A Case of WebRTC demo

Advisor: Professor Chris Pollett
San Jose State University

Written by Yangcha K. Ho

Term: Spring 2018

August 12, 2018

I. Introduction

WebRTC (Web Real-Time Communication) is a framework that enables peer to peer connections. It allows voice, video and data to work collectively in the browser without relying on third party plugins. It uses a collection of standards, protocols, and JavaScript APIs and HTML5. But the collection of APIs and protocols are defined by a variety of Working Groups who have different rates on working on different browsers and operating systems.

When two browsers want to set up a communication between them, they need a broker who helps to communicate between them. The role of a signaling server is this broker, but the WebRTC did not include any specification as to how this signaling server works. It is complexly left to developers who can choose which method fits for them.

Though the standards and protocols used for WebRTC implementations are highly stable, and there are a few differences in prefixed names. That's why it is suggested that we should use *adapter.js* (1) which is a shim to insulate apps from spec changes and prefix differences. Currently, Chrome, Firefox and Opera support. The Web site: http://caniuse.com can be referred for more about WebRTC compatible. (2)

Thought the concept is very simple, but its implementation requires some work. Dave Kilian who wrote "It may not immediately sound like it but setting up a connection with WebRTC is more challenging than you'd first think…" in his "Setting Up WebRTC The Hard Way". (3)

II. Several Components work together

This paper covers one scenario of WebRTC example, video call at which two web browsers can share a simple video call using WebRTC. This document tries to cover as much as detail for a novice developer might find it useful since there are not too many good examples for a WebRTC, though there are a lot of articles about what the WebRTC but lacks solid code examples. There are a lot of snippet of code, but not complete ones. In general, a WebRTC-enabled application needs to do several things take of:

1. Connect users
2. Start signals
3. Find candidates
4. Negotiate media sessions

5. Taking APIs and eventhandlers – will have a section for this.

III. How each component work together

1. Connect users:

One option is both users login in the same website. This page can connect both users to a shared signaling server, using something like the WebSocket API. As an example, In the http://apprtc. appspot.com demo (3), the first user visits http://apprtc. appspot.com, is given a unique URL and he/she forwards this unique URL to the second user. Now both of them have this page.

2. Start signals:

Now that both users are set up to exchange signaling messages in their WebRTC connection. Here, "signaling messages" are simply any form of communication that allows two browsers share WebRTC communication. In WebRTC, we must define how peers exchange information on how to connect to one another before the actual connection can be begin. However, the WebRTC standards don't define exactly how this has to be done, it confuses developers who are new to RTC communication. For this paper, we chose to use ws (WebSocket Server) with Node.js. A WebSocket server is a TCP application listening on any port of a server that follows a specific protocol.

3. Find candidates:

NAT router allows a single IP address to share multiple computers. Computers behind a NAT router can't receive inbound connections. WebRTC implements workarounds that help computers behind NAT routers talk to each other. A common way to achieve this is to use a (STUN) Session Traversal Utilities which simply helps to identify how you can be contacted from the public internet. If the STUN server cannot find a way to connect to your browser from the public internet, you are left to a Traversal Using Relay NAT (TURN) server. WebRTC places this whole process into a single Interactive Connectivity Establishment (ICE) framework that handles connecting to a STUN server and then falling back to a TURN server where required.

4. Negotiate media sessions;

Both the browsers can communicate with each other, they must also share 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 createOffer/createAnswer callback functions built upon the Session Description Protocol (SDP). We will cover more details on createOffer/createAnswer funtions in later this paper.

IV. WebRTC implements three APIs:

1. MediaStream - allows the client (e.g., the web browser) to access the stream, such as the one from a WebCam or microphone;
2. RTCPeerConnection - enable audio or video data transfer, with support for encryption and bandwidth management;
3. RTCDataChannel - enables peer-to-peer communication for any generic data.

1. MediaStream

With getUserMedia(), we can access to webcam and microphone input without a plugin. It's baked directly into the HTML5 browser. In the following snippet we specify video, audio elements.

navigator.getUserMedia accepts three parameters as shown here: a.video/audio contrains, b, successFunction, c. failedFunction

Example snippet of main.js code:

```
...
var constraints = {
      video: true,
      audio: true,
   };

if(navigator.getUserMedia) {
      navigator.getUserMedia(constraints, getUserMediaOK,
          getUserMediaFailed);
      } else {
      alert('Your browser does not support getUserMedia API');
      }

    function getUserMediaOK (stream) {
       window.stream = stream; // make stream available to console
```

```
        videoElement.srcObject = stream;
    }
    function getUserMediaFailed (error) {
    console.error('Error: ', error);
}
```

At html5 page, we can specify

```
<html>
 ...
<video id="localVideo" autoplay muted></video>
<video id="remoteVideo" autoplay></video>
<script src="main.js"></script>
...
```

2. RTCPeerConnection API

The RTCPeerConnection API is the core of WebRTC connection. We can create RTCPeerConnection objects by writing:

```
var pc1 = RTCPeerConnection(urlParameter);
```

where the *urlParameter* argument lists an array of URL objects of STUN and TURN servers, used during the finding of the ICE candidates. You can check at code.google.com for free public STUN servers.

Step a:

1. First, we create an object of RTCPeerConnection API and attach to event handlers as shown below:
    a. var pc1 = new RTCPeerConnection(urlParameter);
    b. pc1.onicecandidate = onicecandidate;
    c. pc1.onaddstream = onaddstream;
    d. pc1.addStream (localstream);
    e. pc1.createOffer = see below for detailed code
    f. document.getElementById("makeCall").addEventListener("click",

```
function(){
  pc1.createOffer().then(function(desc){
       pc1.setLocalDescription(new RTCSessionDescription(desc));
       socket.emit("sdp", JSON.stringify({"sdp": desc}));
});
});
```

Step b:

- you must create three callback functions of in the function onicecandidate; (onIceCandidate, onNegotiationNeeded and onAddStream) and assign them to the RTCPeerConnection.
- The onicecandidate callback that will be called whenever the browser finds a new ICE candidate. we send the candidate to our signaling server so it can sent to the other client. A NULL candidate denotes that the browser is finished gathering candidates.

  An ICE candidate is essentially a description of how to connect to a client. In order for anyone to connect to us, we need to share our ICE candidates with the other client. it will start gathering ICE candidates.

  onNegotiationNeeded: If something happens that requires a new session negotiation, this is triggered.

  ontrack(old name is onaddatream). This is specifuied where you handle the remote video stream from your remote buddy

Step c:
pc1.onaddstream returns remote stream of microphone and camera of other site.

Step d:
pc1.addstream attaches your local microphone and camera.

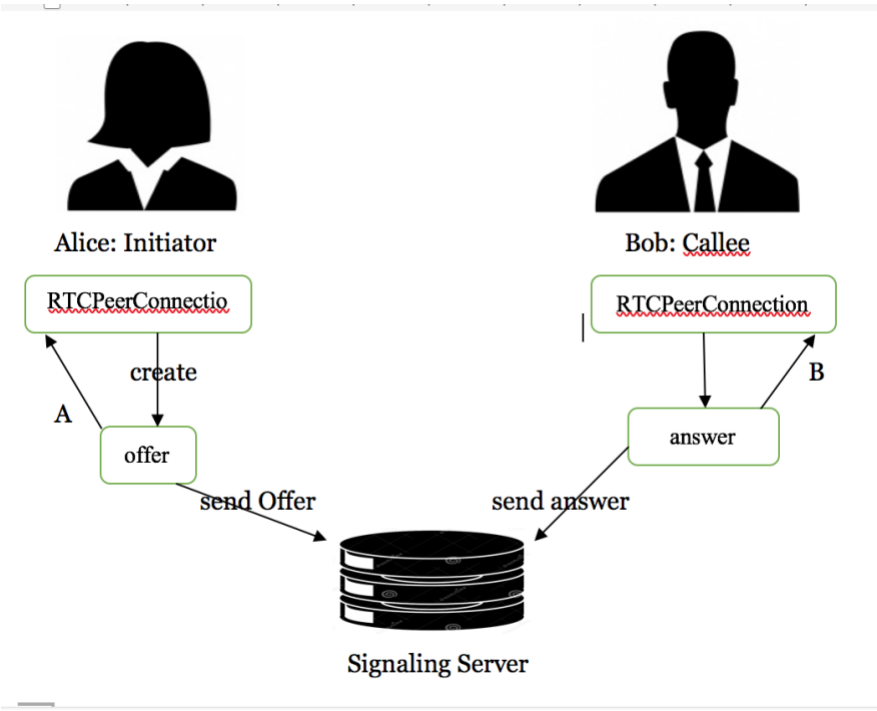
Step e:
If we're the pc1(caller) (we clicked the start button), we create an offer which tells the other client how to interact with us once the network connection is established. We call this protocol Session Description Protocol (SDP) which will be created when we create a setLocalDesription method.

Here, we set the local description to it and then send it to the signaling server to be sent to the other client.

There are following possible situations:

1. A person wants to share camera with his friend(s): so, he will make an offer to his friend to join him. He uses createOffer function.

2. A person who receives an offer from a friend: so, he will join him. He will use createAnswer callback function. The following diagram show how createOffice/createAnswer functions work.



A, B = returning RTCPeerConnection.setLocalDescription()

Creating Offer SDP

```
peerConnection.createOffer(function (sessionDescription) {
   peerConnection.setLocalDescription(sessionDescription);
   // POST-Offer-SDP-For-Other-Peer(sessionDescription.sdp,
       sessionDescription.type);
}, function(error) {
   alert(error);
}, { 'mandatory': { 'OfferToReceiveAudio': true, 'OfferToReceiveVideo': true } });
```

createAnswerer function

Assume that you sent offer sdp to your friend using server.

Now, "process" that offer sdp and then create answer sdp and send it back to offerer:

```
      peerConnection.setRemoteDescription(new RTCSessionDescription(offerSDP));

      And to createAnswer...
      pc1.createAnswer(function (sessionDescription) {
         pc1.setLocalDescription(sessionDescription);
          // POST-answer-SDP-back-to-Offerer(sessionDescription.sdp,
```

```
        sessionDescription.type);
  }, function(error) {
     alert(error);
  }, { 'mandatory': { 'OfferToReceiveAudio': true, 'OfferToReceiveVideo': true } });
```

V. Summarization of process:

We can summarize the above process in a plain English as follows:

Each peer keeps two descriptions, one for the local description, describing itself, and the remote description, describing the other end of the call.

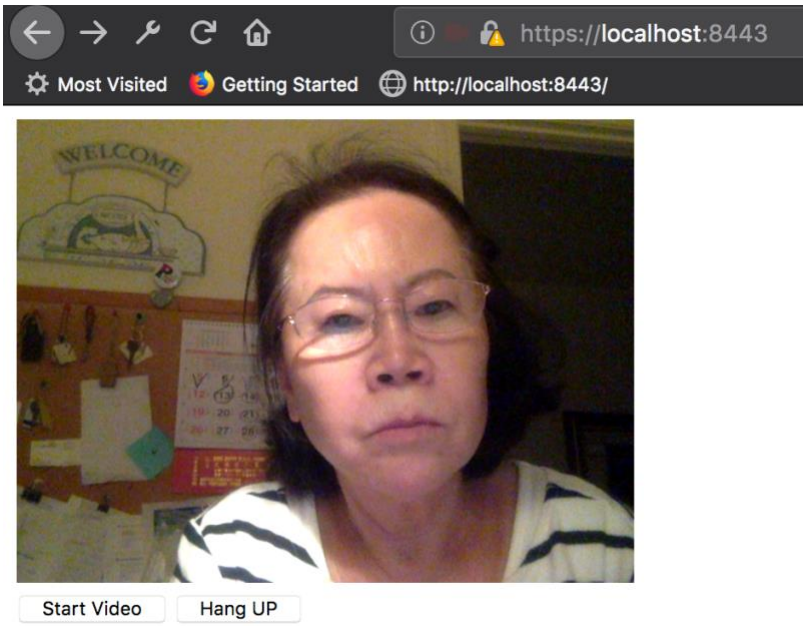
The offer/answer process is performed both when a call is first established, but also any time the call's format or other configuration needs to change.

1. The caller invokes RTCPeerConnection.createOffer() which sets
    a. RTCPeerConnection.setLocalDescription() to set that offer as the *local description*.
    b. Then caller uses the signaling server to transmit the offer to the intended receiver of the call.
2. The recipient receives the offer and calls RTCPeerConnection.setRemoteDescription() to record it as the *remote description*.
    a. The recipient invokes RTCPeerConnection.createAnswer() to respond to caller.

VI. DataChannel:

WebRTC's RTCDataChannel interface allows to transfer data directly from one peer to another. It lets you open a channel between two peers over which you may send and receive arbitrary data. The API is intentionally similar to the WebSocket API ()so that the same programming model can be used for each. Since this paper is dedicated to show a case of video call, I have not had a chance to work on this API. This is my future project.

VII. Example of snap shot



VIII. WebSocket Server: Bringing Sockets to the Web

A WebSocket server is a TCP application listening on any port of a server that follows a specific protocol. This paper uses node.js to implement a simple WebSocket server.

Running this application:

1. npm install
2. npm install ws
3. node server.js
4. At Firefox browser: https://localhost:8090
5. For a security reason this might not open at Chrome which does not allow when security check is caught.
6. This file needs to be hosted either on the Apache or Node servers.

IX. Conclusion:

WebRTC is a pretty new technology, yet a few browsers support this technology and its APIs are still in flux. There is a plethora of information as to how its architecture , but there lack real examples that can be readily run. This paper shows a real case to demonstrate how it can be done in a simplest way.

X. References:

1. https://appr.tc/
2. http://subnets.ru/books/Getting-Started-with-Webrtc-2013-Rob-Manson.pdf

3. http://davekilian.com/webrtc-the-hard-way.html
4. https://www.html5rocks.com/en/tutorials/webrtc/infrastructure/
5. https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API/Writing_WebSocket_servers
6. https://web-engineering.info/node/57
7. https://www.tutorialspoint.com/webrtc/webrtc_rtcpeerconnection_apis.htm
8. https://www.pkc.io/blog/untangling-the-webrtc-flow/
9. https://www.webcodegeeks.com/web-development/webrtc-tutorial-beginners/
10. https://swizec.com/blog/learning-webrtc-peer-peer-communication-part-1/swizec/8359
11. https://shanetully.com/2014/09/a-dead-simple-webrtc-example/
12. https://simpl.info/rtcpeerconnection/
13. https://www.webrtc-experiment.com/docs/how-to-use-rtcpeerconnection-js-v1.1.html