Progressive Video Conferencing

A Project Submitted

in Partial Fulfilment of the Requirements

for the Degree of

Bachelor of Technology

in

Computer Science Engineering

As part of **"CSE-2015 – Computer Networks"** course

By

1. Aditya Rana

2. Madhur Mehta

SCHOOL OF ENGINEERING AND TECHNOLOGY

BML MUNJAL UNIVERSITY GURGAON

November, 2021

Contents

Abstract

For the past years, video conferencing (VC) has become more popular and more reliable as a tool to bridge the distance gap when travel is not an option, impractical or undesired. Video conferencing uses audio and video telecommunications to bring people at different sites together. Understanding what are required for videoconferencing and its application has become one of the major researched topics by various learning institutions and businessmen. In this Computer Networks report, we created a video conferencing application with the emphasis on its application in distance learning.

Introduction

It is only recently that technology has reached a level of stability, usability and affordability which permits its use in real teaching scenarios rather than research projects. The use of video is being hailed as the next advance in electronic communication. Many companies are developing systems to support such concepts as virtual teams, telecommuting, and remote conferencing.

Video conferencing has recently become increasingly popular and disperse in the wake of faster and cheaper internet connections and better technologies. Modern standalone video conferencing units provide advanced video and audio quality due to more efficient compression and can function over normal broadband internet connections. Growing processing power and cheaper accessories, such as webcams, have also made it possible to participate in a video conference using dedicated software on a normal personal computer without any expensive special hardware.

Video conference participants use either VC system, web-based application or on-premise software to interactively communicate with co-workers, students and others in virtual meetings or classrooms. This approach is easier, cheaper and much more convenient to use while also providing easy access to file sharing and variety of others collaborative services. With the explosion of bandwidth, the resources are now available to provide more interaction in the virtual classroom via video conferencing. Using the various technologies available for video conferencing, educators can provide a more interactive distance learning experience by delivering real-time, bidirectional video, voice, and data communications to their distance students, rather than just the standard electronic media.

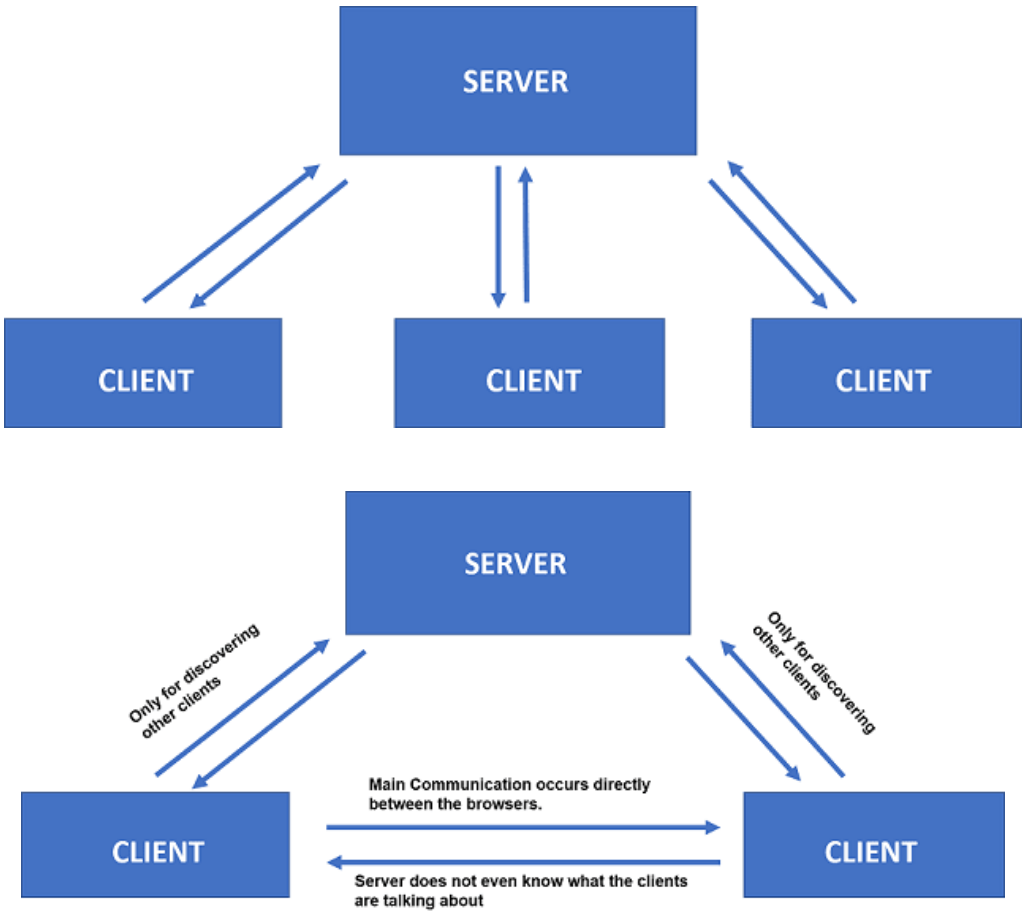
In the creation of this application, we used WebRTC API. WebRTC consists of several interrelated APIs and protocols which work together to achieve Real Time Communication.

What is WebRTC?

Imagine yourself talking to someone who receives your voice 5 secs later. You can realize how annoying it will be.

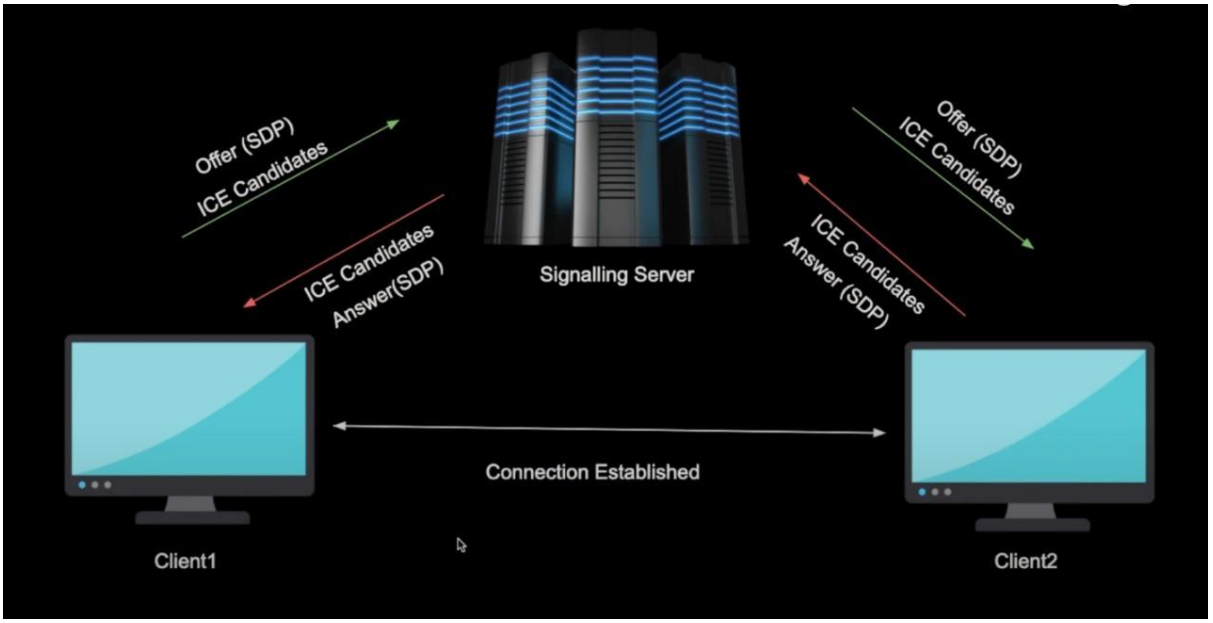
Hence, for video conferencing, we require Real-Time Communication between the browsers. Such communication is possible if we eliminate the server from between. This is why we will have to use WebRTC — an open-source framework providing web browsers and mobile applications with real-time communication via simple APIs.

WebRTC stands for Web Real-Time Communication. It enables peer-to-peer communication without any server in between and allows the exchange of audio, video, and data between the connected peers. With WebRTC, the role of the server is limited to just helping the two peers discover each other and set up a direct connection.



What actually happens is that Client1 send some data to the signalling server and it will send that data to the other client and the client will store this data.

Similarly, client 2 will perform the same action and a connection will get established between two clients.



Client 1 creates an offer and it has a SDP (Session description protocol) and it basically the media configuration of the client and it sends it to server and it sends it to the client2 who stores the information.

So now client 2 has to create an answer in response to that offer (which is configuration of client2) and sends it to client 1 through signalling server.

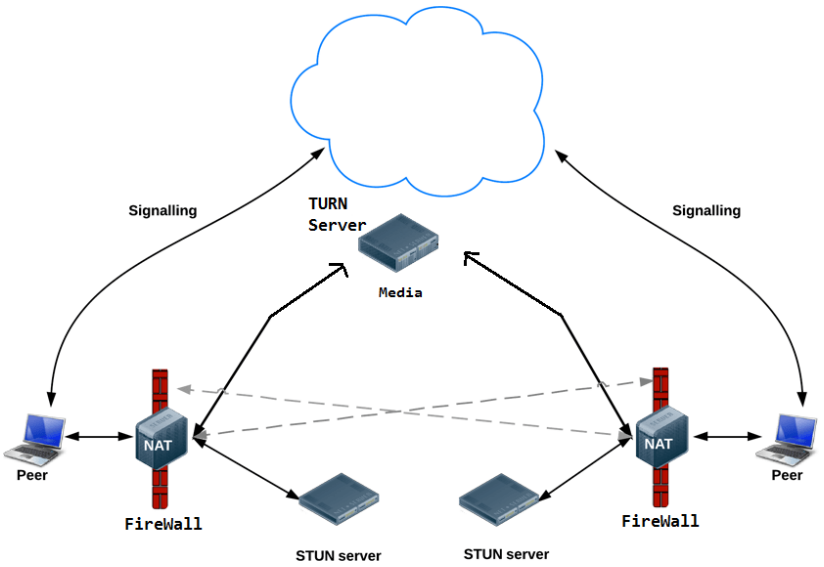
For them to convert with each other they need transfer their network configuration data.

When client1 creates the offer then ICE (Interactive Connectivity Establishment) starts coming out of webRTC API from STUN and TURN server and ore send to client 2 through signalling server. Client 2 also does the same while sending its answer to client1.

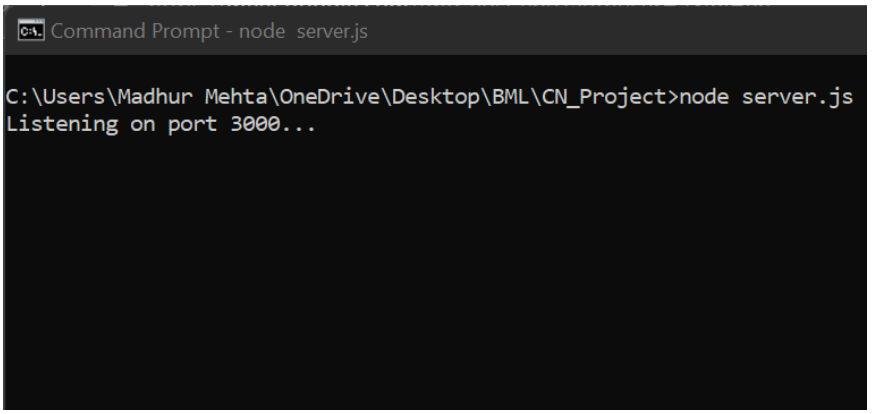
ICE candidates are generated from STUN and TURN servers. For this client 1 has to provide the URLs of the STUN and TURN server to the webRTC API

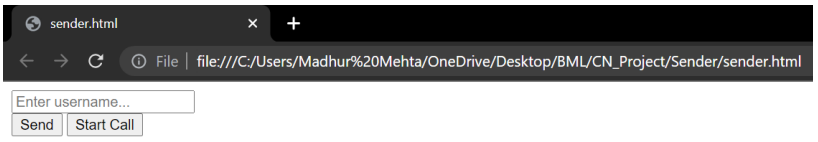
Now it sends the ICE candidates to the client 2 through signalling server. Client 2 also does same in return.

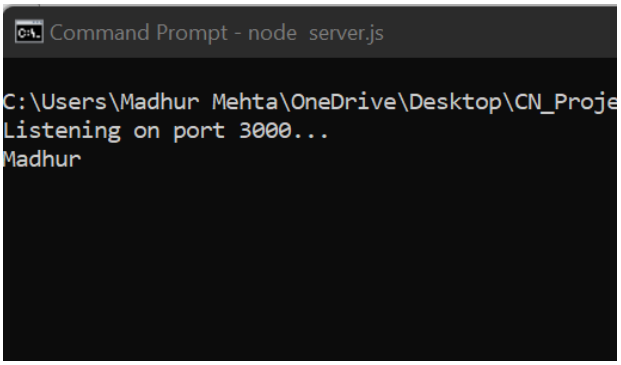
Now both the clients have network information about each other and now they can correct peer to peer.

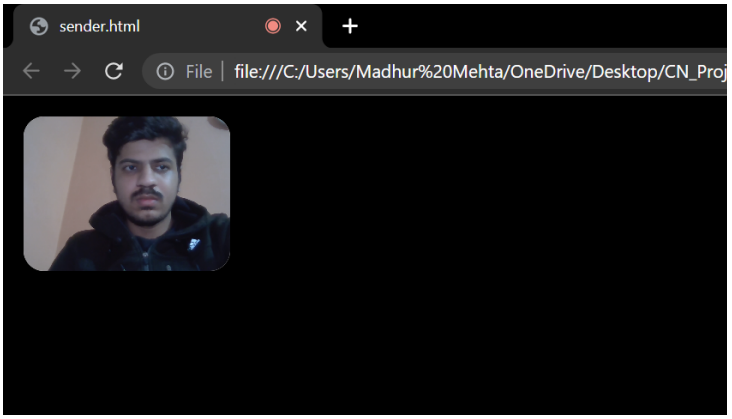


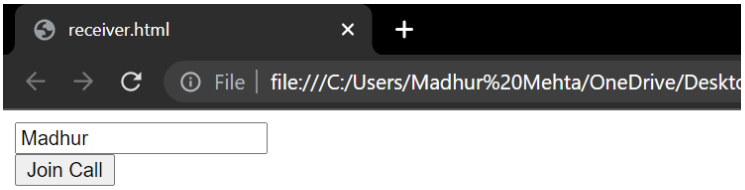
Real Time Sample

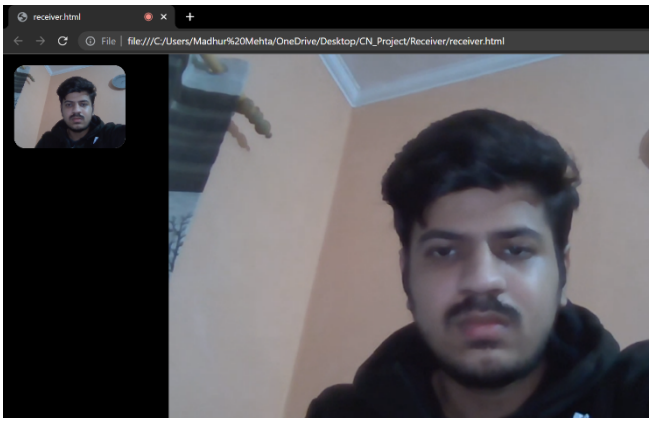





Command Prompt - node server.js
C:\Users\Madhur Mehta\OneDrive\Desktop\CN_Proje
Listening on port 3000...
Madhur


sender.html
File | file:///C:/Users/Madhur%20Mehta/OneDrive/Desktop/CN_Proj


receiver.html
File | file:///C:/Users/Madhur%20Mehta/OneDrive/Deskto
Madhur
Join Call


receiver.html
File | file:///C:/Users/Madhur%20Mehta/OneDrive/Desktop/CN_Project/Receiver/receiver.html

Source Code

Sender.js

```
const webSocket = new WebSocket("ws://127.0.0.1:3000")

webSocket.onmessage = (event) => {
    handleSignallingData(JSON.parse(event.data))
}

function handleSignallingData(data) {
    switch (data.type) {
        case "answer":
            peerConn.setRemoteDescription(data.answer)
            break
        case "candidate":
            peerConn.addIceCandidate(data.candidate)
    }
}

let username
function sendUsername() {

    username = document.getElementById("username-input").value
    sendData({
        type: "store_user"
    })
}

function sendData(data) {
    data.username = username
    webSocket.send(JSON.stringify(data))
}


let localStream
let peerConn
function startCall() {
    document.getElementById("video-call-div")
    .style.display = "inline"

    navigator.getUserMedia({
        video: {
            frameRate: 24,
            width: {
                min: 480, ideal: 720, max: 1280
            },
            aspectRatio: 1.33333
        },
        audio: true
    }, (stream) => {
        localStream = stream
        document.getElementById("local-video").srcObject = localStream

        let configuration = {
            iceServers: [
                {
                    "urls": ["stun:stun.l.google.com:19302",
                    "stun:stun1.l.google.com:19302",
                    "stun:stun2.l.google.com:19302"]
                }
            ]
        }

        peerConn = new RTCPeerConnection(configuration)
        peerConn.addStream(localStream)
```

```
        peerConn.onaddstream = (e) => {
            document.getElementById("remote-video")
            .srcObject = e.stream
        }

        peerConn.onicecandidate = ((e) => {
            if (e.candidate == null)
                return
            sendData({
                type: "store_candidate",
                candidate: e.candidate
            })
        })

        createAndSendOffer()
    }, (error) => {
        console.log(error)
    })
}

function createAndSendOffer() {
    peerConn.createOffer((offer) => {
        sendData({
            type: "store_offer",
            offer: offer
        })

        peerConn.setLocalDescription(offer)
    }, (error) => {
        console.log(error)
    })
}

let isAudio = true
function muteAudio() {
    isAudio = !isAudio
    localStream.getAudioTracks()[0].enabled = isAudio
}

let isVideo = true
function muteVideo() {
    isVideo = !isVideo
    localStream.getVideoTracks()[0].enabled = isVideo
}
```

Receiver.js

```
const webSocket = new WebSocket("ws://127.0.0.1:3000")

webSocket.onmessage = (event) => {
    handleSignallingData(JSON.parse(event.data))
}

function handleSignallingData(data) {
    switch (data.type) {
        case "offer":
            peerConn.setRemoteDescription(data.offer)
            createAndSendAnswer()
            break
        case "candidate":
            peerConn.addIceCandidate(data.candidate)
    }
}

function createAndSendAnswer () {
    peerConn.createAnswer((answer) => {
```

```
        peerConn.setLocalDescription(answer)
        sendData({
            type: "send_answer",
            answer: answer
        })
    }, error => {
        console.log(error)
    })
}

function sendData(data) {
    data.username = username
    webSocket.send(JSON.stringify(data))
}


let localStream
let peerConn
let username

function joinCall() {

    username = document.getElementById("username-input").value

    document.getElementById("video-call-div")
    .style.display = "inline"

    navigator.getUserMedia({
        video: {
            frameRate: 24,
            width: {
                min: 480, ideal: 720, max: 1280
            },
            aspectRatio: 1.33333
        },
        audio: true
    }, (stream) => {
        localStream = stream
        document.getElementById("local-video").srcObject = localStream

        let configuration = {
            iceServers: [
                {
                    "urls": ["stun:stun.l.google.com:19302",
                    "stun:stun1.l.google.com:19302",
                    "stun:stun2.l.google.com:19302"]
                }
            ]
        }

        peerConn = new RTCPeerConnection(configuration)
        peerConn.addStream(localStream)

        peerConn.onaddstream = (e) => {
            document.getElementById("remote-video")
            .srcObject = e.stream
        }

        peerConn.onicecandidate = ((e) => {
            if (e.candidate == null)
                return

            sendData({
                type: "send_candidate",
                candidate: e.candidate
            })
        })
```

```
        sendData({
            type: "join_call"
        })

    }, (error) => {
        console.log(error)
    })
}

let isAudio = true
function muteAudio() {
    isAudio = !isAudio
    localStream.getAudioTracks()[0].enabled = isAudio
}

let isVideo = true
function muteVideo() {
    isVideo = !isVideo
    localStream.getVideoTracks()[0].enabled = isVideo
}
```

Server.js

```
const Socket = require('websocket').server
const http = require("http")

const server = http.createServer((req, res) => {})

server.listen(3000, () => {
    console.log("Listening on port 3000...")
})

const webSocket = new Socket({ httpServer: server })

let users = []

webSocket.on('request', (req) => {
    const connection = req.accept()

    connection.on('message', (message) => {
        const data = JSON.parse(message.utf8Data)

        const user = findUser(data.username)

        switch(data.type) {
            case "store_user":

                if (user != null) {
                    return
                }

                const newUser = {
                     conn: connection,
                     username: data.username
                }

                users.push(newUser)
                console.log(newUser.username)
                break
            case "store_offer":
                if (user == null)
                    return
                user.offer = data.offer
                break
```

```
            case "store_candidate":
                if (user == null) {
                    return
                }
                if (user.candidates == null)
                    user.candidates = []

                user.candidates.push(data.candidate)
                break
            case "send_answer":
                if (user == null) {
                    return
                }

                sendData({
                    type: "answer",
                    answer: data.answer
                }, user.conn)
                break
            case "send_candidate":
                if (user == null) {
                    return
                }

                sendData({
                    type: "candidate",
                    candidate: data.candidate
                }, user.conn)
                break
            case "join_call":
                if (user == null) {
                    return
                }

                sendData({
                    type: "offer",
                    offer: user.offer
                }, connection)

                user.candidates.forEach(candidate => {
                    sendData({
                        type: "candidate",
                        candidate: candidate
                    }, connection)
                })

                break
        }
    })

    connection.on('close', (reason, description) => {
        users.forEach(user => {
            if (user.conn == connection) {
                users.splice(users.indexOf(user), 1)
                return
            }
        })
    })
})

function sendData(data, conn) {
    conn.send(JSON.stringify(data))
}

function findUser(username) {
    for (let i = 0;i < users.length;i++) {
        if (users[i].username == username)
            return users[i]
```

```
    }
}
```

We have also created 2 user interfaces:

A sender UI in which we have username input, send and Start call button. When we put in the username and click send is sent to the server and the server stores the username, when we click on the Start call button the sender will create its offer and it will send it to the server and the server will store that offer against the username that was previously sent, and once the offer is sent the sender will start generating its ICE candidates and then the candidates will also be sent ot the server and the server will store that information.

A receiver UI from which we will join the call by entering the username and clicking on join call which will ask the server for the offer and the ICE candidates of the username that it send, and when the reciever gets the offer and the ICE candidates from the server, then it creates its own answers in response to the offer and it sends the answer to the server and the server sends the answer to the username it tried to call. Also along with it the ICE candidates are gathered by the receiver and send to the server and the server sends it to the username that it tried to call.

Socket Elements

const Socket = require('websocket').server

We use the WebSocket.Server method to create a new WebSocket server

const server = http.createServer((req, res) => {})

 The server does create an IncomingRequest and ServerResponse instance for each request it receives, and passes them to the request event listener - they are the objects that you receive in the typical (req, res) => { … } functions.

const webSocket = new Socket({ httpServer: server })

 Socket elements used for creating a new socket

Once the socket is created, we should listen to events on it. There are totally 4 events:

open – connection established,

message – data received,

error – websocket error,

close – connection closed.

And if we'd like to send something, then socket.send(data) will do that.

getUserMedia(): capture audio and video.

The deprecated Navigator.getUserMedia() method prompts the user for permission to use up to one video input device (such as a camera or shared screen) and up to one audio input device (such as a microphone) as the source for a MediaStream.

RTCPeerConnection (API and signaling: Offer, answer, and candidate)

It is used to stream audio and video between users. Hence, Signalling works together with RTCPeerConnection to establish a direct connection between the browsers.

createOffer()

The createOffer() method of the RTCPeerConnection interface initiates the creation of an SDP offer for the purpose of starting a new WebRTC connection to a remote peer.

setLocalDescription()

The RTCPeerConnection method setLocalDescription() changes the local description associated with the connection. This description specifies the properties of the local end of the connection, including the media format.

Onicecandidate()

The RTCPeerConnection property onicecandidate property is an event handler which specifies a function to be called when the icecandidate event occurs on an RTCPeerConnection instance. This happens whenever the local ICE agent needs to deliver a message to the other peer through the signaling server.

GitHub Link and execution

Link - https://github.com/MadhurMehta07/CN-Project

How to execute: To execute the Project you need to Download the Zip file from the above GitHub link and extract the folder into your system.

1) Open command prompt and change directory to the path where you saved the project folder.

2) Now type the command “node server.js”, there will be a answer “Listening on port 3000…”.

3) Now open the sender html frontend window in your browser and type a username and click on send.

4) “Username” you entered will be written on the cmd window. Similarly open the receiver side and enter the username and join call. Now you can enjoy the video conferencing experience.

Features

Toggle Button:

In the seamless world we always require a new and innovative way to handle our tasks. Hence, we enabled our users to have a very sophisticated experience with their conferencing which will be supported by simple toggle buttons which can allow our users to either control the microphone or camera.

End the call:

Exhausted or ended we have enabled our users to get a quick relief from their meetings with our simple sophisticated solution which led to the innovation of simple touch end button. This button is used by our user base to end their calls and have faster response to end the video conferencing.

Share View:

Beat an official presentation or relaxing fun recreation we bring it together to our users by an end-to-end screen sharing feature which allows the users to have the time to present the screen where it has multiple features which is aimed to be industry-leading so as to let our users to share anything they want to. Starting it from the basic presentations even a picture can be presented which streams with the highest quality inclusive of audio and video.

Conclusion

This integrated video conferencing app is aimed to have a solution at one-stop which can satisfy any user. And we could recreate this video conferencing software by taking the inspiration of various present existing video conferencing software and applications. We would have this video conferencing solution with maximum possible features so as to benefit our customers and users so that they have fullest and immense experience through our seamless and simple software which is dedicated for effortless video conferencing.

Through the process we do remember the pandemic situation and hence have decided to make our video conferencing app and solution the best suited for every age regardless the experience of the user. We enabled a basic feature which can be understood and performed even by the first timers.

Every feature be included in this video conferencing app is made sure that it has the best quality and performance. And we keep our video conferencing solution ever updating so as to fix our user base on to only one video conferencing solution where all the needs and requirements are met. And hence we call or video conferencing solution “Progressive Vide Con”.

Future of our Solution

As already mentioned above we all ourselves progressive and this is proved in our work and dedication to implement every feature into our video conferencing app. We are not done yet. For now, we just have the basic structure ready while we see a bright future of our video conferencing app which includes a “Share Music” which we believe creates a moment between the classes or stressed meets to ease the moment. Also, we do seek legal permissions from every streaming device so as to perfectly enable a feature of Share View to our users effortlessly so that they can you literally any streaming application through our video conferencing solution. We also find new updates coming up day by day to the world of video conferencing and hence is our progressive solution ever updating.