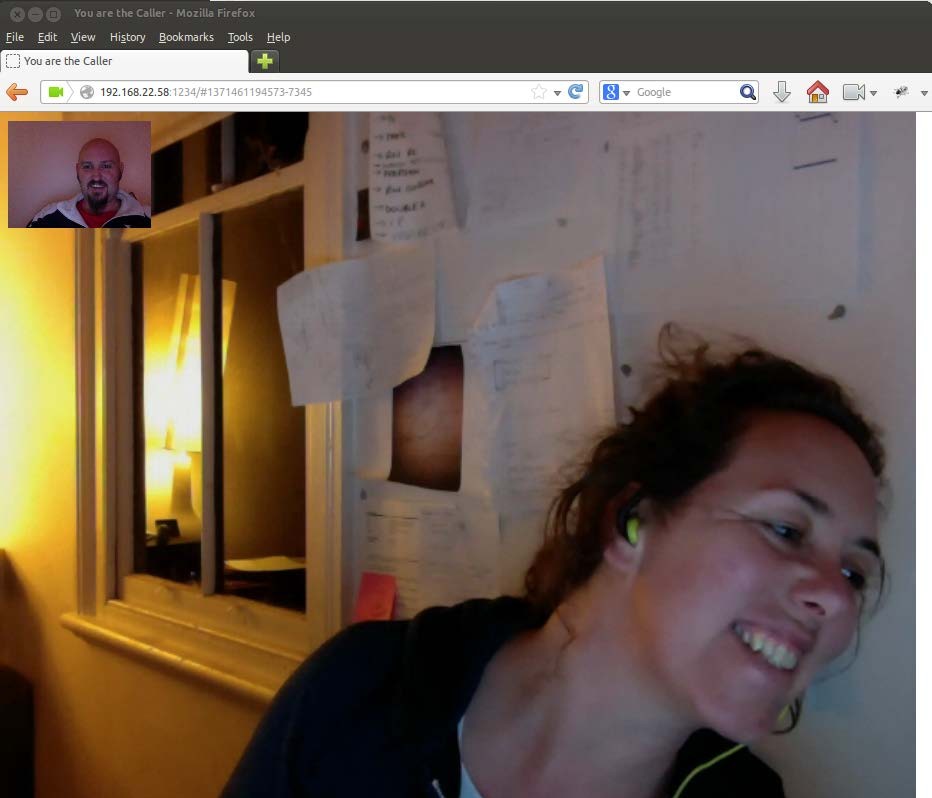
JavaScript APIs standardized by the W3C and other standards definition organizations

will use the conventional camel case format (for example, standardFunctionCall()).

Functions and variables that have been defined for this book will use all lowercase strings and replace word breaks or white space with an underscore (for example, custom\_function\_call()).

The web and WebSocket server functionality in this example application will be implemented using JavaScript and Node.js. It is beyond the scope of this book to provide information on how to install and configure Node.js, but all the information you need can be found at [http://nodejs.org/.](http://nodejs.org/)

However, this book does provide you with well-described working Node.js example code that provides all the functionality you need to run the demonstration applications.



A basic peer-to-peer video call using WebRTC

*Chapter 3*

## Using a web server to connect two users

The very first step is simply to connect two separate users using the Web. We start by creating a standard HTML5 web page that includes a DOCTYPE definition, a document head, and a document body:

<!DOCTYPE html>

<html>

<head>

…

</head>

<body>

…

</body>

</html>

**Downloading the example code**

You can download the example code files for all Packt books you have purchased from your account at [http://www.packtpub.com.](http://www.packtpub.com/) If you purchased this book elsewhere, you can visit http://www.packtpub. com/support and register to have the files e-mailed directly to you.



Then, the first element inside the document head is the webrtc\_polyfill.js script included inline between a pair of <script> tags. The webrtc\_polyfill.js code is exactly what it says it is and is designed to make it easy to write JavaScript that works across all common browser implementations of the WebRTC and MediaStream APIs. Here is an overview of how it works.

First, we set up six global placeholders for the primary features it exposes:

var webrtc\_capable = true;

var rtc\_peer\_connection = null;

var rtc\_session\_description = null; var get\_user\_media = null;

var connect\_stream\_to\_src = null;

var stun\_server = "stun.l.google.com:19302";

These global placeholders are then populated with their final values, based on the

type of browser capabilities that are detected.

rtc\_peer\_connection is a pointer to either the standard RTCPeerConnection, mozRTCPeerConnection if you are using an early Firefox WebRTC implementation, or webkitRTCPeerConnection if you are using an early WebRTC implementation in a WebKit-based browser like Chrome.

*Creating a Real-time Video Call*

rtc\_session\_description is also a pointer to the browser-specific implementation of the RTCSessionDescription constructor. For this, the only real exception is within the early Firefox WebRTC implementation.

get\_user\_media is very similar. It is either a pointer to the standard navigator. getUserMedia, navigator.mozGetUserMedia if you are using an early MediaStream API implementation in Firefox, or navigator.webkitUserMedia if you are using an early MediaStream API implementation in a WebKit-based browser such as Chrome.

connect\_stream\_to\_src is a function that accepts a reference to a MediaStream object and a reference to an HTML5 <video> media element. It then connects the stream to the <video> element so that it can be displayed within the local browser.

Finally, the stun\_server variable holds a pointer to Google's public STUN server. Currently, Firefox requires this to be an IP address, but Chrome supports DNS-based hostnames and ports.

The heart of the browser detection is then handled in a set of simple if/else blocks.

First, it checks if the standard navigator.getUserMedia is supported, else it checks if navigator.mozGetUserMedia is supported (for example, early Firefox MediaStream API), or else if navigator.webkitGetUserMedia is supported (for example, an early WebKit browser MediaStream API).

The final else block then assumes that this is a browser that doesn't support getUserMedia at all. This code also assumes that if getUserMedia is supported in some way, then a matching RTCPeerConnection API is also implicitly supported.

The connect\_stream\_to\_src function then is adapted slightly, based on which type of browser has been detected.

The default standard version directly assigns the media\_stream to the video element's .srcObject property:

connect\_stream\_to\_src = function(media\_stream, media\_element) { media\_element.srcObject = media\_stream;

media\_element.play();

};

Within the early Firefox WebRTC implementations, the <video> media element uses the mozSrcObject property, which can have the media stream object directly assigned to it:

connect\_stream\_to\_src = function(media\_stream, media\_element) { media\_element.mozSrcObject = media\_stream; media\_element.play();

};

*Chapter 3*

Within the early WebKit-based WebRTC implementations, the webkitURL. createObjectURL function is passed the media stream object, and the response from this is then directly assigned to the <video> element's .src property:

connect\_stream\_to\_src = function(media\_stream, media\_element) { media\_element.src = webkitURL.createObjectURL(media\_stream);

};

Once webrtc\_polyfill.js has set up everything, we need to create browser independent WebRTC code; we can then move onto the body of this video call application. The code that defines the basic\_video\_call.js browser side logic for this is included inline within another pair of <script></script> tags.

First, we set up the general variables that we will use throughout the rest of the code.

The call\_token variable is a unique ID that links two users together. It is used to ensure that any signals passing through the signaling server are only exchanged between these two specific users.

var call\_token; // unique token for this call

The signaling\_server is a variable that represents the WebSocket API connection to the signaling server to which both the caller and callee will be connected:

var signaling\_server; // signaling server for this call

The peer\_connection variable represents the actual RTCPeerConnection that will be established between these two users:

var peer\_connection; // peerconnection object

Next, we set up a basic start() function that is called by the pages'

body.onload event:

function start() {

This function essentially detects if you are the caller or the callee, and then sets up the relevant functionality to match. It also sets up a number of common functions that are used by both the caller and the callee.

*Creating a Real-time Video Call*

The first step here is to populate the peer\_connection variable with a real RTCPeerConnection object using the rtc\_peer\_connection constructor setup by webrtc\_polyfill.js. We pass a configuration object to this function that defines the STUN server we would like to use. In this example, we have used a public STUN server provided by Google; however, this is for demonstration

purposes only. If you intend to build a commercial application, you must find a

commercial STUN provider.

// create the WebRTC peer connection object peer\_connection = new rtc\_peer\_connection({

"iceServers": [

{ "url": "stun:"+stun\_server }, // stun server info

]

});

Next, we set up our own function for the peer\_connection.onicecandidate event handler and if ice\_event contains a candidate then we serialize this into a JSON blob and send that to the other caller's browser through the signaling\_server variable:

// generic handler that sends any ice candidates to the other peer peer\_connection.onicecandidate = function (ice\_event) {

…

};

Then, we set up our own function for the peer\_connection.onaddstream handler. This simply receives any new incoming video streams and connects them to a local

<video> element within the local browser, so you can see and hear the person on the other end of the call.

// display remote video streams when they arrive peer\_connection.onaddstream = function (event) {

…

};

Later, we set up our connection to the signaling server using the WebSocket API. This is generic, because this same type of connection is used by both the caller and the callee. It is essential that both are connected to the same signaling server in this basic example.

// setup generic connection to the signaling server using the WebSocket API

signaling\_server = new WebSocket("ws://localhost:1234");

*Chapter 3*

Now, all of the generic functionality has been set up, and we can move onto customizing the rest based on whether you are a caller or a callee. This is simply done by detecting whether the browser has loaded a page with a call\_token hash fragment or not.

If you are the caller, then you are the first person to visit the page, and you

will have no call\_token at all. In this case, we will create one for you and set location.hash so that you can see this in your browser's location bar (for example, http://localhost:1234#1370439693969-433). It is important to note that localhost should be replaced with the hostname or IP address that you actually intend to use, and that this must also be accessible to the other person that intends to join the call.

You can then send this URL to the other person (via e-mail, SMS, carrier pigeon, or whatever method works best for you). Once they load this URL, we will then detect that they already have a call\_token hash fragment defined and will then treat them as the callee.

if (document.location.hash === ""

|| document.location.hash === undefined) { // you are the Caller

…

} else { // you have a hash fragment so you must be the Callee

...

}

}

Following the start() function, we define the detailed implementation of a number of other generic handler functions that are used by either or both the caller and the callee.

First, we implement the function that handles any new descriptions that are set up in the JSEP offer/answer process. This will be described in more detail in the following code snippet:

// handler to process new descriptions

function new\_description\_created(description) {

…

}

Then, we implement the function that handles all the signals we receive from the signaling server from the perspective of the caller. This handles four key scenarios:

1. If signal.type is callee\_arrived, then we start the JSEP offer/answer process. This is described in more detail in the code snippet that follows.
2. If signal.type is new\_ice\_candidate, then this candidate is added to

peer\_connection.

*Creating a Real-time Video Call*

1. If signal.type is new\_description, then we call peer\_connection. setRemoteDescription().
2. Or else you can extend this with your own custom signaling. Here is the function that handles these four scenarios:

// handle signals as a caller

function caller\_signal\_handler(event) { var signal = JSON.parse(event.data);

if (signal.type === "callee\_arrived") {

...

} else if (signal.type === "new\_ice\_candidate") {

...

} else if (signal.type === "new\_description") {

...

} else {

// extend with your own signal types here

}

}

Then, we implement the function that handles all the signals we receive from the signaling server from the perspective of the callee. This is similar to the caller function that we just saw, except it only handles 3 key scenarios:

1. If signal.type is new\_ice\_candidate, then this candidate is added to the peer\_connection.
2. If signal.type is new\_description, then we call peer\_connection. setRemoteDescription(), and if the description contains an offer, then we create an answer.
3. Or else you can extend this with your own custom signaling. Here is the function that handles these three scenarios:

// handle signals as a callee

function callee\_signal\_handler(event) { var signal = JSON.parse(event.data);

if (signal.type === "new\_ice\_candidate") {

...

} else if (signal.type === "new\_description") {

...

} else {

// extend with your own signal types here

}

}

*Chapter 3*

Now, we implement the function that requests access to the local camera's video stream using the getUserMedia call. If this stream is set up successfully, then it is displayed on the local browser page in a <video> element and then added to the peer\_connection object so that it can be sent to the remote peer's browser. If this local stream is not set up successfully, then the error is logged so that the user can be notified.

// setup stream from the local camera function setup\_video() {

get\_user\_media(

{

"audio": true, // request access to local microphone "video": true // request access to local camera

},

function (local\_stream) { // success callback

...

},

log\_error // error callback

);

}

Then, we define a generic error handler function that logs the error and notifies the

user, so they know what has happened:

// generic error handler function log\_error(error) {

…

}

Once all of the JavaScript code has been defined, then we move on to defining some simple CSS based styles. You can obviously customize this as much as you like, but in this example, we have provided the basic styles you need to understand on how you can quickly and easily create a user interface that handles all of the different states required for this video call application.

First, we create a style for the loading\_state, open\_call\_state, local\_video, and

remote\_video ID's, and set them to not be displayed by default.

Then, we set the style for loading\_state to make sure it is displayed by default. This content is shown to both the caller and the callee when they first load the page.

<style> html, body {

padding: 0px; margin: 0px;

font-family: "Arial","Helvetica",sans-serif;

*Creating a Real-time Video Call*

}

#loading\_state { position: absolute; top: 45%;

left: 0px; width: 100%; font-size: 20px;

text-align: center;

}

#open\_call\_state { display: none;

}

#local\_video { position: absolute; top: 10px;

left: 10px; width: 160px; height: 120px;

background: #333333;

}

#remote\_video { position: absolute; top: 0px;

left: 0px; width: 1024px; height: 768px;

background: #999999;

}

</style>

Now, we are ready to set up the HTML body content of the page that binds the CSS and JavaScript together into a working user interface. This example application contains only the minimum content needed to demonstrate a working video call application. From this, you should be able to add your own custom content to adapt this to your needs quickly and easily.

<body onload="start()">

<div id="loading\_state"> loading...

</div>

<div id="open\_call\_state">

<video id="remote\_video"></video>

<video id="local\_video"></video>

</div>

</body>

*Chapter 3*

You now have a web page-based user interface and application that can connect two browsers using WebRTC to set up a peer-to-peer video call. From here, you may like to extend this code to add extra error handling, make the WebSocket connection

automatically reconnect if it gets disconnected, and also customize the HTML and CSS.

Now that we have set up the browser side of the application, let's move on to see how the signaling server side of this application works.