University of Mauritius

FACULTY OF ENGINEERING

Department of Electrical and Electronics Engineering

IMPLEMENTATION OF A WEBRTC VIDEO CONFERENCING AND STREAMING APPLICATION

By

ANOOROOP Neervan (1610474)

March 2019

A project submitted in partial fulfilment for the award of the degree of

BSc (Hons) Electronics with Computer Science

A project supervised by

Dr. T. P Fowdur

Table of Contents

List of Tables

List of Figures

Acknowledgments

First and foremost, I would like to express my deepest gratitude to my supervisor Dr. T. P Fowdur. The latter provided me with every virtue I needed in order to complete this project on time and in style.

I would also like to extend my thanks to Mr. Yogesh Beeharry, lecturer in the department of Electrical and Electronics Engineering for helping me out on the port mapping part of this project.

Last but not least, big thanks to my mum, my dad, my grandma, my uncle, my sister and my best friend for their never-ending support, encouragement and motivation in everything that I do.

UNIVERSITY OF

MAURITIUS

GROUP PROJECT/DISSERTATION DECLARATION

FORM

Name:
Student ID:
Programme of Studies:
Module Code/Name:
Title of Project/Dissertation:
Declaration: In accordance with the appropriate regulations, I hereby submit the above dissertation for examination and I declare that: (i) I have read and understood the sections on **Plagiarism and Fabrication and Falsification of Results** found in the University's "General Information to Students" Handbook (20…./20….) and certify that the dissertation embodies the results of my own work. (ii) I have no objection to submit a soft copy of my dissertation through the Turnitin Platform. I confirm that the hard copies and soft copy, submitted to the Faculty/Centre Registry, and the soft copy (main body, i.e, introduction up to the last Chapter) uploaded through Turnitin Platform are identical in content. (iii) I have adhered to the 'Harvard system of referencing' or a system acceptable as per "The University of Mauritius Referencing Guide" for referencing, quotations and citations in my dissertation. Each contribution to, and quotation in my dissertation from the work of other people has been attributed, and has been cited and referenced. (iv) I have not allowed and will not allow anyone to copy my work with the intention of passing it off as his or her own work. (v) I am aware that I may have to forfeit the certificate/diploma/degree in the event that plagiarism has been detected after the award. (vi) Notwithstanding the supervision provided to me by the University of Mauritius, I warrant that any alleged act(s) of plagiarism during my stay as registered student of the University of Mauritius is entirely my own responsibility and the University of Mauritius and/or its employees shall under no circumstances whatsoever be under any liability of any kind in respect of the aforesaid act(s) of plagiarism.

Signature:	**Date:**

Abstract

In today's world, real-time Internet communication has become a simple and convenient way to communicate with people, regardless of their location. However, the increased number of internet users cannot always give the highest quality of communication. This project presents a thesis on how video conferencing and streaming quality is impacted by the increasing number of users attending a conference or streaming using the most basic WebRTC application. The application is developed using HTML 5 (Hypertext Markup Language 5) and JavaScript alongside Node.js and it includes two real-time communication (RTC) features: Video Conferencing and Video Streaming. In addition to video chatting, the video conferencing part includes a real-time text messaging option and a file sharing option and can accommodate up to four clients. The video streaming part is further broken down into two sections: Live Streaming and File Streaming. The former broadcasts the live preview of a webcam whereas the latter allows the initiator to browse and broadcast a local video. Contrary to video conferencing, video streaming can accommodate an infinite number of clients. All communications are made using offer/answer negotiation between WebRTC clients via a signaling server. Videos were recorded for five minutes on both initiator and client sides using the implemented record button and were compared head to head at the start and at the receiving end using the PSNR metric. As a result, Quality of Experience (QoE) proved to be more stable using Ethernet Cabled connection compared to Wi-Fi connection despite a noticeable decline in PSNR value with an increase in users; as a matter of fact, in video conferencing with Wi-Fi connection, the PSNR was at 40.0888 dB with one user and 28.5186 dB with four users compared to 42.5420 dB with one user and 33.3966 dB with four users with Ethernet Cabled connection. The results obtained proves to be massive to RTC subjects, especially for web developers; for instance, they may get an idea of where to set a limit for number of clients for the Quality of Service (QoS) to still be appreciable.

List of Acronyms

API	Application Program Interface
APL	Advanced Programming Languages
CPU	Central Processing Unit
DMZ	Demilitarized zone
DSL	Digital Subscriber Line
HREF	Hypertext REFerence
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
ICE	Interactive Connectivity Establishment
IETF	Internet Engineering Task Force
IP	Internet Protocol
ITU	International Telecommunication Union
JIT	Journal of Information Technology
JSEP	JavaScript Session Establishment Protocol
KPI	Key Performance Indicator
LAN	Local Area Network
MAX	Maximum
MOS	Mobile Operating System
MSE	Mean Squared Error
NAT	Network address translation
P2P	Peer-to-Peer

PC	Personal Computer
PESQ	Perceptual Evaluation of Speech Quality
PEVQ	Perceptual Evaluation of Video Quality
PSNR	Peak signal-to-noise ratio
QoE	Quality of Experience
QoS	Quality of Service
RTC	Real-Time Communication
RTP	Real-Time Transport Protocol
SDP	Session Description Protocol
SSIM	Structural Similarity Index
STUN	Session Traversal Utilities for NAT
TURN	Traversal Using Relays around NAT
URI	Uniform Resource ID
URL	Uniform Resource Locator
VoIP	Voice over Internet Protocol
VQM	Visual Quality Metrics
WAN	Wide Area Network
WebRTC	Web Real-Time Communication
WWW	World Wide Web
XHR	XML Http Request
XML	Extensible Markup Language

Chapter 1: Introduction

Over the past decades, communication over the internet has evolved in exponential fashion; people went from sending emails back and forth all the way to being able to see and talk to each other in real time. Speaking about real time communication, people usually refer to applications such as Skype, Facebook Messenger, WhatsApp, FaceTime, amongst others. However, instead of downloading these third-party software, real time communication can be taken a step further; for the first time, browsers are able to directly exchange real-time media with other browsers in a peer-to-peer fashion without any plug-ins. This new form of peer-to-peer communication is called Web Real-Time Communication (WebRTC) and it is growing at a swift pace. According to the Cisco Visual Networking Index published in June 2017 [1], live video traffic (streaming, video conferencing) is expected to grow dramatically from three percent of Internet video traffic in 2016 to 13 percent by 2021, which amounts to 1.5 exabytes (1 exabyte = 1 million terabytes) per month in 2016, up to 24 exabytes per month in 2021. As this number continues to rise, the quality of video calls and live streams is becoming more and more questionable. That being the case, the aim for this project is to analyze the quality of WebRTC streams received with reference to the number of peers joining the stream in order to help provide better user experience. A service that is faulty or in other ways not working properly will quickly lose its popularity and customers.

1.1 Overview of previous researches on video transmission

In limited network conditions, Gunkel et al. discussed Quality of Experience (QoE) in video conferencing [2]. They conducted experiments on how for all other participants in the conversation, a single participant with a limited connection can negatively affect the QoE. Through these experiments, the layout, video quality and conversation network limitations were altered. Asymmetric connection experiments were highly relevant because they depicted a real-life environment.

In [3], D. Vucic and L. Skorin-Kapov investigated QoE issues in the context of mobile multi-party tele meetings via WebRTC. A series of interactive, three-party audiovisual tele meeting tests with different smartphone and laptop configurations was conducted in a natural environment. The

results indicated amongst others that especially for videoconferencing on smartphone, participants had lower expectations. The authors also argued that it might be necessary to shift the processing burden (or part of it) required for multi-party tele meetings to a centralized conference media server (as opposed to a full-mesh topology), since many smartphones may not be able to meet the high CPU requirements needed to ensure a smooth QoE.

Potential QoE killers were discussed in [4] in terms of performance statistics. The authors tried to link the performance statistics to the QoE of the users. They conducted a series of tests involving conversations between two parties while collecting feedback from real-time sessions using the WebRTC-internals tool to find specific network parameters that can be linked to the perceived user experience. These results identified video freezes, that can be caused by changing these network parameters, as a key QoE killer. They also suggested that other possible QoE killers were further investigated.

In [5], a first step has been proposed to assess the quality of the video aired by a WebRTC to many viewers. The authors used the Structural Similarity (SSIM) Index [6] as a measurement of video quality for this experiment and the test aimed to measure how many viewers could join in viewing the broadcast while maintaining an acceptable quality of the image. The SSIM measure remained surprisingly stable with values in the interval [0.96, 0.97] as the number of viewers joining the broadcast increased. When the number of customers reached approximately 175, SSIM suddenly fell to values close to 0. For an increase from 1 to 175 viewers, the user experience would probably remain acceptable without loss of quality.

A recent experiment has been published in [7] to evaluate the quality of WebRTC video streaming experience in the broadband mobile network. Various videos in various resolutions (from 720×480 to 1920×1080) were used between the Chrome browser and the Kurento media server as input for a video call over WebRTC. Subjectively, 28 persons gave 1 (low quality) to 5 (excellent quality) for WebRTC videos. The authors then used a number of methods to assess objectively the quality of WebRTC videos, all based on errors calculated between the original video and the WebRTC video.

1.2 Motivation

It is clear from previous works such as [5] and [7] that the assessment of the QoE with regards to WebRTC is very important. However, previous works have not provided the actual PSNR of videos at their source and their destination with an elevated number of users. Moreover, research has not focused on the type of internet connection used; this can be taken a step further by evaluating video quality on Wi-Fi connection as well as Ethernet cable connection.

1.3 Project Objectives

The aim of this project is to implement and test a basic application for video conferencing and streaming that runs through WebRTC protocols, with different numbers of users and different types of internet connection media to examine how these factors affect the Quality of Service (QoS). The application consists of a home page on which the users decide whether to set up a conference or a broadcast by entering either a room number or a stream name respectively. Users are then able to record their video during conferencing or broadcasting, which is then compared to their respective videos received by other users and the PSNRs of these videos are calculated using MATLAB codes. All the codes were written in JavaScript and were modified using NetBeans 8.2 and Node.js was used as the backbone of our signaling server.

1.4 Thesis outline

Chapter 1: Introduction

This chapter briefly discusses the topic and background of the research paper. It includes the research project's goals and provides a summary about the project.

Chapter 2: Literature Review and Theoretical Framework

This chapter examines the background frameworks and discusses the different analysis being made about WebRTC.

Chapter 3: Methodology and Software Implementation

In chapter 3, the implementation and approach of the WebRTC Video Conferencing/Streaming application are elaborated.

Chapter 4: System Testing and Results

The testing and results obtained for the implemented software are discussed in this chapter.

Chapter 5: Conclusion

Chapter 5 concludes the entire thesis and provides guidance on possible future works that can be carried out.

Chapter 2: Literature Review and Theoretical Framework

This chapter gives a detailed scope of previous works done in the field and also presents the basic theoretical framework used in this project.

2.1 Literature Review of Web Real-Time Communication

2.1.1 Web Real-Time Communication (WebRTC)

Web Real-Time Communication (WebRTC) is a new web browsing model standard and industry effort. For the first time, browsers are able exchange media directly in peer-to-peer fashion with other browsers. WebRTC is used in various apps for instance, WhatsApp, Facebook Messenger, appear.in and platforms such as TokBox [8].

The classic semantics of the web architecture are a client-server paradigm in which browsers send a request for content to the web server from a HTTP, which answers with a response with the requested information. A URI or URL is closely linked to the resources provided by the server. The most common WebRTC architecture is where two browsers use the same web application downloaded from the same webpage; in that case the Trapezoid becomes a triangle as shown in Figure 2.1 [9].

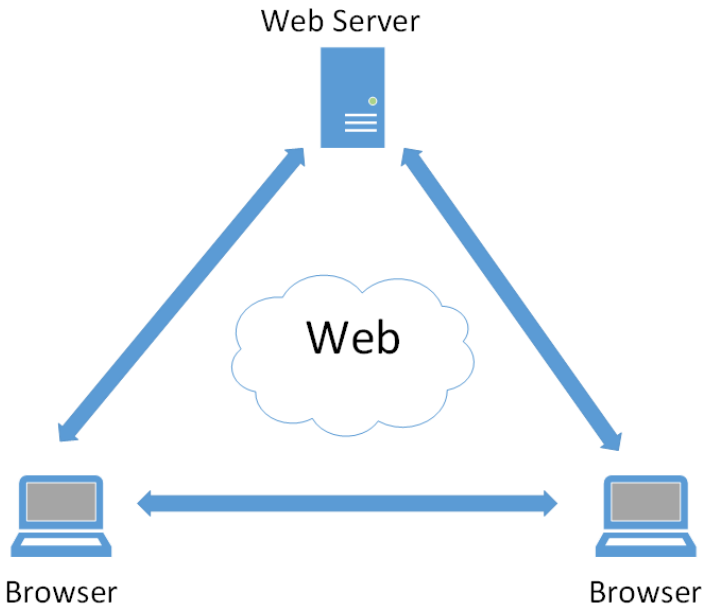

Figure 2.1: The WebRTC Triangle

2.1.2 Evolution of Web Real-Time Communication

In the early 1990s, the World Wide Web (WWW) was first built up on the basis of a page - centered model that used HREF hyperlinks. In this early web model, browsers were navigating from page to page, presenting new content and updating their user interfaces based on html [10].

The year 2000 saw the development of a new Web browsing approach and by the middle of that decade the XHR API became standardized. This new XHR API allowed web developers to create web applications that do not require a new page to update their user interface or content. They have made it possible to use server-based web services that provide access to structured data and pages or other content snippets. This resulted in a completely new web approach, now commonly known as web 2.0. The new XHR API has helped us create a much more dynamic, social website for services such as Gmail, Facebook, Twitter and more.

Now the web is undergoing a further transformation, which allows individual web browsers to stream data to each other directly without having to send it via intermediate servers. This new form of peer-to-peer communication is built on a new set of Web Real-Time Communications Working Group [8] APIs and on a number of protocols standardized by real-time Web browsing communication [9].

Like the implementation of the XHR API, the Web 2.0 revolution was also brought about with the introduction of the new WebRTC standards. It's now time to say hello to the web in real time!

2.1.3 Overview of previous research on WebRTC

Although the quality evaluation methods for multi-party video conversations have not been agreed, there are already several studies in this field. In [11], Berndtsson et al. reported on a number ways and conversational tests aimed at evaluating the subjective quality of tele meeting scenarios. A conversation audiovisual test with two parties indicated that the synchronization of audio and video is very important and that, in cases where the delay is below 600 ms, audio is even more delayed to ensure better synchronization. Further multitasked tests showed that a delay of 800 ms is not acceptable, and that it is shown to have an impact on experienced interactive audios. There were

however some interesting nuances: the delay tolerance in the audio test was higher. The levels of tolerance also differed by the type of task (free conversation versus a question - like task) and social context (in a room versus others only). It has been found in [12] that the degree of participation in the dialog (listening vs. conversation) also has an impact on the experienced quality of the conversation. For their specific set-up, they have also found that this impact can be even greater than the technical factor in question (i.e. the various rates of packet loss).

In [13], the authors reported on a subjective user study involving three-party audiovisual conversations established via mobile devices and aimed to investigate the impact of different video resolutions on QoE. It was also investigated how different bandwidth limitations impacted QoE for different video resolutions. The calls were established using a Web-based Real-Time Communication (WebRTC) application running on the Licode Multipoint Control Unit [14]. One of its findings stated that unlimited bandwidth setting resulted in significantly reduced user perceived overall quality for all resolutions above 480x320, meaning that the capabilities of the tested mobile phones had trouble processing multiple real-time videos with high bitrates and resolutions. Moreover, in every test with unlimited bandwidth, participants reported picture freezing, although the speech was unimpaired and communication was not completely interrupted. Further simulations showed that the amount of generated traffic had a significant influence on the QoE, especially for bitrates higher than 1200kbps for each resolution tested, which may be attributed to high demands on smartphone processing power. However, despite the lack of smartphone processing power, 5.1” screen size remained as an argument that resolutions higher than 640x480 were unnecessary for three-party video conference calls.

Video conferencing QoE is affected on several layers and often suffers because of congestions. To reduce loss and maintain a high level of bandwidth utilization, dynamic congestion control was proposed in [15]. The main objective was to estimate – using a Kalman filter [16] – the end-to-end one-way delay variation which was experienced by packets traveling from a sender to a destination. This estimate was compared with a dynamic threshold and drove the dynamics of a controller located at the receiver which aimed at maintaining queuing delays low, while a loss-based controller located at the sender acted when losses were detected. The link capacity was varied every interval of 50 seconds and four time intervals were considered in which the capacity

was set respectively to 1Mbps, 2.5Mbps, 0.5Mbps, and 1Mbps, such that the video call lasted 200s. It was noticed that, in order to react to the link capacity drop, the algorithm had to reduce the sending rate to keep a low queuing delay and loss ratio.

In [17], the authors performed subjective visual quality tests in an HD four-way desktop video conferencing system, which required audiovisual interaction. They investigated the impact of bitrate and packet-loss on overall, audio, and video quality under different Internet access technologies typical for domestic households: broadband, DSL, and mobile. Different encoding bitrates (256kbs, 1024kbs and 4096kbs) and packet loss rates (0, 0.5%) in groups of four participants were tested with a scenario based on the ITU building blocks task [18]. The influence of group interaction and individual idiosyncrasies were discussed based on different mixed models, and covariates engagement and enjoyment as further explanatory factors were looked at. It was found that 256kbs was still sufficient to provide a fair overall experience, but video quality was noticed to be poor. On the higher bitrate end, most people would not perceive the difference between 1024kbs and 4096kbs, considering in both cases the quality to be close to excellent. Independent on bitrate, packet loss had a small but significant impact, quantifiable in, on average, less than half a point difference on a 5-point ITU scale.

This authors in [19] focused on the QoE assessment of WebRTC-based applications and its contribution is threefold. First, an analysis of how WebRTC topologies affect the quality perceived by users was provided. Second, a group of Key Performance Indicators (KPIs) for estimating the QoE of WebRTC users was proposed. Finally, a systematic survey of the literature on QoE assessment in the WebRTC arena was presented. The results of this review have shown that there was a preference for assessing the quality experienced by end users using QoS parameters rather than directly with QoE methods Regarding QoS, the preferred parameters to evaluate WebRTC applications were: end-to end delay, jitter, packet loss, bandwidth, throughput, and bitrate. Regarding QoE, subjective MOS was the preferred assessment method, followed by several objective methods (PESQ, PEVQ, VQM, PSNR, and SSIM). Furthermore, the congestion control algorithm, implemented at the application layer in the WebRTC stack [20] adapt the WebRTC transmission bitrate when detecting network congestion thus, leading to play an important role in the total amount of end-to-end delay in the WebRTC.

The authors in [21] proposed to create and implement a WebRTC hybrid signaling mechanism named (WebNSM) for video conferencing based on the Socket.io (API) mechanism and Firefox. WebNSM was designed over a combination of different topologies, such as simplex, star and mesh. Therefore, it offered several communications at the same time as one-to-one (unidirectional/bidirectional), one-to-many (unidirectional) and many-to-many (bi-directional) without any downloading or installation. WebRTC video conferencing was accomplished via LAN and WAN networks, including the evaluation of resources in WebRTC like bandwidth consumption, CPU performance, memory usage, Quality of Experience (QoE) and maximum links and RTPs calculation. Furthermore, a novel signaling mechanism was built among different users, devices and networks to offer multi-party video conferencing using various topologies at the same time, as well as other typical features such as using the same server, determining room initiator, keeping the communication active even if the initiator or another peer left, etc. Results showed that mesh topology [22] limited the number of peers as it requested high CPU and high bandwidth speed; as high as the CPU core, it would lead to allow more peers to join, better communication and encoding & decoding.

2.2 Theoretical Framework

WebRTC offers Real-Time Communication (RTC) functionality through simple APIs to browsers and mobile applications however, it proves to be a complex subject which needs to be looked after in a meticulous manner. Henceforth, the core technologies and protocols will be explained and each component in the WebRTC will be described in detail.

2.2.1 JavaScript

JavaScript is a lightweight interpreted or JIT-compiled [23] programming language with first-class functions. While it is most well known as the scripting language for Web pages, many non-browser environments also use it, such as Node.js, Apache CouchDB and Adobe Acrobat. JavaScript is a prototype-based, multi-paradigm, dynamic language, supporting object-oriented, imperative, and declarative (e.g. functional programming) styles [24].

2.2.2 Node.js

Node.js is an open – source and cross - platform JavaScript runtime environment. Node.js runs the V8 JavaScript engine, the core of Google Chrome, outside the browser. Node.js is able to leverage

the work of the engineers who made the fast blazing Chrome JavaScript runtime, enabling Node.js to benefit from the substantial performance improvements and the Just-In-Time compilation that V8 performs [25]. The most common example of Node.js is a web server as shown in Figure 2.2:

```
const server = http.createServer((req, res) => {
  res.statusCode = 200
  res.setHeader('Content-Type', 'text/plain')
  res.end('Hello World!\n')
})

server.listen(port, hostname, () => {
  console.log(`Server running at port 3000`)
})
```

Figure 2.2: Node,js web server

2.2.3 WebRTC API

Application programming interface (API) is a set of subroutine definitions, communication protocols and software building tools. In general, it is a set of clearly defined methods of communication between different components. A good API makes it easier to develop a computer program by providing all the building blocks that are then put together by the programmer.

WebRTC implements three APIs:

- MediaStream (aka getUserMedia)
- RTCPeerConnection
- RTCDataChannel

2.2.3.1 MediaStream

The MediaStream API enables users to have access to media streams, such as cameras and microphones, from local entry terminals. The getUserMedia API [26] was first focused on but has now been officially developed as the MediaStream API. However, the getUserMedia () method is still the main way to start access for local input devices.

MediaStream API has three key points of functionality:

- It gives a developer access to a stream object that represent video and audio streams

- It manages the selection of input user devices in case a user has multiple cameras or microphones on his device
- It provides a security level asking user all the time he wants to fetch the stream

2.2.3.2 RTCPeerConnection

The RTCPeerConnection API is the core of the peer-to-peer connection between each WebRTC enabled browser or peer. Since WebRTC is relatively new, peer connection has various implementation systems [27] for different browsers to make sure that it works in Firefox, Chrome and Opera, and when creating an object, it needs to take into consideration various function calls when creating an object; In Firefox it is called mozRTCPeerConnection, and in Chrome its webkitRTCPeerConnection. To create an object RTCPeerConnection, var peerconnection = RTCPeerConnection (configuration); constructor is used. The peerconnection object is then used in slightly different ways on each client, depending on the caller or the callee.

- Below is a summary of the caller's flow after the peerconnection object is created:
    - Register the onicecandidate handler
    - Register the onaddstream handler
    - Register the message handler
    - Use getUserMedia to access the local camera
    - The JSEP offer/answer process

- The following is a summary of the callee's flow, which is very similar in a lot of ways to the caller's flow, except that it responds to the offer with an answer:
    - Register the onicecandidate handler
    - Register the onaddstream handler
    - Register the message handler
    - Use getUserMedia to access the local camera
    - The JSEP offer/answer process

2.2.3.3 RTCDataChannel

RTCDataChannel enables the arbitrary exchange of data between two peers with customized transport delivery characteristics. Each application using this channel can configure it to provide: a reliable or partially confident delivery of the messages transmitted and in-order or out-of-order delivery of sent messages. It is a bidirectional channel, which is part of the PeerConnection API. When using the channel, the PeerConnection object is called with createDataChannel ("channel name) "to create a dataChannel with a specific name or it is received in an existing PeerConnection channel event of the type RTCDataChannelEvent [28].

2.2.4 Offer and Answer

PeerConnection uses a media information offer/answer structure. The partners must determine and share local and remote audio and video information, such as resolution and codec capabilities, in order to set up a multimedia communication. The message for media exchange is sent via a signaling server and one peer sends an offer using the SDP protocol. The recipient receives the message on the server and uses createAnswer () to send a response to the caller.

2.2.5 Session Description Protocol

The Session Description Protocol (SDP) is a standard to help establish media connections by describing their initialization parameters. In July 2006, the IETF published a revised specification as a proposed IETF standard [29].

The purpose of SDP is to describe multimedia communication sessions, invitations and negotiating parameters between parties. SDP does not provide the true media, but is used to negotiate between the end points, the medium type and other associated properties for streaming.

2.2.6 Network Address Translation

Network Address Translation (NAT) is a technique that maps the internal addresses of the network to external addresses. It allows several local devices to be connected to an external connection (Internet), as shown in Figure 2.3. When a device sends data to the local network outside the local network, the NAT translates the IP address to the external address or public address. When the

NAT receives external address packets, they are translated back in such a way that the packet finds the correct peer of the local network [30].

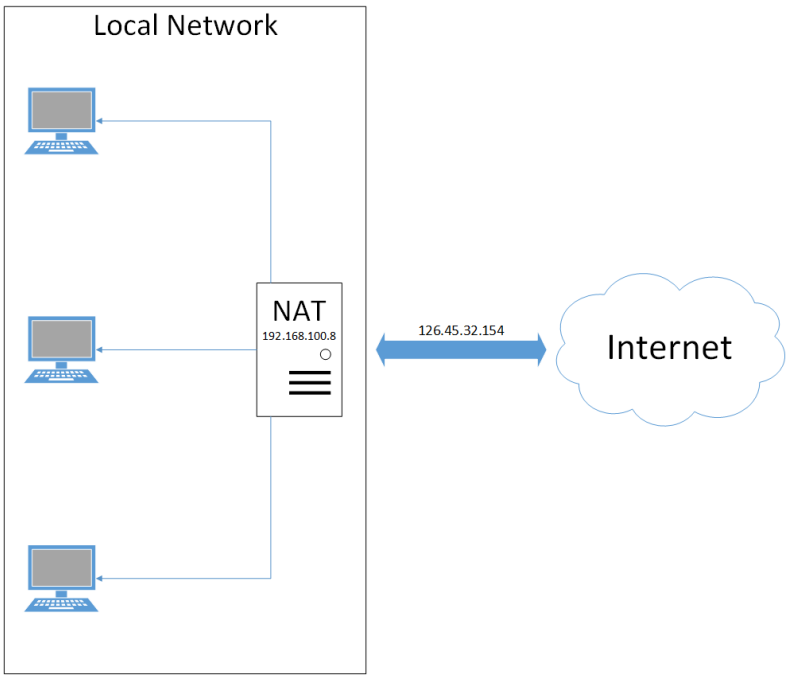

Figure 2.3: Illustrates the function of a NAT

There are a few different NAT configuration categories:

- Open NAT: The Universal Plug & Play (UPnP) complies with minimum restrictions or a device.
- Moderate NAT: minimum limitation, but certain ports or addresses are filtered in the NAT.
- Strict NAT. Ports allocation policy is very limited and many ports and addresses are filtered out.

2.2.7 EventHandlers

EventHandler is a function that detects and handles a particular action, for example, you can add an EventHandler to listen to key presses or mouse clicks. If so, the EventHandler is notified of the action and will deal with it when a key is pressed. In other programming languages, like Java, EventHandlers are often referred to as listeners.

2.2.8 STUN Server

Network Address Translation provides a device with an IP address for use within a private local network, but this address cannot be used externally, and there is no way for WebRTC peers to communicate without a public address. To overcome this problem, WebRTC uses Session Traversal Utilities for NAT (STUN) servers to try and get an external address from a peer.

A STUN server can check the IP and port of an incoming application request that runs after a NAT and return that address as a response to it. The <IP>:<port> can be discovered from a public perspective by WebRTC applications on a STUN server. This allows a pair to receive a publicly available address and then forward it via a signaling server to another pair in order to set up a direct connection [31], see Figure 2.4 to illustrate.

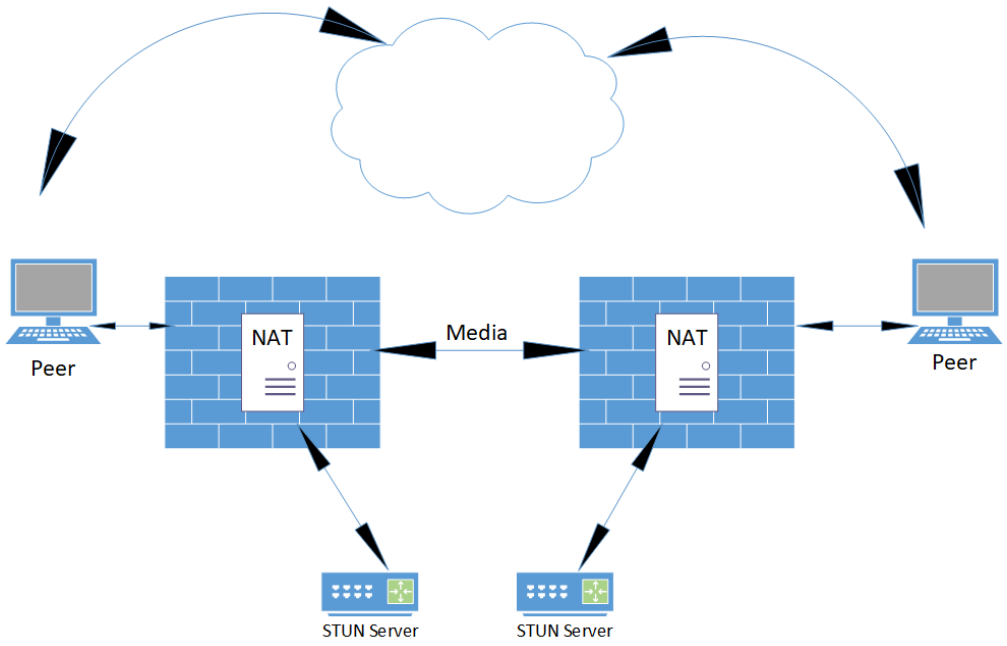

Figure 2.4: Using STUN servers to get public IP: port addresses

2.2.9 TURN Server

The last way to create a P2P connection with WebRTC is by Traversal Using Relays around NAT (TURN) servers. If a host is behind proxies, firewalls or strict NAT's and the STUN cannot get a peer's public IP, then a P2P connection cannot be established. If this happens, instead of the

connection failure, WebRTC will fall back into using TURN servers to relay data between the two hosts.

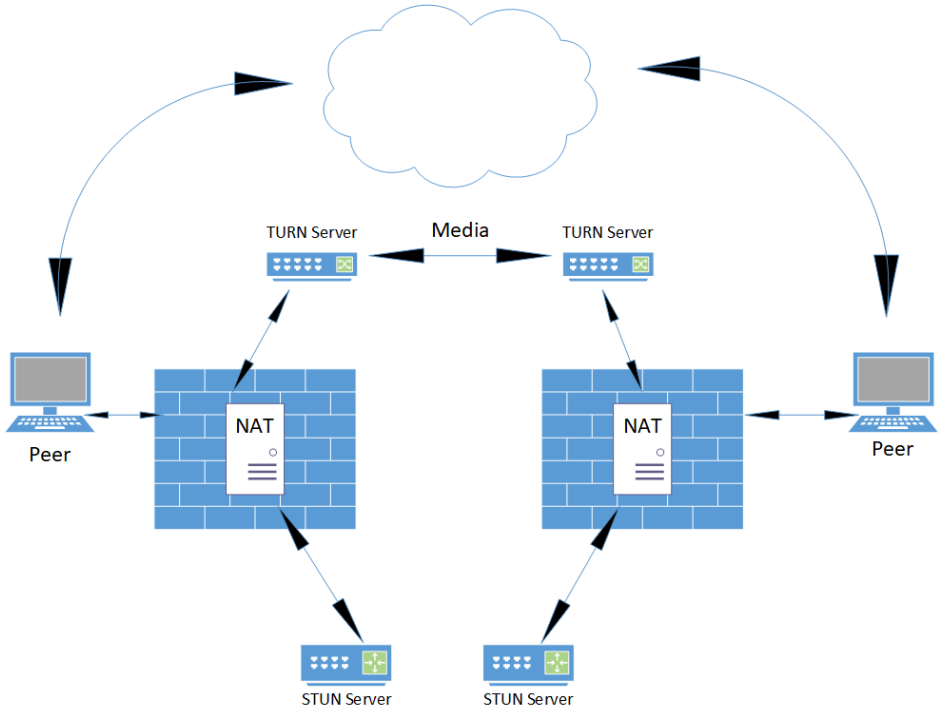

Figure 2.5: TURN in action

According to Webrtcstats [32] and statistics, a company named Bistri published a successful P2P connection in June 2014, 92 percent of all calls. This means that 8 percent of the traffic had to be transmitted via a TURN server.

2.2.10 URL

A Uniform Resource Locator (URL) is a reference to a web resource that sets out a computer network location and a retrieval mechanism. A URL is a specific type of URI [33], though many people use both terms interchangeably. URLs most frequently occur with Web pages referenced (http), but are also used for file transfer (ftp). It will be typically utilized in WebRTC in building a blob URL from an object on MediaStream. In the HTML page, a blob URL shall be used. It is actually necessary for both local and remote streams.

2.2.11 Interactive Connectivity Establishment (ICE)

Interactive Connectivity Establishment (ICE) is a computer-networking framework used to handle NAT, to communicate with interactive media, like Voice over IP (VoIP), instant messaging, and P2P data. Jonathan Rosenberg [34] states that protocols using the offer/answer process have problems crossing NAT and obtaining the correct IP address and peer port.

The WebRTC ICE deployment has two endpoints, known as peers, which want to communicate. Communication can be made indirectly via a signaling server in which SDP messages for offers / answers can be exchanged. At the start of the ICE setup, peers are not familiar with their own topology and do not know whether they are behind one or many NAT's. ICE helps the colleagues to discover sufficient information about their own topology to identify one or more ways of communicating.

2.2.12 Signaling Server

WebRTC supports peer-to-peer communication, but still needs servers for the connectivity complexity caused by NATs and firewalls. In addition, signaling servers are required to exchange metadata used for communication coordination among peers. The WebRTC API does not implement the signaling process, so it is up to the software developer to implement WebRTC for its use. Signaling methods and protocols are, as stated in Sam Dutton [31], not specified under WebRTC standards and do not apply into the API to prevent redundancies and maximize compatibility. The JavaScript Session Establishment Protocol (JSEP) outlines this approach.

JSEP is a protocol that removes the need for browsers to save the state. If browsers had to save the state, the setup would be problematic if signaling data were lost each time a page was reloaded. JSEP pulls the signaling state machine out of the browser and into JavaScript. This removes the browser from the signaling flow, and the only interface required is a way for the application to pass local and remote session descriptions negotiated by the signaling mechanism. The signaling status can instead be stored on the server [35]. JSEP requires peers to exchange offers and responses that include metadata with media information. Figure 2.15, where the caller sends an offer with the metadata as the SDP, provides an illustration of what an offer or response may look like. The offer passes through the server and the callee can then answer or reject the call. If the

callee sends an answer it will set up a WebRTC connection and the peers can start exchanging data.

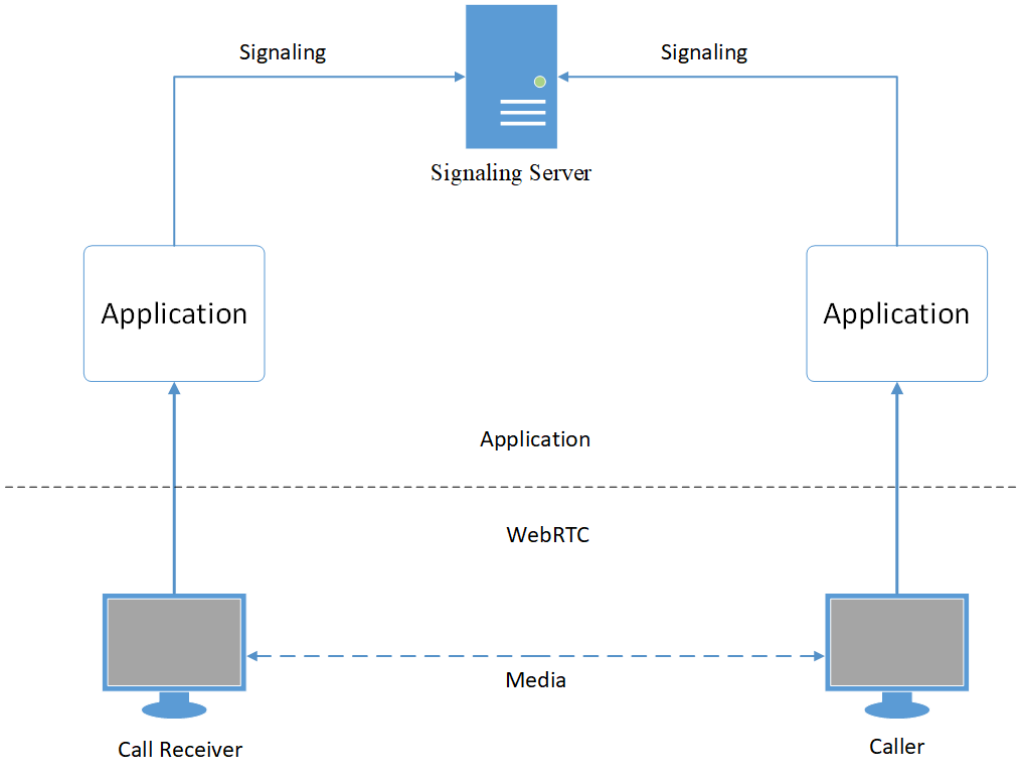

Figure 2.6: JSEP Architecture

Chapter 3: Methodology and Software Implementation

The ultimate goal of this study is to investigate how the quality of WebRTC video conferencing and streaming varies as more and more users join in. This chapter breaks down the implementation details and programming techniques used to build an application that allows a user to set up a conference or a broadcast and share it with clients willing to join in. The application consists of two sections, namely video conferencing and video streaming. While the video conferencing part is limited to four clients, the video streaming part is open to an unlimited number of clients. Moreover, Bootstrap Framework [36] was used to make the design of the application more compelling. This chapter's organization is as follows...

3.1 Overview of WebRTC Application

A simplified overview of the functions implemented in the WebRTC application is shown in Figure 3.1.

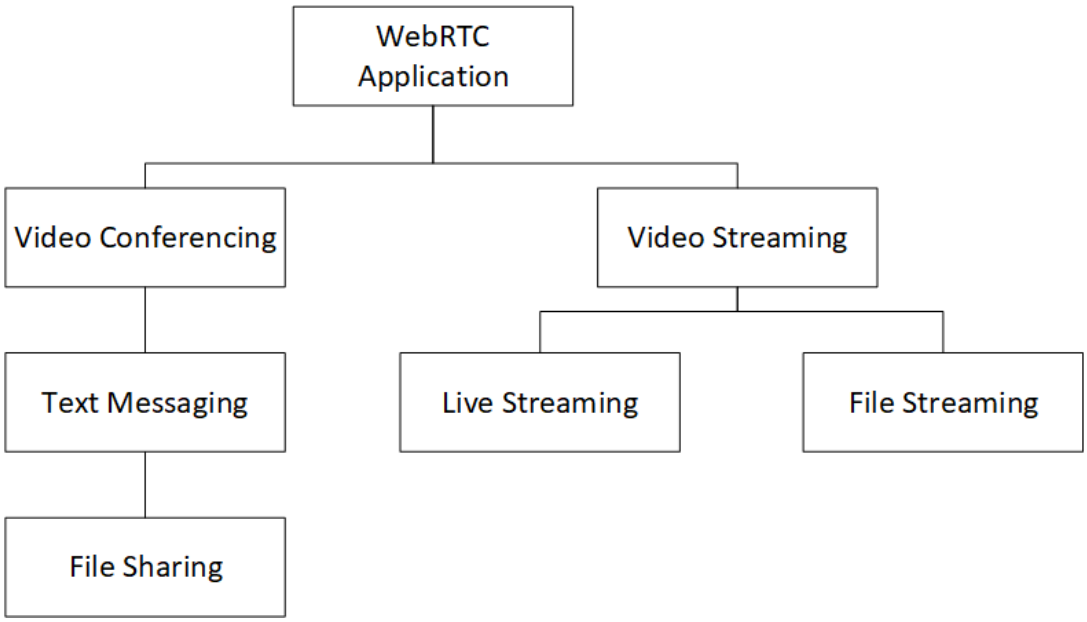

Figure 3.1: Overview of WebRTC application

From the figure above, the application consists of two sections. The first section is video Conferencing and it has a text messaging and file sharing facility

The second section, Video Streaming consists of two further parts: Live Streaming and File Streaming. Live Streaming allows users to share the live preview from the webcam whereas File Streaming allows users to browse a file first before sharing it live as a broadcast.

3.2 Setting up the Signaling Server

The signaling server is the first thing which needs to be set up before any communication begins. Its job is to exchange data between users during the offer/answer negotiations in order to set up a communication path which allows the users to communicate in a peer-to-peer fashion. Setting up our server is as simple as running 'npm start' on Command prompt from the project's folder.

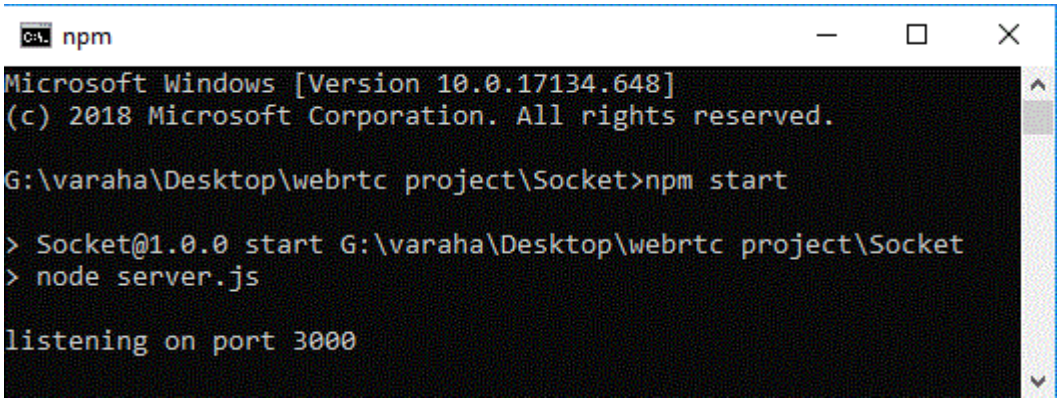

Figure 3.2: Starting the server

As 'npm start' is run, the server is set up and starts listening on port 3000 at initiator's IP address (for e.g. https://192.168.100.8:3000).

Note that https is used instead of http as WebRTC no longer supports sending media over the internet in an unsafe manner [37].

3.3 Home Page

The home page is the first thing displayed upon entering the appropriate URL. It offers two main options namely WebRTC Video Conferencing and WebRTC Video Streaming and consists of two input boxes and four action buttons. Almost all of its functions uses basic JavaScript and HTML codes for their jobs are only to read the data into the input box and redirecting the user to the appropriate room. The only exception is the 'Choose File' option which uses a simple library [38] called 'socket.io-file-client' for its job is to browse a video file from the user's device for broadcasting. Figure 3.3 shows how the homepage looks and specifies the uses of its components.

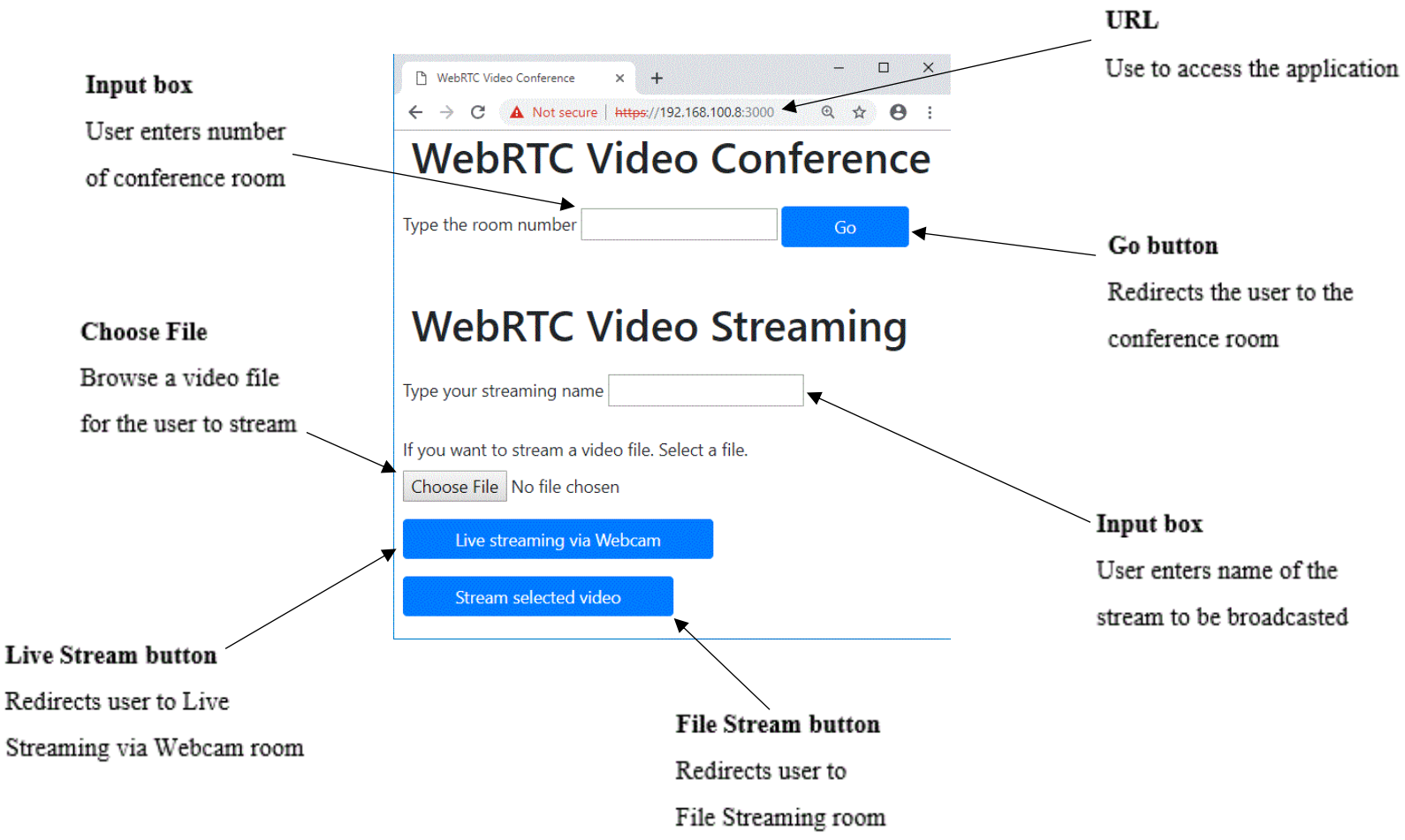

Figure 3.3: Home Page functions

3.3.1 Code structure of Home Page

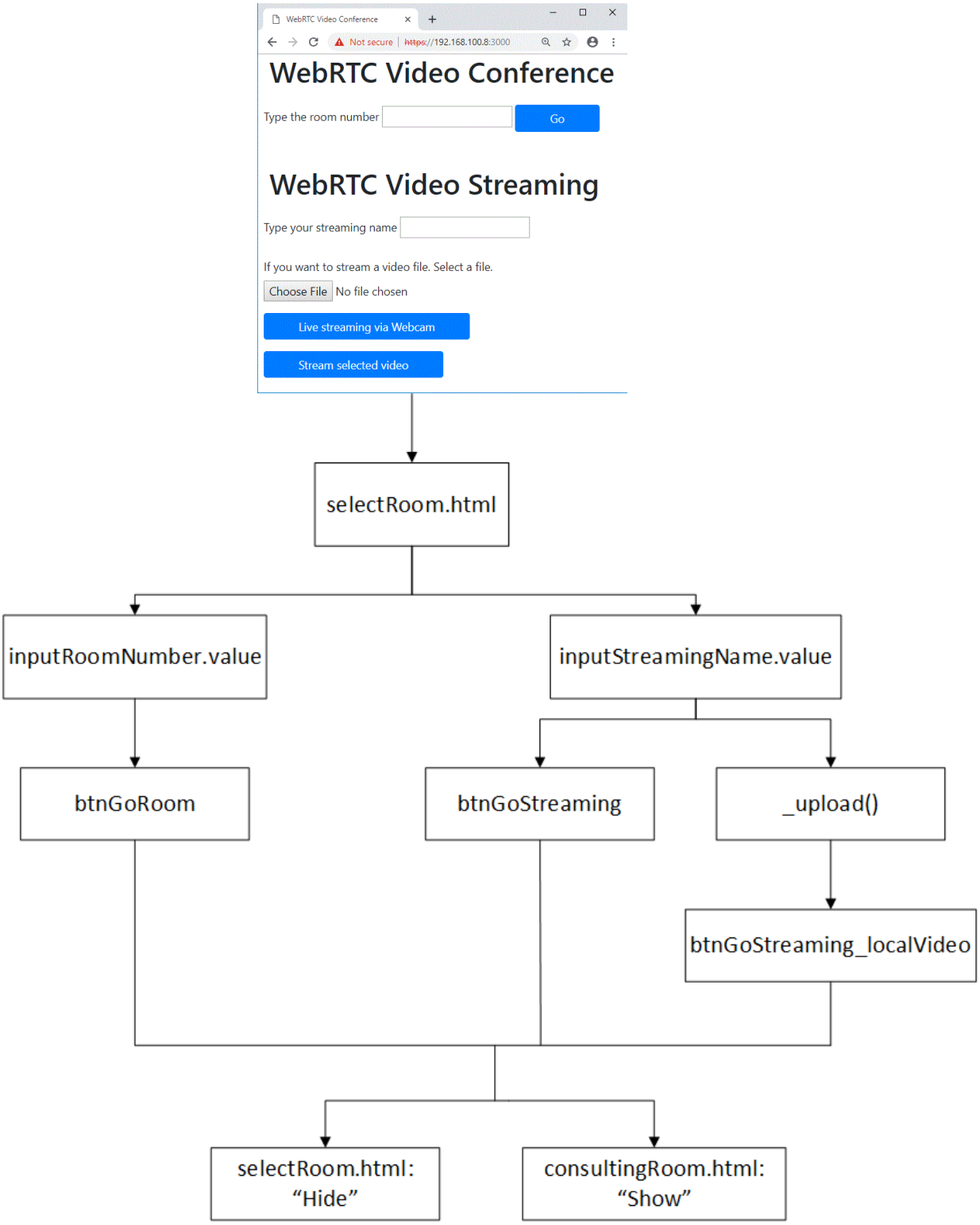

Figure 3.4: Home page code structure

3.4 Video Conferencing Application

3.4.1 Process of WebRTC Video Conferencing

The process of WebRTC video conferencing is shown in Figure 3.5.

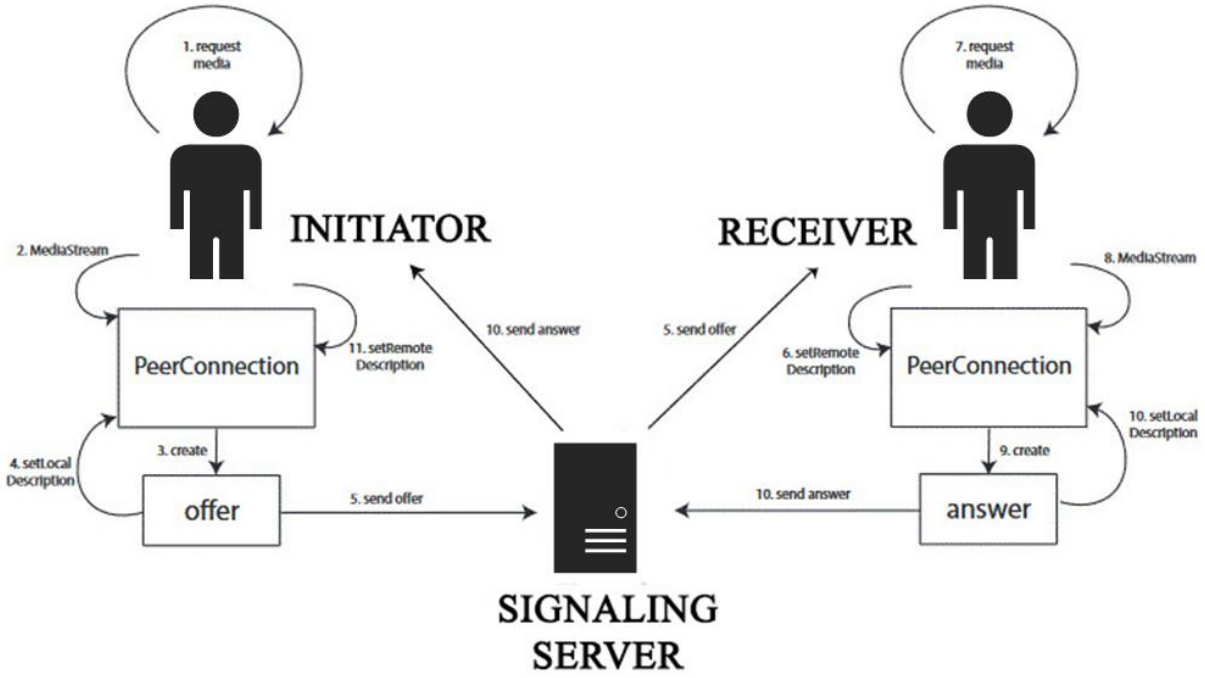

Figure 3.5: Negotiation process of WebRTC

The steps in Figure 3.5 are described below:

1. The initiator does a getUserMedia() request to get access to its audio/video MediaStream and adds it to a new RTCPeerConnection
2. An (SDP) offer is created using the generated RTCPeerConnection
3. This offer is set as the peer connection's localDescription and sent to the receiver via the signaling server
4. Upon receiving the offer, the receiver creates a RTCPeerConnection using its own MediaStream and adds the received offer as its remoteDescription
5. The receiver creates an (SDP) answer from its RTCPeerConnection, adds this as its localDescription and sends this answer to the initiator via the signaling server
6. The initiator receives the answer and sets it as its remoteDescription

Now that both peers have exchanged each other's SDPs and negotiated the settings and type of streams to use, a direct connection between both peers can be set up.

However, the figure above shows how connection is made between two users only though the implemented application can accommodate up to four users. In order to achieve a WebRTC group communication, full mesh [39] topology is used.

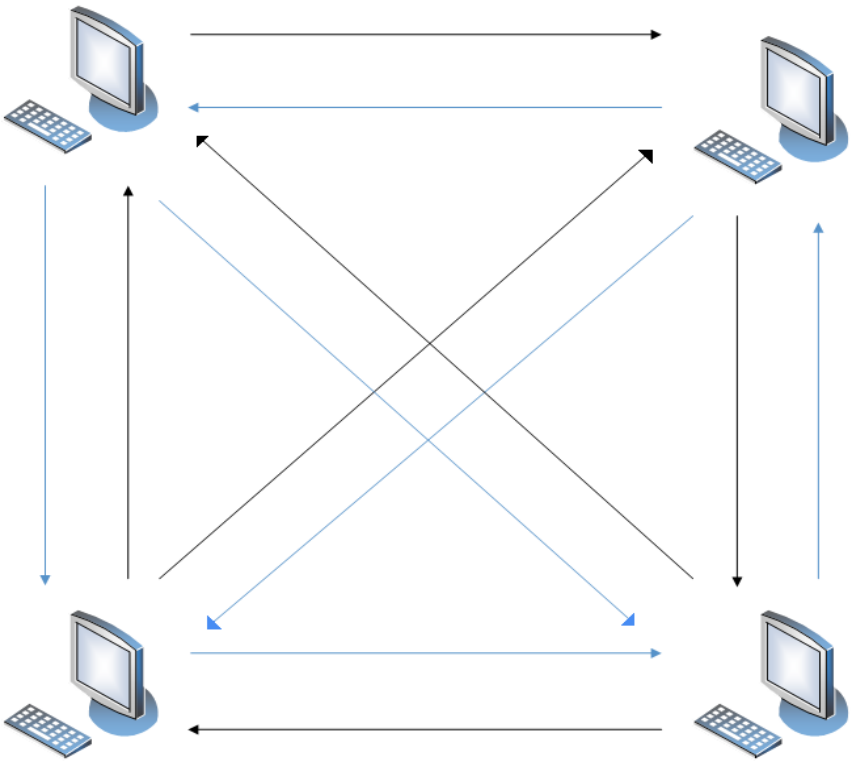

Figure 3.6: Full Mesh Topology

In the full mesh topology, every user is connected individually to the other users in the conference. Even though the initiator is the one who sets up the conference, the latter does not become the main server through which the other clients must go in order to connect among themselves. As shown in Figure 3.6, there are two arrows between every two users thus, showing the peer-to-peer nature of the connection.

3.4.2 Conference Room

After clicking on the ‘Go’ button on the homepage upon entering a room number, the conference room is the first thing presented to the user. It consists of the local and remote videos, recorders that are attached just below the videos, a live messaging option and a file sharing option. Live messaging allows users to share text messages and includes a chat box, two input boxes and a ‘Send’ button. The first input box allows the users to enter their names and the second input box provides space for typing the text message. The file sharing option allows users to share any type of files and consists of a ‘Choose Files’ button which allows the users to choose files and an ‘Upload’ button which allows users to upload the chosen files.

3.4.2.1 Single Client Video Conferencing

Single client video conferencing is where only one client, excluding the initiator joins in a video conference. In other words, it illustrates communication between two users only as shown in Figure 3.7.

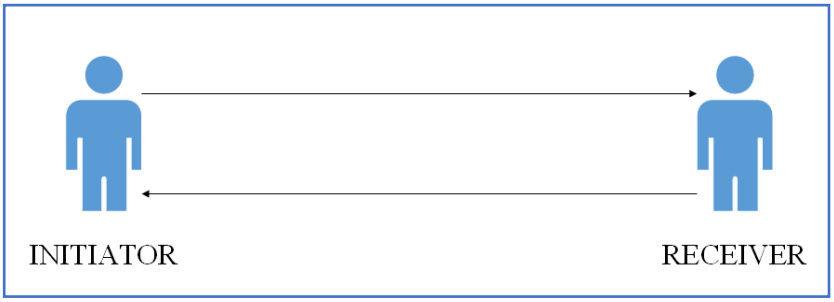

Figure 3.7: Single Client connection

Figure 3.8 shows the conference room on the initiator, having a conference with only the receiver.

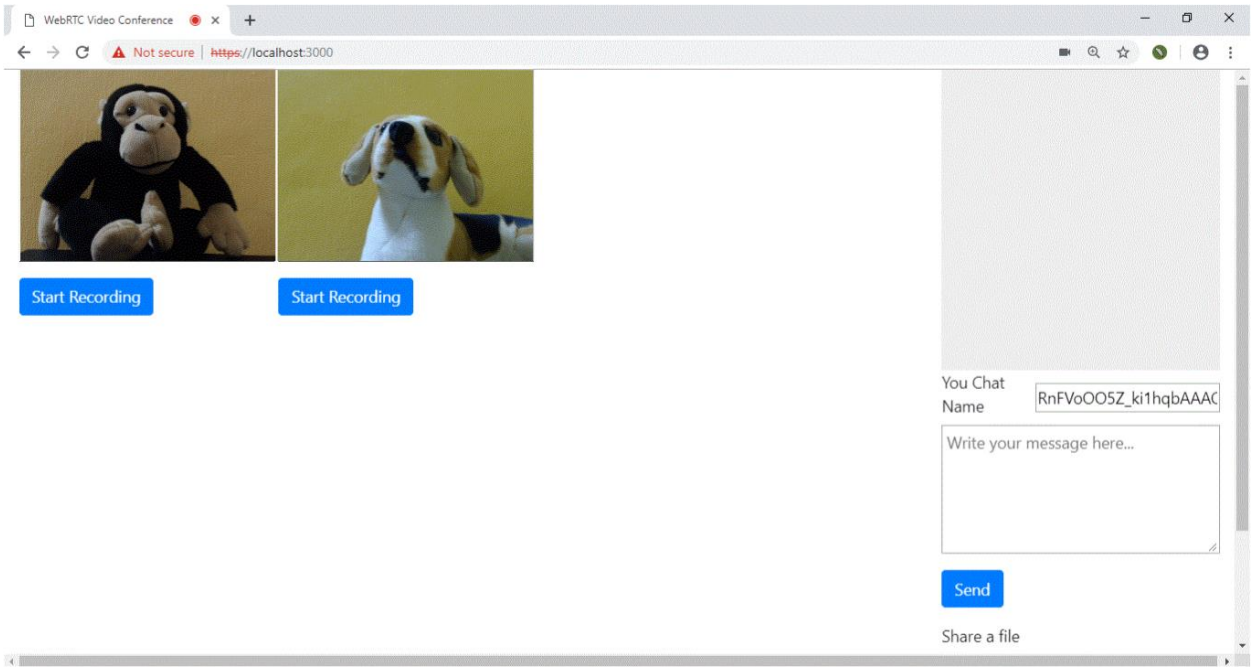

Figure 3.8: Single Client conference

3.4.2.2 Multi-Client Video Conferencing

Multi-client video conferencing is where more than one client, excluding the initiator joins a video conference. Figure 3.9 illustrates communication among four users.

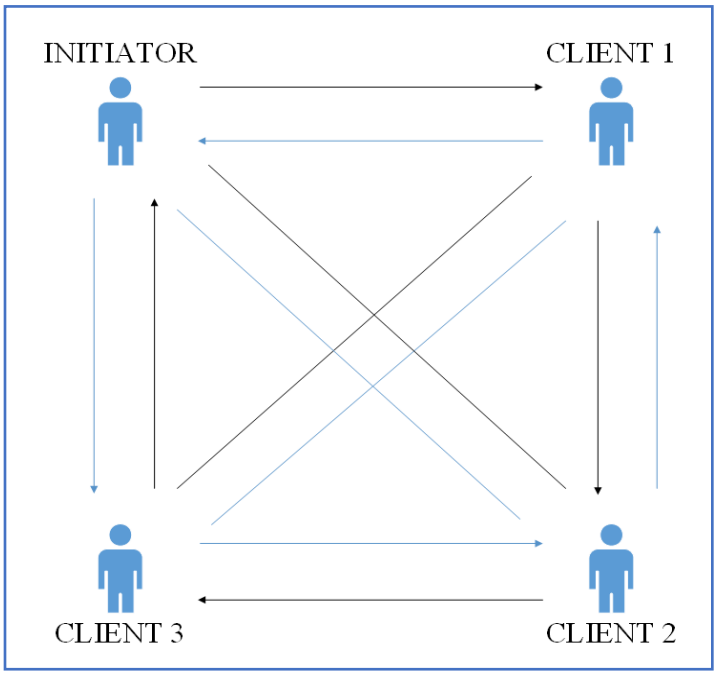

Figure 3.9: Multi-Client connection

Figure 3.10 shows the conference room of the initiator after a second client joined in.

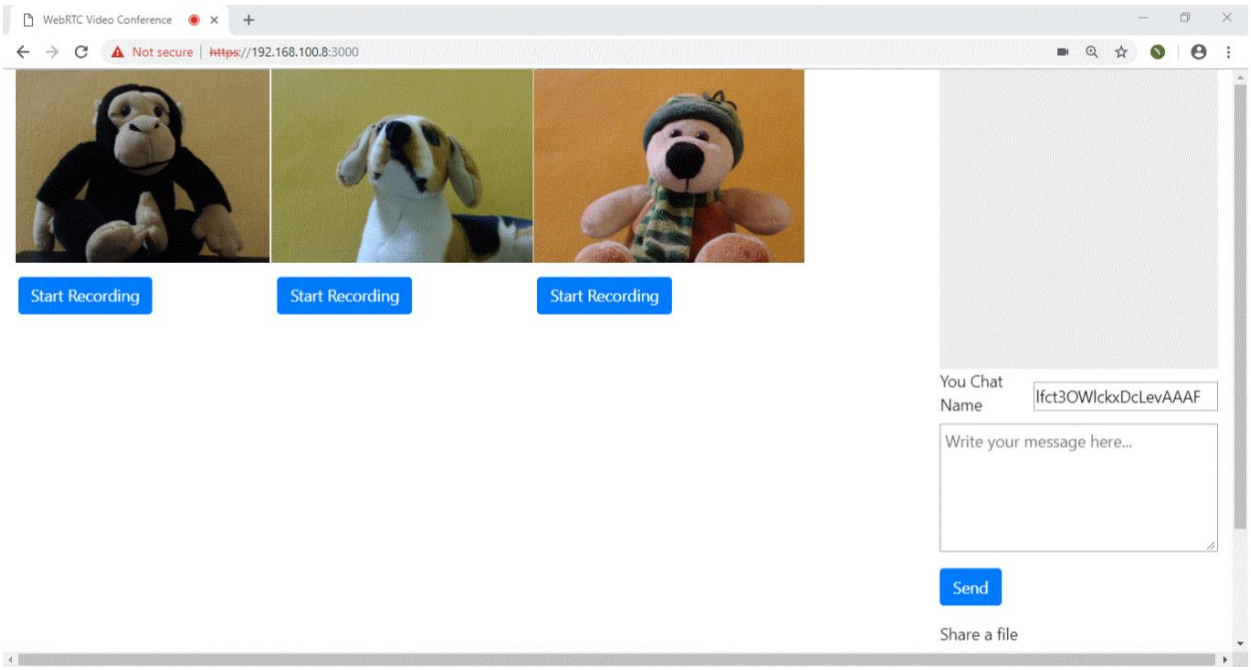

Figure 3.10: Multi-User conference (3 users)

Figure 3.11 shows the conference room of the initiator after a third client joined in.

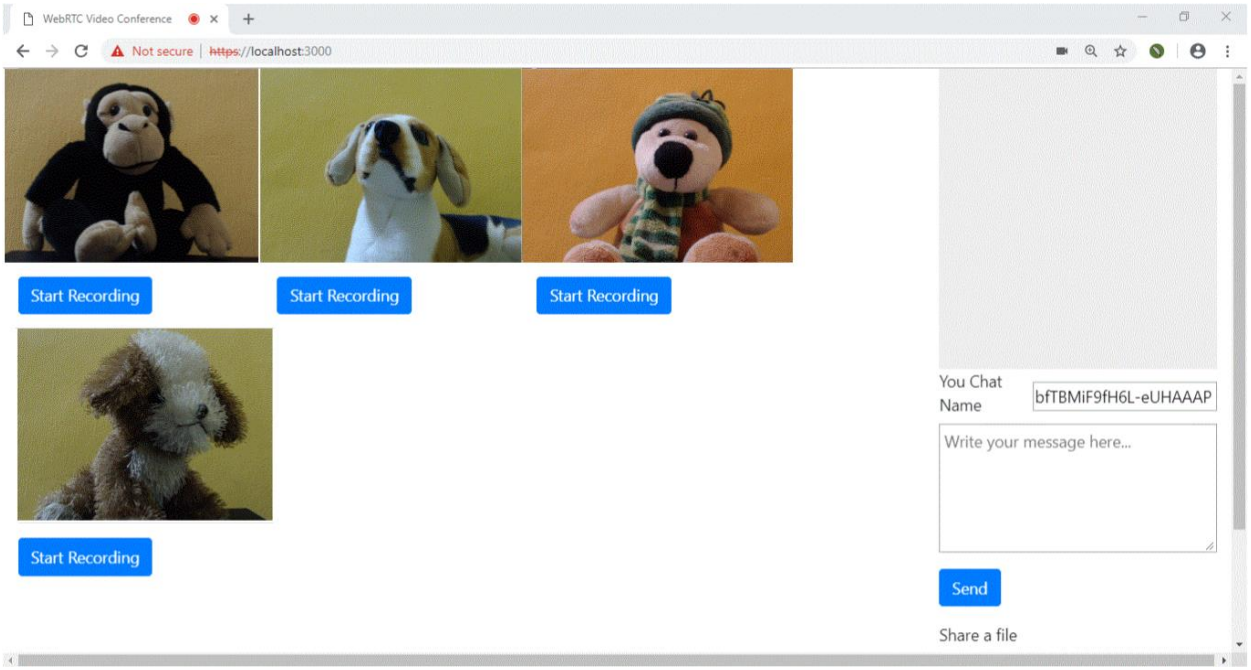

Figure 3.11: Multi-User conference (4 users)

3.4.3 Structure of Video Conferencing Application

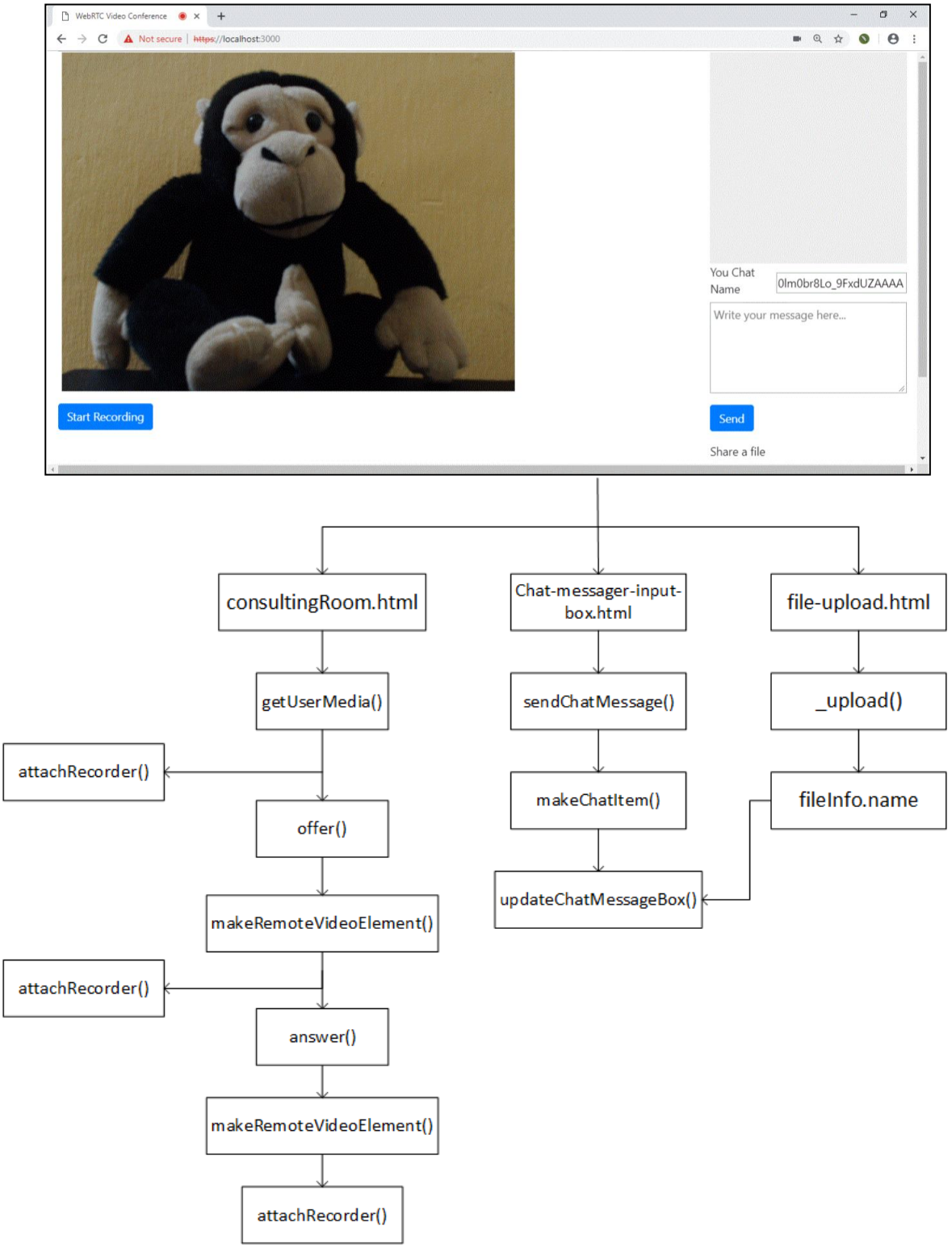

Figure 3.12: Structure of Video Conferencing Application

3.4.4 Getting the local media

This block requests the user's local video as soon as the latter enter the room. One of the main APIs in WebRTC namely, Mediastream is used for accessing the local camera. The function used namely, getUserMedia() can only be called from an HTTPS URL, localhost or a file://URL. Otherwise, the promise from the call will be rejected and getUserMedia() also won't work for cross origin calls from iframes [40]

After accessing the local camera, the initiator then shares the URL and the room number of the conference to a client, namely the receiver, who is then redirected to the homepage of the application. On the homepage, the receiver enters the room number provided and after joining the room, the preview of the local video of the receiver is displayed.

3.4.5 Making an offer

In order for the video of the initiator to appear on the page of the receiver, a negotiation process takes place which included sending on offer to the initiator. The receiver executes the next block namely, offer() which does the following job:

1. Creates a peer connection object for initiator
2. Adds the onicecandidate event handler to peer connection object
3. Sends ice candidates to initiator
4. Adds local stream to peer connection object,
5. Creates offer
6. Ties local description to peer connection object
7. Sends offer to initiator

3.4.6 Making an answer

Similarly to the previous section, a negotiation process takes place for the local video of the receiver to appear on the initiator's page. The next block namely answer(), executes the following algorithm:

1. Receives offer from receiver
2. Creates a peer connection object for receiver
3. Sets remote description of receiver to peer connection
4. Adds the onicecandidate event handler to peer connection object
5. Stores ice candidates of receiver in peer connection object
6. Adds video of receiver to peer connection object by triggering the onaddstream event handler
7. Displays video of receiver using makeRemoteVideoElement()
8. Attaches recorder to received video using attachRecorder()
9. Sends ice candidates to receiver
10. Adds local stream to peer connection object
11. Creates answer
12. Ties local description to peer connection object
13. Sends answer to receiver
14. Sets remote description of initiator to peer connection object
15. Stores ice candidates in peer connection object
16. Adds video of initiator to peer connection object by triggering the onaddstream event handler
17. Displays video of initiator using makeRemoteVideoElement()
18. Attaches recorder to received video using attachRecorder()

3.4.7 Getting the remote media

Using the makeRemoteVideoElement() function, this next block helps in displaying the remote video i.e. the local video of the other peer. In addition to displaying this remote video, it also shapes the size in which the video will appear. These steps are described below:

1. Set width to zero
2. Set height to zero
3. if the video conference application is being used.
4. width = (width of window / 2) -50
5. height = (height of room /2) – 50
6. end if
7. Display remote video with the calculated attributes

3.4.8 Recording the streams

This block is responsible for recording any video displayed in the conference room. Recording of streams proves to be a key part of the project since the local videos will be compared against their corresponding remote videos in order to assess the fluctuation in quality. Recording streams requires no help from the signaling server however, it does need a library namely 'RecordRTC.min' [41]. The recording button is generated by the 'RecordRTC.min' library and it is represented by the attachRecorder() function which is executed when the user:

1. Clicks on 'Start Recording' button
2. Recorder sets maximum recording time, maxRecordingTime to 300000ms (5 minutes)
3. Recording starts
4. Hide 'Start Recording' button
5. Display 'Stop Recording' button
6. if maxRecordingTime == 300000
    a. Recording stops
    b. Hide 'Stop Recording' button
    c. Display 'Download Recording' button
    d. Click on 'Download Recording' button

e. Downloading starts

f. Hide ‘Download Recording’ button

g. Display ‘Start Recording’ button

7. else

a. Click on ‘Stop Recording’ button

h. Recording stops

b. Downloading starts

c. Hide ‘Stop Recording’ button,

d. Display ‘Start Recording’ button

The recording process consists of three phases: start recording, stop recording, and download recording as shown in Figure 3.13.

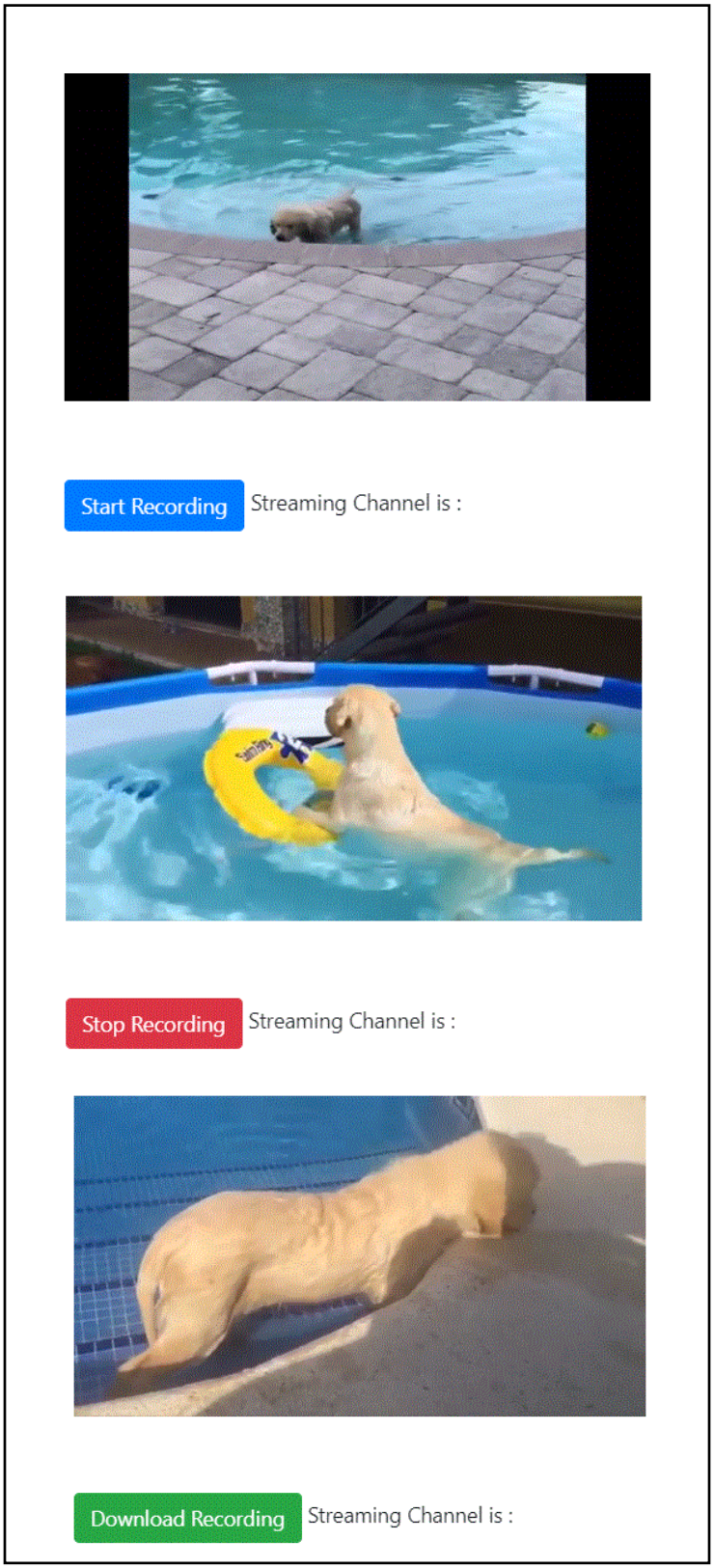

Figure 3.13: Phases of recording

3.4.9 Text Messaging and File Sharing System

The text messaging and file sharing system allow users connected to the room to exchange text messages among themselves and also share any type of file by uploading it, as a result generating a download link which is displayed to user's chat box.

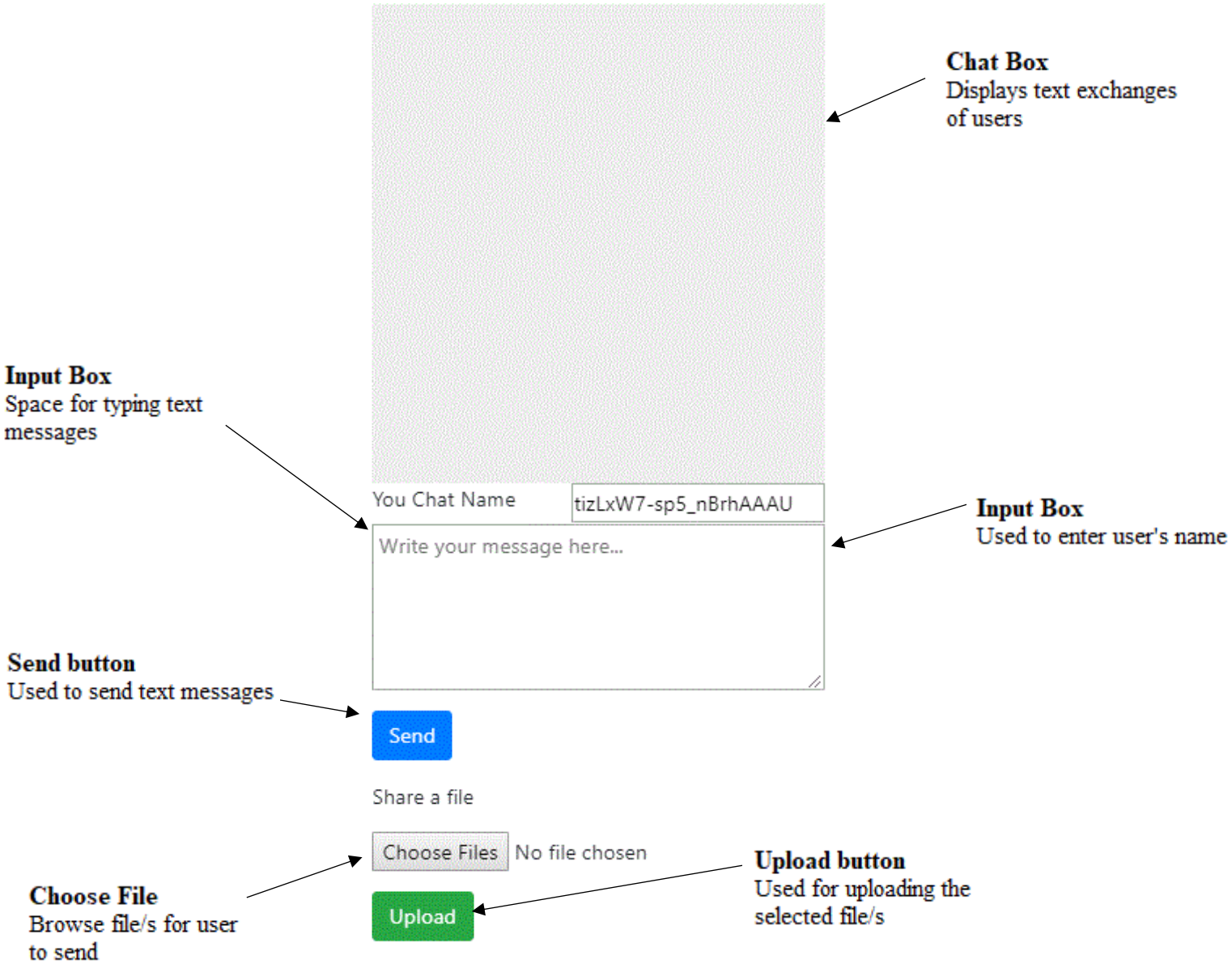

Figure 3.14: Text Messaging and File Sharing System

3.4.9.1 Sending the message

This block contains the main function for sending the text message to all connected clients. The steps are as followed:

1. Get the text message, chatMsg
2. if chatMsg = “”
3. alert “Please enter a chat message...”
4. Get the text message, chatMsg
5. else
6. Send chatMsg to all other users
7. Update chat box using updateChatBox() function

3.4.9.2 Making the Chat Item

This block, driven by the makeChatItem() function provides an HTML template for the chat message. Its job is simply to get the text message and the name of the user who sent the message and display them as shown in Figure 3.15.

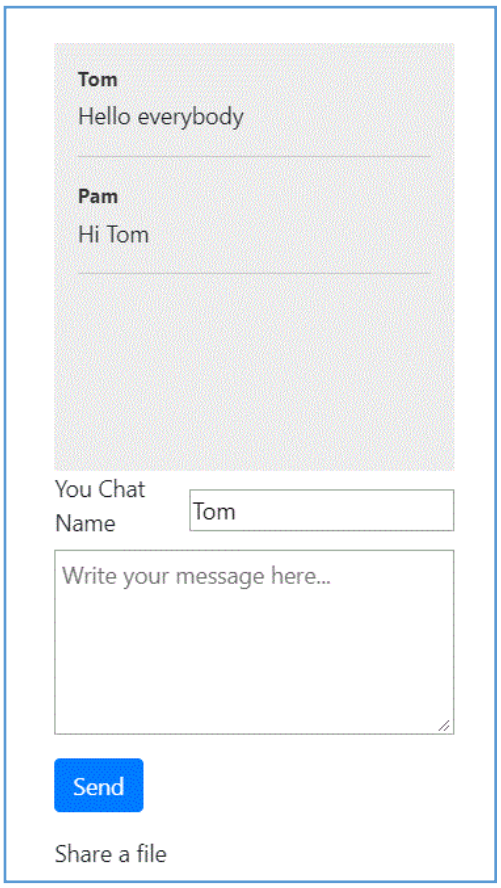

Figure 3.15: Text message preview

3.4.9.3 Update Chat Box

Since the text messaging system is in a real-time system, it needs to be updated all the time. The updateChatMessageBox() function proves to be very crucial at maintaining real-time communication between users and its algorithms are as follows:

1. Make the chat item using makeChatItem()
2. Find the right space, just below the last message to insert the new message

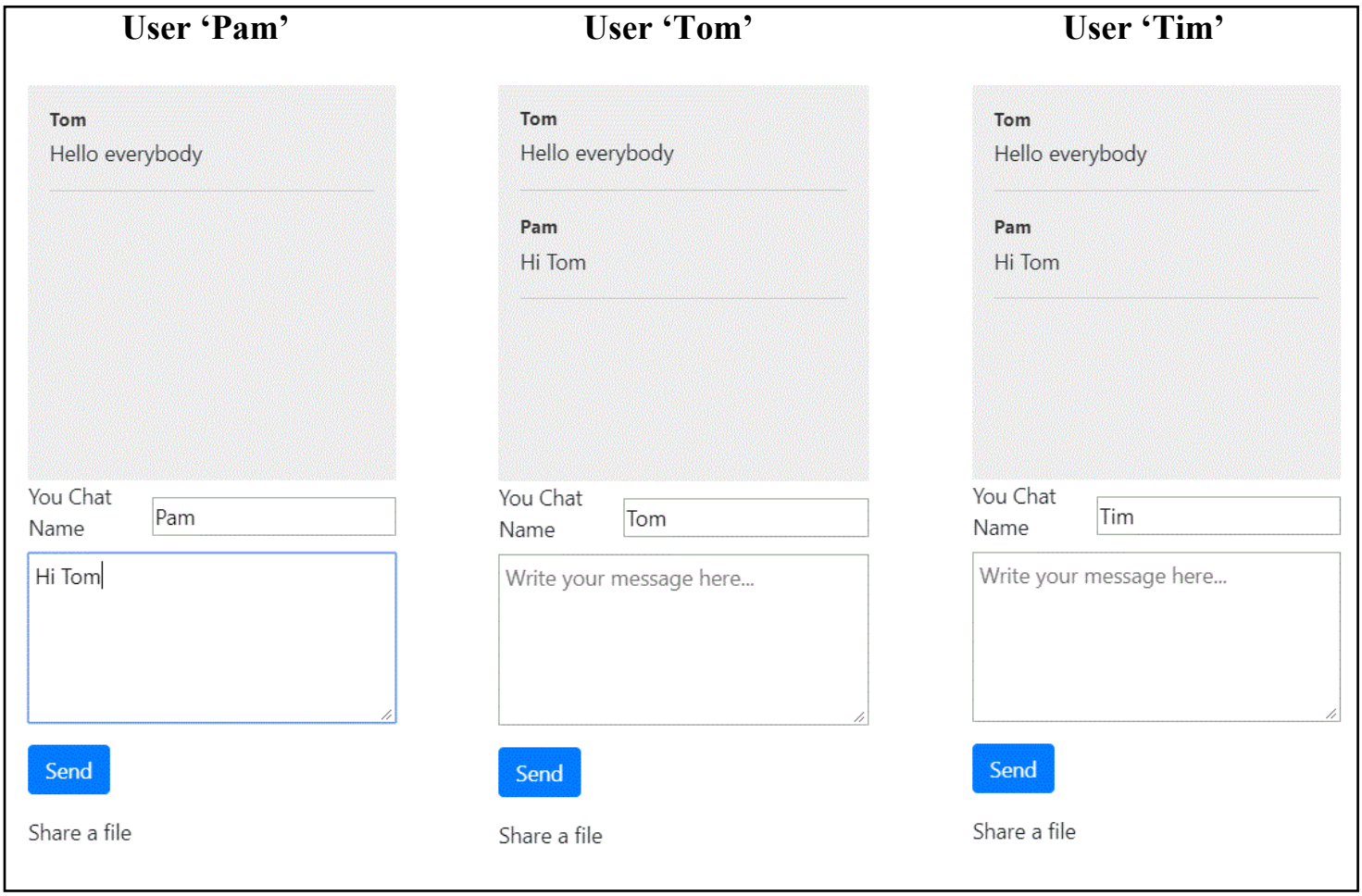

Figure 3.16: Text messaging among three users

Figure 3.16 illustrates user ‘Pam’ receiving a message sent by user ‘Tom’ and sending a message to the other users. Consequently, user ‘Tom’ receives the message from user ‘Pam’. User ‘Tim’ who also was connected to the room receives the message from the first two users.

3.4.9.4 File Upload

File upload uses a library called ‘socket.io-file-client’ [42], which provides the ‘Choose File’ button. This button is driven by the _upload() function and allows users to browse one or several files at a time which will then be shared on as a link on the chat box. Figure 3.17 shows a document file namely ‘roster.docx’ being uploaded and shared by user ‘Tom’

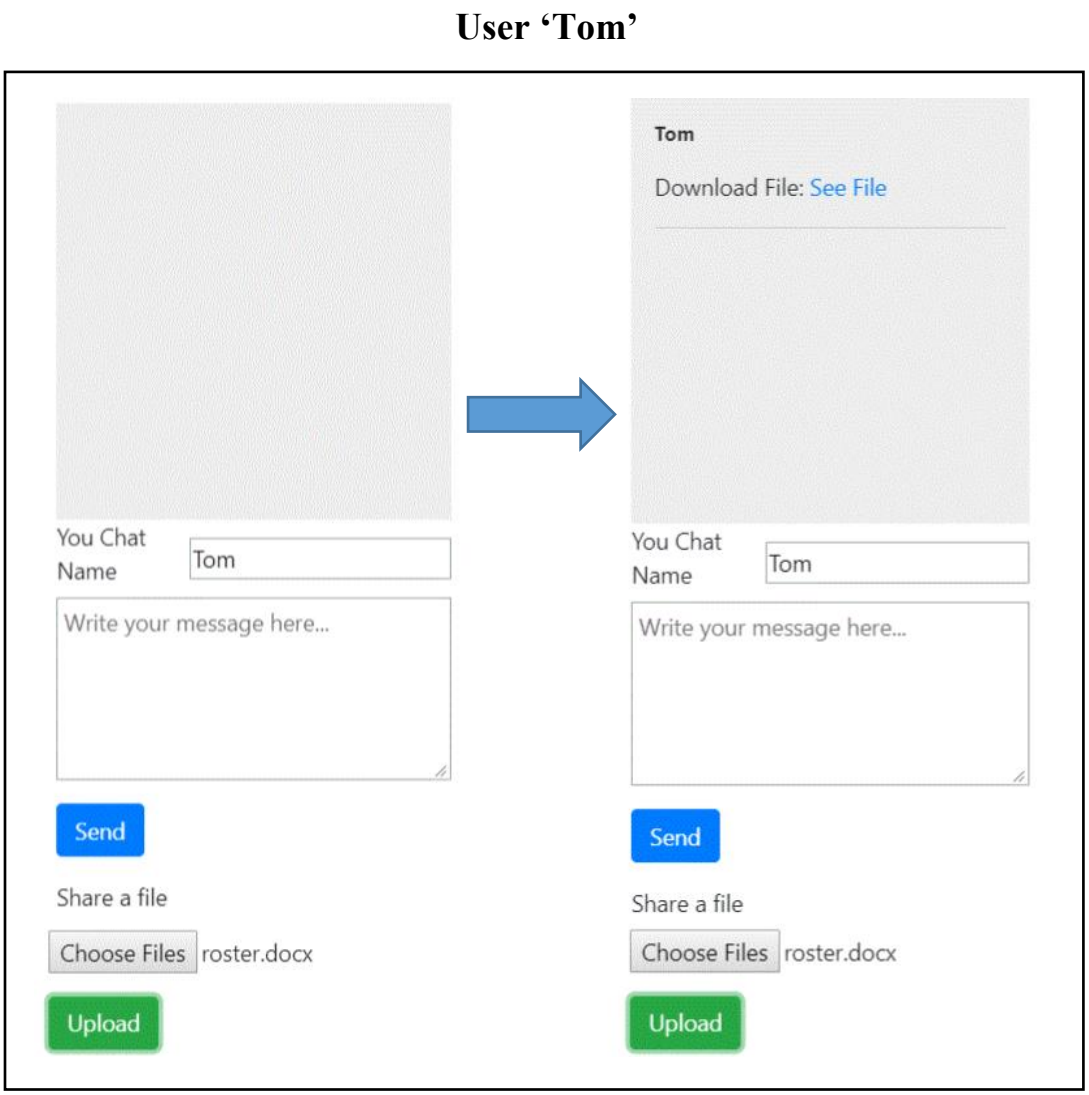

Figure 3.17: Chat box of User 1 (Tom) with selected file

Figure 3.18 shows the file being received by user ‘Pam’ who in turn sent a pdf file namely ‘slide.pdf’. The file was successfully received by user ‘Tom’

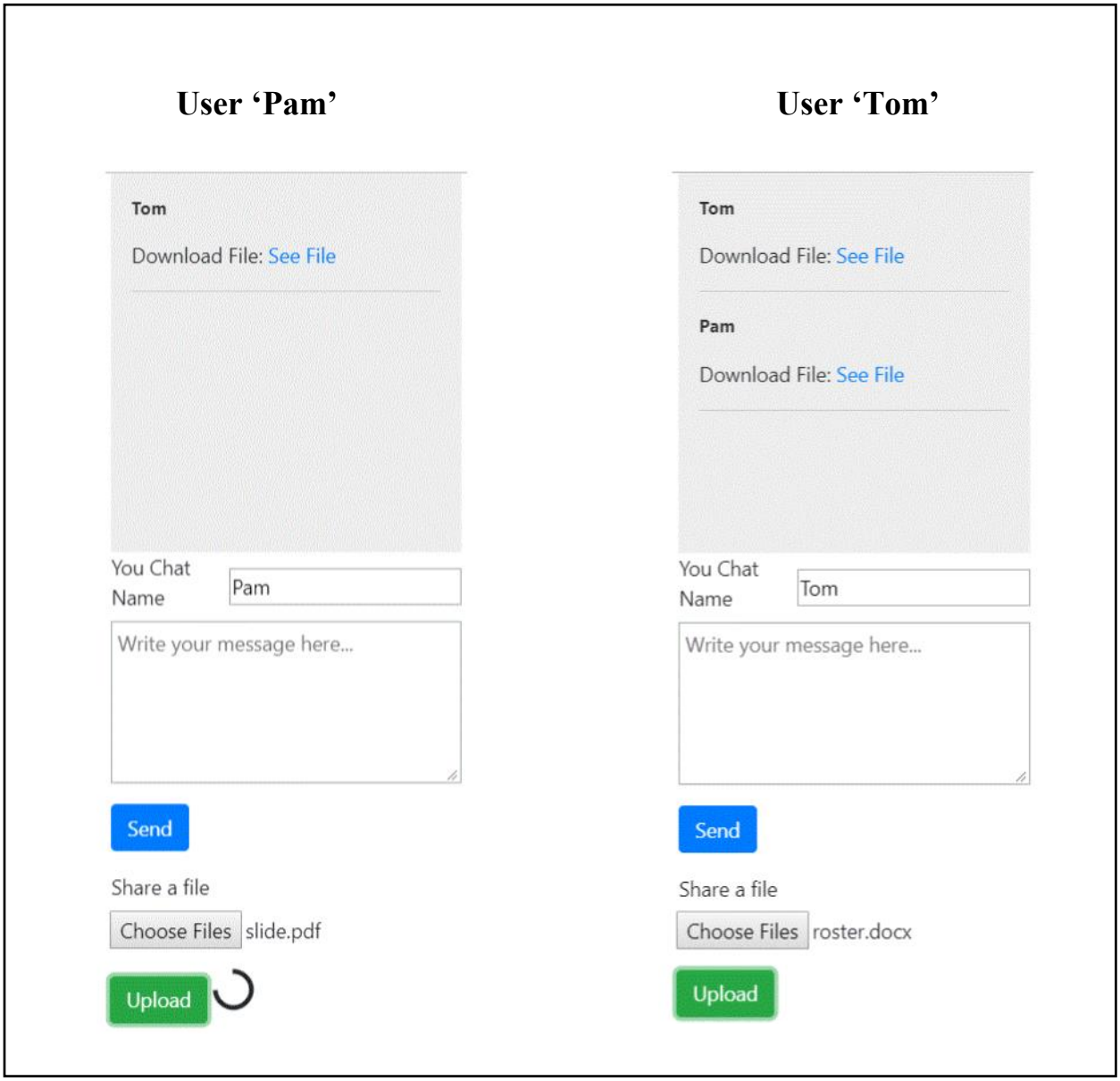

Figure 3.18: File sharing between two users

In addition, files can be shared a couple at a time by browsing all the files together. Figure 3.19 illustrates how user 'Pam' shares three files at a time.

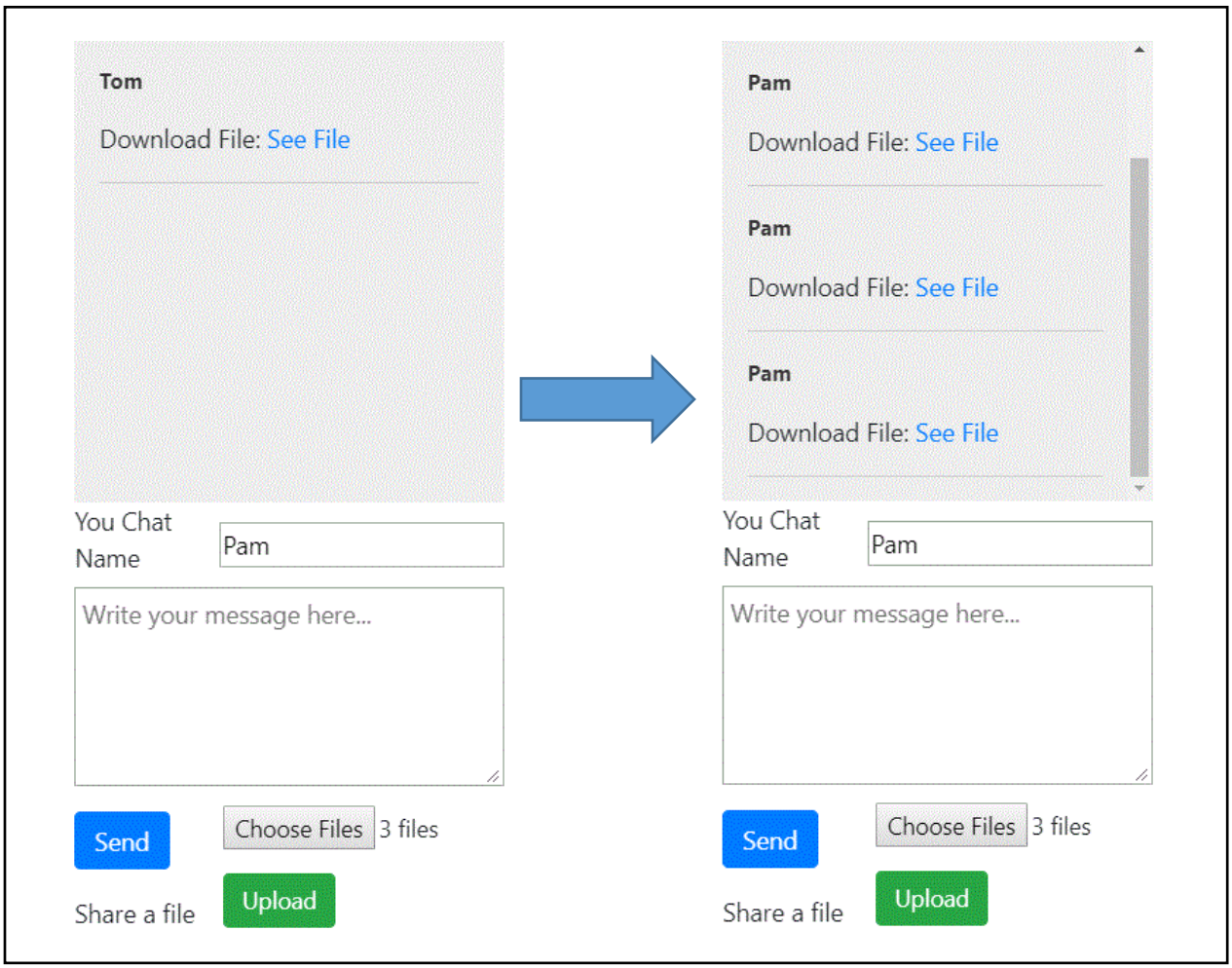

Figure 3.19: Multiple file sharing

3.5 Video Streaming Application

3.5.1 General Structure for WebRTC Video Streaming

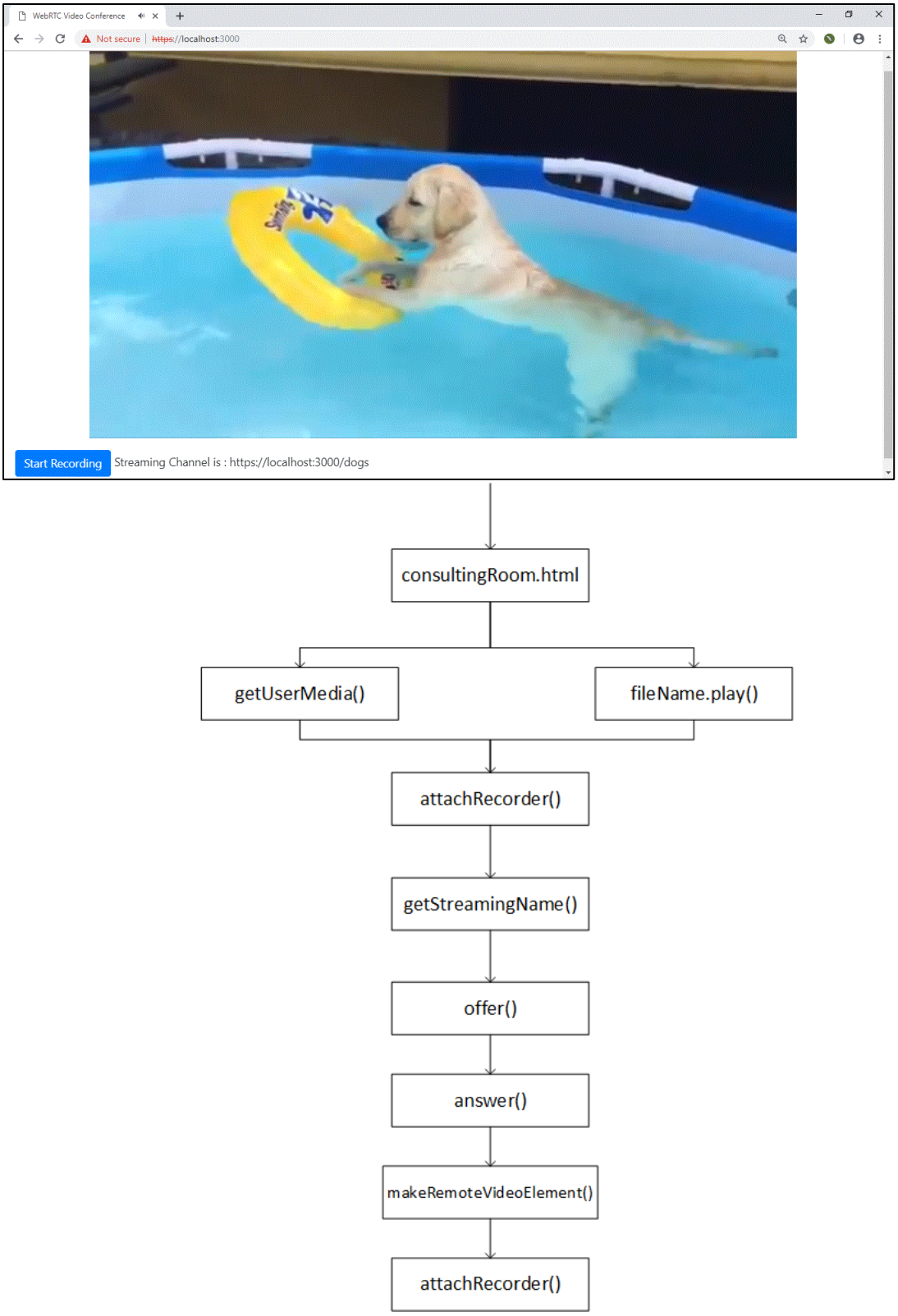

Figure 3.20: Structure of WebRTC Video Streaming

The steps in Figure 3.20 are as follows:

1. The initiator does a getUserMedia() request or selects a local video to access its video MediaStream and adds it to a new RTCPeerConnection object
2. The client creates an (SDP) offer using the generated RTCPeerConnection
3. The offer is set as the peer connection's localDescription and sent to the client via the signaling server
4. Upon receiving the offer, the initiator creates a RTCPeerConnection and adds it as a remoteDescription
5. The initiator creates an (SDP) answer from its RTCPeerConnection, adds this as its localDescription and sends this answer to the client via the signaling server
6. The client receives the answer and sets it as its remoteDescription and gets the video of the initiator

Now that both peers have exchanged each other's SDP's, a direct connection between both peers can be set up. Conversely to video conferencing, the media will be transmitted in only one way i.e. only from the initiator to the client.

3.5.2 Streaming Room

The first thing to do before setting up a broadcast is to enter a stream name on the homepage. Afterwards, a broadcast can be set up by either Live Streaming or File Streaming. Live Streaming allows users to broadcast their live preview from their webcam and it can be done by simply clicking on the 'Live Streaming via Webcam' button. File Streaming on the other hand requires the user to browse a video file first via the 'Choose File' button and broadcast it by clicking on the 'Stream selected video' button. After selecting one of the two options, the user is presented with the streaming room which consists of the selected video or the webcam preview, a recorder and also a link which redirects clients to the broadcast directly.

3.5.3 Generate a streaming link

As mentioned above, a specific link is required for clients to join the stream directly. This block, contained by getStreamingName() does the following:

1. Get the streaming name
2. Add a '/' at the end of the URL
3. Add the streaming name just after the '/' to form the link

3.5.4 Multi-Client connection

In order to set up communication among various users, a star topology [43], as shown in Figure 3.21 is used.

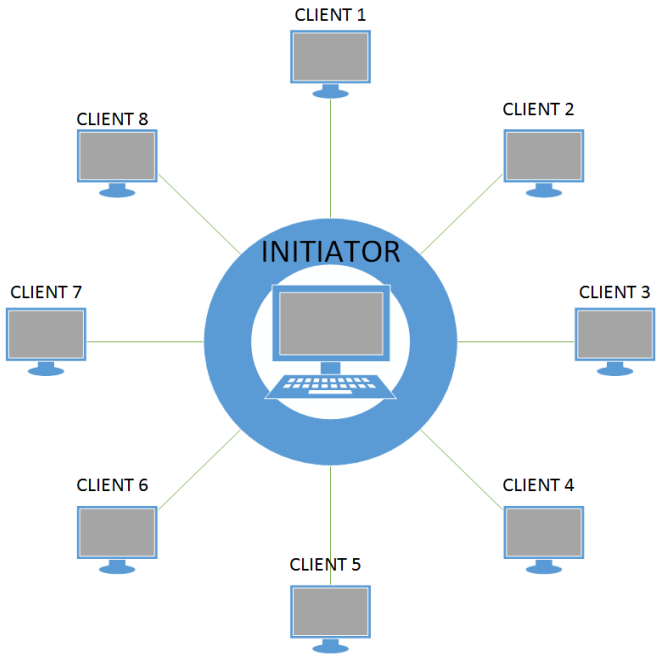

Figure 3.21: Star Topology

In the star topology, all clients are connected to the initiator who acts like a server in that case. After the offer/answer negotiations are completed, individual peer connections are created between the initiator and the clients.

3.5.5 Live Streaming

Live streaming provided users the facility to share the preview of their local video and it is very similar to video conferencing since both uses the local preview from the local camera. The only difference is that video transfer is unidirectional with video streaming and bi-directional with video conferencing. Figure 3.22 shows the streaming room of the initiator.

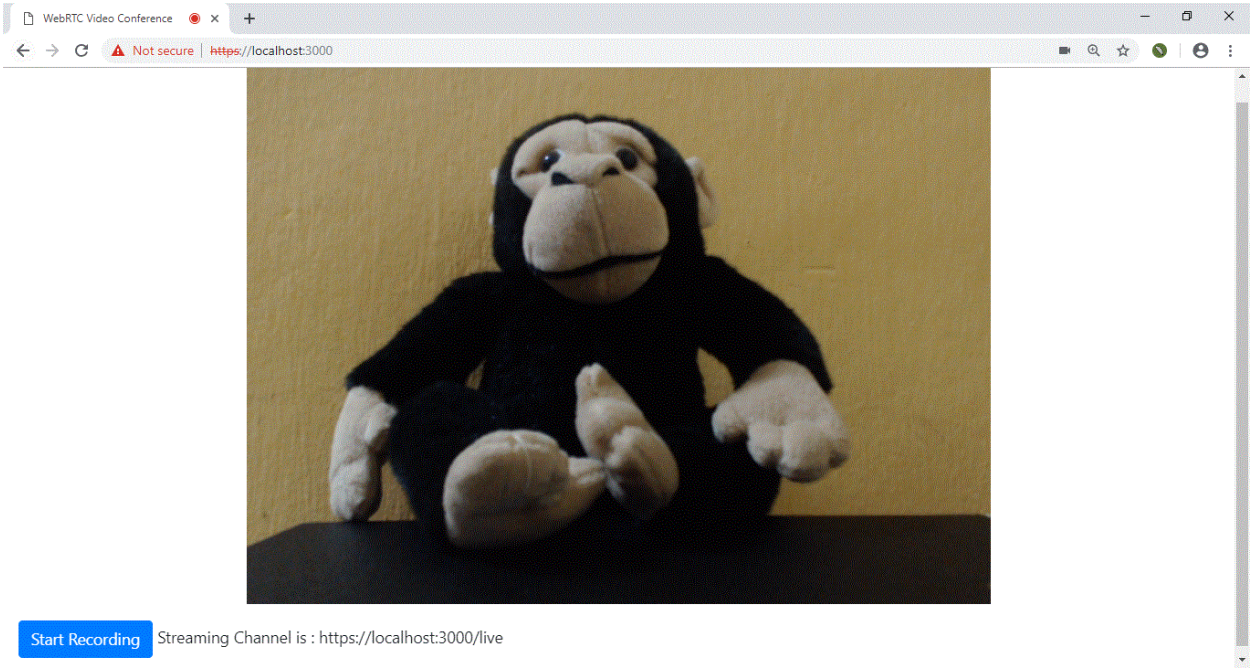

Figure 3.22: Streaming room of initiator (Live Stream)

The initiator shares to the clients the link at the bottom of the screen which is generated by adding '/streamingName' at the end of the URL of the initiator as mentioned by function getStreamingName(). However, the initiator must also share the IP address, which should replace 'localhost' thus altering the link to https://<IP_address>:3000/live. Figure 3.23 shows the streaming room of the client.

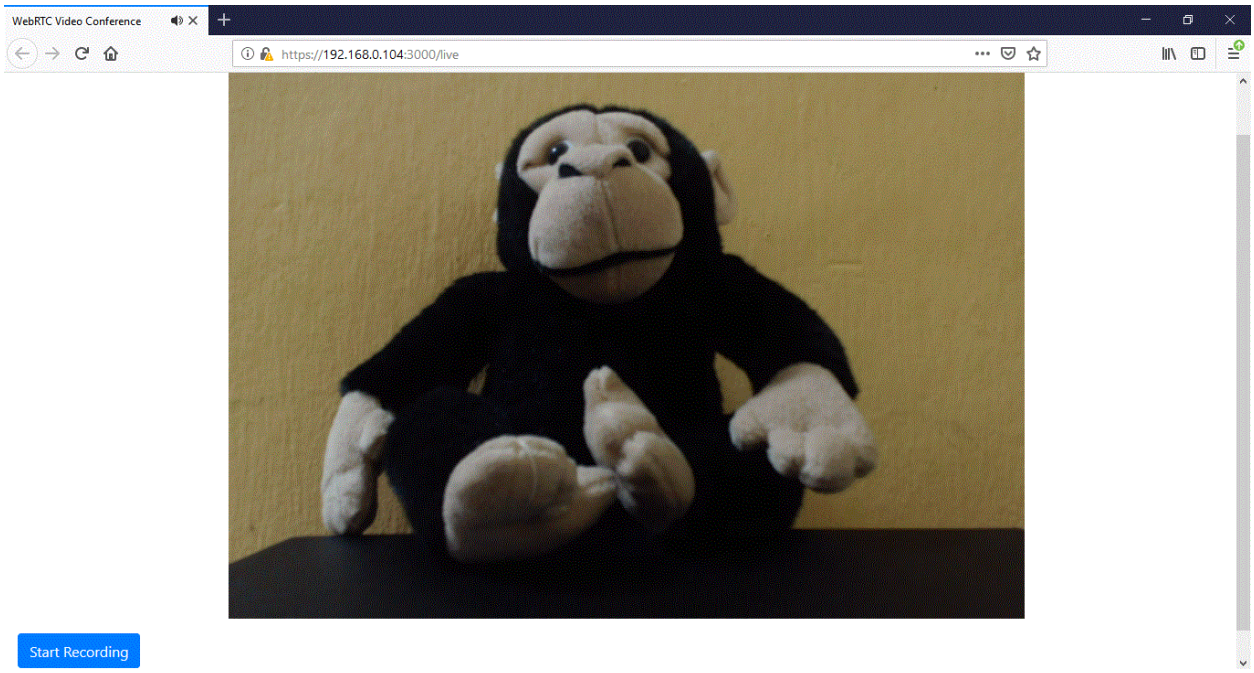

Figure 3.23: Streaming room of client (Live Stream)

3.5.6 File Streaming

File streaming includes broadcasting a video file to various client. File Streaming is almost identical to live streaming; the single difference comes from the type of video each one is broadcasting. Figure 3.22 shows a video file being uploaded, ready for broadcast.

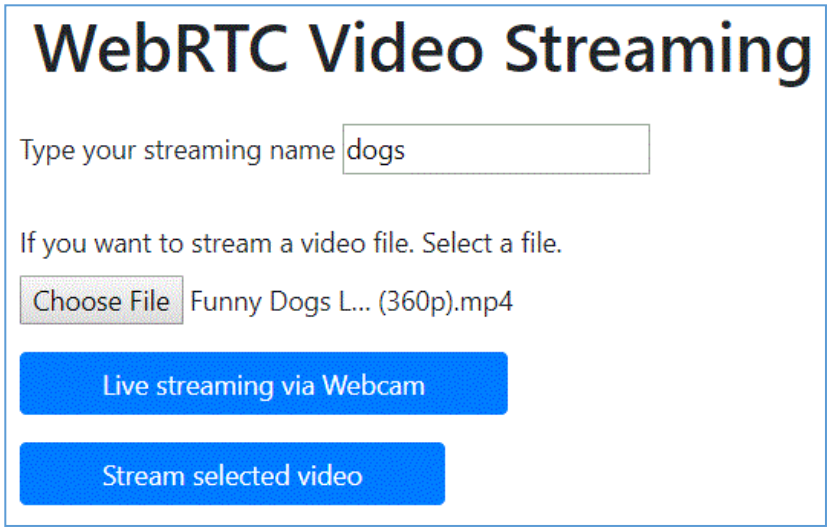

Figure 3.24: Selecting video to broadcast

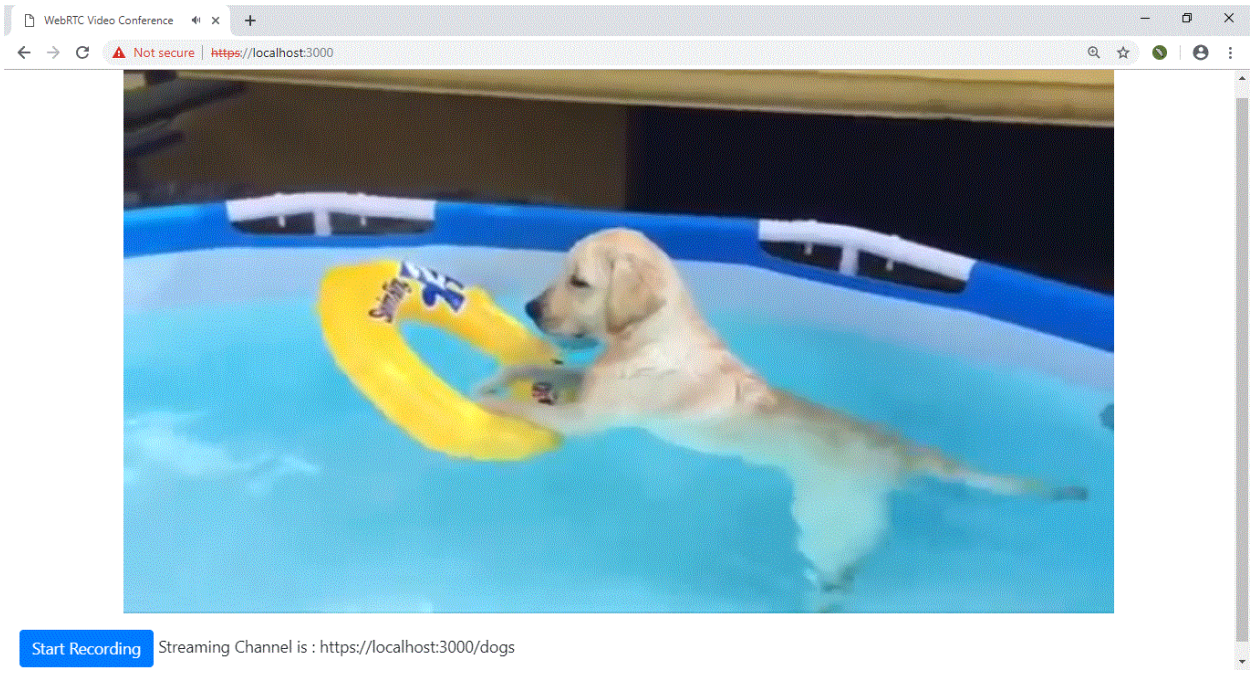

Figure 3.24: Streaming room of initiator (File Stream)

All the functions in Figure 3.20 which includes getUserMedia(), attachRecorder(), makeRemoteVideoElement(), offer() and answer() have already been defined in the video conferencing section and they do exactly the same job for video streaming as well.

Below is a complete algorithm of the video streaming application

1. User input streaming Name, streamingName
2. User uploads a video file
3. if streamingName = “”
4. alert “Please type a room number”
5. User input streamingName
6. else if file is not found
7. alert "Please select a video file"
8. User uploads a video file
9. else
10. Get streamingName

11. Get video file

12. end if

13. end if

14. Hide Homepage

15. Display Conference Room

16. Plays selected file

17. Attach recorder to local video

In order for the video of the initiator to appear on the client's page, the following processes occurred:

1. Client
    a. Creates a peer connection object,for initiator
    b. Adds onicecandidate event handler to peer connection object
    c. Sends ice candidates to initiator,
    d. Creates offer
    e. Ties local description to peer connection object
    f. Sends offer to initiator
2. Initiator
    a. Receives offer from client
    b. Creates a peer connection object for client
    c. Sets remote description of receiver to peer connection
    d. Adds onicecandidate event handler to peer connection object
    e. Sends ice candidates to initiator
    f. Adds local stream to peer connection object

g. Creates answer

h. Ties local description to peer connection object

i. Sends answer to client,

3. Client

a. Receives answer and sets remote description of initiator to peer connection object

b. Stores ice candidates in peer connection object

c. Adds video of initiator to peer connection object by triggering the onaddstream event handler

d. Displays video of initiator

e. Attaches recorder to video of initiator

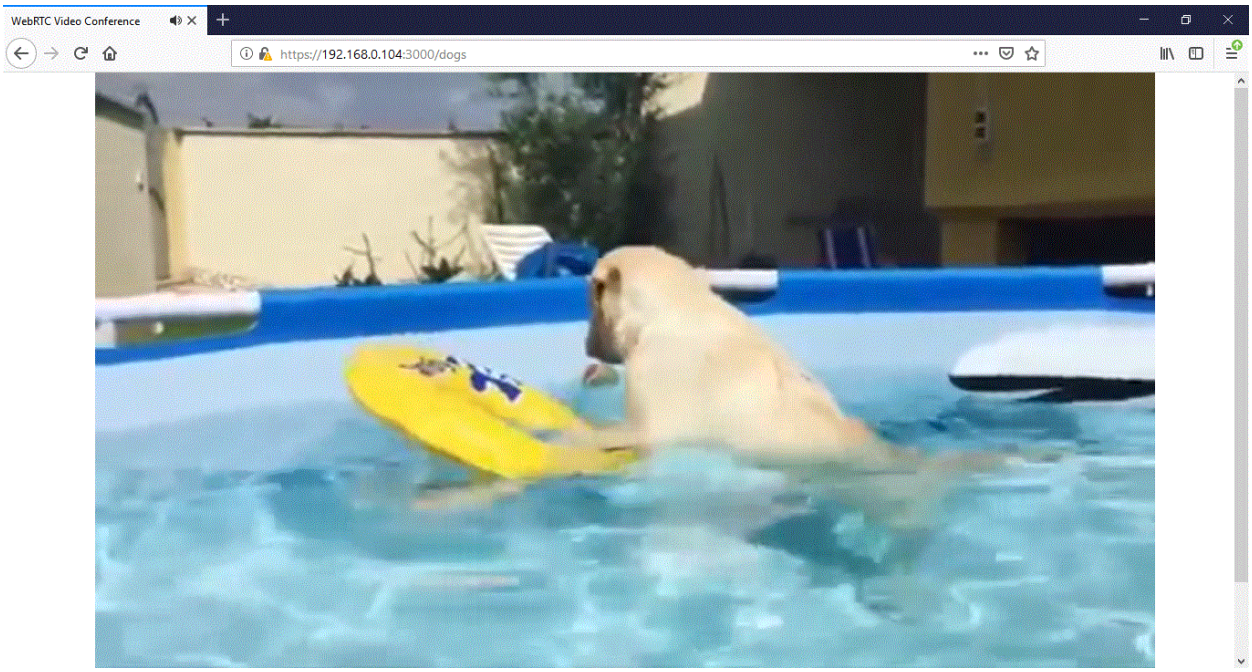

Figure 3.25: Streaming room of client (File Stream)

3.6 Port Mapping

As encountered in the methodologies above, the application requires the private IP address of the users in order to work, which means only people in a certain LAN may connect to each other. Despite the system seeming limited only to a limited area, the application's accessibility can be extended to a worldwide audience by making the necessary amendments to the codes and by configuring the port mapping [44] setting of the router.

The first step is to get every user on a certain LAN to connect to the specific computer hosting the application. This process is known as port forwarding or port mapping and its job is to redirect every user who types in the IP address and port specified by the initiator to the PC of the initiator. Figure 3.26 shows the port forwarding configurations of the initiators router.

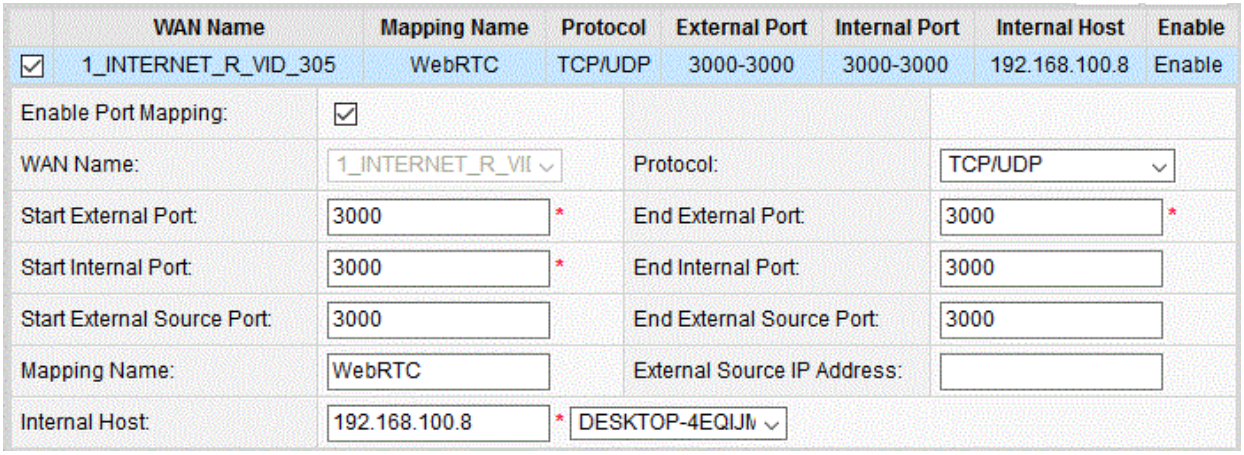

Figure 3.26: Port forwarding configurations

The second step is to configure the router setting by enabling the demilitarized zone (DMZ) [45] at the private IP address of the user as shown in Figure 3.27. The DMZ is a feature that allows only one local user to be exposed to the Internet for special purposes like Internet gaming or video conferencing and it is the key part to making the application accessible globally.

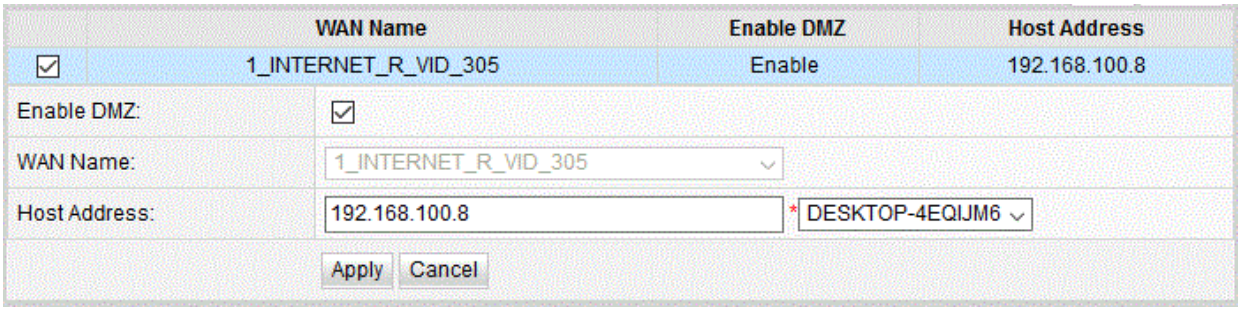

Figure 3.27: Enabling DMZ

Finally, the codes in the server needs to be adjusted by adding the public IP address as the LAN access point as shown in Figure 3.28.

```
var LANAccess = "192.168.100.8";
//listener
https.listen(3000, LANAccess);
console.log('listening on port 3000');
```

Figure 3.28: LAN access code

Now that everything is set up, the rest of the world can connect to the application by simply entering the public IP of the initiator, which the latter can obtain on the internet [46] in the place of the private IP address. Figure 3.29 shows the application being accessed using a public IP address.

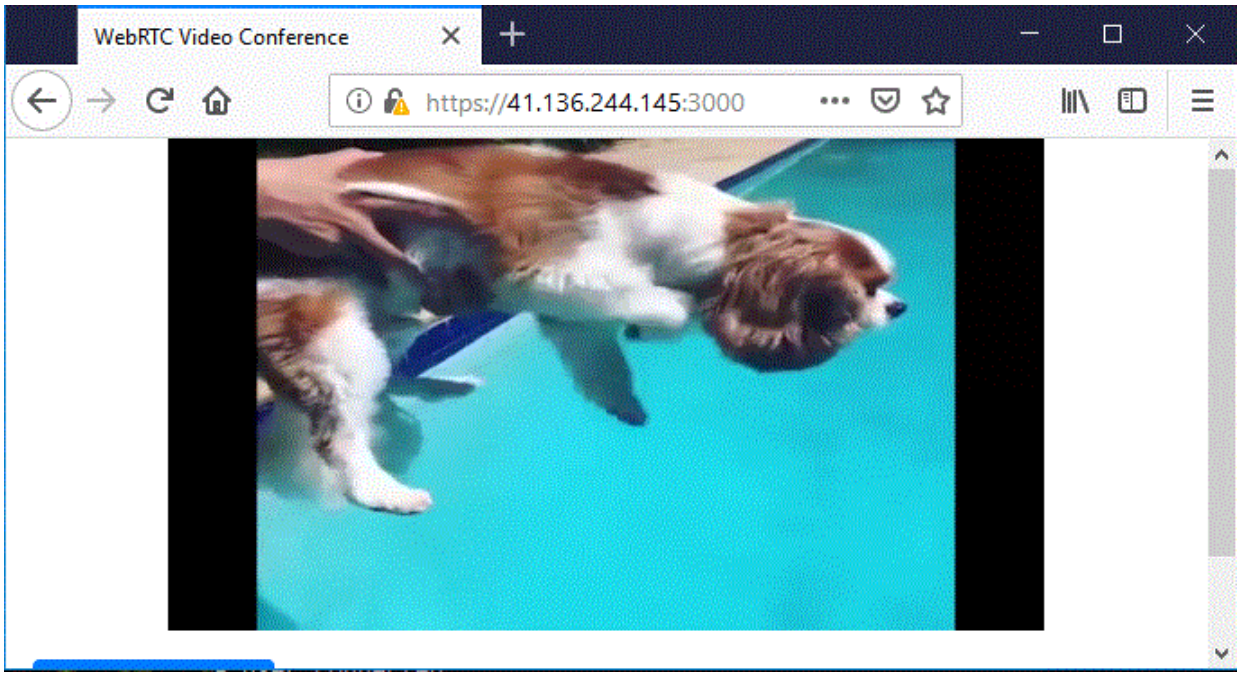

Figure 3.29: Public access to application

Chapter 4: System Testing and Results

System testing proves to be the most key part of any software developed. The aim is to analyze the variation of video quality with reference to the number of users/clients by plotting PSNR against number of users/clients. First and foremost, our video conferencing stream quality was tested with up to four clients using Wi-Fi connection first and then using Ethernet cabled connection. Secondly, the video streaming application, which includes live stream and file stream, was assessed with up to ten clients, again using Wi-Fi connection first and then using Ethernet cable connection. Finally, the variation in video quality was evaluated using two servers with five clients each compares to using only one server with ten clients. For video conferencing, four different PCs were used and for video streaming, only one PC was used. All tests were done on the same LAN and the PCs, the router as well as the internet speed, which was 10Mbps were kept constant during the experiment.

4.1 System Testing

The videos of all users were recorded for five minutes and their quality was evaluated using the PSNR metric [47]. For video conferencing, each client's local video was compared to their remote video that was obtained by other users. With video streaming, one user had set up the broadcast and the stream was compared to the other streams obtained by the other users. When there were more than one client on the conference or broadcast, the average PSNR of all the clients were considered and the same process was repeated for all clients connected to the system. Note that the downloaded videos were analyzed and compared using MATLAB.

The formula for PSNR is:

$$\begin{aligned} \text{PSNR} &= 10.\log_{10}\left(\frac{MAX^2}{MSE}\right) \\ &= 20.\log_{10}\left(\frac{MAX}{\sqrt{MSE}}\right) \\ &= 20.\log_{10}(\text{MAX}) - 10.\log_{10}(\text{MSE}) \end{aligned}$$

Where MAX is the maximum possible pixel value of the image and

MSE is the mean squared error

Note: The higher the PSNR value, the better the video quality.

4.1.1 Result of video conferencing with up to 4 clients using Wi-Fi

The test was performed with up to four users using four different PCs. The aim was to investigate how the quality of the videos altered with users joining in the conference using Wi-Fi connection. Bar charts are used which show the PSNR of videos plotted against the number of users in the conference.

4.1.1.1 Video quality of User 1

Figure 4.1 illustrates a preview of the local video of User 1, which is the leftmost picture followed by its remote videos as perceived by User 2, 3 and 4 respectively.

Figure 4.1: Snapshots of video conferencing of User 1 (Wi-Fi)

The PSNR values of the local video of User 1 in comparison to the same video obtained by the other three users were computed and tabulated. As can be seen in Table 4.1, the quality of video degrades each time a user joins in the call with PSNR being at 40.0888dB with two users all the way to 28.5186dB with four users.

Number of Users	**PSNR**			**Average PSNR**
	User 2	**User 3**	**User 4**	
2	40.0888	-	-	40.0888
3	34.9112	34.9487	-	34.9300
4	28.5289	28.4478	28.5792	28.5186

Table 4.1: Result of Video conferencing for User 1 (Wi-Fi)

The trend of video quality with an increase in number of users is illustrated in Figure 4.2

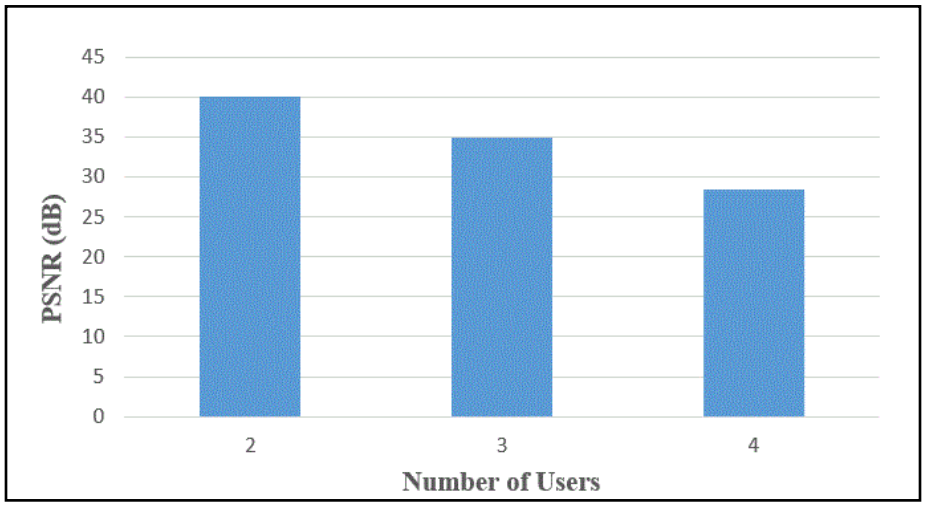

Figure 4.2: Video conferencing quality for User 1 (Wi-Fi)

4.1.1.2 Video quality of User 2

Figure 4.3 illustrates a preview of the local video of User 2, which is the leftmost picture followed by its remote videos as perceived by User 1, 3 and 4 respectively.

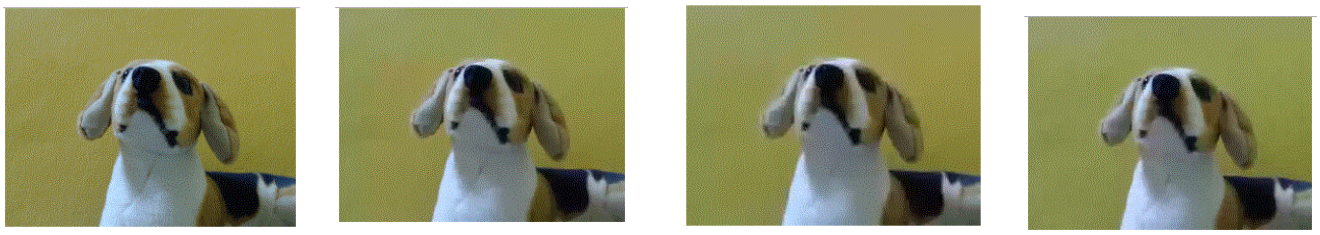

Figure 4.3: Snapshots of video conferencing of User 2 (Wi-Fi)

The PSNR values of the local video of User 2 in comparison to the same video obtained by the other three users were computed and tabulated. As can be seen in Table 4.2, the quality of the video degrades each time a user joins in the call with PSNR being at 37.5212dB with two users all the way to 24.7651dB with four users

Number of Users	PSNR			Average PSNR
	User 2	User 3	User 4	
2	37.5212	-	-	37.5212
3	28.4186	30.8206	-	29.6196
4	25.0635	26.2157	23.0161	24.7651

Table 4.2: Result of Video conferencing for User 2 (Wi-Fi)

The trend of video quality with an increase in number of users is illustrated in Figure 4.4

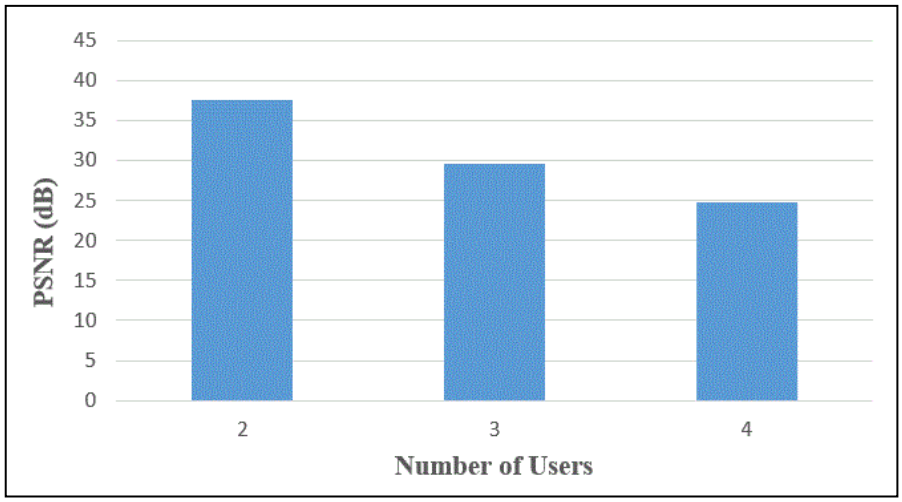

Figure 4.4: Video conferencing quality for User 2 (Wi-Fi)

4.1.1.3 Video quality of User 3

Figure 4.5 illustrates a preview of the local video of User 3, which is the leftmost picture followed by its remote videos as perceived by User 1, 2 and 4 respectively.

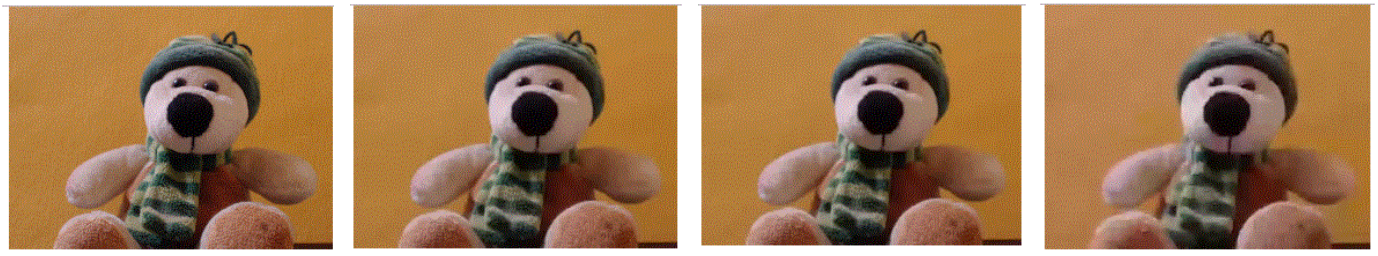

Figure 4.5: Snapshots of video conferencing of User 3 (Wi-Fi)

The PSNR values of the local video of User 3 in comparison to the same video obtained by the other three users were computed and tabulated. As can be seen in Figure 4.3, the quality of the video degrades each time a user joins in the call with PSNR being at 39.5250dB with two users to 33.4041dB with four users. Note that for User 3, the PSNR hasn't gone down by much compared to the other users which states that quality of video hasn't gone down by much.

Number of Users	PSNR			Average PSNR
	User 2	User 3	User 4	
2	39.5250	-	-	39.5250
3	38.8228	38.8250	-	38.6739
4	33.2113	34.1813	34.8196	33.4041

Table 4.3: Result of Video conferencing for User 3 (Wi-Fi)

The trend of video quality with an increase in number of users is illustrated in Figure 4.6

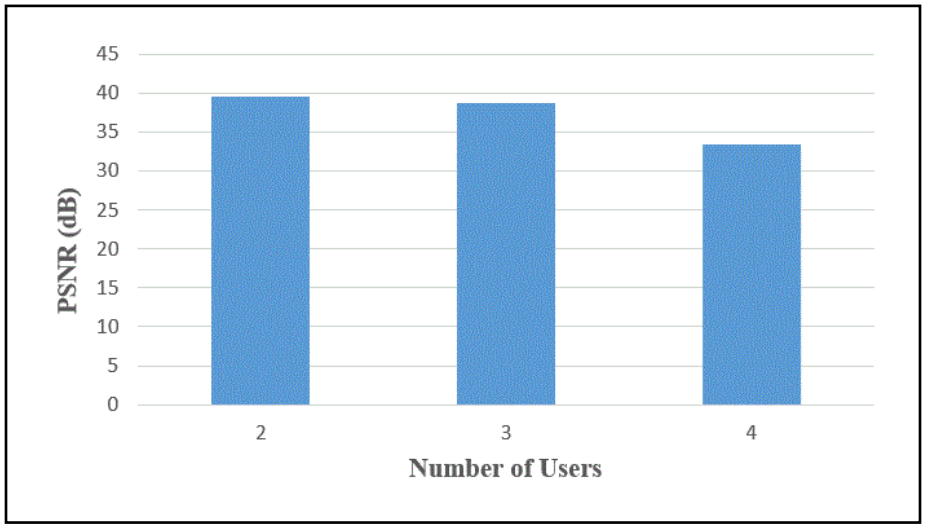

Figure 4.6: Video Conferencing quality for User 3 (Wi-Fi)

4.1.1.4 Video quality of User 4

Figure 4.7 illustrates a preview of the local video of User 4, which is the leftmost picture followed by its remote videos as perceived by User 1, 2 and 3 respectively.

Figure 4.7: Snapshots of video conferencing of User 4 (Wi-Fi)

The PSNR values of the local video of User 4 in comparison to the same video obtained by the other three users were computed and tabulated. As can be seen in Table 4.4, the quality of the video degrades each time a user joins in the call with PSNR being at 34.2463dB with two users all the way to 20.4846dB with four users. Note that for User 4, the PSNR values are quite low compared to the other users which states that quality of video has been very poor.

Number of Users	**PSNR**			**Average PSNR**
	User 2	**User 3**	**User 4**	
2	34.2463	-	-	34.2463
3	29.1797	21.3427	-	25.2612
4	20.4896	20.4823	20.4819	20.4846

Table 4.4: Result of Video conferencing for User 4 (Wi-Fi)

The trend of video quality with an increase in number of users is illustrated in Figure 4.8

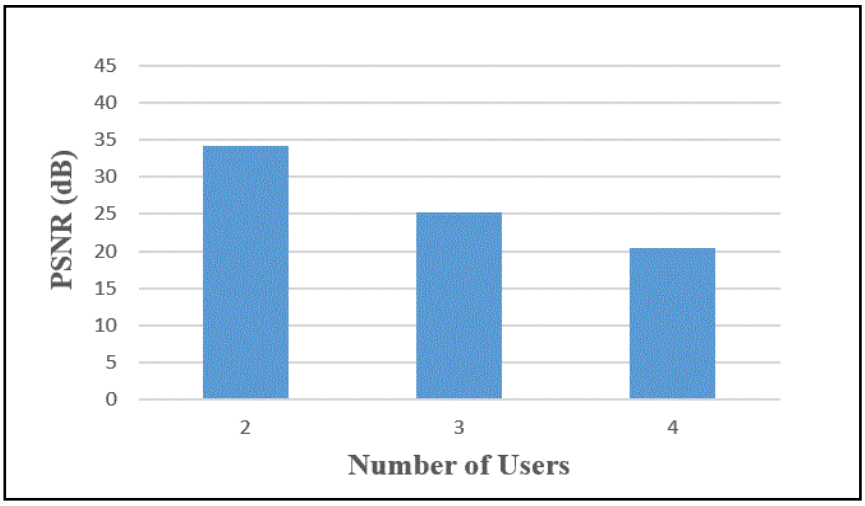

Figure 4.8: Video Conferencing quality for User 4 (Wi-Fi)

4.1.2 Result of video conferencing with up to 4 clients using Ethernet Cable

The test was performed with up to four users using four different PCs. The aim is to use Ethernet Cable connection to investigate how the quality of the videos alters with users joining in the conference. Bar charts are used which show the PSNR of videos plotted against the number of users in the conference. In the results below, we may contemplate that Ethernet Cabled connection provides a better and more stabled quality of video.

4.1.2.1 Video quality of User 1

Figure 4.9 illustrates a preview of the local video of User 1, which is the leftmost picture followed by its remote videos as perceived by User 2, 3 and 4 respectively.

Figure 4.9: Snapshots of video conferencing of User 1 (Ethernet Cable)

The trend of video quality with an increase in number of users is illustrated in Figure 4.10

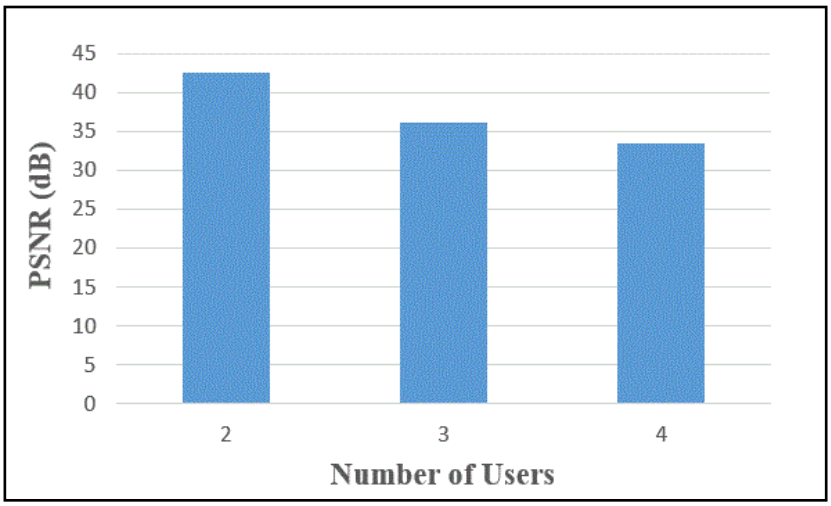

Figure 4.10: Video conferencing quality for User 1 (Ethernet Cable)

4.1.2.2 Video quality of User 2

Figure 4.11 illustrates a preview of the local video of User 2, which is the leftmost picture followed by its remote videos as perceived by User 1, 3 and 4 respectively.

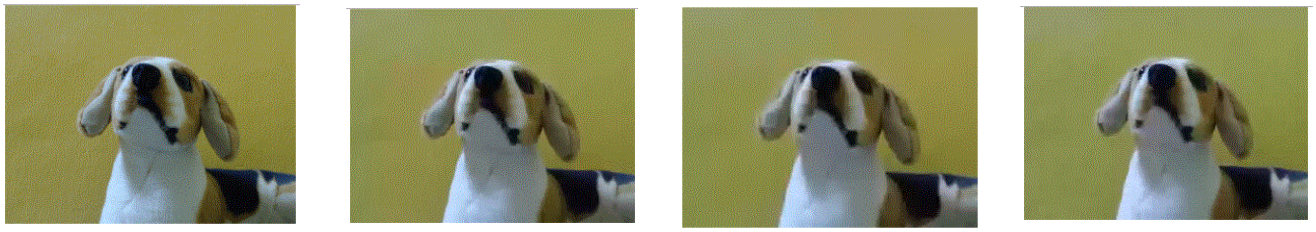

Figure 4.11: Snapshots of video conferencing of User 2 (Ethernet Cable)

The PSNR values of the local video of User 2 in comparison to the same video obtained by the other three users were computed and tabulated. As can be seen in Table 4.6, the quality of the video degrades each time a user joins in the call with PSNR being at 35.9142dB with two users all the way to 26.6463dB with four users.

Number of Users	PSNR			Average PSNR
	User 2	User 3	User 4	
2	35.9142	-	-	35.9142
3	30.0288	30.5441	-	30.2865
4	26.6178	26.6165	27.1047	26.6463

Table 4.6: Result of Video conferencing for User 2 (Ethernet Cable)

The trend of video quality with an increase in number of users is illustrated in Figure 4.12

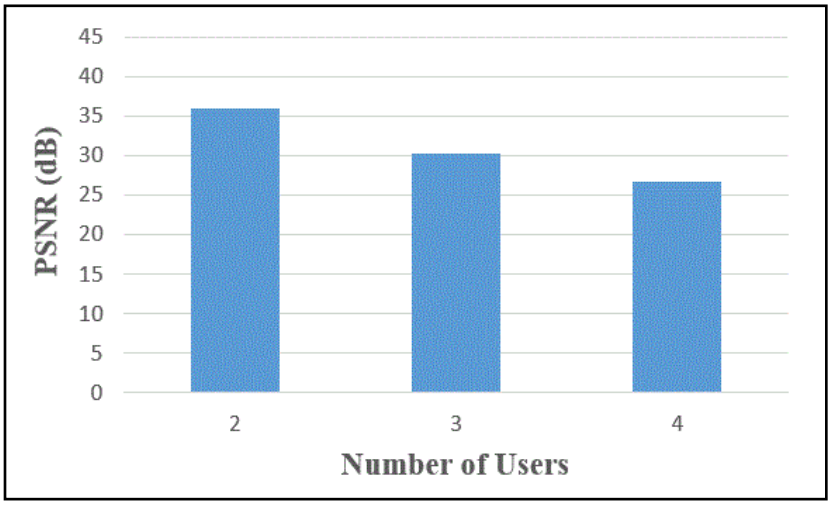

Figure 4.12: Video conferencing quality for User 2 (Ethernet Cable)

4.1.2.3 Video quality of User 3

Figure 4.13 illustrates a preview of the local video of User 3, which is the leftmost picture followed by its remote videos as perceived by User 1, 2 and 4 respectively.

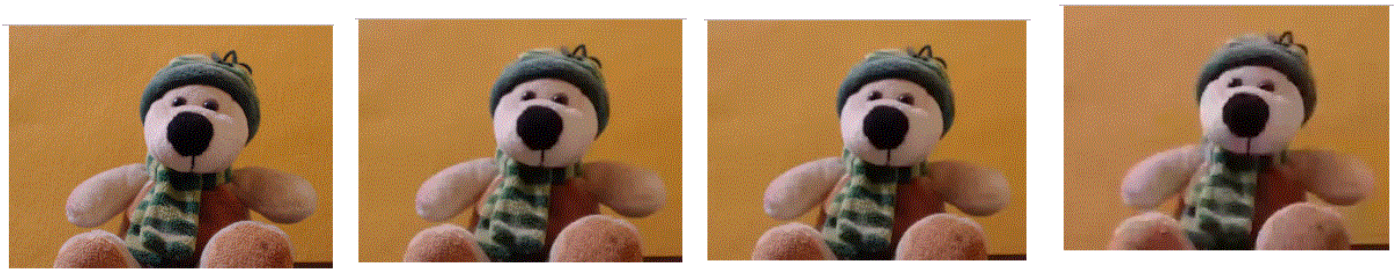

Figure 4.13: Snapshots of video conferencing of User 3 (Ethernet Cable)

The PSNR values of the local video of User 3 in comparison to the same video obtained by the other three users were computed and tabulated. As can be seen in Table 4.7, the quality of the video degrades each time a user joins in the call with PSNR being at 40.1408dB with two users to 36.0473dB with four users.

Number of Users	PSNR			Average PSNR
	User 2	User 3	User 4	
2	40.1408	-	-	40.1408
3	38.6202	38.8621	-	38.7412
4	35.8224	36.4180	35.9014	36.0473

Table 4.7: Result of Video conferencing for User 3 (Ethernet Cable)

The trend of video quality with an increase in number of users is illustrated in Figure 4.14

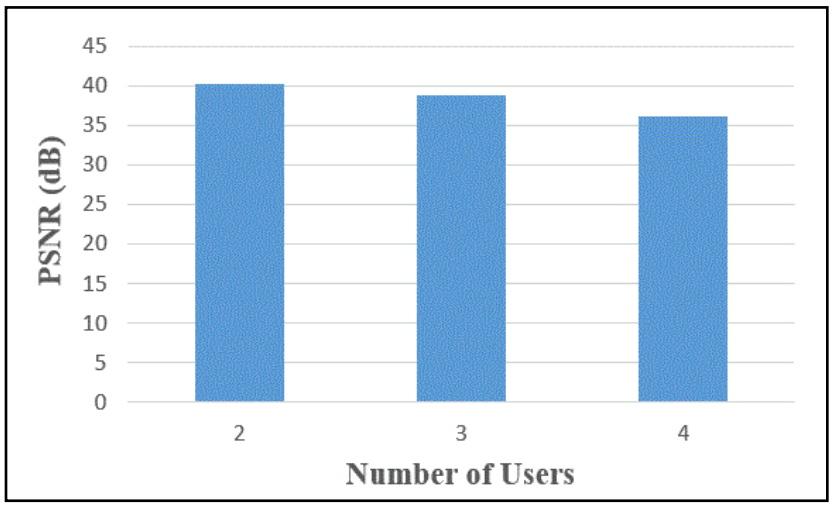

Figure 4.14: Video conferencing quality for User 3 (Ethernet Cable)

4.1.2.4 Video quality of User 4

Figure 4.15 illustrates a preview of the local video of User 4, which is the leftmost picture followed by its remote videos as perceived by User 1, 2 and 3 respectively.

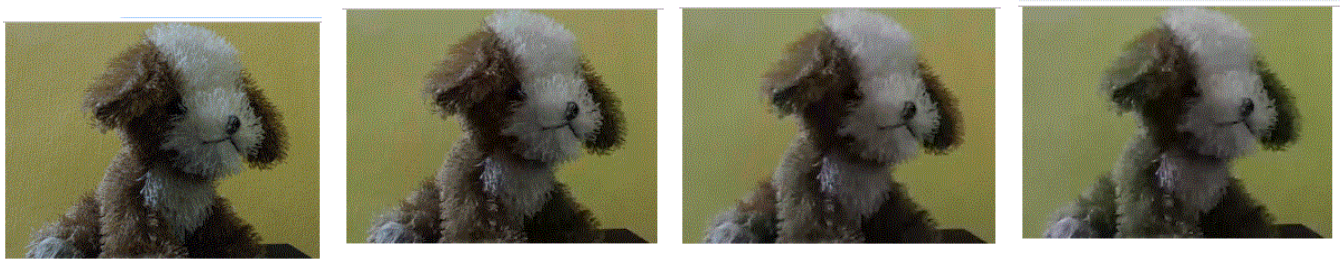

Figure 4.15: Snapshots of video conferencing of User 4 (Ethernet Cable)

The PSNR values of the local video of User 3 in comparison to the same video obtained by the other three users were computed and tabulated. As can be seen in Table 4.8, the quality of the video degrades each time a user joins in the call with PSNR being at 45.0351dB with two users all the way to 27.0375dB with four users

Number of Users	**PSNR**			**Average PSNR**
	User 2	**User 3**	**User 4**	
2	45.0351	-	-	45.0351
3	30.4806	29.8248	-	30.1527
4	27.4162	26.5416	27.1548	27.0375

Table 4.8: Result of Video conferencing for User 4 (Ethernet Cable)

The trend of video quality with an increase in number of users is illustrated in Figure 4.16

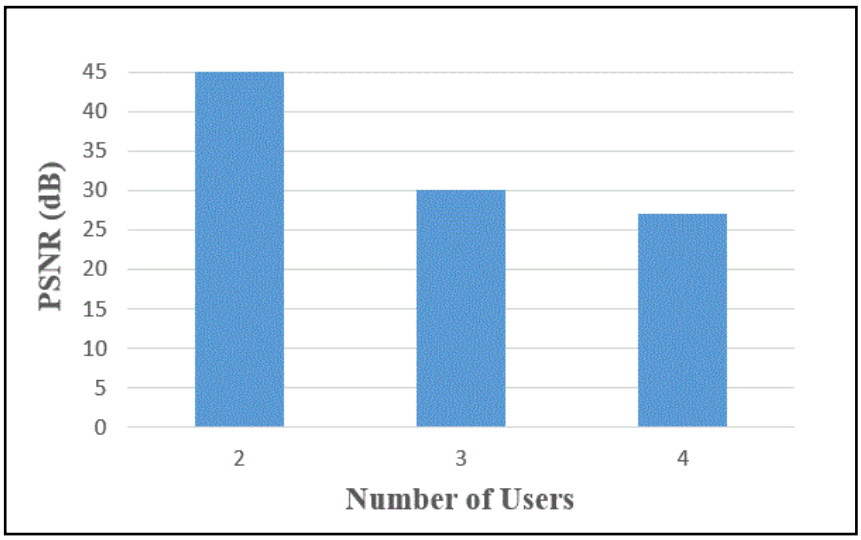

Figure 4.16: Video conferencing quality for User 4 (Ethernet Cable)

4.1.3 Result of video streaming with up to 10 clients using Wi-Fi

The test was performed with up to ten clients, all using one PC. The aim is to investigate how the quality of the videos varies with clients joining in the broadcast using Wi-Fi connection. Bar charts are used which show the PSNR of videos plotted against the number of clients in the broadcast.

4.1.3.1 Live Stream quality of 10 clients

Figure 4.17 shows the local video of the initiator and the video received by the 10th client in the broadcast.

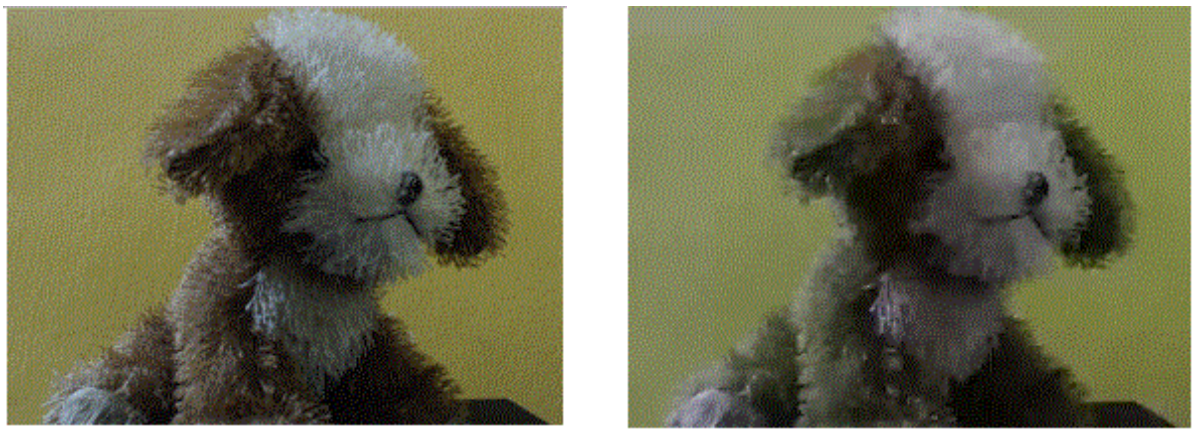

Figure 4.17: Snapshots of Live streaming (Wi-Fi)

The PSNR values of the video being broadcasted by the initiator in comparison with the ones received by all ten connected clients were computed and tabulated. As can be seen in Table 4.9, the quality of the streams decreases each time a client joins in the broadcast with PSNR being at 42.9247 dB with one client all the way to 36.2482 dB with ten clients.

Number of Clients	Average PSNR
1	42.9247
2	42.6618
3	42.2711
4	41.8404
5	41.5324
6	40.5101
7	39.2333
8	38.0132
9	37.1572
10	36.2482

Table 4.9: Result of Live Stream (Wi-Fi)

The trend of video quality with an elevated number of clients is illustrated in Figure 4.18.

Figure 4.18: Live Stream quality (Wi-Fi)

4.1.3.2 File Stream quality of 10 clients (Regular Video)

Figure 4.19 shows the local video of the initiator and the video received by the 10th client in the broadcast.

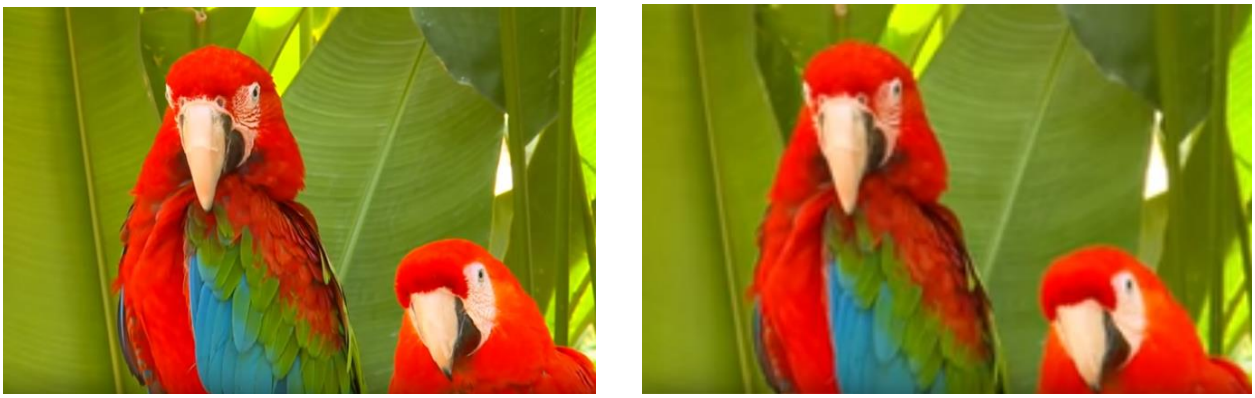

Figure 4.19: Snapshots of File streaming with Regular Video (Wi-Fi)

The PSNR values of the video being broadcasted by the initiator in comparison with the ones received by all ten connected clients were computed and tabulated. As can be seen in Table 4.10, the quality of the streams decreases each time a client joins in the broadcast with PSNR being at 14.1811 dB with one client to 13.2406 dB with 10 clients.

Note: The low PSNR in a regular video is a consequence of frequent change in frames.

Number of Clients	Average PSNR
1	14.1811
2	14.0224
3	13.8262
4	13.5994
5	13.5121
6	13.4809
7	13.4334
8	13.3821
9	13.2902
10	13.2406

Table 4.10: Result of File Stream of Regular Video (Wi-Fi)

The trend of video quality with an elevated number of clients is illustrated in Figure 4.20.

Figure 4.20: File Stream Quality of Regular Video (Wi-Fi)

4.1.3.3 File Stream quality of 10 clients (Still Video)

Figure 4.21 shows the local video of the initiator and the video received by the 10th client in the broadcast.

Figure 4.21: Snapshots of File streaming with Still Video (Wi-Fi)

The PSNR values of the video being broadcasted by the initiator in comparison with the ones received by all ten connected clients were computed and tabulated. As can be seen in Table 4.11,

the quality of the streams decreases each time a client joins in the broadcast with PSNR being at 41.6460dB with one client all the way to 37.1471dB with ten clients.

Number of Clients	Average PSNR
1	41.6460
2	41.6358
3	41.5253
4	41.5014
5	41.4848
6	40.8614
7	39.1106
8	38.5153
9	37.8654
10	37.1471

Table 4.11: Result of File Stream of Still Video (Wi-Fi)

The trend of video quality with an elevated number of clients is illustrated in Figure 4.22.

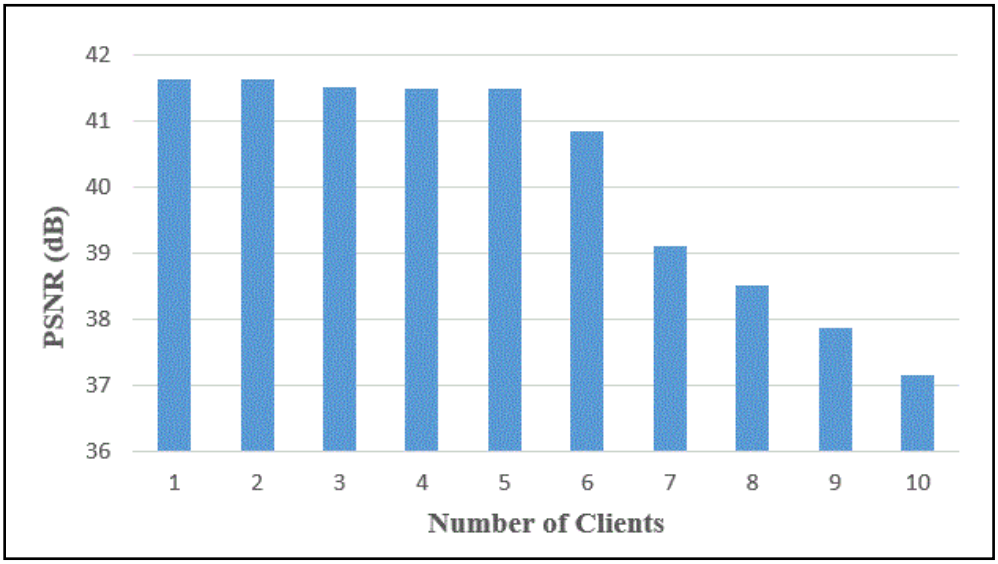

Figure 4.22: File Stream Quality of Still Video (Wi-Fi)

4.1.4 Result of video streaming with up to 10 clients using Ethernet Cable

The test was performed with up to ten clients. The aim is to use Ethernet Cable connection to investigate how the quality of the videos varies with clients joining in the broadcast. Bar charts are used which show the PSNR of videos plotted against the number of clients in the broadcast.

4.1.4.1 Live Stream quality of 10 clients

Figure 4.23 shows the local video of the initiator and the video received by the 10th client in the broadcast.

Figure 4.23: Snapshots of Live streaming (Ethernet Cable)

The PSNR values of the video being broadcasted by the initiator in comparison with the ones received by all ten connected clients were computed and tabulated. As can be seen in Table 4.12, the quality of the streams decreases each time a client joins in the broadcast with PSNR being at 42.3148 dB with one client all the way to 38.9150 dB with ten clients.

Number of Clients	Average PSNR
1	42.3148
2	42.0257
3	41.2705
4	40.3238
5	40.0815
6	39.9214
7	39.6402
8	39.6402
9	39.0539
10	38.9105

Table 4.12: Result of Live Stream (Ethernet Cable)

The trend of video quality with an elevated number of clients is illustrated in Figure 4.24.

Figure 4.24: Live Stream quality (Ethernet Cable)

4.1.4.2 File Stream quality of 10 clients (Regular Video)

Figure 4.25 shows the local video of the initiator and the video received by the 10th client in the broadcast.

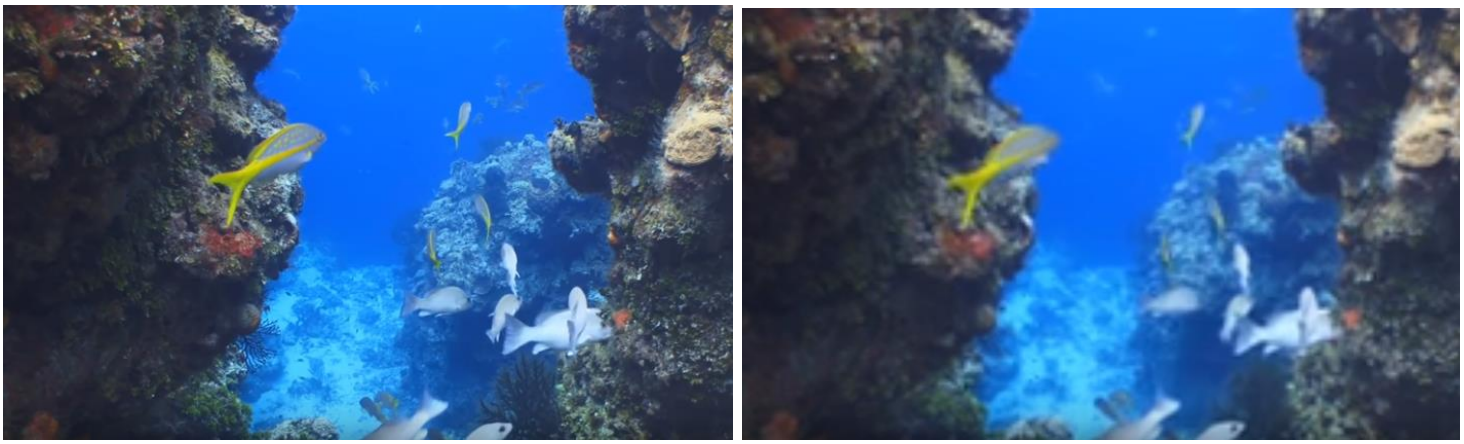

Figure 4.25: Snapshots of File streaming with Regular Video (Ethernet Cable)

The PSNR values of the video being broadcasted by the initiator in comparison with the ones received by all ten connected clients were computed and tabulated. As can be seen in Table 4.13, the quality of the streams decreases each time a client joins in the broadcast with PSNR being at 14.0841 dB with one client to 12.9112 dB with ten clients.

Number of Clients	Average PSNR
1	14.0841
2	13.9140
3	13.7010
4	13.6604
5	13.4205
6	13.2851
7	13.1765
8	13.1294
9	13.0582
10	12.9112

Table 4.13: Result of File Stream of Regular Video (Ethernet Cable)

The trend of video quality with an elevated number of clients is illustrated in Figure 4.26.

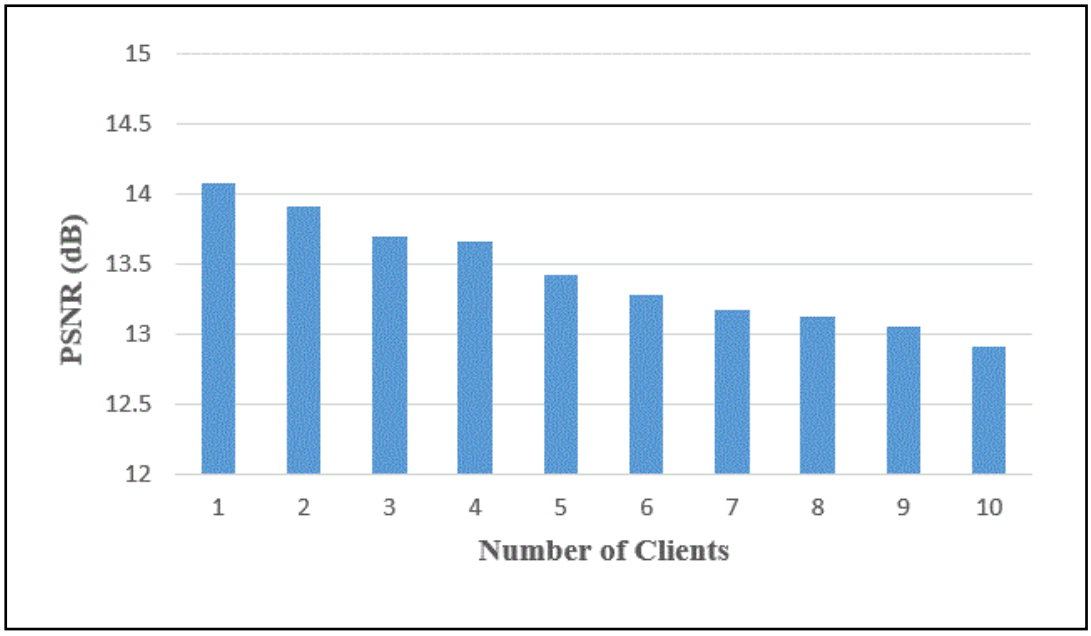

Figure 4.26: File Stream Quality of Regular Video (Ethernet Cable)

4.1.4.3 File Stream quality of 10 clients (Still Video)

Figure 4.27 shows the local video of the initiator and the video received by the 10th client in the broadcast.

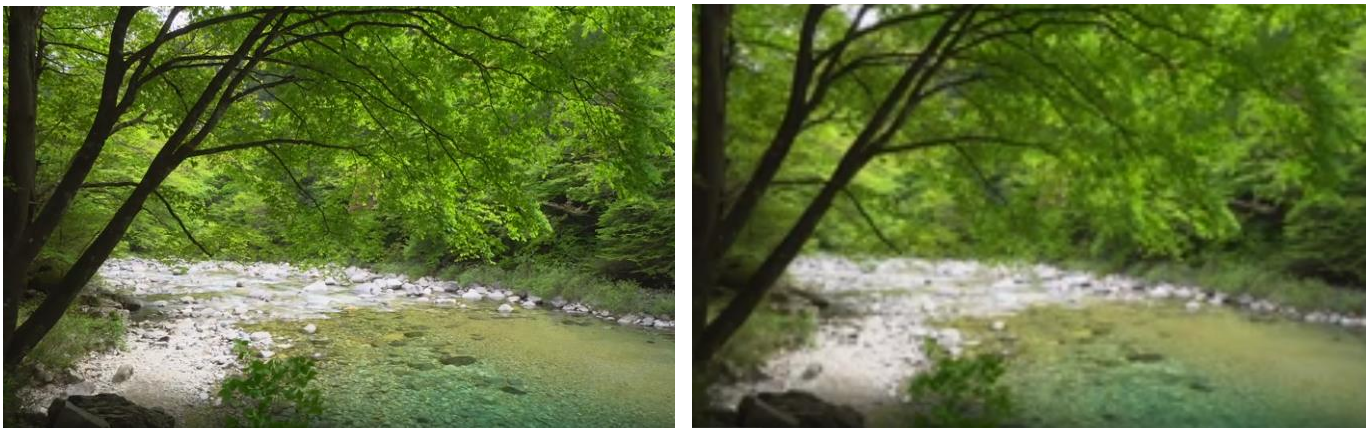

Figure 4.27: Snapshots of File streaming with Still Video (Ethernet Cable)

The PSNR values of the video being broadcasted by the initiator in comparison with the ones received by all ten connected clients were computed and tabulated. As can be seen in Table 4.14, the quality of the streams decreases each time a client joins in the broadcast with PSNR being at 41.396 dB with one client all the way to 37.8952 dB with ten clients.

Number of Clients	Average PSNR
1	41.6396
2	41.6103
3	41.5952
4	41.5685
5	41.4904
6	41.2468
7	40.5827
8	39.2588
9	37.9089
10	37.8952

Table 4.14: File Stream Quality of Still Video (Ethernet Cable)

The trend of video quality with an elevated number of clients is illustrated in Figure 4.28.

Figure 4.28: File Stream Quality of Still Video (Ethernet Cable)

4.1.5 Result of video streaming with 2 servers using Wi-Fi

Contrary to the other testing, instead of assigning all ten clients with one server, two servers were used simultaneously on two different PCs assigned with five clients each. The aim is to see whether increasing the number of servers with the same amount of clients enhances the QoS or not. The blue graph represents the use of two servers: Server 1, which is shown as the first five graphs and Server 2, which is shown as last five graphs. The orange graph represents the use of only one server just as in section 4.1.3.

4.1.5.1 Live Stream quality with two servers

Figure 4.29 shows the comparison between the first and the tenth client for when using two servers with five clients each.

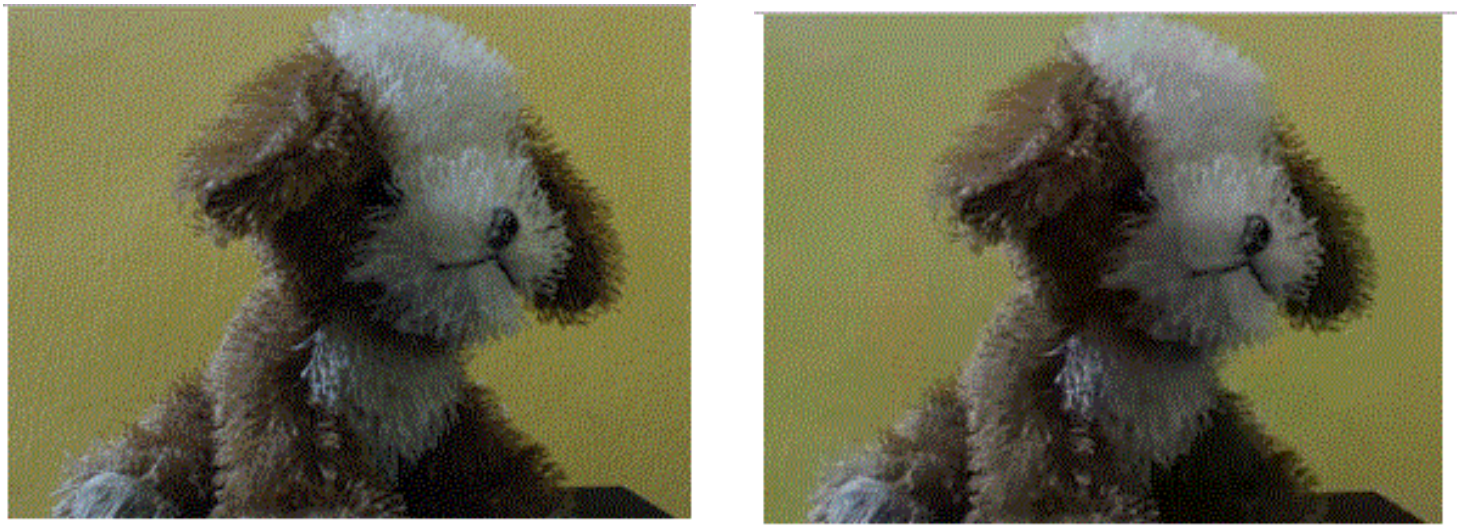

Figure 4.29: Snapshot of Live Stream using two servers (Wi-Fi)

For each of the two servers used, the average PSNR values of videos being broadcasted and videos received were computed and tabulated in Table 4.15 and Table 4.16.

Number of Clients	Average PSNR
1	42.9354
2	42.7321
3	42.5236
4	42.1569
5	41.8365

Table 4.15: Live Stream Quality with Server 1 (Wi-Fi)

Number of Clients	Average PSNR
1	42.8632
2	42.7642
3	42.5369
4	41.9658
5	41.6325

Table 4.16: Live Stream Quality with Server 2 (Wi-Fi)

Figure 4.30 shows the comparison between using one server and two servers for a total of ten clients. For the first five clients, the PSNR of both servers are almost equal but as from the sixth client, the difference seems to be quite significant. This is due to the addition of Server 2, which treats the sixth client as the first client in the broadcast. The least value of PSNR with one server was 36.2482dB compared to 41.6325dB with two servers.

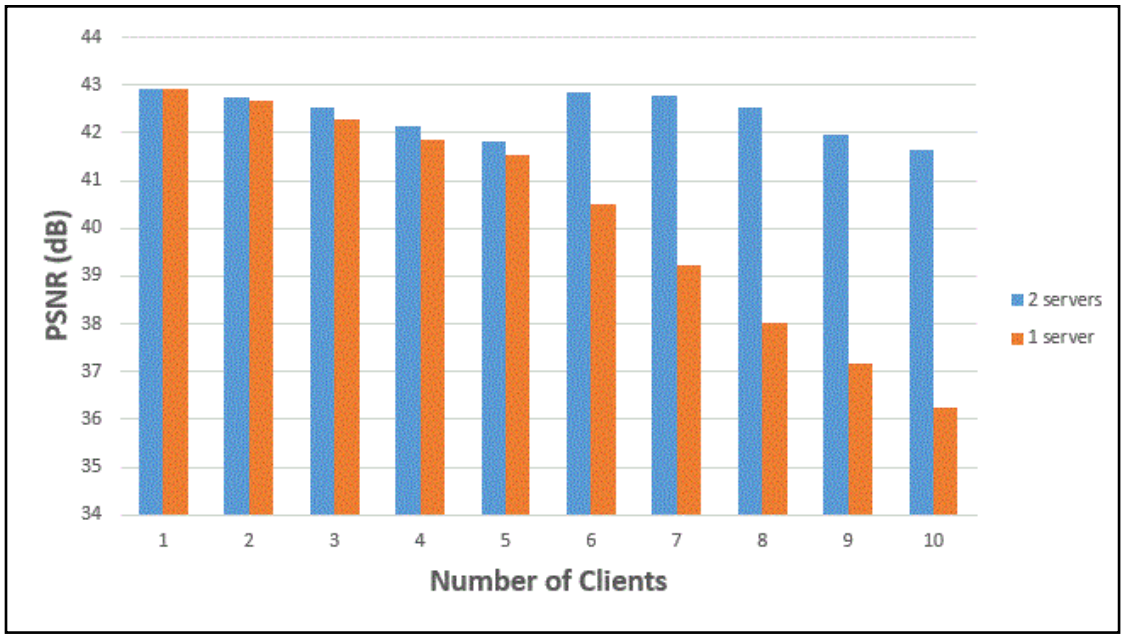

Figure 4.30: Live Stream Quality with two servers (Wi-Fi)

4.1.5.3 File Stream quality with two servers (Regular Video)

Figure 4.31 shows the comparison between the first and the tenth client for when using two servers with five clients each.

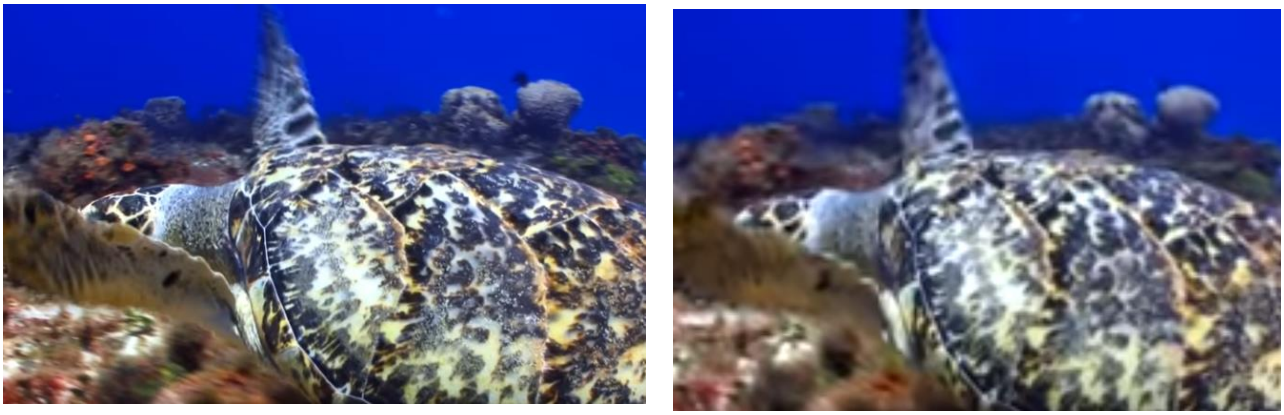

Figure 4.31: Snapshots of File streaming with Regular Video (Wi-Fi)

For each of the two servers used, the average PSNR values of videos being broadcasted and videos received were computed and tabulated in Table 4.17 and Table 4.18.

Number of Clients	Average PSNR
1	14.1684
2	14.0325
3	13.9625
4	13.6125
5	13.5421

Table 4.17: File Stream Quality of Regular video with Server 1 (Wi-Fi)

Number of Clients	Average PSNR
1	14.1754
2	14.1038
3	14.0694
4	13.6954
5	13.5823

Table 4.18: File Stream Quality of Regular video with Server 2 (Wi-Fi)

Figure 4.32 shows the comparison between using one server and two servers for a total of ten clients. For the first five clients, the PSNR of both servers are almost equal but as from the sixth client, the difference seems to be quite significant. This is due to the addition of Server 2 which treats the sixth client as the first client in the broadcast. The least value of PSNR with one server was 13.2406dB compared to 13.5421dBwith two servers.

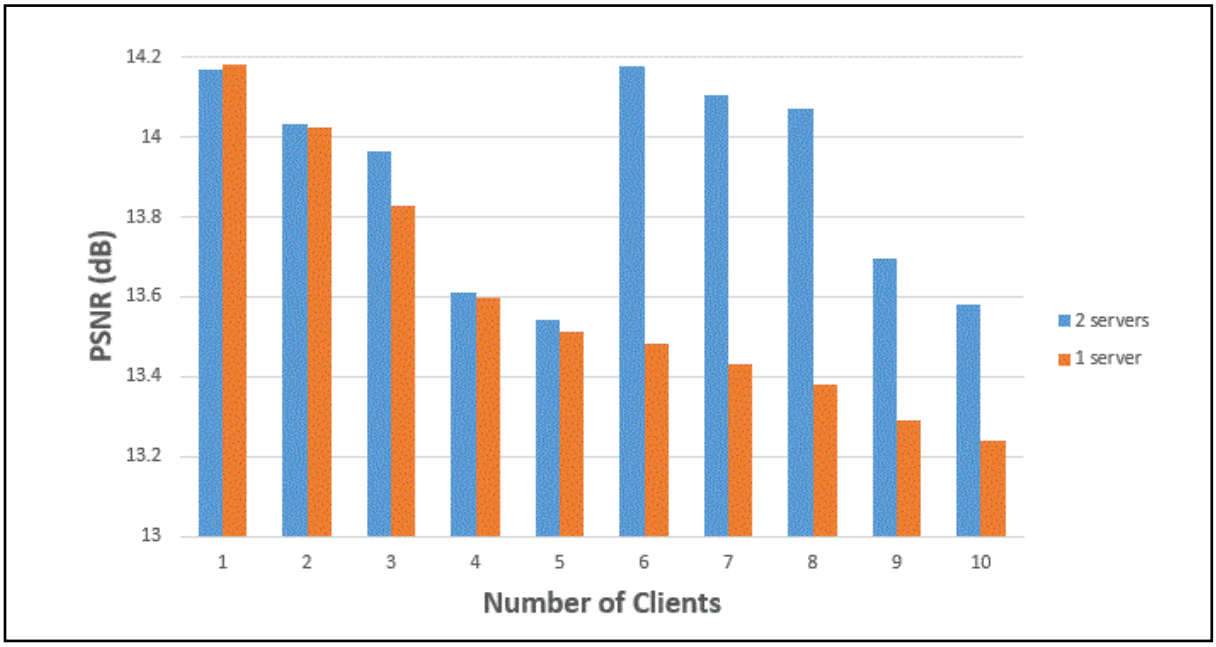

Figure 4.32: File Stream Quality of Regular video with two servers (Wi-Fi)

4.1.5.5 File Stream quality with two servers (Still Video)

Figure 4.33 shows the comparison between the first and the tenth client for when using two servers with five clients each.

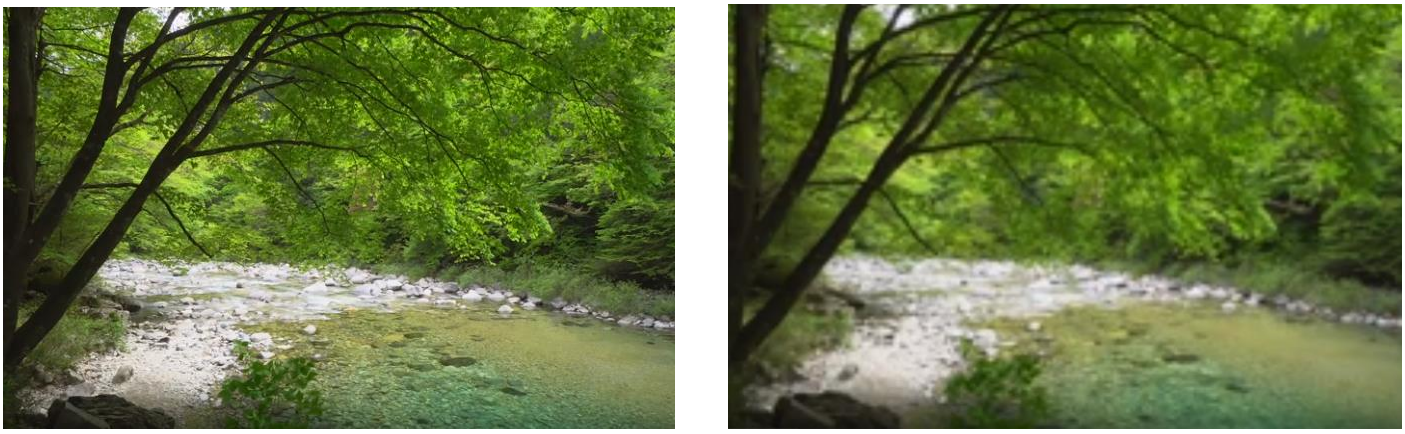

Figure 4.33: Snapshots of File streaming with Still Video (Wi-Fi)

For each of the two servers used, the average PSNR values of videos being broadcasted and videos received were computed and tabulated in Table 4.19 and Table 4.20.

Number of Clients	Average PSNR
1	41.6801
2	41.5431
3	41.4826
4	41.3910
5	41.3556

Table 4.19: File Stream Quality of Still video with Server 1 (Wi-Fi)

Number of Clients	Average PSNR
1	41.3813
2	41.3495
3	41.2408
4	41.1855
5	41.1217

Table 4.20: File Stream Quality of Still video with Server 2 (Wi-Fi)

Figure 4.34 shows the comparison between using one server and two servers for a total of ten clients. For the first five clients, the PSNR of both servers are almost equal but as from the sixth client, the difference seems to be quite significant. This is due to the addition of Server 2 which treats the sixth client as the first client in the broadcast. The least value of PSNR with one server was 41.1217dB compared to 37.1471dB with two servers.

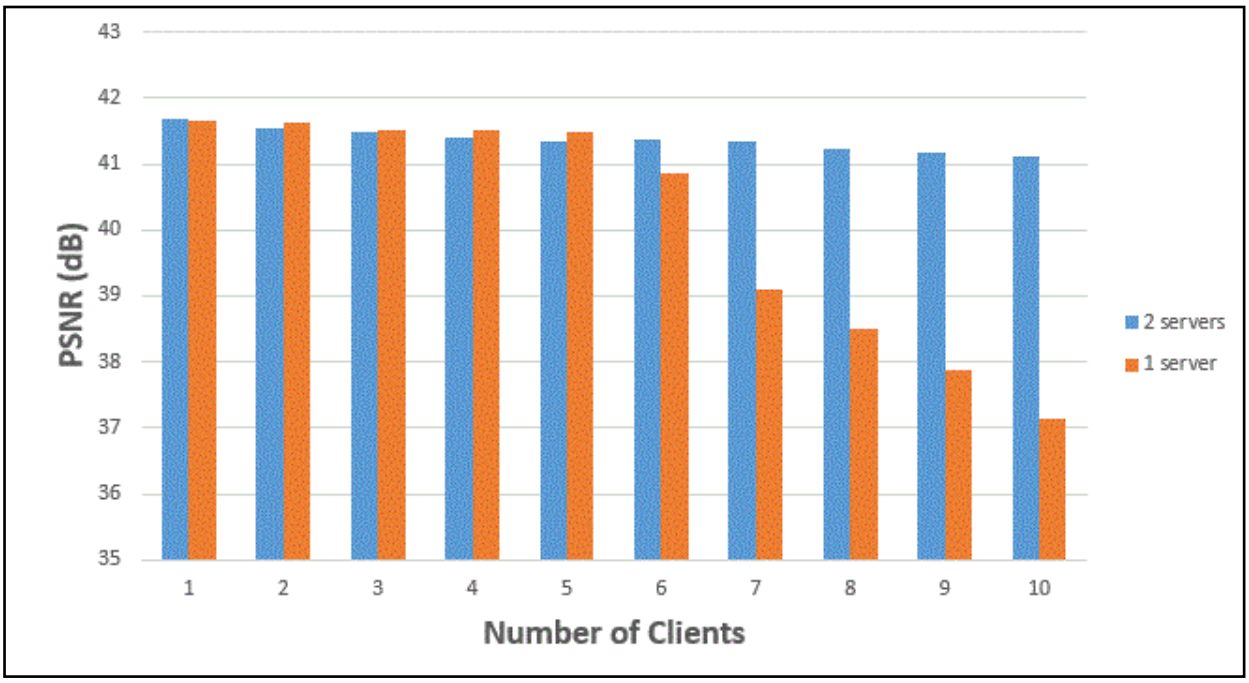

Figure 4.34: File Stream Quality of Still video with two servers (Wi-Fi)

4.1.6 Result of video streaming using 2 servers using Ethernet Cable

Contrary to the other testing, instead of assigning all ten clients with one server, two servers were used simultaneously on two different PCs with five clients each using Ethernet cabled connection. The aim is to see whether increasing the number of servers with the same amount of clients enhances the QoS or not. The blue graph represents using two servers: Server 1, which is shown as the first five graphs and Server 2, which is shown as last five graphs. The orange graph represents only one server just as in section 4.1.4.

4.1.6.1 Live Stream quality with two servers

Figure 4.35 shows the comparison between the first and the tenth client for when using two servers with five clients each.

Figure 4.35: Snapshot of Live Stream using two servers (Ethernet Cable)

For each of the two servers used, the average PSNR values of videos being broadcasted and videos received were computed and tabulated in Table 4.21 and Table 4.22.

Number of Clients	Average PSNR
1	42.1409
2	41.9630
3	41.5441
4	40.8549
5	39.9781

Table 4.21: Live Stream Quality with Server 1 (Ethernet Cable)

Number of Clients	Average PSNR
1	42.8201
2	42.4355
3	41.8040
4	41.2502
5	40.5422

Table 4.22: Live Stream Quality with Server 2 (Ethernet Cable)

Figure 4.36 shows the comparison between using one server and two servers for a total of ten clients. For the first five clients, the PSNR of both servers are almost equal but as from the sixth client, the difference seems to be quite significant. This is due to the addition of Server 2 which treats the sixth client as the first client in the broadcast. The least value of PSNR with one server was 39.9781dB compared to 38.9105dB with two servers. Results shows that setting up a new server for every five clients helps keeping the QoE acceptable.

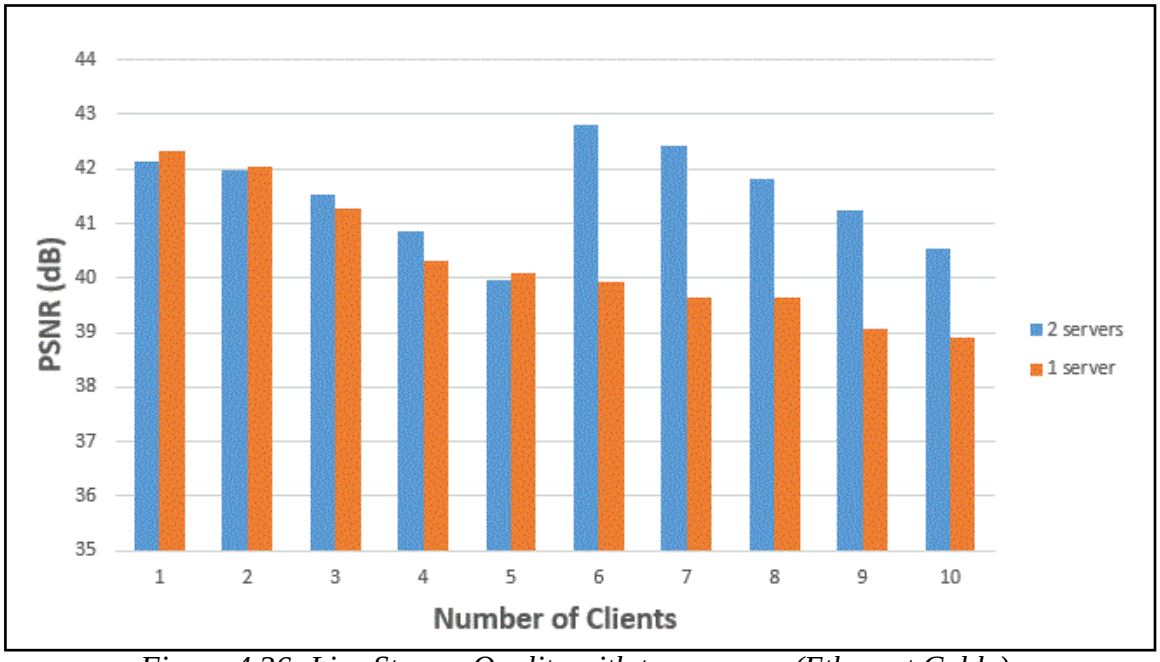

Figure 4.36: Live Stream Quality with two servers (Ethernet Cable)

4.1.6.3 File Stream quality with two servers (Regular Video)

Figure 4.37 shows the comparison between the first and the tenth client for when using two servers with five clients each.

Figure 4.37: Snapshots of File streaming with Regular Video (Ethernet Cable)

For each of the two servers used, the average PSNR values of videos being broadcasted and videos received were computed and tabulated in Table 4.23 and Table 4.24.

Number of Clients	Average PSNR
1	14.0545
2	13.9225
3	13.6841
4	13.5162
5	13.4024

Table 4.23: File Stream Quality of Regular video with Server 1 (Ethernet Cable)

Number of Clients	Average PSNR
1	14.0854
2	13.8825
3	13.7289
4	13.6401
5	13.5562

Table 4.24: File Stream Quality of Regular video with Server 2 (Ethernet Cable)

Figure 4.38 shows the comparison between using one server and two servers for a total of ten clients. For the first five clients, the PSNR of both servers are almost equal but as from the sixth client, the difference seems to be quite significant. This is due to the addition of Server 2 which treats the sixth client as the first client in the broadcast. The least value of PSNR with one server was 13.4024dB compared to 12.9112dB with two servers.

Figure 4.38: File Stream Quality of Regular video with two servers (Ethernet Cable)

4.1.6.5 File Stream quality with two servers (Still Video)

Figure 4.39 shows the comparison between the first and the tenth client for when using two servers with five clients each.

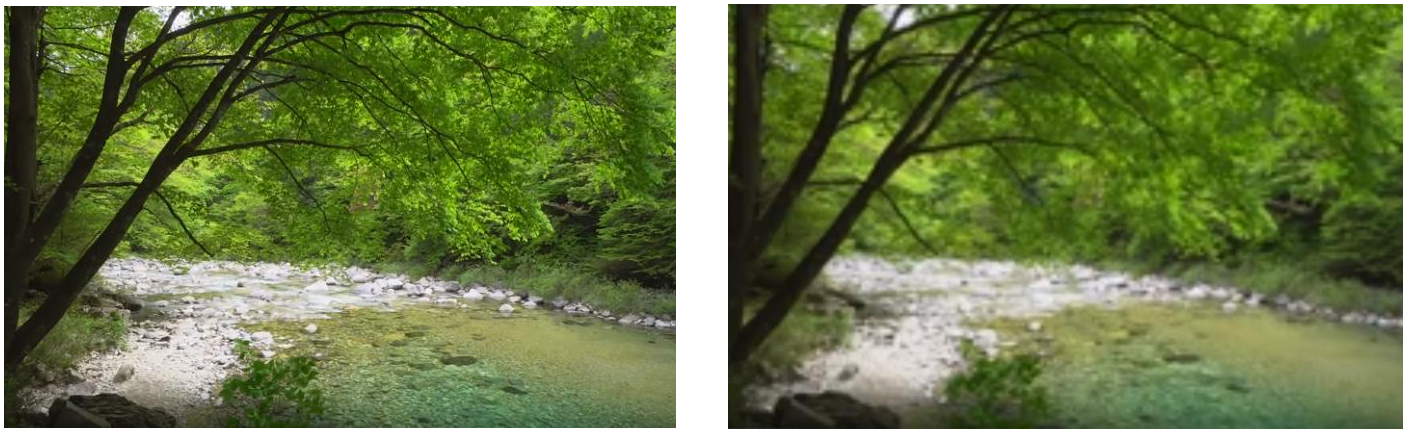

Figure 4.39: Snapshots of File streaming with Still Video (Ethernet)

For each of the two servers used, the average PSNR values of videos being broadcasted and videos received were computed and tabulated in Table 4.25 and Table 4.26.

Number of Clients	**Average PSNR**
1	41.4529
2	41.2592
3	41.1348
4	41.0525
5	40.9253

Table 4.25: File Stream Quality of Still video with Server 1 (Ethernet Cable)

Number of Clients	**Average PSNR**
1	41.8458
2	41.7542
3	41.6728
4	41.6210
5	41.5102

Table 4.26: File Stream Quality of Still video with Server 2 (Ethernet Cable)

Figure 4.40 shows the comparison between using one server and two servers for a total of ten clients. For the first five clients, the PSNR of both servers are almost equal but as from the sixth client, the difference seems to be quite significant. This is due to the addition of Server 2, which treats the sixth client as the first client in the broadcast. The least value of PSNR with one server was 40.9253dB compared to 37.8952dB with two servers.

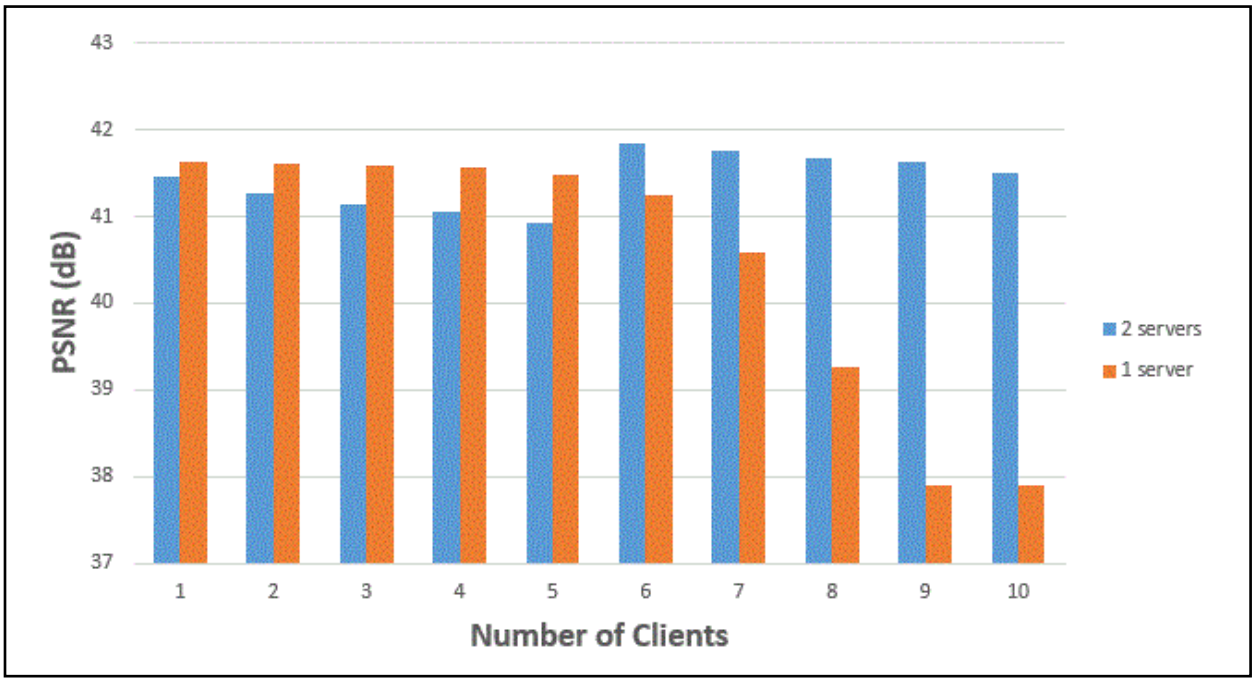

Figure 4.40: File Stream Quality of Still video with two servers (Ethernet Cable)

Chapter 5: Conclusion and future works

The aim of this thesis was to investigate how the quality of video streaming and conferencing vary as number of users increases. A classic WebRTC video conferencing and streaming application was implemented with HTML and JavaScript codes alongside Node.js as a backbone for the signaling server. The video conferencing part consisted of a video chatting system as well as a text messaging system with file sharing and supported up to four users. The video streaming part consisted of a two further sections, Live Streaming which allowed users to stream the preview of their local camera and File Streaming which allowed users to browse and stream a video file from their local PC. Conversely to the Video Conferencing part, the Video Streaming part could accommodate an unlimited number of clients. Using a recorder option, the required videos were recorded and tested with up to four clients with video conferencing and up to ten clients with video streaming on two different types of internet connection namely Wi-Fi and Ethernet cable. Moreover, an assessment was made about the QoE of videos with a shared amount of clients in two servers as opposed to assigning all clients to only one server. The PSNR metric was used as a measure of video quality and aftercomputing PSNR values of the videos using MATLAB, bar charts were generated and the values were tabulated.

The results obtained showed that video quality degrades as more and more users joined in the conference or broadcast and the trend previews an unacceptable QoS with huge amount of users. The research was based on two common internet connections: Wi-Fi and Ethernet Cable, which demarcates from previous works whereby only single schemes were tested. Both types of connection provided good video qualities but Ethernet Cabled connection proved to maintain a more stable connection where the PNSR didn’t dropped as much as for Wi-Fi connection.

Conversely to [5] where the quality of video was acceptable with up to 175 users, the thesis stated otherwise. The suggested architecture in [5] worked in such a way that QoE was not affected by the number of users joining a conference contrary to the architecture employed in this application, which is mesh topology where pipelining occurred. The trend of how video quality varied contributes a lot to the field of RTC by showing how quality of real-time video drops when connection is shared among several users. However, the most interesting contribution of this work

was to find a way of getting better QoS by launching several servers on various PCs. Each server was able to accommodate five clients while maintaining good QoS, which indicated that for a better QoE, it is best to launch various servers while streaming the same video.

There was one limitation in performance when it came to recording videos with four users in video conferencing and above eight users for video streaming since accommodating these amount of users required many processes to be executed thus, demanding very high CPU processing power [48]. These processes caused serious lagging of the videos as well as the PC itself.

The main conclusion is that the amount of users is inversely proportional to the QoS of a WebRTC video conferencing and streaming application. This conclusion is driven by the fact that connection is very limited and when several users join in, the limited connection is shared among themselves; every user gets only a part of the connection, which leads to smaller share of connection for each user as the number of users grows.

Nowadays everybody is connected to the Internet by means of mobile phones, computers, tablets and even smart television. However, the treats present on the internet are unforeseen and may lead to breach of privacy, stealing of confidential information, amongst others. Therefore, an interesting future work will be to focus more on the security side of WebRTC. Furthermore, as encountered as a limitation in the testing process of the application, CPU power proved to be a continuous dependency regarding the QoE obtained. A compelling future work will be to test the implemented application on different types of PCs, with each processing a different CPU processing power.

References

[1] "VNI Global Fixed and Mobile Internet Traffic Forecasts," Visual Networking Index, Cisco, 2017. [Online]. Available: https://www.cisco.com/c/en/us/solutions/service-provider/visual-networking-index-vni/index.html.

[2] S. Gunkel, M. Schmitt, and P. Cesar, "A QoE study of different stream and layout configurations in video conferencing under limited network conditions," In Quality of Multimedia Experience (QoMEX), 2015.

[3] D. Vucic and L. Skorin-Kapov, "The impact of mobile device factors on QoE for multi-party video conferencing via WebRTC," Telecommunications (ConTEL), 2015.

[4] D. Ammar, K. De Moor, M. Fiedler, and P. Heegaard, "Video QoE killer and performance statistics in WebRTC-based video communication," IEEE Sixth International Conference on Communications and Electronics (ICCE), 2016.

[5] B. Garcia et al., "WebRTC Testing: State of Art," WebRTC Testing: Challenges and Practical Solutions, 2017.

[6] Wang et al., "Image Quality Assessment: From Error Visibility to Structural Similarity," 2004.

[7] Yevgeniya Sulema et al., "Quality of Experience Estimation for WebRTC-based Video Streaming," 2018.

[8] "TokBox," [Online]. Available: https://en.wikipedia.org/wiki/TokBox. [Accessed 30 March 2019].

[9] Salvatore Loreto & Simon Pietro Romano, in *Real-Time Communication with WebRTC*, 2014, p. 2.

[10] "Hyperlink," [Online]. Available: https://en.wikipedia.org/wiki/Hyperlink. [Accessed 30 March 2019].

[11] G. Berndtsson, M. Folkesson, and V. Kulyk, "Subjective quality assessment of video conferences and telemeetings," 2012.

[12] J. Skowronek, F. Schiffner, and A. Raake, "On the influence of involvement on the quality of multiparty conferencing," 2013.

[13] Lea Skorin-Kapov, Mirko Suznijevi, "The impact of bandwidth limitations and video resolution size on QoE for WebRTC-based mobile multi-party video conferencing," 2016.

[14] T. Team, "Multipoint Control Unit (MCU)," [Online]. Available: https://trueconf.com/blog/wiki/multipoint-control-unit. [Accessed 2019 January 2019].

[15] G. Carlucci, L. De Cicco, S. Holmer, S. Mascolo, "Analysis and design of the google congestion control for web real-time comminication(WebRTC)," 2016.

[16] "Kalman filter," [Online]. Available: https://en.wikipedia.org/wiki/Kalman_filter. [Accessed 14 March 2019].

[17] M. Schmitt, J. Redi, P. Cesar, D. Bulterman, "1Mbps is enough: Video Quality and Individual Idiosyncrasies in Multiparty HD Video-Conferencing," 2016.

[18] "About International Telecommunication Union (ITU)," [Online]. Available: https://www.itu.int/en/about/Pages/default.aspx. [Accessed 06 Febuary 2019].

[19] Boni Garc__a _ Micael Gallego _ Francisco, "Understanding and Estimating Quality of Experience in WebRTC Applications," 2018.

[20] "Architecture," [Online]. Available: https://webrtc.org/architecture/. [Accessed 26 March 2019].

[21] Naktal Moaid Edan, A. Al-Sherbaz, Scott John Turner, "Design and implement a hybrid WebRTC Signalling mechanism for unidirectional & bi-directional video conferencing," 2018.

[22] "Mesh," [Online]. Available: https://webrtcglossary.com/mesh/. [Accessed 30 January 2019].

[23] M. Aboullaite, "Understanding JIT compiler (just-in-time compiler)," 31 August 2017. [Online]. Available: https://aboullaite.me/understanding-jit-compiler-just-in-time-compiler/.

[24] "JavaScript," 18 March 2019. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript.

[25] "Introduction to Node.js," 24 March 2019. [Online]. Available: https://nodejs.dev/.

[26] A. D. Rosa, "An Introduction to the getUserMedia API," 02 January 2014. [Online]. Available: https://www.sitepoint.com/introduction-getusermedia-api/.

[27] "RTCPeerConnection," 18 March 2019. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/RTCPeerConnection.

[28] K.T.Tokeshu, Moppers, "RTCDataChannel," May 2015. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/RTCDataChannel.

[29] V. J. Mark Handley, "SDP: Session description protocol," July 2006. [Online]. Available: http://tools.ietf.org/html/rfc4566.

[30] M. H. P. Srisuresh, "IP network address translator (NAT) terminology and considerations," August 1999. [Online]. Available: http://tools.ietf.org/html/rfc2663.

[31] S. Dutton, "Webrtc in the real world: Stun, turn and signaling," November 2013. [Online]. Available: http://www.html5rocks.com/en/tutorials/webrtc/infrastructure/.

[32] "WebRTC revolution in progress," WebRTCstats, June 2014. [Online]. Available: http://webrtcstats.com/webrtc-revolution-in-progress/.

[33] "Uniform Resource Identifier (URI)," Techopedia, [Online]. Available: https://www.techopedia.com/definition/25550/uniform-resource-identifier-uri.

[34] S. Dutton, "RTCDataChannel for chrome," Febuary 2013. [Online]. Available: http://updates.html5rocks.com/2013/02/WebRTC-data-channels-API-changes-and-Chrome-talks-to-Firefox.

[35] J. Uberti, "Javascript session establishment protocol (jsep)," December 2011. [Online]. Available: http://lists.w3.org/Archives/Public/public-webrtc/2012Jan/att-0002/JavascriptSessionEstablishmentProtocol.pdf.

[36] "Bootstrap (front-end framework)," 5 Febuary 2019. [Online]. Available: https://en.wikipedia.org/wiki/Bootstrap_(front-end_framework).

[37] Iwashi, "A Study of WebRTC Security," 2016. [Online]. Available: http://webrtc-security.github.io/.

[38] "JavaScript libraries," [Online]. Available: https://webplatform.github.io/docs/concepts/programming/javascript/libraries/. [Accessed 1 March 2019].

[39] "WebRTC 02: Many-To-Many connectivity," [Online]. Available: https://deepstreamhub.com/tutorials/protocols/webrtc-full-mesh/. [Accessed March 15 2019].

[40] "Deprecating Permissions in Cross-Origin Iframes," The Chromium Projects, [Online]. Available: https://sites.google.com/a/chromium.org/dev/Home/chromium-security/deprecating-permissions-in-cross-origin-iframes. [Accessed 24 November 2018].

[41] M. Khan, "RecordRTC.js | Live Demo," 2 Febuary 2019. [Online]. Available: https://recordrtc.org/.

[42] "socket.io-file-client," npm, 2018. [Online]. Available: https://www.npmjs.com/package/socket.io-file-client.

[43] "Star topology," Telecom ABC, [Online]. Available: http://www.telecomabc.com/s/star.html. [Accessed 6 January 2019].

[44] "Understanding Port Forwarding – Beginners Guide," 16 October 2018. [Online]. Available: https://stevessmarthomeguide.com/understanding-port-forwarding/.

[45] M. Rouse, "DMZ (networking)," June 2018. [Online]. Available: https://searchsecurity.techtarget.com/definition/DMZ.

[46] "My IP Address Is:," WhatIsMyIPAddress, [Online]. Available: https://whatismyipaddress.com/. [Accessed 30 March 2019].

[47] "PSNR," MathWorks, [Online]. Available: https://www.mathworks.com/help/vision/ref/psnr.html. [Accessed 5 January 2019].

[48] "Computer performance," 11 March 2019. [Online]. Available: https://en.wikipedia.org/wiki/Computer_performance.

APPENDIX 1

HTML CODES (index.html)

```
<!DOCTYPE html>
<head>
   <title>WebRTC Video Conference</title>
   <meta name="viewport" content="width=device-width, initial-scale=1">
   <style>
      video{
         display:block;
      }
      button{
         display:inline-block;
         margin:8px;
      }
      .hide{
         display:none !important;
      }
      #video-div{
         display: inline-block !important;
         margin-bottom: 16px;
      }

   </style>
   <link rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.min.css"
integrity="sha384ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/iJTQUOhcWr7x9JvoRxT
2MZw1T" crossorigin="anonymous">


</head>
```

```
<body>
    <!--This is the div that contains the form for typing the room number-->
    <div id="selectRoom" class="ml-2">
        <h1 class="mb-4 ml-2">WebRTC Video Conference</h1>
        <label>Type the room number</label>
        <input id="roomNumber" type="text"/>
        <button role="button" class="btn btn-primary pl-5 pr-5" id="goRoom">Go</button>

        <h1 class="mb-4 ml-2 mt-5">WebRTC Video Streaming</h1>
        <label>Type your streaming name</label>
        <input id="streamingName" type="text"/>
        <br><br>
        <label>If you want to stream a video file. Select a file.</label><br>
        <input type="file" accept="video/*" id="videoFile2Stream"><br>
        <button role="button" class="btn btn-primary pl-5 pr-5 mt-3" id="goStreaming">Live
streaming via Webcam</button><br>
        <button role="button" class="btn btn-primary pl-5 pr-5 mt-3"
id="goStreaming_local_video">Stream selected video</button>

    </div>

    <!--This is the div that contains video streams-->

    <div id="video-conference-wrapper" class=" hide">
        <div class="row" style="padding:0px;margin:0px">

            <div class="col-sm-12 col-md-12 col-lg-9 col-xl-9" id="video-col"
style="height:100vh;overflow-y: auto">
                <div id="consultingRoom" style="display:none">
```

```
            <div id="video-div" class="localVideoDiv">
                <video id="localVideo" class="mb-3" autoplay></video>
                <button role="button" class="btn btn-primary" id="start-recording-local-video">Start Recording</button>
                <button role="button" class="btn btn-danger hide" id="stop-recording-local-video">Stop Recording</button>
                <button role="button" class="btn btn-success hide" id="download-recording-local-video">Download Recording</button>
            </div>
        </div>
    </div>

    <div id="chat-col" class="col-sm-12 col-md-12 col-lg-3 col-xl-3">
        <div id="chat-message-box" style="height:50vh;background:#eee;width:100%;overflow-y:auto">

        </div>
        <div id="chat-messeger-input-box">
            <form onsubmit="event.preventDefault();sendChatMessage()" style="width:100%" id="chat-message-form">
                <div class="d-flex align-items-center justify-content-between">
                    <label>You Chat Name</label>
                    <input required id="chat-name-input" class="ml-2" placeholder="Enter your chat name..."/>
                </div>

                <textarea required class="form-group d-block p-1" style="width:100%" rows="5" placeholder="Write your message here..." id="chat-message-input"></textarea>
                <button type="submit" class="btn btn-primary">Send</button>
            </form>
```

```
            <div class="file-upload-div mt-3 mb-5">
                <p>Share a file</p>
                <form id="file-upload-form">
                    <input type="file" id="to-upload-file" multiple class="mb-3"/>
                    <input type="submit" value="Upload" class="btn btn-success"/>
                    <div class="spinner-border hide" id="file-upload-spinner" role="status">
                        <span class="sr-only">Loading...</span>
                    </div>
                </form>
            </div>


        </div>
    </div>


</div>
</div>


<!--These are the required javascripts libraries-->
<script src="https://code.jquery.com/jquery-3.3.1.slim.min.js" integrity="sha384-
q8i/X+965DzO0rT7abK41JStQIAqVgRVzpbzo5smXKp4YfRvH+8abtTE1Pi6jizo"
crossorigin="anonymous"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.14.7/umd/popper.min.js"
integrity="sha384-
UO2eT0CpHqdSJQ6hJty5KVphtPhzWj9WO1clHTMGa3JDZwrnQq4sF86dIHNDz0W1"
crossorigin="anonymous"></script>
<script src="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/js/bootstrap.min.js"
integrity="sha384-
JjSmVgyd0p3pXB1rRibZUAYoIIy6OrQ6VrjIEaFf/nJGzIxFDsf4x0xIM+B07jRM"
crossorigin="anonymous"></script>
<script src="socket.io-file-client.js"></script>
```

```
    <script src="RecordRTC.min.js"></script>
    <script src="/socket.io/socket.io.js"></script>
    <script src="client.js"></script>
</body>
```

APPENDIX 2

Client Code (Client.js)

```
//here we get a reference to the webpage elements
var divSelectRoom = document.getElementById("selectRoom");
var divConsultingRoom = document.getElementById("consultingRoom");
var inputRoomNumber = document.getElementById("roomNumber");
var inputStreamingName = document.getElementById("streamingName");
var btnGoRoom = document.getElementById("goRoom");
var btnGoStreaming = document.getElementById("goStreaming");
var btnGoStreaming_localVideo = document.getElementById("goStreaming_local_video");
var localVideo = document.getElementById("localVideo");
var fileUploadForm = document.getElementById("file-upload-form");

//these are the global variables
var streamingName;
var roomNumber;
var localStream;
var remoteStreams = {};
var RTCPeerConnection = {};  // all the rtc connections
var currentRemoteConnections = 0;
var maxRecordingTimeMS = 300000;  // DEFAULT maximun recording time of a video stream in Milliseconds
var isStreamingLocalVideo = false; // FALSE if the user is using the web cam video to stream.
var isVideoConference = false; // FALSE is the set up is using the video/ file streaming

//var connections = {};
var socketID = null;
```

```
var rtcPeerConnection;
```

//these are the STUN servers

```
var iceServers={
    'iceServers':[
        {'urls':'stun:stun.services.mozilla.com'},
        {'urls':'stun:stun.1.google.com:19302'},
        {
            'urls': 'turn:numb.viagenie.ca',
            'username' : 'saxkwi12@outlook.com',
            'credential' : '123456',
        }
    ]
};
var streamConstraints = {audio:true, video:true};
var isCaller;
```

//Here we connect to the socket.io server

```
var socket = io();
```

//Here we add a click event to the button

```
btnGoRoom.onclick = function(){
    if (inputRoomNumber.value === ""){
        alert("Please type a room number");
    }else{
        roomNumber = inputRoomNumber.value;//we take the value from the element
        socket.emit('create or join', roomNumber);//we send a message to server
    }
};
```

```
// when a new client joins the room
socket.on('create', function(room, id, clients){
    $('#video-conference-wrapper').removeClass('hide');
    divSelectRoom.style = "display:none";       //hide selectRoom div
    divConsultingRoom.style = "display:block";  //show consultingRoom div
    socketID = socket.id;
    $("#chat-name-input").val(socketID);
    isVideoConference = true;


    navigator.mediaDevices.getUserMedia(streamConstraints).then(function(stream){
        // get and set local media stream
        localStream = stream;
        localVideo.srcObject = stream;


        attachRecorder(stream, "recording-local-video");


        if(clients.length < 2){return;}
        for(var i = 0; i < clients.length; i++){
            // for all the clients in a room except yourself
            var socketListID = clients[i];
            if(socketListID == socketID){continue;}
            let slid = socketListID;    // (target client) using let to maintain the state of variable
asyncronusly
            if(!RTCPeerConnection[socketListID]){
                // when a new connection is created
                RTCPeerConnection[socketListID] = new RTCPeerConnection(iceServers);
                RTCPeerConnection[socketListID].onicecandidate = function(event){
                    // when this connection's candidates arrive
                    // send them to the other client who has been
```

```
            // sent an offer by this connection.
            if (event.candidate){
                //console.log("CON_ -> sending ice candidate");
                socket.emit('con_candidate',
                    roomNumber, // my roomNumber
                    socketID,   // my socketID
                    slid,       // socketID of the target client
                    {
                        type:'candidate',
                        label:event.candidate.sdpMLineIndex,
                        id:event.candidate.sdpMid,
                        candidate:event.candidate.candidate,
                        room:roomNumber
                    }
                );
            }
        }


        RTCPeerConnection[socketListID].onaddstream = function(event){
            // when a stream is added to the connection.
            $("#consultingRoom").append(
                makeRemoteVideoElement(currentRemoteConnections)
            )
            var _remoteVideo = document.getElementById("remoteVideo" +
currentRemoteConnections);
            _remoteVideo.srcObject = event.stream;
            remoteStreams[slid] = {stream: event.stream, htmlVideoObject : $('.video-div-' +
currentRemoteConnections)};
```

```
                attachRecorder(event.stream, 'recording-remote-video-'+
currentRemoteConnections);
                currentRemoteConnections += 1;
            };

            RTCPeerConnection[socketListID].addStream(localStream);

            RTCPeerConnection[socketListID].createOffer(function (sessionDescription){
                // creating an offer and sending it to the
                // target client
                RTCPeerConnection[slid].setLocalDescription(sessionDescription);
                socket.emit('con_offer', roomNumber, socketID, slid, {
                    type:'offer',
                    sdp:sessionDescription,
                    room:roomNumber
                });
            }, function(e){console.log(e);});
        }
    }
  }).catch(function(err){
  console.log("An error occured when accessing media devices");
  });
});


// when an offer from a newly connected client arrives
socket.on('con_offer', function(room, fromID, toID, data){

   //console.log('con_offer');
```

```
  if(toID != socketID){return;}
  if(RTCPeerConnection[fromID]){return;}
  let fid = fromID; // (target client who offered a connection) using let to maintain the state of variable asyncronusly
  RTCPeerConnection[fromID] = new RTCPeerConnection(iceServers);
  RTCPeerConnection[fromID].onicecandidate = function(event){
    if (event.candidate){
      // when the candidates of this client arrive.
      // send them to the client who offered a webrtc connection
      console.log("CON_ -> sending ice candidate");
      socket.emit('con_candidate',
        roomNumber, // my room number
        socketID,   // my socketID
        fid,        // socket id of target client who offered a connection
        {
          type:'candidate',
          label:event.candidate.sdpMLineIndex,
          id:event.candidate.sdpMid,
          candidate:event.candidate.candidate,
          room:roomNumber
        }
      );
    }
  };
  RTCPeerConnection[fromID].onaddstream = function(event){
    // when a stream is added to the connection.
    $("#consultingRoom").append(
      makeRemoteVideoElement(currentRemoteConnections)
    )
```

```
        var _remoteVideo = document.getElementById("remoteVideo" +
currentRemoteConnections);
        _remoteVideo.srcObject = event.stream;
        remoteStreams[fid] = {stream: event.stream, htmlVideoObject : $('.video-div-' +
currentRemoteConnections)};
        attachRecorder(event.stream, 'recording-remote-video-'+ currentRemoteConnections);
        currentRemoteConnections += 1;
    };
    RTCPeerConnection[fromID].addStream(localStream);
    RTCPeerConnection[fromID].setRemoteDescription(new RTCSessionDescription(data.sdp));

    RTCPeerConnection[fromID].createAnswer(function (sessionDescription){
        // create an answer and send it to the target client who offered a connection
        RTCPeerConnection[fid].setLocalDescription(sessionDescription);
        socket.emit('con_answer',roomNumber, socketID, fid, {
            type:'answer',
            sdp:sessionDescription,
            room:roomNumber
        });
    }, function(e){console.log(e);});

});
```

// when the answer from the offering entity comes.

```
socket.on('con_answer', function(room, fromID, toID, data){
    console.log('con_answer')
    if(socketID != toID) {return;}
    RTCPeerConnection[fromID].setRemoteDescription(new RTCSessionDescription(data.sdp));
```

```
});


// when the candidata from the other connection arrives.
socket.on('con_candidate', function(room, fromID, toID, data){
    //create a candidate object
    if(socketID != toID) {return;}
    var candidate = new RTCIceCandidate({
        sdpMLineIndex:data.label,
        candidate:data.candidate
    });
    //stores candidate
    if(RTCPeerConnection[fromID]){
        RTCPeerConnection[fromID].addIceCandidate(candidate);
    }else{
        console.log('Error: fromID (' + fromID + ') does not exist in connections');
    }
});


socket.on('full', function(room){
    // when the max number of client have reached
    alert('Cannot connect to this room')
});
socket.on('disconnect', function(sid){
    if(!remoteStreams[sid]){return;}
    delete RTCPeerConnection[sid];
    remoteStreams[sid].htmlVideoObject.remove();
    delete remoteStreams[sid];
```

```
    delete RTCPeerConnection[sid]
});

//  when a chat message arrives from the server
socket.on('chat_message', function(room, fromID, message){
    updateChatMessageBox(message);
});

//  send a chat message to the server to be broadcasted and
//  a copy of it is also shown on the current client's page.
function sendChatMessage(){
    var chatMsg = $("#chat-message-input").val();
    if(chatMsg === ""){alert("Please enter a chat message...");return;}
    $("#chat-message-input").val("");
    var formatedMessage = {
        id: socketID,
        name: $("#chat-name-input").val() === "" ? socketID : $("#chat-name-input").val(),      // the public chat name
        content: chatMsg,
    }
    socket.emit('chat_message', roomNumber, socketID, formatedMessage);
    updateChatMessageBox(formatedMessage);

}

//  update the message box in the HTML
function updateChatMessageBox(message){
    $("#chat-message-box").append(makeChatItem(message));
    var messageBody = document.querySelector('#chat-message-box');
```

```
    messageBody.scrollTop = messageBody.scrollHeight - messageBody.clientHeight;
}
```

```
//  HTML template for the chat message.
function makeChatItem(message){
    return `
        <div class="chat-item m-3" style="border-bottom:1px solid #ccc">
            <p class="mb-1" style="font-size:14px;color:#343434"><strong>` + message.name +
`</strong></p>
            <p>` + message.content + `</p>
        </div>
    `;
}
```

```
//  This function makes the remote video element
//  as per unique id which is given.
function makeRemoteVideoElement(unique_id){
    var videoWidth = "";
    var videoHeight = "";
    if(isVideoConference){
        // if the video conference application is being used.
        var crHeight = $(window).innerHeight();
        var crWidth = $('#consultingRoom').innerWidth();

        var rvWidth = crWidth / 2 - 50;
        var rvHeight = crHeight / 2 - 100;

        videoHeight = "height="'+ rvHeight +'"";
        //videoWidth = "width="'+ rvWidth +'"";
```

```
        $("#localVideo").attr('height', rvHeight);
    }


    return `
    <div id="video-div" class="video-div-` + unique_id + `">
        <video class="mb-3" id="remoteVideo` + unique_id + `" autoplay ${videoHeight}
${videoWidth}></video>
        <button role="button" class="btn btn-primary" id="start-recording-remote-video-`+
unique_id +`">Start Recording</button>
        <button role="button" class="btn btn-danger hide" id="stop-recording-remote-video-`+
unique_id +`" >Stop Recording</button>
        <button role="button" class="btn btn-success hide" id="download-recording-remote-video-
`+ unique_id +`" >Download Recording</button>
    </div>
    `;
}
```

```
//  This function attaches the recording functionality to the stream.
//  the function inputs are the media stream and the
//  generic name of the start, stop and download button.
```

```
function attachRecorder(stream, genericName){
    let _stream = stream;   // scope specfic stream
    let startBtn = $("#start-" + genericName);
    let stopBtn = $("#stop-" + genericName);
    let downloadBtn = $("#download-" + genericName);


    // when start button is clicked
    startBtn.on('click', function(){
        startBtn.addClass('hide');
```

```
// make the recorder object
let recorder = RecordRTC(stream,{
    type : 'video',
});
// start recording of the video stream
recorder.startRecording();
// if the recording exceeds the max recording time the automatically
// stop recording and generate a download able file and set it to the
// download link.
let recorderTimeout = setTimeout(function(){
    stopBtn.addClass('hide');
    stopBtn.off('click');
    downloadBtn.removeClass('hide');
    recorder.stopRecording(function(){
        let blob = recorder.getBlob(); // scope level video content blob.
        // when the download button is clicked.
        downloadBtn.on('click', function(){
            startBtn.removeClass('hide');
            downloadBtn.addClass('hide');
            downloadBtn.off('click');
            invokeSaveAsDialog(blob);
        });
    });
}, maxRecordingTimeMS);
stopBtn.removeClass('hide');
// if the stop button is clicked before the default timeout then
// stop the recorder timeout adn download the video file.
stopBtn.on('click', function(){
```

```
            clearTimeout(recorderTimeout);
            recorder.stopRecording(function(){
                let blob = recorder.getBlob();
                BLOB = blob;
                //var newblob = new Blob(blob, {type: 'video/mp4'});
                //console.log(newblob);
                invokeSaveAsDialog(blob);


                stopBtn.addClass('hide');
                stopBtn.off('click');
                startBtn.removeClass('hide');
            });
        })
    })
}
```

// the file upload system

```
var uploader = new SocketIOFileClient(socket);
```

// when the file starts uploading

```
uploader.on('start', function(fileInfo) {
    $('#file-upload-spinner').removeClass('hide');
    console.log('Start uploading', fileInfo);
});
```

// when the file starts to get streamed to the server

```
uploader.on('stream', function(fileInfo) {
    console.log('Streaming... sent ' + fileInfo.sent + ' bytes.');
```

```
});
```

// when the file upload has been completed.

// remove the spinner

// make a message

// broad cast the message to all the clients

```
uploader.on('complete', function(fileInfo) {
    $('#file-upload-spinner').addClass('hide');
    console.log('Upload Complete', fileInfo);
    var fname = fileInfo.name;
    var formatedMessage = {
        id: socketID,
        name: $("#chat-name-input").val() === "" ? socketID : $("#chat-name-input").val(),      // the public chat name
        content: `<p>Download File: <a target="_blank" href="/dataChat_vidConf/` + fname + `">See File</a></p>`,
    }
    socket.emit('chat_message', roomNumber, socketID, formatedMessage);
    updateChatMessageBox(formatedMessage);
});
```

// when the file uploading encounters an error

```
uploader.on('error', function(err) {
    $('#file-upload-spinner').addClass('hide');
    alert('There was an error, while uploading the file. Please upload the file again');
    console.log('Error', err);
});
```

// when the file uploading gets aborted

```
uploader.on('abort', function(fileInfo) {
    $('#file-upload-spinner').addClass('hide');
    console.log('Aborted: ', fileInfo);
});
```

// when the file form gets submitted,

```
fileUploadForm.onsubmit = function(event){
    event.preventDefault();
    var fileEl = document.getElementById('to-upload-file');
    uploader.upload(fileEl);
};
```

// Video streaming system

```
String.prototype.count=function(s1) {
    return (this.length - this.replace(new RegExp(s1,"g"), '').length) / s1.length;
}
```

// get the name of the streaming channel

```
function getStreamingName(){
    var pn = window.location.pathname;
    pn = pn.substr(1);
    if(pn == "") return null;
    if(pn.count('/') > 1) return null;
    return pn;
}
```

// button to create the streaming channel

```
btnGoStreaming.onclick = function(){
```

```
    if(inputStreamingName.value === ""){
        alert("Please type a room number");
    }else{
        streamingName = inputStreamingName.value;
        roomNumber = streamingName;
        socket.emit('create_stream', streamingName);
        isStreamingLocalVideo = false;
    }
}
```

```
// button to create the streaming channel if the user want to stream the
// local video.
if(btnGoStreaming_localVideo){
    btnGoStreaming_localVideo.onclick = function(){
        if(inputStreamingName.value === ""){
            alert("Please type a room number");
        }else if(!document.getElementById("videoFile2Stream").files[0]){
            alert("Please select a video file");
        }
        else{
            streamingName = inputStreamingName.value;
            roomNumber = streamingName;
            socket.emit('create_stream', streamingName);
            isStreamingLocalVideo = true;
        }
}
}
```

```
// if an error for streaming is received
socket.on("error_stream", function(error){
    alert(error);
});

// creating the stream
socket.on('create_stream', function(streamName){
    $('#video-conference-wrapper').removeClass('hide');
    divSelectRoom.style = "display:none";       //hide selectRoom div
    divConsultingRoom.style = "display:block";  //show consultingRoom div
    socketID = socket.id;
    $("#chat-name-input").val(socket.id);
    $(".localVideoDiv").attr('style', 'display:block !important');
    $("#localVideo").attr('style', 'width:100%;height:93vh');
    $('#chat-col').addClass('hide');
    $('#video-col').removeClass('col-lg-9 col-xl-9');
    //$("#chat-message-form").addClass('hide');

    $('.localVideoDiv').append('Streaming Channel is :    ' + window.location.href + streamName);

    if(isStreamingLocalVideo){
        var file = document.getElementById("videoFile2Stream").files[0];
        localVideo.src = URL.createObjectURL(file);
        if(navigator.userAgent.indexOf('Firefox') > -1){
            // using fire fox
            localStream = localVideo.mozCaptureStream();
        }else{
            // using other browser
```

```
            localStream = localVideo.captureStream();
        }
        attachRecorder(localStream, "recording-local-video");
        return;
    }

    navigator.mediaDevices.getUserMedia(streamConstraints).then(function(stream){
        // get and set local media stream
        localStream = stream;
        localVideo.srcObject = stream;
        attachRecorder(stream, "recording-local-video");



    }).catch(function(err){
    console.log("An error occured when accessing media devices");
    });
});
// when a client requests to join the stream make an offer to it
socket.on('request_join_stream', function(_channel, _fromID, _toID){
    let socketID = socket.id;
    let fromID = _fromID;
    let toID = _toID;
    let channel = _channel;
    if(toID != socketID){return;}
    if(RTCPeerConnection[fromID]){return;}
    RTCPeerConnection[fromID] = new RTCPeerConnection(iceServers);
    RTCPeerConnection[fromID].onicecandidate = function(event){
        // when this connection's candidates arrive
```

```
    // send them to the other client who has been
    // sent an offer by this connection.
    if (event.candidate){
        //console.log("CON_ -> sending ice candidate");
        socket.emit('con_candidate',
            channel,    // channel
            socketID,   // my socketID
            fromID,       // socketID of the target client
            {
                type:'candidate',
                label:event.candidate.sdpMLineIndex,
                id:event.candidate.sdpMid,
                candidate:event.candidate.candidate,
                room:roomNumber
            }
        );
    }
}
RTCPeerConnection[fromID].addStream(localStream);
RTCPeerConnection[fromID].createOffer(function (sessionDescription){
    // creating an offer and sending it to the
    // target client
    RTCPeerConnection[fromID].setLocalDescription(sessionDescription);
    socket.emit('con_offer_stream', channel, socketID, fromID, {
        type:'offer',
        sdp:sessionDescription,
        room:channel
    });
```

```
    }, function(e){console.log(e);});


});
```

// when an offer from the streaming entity

```
socket.on('con_offer_stream', function(_channel, _fromID, _toID, _data){
    let socketID = socket.id;
    let fromID = _fromID;
    let toID = _toID;
    let channel = _channel;
    let data = _data;
    if(toID != socketID){return;}
    if(RTCPeerConnection[fromID]){return;}
    RTCPeerConnection[fromID] = new RTCPeerConnection(iceServers);
    RTCPeerConnection[fromID].onicecandidate = function(event){
        if (event.candidate){
            // when the candidates of this client arrive.
            // send them to the client who offered a webrtc connection
            socket.emit('con_candidate',
                channel, // my room number
                socketID,   // my socketID
                fromID,        // socket id of target client who offered a connection
                {
                    type:'candidate',
                    label:event.candidate.sdpMLineIndex,
                    id:event.candidate.sdpMid,
                    candidate:event.candidate.candidate,
                    room:roomNumber
```

```
            }
        );
    }
};
RTCPeerConnection[fromID].onaddstream = function(event){
    // when a stream is added to the connection.
    $("#consultingRoom").append(
        makeRemoteVideoElement(currentRemoteConnections)
    )
    var _remoteVideo = document.getElementById("remoteVideo" +
currentRemoteConnections);
    _remoteVideo.srcObject = event.stream;
    remoteStreams[fromID] = {stream: event.stream, htmlVideoObject : $('.video-div-' +
currentRemoteConnections)};
    attachRecorder(event.stream, 'recording-remote-video-'+ currentRemoteConnections);
    currentRemoteConnections += 1;
};

RTCPeerConnection[fromID].setRemoteDescription(new RTCSessionDescription(data.sdp));
RTCPeerConnection[fromID].createAnswer(function (sessionDescription){
    // create an answer and send it to the target client who offered a connection
    RTCPeerConnection[fromID].setLocalDescription(sessionDescription);
    socket.emit('con_answer_stream',channel, socketID, fromID, {
        type:'answer',
        sdp:sessionDescription,
        room:channel
    });
}, function(e){console.log(e);});
```

```
});
```

// when the answer from the offering entity comes.

```
socket.on('con_answer_stream', function(channel, fromID, toID, data){
    console.log('con_answer_stream')
    if(socketID != toID) {return;}
    RTCPeerConnection[fromID].setRemoteDescription(new RTCSessionDescription(data.sdp));
});
```

// when the window is loaded check if the url corresponds to that for the

// joining with the streaming channel. If it is do the necessary work.

```
var streamName = getStreamingName();
if(streamName){
    roomNumber = streamName;
    streamingName = streamName;
    socket.emit('join_stream', streamName);
    $('#video-conference-wrapper').removeClass('hide');
    divSelectRoom.style = "display:none";       //hide selectRoom div
    divConsultingRoom.style = "display:block";  //show consultingRoom div
    $(".localVideoDiv").addClass('hide');
    $("#chat-messeger-input-box").addClass('hide');
}
```

APPENDIX 3

Server (Server.js)

//requiring libraries

```
const fs = require('fs');
const express = require('express');
const app = express();
const SocketIOFile = require('socket.io-file');
const path = require('path');


var privateKey = fs.readFileSync('certificates/key.pem', 'utf8');
var certificate = fs.readFileSync('certificates/cert.pem', 'utf8');


var credentials = {key: privateKey, cert:certificate};


var https = require('https').Server(credentials, app);;
var io = require('socket.io')(https);


var chatUploadFilePath = "public/dataChat_vidConf/";
var fileUploadCount = 0;
var clientFileDB = {};
var videoStreamsDB = {};    // the database for keeping track of all the streaming name, their sources and their clients
var maxClients = 4;
```

//static hosting using express

```
app.set('view engine', 'pug')
app.set('views', path.join(__dirname, '/public'));
```

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


app.get('/:streamingName', function(req, res){
   res.render('index2', {streamingName : req.param.streamingName});
});

//signaling handlers
io.on('connection', function(socket){
   console.log('a user connected');


   //when client emits create or join
   socket.on('create or join', function(room){
      console.log('create or join', room);


      if(videoStreamsDB[room]){
         socket.emit('error_stream', 'Error! please select another room name. There already exists a room with this name in the system');
         return;
      }

      //count number of users on room
      var myRoom = io.sockets.adapter.rooms[room]||{length:0};
      var numClients = myRoom.length;
      console.log('Room ', room, ' has ', numClients, ' clients');


      if(numClients < maxClients){
         socket.join(room);
```

```
            socket.emit('create', room, socket.id, numClients != 0 ? Object.keys(myRoom.sockets) :
[]);
        }else{
            socket.emit('full', room);
        }
    });

    socket.on('con_offer', function(room, fromID, toID, data){
        socket.broadcast.to(room).emit('con_offer', room, fromID, toID, data);
    });

    socket.on('con_answer', function(room, fromID, toID, data){
        socket.broadcast.to(room).emit('con_answer', room, fromID, toID, data);
    });

    socket.on('con_candidate', function(room, fromID, toID, data){
        socket.broadcast.to(room).emit('con_candidate', room, fromID, toID, data);
    });

    socket.on('chat_message', function(room, fromID, message){
        socket.broadcast.to(room).emit("chat_message", room, fromID, message);
    });

    socket.on('disconnect', function(){
        // when the client disconnects from the session

        io.sockets.emit('disconnect', socket.id);
        // delete all the file which are uploaded by the client
        // when he disconnects.
```

```
    var streamingChannels = Object.keys(videoStreamsDB);
    for(var i = 0; i < streamingChannels.length; i++){
        if(videoStreamsDB[streamingChannels[i]].source == socket.id){
            delete videoStreamsDB[streamingChannels[i]];
        }
    }

    var clientFiles = clientFileDB[socket.id];
    if(!clientFiles){return;}
    for(var i = 0; i < clientFiles.length; i++){
        var element = clientFiles[i];
        fs.unlink(chatUploadFilePath + element, function(){});
    }
})

// File upload system.
var uploader = new SocketIOFile(socket, {
    uploadDir: chatUploadFilePath,
    chunkSize: 10240,
    transmissionDelay: 0,
    overwrite: true,
    rename: function(filename, fileinfo){
        // renaming the file before it is stored in the database
        var file = path.parse(filename);
        var fname = file.name;
        var ext = file.ext;
        return `${fname}_${socket.id}_${fileUploadCount++}.${ext}`;
    }
```

```
});

// when the file uploading start
uploader.on('start', (fileInfo) => {
    console.log('Start uploading');
    console.log(fileInfo);
});

// when the file starts streaming to the server
uploader.on('stream', (fileInfo) => {
    console.log(`${fileInfo.wrote} / ${fileInfo.size} byte(s)`);
});

// when the file upload had be completed
uploader.on('complete', (fileInfo) => {
    console.log('Upload Complete.');
    console.log(fileInfo);
    // keeping track of which file is uploaded by which client
    if(clientFileDB[socket.id]){
        clientFileDB[socket.id].push(fileInfo.name);
    }else{
        clientFileDB[socket.id] = [fileInfo.name];
    }
});

// when the file uploading encounters an error
uploader.on('error', (err) => {
    console.log('Error!', err);
```

```
    });

    // when the file uploading is aborted
    uploader.on('abort', (fileInfo) => {
        console.log('Aborted: ', fileInfo);
    });

    // video streaming system
    socket.on('create_stream',function(streamingName){
        // check if the the channel/ room name is available
        var myRoom = io.sockets.adapter.rooms[streamingName]||{length:0};
        var numClients = myRoom.length;
        if(videoStreamsDB[streamingName] || numClients != 0){
            socket.emit('error_stream', 'Error! This streaming name has already been taken please pick a new name.');
            return;
        }
        videoStreamsDB[streamingName] = {
            source : socket.id,
            clients : [],
        };
        socket.join(streamingName);
        socket.emit('create_stream', streamingName);
    })

    // when the server receives the join stream command
    socket.on('join_stream', function(streamingName_to_connect){
        // HERE streamingName_to_connect == channel
        var channel = streamingName_to_connect;
```

```
        if(!videoStreamsDB[channel]){
            socket.emit('error_stream', 'Error! Cannot connect to a channel which does not exist. Please, try a valid channel name');
        }else{
            videoStreamsDB[channel].clients.push(socket.id);
            socket.join(channel);
            socket.broadcast.to(channel).emit('request_join_stream', channel, socket.id, videoStreamsDB[channel].source);
        }
    })

    // when the server receives the offer of the streaming entity
    socket.on('con_offer_stream', function(channel, fromID, toID, data){
        socket.broadcast.to(channel).emit('con_offer_stream', channel, fromID, toID, data);
    });

    // when the server receives the answer from the receiver entity
    socket.on('con_answer_stream', function(channel, fromID, toID, data){
        socket.broadcast.to(channel).emit('con_answer_stream', channel, fromID, toID, data);
    });

});

var LANAccess = "0.0.0.0";
//listener
https.listen(3000, LANAccess);
console.log('listening on https://localhost:3000');
```