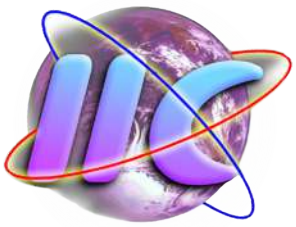


INSTITUTE OF INFORMATICS AND COMMUNICATION
UNIVERSITY OF DELHI SOUTH CAMPUS

WebRTC Driven Communication Platform

-

WebRTC based web application for full-duplex video communication

- HITESH KUMAR
- 4115
- M.SC. INFORMATICS 2016-18

DECLARATION

I hereby declare that the project entitled “WebRTC driven communication platform” being submitted by me at Institute of Informatics and Communication, University of Delhi South Campus towards the partial fulfilment of the requirement for the award of Degree of MSc. Informatics. I hereby also declare that this Project work is my original work. The matter embodied in this project report has not been submitted to any other Institution or University for the award of any degree, diploma, certificate etc.

Hitesh Kumar

ACKNOWLEDGEMENT

I wish to express my deep sense of gratitude and indebtedness to Dr. Sanjeev Singh for his priceless technical guidance and active supervision at all stages of this work. He played a key role at every stage of this project.
I would also like to extend my sincere thanks to my parents & member of Institute of Informatics and Communication (IIC) for their kind cooperation and encouragement which helped me in completion of this project.

Thank You.

Hitesh Kumar

CERTIFICATE

This is to certify that the summer training project titled “WebRTC driven communication platform” in the partial fulfilled of the requirement of the degree M.Sc. Informatics of Institute of Informatics and Communication, University of Delhi South Campus is a record of bonafide work carried out under my supervision and guidance. The period of summer training was from 25-05-2017 to 18-07-2017.

Dr. Sanjeev Singh
(Project Coordinator)

TABLE OF CONTENTS

INTRODUCTION

Web technologies have experienced a major development during the last years. The power and versatility demonstrated by these technologies point the browser as the appropriate platform for the development of a new breed of applications. No need of installation, always updated and available worldwide, are some of the advantages inherent to this kind of applications.

Imagine a world where your phone, TV and computer could all communicate on a common platform. Imagine it was easy to add video chat and peer-to-peer data sharing to your web application. That's the vision of WebRTC : Real-time communication without plugins.

Included as one more API of the HTML5 specification, the WebRTC web API enables web browsers with real-time communications capabilities using simple JavaScript. WebRTC has already shown a great ability establishing high quality videoconferences.

WebRTC is being standardized by the World Wide Web Consortium (W3C) and the Internet Engineering Task Force (IETF). The reference implementation is released as free software under the terms of a BSD license. OpenWebRTC provides another free implementation based on the multimedia framework GStreamer.

WebRTC is supported in the following browsers.

- Desktop PC
    - Microsoft Edge 12
    - Google Chrome 28
    - Mozilla Firefox 22
    - Safari 11
    - Opera 18
    - Vivaldi 1.9
- Android
    - Google Chrome 28 (enabled by default since 29)
    - Mozilla Firefox 24

        - Opera Mobile 12
- Chrome OS
- Firefox OS
- Blackberry 10
- iOS 11
        - MobileSafari/WebKit
- Tizen 3.0

HISTORY OF WebRTC

One of the last major challenges for the web is to enable human communication via voice and video: Real Time Communication, RTC for short. RTC should be as natural in a web application as entering text in a text input. Without it, we're limited in our ability to innovate and develop new ways for people to interact.

Historically, RTC has been corporate and complex, requiring expensive audio and video technologies to be licensed or developed in house. Integrating RTC technology with existing content, data and services has been difficult and time consuming, particularly on the web.

Gmail video chat became popular in 2008, and in 2011 Google introduced Hangouts, which use the Google Talk service (as does Gmail). Google bought GIPS, a company which had developed many components required for RTC, such as codecs and echo cancellation techniques. Google open sourced the technologies developed by GIPS and engaged with relevant standards bodies at the IETF and W3C to ensure industry consensus. In May 2011, Ericsson built the first implementation of WebRTC.

WebRTC has now implemented open standards for real-time, plugin-free video, audio and data communication. The need is real:

- Many web services already use RTC, but need downloads, native apps or plugins. These includes Skype, Facebook (which uses Skype) and Google Hangouts (which use the Google Talk plugin).
- Downloading, installing and updating plugins can be complex, error prone and annoying.
- Plugins can be difficult to deploy, debug, troubleshoot, test and maintain—and may require licensing and integration with complex, expensive technology. It's often difficult to persuade people to install plugins in the first place!

The guiding principles of the WebRTC project are that its APIs should be open source, free, standardized, built into web browsers and more efficient than existing technologies.

WebRTC APIs

WebRTC applications need to do several things:

- Get streaming audio, video or other data.
- Get network information such as IP addresses and ports, and exchange this with other WebRTC clients (known as *peers*) to enable connection, even through NATs and firewalls.
- Coordinate signaling communication to report errors and initiate or close sessions.
- Exchange information about media and client capability, such as resolution and codecs.
- Communicate streaming audio, video or data.

To acquire and communicate streaming data, WebRTC implements the following APIs:

- MediaStream: get access to data streams, such as from the user's camera and microphone.
- RTCPeerConnection: audio or video calling, with facilities for encryption and bandwidth management.
- RTCDataChannel: peer-to-peer communication of generic data.

MediaStream (aka getUserMedia)

We use the MediaStream API in order to gain access to the user's camera and microphone. As stated in the W3C Editor's Draft titled Media Capture and Streams, the MediaStream interface is used to represent streams of media data, typically (but
not necessarily) of audio and/or video content.

The MediaStream API represents synchronized streams of media. For example, a stream taken from camera and microphone input has synchronized video and audio tracks.

Each MediaStream has an input, which might be a MediaStream generated by navigator.getUserMedia(), and an output, which might be passed to a video element or an RTCPeerConnection.

```
navigator.getUserMedia(constraints, successCallback, errorCallback);
```

The getUserMedia() method takes three parameters:

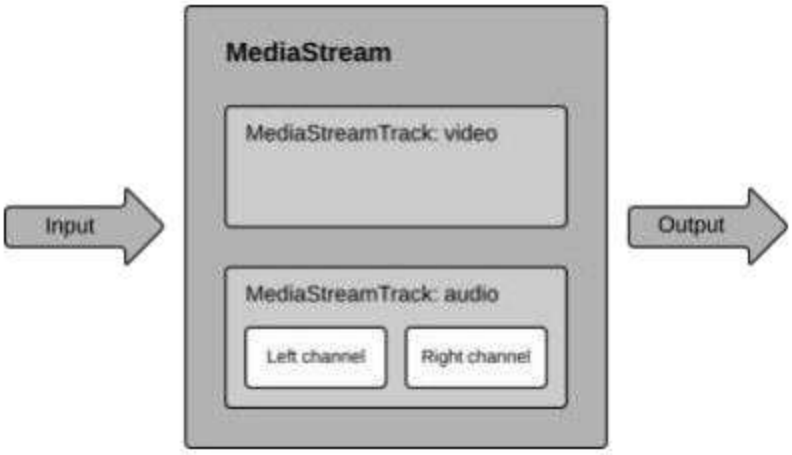
- A constraints object.
- A success callback which, if called, is passed a MediaStream.
- A failure callback which, if called, is passed an error object.

constraints: This variable allow us to indicate constraints on the MediaStreamTracks we want to obtain. In our case the value of this variable is: {video: true, audio: true}

successCallback: This parameter indicates the function that will be called if the getUserMedia request is successful. In our case the local video will be attached to the HTML video element located in the local user area with that purpose and the user will start calling the rest of the users that have already joined the room.

errorCallback: This parameter indicates which function will be called if the getUserMedia request fails. In this case, the user will be alerted about the error.

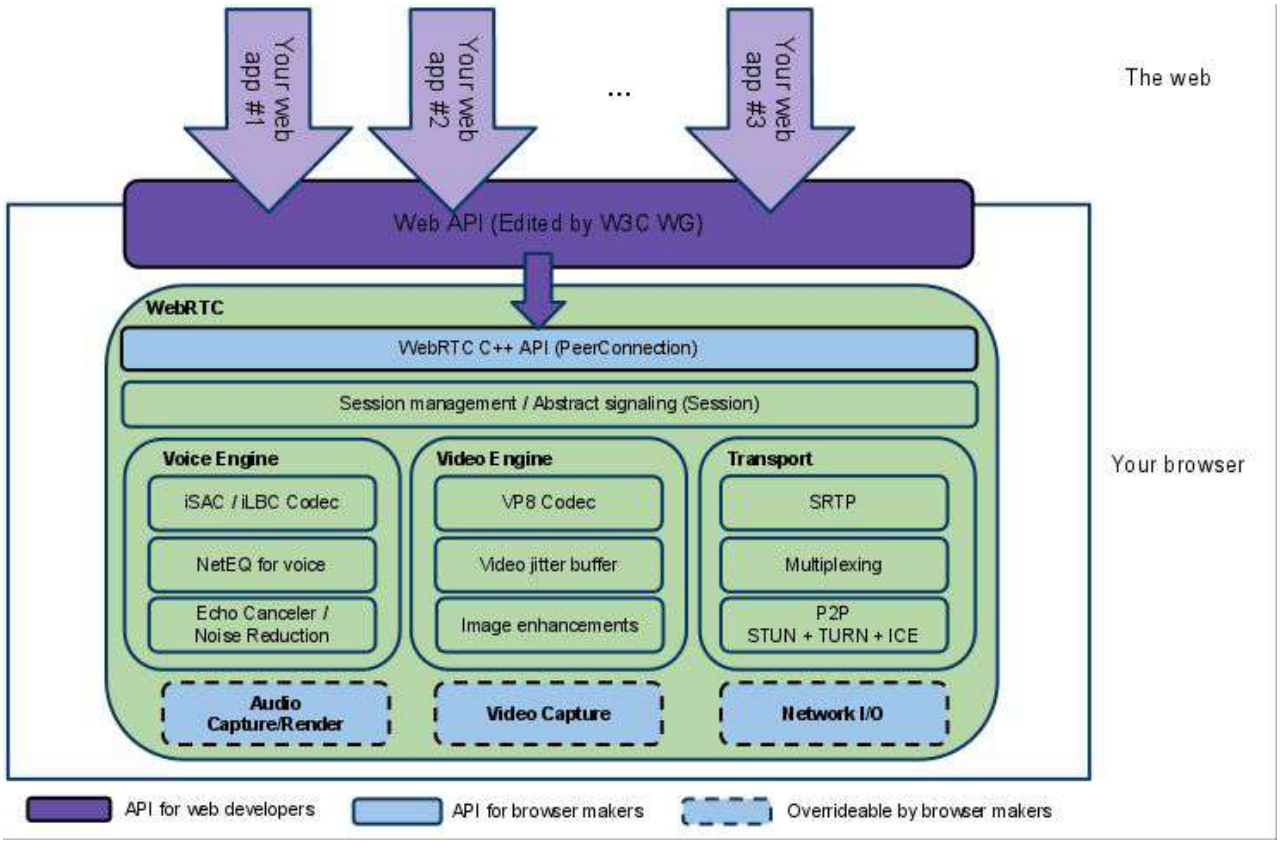
If the request is successful, we will obtain a MediaStream object as the one represented in the picture below. All the MediaStreamTracks inside a MediaStream object are automatically synchronized.



RTCPeerConnection

We use the RTCPeerConnection API for establishing peer to peer connections between users. This API is specifically designed for establishing audio and video conferences. It is almost transparent for the programmer. Some of their built-in duties are :

- Connecting to remote peers using NAT-traversal technologies such as ICE, STUN, and TURN.
- Managing the media engines (codecs, echo cancellation, noise reduction...).
- Sending the locally-produced streams to remote peers and receiving streams from remote peers.
- Sending arbitrary data directly to remote peers.
- Taking care of the security, using the most appropriate secure protocol for each of the WebRTC tasks. It uses HTTPS for the signaling, Secure RTP for the media and DTLs for data channel.



WebRTC architecture (image taken from webrtc.org)

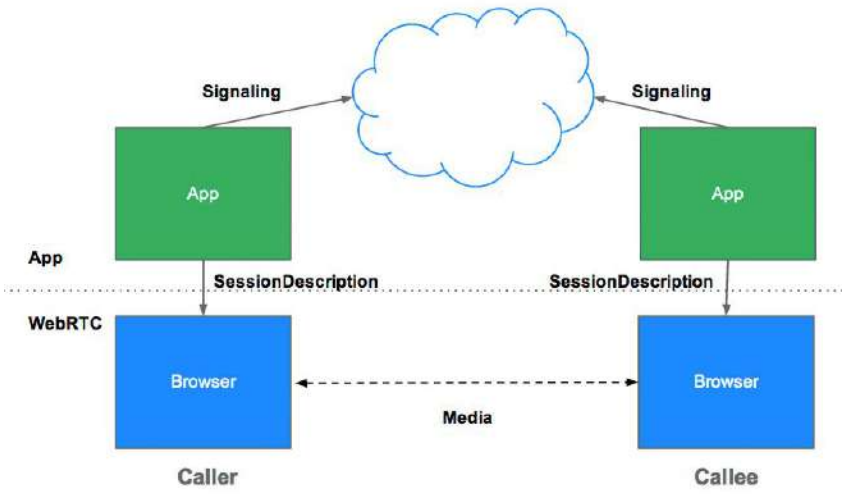
Signaling: session control, network and media information

WebRTC uses RTCPeerConnection to communicate streaming data between browsers (aka peers), but also needs a mechanism to coordinate communication and to send control messages, a process known as signaling. Signaling methods and protocols are not specified by WebRTC: signaling is not part of the RTCPeerConnection API. Instead, WebRTC app developers can choose whatever messaging protocol they prefer, such as SIP or XMPP, and any appropriate duplex (two-way) communication channel.

Signaling is used to exchange three types of information:

- Session control messages: to initialize or close communication and report errors.
- Network configuration: to the outside world, what's my computer's IP address and port?
- Media capabilities: what codecs and resolutions can be handled by my browser and the browser it wants to communicate with?

The exchange of information via signaling must have completed successfully before peer-to-peer streaming can begin.



Once the signaling process has completed successfully, data can be streamed directly peer to peer, between the caller and callee—or if that fails, via an intermediary relay server. Streaming is the job of RTCPeerConnection.

Is WebRTC serverless ?

In the real world, WebRTC needs servers, however simple, so the following can happen:

- Users discover each other and exchange 'real world' details such as names.
- WebRTC client applications (peers) exchange network information.
- Peers exchange data about media such as video format and resolution.
- WebRTC client applications traverse NAT gateways and firewalls.

In other words, WebRTC needs four types of server-side functionality:

- User discovery and communication.
- Signaling.
- NAT/firewall traversal.
- Relay servers in case peer-to-peer communication fails.

NAT traversal, peer-to-peer networking, and the requirements for building a server app for user discovery and signaling, are beyond the scope of this article. Suffice to say that the STUN protocol and its extension TURN are used by the ICE framework to enable RTCPeerConnection to cope with NAT traversal and other network vagaries.

ICE / STUN / TURN

ICE, STUN and TURN are different mechanism for obtaining possible addresses where a peer can contact another peer. As stated in the RFC 5245[12], ICE is an extension to the offer/answer model, and works by including a multiplicity of IP addresses and ports in SDP offers and answers, which are then tested for connectivity by peer-to-peer connectivity checks. The IP addresses and ports included in the SDP and the connectivity checks are performed using the Session Traversal Utilities for NAT (STUN) protocol and its extension, Traversal Using Relay NAT (TURN). ICE can be used by any protocol utilizing the offer/answer model, such as the Session Initiation Protocol (SIP).

While STUN works most of the times, in some very difficult situations TURN is the only option. TURN enables the communication between two users that can't find each other because of NATs by relaying their media. This is very expensive in system resources. In addition, there are some security flaws.

STUN Servers

Session Traversal Utilities for NAT (STUN) :
simple protocol for discovering the server-reflexive address.

– Client: Where do you see me at?
– Server: I see you at 206.123.31.67:55123.

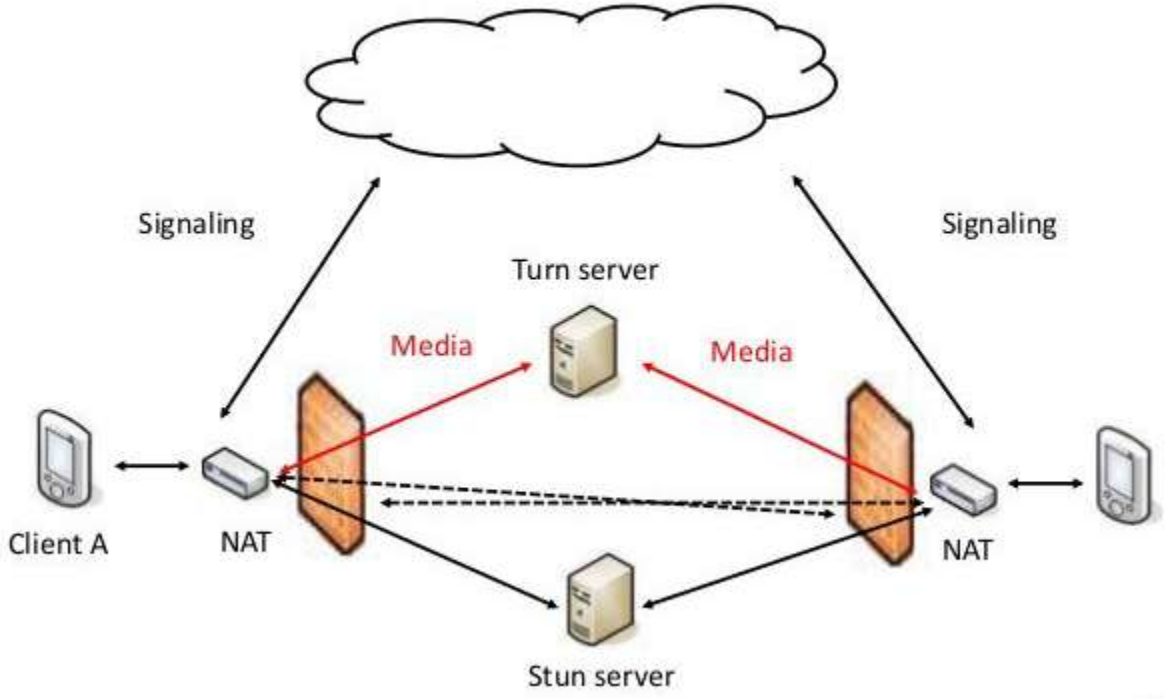
A STUN server is only used during the connection setup and once that session has been established, media will flow directly between clients.

TURN Servers

If a STUN server cannot establish the connection, TURN (Traversal Using Relays around NAT) comes into picture.

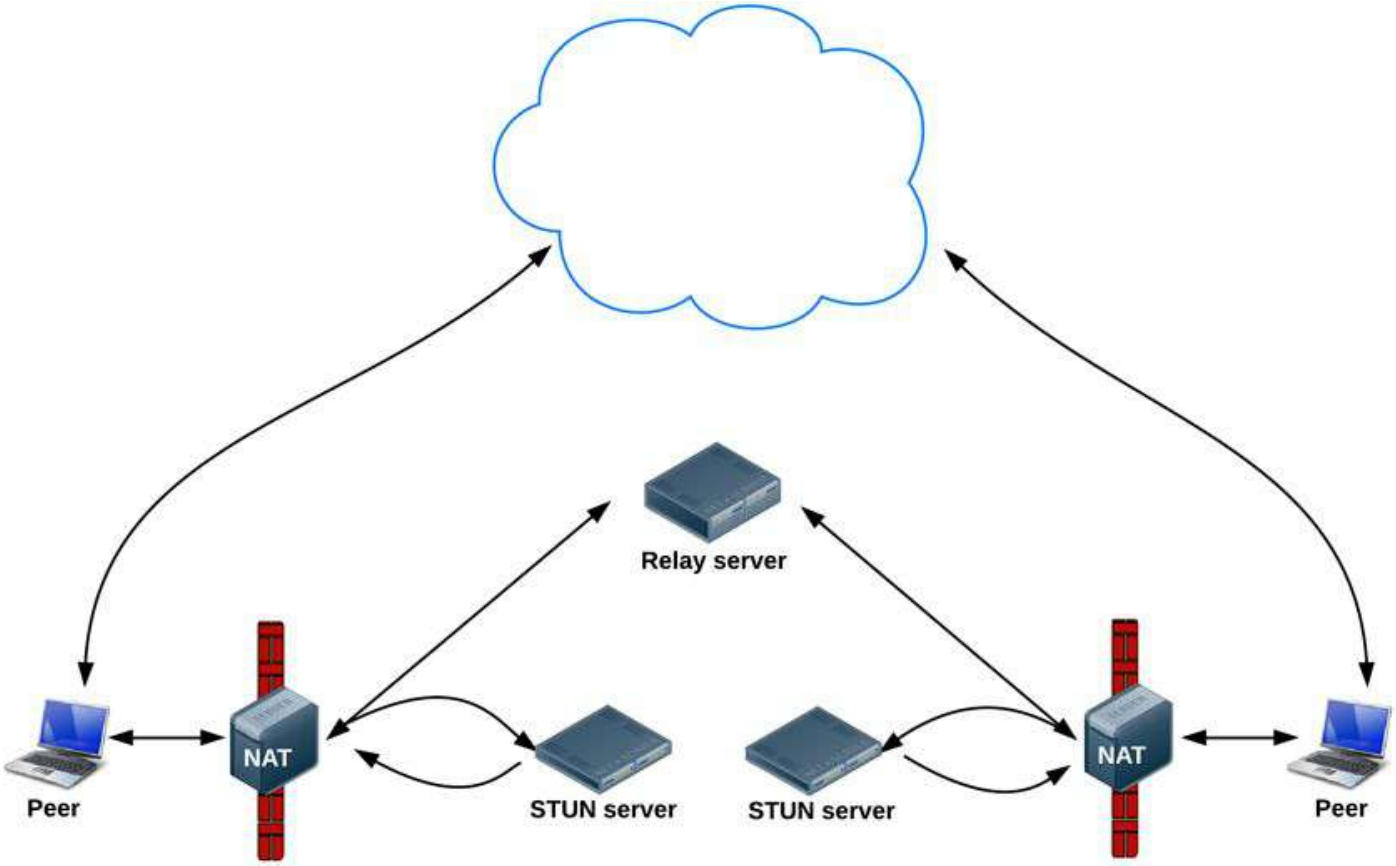
TURN Server allocates some of its resources to the client. The client then uses the TURN server for the media flow. It acts as a relay between different peers. A TURN server provides a relayed address to the clients which is then used by the clients for communication.

Unlike STUN, TURN remains in between the peers throughout the session.



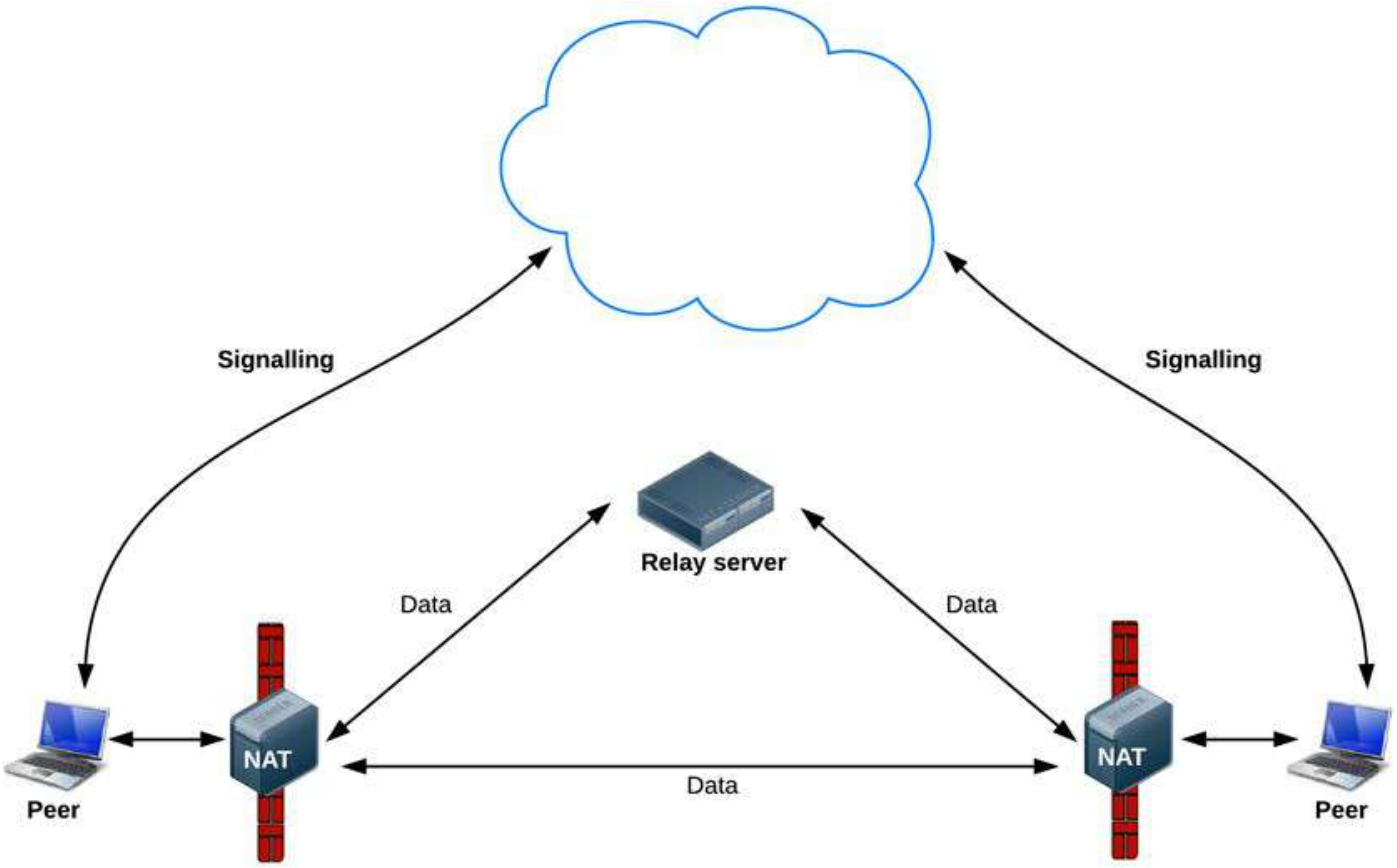
ICE
Interactive Connectivity Establishment

ICE is a framework for connecting peers, such as two video chat clients. Initially, ICE tries to connect peers directly, with the lowest possible latency, via UDP. In this process, STUN servers have a single task: to enable a peer behind a NAT to find out its public address and port.



Finding connection candidates

If UDP fails, ICE tries TCP: first HTTP, then HTTPS. If direct connection fails—in particular, because of enterprise NAT traversal and firewalls—ICE uses an intermediary (relay) TURN server. In other words, ICE will first use STUN with UDP to directly connect peers and, if that fails, will fall back to a TURN relay server. The expression 'finding candidates' refers to the process of finding network interfaces and ports.



WebRTC data pathways

RTCDataChannel

As well as audio and video, WebRTC supports real-time communication for other types of data.

The RTCDataChannel API enables peer-to-peer exchange of arbitrary data, with low latency and high throughput.

There are many potential use cases for the API, including:

- Gaming
- Remote desktop applications
- Real-time text chat
- File transfer
- Decentralized networks

The API has several features to make the most of RTCPeerConnection and enable powerful and flexible peer-to-peer communication:

- Leveraging of RTCPeerConnection session setup.
- Multiple simultaneous channels, with prioritization.
- Reliable and unreliable delivery semantics.
- Built-in security (DTLS) and congestion control.
- Ability to use with or without audio or video.

Communication occurs directly between browsers, so RTCDataChannel can be much faster than WebSocket even if a relay (TURN) server is required when 'hole punching' to cope with firewalls and NATs fails.

SECURITY

There are several ways a real-time communication application or plugin might compromise security. For example:

- Unencrypted media or data might be intercepted en route between browsers, or between a browser and a server.
- An application might record and distribute video or audio without the user knowing.
- Malware or viruses might be installed alongside an apparently innocuous plugin or application.

WebRTC has several features to avoid these problems:

- WebRTC implementations use secure protocols such as DTLS and SRTP.
- Encryption is mandatory for all WebRTC components, including signaling mechanisms.
- WebRTC is not a plugin: its components run in the browser sandbox and not in a separate process, components do not require separate installation, and are updated whenever the browser is updated.
- Camera and microphone access must be granted explicitly and, when the camera or microphone are running, this is clearly shown by the user interface.

CONCLUSION

The APIs and standards of WebRTC can democratize and decentralize tools for content creation and communication—for telephony, gaming, video production, music making, news gathering and many other applications.

We look forward to what JavaScript developers make of WebRTC as it becomes widely implemented.

> *'Potentially, WebRTC and HTML5 could enable the same transformation for real-time communications that the original browser did for information.'*
>
> Phil Edholm (Blogger)

WebRTC Frameworks

There are many JavaScript frameworks available which implement WebRTC and provide easy to use APIs to build WebRTC applications

- simpleWebRTC (https://simplewebrtc.com/)
- PeerJS (http://peerjs.com/)
- easyRTC (https://easyrtc.com/)

PeerJS

PeerJS simplifies WebRTC peer-to-peer data, video, and audio calls.

PeerJS wraps the browser's WebRTC implementation to provide a complete, configurable, and easy-to-use peer-to-peer connection API. Equipped with nothing but an ID, a peer can create a P2P data or media stream connection to a remote peer.

Setup

Include the library

```
<script src="http://cdn.peerjs.com/0.3/peer.js"></script>
```

Create a peer

Get a free API key. Your id only needs to be unique to the namespace of your API key.

```
var peer = new Peer('pick-an-id', {key: 'myapikey'});
// You can pick your own id or omit the id if you want to get a random one from the server.
```

Data connections

Connect

```
var conn = peer.connect('another-peers-id');
conn.on('open', function(){
  conn.send('hi!');
});
```

Receive

```
peer.on('connection', function(conn) {
  conn.on('data', function(data){
    // Will print 'hi!'
```

```
    console.log(data);
  });
});
```

Media calls

Call

```
var getUserMedia = navigator.getUserMedia || navigator.webkitGetUserMedia || navigator.mozGetUserMedia;
getUserMedia({video: true, audio: true}, function(stream) {
  var call = peer.call('another-peers-id', stream);
  call.on('stream', function(remoteStream) {
    // Show stream in some video/canvas element.
  });
}, function(err) {
  console.log('Failed to get local stream' ,err);
});
```

Answer

```
var getUserMedia = navigator.getUserMedia || navigator.webkitGetUserMedia || navigator.mozGetUserMedia;
peer.on('call', function(call) {
  getUserMedia({video: true, audio: true}, function(stream) {
    call.answer(stream); // Answer the call with an A/V stream.
    call.on('stream', function(remoteStream) {
      // Show stream in some video/canvas element.
    });
  }, function(err) {
    console.log('Failed to get local stream' ,err);
  });
});
```

PeerServer

To broker connections, PeerJS connects to a PeerServer. Note that no peer-to-peer data goes through the server; The server acts only as a connection broker.

PeerServer Cloud

If you don't want to run your own PeerServer, PeerJS offer a free cloud-hosted version of PeerServer. Start by getting a PeerServer Cloud API Key!

Run your own

PeerServer is open source and is written in node.js. You can easily run your own.

Use of STUN/TURN Servers

When creating your Peer object, pass in the ICE servers as the config key of the options hash.

```
var peer = new Peer({
  config: {'iceServers': [
    { url: 'stun:stun.l.google.com:19302' },
    { url: 'turn:homeo@turn.bistri.com:80', credential: 'homeo' }
  ]}
});
```

HxWebRTC (Application made using PeerJS)

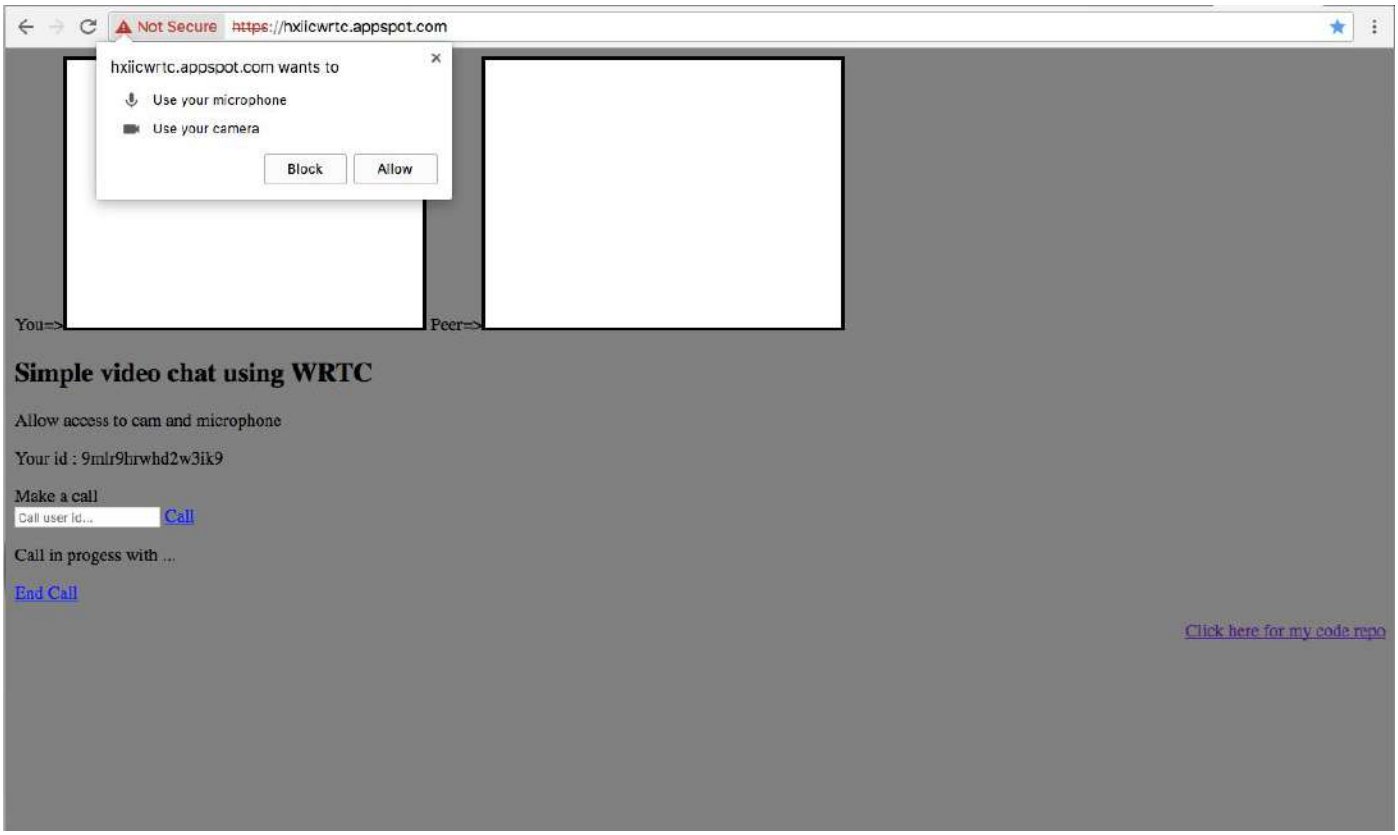
A simple two way video chat using WebRTC

The app is implemented using peerjs javascript library and google stun servers and numb.viagenie.ca turn server.

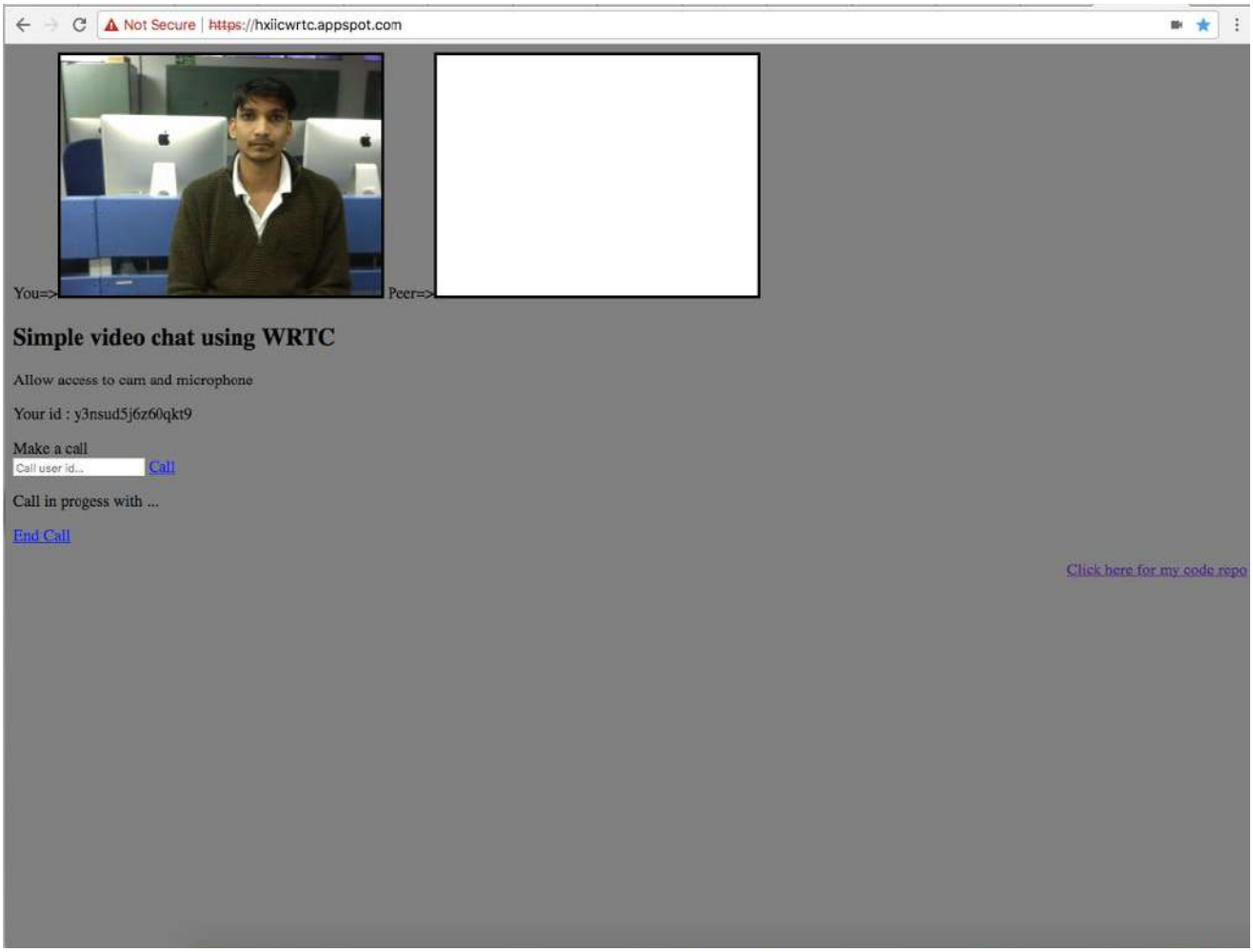
It is a full duplex video and audio communication web application which is deployed on google cloud server (google app engine) and can be found at :

https://hxiicwrtc.appspot.com

The home page of the application looks like this :

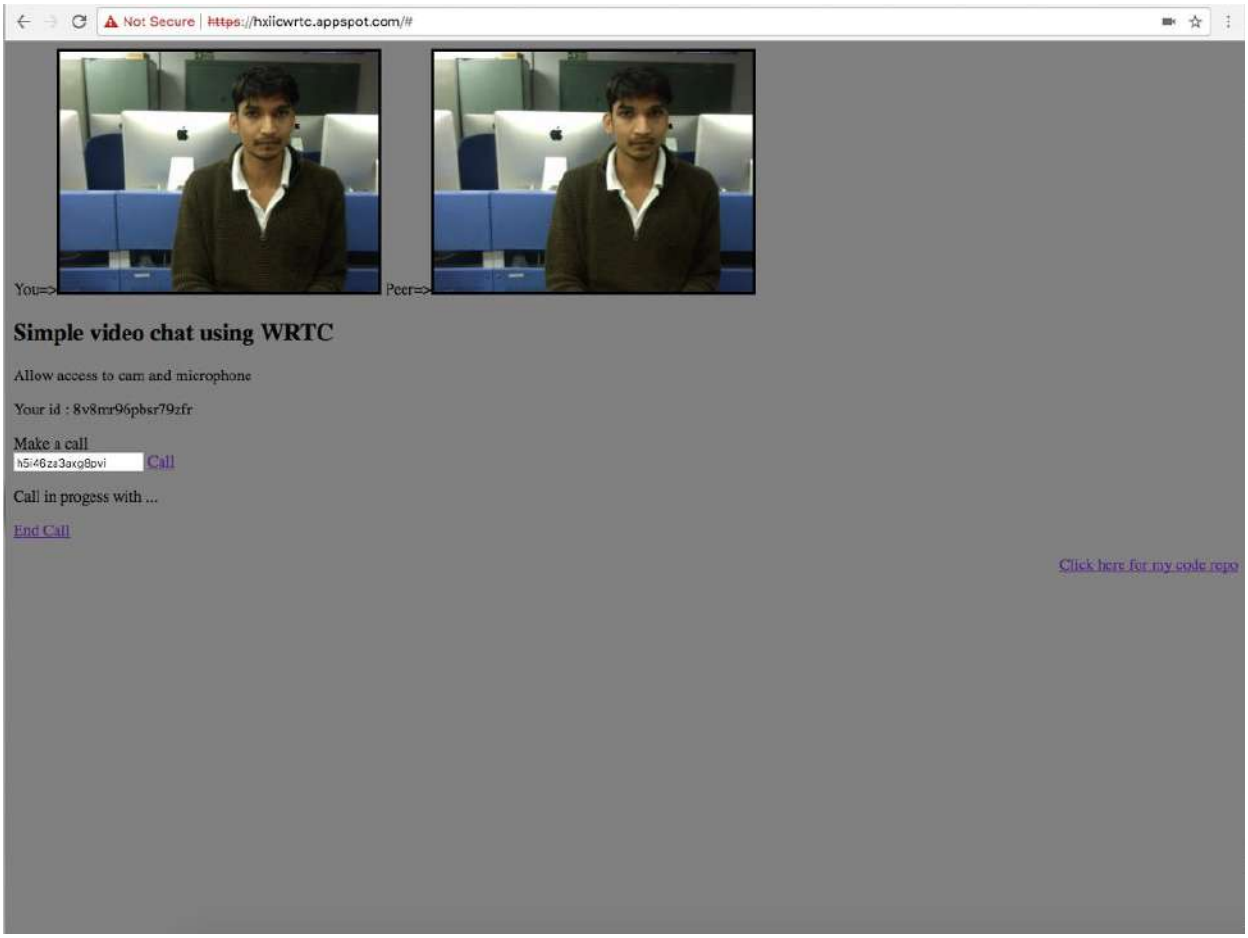


User have to explicitly allow access to the browser to use camera and microphone.



The code generates your id through which other peers can call you as soon as you go on the home page. Then enter the id of the person you want to call and after clicking on the call button, you are connected with the other peer.

For testing purpose, i am acting as both the peers in the screenshots. I am connecting between two different tabs in the browsers.



The shortcoming of this application is that it can only connect two peers at a time in one call.

The application's source code can be found at :

http://10.107.88.100/hiteyadav/HxWebRTC-firstapp

https://github.com/hitex-loco/hxwebrtc

simpleWebRTC

This is how simpleWebRTC works

A dab of HTML

```
<!DOCTYPE html>
<html>
   <head>
      <script src="https://simplewebrtc.com/latest-v2.js"></script>
   </head>
   <body>
      <video height="300" id="localVideo"></video>
      <div id="remotesVideos"></div>
   </body>
</html>
```

Stir in the WebRTC object

```
var webrtc = new SimpleWebRTC({
  // the id/element dom element that will hold "our" video
  localVideoEl: 'localVideo',
  // the id/element dom element that will hold remote videos
  remoteVideosEl: 'remotesVideos',
  // immediately ask for camera access
  autoRequestMedia: true
});
```

And join when ready

```
// we have to wait until it's ready
webrtc.on('readyToCall', function () {
  // you can name it anything
  webrtc.joinRoom('your awesome room name');
});
```

Signaling server used

SimpleWebRTC uses the SimpleWebRTC.com sandbox server and is only for development and testing purposes. This server does not provide media relay facilities so there might be connectivity issues.

The signaling server is open source (MIT) licensed as well. You can find it here: github.com/andyet/signalmaster.

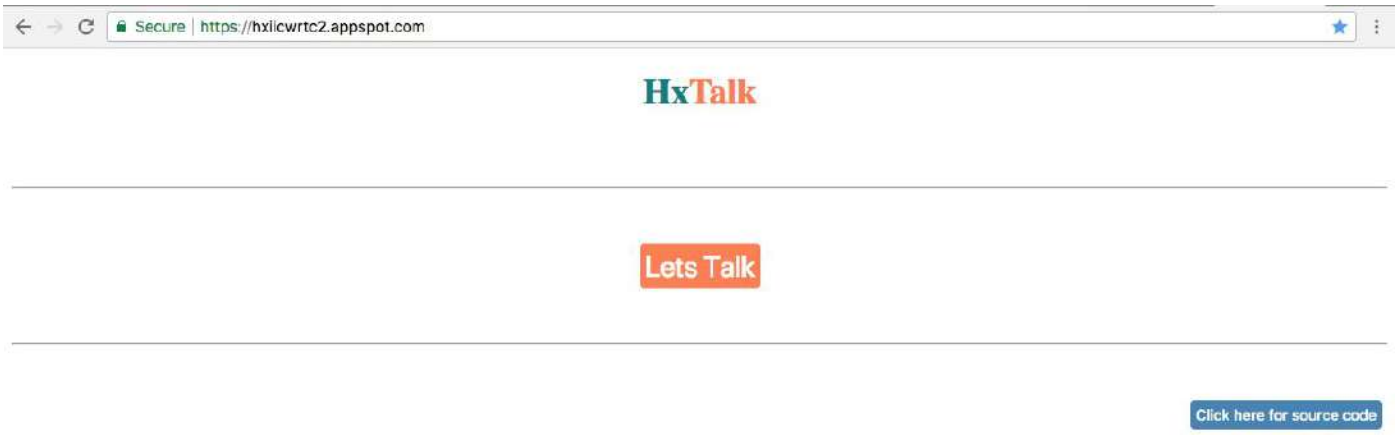
HxTalk (Application made using simpleWebRTC)

A simple multi party video chat application built using simpleWebRTC JavaScript library.

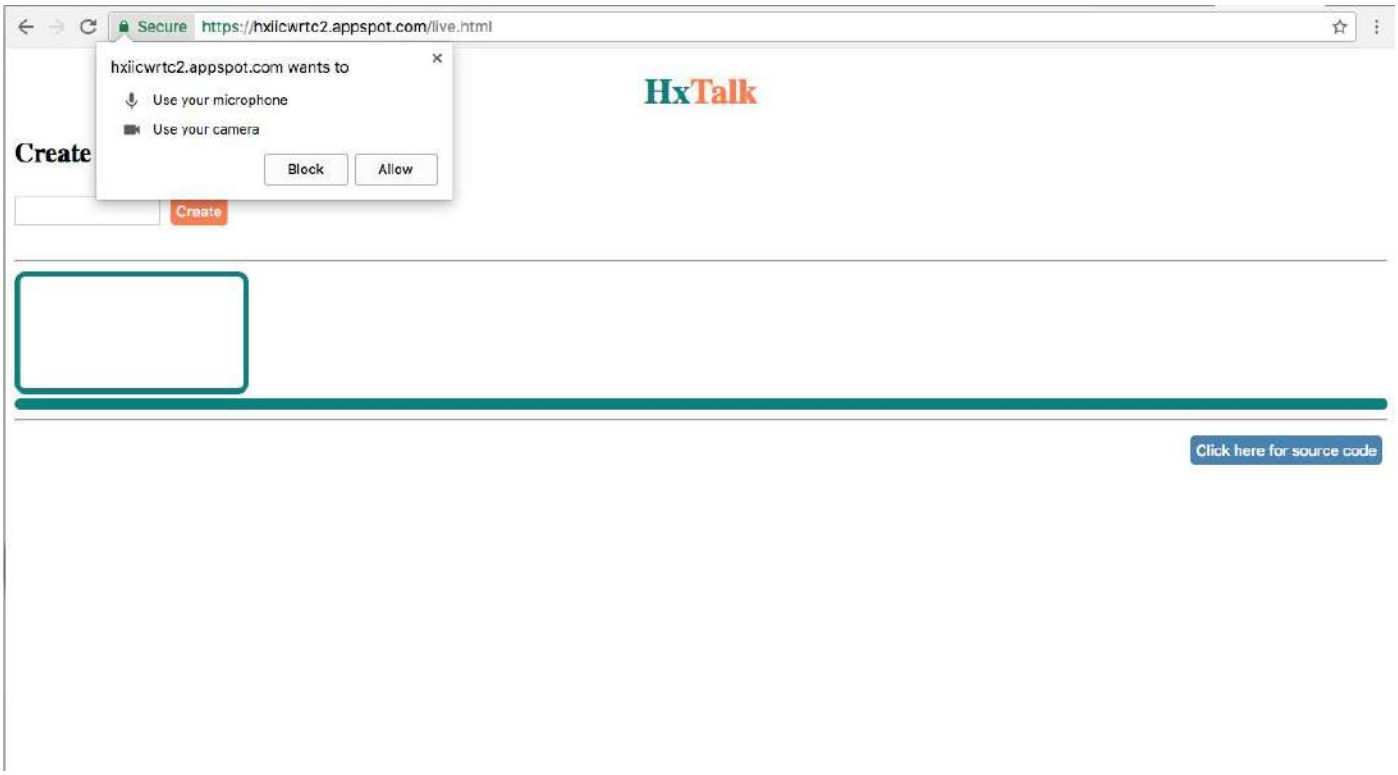
It is also a full duplex video and audio communication web application which overcomes the shortcoming of my previous application as it a supports multi party. In this application more than two peers can connect in the same call and this is also deployed on google cloud server(app engine) and can be found at :

https://hxiicwrtc2.appspot.com

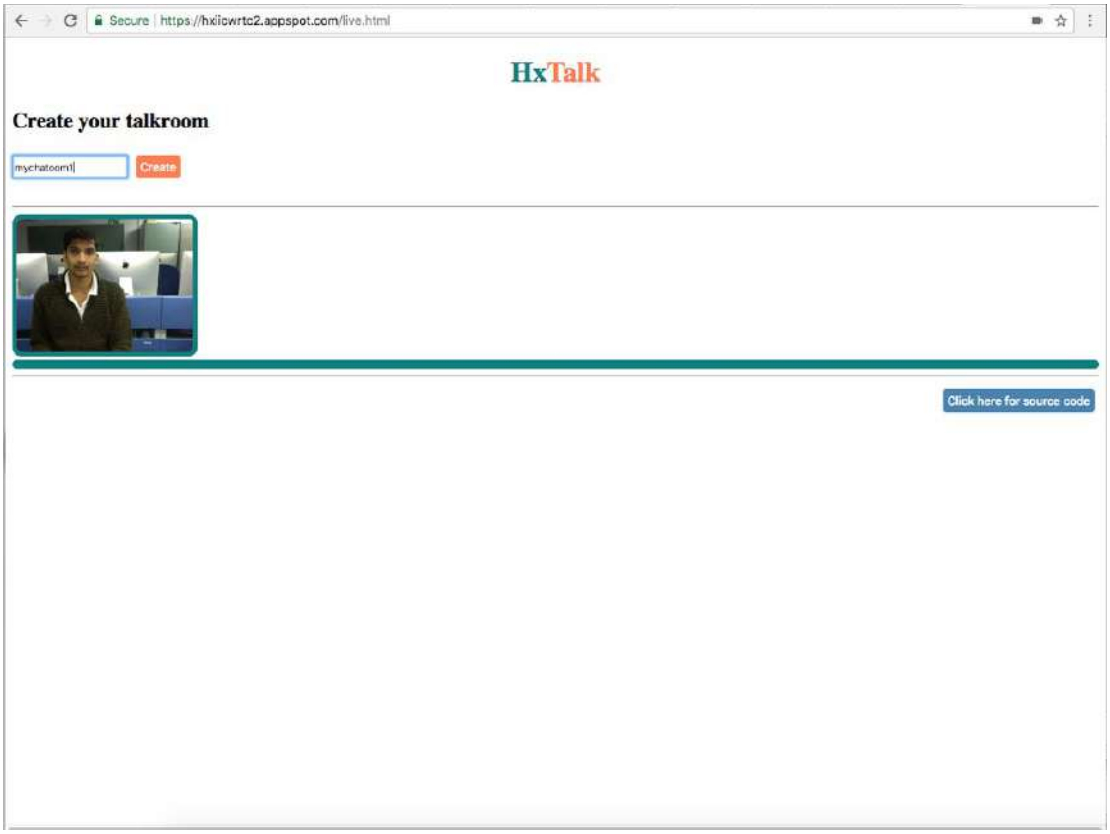
The home page of the application looks like this :

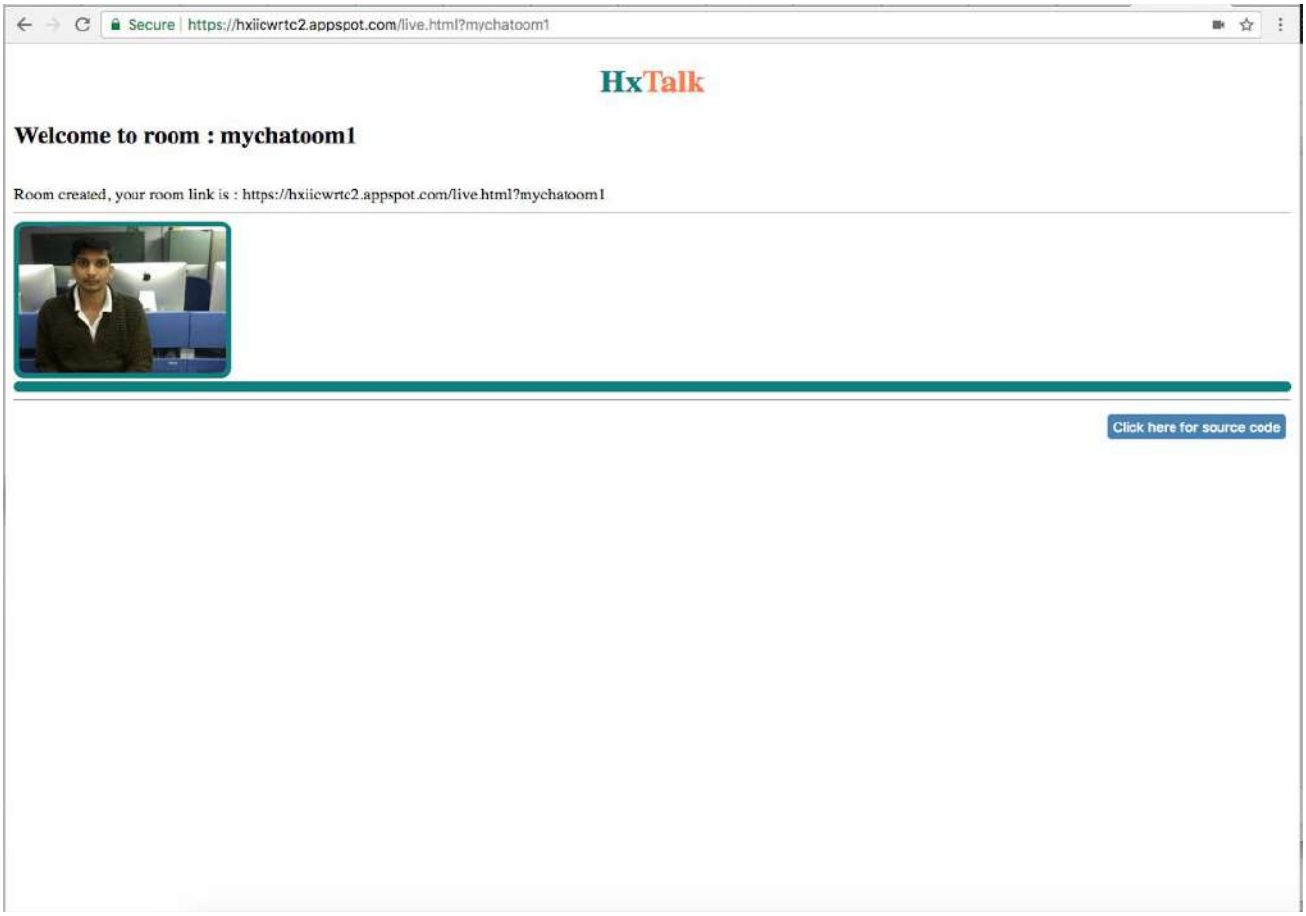


Here, we have to create our chatroom and then we can send our friends the link of our chatroom so that they can join in the chatroom.

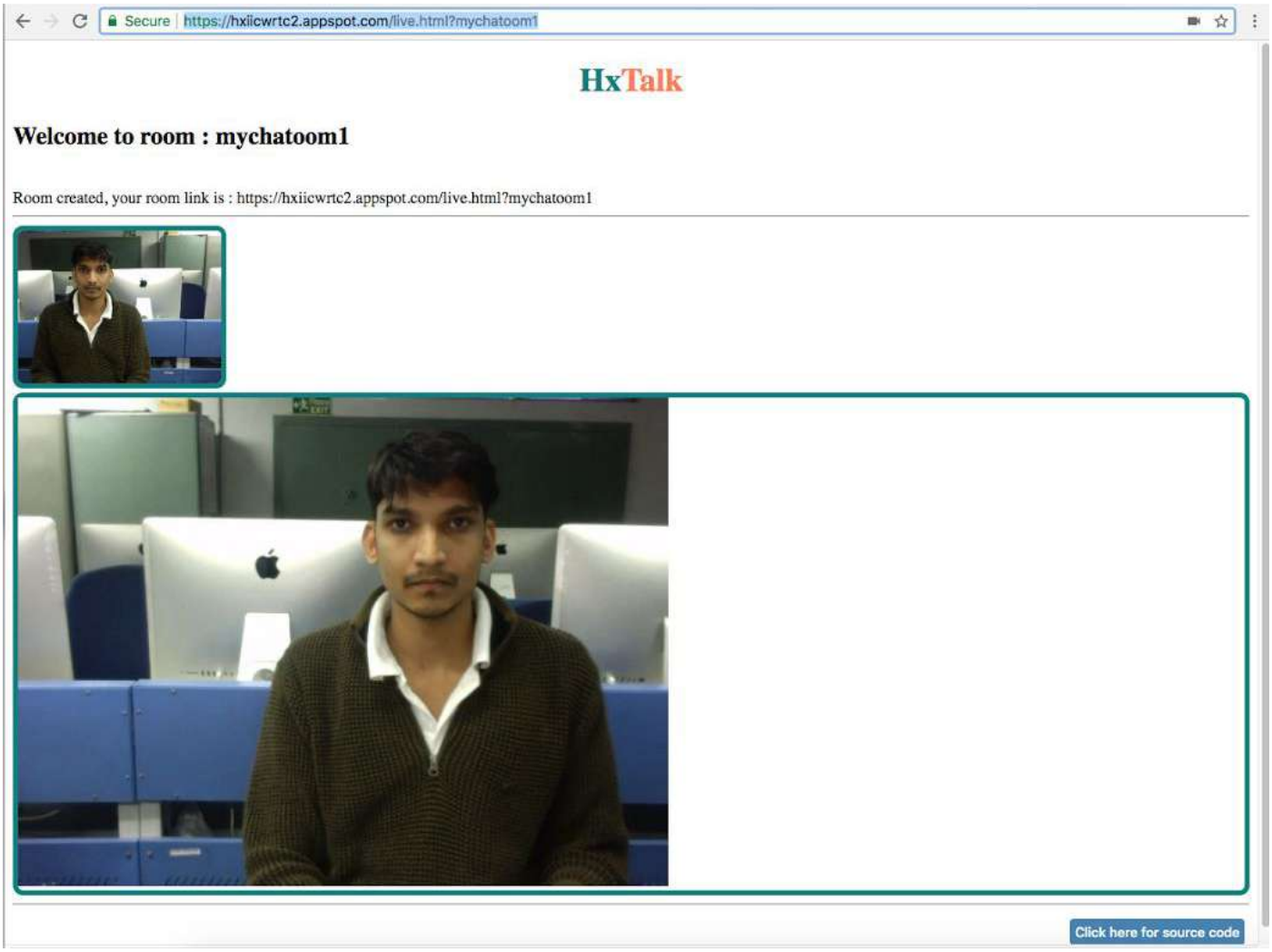


Access to camera and microphone has to be provided explicitly.





When other peers join the chat room, their corresponding video elements are added accordingly.



For testing purpose, i am acting as both the peers in the screenshots. I am connecting between two different tabs in the browsers.

The application's source code can be found at :

http://10.107.88.100/hiteyadav/HxTalk

https://github.com/hitex-loco/HxTalk

REFERENCES

https://webrtc.org/

https://codelabs.developers.google.com/codelabs/webrtc-web/

http://io13webrtc.appspot.com/

https://www.html5rocks.com/en/tutorials/webrtc/basics/

http://peerjs.com/

https://simplewebrtc.com/

https://en.wikipedia.org/wiki/WebRTC

Also, some WebRTC related documentation made by me and this report can be found on my github repository :

https://github.com/hitex-loco/WebRTC_Docs