A WebRTC Video Chat Implementation Within the Yioop Search Engine

A Project Presented to

The Faculty of the Department of Computer Science San Jose State University

In Partial Fulfillment of

the Requirements for the Degree Master of Science

By

Yangcha K. Ho

May 2019

SAN JOSÉ STATE UNIVERSITY

The Undersigned Thesis Committee Approves the Thesis Titled

A WebRTC Video Chat Implementation Within the Yioop Search Engine

By Yangcha K. Ho

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

____

Dr. Chris Pollett, Department of Computer Science 05/20/2019

____

Dr. Melody Moh, Department of Computer Science 05/20/2019

____

Dr. Thomas Austin, Department of Computer Science 05/20/2019

Abstract

Web real-time communication (abbreviated as WebRTC) is one of the latest Web application technologies that allows voice, video, and data to work collectively in a browser without a need for third-party plugins or proprietary software installation. When two browsers from different locations communicate with each other, they must know how to locate each other, bypass security and firewall protections, and transmit all multimedia communications in real time. This project not only illustrates how WebRTC technology works but also walks through a real example of video chat-style application. The application communicates between two remote users using WebSocket and the data encryption algorithm specified in WebRTC technology. This project concludes with a description of the WebRTC video chat application's implementation in Yioop.com, a PHP-based internet search engine.

Acknowledgements

This project would not have seen daylight without the excellent tutelage and staunch support of Dr. Chris Pollett, who has been my advisor. He first introduced me to the concept of WebRTC, of which I had never before heard, and I was a little bit apprehensive at the beginning about working with the technology. Now, I deeply appreciate the opportunity and challenges that I have overcome to work with this technology. He has been patient with me from the beginning of this research project to the end and has also shown me there is more than one way to solve every problem. His problem-solving strategy has helped me tremendously whenever I have come across numerous roadblocks. I would also like to thank Dr. Austin, who taught me how to work with JavaScript, providing numerous tips and tricks that laid the groundwork for the application described in this paper. My special thanks also go to Dr. Moh, who taught me how to work with cloud computing and gave me the chance to work with Amazon Web Services. This experience serves as the backbone of this research project. Finally, I would like to thank my family, whose inspiration and unwavering support in allowing me to pursue my dream got me through tough times.

TABLE OF CONTENTS

I. Introduction 9

II. Background of WebRTC Technology 13

III. Example Walkthrough 30

IV. WebRTC Security 32

V. A Simple Video Chat WebRTC Application on AWS 34

VI. Interface with Yioop.com 35

VII. Conclusion 39

References 40

TABLE OF FIGURES

Figure 1. An Example of SDP …………………………………………………………. 20

Figure 2. Mnemonic names of key used in SDP ……..……………………….……….. 20

Figure 3. createOffer() vs createAnswer() SDP exchange between peers……………..…. 22

Figure 4. An example of NAT table ………………………………….………..……... 24

Figure 5. An example of STUN server vs your desk top ……………………………...… 25

Figure 6. Example of STUN/TURN server ………………………………….………...26

Figure 7. WebRTC protocol stack.……….………………………………………… 33

Figure 8. Snap shot taken from WebRTC video chat inside Yioop.com ………………... 36

Figure 9. Relationship between Yioop, WebSovckets, and Signal Server………………. 38

LIST OF ABBREVIATIONS

API – Application Programming Interface

AWS – Amazon Web Services

ICE - Interactive Connectivity Establishment (ICE)

NAT – Network Address Translation

pc – RTCPeerConnection

SDP - Session Description Protocol

SSL - Secure Sockets Layer

STUN - Session Traversal of User Datagram Protocol [UDP]

TURN - Traversal Using Relays around NAT

UDP - User Datagram Protocol

WebRTC – Web Real Time Communication

Yioop – Yioop.com

I. Introduction

Web real-time communication (WebRTC) is one of the latest communication technologies, and its standards are still under development. Open-sourced by Google in 2011, this technology allows users to share real-time media streaming without depending on a third-party plugin. Now, users with WebRTC-capable browsers can make calls, share files, or participate in video conferences with other users who have the same internet browsers, free of charge. The technology is not a single piece of software but utilizes a collection of many components such as encryption algorithms, HTML5, JavaScript APIs, and several network protocols whose working groups have their own speed on working on their browsers and operating systems. The World Wide Web Consortium (W3C) [1] is responsible for API standardization, and various protocols are standardized by the IETF [1].

When two browsers want to establish communication, they need a broker to help them connect with each other. The role of a signaling server is this broker, but the WebRTC does not specify how this signaling server should be implemented. Although this lack of specification creates much creative room for developers, it creates too many choices, which frustrates them. The research work in this paper uses a homemade signal server written in the PHP language and utilizing Apache Web Server, as well as WebSocket for actively pulling data from both clients and servers.

Because the standards and protocols used in WebRTC implementations are changing, a few minor differences are seen when using different browsers. Some developers prefer to use *adapter.js* [1], which takes care of the differences in browsers and aids in coping with constant specification changes. The website http://caniuse.com lists browsers compatible with the WebRTC changes [1]. For this paper, I decided to follow the W3C standard.

One of the challenges of working with a WebRTC application is building a network connection for two unknown clients. If the two clients share the same network connection, this does not present a problem. However, connecting two browsers located in two unknown locations that pass through different NAT devices requires figuring out how to pass through SSL (Secure Socket Layer), STUN server, Turn Server, and Web Server to reach the other client, which has its own firewall structure. The research in this paper utilizes a homemade signal server that uses WebSocket for pulling and pushing data from either side of the clients and server on top of the TCP.

As of this writing, WebRTC technology is supported by Chrome, Firefox, and Opera running on PC operating systems such as Mac OSX, Windows, Linux, and Android. We can apply this technology in (a) real-time audio and/or video calls, (b) teaching students in remote villages with internet connections, and (c) data/music sharing between users, just to name a few cases.

Today, for smartphones such as iPhones, FaceTime is commonly used for video chatting with another party. It is a typical example of real-time communication that we take for granted nowadays. In the past, applications with video and audio features for real-time communication relied on Flash, Java Applets, or another third-party software to work. When video/audio streams are transmitted over a network, video frames or audio waves are divided first into smaller sizes, which are compressed to facilitate transmission across a network. The other end of the network reverses the packets through decompression. Now, WebRTC bakes these codecs into browser APIs and HTML5. In May 2010, Google bought GIPS (Global IP Solutions), which specialized in the codes and echo cancellation used in videoconferencing software. In May 2011, Google released most of these technologies as open source. WebRTC has been steadily increasing in popularity; since its inception in 2011, it has gained a wide acceptance among developers, and much of this technology has already been commercialized (e.g., tokbox.com, scaledrone.com). Many example demonstrations of WebRTC technology and self-help sites are available to explain what WebRTC is and how to implement it on a desktop. However, commercial sites charge for their services, whereas most free self-help tutorial sites show two browsers running on one desktop sharing the same memory space on one desktop. These tutorial sites provide the basic concepts for WebRTC, but they seldom use two browsers located in different locations, and they rarely work with the encryption technology that WebRTC requires.

This project differs from the many self-help tutoring sites in these regards. Although this research was done in an academic setting, it contains all the necessary components comparable to those found on commercial WebRTC sites. First, it is free, and it encrypts all transactions using a legitimate SSL certificate. It uses a STUN server and a legitimate TURN server, as well as WebSocket for pulling data from both clients. It also runs on two different browsers located on two different desktops rather than on a single desktop. These are the major ways in which this project differs from the self-help teaching WebRTC sites and commercial sites. This project is available free, without charge, to any user with an internet connection.

This project is divided into several chapters to present WebRTC technology in a more meaningful way. Chapter I addresses an introduction about what WebRTC is about and who is responsible for this technology. Chapter II gives background information as to how each component supports building a WebRTC application, along with detailed information about the component.

Chapter III puts all the elements together and walks the reader through a pseudo-example, step by step. Chapter IV discusses the WebRTC security requirements and followed by Chapter V explaining the steps involved implementing a video chat WebRTC application on Amazon Web Services. Chapter VI discusses the steps for porting this technology into a Yioop environment from both a user and a technical perspective. This paper concludes with Chapter VII with final words on the author’s perspective.

II. Background of WebRTC Technology

With WebRTC technology, we can make real-time video calls to people on the other side of the country, share music and data with our friends, or listen to lions roaring in Africa in real time. However, these technologies are not readily available for free. WebRTC provides JavaScript APIs, protocol rules, and encryption guidelines with which we can build an application for real-time communication [2].

Three APIs come with WebRTC, along with many other underlying components such as an encryption framework, STUN/TURN servers, a signaling server, ICE, SDP, NAT, UDP, and TCP to make direct peer-to-peer connection possible. Next, we are going to look at each of these components in more detail.

The three JavaScript APIs are (a) the getUserMedia API, which is responsible for a browser accessing audio and video streams; (b) the RTCPeerConnection API, which handles major communication work as well as exchanging media streams; and (c) the RTCDataChannel API, responsible for transferring data from one peer to another.

1. getUserMedia() API

Before HTML5 is on board, if we want to capture audio or video stream on a computer, we must rely on Flash or JavaScript-based plugins to do the job. Now, this hardware accessing capability is directly baked into HTML5, and video and audio are integrated into a browser. The browser can access a user’s camera and microphone by specifying video and audio tags on the HTML5 page.

In the video chat WebRTC application, we specify these tags in the method navigator.getUserMedia(), which needs two additional parameters to handle what to do when the video/audio streams are captured successfully, and what happens when it fails to capture them. When running this application in a browser that specifically asks users' permission whether it is okay to use webcam and audio, in a WebRTC application, encryption is one of the required processes for all layers of the protocol. Messages passed through between browsers are not readable if they are stolen or hijacked in the middle of the transmission. We will talk more about these encryption attributes in the Security section.

Although we have a half-decent code to specify video/audio streams in the navigator.getUserMedia() method, at most, this method alone does not go beyond showing the user's face. It must be registered with RTCPeerConnection().

2. RTCPeerConnection API

The RTCPeerConnection object is responsible for all levels of components in connecting two browsers so that they can share real-time media. It initializes a connection and gathers ICE candidates, which means gathering the browser's public IP number and port address between two browsers. When two browsers want to share data with each other, they must have a way to exchange three different kinds of information. First, they must determine when to connect or close through the session control information. Next, they must find a way to exchange network data such as the IP address and port numbers of each peer. Finally, they must know how to handle media data such as the codecs and media types of the peers used in the connection.

Here we describe succinctly how RTCPeerConnection API can be used to provide a general idea. In reality, this API uses a lot more events, methods, and properties than are given here. A lot of underlying protocols accompany this API, of which we will give more detailed information in this section.

The parameter used to create the RTCPeerConnection object lists an array of STUN and TURN servers used for locating the ICE candidates. Google provides free public STUN servers at code.google.com, but not many free TURN servers are available, and reliable TURN servers are commercial. This paper uses a public TURN server from https://github.com/pions/turn.

First, a caller (peer) must create a new RTCPeerConnection object, which we name pc. Now, pc must create an ICE candidate using onicecandidate(), which returns a list of ice candidates, returned by STUN/TURN servers, which contain the current browser's public IP address, as well its port number, which are passed to the target peer through the signal server.

Next, the pc object obtains a local and remote media stream through the getUserMedia method. This media steam must be attached to RTCPeerConnection via the onaddstream() method for a remote media stream and *addStream()* method for a local video/audio stream.

The caller pc must create an offer using the createOffer() method. This offer contains the codecs, encryption methods, and any candidates already gathered by the ICE agent, all of which are wrapped inside SDP, which is passed as a parameter of a new RTCSessionDescription(offer) object. Then pc uses the RTCPeerConnection.setLocalDescription() method to send to its target peer through a signaling server.

When the callee pc receives the offer from the previous step, it goes through the same process as the createOffer() step, except that this time, it uses the createAnswer() method, which is wrapped inside the pc.setRemoteDescription() method. However, the RTCPeerConnection() API does not work by itself; it relies on several protocols and underlying supporting architectures to make the connection work. This section might be the right place to address these underlying technologies in association with the RTCPeerConnection API.

3. What Is a Signaling Server?

Now, we know that WebRTC allows real-time peer-to-peer communication for sharing between two peers. Behind the connection, each peer must go through finding the other peer, which is behind a different firewall, router, or network. Each peer also needs to figure out the other peer's codecs, settings, bandwidth, IP address, and port accessed by outsider. This connection cannot be established by these two peers alone; they need a medium through which they can connect. A signaling server fits the bill for establishing and coordinating communication between these two peers.

In addition, the connection between these two peers must be secure so that the original packets in transit will not be readable or modifiable if either peer is attacked during packet transmission; this is a mandatory WebRTC requirement. However, this signaling process has not been defined by the WebRTC specification, and an application developer who wants to develop a WebRTC application must figure out how to build it by him- or herself. A plausible reason for this lack of information might be to make room for interoperability among different protocols; its outline can be found by the JavaScript Session Establishment Protocol [2].

We can use any language or any protocols to build a signaling server; this research paper provides two ways to build one. One is written for a WebRTC video chat application whose signaling server is written in Node.js using WebSocket. The other, which is ported inside Yioop.com, is written in PHP in conjunction with WebSocket to work in the Yioop.com internet search engine. The basic signaling process concept is the same in both cases: to exchange messages between two browsers.

Several commercial signaling servers are available, such as Asterisk and OnSip. Skype, which is a typical example of a real-time audio and video communication technology, uses its own proprietary protocol for a signaling server, and its service is not free. Google “Hangouts” is free, but its software must be downloaded first.

When starting the signaling process, the two browsers do not know each other's codecs or the media types that will be used during the connection. Interactive Connectivity Establishment (ICE) handles this negotiation process between the two browsers. Each ICE candidate contains its IP address and port number, which the other peer can understand. Two peers must exchange ICE candidates to establish a connection. As soon as the two peers agree upon ICE candidates, they begin to exchange the video stream and data. Even after their connection has been established, they continue exchanging ICE candidates, hoping to find better options during the process until the current session ends.

To give a real example of how ICE candidates are coded in an application, I have each ICE candidate contain a JSON string message of type "candidate," which is sent over the signaling server to the remote peer.

Once the caller, which we name Alice, finishes gathering ICE candidates, Alice creates an offer to initiate the call to the other party. This offer, which is in the Session Description Protocol (SDP) format, is delivered to the target peer, which we name Bob. Bob creates an answer message in SDP format in response to the offer from Alice. My signaling server uses WebSocket to transmit offer messages with the type "webrtcmessage." Now, Alice can share audio/video stream with Bob, and the connection between them has been established, providing all the supporting protocols have been set up.

4. Session Description Protocol (SDP)

We have discussed the importance of a signal server in the WebRTC application in the previous section and the critical part it plays in exchanging data between two peers; however, the signal server cannot work alone. It needs support from several other underlying protocols to perform its function, of which SDP is one.

The main role of SDP is to share media-based information with other peers over a network. SDP includes the name, purpose of the session, media type, protocols, codec and its settings, timing, and transport information. An SDP description is created when the RTCPeerConnection object starts collecting ICE candidates for setting up a connection with another user.

SDP has been around for a while (since the late 1990s) for media-based connections, and it has seasoned out through numerous other types of applications, such as phones, before it began to be heavily used in WebRTC.

SDP has a text-based format and comprises a set of key-value pairs, with a line breaker at the end. Here is one example: “<key>=<value>\n”. This key uses a single character that stands for the type of value, with the value being a machine-readable configuration value. It uses mnemonic names such as those shown below.

A copy of SDP is given below, as quoted from RFP 2327:

```
v = 0
o = mhandley2890844526 2890842807 IN IP4 126.16.64.4
s = SDP Seminar
i = A Seminar on the session description protocol
u = http://www.cs.ucl.ac.uk/staff/M.Handley/sdp.03.ps
e = mjh@isi.edu(Mark Handley)
c = IN IP4 224.2.17.12/127
t = 2873397496 2873404696
a = recvonly
m = audio 49170 RTP/AVP 0
m = video 51372 RTP/AVP 31
m = application 32416udp wb
a = orient:portrait
```

Figure 1. An example of SDP [5]

Session Description Parameters with optional parameter being a *
v = (protocol version)
o = (owner/creator and session identifier)
s = (session name)
i =* (session information)
u =* (URI of description)
e =* (email address)
p =* (phone number)
c =* (connection information – not required if included in all media)
b =* (bandwidth information)
z =* (time zone adjustments)
k =* (encryption key)
a =* (zero or more session attribute lines)

Figure 2. Mnemonic names of keys used in SDP

Having introduced the SDP’s role as media relay, we will present it in a pseudo - example to show how it is used in the video chat application. First, we create a new object from RTCPeerConnection classes, which we name pc. Then, the pc object creates an SDP description from a local device using the createOffer() method, sets it as a local description of the current session, and sends it to the target peer. This process initiates a new WebRTC connection to a remote peer, while the signaling server helps to establish the connection between peers and allows the SDP data to flow between them.

SDP, a string-based protocol, passes the media information to the other peers. The JavaScript Session Establishment Protocol [2] abstracts out these inner SDP settings, and we are not concerned about how this session information has passed between the two browsers. Application developers have some freedom on how the SDP is encoded in their application. Next, we can extend more lines to getUserMedia(), which has been described in the previous section. However, the media stream that we captured through getUserMedia() does not go anywhere. This method must be attached to the pc object, which calls createOffer() to generate the SDP description.

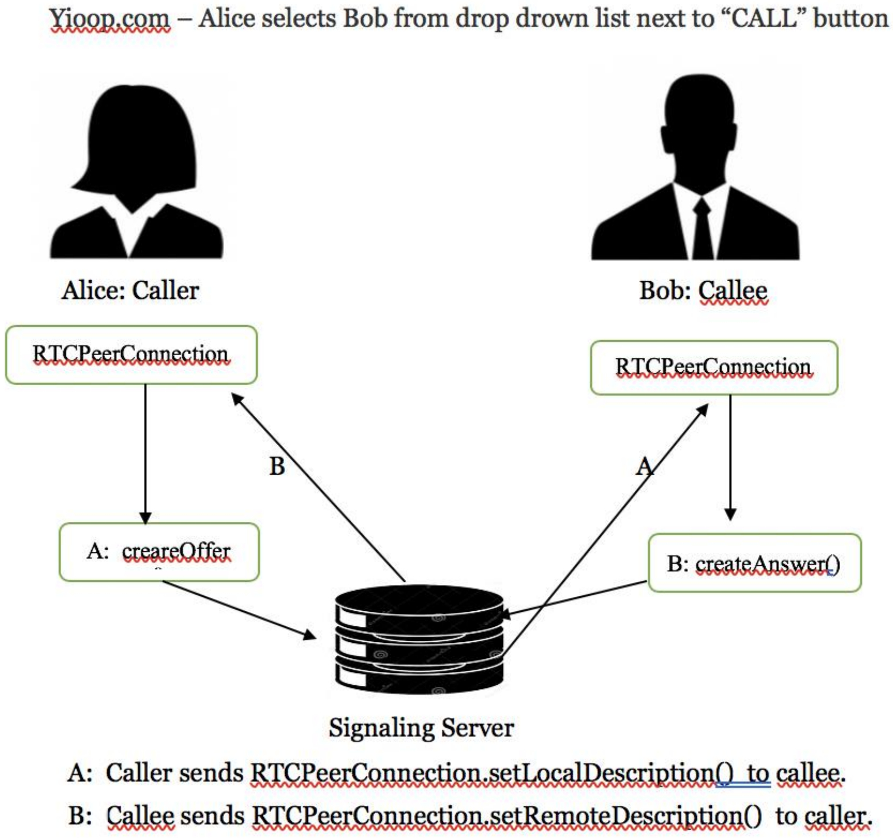
In a nutshell, the WebRTC video chat application goes through the following steps in conjunction with the SDP perspective:

a. Alice creates a new object of the signaling server and a new object of RTCPeerConnection connection and calls them signalServer and pc.

b. Alice attaches getUserMedia() method to pc.

c. Alice creates the SDP (offer) description and attaches it to a local description of the current connection.

d. Alice sends the generated SDP offer to remote peer via signalServer, and Bob returns an answer in an SDP format in a remote description using signalServer. Now, both peers have established a connection. This scenario is pictured in the following diagram [2].



Yioop.com – Alice selects Bob from drop drown list next to “CALL” button

A: Caller sends RTCPeerConnection.setLocalDescription() to callee.

B: Callee sends RTCPeerConnection.setRemoteDescription() to caller.

Figure 3. createOffer() vs. createAnswer() SDP exchange between peers

5. UDP (User Datagram Protocol)

In a WebRTC application, media streams are transmitted in real time. To meet this real-time communication requirement, WebRTC chooses to use UDP, preferring timeliness over reliability and low latency over the ordered stream of packets used in TCP. As a result, even if audio and video streaming applications occasionally lose a few packets, the audio and video codecs can fill in small data gaps, and most users do not notice a difference.

UDP offers no promises on reliability, does not guarantee the order of the data, and delivers each packet to the application the moment it arrives.

WebRTC uses UDP at the transport layer: latency and timeliness are critical for UDP. However, UDP does not work alone; it needs support from other NATs layers and firewalls, negotiates the parameters for each stream, provides encryption of user data, and implements congestion and flow control.

6. Network Address Translation (NAT)

NAT is a process where a network device, usually a firewall or a router, assigns one external IP address that is mapped to a computer (or group of computers) inside a private network. This allows several local devices to connect to one public IP address to conserve the IPv4 address, which reaches its limitation on numbering scheme. When a device on the local network tries to send packets to outside of its network, the NAT translates the IP address to match the external address.

NAT devices also play an important role for screening outside calls for security.

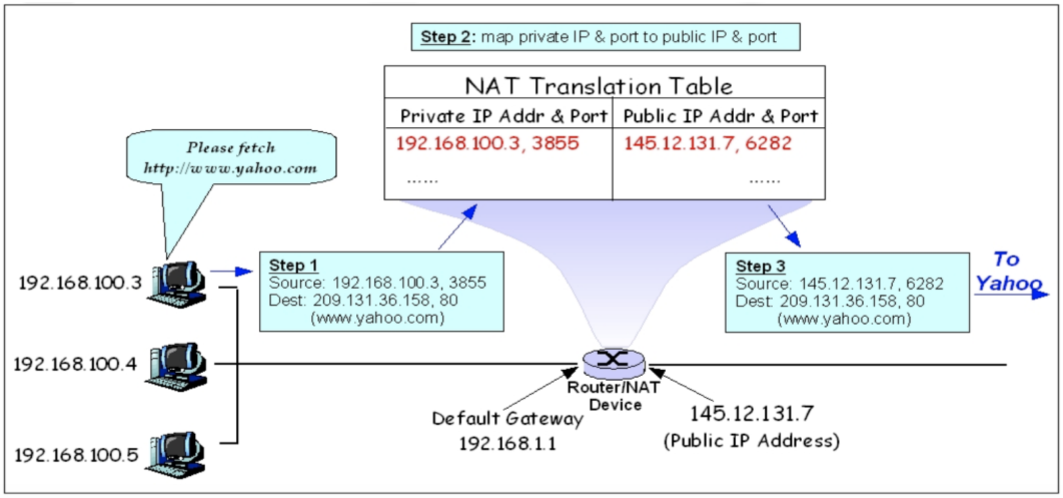


Figure 4. Network address translation

7. Session Traversal Utilities for NAT (STUN) Servers

The STUN server is a standardized set of methods of NAT traversal that includes a network protocol for real-time voice, video, messaging, and other interactive communications. More simply, the main role of the STUN server is to respond to a public IP address. Clients in a WebRTC application use a STUN server to determine their public IP address, and the ICE framework in WebRTC handles finding a suitable STUN server during connection establishment. STUN servers are available free on public sites; suitable servers can be found at https://gist.github.com/zziuni/3741933. STUN servers may not work in some cases because of some network architectures or NAT device types, so STUN servers work along with TURN and ICE, which the next section will discuss.

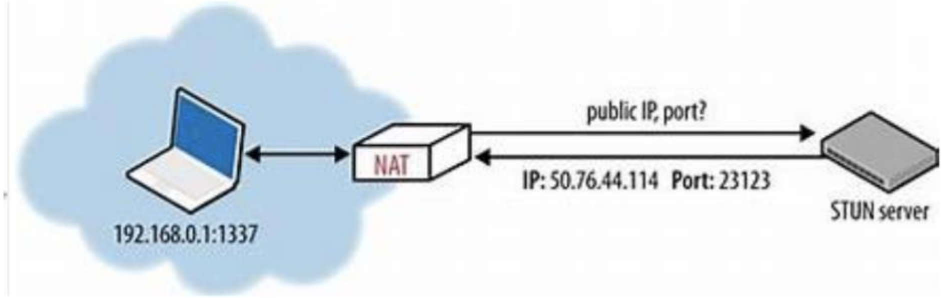


Figure 5. An example of STUN server vs. a desktop

8. Traversal Using Relays around NAT (TURN) Servers

A TURN server is responsible for transmitting audio/video/data streaming between peers. When these data are traversing from one browser to another, they must pass through different network devices. Most of the time, the STUN server is good enough, but sometimes this might not work because of network architecture or firewall devices. Then, we rely on a TURN server, which serves as a relay point to allow the media data to flow through it. The RTCPeerConnection() interface tries to establish a direct peer-to-peer connection between peers over STUN servers. Sometimes, however, this might not work; then, TURN servers are used for relaying stream data between endpoints. TURN servers require high bandwidth, such that most of the time free public TURN servers are rare. Use of a reliable TURN server may require payment.

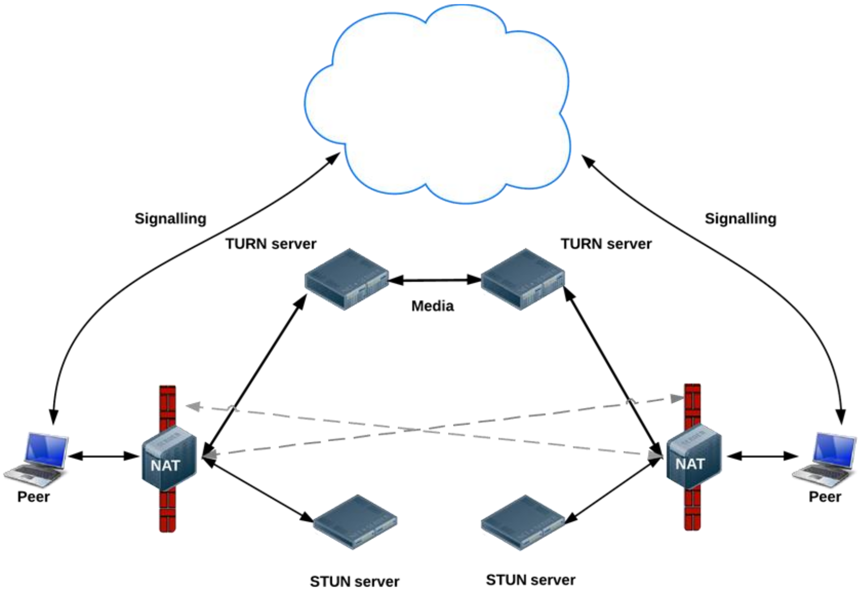


Figure 6. Example of a STUN/TURN server [6]

9. Interactive Connectivity Establishment (ICE)

Once the signaling server has been set up, we need to use ICE to get around with NATs and firewalls. In the real world, most peers sit behind some kind of NAT device. If a peer inside NAT wants to reach an external IP address, it must go through NAT to reach the targeted public IP address. WebRTC apps use the ICE framework to get around this NAT device to find the best option to connect peers. ICE first tries to find the host address by querying its operating system. Sometimes this search might not work because of NAT devices or firewalls; then, ICE relies on a STUN server to obtain its targeted external address. If this still fails, it resorts to a TURN server as a last resort. An ICE candidate is essentially a description of how to connect to a client. For a client to connect to another client, both clients must share their ICE candidates with each other.

Let us assume that our users, Alice and Bob, are both using a WebRTC video chat application, and that Alice wants to call Bob. Here is what happens next:

To connect to Bob’s browser, Alice’s browser must generate a Session Description Protocol (SDP) offer. The SDP generation process begins when the application she is using calls createOffer on an RTCPeerConnection object.

An SDP offer contains a bunch of information about the session Alice’s browser wants to establish: what codecs to use, whether this will be an audio or video session, and more. It also contains a list of ICE candidates, which are the IP and port pairs that Bob’s browser can attempt to use to connect to Alice.

To build the list of ICE candidates, Alice’s browser makes a series of requests to a STUN server. The server returns the public IP address and port pair that originated the request. Alice’s browser adds each pair to the list of ICE candidates. This process is called gathering ICE candidates. Once Alice’s browser has finished gathering ICE candidates, it can return an SDP.

Next, Alice’s browser needs to pass the SDP to Bob’s browser through a signaling channel between the browsers; WebRTC leaves this signaling implementation up to the developer. The ins and outs of signaling are beyond the scope of this discussion, but let us assume Bob receives Alice’s SDP offer via some signaling channel. Now, Bob’s browser needs to generate an SDP answer. Bob’s browser follows the same steps that Alice’s browser used above (e.g., gathering ICE candidates). Bob’s browser then must return this SDP answer to Alice’s browser.

Once Alice and Bob have exchanged SDPs, they then perform a series of connectivity checks. The ICE algorithm in each browser takes a candidate IP/port pair from the list it received in the other party’s SDP and sends it a STUN request. If a response comes back from the other browser, the originating browser considers the check successful and will mark that IP/port pair as a valid ICE candidate. After connectivity checks have finished on all of the IP/port pairs, the browsers negotiate and decide to use one of the remaining valid pairs. Once a pair is selected, media begins flowing between the browsers. This entire process usually takes milliseconds.

requests to the TURN server to obtain a media relay address. If the browsers cannot find an IP/port pair that passes connectivity checks, they will make STUN. A relay address is a public IP address and port that will forward packets received to and from the browser to set up the relay address. This relay address is then added to the candidate list and exchanged via the signaling channel.

If one is building a WebRTC application, the WebRTC stack includes an ICE agent that takes care of most of this. One just needs to implement a signaling mechanism to exchange SDPs and send along new ICE candidates whenever they are discovered [2].

10. RTCDataChannel API

WebRTC allows not only the transmission of video and audio streams, as we have discussed in the previous section, but also the transmission of arbitrary data over a network using RTCDataChannel API. The RTCDataChannel.createDataChannel() method allows users to create a channel between two peers over which they may exchange data. This API closely resembles the WebSocket API, so that users can use the same programming model. Because the video chat application in this paper does not use the RTCDataChannel API, our discussion about RTCDataChannel is limited to this.

III. Example Walkthrough

We have thus covered most of the components used in WebRTC, and it is time to walk through an example from the perspective of the two users. Here, Alice starts a connection with Bob, another client, who wants to share a video chat with her.

First, WebRTC starts a new signal server to initiate a new signal server. Then Alice creates a new RTCPeerConnection object, which is attached to an onicecandidate () EventHandler. This EventHandler prompts the local ICE agent to pass a message to its peer over the signaling server. Alice captures her video and audio streams using navigator.mediaDevices.getUserMedia(). Then, Alice calls RTCPeerConnection.addTrack() to attach her video and audio stream to the pc instance. Alice creates an offer using the RTCPeerConnection.createOffer() method. Alice sets her SDP offer as the local description by calling RTCPeerConnection.setLocalDescription(). This localDescription is her end of the connection description. This pc.setLocalDescription() starts gathering candidates using STUN servers and sends the gathered ICD candidates to the other peer over the signal server.

Bob receives the offer from Alice and calls RTCPeerConnection.setRemoteDescription() to set it as his remote description, which is also Alice's description. Bob also captures the media stream and attaches it by calling RTCPeerConnection.onaddstream(). Bob then creates an answer to respond to Alice's offer by calling RTCPeerConnection.createAnswer(). The return value from this call is wrapped inside the function call, RTCPeerConnection.setLocalDescription(createdAnswer), which sets the answer as its local description, and is sent to the other peer over the signal server.

Now, Alice receives the answer from Bob and calls RTCPeerConnection.setRemoteDescription() to set it as her remote description of her connection. Both share the description, and the media stream passes through to both ends.

IV. WebRTC Security

Because real-time communication can transmit audio, data, and video streams in real time, there are many opportunities for that information to leak to a third party during transmission. This can happen during peer-to-peer communication or peer-to-server communication, with a third party acting as the man in the middle, hijacking messages. Encryption prevents a third party from eavesdropping or acting as the man in the middle to massage the original data. Encryption is a mandatory feature of WebRTC and is enforced in all implementations of the protocol.

The encryption technology must satisfy several requirements for use in peer applications. If messages are stolen during transmission, they must not be readable. If a man in the middle is able to attack a connected peer, the message in transit cannot be editable. The encryption algorithm should use the highest bandwidth possible between the clients; Datagram Transport Layer Security (DTLS) fits these requirements. The reason that DTLS was chosen was to have a simple and easy-to-use protocol such as TLS and because it works with the UDP transport layer. It is modeled after the TLS protocol, carrying many similar features in addition to supporting UDP.

Different encryption protocols are used depending on channel type; for instance, data streams are encrypted using DTLS, and media streams are encrypted using the Secure Real-Time Transport Protocol [5].

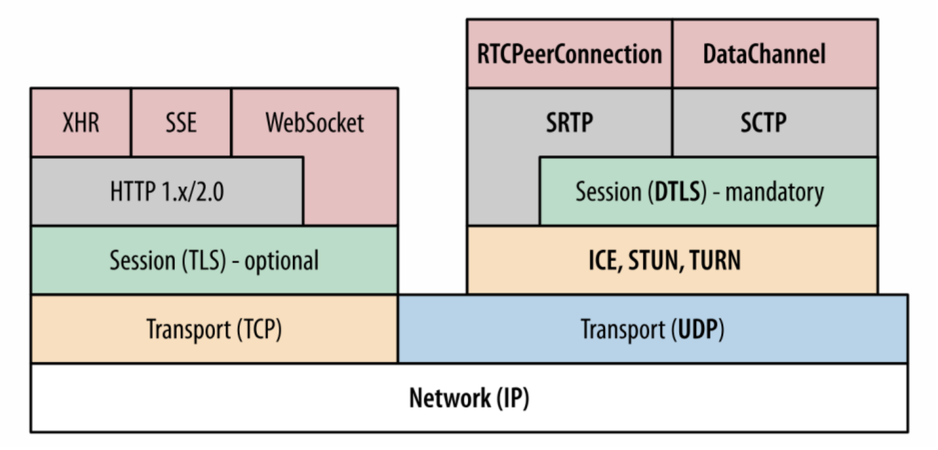


Figure 7. WebRTC protocol stack [5]

With WebRTC applications, end users must grant explicit permission before the browser is allowed access to their local devices. The getUserMedia() method in WebRTC requires security permission to use webcam or audio features. Furthermore, when the camera and/or microphone are active, the browser will display an “active” indicator, usually found in the browser tab. WebRTC security measures ensure that media is automatically encrypted. For this project, I generated SSL certificates with letsencrypt from https://letsencrypt.org.

V. A Simple Video Chat WebRTC Application on AWS

Because I need a signal server to run to connect two browsers, I need a public VPN to host my application, and Amazon Web Service (AWS) seems a good place to run this application. One handy thing about AWS is that it allows us to create a domain name, along with an option to create SSL certificates for users.

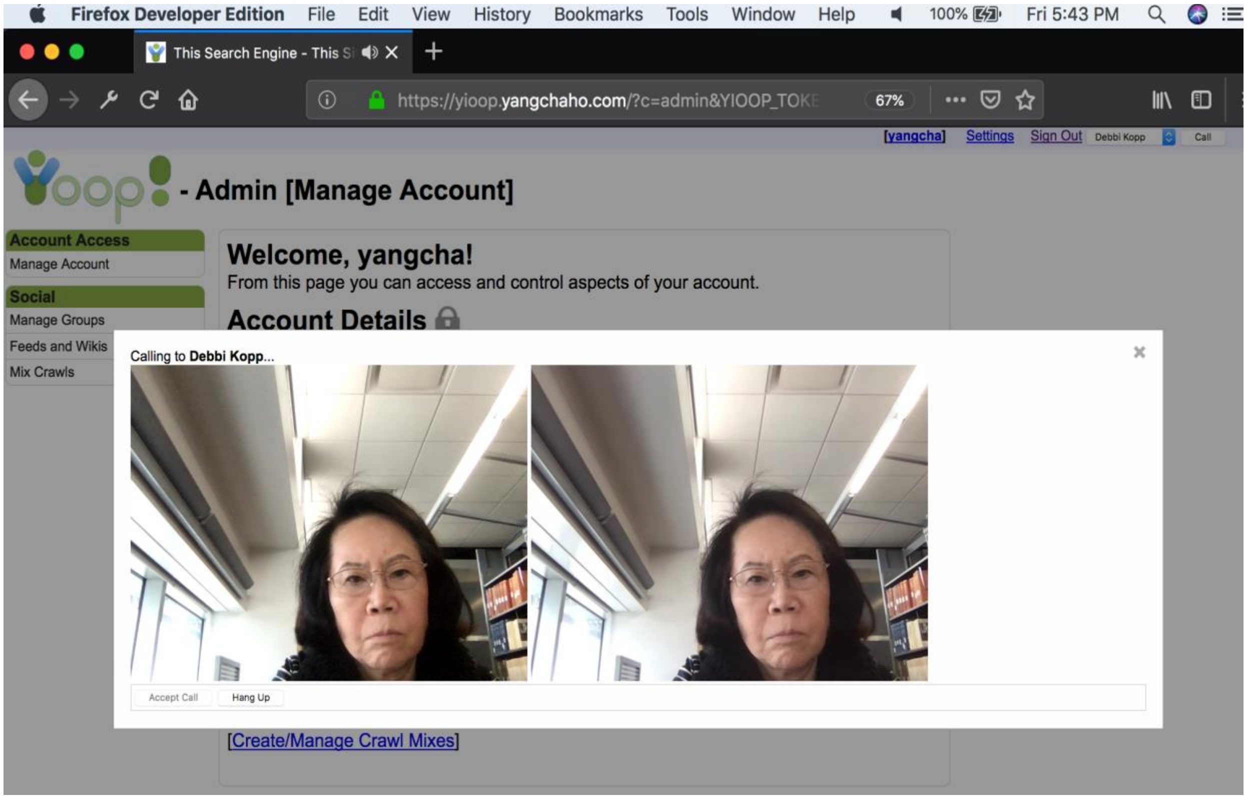
I grouped my signal server task into two stages: the first step was to set up an environment where this application could run, and the second step was to provide all the tools so that the application could run. In more detail, the first step was to (a) create an account with AWS and create an EC instance with Centos or Ubuntu with the smallest unit available (there is a charge for this service). Then I created a domain name for my application: yangchaho.com. You can use any name you want, but there is a charge for this service.

(b) Next, I allowed this domain name to point to the IP address of the instance and set up a security group to specify which port(s) will be used for my application. I signed on to the instance again, installed ngix, PM2, node.js, and possibly other related software, and put the webrtc project folder somewhere on the instance. I used PM2 to run the node.js server as a service in the background. The last step was to enter this command on my browser: https://webrtc.yangchaho.com:16443.

VI. Interface with Yioop.com

1. How the Application Works from the User Perspective

This section describes the process by which the WebRTC video chat application developed in the previous section is ported inside the Yioop.com search engine (“Yioop” for short). A user named Alice logs in to Yioop and creates her user ID, providing all the necessary credentials in her browser to create it. Then she signs in to her account on Yioop. Bob, another user, does the same process to obtain his user ID. He signs in to Yioop in another browser and wants to have a video chat with Alice, who is a thousand miles away. He clicks the call button in the upper right-hand corner, which lists all the users currently logged in to Yioop. Bob, who selects Alice from the call button dropdown list, wants to connect with her. Now, on Alice’s screen, she sees herself on the screen, and underneath her video face is an accept button in the lower left-hand corner. Bob must accept the call in the same way. Now, both are connected and are able to chat with each other. A sample demo screenshot is shown in Figure 8.



Firefox Developer Edition File Edit View History Bookmarks Tools Window Help
100%
Fri 5:43 PM
This Search Engine - This S
https://yioop.yangchaho.com/?c=admin&YIOOP_TOKE
67%
[yangcha] Settings Sign Out Debbi Kopp Call
Account Access
Manage Account
Social
Manage Groups
Feeds and Wikis
Mix Crawls
Accept Call
Hang Up
[Create/Manage Crawl Mixes]

Yioop! - Admin [Manage Account]

Welcome, yangcha!

From this page you can access and control aspects of your account.

Account Details

Calling to Debbi Kopp...

Figure 8. Snapshot taken from WebRTC video chat inside Yioop.com

2. Technical Perspective

When a user logs in to Yioop, it opens the WebSocket connection to the signaling server and maintains it until the web page closes. Then, the WebSocket connection drops, and the signaling server knows that the user is offline. This way, the signaling server always “sees” who is logged in “online.” Suppose you just log in, and Yioop opens a WebSocket connection to the signaling server. Once the connection has been made, the signaling server sends back to Yioop a list of all users who are currently online. Here, we can see the list of the online users on Yioop.

Suppose Bob logs in to Yioop and selects Alice from the dropdown list—assuming that Alice has already logged in. Both Bob and Alice are connected to the signaling server via the WebSocket protocol, and the signaling server can send messages to each one. When Bob clicks “call” to call Alice, Yioop sends a message to the signaling server and informs it that Bob wants to call Alice. At the same time, the signaling server sends the message to Bob. Then, Yioop shows the green circle, indicating another user is calling him; the callers exchange relevant WebRTC data and establish the call. We put a WebSocket server into the signaling server and put the WebSocket client part into the Yioop page. Then, any user who logs onto the page will establish a connection with the WebSocket server, thus enabling all logged in users to communicate with one another via the WebSocket server (which is part of the signaling server). This application is written in PHP, which runs on the server and listens for WebSocket connections on TCP port 2002. A snap shot of the relationship between signal server, WebSockets, Https, and Yioop is given in Figure 9.

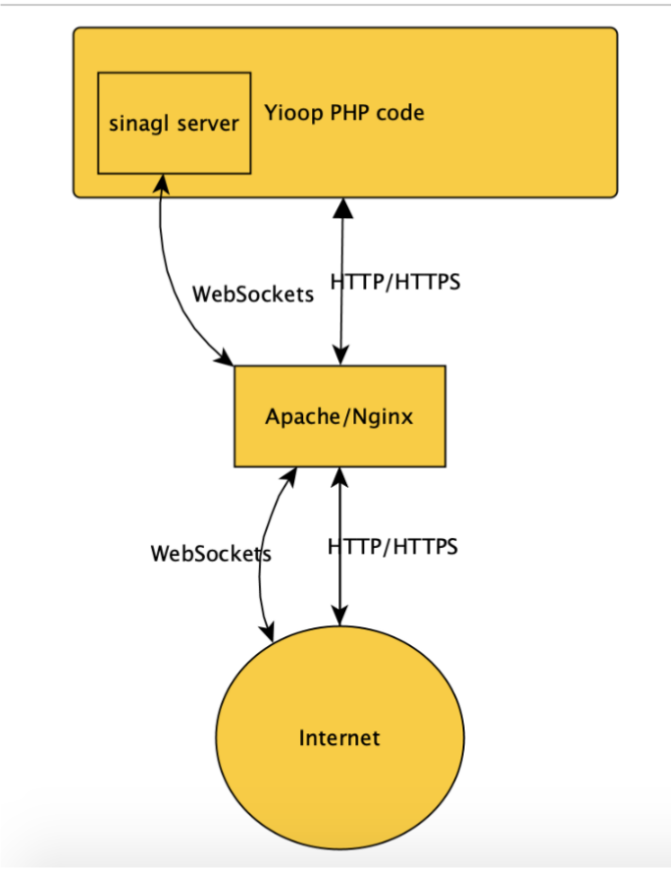


Figure 9. Relationship between Yioop, WebSovckets, and Signal Server

VII. Conclusion

WebRTC is a pretty new technology. Most browsers support it; however, its APIs are still in the development stage. There is a plethora of information as to how its architecture works, but good examples that can be readily run are lacking. The reason for this is that there is some overhead involved in setting up even a very simple WebRTC application, such as finding a reliable TURN server, getting an encrypted SSL certificate, setting up a domain name, and finding a public server to host this application.

However, once you set up the running environment, the communication between peers flows more smoothly. There is a lot of potential in the near future for remote locations where direct traffic is limited. For example, this application could be a strong candidate for reaching people living in rural areas, where commuting is not convenient for medical help or educational purposes. For the next task, I would like to continue with this technology to enable multiple users to participate in a conference or classroom. The current technology is limited to a one-to-one connection between two clients. Hopefully, we could extend this to one client, such as a teacher, and to many other clients (students) in a scenario such as teaching in a remote village.

References

[1] Manson, Rob. *Getting Started with WebRTC*. Packt Publishing Ltd, 2013.

[2] Grigorik, Ilya. *High Performance Browser Networking: What every web developer should know about networking and web performance*. " O'Reilly Media, Inc.", 2013.

[3] Sergiienko, Andrii. *WebRTC Blueprints*. Packt Publishing Ltd, 2014.

[4] Sergiienko, Andrii. *WebRTC Cookbook*. Packt Publishing Ltd, 2015.

[5] Ristic, Dan. *Learning WebRTC*. Packt Publishing Ltd, 2015

[6] Dutton, Sam. "Getting started with WebRTC." *HTML5 Rocks*23 (2012).

[7] Commons, Wikimedia. "Wikimedia commons." *Retrieved June*2 (2012).