**VELAGAPUDI RAMAKRISHNA SIDDHARTHA ENGINEERING COLLEGE**

**(AUTONOMOUS - AFFILIATED TO JNTU-K, KAKINADA)**

**Approved by AICTE & Accredited by NBA**

**KANURU, VIJAYAWADA-7**

**ACADEMIC YEAR**

**(2022-2023)**



**20IT5301 – Computer Networks**

**Project**

Submitted to: Submitted by:

K. Pranathi M . Venkata Sri Sai Surya(208W1A1228)

IT Department P. Uday Manikanta(208W1A1241)

VRSEC P. Chaitanya Swaroop(208W1A1243)

**AIM:**

Implement Video Chat system (Look for WebRTC)

**PROBLEM:**

Our primary goal is to create a video chat with the people who are connected to local host i.e WebRTC

**DESCRIPTION:**

WebRTC (Web Real Time Communications) is an open source communication technology for mobile and desktop platforms. The technology is built on APIs that require no plugins, and, since the first stable release in 2018, WebRTC has gathered support from all major web browsers and operating systems.

**Socket.IO:**

Socket.IO is a library that enables real-time, bidirectional and event-based communication between the browser and the server. It consists of:

* a Node.js server.
* a Javascript client library for the browser.

### PROCEDURE:

### Step 1. Setting up the Project

Let’s begin with setting up the project.

### ****Step 1.1. Download Node.Js****

* You can download Node.Js for your platform by clicking on [this](https://nodejs.org/en/download/) link. Downloading Node.Js will automatically install NPM (**Node Package Manager)**on your PC. NPM is the default Package Manager for Node.Js

**Step 1.2. Create a node project**

* Create a New Folder. This folder will be the **root directory** for our project.
* Open terminal/CMD in this folder and run the command npm init .
* Press the Enter Key continuosly to skip the additional configurations for the project and write YES when prompted.
* This will create a file package.json in the root directory of the project. This file will contain all the necessary information regarding our project like project dependencies.

**Step 1.3. Installing dependencies**

* In the terminal, run the following command. It will install the dependencies — Express.JS and socket.IO in our project.

npm install express@4.17.1 socket.io@1.2.0 --save

* The flag --save will save the name and versions of these dependencies in **package.json** for future reference.
* After the above command has finished execution, you will see a folder node\_modules created in the root directory of the project. This folder contains the dependencies that we have just installed.

Now we have finished setting up the project. The following is the project structure at this stage.

# Step 2. Creating The BackEnd

Let us now begin writing the code for the backend. Before we begin, let’s revise a few points from the previous tutorial.

* We need a backend server for signaling.
* Certain information — Candidate (network) information & Media Codecs must be exchanged between the two peers before a direct connection can be made between them using WebRTC.
* Signaling refers to the mechanism using which two peers exchange this information

The above points tell us that we have to implement a mechanism using which two clients (browsers) can send messages to each other. We will use Socket.IO for this purpose. Socket.IO is suited to learning about WebRTC signaling because of its built-in concept of ‘rooms’. Let’s first discuss what is Socket.IO

**Socket.IO**

* Socket.IO consists of two parts— client Library& server Library. Obviously, the client library is used on the client-side & server library is used on the server-side.
* Socket.IO helps in implementing the following — Let’s say four clients are connected to the server. When the server receives a new message from one client, it should notify all the other clients and also forward this message to the other client. It is similar to a group chat.
* In Socket.IO, each message, that is sent to the server or received from the server, is associated with an event. So, if a client sends a message to the server on a particular event, the server will forward this message to only those clients that are listening to this corresponding event.
* There are some reserved events. However, we can also define custom events. To know about the reserved events, you can visit this link.
* Also, the clients can join a room and ask the server to send the message to only those clients that have joined a particular room.

Now that we have discussed Socket.IO, we can begin implementing the backend server

**Step 2.1. Create a file index.js**

* In the Express framework, index.js is the starting point for our server by default. So create a file index.js at the root level of our project.

**Step 2.2. Create a Public folder and a views folder**

* Create the following folders at the root level of our project  
  \* public — contains the static files like CSS and JS files for the frontend  
  \* views — contains the views for the frontend
* Our website will only contain one page. Inside the views folder, create a file index.ejs that will contain the HTML code for the frontend. Expresses uses ejs as the templating engine.
* The project structure will now look like the following

**Step 2.3. Initialize Express and an HTTP Server**

* Now, we must initialize Express, HTTP server, and Socket.IO for our backend. To do this, paste the following code in the index.js located at the root level of the project

### ****Step 2.3. Implement Socket.IO****

* Now, it is time to implement Socket.IO in the backend.
* So, now we have implemented the backend of our website. The following is the complete code of index.js

**CODE(Index.js):**

'use strict';

//Loading dependencies & initializing express

var os = require('os');

var express = require('express');

var app = express();

var http = require('http');

//For signalling in WebRTC

var socketIO = require('socket.io');

app.use(express.static('public'))

app.get("/", function(req, res){

res.render("index.ejs");

});

var server = http.createServer(app);

server.listen(process.env.PORT || 8000);

var io = socketIO(server);

io.sockets.on('connection', function(socket) {

// Convenience function to log server messages on the client.

// Arguments is an array like object which contains all the arguments of log().

// To push all the arguments of log() in array, we have to use apply().

function log() {

var array = ['Message from server:'];

array.push.apply(array, arguments);

socket.emit('log', array);

}

//Defining Socket Connections

socket.on('message', function(message, room) {

log('Client said: ', message);

// for a real app, would be room-only (not broadcast)

socket.in(room).emit('message', message, room);

});

socket.on('create or join', function(room) {

log('Received request to create or join room ' + room);

var clientsInRoom = io.sockets.adapter.rooms[room];

var numClients = clientsInRoom ? Object.keys(clientsInRoom.sockets).length : 0;

log('Room ' + room + ' now has ' + numClients + ' client(s)');

if (numClients === 0) {

socket.join(room);

log('Client ID ' + socket.id + ' created room ' + room);

socket.emit('created', room, socket.id);

} else if (numClients === 1) {

log('Client ID ' + socket.id + ' joined room ' + room);

io.sockets.in(room).emit('join', room);

socket.join(room);

socket.emit('joined', room, socket.id);

io.sockets.in(room).emit('ready');

} else { // max two clients

socket.emit('full', room);

}

});

socket.on('ipaddr', function() {

var ifaces = os.networkInterfaces();

for (var dev in ifaces) {

ifaces[dev].forEach(function(details) {

if (details.family === 'IPv4' && details.address !== '127.0.0.1') {

socket.emit('ipaddr', details.address);

}

});

}

});

socket.on('bye', function(){

console.log('received bye');

});

});

**Step 3. Creating the FrontEnd of our website**

* Now, let’s create the frontend of our website

**Step 3.1. Create the HTML file**

• Let’s create the HTML file for our frontend.

• We will define the CSS and Javascript for the front-end in public/css/styles.css and public/js/main.js respectively. Hence, we must import those files. In the backend, we explicitly set public it as the default directory for serving static files. Hence, we will import the files from css/styles.css & js/main.js in HTML.

• We will also import the client library for socket.io

• We will also import adapter.js for WebRTC because implementations of WebRTC are still evolving, and because each browser has different levels of support for codecs and WebRTC features. The adapter is a JavaScript shim that lets your code be written to the specification so that it will “just work” in all browsers with WebRTC support.

• We discussed STURN/TURN servers in the previous tutorials. We will import the TURN/STUN URLs from public/js/config.js . We will create this file later in this tutorial.

**CODE:**

<!DOCTYPE html>

<html>

<head>

<title>WebTutsPlus WebCon</title>

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<!-- Import Google Fonts -->

<link href="https://fonts.googleapis.com/css2?family=Baloo+Tamma+2:wght@400;500;600&family=Josefin+Slab&display=swap" rel="stylesheet">

<!-- Import CSS file -->

<link rel="stylesheet" href="/css/styles.css">

<!-- <link rel="stylesheet" href="/css/main.css" /> -->

</head>

<body class="h-100">

<div class="h-100" id="video\_display">

<div id ="video\_container" class="align-items-center" style="margin-top: 10%;">

<div class="local\_div" id="div1" style="">

<!-- For playing local video -->

<video id="localVideo" class="" autoplay muted playsinline></video>

</div>

<div class="remote\_div" id="div2">

<!-- For playing local audio -->

<video id="remoteVideo" class="" autoplay playsinline></video>

</div>

</div>

</div>

<!-- Import SocketIO for signalling -->

<script src="/socket.io/socket.io.js"></script>

<!-- Import WebRTC adapter for compatibility with all the browsers -->

<script src="https://webrtc.github.io/adapter/adapter-latest.js"></script>

<!-- Import TURN config -->

<script src="js\config.js"></script>

<!-- Import script containing WebRTC related functions -->

<script src="js\main.js"></script>

</body>

</html>

Step 3.2. Add the CSS code

**Step 3.3. Add the JS file**

• Now, let’s add javascript to our frontend. We had already the file public/js/main.js in index.ejs . It is in this file, we will implement the various methods for using WebRTC and client library of Socket.IO

• A lot of messages will be exchanged between the two clients before a direct connection is created between them. We saw this in details in the previous tutorial when we gave the example of Amy and Bernadette. It is highly recommended that you read that example. We have simply implemented each step mentioned in that article using Socket.IO

|  |
| --- |
| 'use strict'; |
|  |  |
|  | //Defining some global utility variables |
|  | var isChannelReady = false; |
|  | var isInitiator = false; |
|  | var isStarted = false; |
|  | var localStream; |
|  | var pc; |
|  | var remoteStream; |
|  | var turnReady; |
|  |  |
|  | //Initialize turn/stun server here |
|  | //turnconfig will be defined in public/js/config.js |
|  | var pcConfig = turnConfig; |
|  |  |
|  | //Set local stream constraints |
|  | var localStreamConstraints = { |
|  | audio: true, |
|  | video: true |
|  | }; |
|  |  |
|  |  |
|  | // Prompting for room name: |
|  | var room = prompt('Enter room name:'); |
|  |  |
|  | //Initializing socket.io |
|  | var socket = io.connect(); |
|  |  |
|  | //Ask server to add in the room if room name is provided by the user |
|  | if (room !== '') { |
|  | socket.emit('create or join', room); |
|  | console.log('Attempted to create or join room', room); |
|  | } |
|  |  |
|  | //Defining socket events |
|  |  |
|  | //Event - Client has created the room i.e. is the first member of the room |
|  | socket.on('created', function(room) { |
|  | console.log('Created room ' + room); |
|  | isInitiator = true; |
|  | }); |
|  |  |
|  | //Event - Room is full |
|  | socket.on('full', function(room) { |
|  | console.log('Room ' + room + ' is full'); |
|  | }); |
|  |  |
|  | //Event - Another client tries to join room |
|  | socket.on('join', function (room){ |
|  | console.log('Another peer made a request to join room ' + room); |
|  | console.log('This peer is the initiator of room ' + room + '!'); |
|  | isChannelReady = true; |
|  | }); |
|  |  |
|  | //Event - Client has joined the room |
|  | socket.on('joined', function(room) { |
|  | console.log('joined: ' + room); |
|  | isChannelReady = true; |
|  | }); |
|  |  |
|  | //Event - server asks to log a message |
|  | socket.on('log', function(array) { |
|  | console.log.apply(console, array); |
|  | }); |
|  |  |
|  |  |
|  | //Event - for sending meta for establishing a direct connection using WebRTC |
|  | //The Driver code |
|  | socket.on('message', function(message, room) { |
|  | console.log('Client received message:', message, room); |
|  | if (message === 'got user media') { |
|  | maybeStart(); |
|  | } else if (message.type === 'offer') { |
|  | if (!isInitiator && !isStarted) { |
|  | maybeStart(); |
|  | } |
|  | pc.setRemoteDescription(new RTCSessionDescription(message)); |
|  | doAnswer(); |
|  | } else if (message.type === 'answer' && isStarted) { |
|  | pc.setRemoteDescription(new RTCSessionDescription(message)); |
|  | } else if (message.type === 'candidate' && isStarted) { |
|  | var candidate = new RTCIceCandidate({ |
|  | sdpMLineIndex: message.label, |
|  | candidate: message.candidate |
|  | }); |
|  | pc.addIceCandidate(candidate); |
|  | } else if (message === 'bye' && isStarted) { |
|  | handleRemoteHangup(); |
|  | } |
|  | }); |
|  |  |
|  |  |
|  |  |
|  | //Function to send message in a room |
|  | function sendMessage(message, room) { |
|  | console.log('Client sending message: ', message, room); |
|  | socket.emit('message', message, room); |
|  | } |
|  |  |
|  |  |
|  |  |
|  | //Displaying Local Stream and Remote Stream on webpage |
|  | var localVideo = document.querySelector('#localVideo'); |
|  | var remoteVideo = document.querySelector('#remoteVideo'); |
|  | console.log("Going to find Local media"); |
|  | navigator.mediaDevices.getUserMedia(localStreamConstraints) |
|  | .then(gotStream) |
|  | .catch(function(e) { |
|  | alert('getUserMedia() error: ' + e.name); |
|  | }); |
|  |  |
|  | //If found local stream |
|  | function gotStream(stream) { |
|  | console.log('Adding local stream.'); |
|  | localStream = stream; |
|  | localVideo.srcObject = stream; |
|  | sendMessage('got user media', room); |
|  | if (isInitiator) { |
|  | maybeStart(); |
|  | } |
|  | } |
|  |  |
|  |  |
|  | console.log('Getting user media with constraints', localStreamConstraints); |
|  |  |
|  | //If initiator, create the peer connection |
|  | function maybeStart() { |
|  | console.log('>>>>>>> maybeStart() ', isStarted, localStream, isChannelReady); |
|  | if (!isStarted && typeof localStream !== 'undefined' && isChannelReady) { |
|  | console.log('>>>>>> creating peer connection'); |
|  | createPeerConnection(); |
|  | pc.addStream(localStream); |
|  | isStarted = true; |
|  | console.log('isInitiator', isInitiator); |
|  | if (isInitiator) { |
|  | doCall(); |
|  | } |
|  | } |
|  | } |
|  |  |
|  | //Sending bye if user closes the window |
|  | window.onbeforeunload = function() { |
|  | sendMessage('bye', room); |
|  | }; |
|  |  |
|  |  |
|  | //Creating peer connection |
|  | function createPeerConnection() { |
|  | try { |
|  | pc = new RTCPeerConnection(pcConfig); |
|  | pc.onicecandidate = handleIceCandidate; |
|  | pc.onaddstream = handleRemoteStreamAdded; |
|  | pc.onremovestream = handleRemoteStreamRemoved; |
|  | console.log('Created RTCPeerConnnection'); |
|  | } catch (e) { |
|  | console.log('Failed to create PeerConnection, exception: ' + e.message); |
|  | alert('Cannot create RTCPeerConnection object.'); |
|  | return; |
|  | } |
|  | } |
|  |  |
|  | //Function to handle Ice candidates generated by the browser |
|  | function handleIceCandidate(event) { |
|  | console.log('icecandidate event: ', event); |
|  | if (event.candidate) { |
|  | sendMessage({ |
|  | type: 'candidate', |
|  | label: event.candidate.sdpMLineIndex, |
|  | id: event.candidate.sdpMid, |
|  | candidate: event.candidate.candidate |
|  | }, room); |
|  | } else { |
|  | console.log('End of candidates.'); |
|  | } |
|  | } |
|  |  |
|  | function handleCreateOfferError(event) { |
|  | console.log('createOffer() error: ', event); |
|  | } |
|  |  |
|  | //Function to create offer |
|  | function doCall() { |
|  | console.log('Sending offer to peer'); |
|  | pc.createOffer(setLocalAndSendMessage, handleCreateOfferError); |
|  | } |
|  |  |
|  | //Function to create answer for the received offer |
|  | function doAnswer() { |
|  | console.log('Sending answer to peer.'); |
|  | pc.createAnswer().then( |
|  | setLocalAndSendMessage, |
|  | onCreateSessionDescriptionError |
|  | ); |
|  | } |
|  |  |
|  | //Function to set description of local media |
|  | function setLocalAndSendMessage(sessionDescription) { |
|  | pc.setLocalDescription(sessionDescription); |
|  | console.log('setLocalAndSendMessage sending message', sessionDescription); |
|  | sendMessage(sessionDescription, room); |
|  | } |
|  |  |
|  | function onCreateSessionDescriptionError(error) { |
|  | trace('Failed to create session description: ' + error.toString()); |
|  | } |
|  |  |
|  | //Function to play remote stream as soon as this client receives it |
|  | function handleRemoteStreamAdded(event) { |
|  | console.log('Remote stream added.'); |
|  | remoteStream = event.stream; |
|  | remoteVideo.srcObject = remoteStream; |
|  | } |
|  |  |
|  | function handleRemoteStreamRemoved(event) { |
|  | console.log('Remote stream removed. Event: ', event); |
|  | } |
|  |  |
|  | function hangup() { |
|  | console.log('Hanging up.'); |
|  | stop(); |
|  | sendMessage('bye',room); |

**Step 3.4. Add the STUN/TURN URLs in config.js**

To make this website in the real world, we must specify TURN/STUN configuration to RTCPeerConnection() . There are a lot of companies that provide free STUN/TURN servers. We will use the servers offered by XirSys.

Steps to obtain the TURN/STUN URLs from XirSys are mentioned in this README file

Paste the obtained configurations in public/js/config.js

Following is how config.js will look. (The urls will be different

turnConfig = {

iceServers: [

{

urls: [ "stun:bn-turn1.xirsys.com" ]

},

{

username: "0kYXFmQL9xojOrUy4VFemlTnNPVFZpp7jfPjpB3AjxahuRe4QWrCs6Ll1vDc7TTjAAAAAGAG2whXZWJUdXRzUGx1cw==",

credential: "285ff060-5a58-11eb-b269-0242ac140004",

urls: [

"turn:bn-turn1.xirsys.com:80?transport=udp",

"turn:bn-turn1.xirsys.com:3478?transport=udp",

"turn:bn-turn1.xirsys.com:80?transport=tcp",

"turn:bn-turn1.xirsys.com:3478?transport=tcp",

"turns:bn-turn1.xirsys.com:443?transport=tcp",

"turns:bn-turn1.xirsys.com:5349?transport=tcp"

]

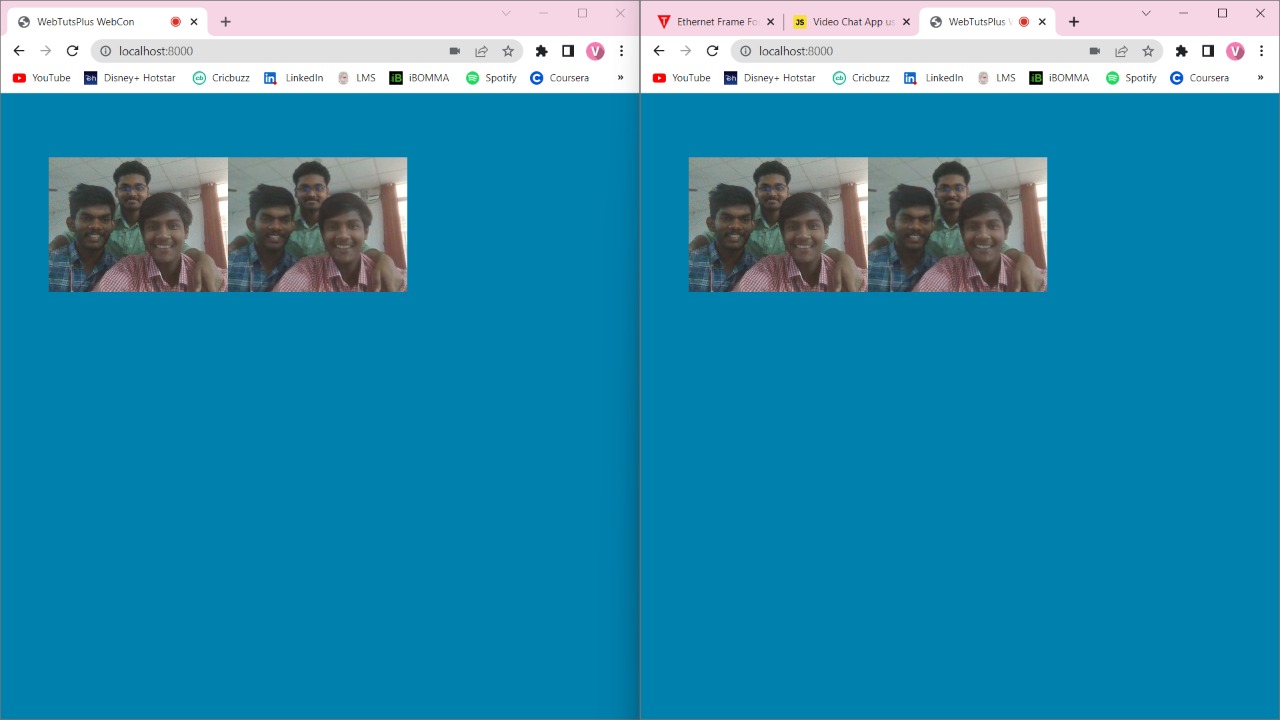
}

]

}

**Result:**

* Open a terminal in the root directory of our project.
* Run the following command — node index.js .
* Open Google Chrome and visit localhost:8000 . Enter a room name (say foo). You should see your video.
* Open a new tab and visit localhost:8000 . Enter the same room name (foo). You should now see two video elements.
* When two users are connected :



* When single user is disconnected :

