NTNU

Norwegian University of
Science and Technology

Research on Development and Evaluation of WebRTC Signaling based on XMPP

Chun Fan

Master of Telematics - Communication Networks and Networked Services
Submission date: June 2017
Supervisor: Min Xie, IIK

Norwegian University of Science and Technology
Department of Information Security and Communication Technology

NTNU – Trondheim
Norwegian University of
Science and Technology

Research on Development and Evaluation of WebRTC Signaling based on XMPP

Chun Fan

Submission date: June 2017
Responsible professor: Min Xie, ITEM
Supervisor: Min Xie, ITEM

Norwegian University of Science and Technology
Department of Telematics

Abstract

Web is becoming more and more popular. Web Real-time Communication (WebRTC), as one of the new technologies based on Web, makes real-time communication easy to be used through the Web. However, WebRTC is not a standardization responsible for specifying protocols. Therefore, there are many solutions proposed in WebRTC signaling. JavaScript Object Notation (JSON) and EXtensible Markup Language (XML) via XMLHttpRequest (XHR), both of them support defining objects and transferring data, and thus can be used in solutions of implementing signaling. HTTP Comet and WebSocket, that enable pushing data from server to client, can be good solutions for implementing signaling. Mature protocols such Session Initiation Protocol (SIP) and Extensible Messaging and Presence Protocol (XMPP) are also good solutions. Moreover, combination of XMPP, SIP, Jingle or constructing own protocol carrying Session Description Protocol-like (SDP -like) information can also work for signaling.

XMPP as a mature protocol, which is open standard for messaging and presence, is one of those proposed signaling solutions for WebRTC. As XMPP has features of openness, standardization, provenness, decentralized, security, flexibility and diversity, it is chosen as solution for WebRTC signaling to be studied.

In order to verify whether XMPP is a valid solution for WebRTC signaling, this project develops WebRTC application with two solutions for WebRTC signaling. The first one is a straightforward way of using XMPP whereas the second one introduces modifications targeting to improve the performance.

The project first designed and implemented one model that WebRTC supported browser connects to XMPP server directly which is the straightforward way to achieve. Since XMPP is a heavy text based protocol which may lead to too much traffic through the Internet, this project designed and implemented another model that building a middleware server on the same sever as XMPP server. So WebRTC browser communicates with the middleware sever with core information and the middleware server communicates with XMPP server with XMPP supported message, and thus the majority of traffic are moved from Internet to local. It is then expected that the latter model may improve performance of delay for WebRTC signaling.

In addition, to analyze two models, the validation methods have also been implemented. These methods includes setting timestamps according to different steps in signaling process, capturing and computing data flow size

according to messages sent and received during signaling process, and also recording number of candidates which affect the performance of signaling process.

Besides, many technologies have been studied and applied during implementations, as well as many challenges.

Analysis was then conducted after implementation. It is demonstrated that XMPP as WebRTC signaling method can be a valid solution, since call can be set up successfully between users. In addition, methods of measurement including recoding timestamp, data flow size and number of candidates also work fine, since all data which is needed are recored. It also finds out in quantity that building a middleware server together with XMPP server can move most data from Internet to local. However, building a middleware server can not guarantee less delay. But the delay is more stable in the model with middlware than in the model with direct connection to XMPP server. Interactive Connectivity Establishment candidate (ICE candidate) as an important role in WebRTC signaling, which is an object containing information for establishing communication between peers, affects much on delay performance of WebRTC signaling. More research on ICE candidate can help to improve the delay performance of WebRTC signaling.

WebRTC is an interesting topic and it is proved by this project that WebRTC is a nice technology for peer to peer communication, and XMPP works fine as a solution for WebRTC signaling.

Preface

Once upon a time, there was a summer job proposed by Telenor. The summer job is about WebRTC and it was my first time hear about WebRTC. I got a chance to the interview and the interviewer is Min Xie. Though the summer job was canceled since budget, I tried to work as a volunteer for the research on a product from Telenor. The product is called Appear.in which is based on WebRTC. During half year together with WebRTC, I felt a great interest in WebRTC. And therefore, I chose WebRTC as my master thesis topic.

Thanks very much that I met Min and Min gave so many advice and suggestions not only about study but also research. She helps me a lot to improve my ability of researching. She pointed out my strengths and enhance them while found out my weaknesses and strengthen them. Also she taught me how to design methodology to conduct my research, how to find out possibilities to make deep research and also how to write a good research report. Those instructions help really a lot.

Meanwhile, I also want to thank my former girlfriend who is my wife now. She has been supporting my study even though we have lived with a long distance for many years. She gives me love and courage. This makes me do my best effort on my work.

I also have to thank my family, especially my parents. Without my family, I can not support to study abroad. My parents always want to give me the best opportunities in education. I knew it was big stress for them to make the decision to send me here in NTNU for my further study. However, I believe that this is the greatest decision they ever made that changes my future.

In addition, I want to thank all my friends. They make my life enjoyable. Without them, I cannot make delicious food, I cannot take nice pictures, I cannot sing wonderful songs, etc.

Thank you all!

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1
Introduction

This chapter gives an overview of this thesis. As Web is one of the most popular ways to surf the Internet, it becomes part of people‘s life for 47% population in the world and 79% population in Europe in 2016[3]. Web real-time communication (WebRTC) as a Web based real-time communication technology provides people more convenient possibilities to get in touch with other people through the Internet. However, a new generation of technology could not always be perfect. It therefore has many existing or potential problems. This chapter then describes a set of problems which may occur in WebRTC. Meanwhile, objectives to achieve of this thesis are also included.

1.1 Overview

The World Wide Web (abbreviated WWW or the Web) is an information space where Uniform Resource Locators (URLs) identify documents and other web resources, interlinked by hypertext links, and can be accessed via the Internet.[4] It is quite convenient to surf the Internet as long as you have a browser installed on your device, no matter it is a PC or a mobile. Therefore technologies based on Web would provide big potential nowadays.

WebRTC is one of the technologies. It enables rich, high-quality RTC applications to be developed for browser, mobile platforms, and IoT devices, and allow them all to communicate via a common set of protocols.[5]

WebRTC makes real-time communication easy to be used through the Web. However, signaling methods and protocols are not specified by WebRTC. Therefore there are many solutions proposed in WebRTC signaling.[6] Extensible Messaging and Presence Protocol (XMPP) as a mature protocol, which is open standard for messaging and presence, is one of those proposed signaling solutions for WebRTC. There is quite little research on implementation of WebRTC based on XMPP and on quantified performance analysis. To find out whether XMPP’s capability is suitable as signaling solution, this project is conducted to validate and evaluate the delay performance of WebRTC signaling method based on XMPP.

1.2 Problem Statement

Since WebRTC does not specify signaling implementation, many different solutions have been given by researchers and engineers. JSON or XML via XMLHttpRequest (XHR), HTTP Comet or WebSocket can be good solutions for implementing signaling. Mature protocols such SIP and XMPP are also good solutions. In addition, combination of XMPP, SIP, Jingle or constructing own protocol carrying SDP-like information can also make work for signaling. [1][7][8][9][10]

Among the solutions, XMPP is one of those discussed frequently. XMPP is the Extensible Messaging and Presence Protocol, which has been designed as the open standard for messaging and presence. As XMPP has features of openness, standardization, provenness, decentralized, security, flexibility and diversity, it is becoming more and more popular.[11] XMPP is a set of open technologies for instant messaging, presence, multi-party chat, lightweight middleware, and generalized routing of XML data. Instant message and presence enable information exchange for signaling. Multi-party chat supports signaling for multiple clients. Lightweight middlware makes signaling easy and generalized routing of XML data makes signaling scalable. Thus, XMPP can help to set up a signaling server and can be a good signaling solution for WebRTC.

However, a coin has two sides. XMPP as signaling protocol by WebRTC has many advantages, meanwhile it has disadvantages as well. One of disadvantages is heavy text-based communication. Since XML is text based, normal XMPP has a higher network overhead compared to JSON. This disadvantage is considered as barrier which may have negative effect on the performance when XMPP is used as signaling solution for WebRTC. To improve the solution, several objectives as set as follows.

1.3 Objectives

The main objective is to evaluate if XMPP is a good candidate for WebRTC signaling. In order to achieve this, the following have been done:

1. **Design and develop a direct XMPP-based signaling protocol**. In order to find out that XMPP is an effective solution for WebRTC signaling, it is necessary to design and implement WebRTC with XMPP first. So a model in which client connect to XMPP server directly is proposed and implemented.

2. **Design and develop an indirect XMPP-middleware signaling protocol**. Since XMPP is heavy text-based communication protocol, it is expected that by adding a middleware between client and XMPP server which is located at the same server would reduce data flow between client and server, that is, data flow between client and middleware through the Internet is reducing while data flow between middleware

and XMPP server is increasing. Transmission inside the same machine is considered much faster than over the Internet. Then it may improve delay performance for WebRTC signaling. Therefore, another model with middleware is proposed and implemented.

3. **Propose an evaluation method**. After implementations have been done, analysis of performance will be planned. To achieve this, approach of measurement shall be validated. The first thing is splitting signaling process in implementations into steps. Then timestamps will be set between each two steps and data size will be captured as well. So size of data flow and time cost will be recorded, as well as number of candidates.

4. **Test and compare the two signaling protocols**. Based on the records, performances can be analyzed. Then performances for both models can be compared in quantity and in the end results can be concluded. The results can be a guideline for other researchers in relevant fields.

Chapter 2
Background and Related Work

This chapter summarizes background and related work according to this thesis. A general research on WebRTC is first conducted. Then research on signaling is followed. In order to make a deep understanding of signaling of WebRTC, research on existing signaling solutions for WebRTC is conducted.

2.1 Research on WebRTC

Web real-time communication (WebRTC) enables rich, high-quality RTC applications to be developed for browser, mobile platforms, and IoT devices, and allow them all to communicate via a common set of protocols.

In Figure 2.1, there are three layers, Web Server on the top, application developed with JavaScript/HTML/CSS in the middle and Native OS at the bottom. The application connects to Web Server with HTTP/WebSocket. HTTP is used to send request and receive response. Basically, it is used for fetching application. WebSocket is used for setting up connection to WebServer and enable server to push data to the application. The application helps to call WebRTC API in order to control Web browser and invoke Browser RTC functions, then utilize resources from native OS or communication with remote client.

Figure 2.2 shows the WebRTC trapezoid and real-time communication in the browser. From left bottom, browser on a client sets up communication with Web server using HTTP/WebSockets, and the same for browser on another client at right bottom. Web servers help to set up signaling path. By exchange information through the signaling path, media path can be built up between clients. To achieve signaling, one client can send information to a certain remote client. While the remote client receives the information and send information back to the sender.

WebRTC makes real-time communication easy to be implemented through the Web. However, signaling methods and protocols are not specified by WebRTC.[6] Therefore, research on signaling and WebRTC signaling should be conducted.

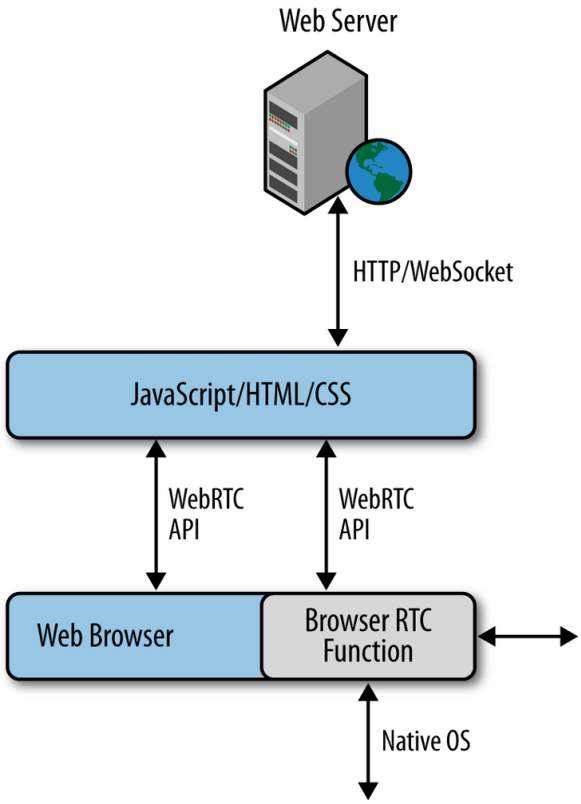

Figure 2.1: WebRTC in Browser[1]

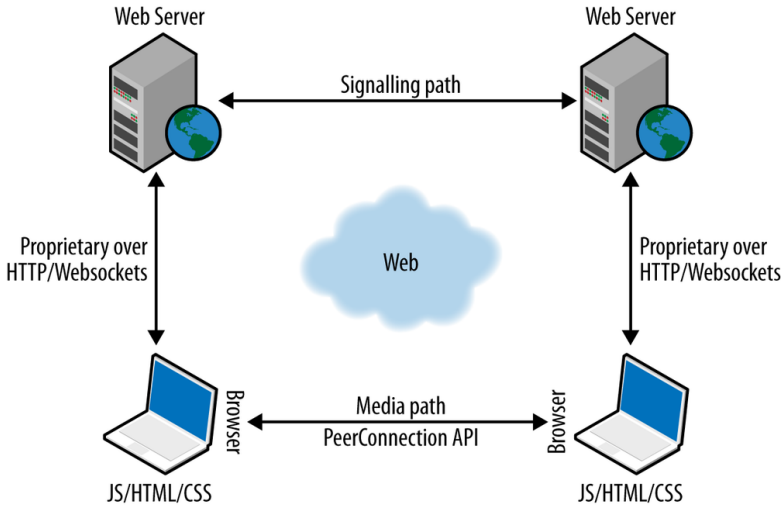

Figure 2.2: WebRTC Trapezoid[1]

2.2 Research on Signaling

To establish a connection between two peers in WebRTC and enable a call, signaling is a necessary part.

The earliest telephone exchanges were “manual” switchboards, in which all calls were set up and taken down by operators. To make a call, the subscriber starts by sending a ringing signal. This alerted an operator, who would connect her telephone to the calling line, and ask for the called number. The operator then would connect her telephone to the called line, and ring the line. After answer by the called party, the operator would establish the connection. Signaling as we know it today started around 1890, with the invention, by Almon B. Strowger (a Kansas City undertaker), of an automatic switchboard that could receive the called number dialed by the calling subscriber, and would then automatically set up the connection. During the past 100 years, signaling applications and technology have evolved in parallel with the developments in telecommunications.[12]

Signaling is a significant part in telecommunication. The two parties cannot communicate with each other without signaling even though they have cable connected with each other. So there are many protocols developed to implement signaling process.

A signaling protocol is a type of protocol used to identify signaling encapsulation.[13] Signaling can be done by many protocols associated with WebRTC such as XMPP, SIP, Jingle and so on.

XMPP as signaling protocol

XMPP is a set of open technologies for instant messaging, presence, multi-party chat, voice and video calls, collaboration, lightweight middleware, content syndication, and generalized routing of XML data.[11] The XMPP specifications were published as RFC 3920 and RFC 3921 in 2004, and the XMPP Standards Foundation continues to publish many XMPP Extension Protocols. In 2011 the core RFCs were revised, resulting in the most up-to-date specifications (RFC 6120, RFC 6121, and RFC 7622).[14][15][16][17][18]

As a mature protocol, it is one of those proposed signaling solutions for WebRTC, since its features such as instant message, presence and multi-party chat support the functionalities of WebRTC signaling and its features such as lightweight middleware, content syndication, and generalized routing of XML data support the non functionalities of WebRTC signaling. The functionalities make WebRTC signaling available and the non-functionalities make WebRTC signaling performance well. The Internet Engineering Task Force (IETF) has formalized the core XML streaming protocols as an approved instant messaging and presence technology.

SIP as signaling protocol

According to IETF, SIP is an application-layer control (signaling) protocol for creating, modifying, and terminating sessions with one or more participants. These sessions include Internet telephone calls, multimedia distribution, and multimedia conferences.[19] SIP is being used to construct peer-to-peer systems, residential telephony services, PBX replacement systems, and large-scale carrier next-generation networks, such as the IMS (IP Multimedia Subsystem) of the 3GPP (Third Generation Partnership Project).[20]

SIP, compared with XMPP, has more functions than instant message. It then of course supports the part as WebRTC signaling.

Jingle as signaling protocol

Besides, Jingle is an XMPP protocol extension for initiating and managing peer-to-peer media sessions between two XMPP entities in a way that is interoperable with existing Internet standards. The protocol provides a pluggable model that enables the core session management semantics (compatible with SIP) to be used for a wide variety of application types (e.g., voice chat, video chat, file transfer) and with a wide variety of transport methods (e.g., TCP, UDP, ICE, application-specific transports).[21]

Jingle, similar to SIP, it supports more functions such as video call and audio call. So it supports WebRTC signaling the same as XMPP and SIP.

XMPP vs SIP and Jingle

Basically, XMPP is famous for its supporting instant message. Although SIP and Jingle also support instant message, these two protocols are more supportive to multimedia communication. They have additional functions more than the WebRTC signaling needs. For researching from essential, XMPP is a better choice. Therefore, XMPP is chosen as target to be researched here as the way to implement WebRTC signaling.

2.3 Existing Signaling Requirements and Solutions for WebRTC

2.3.1 Message Exchange Process of WebRTC Signaling

Message exchange is the most significant part of WebRTC signaling. In book, "Real-Time Communication with WebRTC", the message exchange process of signaling has been explained by examples.[1]

In message exchange process of WebRTC signaling, there are several steps. RTC PeerConnection[6] is the WebRTC component that handles stable and efficient communication of streaming data between peers. First a *RTCPeerConnection* is created on caller side. The MediaStream[6] API represents synchronized streams of media. For example, a stream

taken from camera and microphone input has synchronized video and audio tracks. After the RTCPeerConnection has been created, local *MediaStream* is added to it. The Session Description Protocol (SDP)[6] is a format for describing streaming media initialization parameters. SDP is intended for describing multimedia communication sessions for the purposes of session announcement, session invitation, and parameter negotiation. The RTCPeerConnection adds MediaStream and then create an offer which contains *SDP* information. The RTCPeerConnection set the offer as local description and then send it to callee. As long as the callee receives the offer from the caller, it creates a RTCPeerConnection as well. The same as caller side, callee adds local MediaStream to the RTCPeerConnection, too. Right after, the callee set the offer as remote description to the RTCPeerConnection. Then the RTCPeerConnection on callee side creates an answer. Answer is the same as offer. It is just with another name which is distinct from offer. Afterwards, the answer is set as local description to the RTCPeerConnection on callee side, then sent to caller. As long as the caller receives the answer, it set the answer as remote description to the RTCPeerConnection. After creating the PeerConnection and passing in the available STUN and TURN servers, an event will be fired once the ICE framework has found some "candidates" that will allow you to connect with a peer. Knowing the message exchange process of WebRTC signaling helps to design and implement the solutions.

2.3.2 Potential Signaling Solutions for WebRTC

Copy and Paste Manually

Copying necessary SDP information and pasting to the target can be a solution. This sounds like a stupid way but it works. As long as the necessary SDP information for WebRTC are exchanged for both peers, both peers can then establish a connection. However, the problem is that it cannot work as an application since it is not programmatic and it is not feasible when scenario becomes complex.

Comet

Comet is a web application model in which a long-held HTTP request allows a web server to push data to a browser, without the browser explicitly requesting it.[22][23] So by applying Comet, it enables that server can push messages to client. Thus, message can be sent from server forward to a target as long as server receive message from a source and then execute message exchange process. But it does not define own proprietary signaling messages. It therefore needs support from signaling protocols such as XMPP, SIP, Jingle and so on.

WebSocket

WebSocket is an advanced technology that makes it possible to open an interactive communication session between the user's browser and a server. With this API, you can send messages to a server and receive event-driven responses without having to poll the

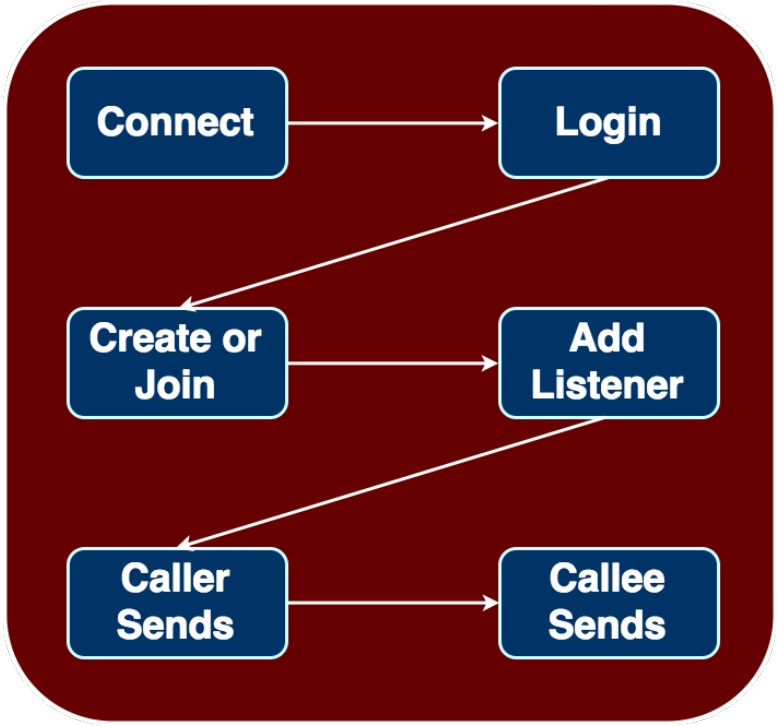

Figure 2.3: Process of Applying XMPP

server for a reply.[24] The same as Comet, it needs support from signaling protocols but it consumes less resource than Comet.

XMPP or SIP over WebSocket

WebSocket provides pretty good solution for transfer messages. Combined with XMPP or SIP will enable the whole process of WebRTC signaling. However, WebRTC supports media communication for peers, only IM is enough for the signaling. SIP is more than a IM protocol and the extra functions are not necessary for WebRTC signaling. Therefore, XMPP is preferred here in this project. SIP may also be a good solution, but since time is limited in this project, SIP as a solution will be researched in the future work.

2.3.3 Integration of XMPP and WebRTC

To integrate XMPP into WebRTC signaling, study on message exchange process of XMPP is necessary.See Figure 2.3, first thing first, client need to connect to XMPP server. Therefore, creating of connection to XMPP from client comes before everything. Then from client, user need to login to XMPP server. If the user has not registered in XMPP server, then

registration should be done before login. To support Multiple User Chat (MUC), a chat room also should be created. Afterwards, callers and callees should then join the same chat room. To start a call, one user as a caller sends messages such as offer from caller, answer from callee or candidate information to other callees through the chat room. As long as XMPP receive messages from user, it then sends to forward to certain targets, i.e. one or more callees. By implementing message exchange as above, WebRTC signaling can manage to integrate XMPP.

Chapter 3
Methodology

To achieve the objectives mentioned in Chapter 1, the research is planned to conduct a literature review of relevant topics, design models based on XMPP and then implement them. Afterwards, analysis will be conducted based on the models.

3.1 Literature Review

Basic theories are important for research. To conduct such research on WebRTC signaling, a clear understanding of WebRTC is necessary. The research on WebRTC helps to know what WebRTC is, how WebRTC looks like and how it works. Since signaling methods and protocols are not specified by WebRTC, research on signaling is also needed. The research on signaling helps to know what signaling is used for, how it works and which protocols it has. Also, it lists several signaling protocols associated with WebRTC signaling. In addition, a set of existing signaling solutions for WebRTC are studied in order to make a better solution based on others' research.

3.2 Model Design

In order to avoid the problems mentioned in Chapter 1, utilizing XMPP server as a signaling server for WebRTC may be a good solution. Designing an architecture by setting up a middleware server between XMPP server and the WebRTC clients can help. It could reduce data flow between client and server. In such way the time cost that one client sends message to server and server sends forward to another client may reduce. To find out if the solution works, two models mentioned in objectives in Chapter 1 are designed, implemented, analyzed and compared.

The first one is a simple model that client connect to XMPP server directly. In Figure 3.1, it shows the one without middleware server will enable the WebRTC client to communicate directly with the XMPP server. All the contents are being transfered through the channel between them.

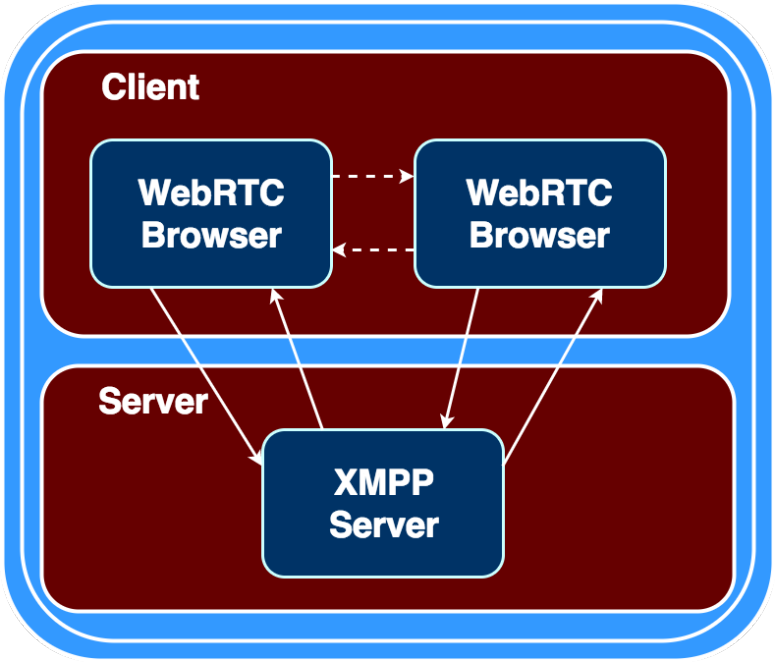

Figure 3.1: WebRTC Model Design with Direct Connection

As mentioned in Chapter 1, since XMPP is heavy text-based communication protocol, it is expected that by adding a middleware between client and XMPP server which is located at the same server would reduce data flow between client and server, that is, data flow between client and middleware through the Internet is reducing while data flow between middleware and XMPP server is increasing. Transmission inside the same machine is considered much faster than over the Internet.

In Figure 3.2, it shows the model with middleware server. The middelware server is set together with the XMPP server on the same machine. It will work in such a way that the WebRTC client sends request to the middelware server, the middelware server communicate with the XMPP server with the majority part of the contents, and then the middelware server pushes a response to the WebRTC client.

Both models can apply the process of WebRTC signaling according to the previous research in Chapter Chapter 2.

3.3 Technology in Use

As signaling is complex, many technologies are used in the project. WebRTC components are accessed with JavaScript APIs. Currently in development are the Network Stream

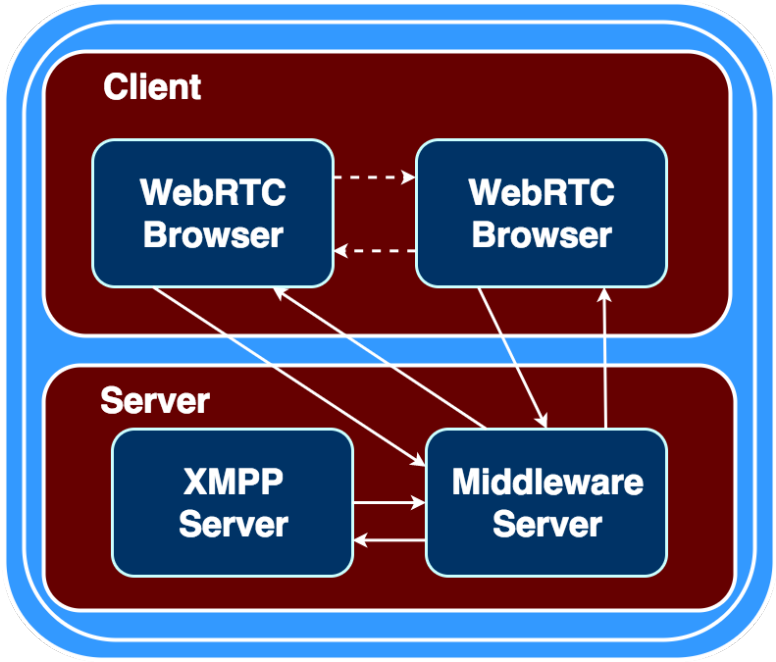

Figure 3.2: WebRTC Model Design with Indirect Connection

API, which represents an audio or video data stream, and the PeerConnection API. The WebSocket Protocol enables two-way communication between a client running untrusted code in a controlled environment to a remote host that has opted-in to communications from that code. STOMP provides an interoperable wire format so that STOMP clients can communicate with any STOMP message broker to provide easy and widespread messaging interoperability among many languages, platforms and brokers. Ejabberd is fully open source, secure, flexible, interoperable and professionally maintained. Smack is an Open Source XMPP (Jabber) client library for instant messaging and presence. Strophe.js is an XMPP library for JavaScript. Its primary purpose is to enable web-based, real-time XMPP applications that run in any browser.

- **WebRTC API**[25]. To access native resources such as fetching the MediaStream and communication from peer to peer, *WebRTC* APIs are necessary for the requirement.

- **Ejabberd**[26]. XMPP is just a protocol. So a server which implement XMPP is needed for the solution. *Ejabberd* is such an XMPP server.

- **WebSocket**[24]. Client sends data to server meanwhile server need to push data to client. So *WebSocket* plays the role.

- **STOMP**[27]. Since JSON is text-based format, STOMP is then used for messaging. *STOMP* is the Simple (or Streaming) Text Orientated Messaging Protocol.

- **Smack**[28]. Middleware is planned to be implemented by Java. In order to enable communication between middelware and Ejabberd, such a Java library which supports to send message to and receive message from server is neccessary. *Smack* is the proper one.

- **Strophe.js**[29]. To enable communication directly from client to XMPP server, *Strophe.js* as a JavaScript library is needed to complete the task.

3.4 Model Implementation

Practice makes theory come true. After designing, the next step is implementation.

Implementing the model enabling communication directly from client to XMPP server will be conducted in the following way. *Ejabberd* will be installed on the server side. This is how XMPP is implemented. Through this server, client can send message to another client. *Javascript* will be used as programming language on client side. JavaScript helps to call WebRTC APIs, to communicate with server and to implement the process. *Strophe.js* is an XMPP library for JavaScript. Its primary purpose is to enable web-based, real-time XMPP applications that run in any browser. This is used for communicating with Ejabberd from the client side. *WebRTC APIs* will then be called to use native resources. First local stream will be fetched and then signaling process will be implemented. After the signaling process has done, communication between clients via WebRTC should work.

The same as the former model, Ejabberd, Javascript, WebRTC API will be used to implement the model with middleware server. Besides, there are serveral other steps needed. *Java* as programming language will be used to develop the middleware. *Spring Framework* will be used improving the development and supporting WebSocket on the middleware side. Then *SockJS* will enable WebSocket on client side to enable communication between client and middleware. Afterwards, *Smack* will be used in middleware to enable communication between middleware and Ejabberd.

3.5 Validation Method Implementation

Experiments will be conducted right after implementation. To conduct the experiments, validation method shall be designed and implemented. Time cost and data flow size are two of the most significant parameters which are considered affecting the performance of WebRTC signaling and both of them need to be recorded.

Table 3.1: Data to be Collected

Index	Name
1	Total Time Cost
2	Signaling Time Cost
3	Total Data Size
4	Answer Size
5	Candidate Size
6	Number of Generated Candidates
7	Number of Received Candidates

In order to obtain time cost of WebRTC signaling, timestamps will be set between each step of WebRTC signaling process. By calculating the time difference between each two timestamps, the time cost can be obtained. The details will be explained in Section 4.4.2

To obtain data flow size, there are two ways. One is to capture the data flow when sending data from sender and the other one is to capture the data flow when receiving data by receiver. As long as the data flow is captured, the data flow size can then be obtained. The details will be explained in Section 4.4.1

After implementing the validation method, data collection can then be executed.

3.6 Data Collection

To analyze performance of the signaling process of WebRTC, the following data are about to be collected, see Table 3.1.

1. Total Time Cost. This represent the total time spent in the entire signaling process.
2. Signaling Time Cost. This represent the total time spent in initiating call, sending offer, sending answer but without ICE Candidates exchanging.
3. Total Data Size. This represent all the data flow that has been exchange during the entire signaling process.
4. Answer Size. This represent the data size of an answer SDP.
5. Candidate Size. This represent the data size of a ICE Candidate.
6. Number of Generated Candidates. This represent the total number of ICE Candidates that are generated and added to local peer.
7. Number of Received Candidates. This represent the total number of ICE Candidates that are received from a remote peer after ICE Candidate exchange.

3.7 Results and Analysis

Having collected the data needed, the implementations should work well. Results can be shown by making screen shots and tables. To analyze the results, more and deeper research will be conducted according to the results.

Chapter 4
Model Design

This chapter describes what are the two models about, how they are designed and why they are designed in the certain way. This chapter also describes how caller and callee interact with each other and manage to communicate with each other via video call, audio call or text message.

4.1 Design of Architecture

4.1.1 Architecture Design of Direct Connection to XMPP Server

Basically, it is a Client-Server architecture, see Figure 4.1.

In order to describe clearly, it designs the model with two-party calls. So on client side, there are two participants. To enable a call, either a video call or an audio call, it needs two parts. One is caller and the other is callee. Both caller and callee are using Web, so two clients in the architecture are actually two Web browsers. Additionally, not all browser support WebRTC, the two browsers should be WebRTC supported browsers.

On server side, it provides such functions that receive data from one client and send data forward to another client, since two WebRTC browsers are private and they do not know much about each other. To enable such functions, XMPP applies here.

To start a call, WebRTC browsers need to call WebRTC APIs to complete the tasks. It first need to check whether the browser supports WebRTC or not. If the browser supports WebRTC, it then loads resource, either video resource, audio resource or other kinds of resources, such as video stream, audio stream or ICE candidate.

And then clients exchange the necessary information through the server based on XMPP. In order to connect to XMPP server, stroph.js is being used here. Stronph.js enables a connection between the client and the XMPP server via WebSocket which means it enables sending data from client and pushing data from server.

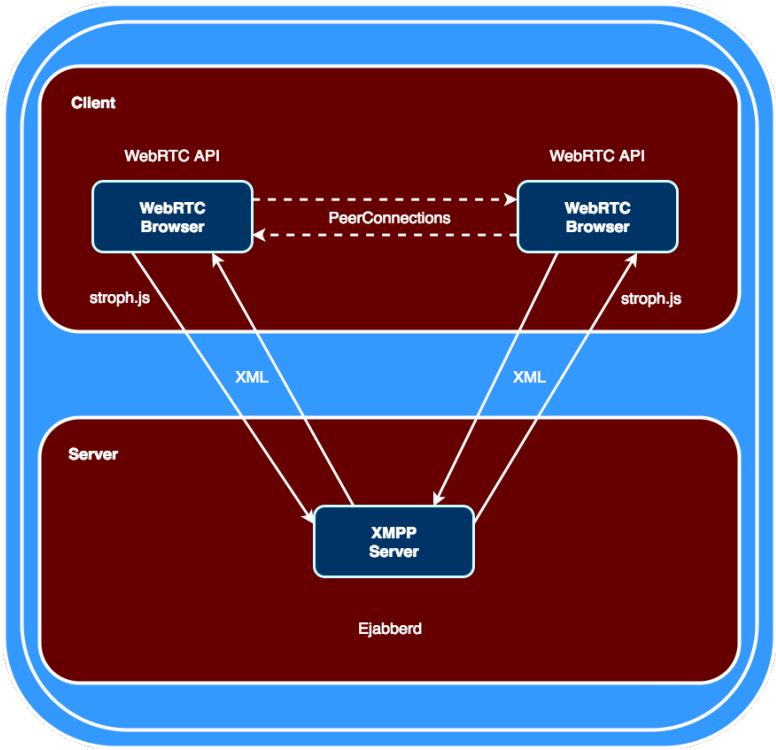

Figure 4.1: Architecture Design of Direct Connection to XMPP

Ejabberd is used as the XMPP server which provides core functions of XMPP. Since XMPP server is XML based, the data format between Ejabberd and clients is XML, which is generated by stroph.js.

After exchanging necessary information between two browsers, the two browsers can then communicate with each other through the peer connections.

According to the architecture described above, a client can send messages to another client or another group of clients whenever it wants and a server can push messages to any clients as long as it receives such messages.

4.1.2 Architecture Design of Indirect Connection to XMPP Server

The same as previous model, it is a Client-Server architecture, see Figure 4.2. Since XMPP is heavy text-based communication protocol, it is expected that by adding a middleware between client and XMPP server which is located at the same server would reduce data flow between client and server, that is, data flow between client and middleware through the Internet is reducing while data flow between middleware and XMPP server is increasing.

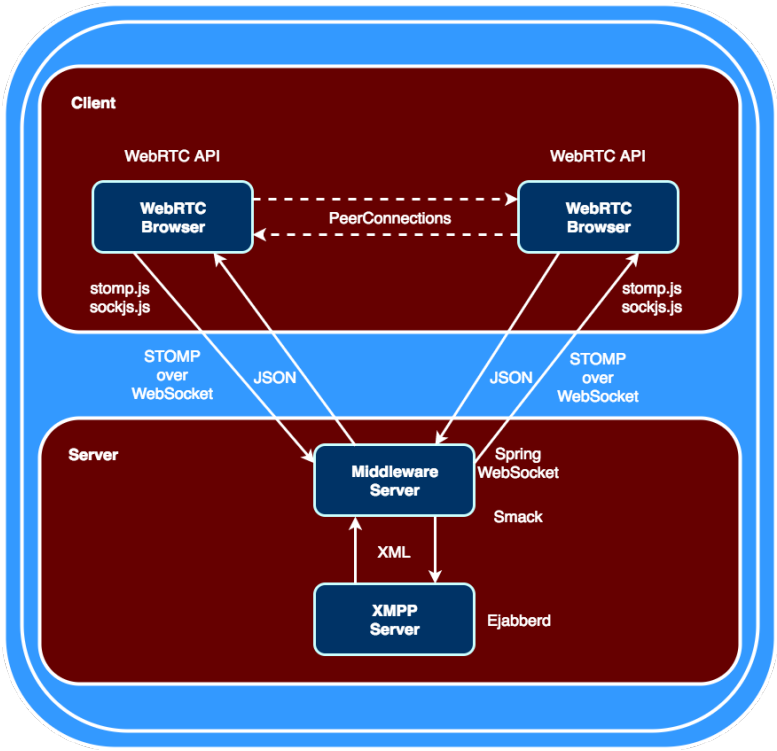

Figure 4.2: Architecture Design of Indirect Connection to XMPP

Transmission inside the same machine is considered much faster than over the Internet. So the difference of architecture between direct and indirect connection to XMPP server is that there is an extra part which is called middleware server in the model. Clients do not connect directly to XMPP server. Instead, clients connect to the middleware server and the middleware server connects to XMPP server. Middleware server and XMPP server are running on the same server.

To make it work, Sock.js run on the client side and Spring WebSocket run on the server side enable the connection between client and server through WebSocket. Stomp.js enables text based messages over WebSocket. Therefore the clients can communicate with the middleware server with the data in JSON format.

The middleware server then communicates with XMPP server with Smack. The middleware server is a Java based server. Smack, the Java based library helps to generate XMPP based messages and also helps to send and push XML data between the middleware server and the XMPP server, Ejabberd.

The same as previous model, after exchanging necessary information between two

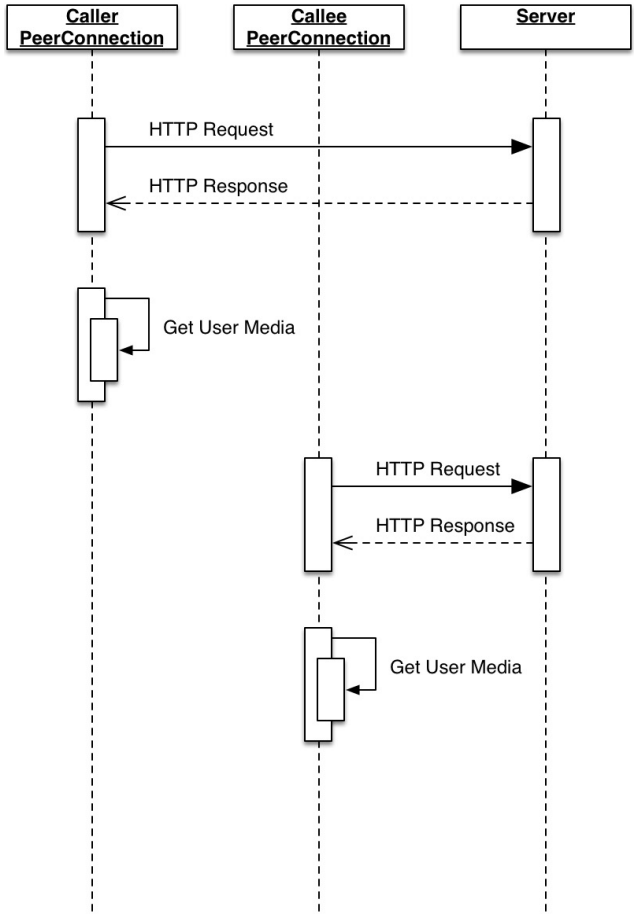

Figure 4.3: Signaling Flow of Models: Initiation of WebRTC

browsers, the two browsers can then communicate with each other through the peer connections.

4.2 Design of Signaling Flow

Figure 4.3 shows how WebRTC initiates when application starts up. Both caller and callee send HTTP Request to application server. Application is developed as a Web application. It therefore accesses to the Web server, fetches Web application. The core part is JavaScript code. After loading code locally, WebRTC API are called by the code in browser to Get User Media. Then browser can represent local user media such as video stream and audio stream.

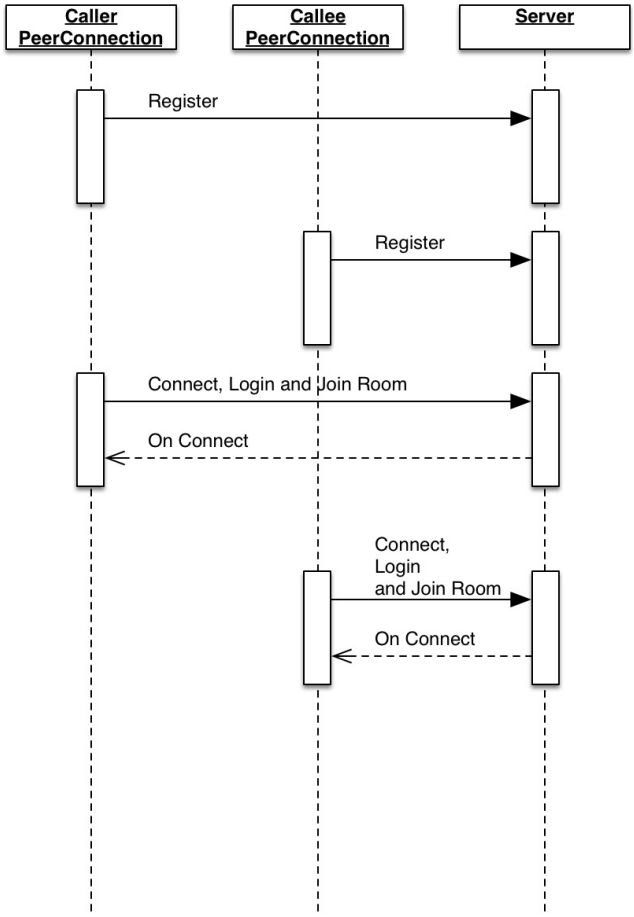

Figure 4.4: Signaling Flow of Models: Connection of WebRTC

Figure 4.4 explains how caller and callee get connected. First of all, caller and callee should register by generating a universally unique identifier (UUID). By using the UUID, caller and callee register on the XMPP server. After having an account on the XMPP server, caller and callee can then connect and login to the XMPP server. In order to support multi user chat (MUC), chat room is used here. Caller and callee create or join the same room to be able to have call together. When all participants have joined the room, the connection establishes successfully. The server then sends a feedback to the clients so that the clients can continue next steps.

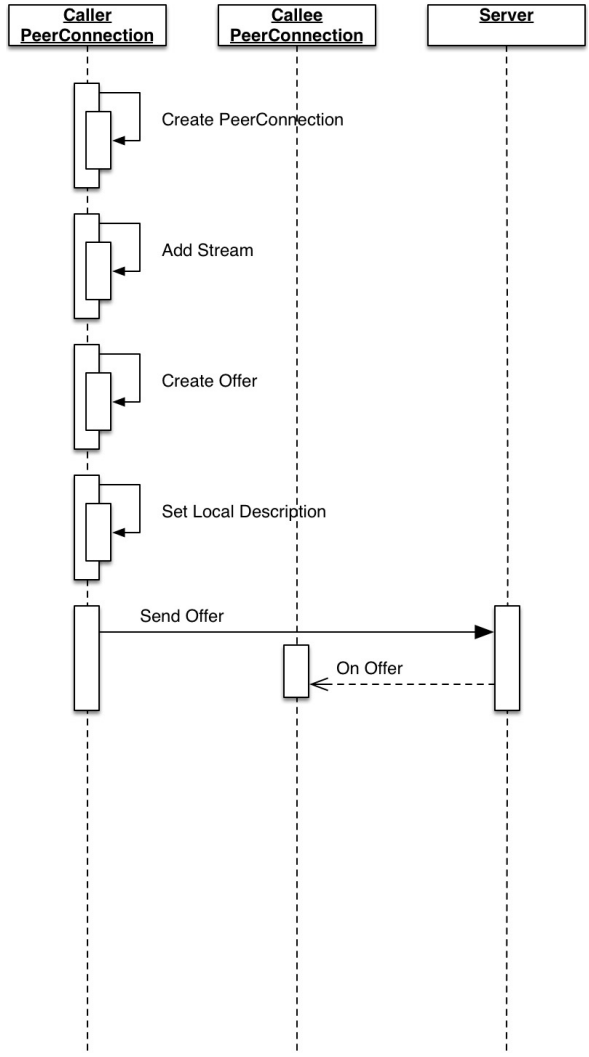

Figure 4.5: Signaling Flow of Models: Offer of WebRTC

From Figure 4.5 it starts to represent the core steps of WebRTC signaling. *PeerConnection* is an object applied by *WebRTC API* which implements functions to establish connection between peers. First, caller creates a *PeerConnection.* Then the local media stream is added to the *PeerConnection.* After that, the WebRTC signaling process starts with creating an *Offer.* When *Offer* is created successfully, the *PeerConnection* sets local description immediately for the *Offer* and then the *Offer* is sent to callee. Callee will get the *Offer* and process next steps.

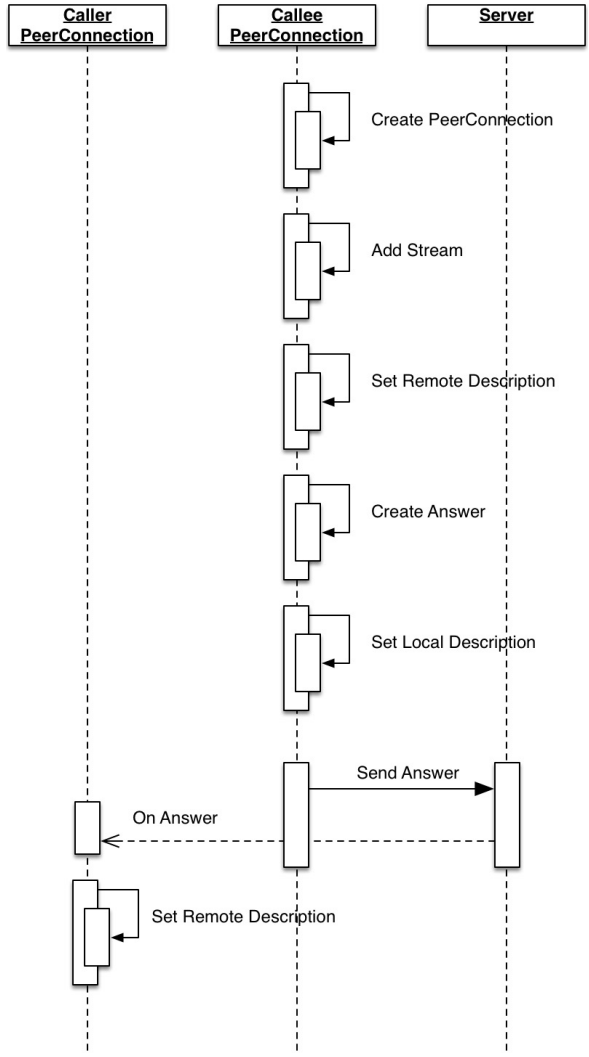

Figure 4.6: Signaling Flow of Models: Answer of WebRTC

Figure 4.6 explains the steps how callee answers a call while caller is offering the call. The same as caller, callee creates a *PeerConnection* first. Then the local media stream of callee is added to the *PeerConnection*. As long as caller sends *Offer* and callee receives the *Offer*, the *PeerConnection* of callee sets remote description from caller carried in the *Offer*. The *PeerConnection* of callee then creates an *Answer* according to the *Offer*. When the *Answer* is created successfully, the *PeerConnection* of callee sets local description immediately for the *Answer* and then the *Answer* is sent to caller. Caller will receive the *Answer* and the *PeerConnection* of caller will set remote description carried in the *Answer* from callee.

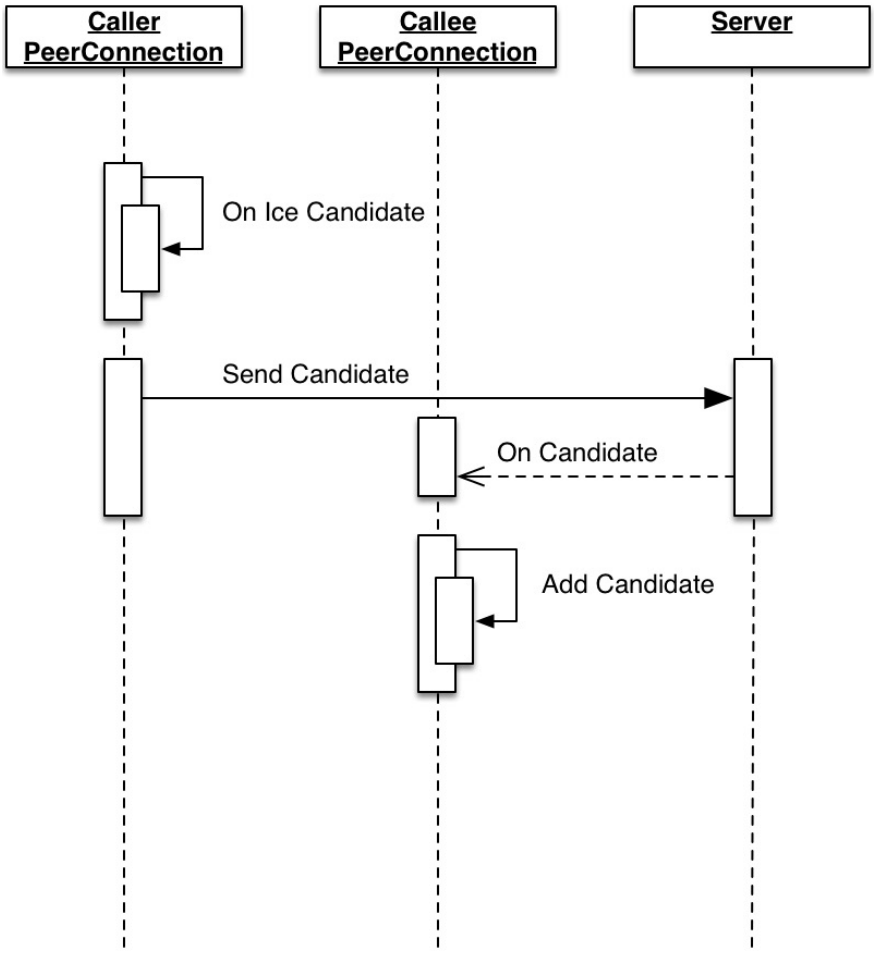

Figure 4.7: Signaling Flow of Models: Candidate of WebRTC (a)

Figure 4.7 shows how candidates are exchanged after *Offer* and *Answer* have been exchanged. The *PeerConnections* has a listener for candidates. As long as there is a new candidate generated on caller, the candidate will be sent to callee. When callee receives the candidate, it will be add to its local *PeerConnection*.

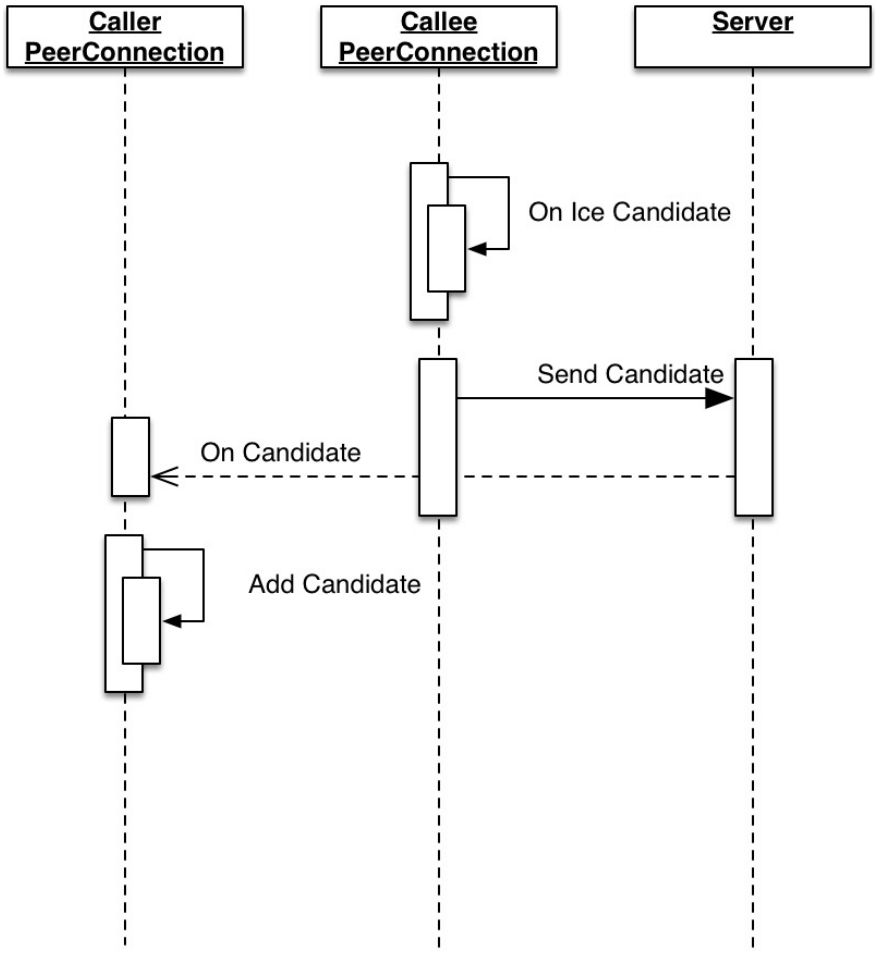

Figure 4.8: Signaling Flow of Models: Candidate of WebRTC (b)

Similar to Figure 4.7, Figure 4.8 shows how candidates are exchanged after *Offer* and *Answer* have been exchanged on callee. The *PeerConnections* has a listener for candidates. As long as there is a new candidate generated on callee, the candidate will be sent to caller. When caller receives the candidate, it will be add to its local *PeerConnection*.

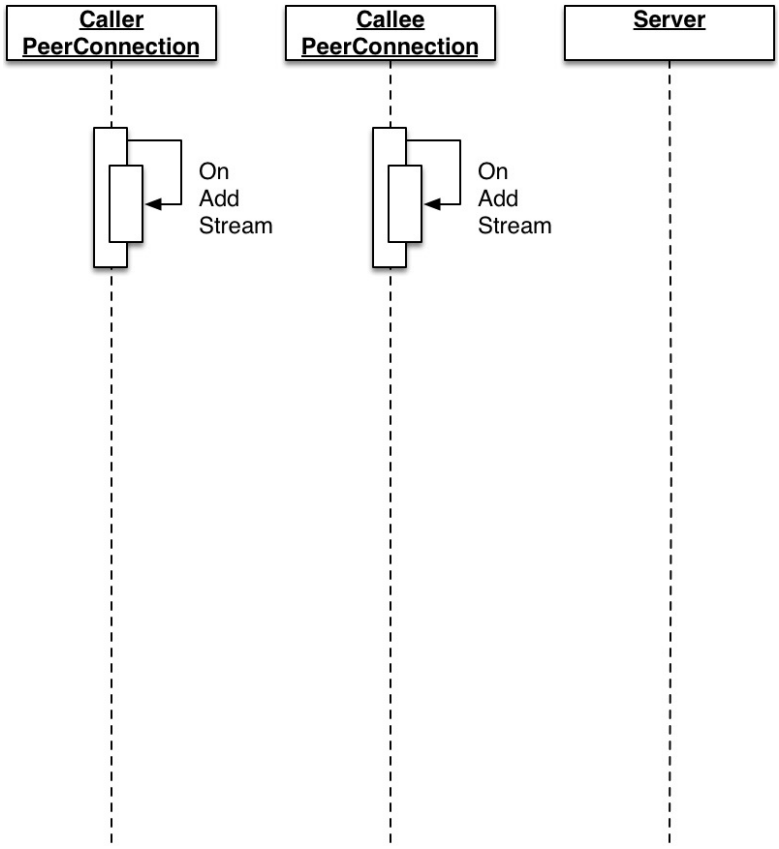

Figure 4.9: Signaling Flow of Models: Stream of WebRTC

In Figure 4.9, streams are added to *PeerConnections* on both caller and callee after necessary information has been exchanged for both of them. In the end, both *PeerConnections* can establish a connection. Therefore, a call can be set up between caller and callee.

4.3 Design of Data Format

4.3.1 JSON Format

JSON Object is defined as the basic object in the signaling process including connect, start, offer, answer and candidate. In Implementation 4.1, it shows an example of data in JSON format.

For each of candidate, there is a field called 'type'. 'type' indicates their types, among connect, start, offer, answer and candidate.

Another filed is called 'from'. 'from' represents where the message comes from, therefore it knows where the response should be send back to. The type of 'from' is UUID.

The third field is a dynamic field, which stores the data according to the object type. See Implementation 4.1, for example, if the type is candidate, then the third filed is called candidate and candidate is stored in this field.

Implementation 4.1 JSON Data Format for Candidate

```
{
   type: "candidate",
   from: "f64363ab40d4403da95304c8e2aeb5a1",
   candidate: {
            "candidate":"candidate:2241210590 1 udp 2122260223
               192.168.2.101 57343 typ host generation 0
               ufrag 8jg3 network-id 1 network-cost 10",
            "sdpMid":"audio",
            "sdpMLineIndex":0
   }
}
```

4.3.2 XML Format

XML format is the basic format for XMPP. Implementation 4.2 shows an example of data in XML format in XMPP.

The root tag is <message>. Tag <message> has the name space "jabber:client". Tag <message> also has attributes *from*, *to*, *type* and *id*. *id* identifies each message. *from* and *to* represent where the message comes from and where the message goes to. While type indicates the type of message. Here in the example, the id is ab627d95-9931-402e-97c0-0f34ff95ede1. The message comes from *test@conference.f2f.chat/ddb88c89e9c14e208e5d41d894811399* and will be sent to textitddb88c89e9c14e208e5d41d894811399@f2f.chat/97889940814956268036619 91. And the type of message is *groupchat*.

Implementation 4.2 XML Data Format for Candidate

```
<message xmlns="jabber:client" from="test@conference.f2f.chat/
    ddb88c89e9c14e208e5d41d894811399" to="
    ddb88c89e9c14e208e5d41d894811399@f2f.chat
    /97889940814956268036619 91" type="groupchat" id="ab627d95
    -9931-402e-97c0-0f34ff95ede1">
   <body>
      {
         type: "candidate",
         from: "f64363ab40d4403da95304c8e2aeb5a1",
         candidate: {
               "candidate":"candidate:2241210590 1 udp
                  2122260223
                  192.168.2.101 57343 typ host generation 0
                  ufrag 8jg3 network-id 1 network-cost 10",
               "sdpMid":"audio",
               "sdpMLineIndex":0
         }
      }
   </body>
   <x xmlns="jabber:x:event"><composing/></x>
</message>
```

In tag <message>, there is a tag <body> which contains the core information. The content of information is the same as in JSON format.

4.4 Design of Measurement

4.4.1 Measurement of Data Size

Since two models are designed in different ways, the data transfer between client and server has different data format and different data size. Data size will affect the performance of signaling, particularly when network condition is not good, because when data size is larger it takes longer time for transferring. Therefore, measuring data size is necessary. Sizeof.js as a third-party JavaScript library is used for calculate the data size and display it. There are two options to capture data flow and calculate data size. One is to capture the data when it is ready to send and the other one is to capture the data when the data is received. Since the function sending data are implemented in other third-party JavaScript libraries while the function receiving data are implemented in own implementation, the latter option is a better solution in order to avoid modification to source code of other libraries.

4.4.2 Measurement of Time Cost

Time cost apparently shows how fast the signaling process is. Measuring time cost for the signaling process is a good approach to analyze the performance of signaling process. To measure the time cost, recording timestamps in each steps and calculate time difference between each steps will work. Therefore, timestamps are set whenever the following events are triggered: onInit, onOffer, onAnswer, onCandidate, onAddStream, see Figure 4.3, 4.4, 4.5, 4.6, 4.7, 4.8, 4.9. For one call, onInit, onOffer and onAnswer happened once and onCandidate may happens several times. When onAddStream is called to add remote stream to a local peer connection, the signaling process is done and the call would be successful.

4.4.3 Measurement of Number of Candidate

Each time there is a candidate added to the peer connection, the function onCandidate is triggered. Since in the measurement of time cost, timestamps are added in the function onCandidate, the remark of adding candidates and the timestamps are displayed. By counting the times onCandidate is triggered, the number of candidate can be measured. Since candidate contains significant information about address of peer, and it is generated dynamically, the number of generated would affect the performance of signaling. Therefore it is considered to be one of the parameters that would affect the performance and would be measured.

Chapter 5
Model Implementation

This chapter describes how the models are implemented. And it also explains challenges during the implementation.

5.1 Implementation of XMPP Based Server

Ejabberd has been in development since 2002 and is used all over the world to power the largest XMPP deployments. This project is so versatile that you can deploy it and customize it for very large scale, no matter what is your use case.[26] Ejabberd brings configurability, scalability and fault-tolerance to the core feature of XMPP – routing messages. Its architecture is based on a set of pluggable modules that enable different features. The ones used here are listed below:

- One-to-one messaging
- Store-and-forward (offline messages)
- Contact list (roster) and presence
- Groupchat: MUC (Multi-User Chat)
- Messaging archiving with Message Archive Management (MAM)

5.1.1 Installation of Ejabberd

Command 5.1 Installation of Ejabberd on Debian

```
$ sudo ./ejabberd-version-linux-x86_64-installer.run
```

To install Ejabberd on Debian, the installer for Linux version can be found and downloaded from Ejabberd's official site. Installation can be done by running the following command in terminal.

5.1.2 Configuration of Ejabberd

Command 5.2 Configuration of Ejabberd on Debian

```
...
hosts:
  - "localhost"
  - "f2f.chat"
...
listen:
  -
    port: 5222
    module: ejabberd_c2s
    certfile: "/home/chun/Development/ejabberd/conf/server.pem"
    starttls: true
  -
    port: 5280
    module: ejabberd_http
    request_handlers:
      "/websocket": ejabberd_http_ws
...
acl:
  admin:
    user:
      - "admin@f2f.chat"
...
registration_timeout: infinity
...
```

By default, path of configuration file is '/etc/ejabberd/ejabberd.yml'. *hosts* defines domains served by ejabberd. Here the hosts are set as *localhost* and *f2f.chat*. *listen* configures ports listened by ejabberd. *Port 5222* enables module ejabberd_c2s which supports TLS connection while *Port 5280* enables ejabberd_http which supports HTTP connection. Configuring *request_handlers* makes it support WebSocket through the connection. *acl* stands for access control lists. Changing admin:user: to *admin@f2f.chat* enables control from remote code on f2f.chat. Setting *registration_timeout* to *infinity* enables that infinity of users can register at the same time.

5.1.3 Startup of Ejabberd

Command 5.3 Startup of Ejabberd on Debian

```
$ sudo ./ejabberdctl
```

By running ./ejabberdctl on ejabberd server, ejabberd can be started.

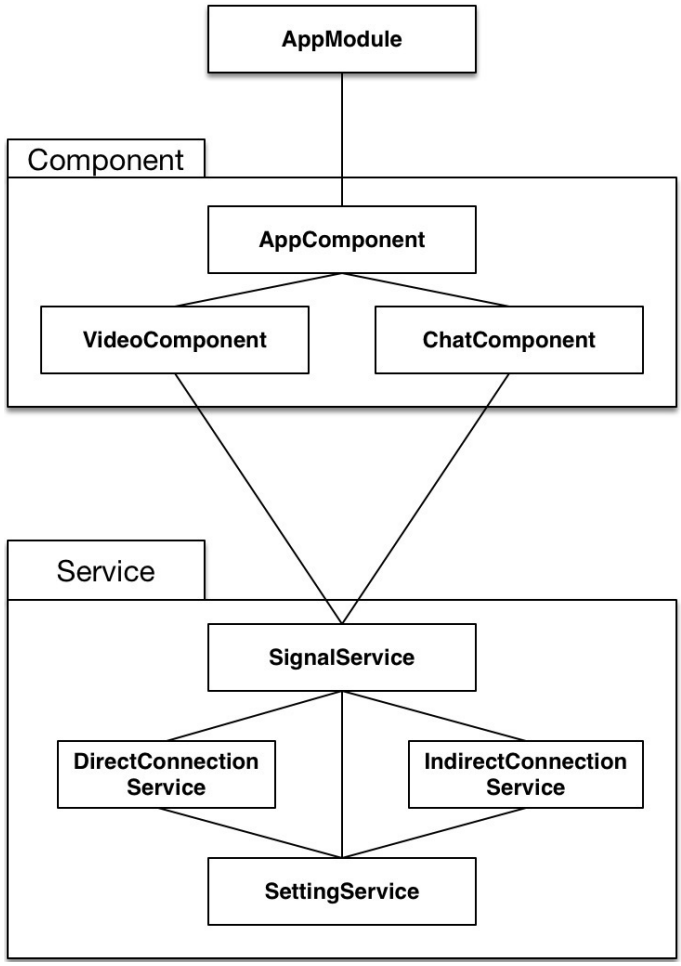

Figure 5.1: Front End Framework: Angular

There are also other relevant commands used in the implementation, but they are not important in this thesis, so they are not discussed here.

5.2 Implementation of WebRTC based Web Application

5.2.1 Framework

Frontend Framework: Angular

AngularJS is a toolset for building the framework for Web application. It makes environment of development expressive, readable, and quick to develop.[30] Angular is therefore used

as front end framework here. Since the application is not too complex, two main layers are applied here including component layer and service layer, see Figure 5.1.

By applying AppModule in index page, everything in front end will be loaded. AppModule then imports and applies AppComponent.

In component layer, each component defines different part in a Web page. AppComponent is the base component of the front end component. It defines basic layout of the application. In AppComponent, it imports and applies VideoComponent and ChatComponent. VideoComponent defines layout of video part when a call is setting up. The video part displays a gray background if only audio is available. ChatComponent defines layout of chat part. It displays how dialog looks like in browser. Since in later experiment data need to be displayed, it will be displayed together with dialog in ChatComponent.

Service layer provides concrete services to components. SignalService are used in both VideoComponent and ChatCompoennt. It implements how components execute signaling process. Meanwhile, it imports and applies either DirectConnectionService or IndirectConnectionServcie. DirectConnectionService implements signaling method that connect to Ejabberd server directly without a middleware while IndirectConnectionService connects to a middleware deployed at the same server as Ejabberd server and then connect to Ejabberd through the middleware. SettingService provide basic settings supporting the application.

Back End Framework: Spring

Spring helps to build simple, portable, fast and flexible JVM-based systems and applications.[31] So Spring is used as back end framework here. Three main layers are applied here including controller layer, service layer and WebSocket layer, see Figure 5.2.

In controller layer, CallController class is the main class used as a dispatcher. When a request coming from client, CallController dispatches the request to corresponding services including index page, call page, register service and WebSocket service.

In service layer, CallService class implements all the services needed for the application. Service layer provides open connection service, close connection service, initiate service, register service, login service, create or join room service and send message service.

In WebSocket layer, it provides WebSocketConfig class which set necessary configuration for WebSocket. It also provides CallWebSocketHandler class which enable abilities to handle connection between client and server via WebSocket. In addition, it provides CallMessageListener class. This class is used for handle the incoming and outgoing messages via WebSocket.

There are also two other layers called filter layer and utility layer. Filter layer provides

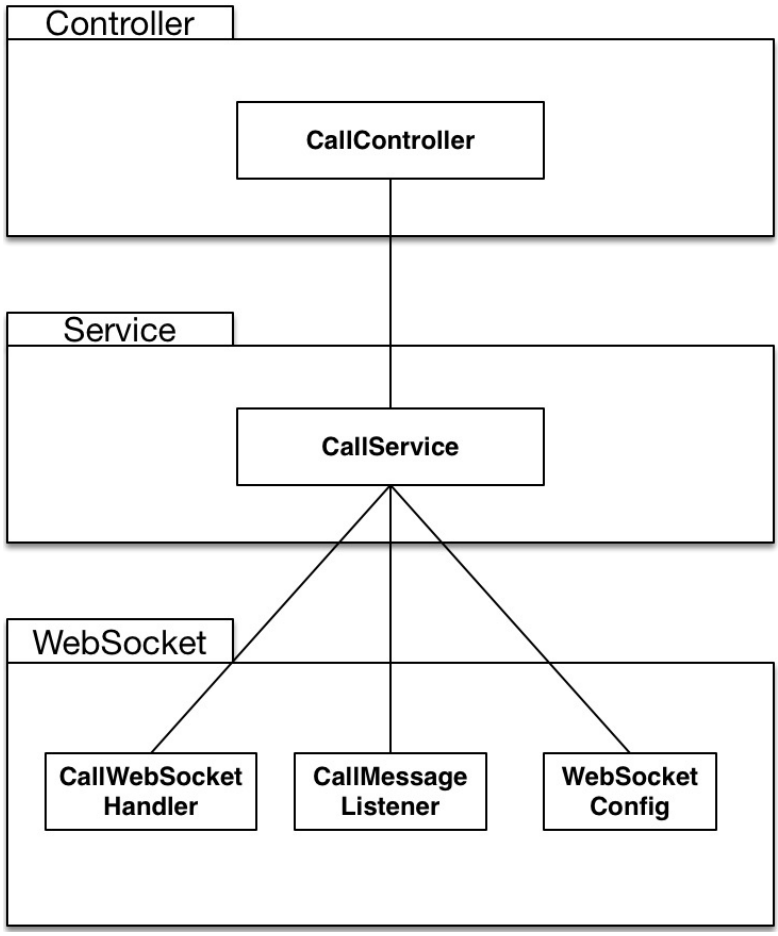

Figure 5.2: Back End Framework: Spring

class to handle request after request send from client but before request handled by server. Here CORSFilter enables Cross-Origin Resource Sharing (CORS) for Java web application. And in utility layer it implements classes as utilities such as static parameters.

5.2.2 Core Functions

The core processes have been explained in Chapter 4. Here is the implementation of them.

onInit

Function *onInit* is executed when application is loaded. It first checks whether browser support WebRTC or not by calling functions in WebRTC API. *navigator.getUserMedia* checks in general while *navigator.webkitGetUserMedia* and *navigator.mozGetUserMedia* check for Chrome and Firefox. If browser supports WebRTC, it then get resources such as video stream and audio stream. As long as resources are loaded successfully, it will register an UUID and connect to server, either Ejabberd server or middleware server. It depends on which signaling method used to connect to either Ejabberd server or middleware server. More details will be explained in section 5.3 and 5.4.

onConnect

Function *onConnect* is executed to prepare for a call when successfully connecting to the server.

onStart

Function *onStart* obviously means starting of a process. This process stands for signaling process. It first constructs a *RTCPeerConnection*. Then there are two interfaces need to be implemented for further use. One is *onicecandidate* and the other one is *onaddstream*. In later process, these two interfaces will be called when new candidates and new streams are added to this peer connection. Then by calling *addStream*, the local stream can be added to the peer connection. The next step is then generating offer. After generating offer successfully, the local description is set by constructing *RTCSessionDescription* according to the offer and the offer is then sent to callee.

onOffer

Function *onOffer* handles the situation when it receives an offer from the server. Thus it plays the role as callee. As a callee, it constructs a *RTCPeerConnection* for the remote role, caller. The same as above, two interfaces *onicecandidate* and *onaddstream* are implemented. And then by calling *addStream*, the local stream can be added to the peer connection. Since the offer carries a *RTCSessionDescription*, the peer connection set it as a remote description. The next step is then generating answer. After generating answer successfully, the local description is set by constructing *RTCSessionDescription* according to the answer and the answer is then sent to caller.

onAnswer

Function *onAnswer* is called by caller when callee received offer and sent answer back. As long as the answer is received by caller, the peer connection of caller set remote description according to the *RTCSessionDescription* carried in the answer.

onCandidate

Having exchanged offer and answer and set local description and remote description for both peer connections, the event *onicecandidate* to both peer connections shall be triggered. They will then send candidates to each other. By receiving candidates, function *onCandidate* are called. It adds those remote candidates to the local peer connection.

onChat

Function *onChat* is executed when receiving a chatting message. It displays the received chatting message to the *ChatComponent* on the Web page.

sendChat

Function *sendChat* is used to send data to a specific user. When answer and candidates are sent to target, *sendChat* is called.

sendGroupChat

Function *sendGroupChat* is used to send data to users in a specific room. It is used when a new joined user sends offer to everybody in the room.

swicher

Function *swicher* in Algorithm 5.4 works as a dispatcher. Every time there is a data coming in and received, it checks the data type and then dispatches it to the respective functions. The data types include *start*, *connect*, *offer*, *answer*, *candidate* and *chat*.

5.3 Implementation of Signaling for Direct Connection to XMPP Server

register

In the function *register*, the third-party library Stroph.js, mentioned in Section 3.3, is applied. It is used to set up connection to Ejabberd server through WebSocket service *https://f2f.chat:5280/http-bind*.

Algorithm 5.4 Function Swicher

```
swicher = (data) => {
   switch (data.type) {
      case "start":
         this.onStart();
         break;
      case "connect":
         this.onConnect(data);
         break;
      case "offer":
         this.onOffer(data);
         break;
      case "answer":
         this.onAnswer(data);
         break;
      case "candidate":
         this.onCandidate(data.candidate);
         break;
      case "chat":
         this.onChat(data);
         break;
      default:
         break;
   }
}
```

registerCallback

When feedback is sent back from Ejabberd server, function *registerCallback* will be called. According to feedback status, it executes different programs. Status include *REGISTER*, *REGISTERED*, *CONFLICT*, *NOTACCEPTABLE*, *REGIFAIL* and *CONNECTED*. Except for those status, it executes nothing.

msg_handler_cb

Function *msg_handler_cb* is called to handle received messages. Every time Ejabberd push a message to client, *msg_handler_cb* is triggered. It parses the message and extracts the core data from the original data from Ejabberd server. And then it sends the data to *swicher* for further handling.

createRoom

Function *createRoom* is used for creating a chat room for multi user chat.

sendChat

It implements the interface *sendChat* defined in *SignalService*.

sendGroupChat

It implements the interface *sendGroupChat* defined in *SignalService*.

5.4 Implementation of Signaling for Indirect Connection to XMPP Server

5.4.1 Core Functions at Front End

register

In function *register*, it send request to back end and the back end generates a UUID for new user. In addition, it registers new user on Ejabberd server by utilizing the UUID as part of the account.

connect

Function *connect* is used for establish a connection between client and back end server. By applying the third-party library SockJS, it first generates a WebSocket instance establishing a connection to the target server. Then by applying the third-party library Stomp.js, it generates a stomp client to support text based message through WebSocket. After that, the stomp client subscribes the service provided from the back end server.

sendChat

It implements the interface *sendChat* defined in *SignalService*.

sendGroupChat

It implements the interface *sendGroupChat* defined in *SignalService*.

5.4.2 Core Functions at Back End

CallController

CallController works as a dispatcher. When request is sent from client, *CallController* handles the request and call the corresponding service including registering user and establishing WebSocket.

CallService: register

register is used to register a new account in XMPP server.

CallService: login

After registering a new account, function *login* can be used to login to XMPP server to enable chatting.

CallService: send

Obviously, function *send* is used for sending messages through XMPP server.

CallService: openConnection

openConnection is used to open the *XMPPTCPConnection* which is an API provided by *Smack* to establish connection to XMPP server.

CallService: closeConnection

closeConnection is used to close the *XMPPTCPConnection* in order to terminate the connection.

WebSocketConfig

WebSocketConfig configures the WebSocket connected with client. It defines the application destination prefixes. It also defines the URL which is called endpoint. It enables Cross-origin resource sharing. It also sets the WebSocket handler. In addition, it enables the support for *SockJS*.

CallMessageListener

CallMessageListener listens to the messages. As long as there is a new coming message, it will handle it and send forward to the target.

CallWebSocketHandler

CallWebSocketHandler handles things such as afterConnected, handleException, handleTransportError, getPayloadType and handleFrame.

5.5 Implementation of Measurement

sizeof

Function *sizeof* is a third-party library which is used for computing data size. By setting it to where data is received and display it in dialog, the data flow size can be captured.

printTimestamp

Function *printTimestamp* is used for displaying timestamp. This function can be set anywhere needed.

5.6 Challenges in Implementation

It is full of challenges on the way of implementation. But it is exciting when solving hard problems.

5.6.1 Echo problem

When video and audio resources were first loaded locally, it is full of noise. A lot of time were spent on research how WebRTC acts with echo problem. Some were talking about theory while some were recommanding tools. However, a super simple solution were found when I muted the local audio by accident. Then the world became calm.

5.6.2 Advanced configuration for Ejabberd

Many rare exceptions happened during implementation with using Ejabberd. It has many settings by default, but seems not working in my situation. In order to make things work, I have changed many places in the configuration file.

Finding *registration_timeout* and setting it to *infinity* enable that I can register new users as many as possible. Otherwise, it threw some rare exceptions while I registered too many new users in a short time.

Creating account over insecure connection will throw an exception in future versions of Smack if AccountManagerṡensitiveOperationOverInsecureConnection(true) is not set. It was due to trusted_network tag had value as *loopback:allow* in ejabberd config file. I changed it to *all:allow*. Things started to work.

Owner privileges are required. By setting *all: allow* for creating MUC can solve the problem.

5.6.3 Strict order of Signaling Process

As described in section 4.2, caller and callee connect to server and exchange necessary information in order to establish peer connection between each other. This process has to be strictly in the same order as described. Otherwise, it would throw an exception showing the current status. However, it did not give more information about what the previous status is and what the next status is.

5.6.4 Security of Signaling Process

In order to secure the signaling process, the connections between peer and server are secured by TLS. All the connections are transferred through HTTPS. When testing locally, I use self signed certificate. When testing on Amazon server, I use certificate provided by Amazon.

Chapter 6
Results and Analysis

This chapter shows how the application looks like, describes what kind of data are collected, how the data are collected and how the data are handled from experiments. By analyzing the data, it concludes some results and based on the results deeper research are conducted.

6.1 Screen Shots

This section show how application looks like from startup to connection established.

Figure 6.1 shows the screen shot of initiating application.

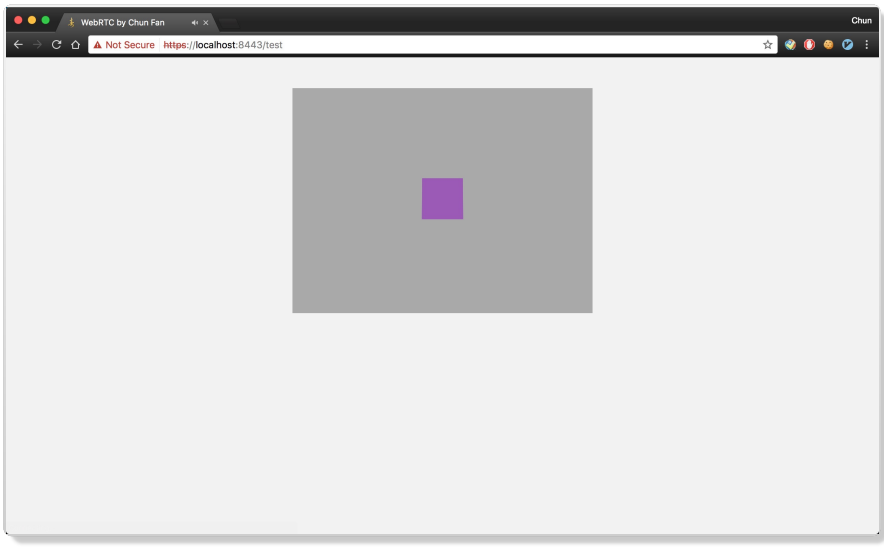

Figure 6.1: Screen Shots: Initiating Application

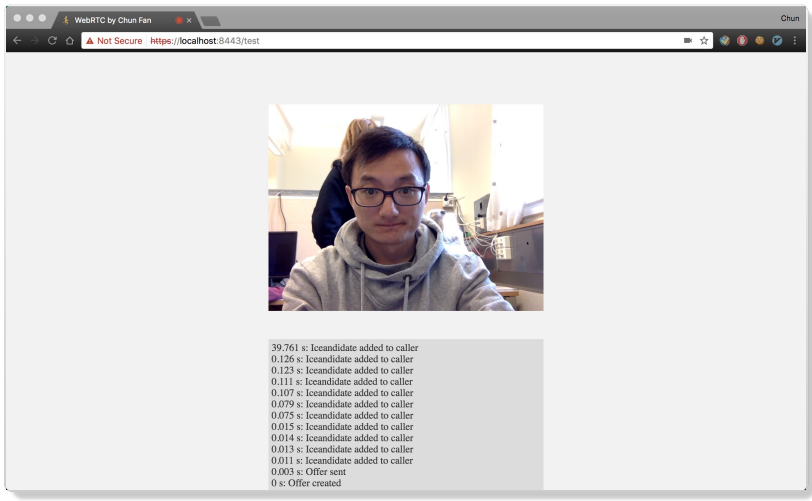

Figure 6.2: Screen Shots: Application Initiated

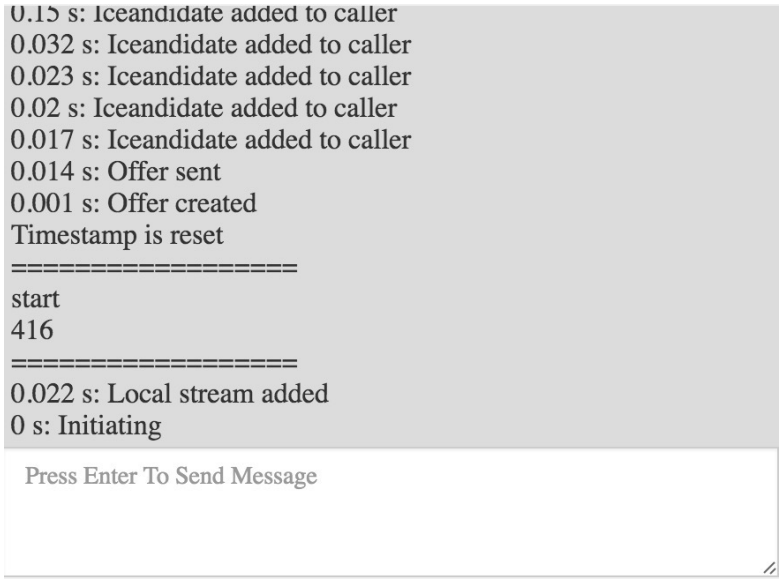

Figure 6.3: Screen Shots: Dialog

Figure 6.2 shows the screen shot of application having initiated.

Figure 6.3 shows the screen shot of dialog of chatting filed. This chatting field is used for chatting. Meanwhile, it is used for displaying time costs and data flow sizes.

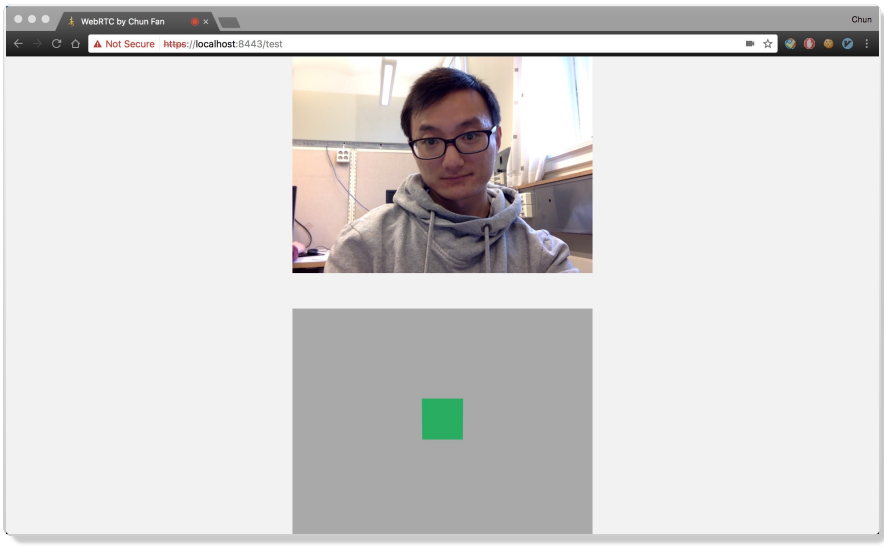

Figure 6.4: Screen Shots: Joining of Participant

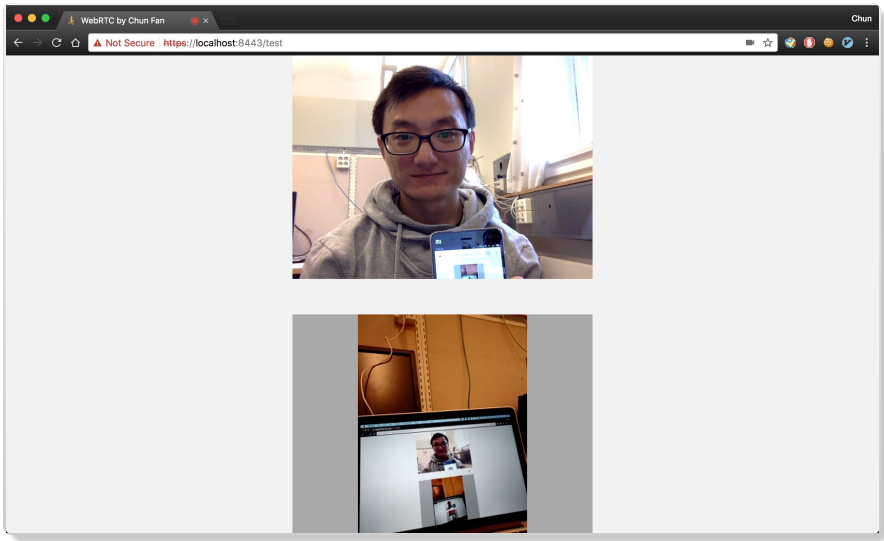

Figure 6.5: Screen Shots: Participant Joined

Figure 6.4 shows the screen shot of joining of another participant.

Figure 6.5 shows the screen shot of another participant having joined.

Table 6.1: Experiment Environment 1

Location	Trondheim, Norway
OS	Windows
Process	AMD A10-5800K APU with Radeon™ HD Graphics 3.8GHz
Memory	16 GB
System Type	Windows 10 Pro 64-bit Operating System, x64-based processor
Browser	Google Chrome Version 56.0.2924.87
Signaling Type	Direct and Indirect

Table 6.2: Experiment Environment 2

Location	Trondheim, Norway
OS	Mac
Process	2.8 GHz Intel Core i5
Memory	16 GB 1600 MHz DDR3
System Type	OS X EI Captain Version 10.11.6
Browser	Google Chrome Version 58.0.3029.81 (64-bit)
Signaling Type	Direct and Indirect

Table 6.3: Experiment Environment 3

Location	Wuhan, China
OS	Windows
Process	Intel® Core™ i3-4150 CPU @ 3.50GHz 3.50GHz
Memory	4.00GB (3.71GB usable)
System Type	Windows 8 Pro 64-bit Operating Syste, x64-based processor
Browser	Google Chrome Version 58.0.3029.81 (64-bit)
Signaling Type	Direct and Indirect

6.2 Environment of Experiments

According to different conditions such as location, operating system, processor, memory, system type and browser, several groups of experiments have been conducted and data from the following cases have been collected showing in the following tables from Table 6.1 to Table 6.3.

Table 6.4: Data Flow Size

Index	Direct Model (byte)	Indirect Model (byte)
Experiment 1	716973	12066.8
Experiment 2	711600	12047.8
Experiment 3	447945.2	11437.6

6.3 Collections of Data

In order to collect the data needed as results to be analyzed, the experiment is conducted as follows.

1. Select a callee and record basic information of the callee.
2. Then select a caller and record basic information of the caller.
3. As callee, visit the application on https://direct.f2f.chat/test.
4. Until callee successfully get ready for call, the caller visit the application on web. Temporarily the application is running on https://direct.f2f.chat/test.
5. Collect the data including candidate number, total time cost, total data flow.
6. Once again, as callee, visit the application on https://indirect.f2f.chat/test.
7. Until callee successfully get ready for call, the caller visit the application on web. Temporarily the application is running on https://indirect.f2f.chat/test.
8. Collect data the same as above.
9. Repeat steps above ten times.

Since candidate numbers in each case are always the same in the ten times' experiment, they are kept the same as from the original data. The same as candidate numbers, received candidate numbers are handled in the same way. Total time costs in each case are calculated as an average value from the ten times' experiment. The same as total time costs, signaling time costs without candidate are calculated in the same way. Total message size in each case are calculated as an average value from the ten times' experiment. Answer size and candidate size are calculated in the same way as total message size.

6.4 Analysis of Data

6.4.1 Results

Data Flow Size

Table 6.4 shows the summary of data flow sizes from the experiments. From Table C.1 and Table C.2, we can see that in experiment 1 the average data flow sizes are 716973 bytes for the Direct model and 12066.8 bytes for the Indirect model. Respectively in experiment 2, the average data flow sizes are 711600 bytes and 12047.8 bytes, see Table C.3 and Table C.4.

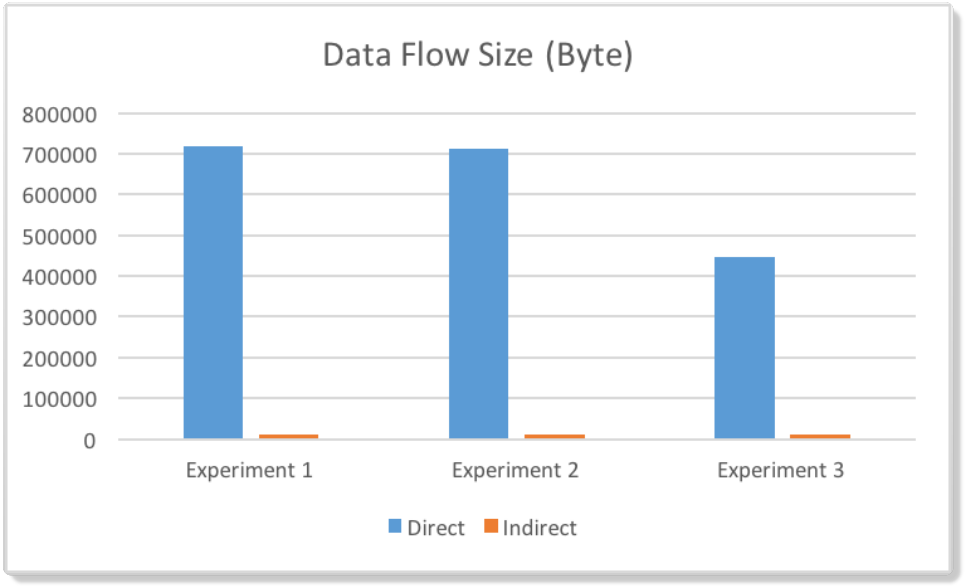

Figure 6.6: Data Flow Size in Different Experiments

Table 6.5: Time Cost

Index	Direct Model (second)	Indirect Model (second)
Experiment 1	11.939	7.347
Experiment 2	2.292	6.110
Experiment 3	5.653	7.397

And in experiment 3, the average data flow sizes are 447945.2 bytes and 11437.6 bytes, see Table C.5 and Table C.6.

So from Figure 6.6 we can see obviously, the size of data flow in the model by using a middlware server is much less than the size of data flow in the model without a middleware. Data flow size affects the transfer time between peer to peer. When data flow size is larger, it takes longer time for transferring. Thus, the indirect model does improve the performance of WebRTC signaling according to the data flow size.

Time Cost

Table 6.5 shows the summary of time cost from the experiments. From Table C.1 and Table C.2, we can see that in experiment 1 the average time cost in direct signaling model is more than the average time cost in indirect signaling model which are 11.939 seconds and 7.347 seconds. While in experiment 2 the average time costs are 2.292 seconds and 6.110

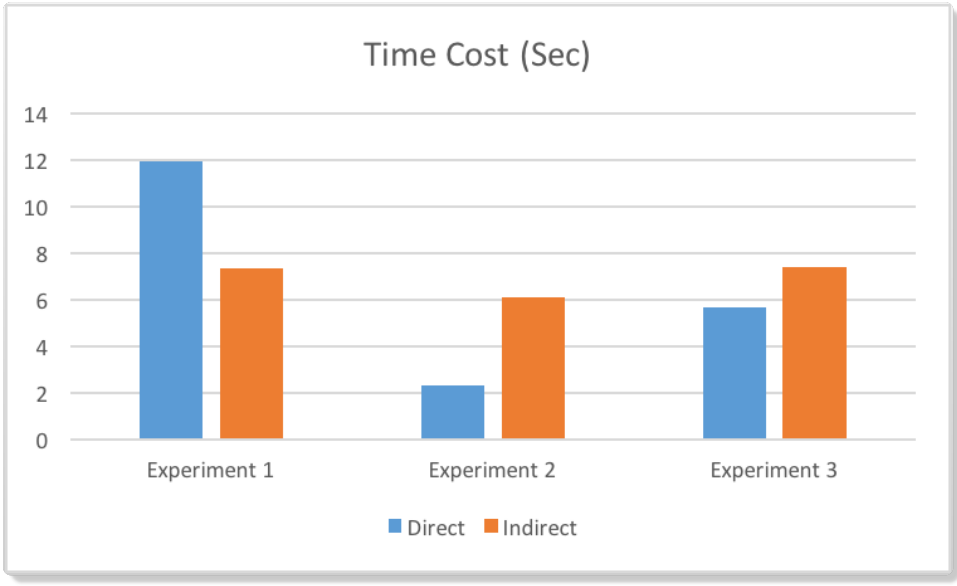

Figure 6.7: Time Cost in Different Experiments

seconds respectively and in experiment 3 the average time costs are 5.653 seconds and 7.397 seconds respectively which shows the average time cost in direct signaling model is less than the average time cost in indirect signaling model. This does not indicates which method is absolutely faster than another method. However, we can see that when the number of candidates is increasing from one case to another, the average time costs increase differently. Comparing experiment 2 and experiment 1, the numbers of candidates increase from 5 to 17 and from 6 to 17 while the average time costs increase from 2.292 seconds to 11.939 seconds and from 6.110 seconds to 7.347 seconds. And also, in all cases the average time costs in direct signaling model varies from 2.292 seconds to 11.939 seconds while the average time costs in indirect signaling model varies from 6.110 seconds to 7.397 seconds.

So as we can see from Figure 6.7 the average time costs in indirect signaling model looks more stable than the average time costs in direct signaling model. More stable means that, when the number of candidates grows even larger, the time cost of WebRTC signaling would not increase that much.

Number of Candidates

Table 6.6 shows the summary of number of candidates from the experiments. In Figure 6.8, it shows that in experiment 1 the number of candidates is 17 both by using direct signaling model and indirect signaling model while in experiment 2 the number of candidates is 5 by using direct signaling model and 6 by using indirect signaling Model. Respectively in experiment 3, the numbers of candidates are 8 and 6. So the signaling method does not

Table 6.6: Number of Candidates

Index	Direct Model (second)	Indirect Model (second)
Experiment 1	17	17
Experiment 2	5	6
Experiment 3	8	6

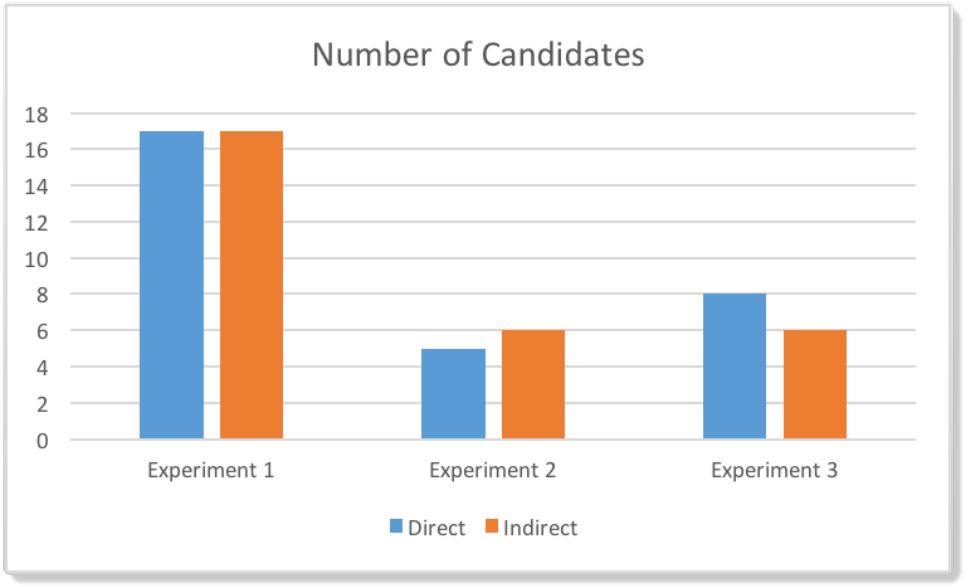

Figure 6.8: Number of Candidates in Different Experiments

affect much on the number of candidates . However, number of candidates varies a lot according to different conditions and affects the performance of the signaling process. To clearly understand how candidates influence the performance of the signaling process, additional research should be conducted.

6.4.2 Reasoning

Research on ICE Candidate

In order to figure out why the candidate numbers are so various according to different conditions, a deeper research on WebRTC *ICE Candidate* are conducted.

As described in abstract, *ICE Candidate* is an object which contains information for establishing communication between peers. Each *ICE Candidate* describes a method which the originating peer is able to communicate. Each peer sends candidates in the order of

discovery, and keeps sending until it runs out of suggestions, even if media has already started streaming. Once the two peers suggest a compatible candidate, media begins to flow. If they later agree on a better pairing (usually higher-performance), the stream may change formats as needed.[25]

In the signaling process, the offer and answer have been exchanged and carried the information to set up calls between users. But they do not carry information about addresses except for local address. However, the ICE candidates carry the information about target addresses. Example of a candidate in Implementation 6.1 shows more information about the addresses. Therefore, to enable the communication between peers after exchanging, the two peers will exchange *ICE Candidate*s.

Implementation 6.1 Example of *ICE Candidate*

```
"candidate:2241210590 1 udp 2122260223 192.168.2.101 54318 typ
    host generation 0 ufrag xTxe network-id 1 network-cost 10"
```

Implementation 6.2 Interface of *ICE Candidate*[2]

```
[Constructor(RTCIceCandidateInit candidateInitDict)]
interface RTCIceCandidate {
   readonly attribute DOMString candidate;
   readonly attribute DOMString? sdpMid;
   readonly attribute unsigned short? sdpMLineIndex;
   readonly attribute DOMString? foundation;
   readonly attribute unsigned long? priority;
   readonly attribute DOMString? ip;
   readonly attribute RTCIceProtocol? protocol;
   readonly attribute unsigned short? port;
   readonly attribute RTCIceCandidateType?type;
   readonly attribute RTCIceTcpCandidateType? tcpType;
   readonly attribute DOMString? relatedAddress;
   readonly attribute unsigned short? relatedPort;
   readonly attribute DOMString? ufrag;
   serializer = {candidate, sdpMid, sdpMLineIndex, ufrag};
};
```

In model design, it mentioned that the *ICE Candidate*s are exchanged during the signaling process. When candidates are generated by a client, they are then sent to other clients. In implementation, the RTCIceCandidate interface in Impelmentation 6.2 were called to execute *ICE Candidate*. The RTCIceCandidate() constructor takes a dictionary argument, candidateInitDict, whose content is used to initialize the new RTCIceCandidate object. When run, if both the sdpMid and sdpMLineIndex dictionary members are null, throw a TypeError. Besides the constructor, RTCIceCandidate also has thirteen attributes

including candidate, sdpMid, sdpMLineIndex, foundation, priority, ip, protocol, port, type, tcpType, relatedAddress, relatedPort, ufrag.

- candidate. This carries the candidate-attribute as defined in section 15.1 of [ICE][32]. If this RTCIceCandidate represents an end-of-candidates indication, candidate is an empty string.
- sdpMid. This contains the identifier of the "media stream identification" as defined in [RFC5888] for the media component this candidate is associated with.
- sdpMLineIndex. This indicates the index (starting at zero) of the media description in the SDP this candidate is associated with.
- foundation. This is a unique identifier that allows ICE to correlate candidates that appear on multiple RTCIceTransports.
- priority. This is the assigned priority of the candidate.
- ip. This is the IP address of the candidate.
- protocol. This is the protocol of the candidate (udp/tcp).
- port. This is the port of the candidate.
- type. This is the type of the candidate.
- tcpType. If protocol is tcp, tcpType represents the type of TCP candidate.
- relatedAddress. For a candidate that is derived from another, such as a relay or reflexive candidate, the relatedAddress is the IP address of the candidate that it is derived from. For host candidates, the relatedAddress is null.
- relatedPort. For a candidate that is derived from another, such as a relay or reflexive candidate, the relatedPort is the port of the candidate that it is derived from. For host candidates, the relatedPort is null.
- ufrag. This carries the ufrag as defined in section 15.4 of [ICE][32].

Because of the big group of parameters, any change of network condition will lead to generate a new *ICE candidate*. The number of candidates may be affected by IP address, port for a particular transport protocol, physical or logical network interfaces, Virtual Private Network (VPN) or Mobile IP (MIP), private network, public Internet. Therefore the number of candidates varies a lot according to different conditions. As one network condition differs from another, the number of candidates according to the specific network will change.

In order to generate the *ICE Candidate*s and exchange them, the following steps should be executed.[32].

1. Gathering Candidate Addresses.
2. Connectivity Checks.
3. Sorting Candidates.
4. Frozen Candidates.
5. Security for Checks.
6. Concluding ICE.

As explained above, the process from generating candidate address to concluding ICE and exchange *ICE Candidate*s is a complex process. Each step needs time to compute. That

```
chun@ChunMac:~|⇒  ifconfig
lo0: flags=8049<UP,LOOPBACK,RUNNING,MULTICAST> mtu 16384
        options=3<RXCSUM,TXCSUM>
        inet6 ::1 prefixlen 128
        inet 127.0.0.1 netmask 0xff000000
        inet6 fe80::1%lo0 prefixlen 64 scopeid 0x1
        nd6 options=1<PERFORMNUD>
gif0: flags=8010<POINTOPOINT,MULTICAST> mtu 1280
stf0: flags=0<> mtu 1280
en0: flags=8863<UP,BROADCAST,SMART,RUNNING,SIMPLEX,MULTICAST> mtu 1500
        ether 60:f8:1d:ba:1a:d6
        inet6 fe80::62f8:1dff:feba:1ad6%en0 prefixlen 64 scopeid 0x4
        inet6 2001:700:300:4010:62f8:1dff:feba:1ad6 prefixlen 64 autoconf
        inet6 2001:700:300:4010:8095:f72a:29d8:30b5 prefixlen 64 autoconf temporary
        inet 10.22.67.94 netmask 0xfffffc00 broadcast 10.22.67.255
        nd6 options=1<PERFORMNUD>
        media: autoselect
        status: active
```

Figure 6.9: Network Condition of Experiment

is why when the number of candidates increases, the time cost of signaling process is increasing.

6.4.3 Additional Experiment on ICE Candidate

To make sure everything is correct, an additional experiment on *ICE Candidate* has been conducted.

Figure 6.9 shows the network condition of the experiment. It contains both IPv4 and IPv6 addresses. IPv4 address is 10.22.67.94 while IPv6 address is 2001:700:300:4010:8095:f72a:29d8:30b5. Implementation 6.3 displays the generated candidates and Implementation 6.4 displays the received candidates. Apparently, each IP address has at least one candidate including IPv4 and IPv6. And for both UDP and TCP, it has different candidates. *sdpMid* has two types here, video and audio. So *sdpMid* also leads to more candidates.

As we can see from Implementation 6.4, not all candidates are exchanged. Even though not all candidates are exchanged, the process of handling candidates takes time. The greater the number of candidates is, the more signaling time costs. So optimizing the way it handling *ICE Candidate* can help to optimize the performance of WebRTC signaling.

Therefore in the experiments, the more candidates it is, the longer delay it would be according to different situations.

Implementation 6.3 Generated Candidates


```
{"candidate":"candidate:2636380675 1 udp 2122262783
    2001:700:300:4010:8095:f72a:29d8:30b5 54971 typ host
    generation 0 ufrag TtVx network-id 2 network-cost 10","
    sdpMid":"audio","sdpMLineIndex":0}

{"candidate":"candidate:2782990128 1 udp 2122194687 10.22.67.94
     60359 typ host generation 0 ufrag TtVx network-id 1
    network-cost 10","sdpMid":"audio","sdpMLineIndex":0}

{"candidate":"candidate:2636380675 1 udp 2122262783
    2001:700:300:4010:8095:f72a:29d8:30b5 60360 typ host
    generation 0 ufrag TtVx network-id 2 network-cost 10","
    sdpMid":"video","sdpMLineIndex":1}

{"candidate":"candidate:2782990128 1 udp 2122194687 10.22.67.94
     65495 typ host generation 0 ufrag TtVx network-id 1
    network-cost 10","sdpMid":"video","sdpMLineIndex":1}

{"candidate":"candidate:213811429 1 udp 1685987071
    129.241.228.97 40493 typ srflx raddr 10.22.67.94 rport
    60359 generation 0 ufrag TtVx network-id 1 network-cost
    10","sdpMid":"audio","sdpMLineIndex":0}

{"candidate":"candidate:213811429 1 udp 1685987071
    129.241.228.97 37787 typ srflx raddr 10.22.67.94 rport
    65495 generation 0 ufrag TtVx network-id 1 network-cost
    10","sdpMid":"video","sdpMLineIndex":1}

{"candidate":"candidate:3550480115 1 tcp 1518283007
    2001:700:300:4010:8095:f72a:29d8:30b5 9 typ host tcptype
    active generation 0 ufrag TtVx network-id 2 network-cost
    10","sdpMid":"audio","sdpMLineIndex":0}

{"candidate":"candidate:3949130688 1 tcp 1518214911 10.22.67.94
     9 typ host tcptype active generation 0 ufrag TtVx network-
    id 1 network-cost 10","sdpMid":"audio","sdpMLineIndex":0}

{"candidate":"candidate:3550480115 1 tcp 1518283007
    2001:700:300:4010:8095:f72a:29d8:30b5 9 typ host tcptype
    active generation 0 ufrag TtVx network-id 2 network-cost
    10","sdpMid":"video","sdpMLineIndex":1}

{"candidate":"candidate:3949130688 1 tcp 1518214911 10.22.67.94
     9 typ host tcptype active generation 0 ufrag TtVx network-
    id 1 network-cost 10","sdpMid":"video","sdpMLineIndex":1}
```

Implementation 6.4 Received Candidates

```
{"candidate":"candidate:2636380675 1 udp 2122262783
    2001:700:300:4010:8095:f72a:29d8:30b5 51873 typ host
    generation 0 ufrag RhUW network-id 2 network-cost 10","
    sdpMid":"audio","sdpMLineIndex":0}

{"candidate":"candidate:2782990128 1 udp 2122194687 10.22.67.94
     49691 typ host generation 0 ufrag RhUW network-id 1
    network-cost 10","sdpMid":"audio","sdpMLineIndex":0}

{"candidate":"candidate:213811429 1 udp 1685987071
    129.241.228.97 63125 typ srflx raddr 10.22.67.94 rport
    49691 generation 0 ufrag RhUW network-id 1 network-cost
    10","sdpMid":"audio","sdpMLineIndex":0}

{"candidate":"candidate:3550480115 1 tcp 1518283007
    2001:700:300:4010:8095:f72a:29d8:30b5 9 typ host tcptype
    active generation 0 ufrag RhUW network-id 2 network-cost
    10","sdpMid":"audio","sdpMLineIndex":0}

{"candidate":"candidate:3949130688 1 tcp 1518214911 10.22.67.94
     9 typ host tcptype active generation 0 ufrag RhUW network-
    id 1 network-cost 10","sdpMid":"audio","sdpMLineIndex":0}
```

Chapter 7
Conclusion

WebRTC as a new technology based on Web is becoming more and more mature. W3C defines the standard of WebRTC APIs used for establishing connection to remote peers, sending and receiving tracks from remote peers and sending arbitrary data directly to remote peers. IETF defines a set of standards for WebRTC communication between peers. Even though the standard for WebRTC signaling has not been defined yet, there are many good solutions for it. XMPP as a mature protocol, which is open standard for messaging and presence, is one of those proposed signaling solutions for WebRTC.

After studying on WebRTC, on signaling and on XMPP as a solution for WebRTC signaling, this research first designed one model that WebRTC supported browser connects to XMPP server directly which is the basic way to design. As thinking of that XMPP is a heavy text based protocol which may lead to too much traffic through the Internet, this research designed another model that building a middleware server on the same sever as XMPP server, so that WebRTC browser communicates with the middleware sever with core information and the middleware server communicates with XMPP server with XMPP supported message, and thus the majority of traffic are moved from Internet to local. It then assumed that the latter model may improve performance of delay for WebRTC signaling.

In order to validate the assumption, two models have been implemented. In addition, to analyze two models, the validation methods have also been implemented. Many technologies have been used during implementations. JavaScript is used as the main development language for front end and Java for back end. Angular is chosen as front end framework and Spring is chosen as back end framework. Ejabberd is used as XMPP server. For the model with direct connection to XMPP server, StrophJS is used to establish WebSocket connection between WebRTC supported browser and Ejabberd, while for the model with middleware, SockJs and STOMP are used to transfer data over WebSocket and Smack is used for back end communicating with Ejabberd.

Even though many challenges were faced during implementation such as echo problem, advanced configuration for Ejabberd, strict order of signaling process and security of

signaling process, it is an exciting work and in the end both models work. Therefore, it convinced that XMPP as WebRTC signaling method can be a valid solution. In addition, methods of measurement also work fine.

Analysis was then followed by experiments after implementation. Three experiments have been done according to different situations such as location, operation system, process, memory, browser and signaling types. The analysis of delay performance based on the three experiments has been conducted to find out in quantity that building a middleware server together with XMPP server can move most data from Internet to local. However, building a middleware server can not guarantee less delay. But the delay is more stable in the model with middlware than in the model with direct connection to XMPP server, since the delays for the model with direct connection to XMPP server vary over ten seconds while the delays for the model with middleware vary less than one second.

From figures about delay and number of candidates, it looks that they may have certain relationship between each other, thus additional research and experiment on ICE candidate have been conducted. It finds out that number of candidates does affect the delay performance of WebRTC signaling. The less the number of candidates is, the less delay it will be. ICE candidate message consists of many elements including candidate, sdbMid, sdpMLineIndex, foundation, priority, ip, protocol, port, type, tcpType, relatedAddress, relatedPort and ufrag. As long as any of these elements has new parameters, a new candidate would be generated, the number of candidates thus would increase. In addition, process of ICE candidates is a complex process with six core steps which take time. Therefore, as condition of network becomes more and more complex, the number of candidates will be greater, leading to longer delay of WebRTC signaling process.

WebRTC is a nice technology for peer to peer communication, and XMPP works fine as a solution for WebRTC signaling. ICE candidate as an important role in WebRTC signaling affects much on delay performance of WebRTC signaling. More research on ICE candidate can help to improve the delay performance of WebRTC signaling.

References

[1] S. Loreto, *Realtime Communication with WebRTC : Peer-to-Peer in the Browser*. Salvatore Loreto and Simon Pietro Romano: O'Reilly Media, 2014.

[2] C. J. A. N. B. A. Adam Bergkvist, Daniel C. Burnett, "Webrtc 1.0: Real-time communication between browsers (w3c editor's draft 13 march 2017)." https://w3c.github.io/webrtc-pc/archives/20170313/webrtc.html, 2017. Online; accessed 1 May 2017.

[3] I. T. U. I. Telecommunication Development Bureau, "Ict facts and figures 2005, 2010, 2016." http://www.itu.int/en/ITU-D/Statistics/Pages/facts/default.aspx, 2016. Online; accessed May 24th 2015.

[4] W3C, "What is the difference between the web and the internet?." W3C Help and FAQ, 2009. Retrieved 16 July 2015.

[5] webrtc.org, "Webrtc is a free, open project." Home page, 2016. Retrieved 1 October 2016.

[6] S. Dutton, "Getting started with webrtc." https://www.html5rocks.com/en/tutorials/webrtc/basics/, 2016. Online; accessed 1 Sepetmber 2016.

[7] B. Sredojev, D. Samardzija, and D. Posarac, "Webrtc technology overview and signaling solution design and implementation," in *Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2015 38th International Convention on*, pp. 1006–1009, May 2015.

[8] M. Adeyeye, I. Makitla, and T. Fogwill, "Determining the signalling overhead of two common webrtc methods: Json via xmlhttprequest and sip over websocket," in *AFRICON, 2013*, pp. 1–5, Sept 2013.

[9] P. Saint-Andre, "Jingle: Jabber does multimedia," *IEEE MultiMedia*, vol. 14, pp. 90–94, Jan 2007.

[10] J. Paik and D. H. Lee, "Scalable signaling protocol for web real-time communication based on a distributed hash table," *Computer Communications*, vol. 70, pp. 28–39, 2015.

[11] xmpp.org, "An overview of xmpp." https://xmpp.org/about/technology-overview.html, 2016. Online; accessed 1 Sepetmber 2016.

[12] J. G. van Bosse, *Signaling in Telecommunication Networks*. Northeastern University: Wiley, 2002.

[13] Wikipedia, "Signaling protocol." https://en.wikipedia.org/wiki/Signaling_protocol, 2016. Online; accessed 1 Sepetmber 2016.

[14] N. W. Group, "Extensible messaging and presence protocol (xmpp): Core." https://xmpp.org/rfcs/rfc3920.html, 2004. Online; accessed 1 Sepetmber 2016.

[15] N. W. Group, "Extensible messaging and presence protocol (xmpp): Instant messaging and presence." https://xmpp.org/rfcs/rfc3921.html, 2004. Online; accessed 1 Sepetmber 2016.

[16] I. E. T. F. (IETF), "Extensible messaging and presence protocol (xmpp): Core." https://xmpp.org/rfcs/rfc6120.html, 2011. Online; accessed 1 Sepetmber 2016.

[17] I. E. T. F. (IETF), "Extensible messaging and presence protocol (xmpp): Instant messaging and presence." https://xmpp.org/rfcs/rfc6121.html, 2011. Online; accessed 1 Sepetmber 2016.

[18] I. E. T. F. (IETF), "Extensible messaging and presence protocol (xmpp): Address format." https://tools.ietf.org/html/rfc7622, 2015. Online; accessed 1 Sepetmber 2016.

[19] e. a. Rosenberg, "Sip: Session initiation protocol." https://www.ietf.org/rfc/rfc3261.txt, 2002. Online; accessed 1 Sepetmber 2016.

[20] R. Sparks, "Sip: Basics and beyond," *Queue*, vol. 5, pp. 22–33, Mar. 2007.

[21] P. S.-A. R. M. S. E. J. H. Scott Ludwig, Joe Beda, "Xep-0166: Jingle." https://xmpp.org/extensions/xep-0166.html, 2016. Online; accessed 1 Sepetmber 2016.

[22] P. Krill, "Ajax alliance recognizes mashups," *InfoWorld*.

[23] C. C. McCarthy, Dennis, *Comet and Reverse Ajax: The Next-Generation Ajax 2.0*. Apress, 2008.

[24] A. M. I. Fette, "The websocket protocol." https://tools.ietf.org/html/rfc6455, 2016. Online; accessed 1 Sepetmber 2016.

[25] Mozilla, "Webrtc api." https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API, 2017. Online; accessed 1 May 2017.

[26] Ejabberd, "Ejabberd robust, scalable and extensible xmpp server." https://www.ejabberd.im/, 2016. Online; accessed 1 Sepetmber 2016.

[27] STOMP, "Stomp the simple text oriented messaging protocol." https://stomp.github.io/, 2016. Online; accessed 1 Sepetmber 2016.

[28] Smack, "Smack overview." http://download.igniterealtime.org/smack/docs/latest/documentation/overview.html, 2016. Online; accessed 1 Sepetmber 2016.

[29] J. Moffitt, “Strophe.js an xmpp library for javascript.” http://strophe.im/strophejs/, 2016. Online; accessed 1 Sepetmber 2016.

[30] Angular, “Angular.” https://angularjs.org/, 2016. Online; accessed 1 Sepetmber 2016.

[31] Spring, “Let’s build a better enterprise.” https://spring.io/, 2016. Online; accessed 1 Sepetmber 2016.

[32] J. Rosenberg, “Interactive connectivity establishment (ice): A protocol for network address translator (nat) traversal for offer/answer protocols.” https://tools.ietf.org/html/rfc5245, 2016. Online; accessed 1 Sepetmber 2016.

Chapter A
Front End Implementation

A.1 Component Layer

A.1.1 AppComponent

```
import {Component, AfterViewInit, NgZone} from '@angular/core'

import {SignalService} from "../service/signal2";
import {SettingService} from "../service/setting";
import {DirectConnectionService} from "../service/
    directConnection";
import {IndirectConnectionService} from "../service/
    indirectConnection";

@Component({
   selector: 'my-app',
   template: `
   <div class="video-background">
      <div class="video-background" *ngFor="let video of
          videos; let first = first; let i = index;">
         <my-video [(videoId)]="video.videoId" [(isMuted)]="
             video.isMuted" [(isLoaded)]="video.isLoaded" [(
             source)]="video.source"></my-video>
      </div>
   </div>
   <div class="chat-background">
      <my-chat #msgs></my-chat>
   </div>
   `,
   providers: [SettingService, SignalService,
       DirectConnectionService, IndirectConnectionService]
})

export class App implements AfterViewInit {
```

```
   videos = [
      {videoId: 'localVideo', isMuted: 'muted', isLoaded: true,
          source: null}
   ];

   constructor(private signal:SignalService, private setting:
       SettingService, private zone:NgZone) {
      this.signal = signal;
   }

   ngAfterViewInit():void {
      this.setting.setVideos(this.videos);
      this.setting.videosUpdated.subscribe(
         (videos) => {
            this.zone.run(
               ()=> {
                  this.videos = videos;
               }
            );
         }
      );
      this.signal.onInit();
   }
}
```

A.1.2 VideoComponent

```
import {Component, Input} from '@angular/core';

@Component({
   selector: 'my-video',
   template: `
   <div class="video-content" *ngIf="!isLoaded">
      <my-video-loader></my-video-loader>
   </div>
   <div class="video-content" *ngIf="isLoaded">
      <video autoplay id="{{videoId}}" muted="{{isMuted}}"
         class="video-container" src="{{source}}"></video>
   </div>
   `,
   providers: []
})

export class Video {
   @Input() videoId:string;
   @Input() isMuted:string;
   @Input() isLoaded:boolean;
   @Input() source:any;
}
```

A.1.3 ChatComponent

```
import {Component, NgZone} from '@angular/core';
import {SettingService} from "../service/setting";
import {SignalService} from "../service/signal2";

@Component({
   selector: 'my-chat',
   template: `
   <div class="chat-container">
      <div class="form-group">
         <div class="message-container">
            <p *ngFor="let msg of msgs; let first = first;
                let i = index;">
               {{ msg }}
            </p>
         </div>
         <textarea class="form-control" rows="3" id="comment"
             placeholder="Press Enter To Send Message"
         (keyup.enter)="onEnter()" [(ngModel)]="inputValue"></
             textarea>
      </div>
   </div>
   `,
   providers: []
})

export class Chat {
   msgs = [];
   inputValue:string;

   constructor(private signal:SignalService, private setting:
       SettingService, private zone:NgZone) {
      this.signal = signal;
      this.setting = setting;

      this.setting.chatsUpdated.subscribe(
         (msgs)=> {
            this.zone.run(
               ()=> {
                  this.msgs = msgs;
                  this.msgs.reverse();
               }
            );
         }
      );
```

```
    }

    onEnter() {
        console.log(this.inputValue);
        this.setting.addChat("Me: " + this.inputValue);
        this.signal.sendGroupChat({
            'type': 'chat',
            'from': this.setting.username,
            'roomName': this.setting.roomName,
            'username': this.setting.username,
            'msg': this.inputValue
        });
        this.inputValue = '';
    }
}
```

A.2 Service Layer

A.2.1 SignalService

```
import {Injectable} from '@angular/core';
import {Http} from '@angular/http';
import {Cookie} from 'ng2-cookies/ng2-cookies';
import 'rxjs/Rx';
import 'webrtc-adapter';
import {SettingService} from "./setting";
import {DirectConnectionService} from "./directConnection";
import {IndirectConnectionService} from "./indirectConnection"
    ;

declare let navigator:any;

@Injectable()
export class SignalService {
   private remoteVideoId;
   private remoteVideoIndex;
   private callerPC;
   private calleePC;
   private localPC;

   constructor(private http:Http, private setting:
       SettingService, private connection:
       DirectConnectionService) {
      this.http = http;
      this.setting = setting;
      this.connection = connection;
   }

   onInit = ()=> {
      this.setting.currentTime = Date.now();
      this.setting.currentSize = 0;
      this.printTimestamp("Initiating");

      // Initiate room attributes
      this.setting.roomName = window.location.pathname.split('/
          ')[1];
      this.setting.username = Cookie.get('webrtc_username');
      this.setting.serverUrl = window.location.protocol + "//"
           + window.location.hostname + ":" + window.location.
          port;
      this.setting.ejabberdUrl = "https://ejabberd.f2f.chat
          :5280/http-bind";
```

```
   // Configure video attributes
   if (this.hasUserMedia()) {
      navigator.getUserMedia = navigator.getUserMedia ||
          navigator.webkitGetUserMedia
         || navigator.mozGetUserMedia;

      //enabling video and audio channels
      navigator.getUserMedia(
         {video: true, audio: true}, stream => {
            let videos = this.setting.getVideos();
            this.addStream(videos[0].videoId, stream);
            this.setting.setVideos(videos);
            this.setting.localStream = stream;
            this.printTimestamp("Local stream added");
            if (!this.setting.username) {
               this.connection.register(this);
            }
            else {
               this.connection.connect(this);
            }
         }, err => {
            console.log(err);
         });

   } else {
      alert("WebRTC is not supported");
   }
};

//check if the browser supports the WebRTC
hasUserMedia = () => {
   return !!(navigator.getUserMedia || navigator.
       webkitGetUserMedia ||
   navigator.mozGetUserMedia);
};

addStream = (videoId, stream) => {
   let video = <HTMLVideoElement>document.getElementById(
       videoId);
   video.srcObject = stream;
};

swicher(data) {
   console.log(data);
```

```
      switch (data.type) {
         case "start":
            this.onStart();
            break;
         case "connect":
            this.onConnect(data);
            break;
         case "offer":
            this.onOffer(data);
            break;
         case "answer":
            this.onAnswer(data);
            break;
         case "candidate":
            this.onCandidate(data.candidate);
            break;
         case "chat":
            this.onChat(data);
            break;
         default:
            break;
      }
   }

   onConnect = (data) => {
      let videos = this.setting.getVideos();
      this.remoteVideoIndex = videos.length;
      this.remoteVideoId = 'remoteVideo' + this.
          remoteVideoIndex;
      videos.push({videoId: this.remoteVideoId, isMuted: '',
          isLoaded: false, source: null});
      this.setting.setVideos(videos);
   };

   onStart = () => {
      let offerOptions = {
         offerToReceiveAudio: true,
         offerToReceiveVideo: true
      };

      this.callerPC = new RTCPeerConnection(this.setting.
          peerConnectionConfig);
      this.callerPC.onicecandidate = event => {
         this.printTimestamp("Iceandidate added to caller");
         if (event && event.candidate) {
```

```
            this.localPC = this.callerPC;
            this.sendGroupChat({
               type: "candidate",
               from: this.setting.username,
               candidate: event.candidate
            });
         }
      };
      this.callerPC.onaddstream = event => {
         let videos = this.setting.getVideos();
         if (videos[this.remoteVideoIndex] !== undefined) {
            videos[this.remoteVideoIndex].isLoaded = true;
            this.setting.setVideos(videos);
            this.addStream(this.remoteVideoId, event.stream);
            this.printTimestamp("Remote_stream_added");
            this.printTimestamp("Done!");
         }
      };
      this.callerPC.addStream(this.setting.localStream);

      // 1. This is the first step, create offer and send
      this.callerPC.createOffer(offerOptions).then(offer=> {
         this.resetTimestamp();
         this.printTimestamp("Offer_created");
         this.callerPC.setLocalDescription(new
             RTCSessionDescription(offer), () => {

            this.printTimestamp("Offer_sent");
            this.localPC = this.callerPC;
            this.sendGroupChat({
               type: "offer",
               from: this.setting.username,
               offer: offer
            });
         }, err => {
            console.log(err);
         });
      }).catch(err=> {
         console.log(err);
      });
   };

   onOffer = (offer) => {
      this.resetTimestamp();
```

```
this.calleePC = new RTCPeerConnection(this.setting.
    peerConnectionConfig);
this.calleePC.onicecandidate = event => {
   if (event && event.candidate) {
      this.printTimestamp("Iceandidate added to callee ");
      this.sendGroupChat({
         type: "candidate",
         from: this.setting.username,
         candidate: event.candidate
      });
   }
};
this.calleePC.onaddstream = event=> {
   let videos = this.setting.getVideos();
   if (videos[this.remoteVideoIndex] !== undefined) {
      videos[this.remoteVideoIndex].isLoaded = true;
      this.setting.setVideos(videos);
      this.addStream(this.remoteVideoId, event.stream);
      this.printTimestamp("Remote stream added");
      this.printTimestamp("Done!");
   }
};
this.calleePC.addStream(this.setting.localStream);

// 2. This is the second step, get offer and send answer
this.calleePC.setRemoteDescription(new
    RTCSessionDescription(offer.offer), () => {
   this.calleePC.createAnswer(answer => {
      this.printTimestamp("Answer created");
      this.calleePC.setLocalDescription(new
          RTCSessionDescription(answer), () => {
         this.localPC = this.calleePC;
         this.sendChat({
            type: "answer",
            from: this.setting.username,
            to: offer.from,
            answer: answer
         });
      });
   }, err=> {
      console.log(err);
   });

}, err=> {
```

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

};

onAnswer = (answer) => {
   let videos = this.setting.getVideos();
   this.remoteVideoIndex = videos.length;
   this.remoteVideoId = 'remoteVideo' + this.
      remoteVideoIndex;
   videos.push({videoId: this.remoteVideoId, isMuted: '',
      isLoaded: false, source: null});
   this.setting.setVideos(videos);

   // 3. This is the third step, get answer
   this.callerPC.setRemoteDescription(new
      RTCSessionDescription(answer.answer), () => {
      this.printTimestamp("Answer␣received");
   }, err => {
      console.log(err);
   });

};

onCandidate = (candidate) => {
   this.printTimestamp("Candidate␣received");
   if (candidate)
      this.localPC.addIceCandidate(new RTCIceCandidate(
         candidate));
};

onChat = (data) => {
   this.setting.addChat(data.username + ":␣" + data.msg);
};

sendChat = (data) => {
   this.connection.sendChat(data);
};

sendGroupChat = (data) => {
   this.connection.sendGroupChat(data);
};

printTimestamp = (value)=> {
```

```
      this.setting.addChat((Date.now() - this.setting.
          currentTime) / 1000 + "␣s:␣" + value);
   };

   resetTimestamp = ()=> {
      this.setting.currentTime = Date.now();
      this.setting.addChat("Timestamp␣is␣reset");
   };
}
```

A.2.2 DirectConnectionService

```
import {Injectable} from '@angular/core';
import {Http, Headers} from '@angular/http';
import 'rxjs/Rx';
import 'webrtc-adapter';
import {SettingService} from "./setting";

declare let Strophe:any;
declare let $pres:any;
declare let $:any;
declare let uuid:any;
declare let sizeof:any;

@Injectable()
export class DirectConnectionService {

   private connection:any;
   private signal:any;

   constructor(private http:Http, private setting:
       SettingService) {
      this.http = http;
      this.setting = setting;
   }

   registerCallback = (status) => {
      if (status === Strophe.Status.REGISTER) {
         // fill out the fields
         this.setting.username = uuid.v4().split("-").join("");
         this.connection.register.fields.username = this.
            setting.username;
         this.connection.register.fields.password = this.
            setting.password;
         // calling submit will continue the registration
            process
         this.connection.register.submit();
      } else if (status === Strophe.Status.REGISTERED) {
         console.log("registered!");
         // calling login will authenticate the registered JID.
         this.connection.authenticate();
      } else if (status === Strophe.Status.CONFLICT) {
         console.log("Contact already existed!");
      } else if (status === Strophe.Status.NOTACCEPTABLE) {
         console.log("Registration form not properly filled
            out.")
```

```
    } else if (status === Strophe.Status.REGIFAIL) {
       console.log("The␣Server␣does␣not␣support␣In-Band␣
          Registration")
    } else if (status === Strophe.Status.CONNECTED) {
       console.log("Connected");
       this.createRoom();
       // do something after successful authentication
    } else {
       console.log("Do␣nothing");
       // Do other stuff
    }
};

register = (signal) => {
    this.signal = signal;
    this.connection = new Strophe.Connection("https://f2f.
       chat:5280/http-bind");
    this.connection.register.connect("f2f.chat", this.
       registerCallback);

};

msg_handler_cb = (msg)=> {
    // console.log(msg);
    let from = $(msg).attr("from");
    let fromUser = from.split('/')[1];
    let to = $(msg).attr("to");
    let toUser = from.split('@')[0];
    if (fromUser == this.setting.username || fromUser ==
       toUser)
       return true;
    this.setting.currentSize += sizeof(msg);
    this.setting.addChat("==================");
    this.setting.addChat(this.setting.currentSize);
    let data = JSON.parse($(msg).find("body").text());
    this.setting.addChat(data.type);
    this.setting.addChat("==================");
    // console.log(data.type);
    this.signal.swicher(data);
    return true;
};

createRoom = ()=> {
    this.connection.muc.init(this.connection);
    this.connection.muc.join(
```

```
            this.setting.roomName + "@conference.f2f.chat",
            this.setting.username,
            this.msg_handler_cb,
            this.pres_handler_cb,
            null,
            null,
            {maxstanzas: 0});

         this.sendGroupChat({
            type: 'connect',
            from: this.setting.username
         });
         this.signal.swicher({
            type: 'start'
         });
      };


      sendChat = (data) => {
         // console.log("send" + JSON.stringify(data));
         // this.connection.muc.message(data.to + "@f2f.chat",
             null, JSON.stringify(data), null, "chat", null);
         this.sendGroupChat(data);
      };

      sendGroupChat = (data) => {
         console.log(JSON.stringify(data));
         // console.log("send" + JSON.stringify(data));
         this.connection.muc.groupchat(this.setting.roomName + "
             @conference.f2f.chat", JSON.stringify(data), null,
             null);
      };
   }
```

A.2.3 IndirectConnectionService

```
import {Injectable} from '@angular/core';
import {Http} from '@angular/http';
import 'rxjs/Rx';
import 'webrtc-adapter';
import {SettingService} from "./setting";

declare let Strophe:any;
declare let SockJS:any;
declare let Stomp:any;
declare let sizeof:any;

@Injectable()
export class IndirectConnectionService {

   private stompClient:any;
   private signal:any;

   constructor(private http:Http, private setting:
       SettingService) {
      this.http = http;
      this.setting = setting;
   }

   register = (signal) => {
      this.signal = signal;
      this.http.get(this.setting.serverUrl + "/register")
         .subscribe(data => {
            if (data != null) {
               // console.log(data.text());
               this.setting.username = data.text();
               // Cookie.set("webrtc_username", this.setting.
                   username);
               this.connect(signal);
            }
            else {
               console.log("Error")
            }
         });
   }

   connect = (signal)=> {
      this.signal = signal;
      let socket = new SockJS(this.setting.serverUrl + '/call'
          );
```

```
      this.stompClient = Stomp.over(socket);
      this.stompClient.debug = null;
      this.stompClient.connect({}, () => {
         this.stompClient.subscribe('/receive/call/' + this.
             setting.roomName + '/' + this.setting.username,
             msg => {
            this.setting.currentSize += sizeof(msg);
            this.setting.addChat("==================");
            this.setting.addChat(this.setting.currentSize);
            let data = JSON.parse(msg.body);
            this.setting.addChat(data.type);
            this.setting.addChat("==================");
            this.signal.swicher(data);
         }, err => {
            console.log(err);
         });

         this.sendGroupChat({
            type: 'connect'
         });
      });
   };

   sendChat = (data) => {
      this.stompClient.send("/app/call/" + this.setting.
          roomName + "/" + this.setting.username, {}, JSON.
          stringify(data));
   };

   sendGroupChat = (data)=> {
      this.stompClient.send("/app/call/" + this.setting.
          roomName + "/" + this.setting.username, {}, JSON.
          stringify(data));
   };
}
```

A.2.4 SettingService

```
import {Injectable, EventEmitter, Output} from '@angular/core'
    ;

@Injectable()
export class SettingService {

   videosUpdated:EventEmitter<any> = new EventEmitter();
   chatsUpdated:EventEmitter<any> = new EventEmitter();

   private videos = [];
   private chats = [];

   public roomName:string = window.location.pathname.split('/'
       )[1];
   public username:string;
   public password:string = "12345";
   public serverUrl:string;
   public ejabberdUrl:string;
   public peerConnectionConfig = {
      // "iceServers": [{"urls": "stun:stun.l.google.com
          :19302"}]
      "iceServers": [{"urls": "stun:chun.no:3479"}, {"urls": "
          stun:stun.l.google.com:19302"}]
   };
   public localStream:any;
   public currentTime;
   public currentSize:number;


   getVideos() {
      return this.videos;
   }

   setVideos(value) {
      this.videos = value;
      this.videosUpdated.emit(this.videos);
   }

   getChats() {
      return this.chats;
   }

   addChat(value) {
      this.chats.reverse();
```

```
        this.chats.push(value);
        this.chatsUpdated.emit(this.chats);
    }
}
```

Chapter B
Back End Implementation

B.1 Controller Layer

B.1.1 CallController

```
package com.aprilchun.test.controller;

import com.aprilchun.test.service.CallService;
import org.jivesoftware.smack.AbstractXMPPConnection;
import org.jivesoftware.smack.ConnectionConfiguration;
import org.jivesoftware.smack.tcp.XMPPTCPConnection;
import org.jivesoftware.smack.tcp.
    XMPPTCPConnectionConfiguration;
import org.jivesoftware.smackx.muc.MultiUserChat;
import org.jivesoftware.smackx.muc.MultiUserChatManager;
import org.jivesoftware.smackx.xdata.Form;
import org.jivesoftware.smackx.xdata.packet.DataForm;
import org.json.JSONObject;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Scope;
import org.springframework.messaging.handler.annotation.
    DestinationVariable;
import org.springframework.messaging.handler.annotation.
    MessageMapping;
import org.springframework.messaging.handler.annotation.SendTo
    ;
import org.springframework.messaging.simp.
    SimpMessageHeaderAccessor;
import org.springframework.messaging.simp.
    SimpMessagingTemplate;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.*;
```

```
import javax.servlet.http.HttpServletRequest;
import java.util.Map;
import java.util.UUID;

@Scope("prototype")
@Controller
public class CallController {
   @Autowired
   private CallService callService;
   @Autowired
   private SimpMessagingTemplate template;

   @RequestMapping(value = "")
   public String index(Model model) {
      return "index";
   }

   @RequestMapping(value = "/{urlPath}")
   public String webrtc(@PathVariable("urlPath") String
       urlPath, HttpServletRequest request) {
      //return "webrtc";
      return "forward:/static/client/index.html";
   }

   @ResponseBody
   @RequestMapping(value = "/register", method = RequestMethod.
       GET, produces = "text/plain;_charset=utf-8")
   public String register() {
      String username = UUID.randomUUID().toString().replaceAll
          ("-", "");
      return this.callService.register(username);
   }

   @MessageMapping("/call/{roomName}/{username}")
   @SendTo("/receive/call/{roomName}/{username}")
   public String call(@DestinationVariable String roomName,
       @DestinationVariable String username,
       SimpMessageHeaderAccessor headerAccessor, String message)
        {
      try {
         Map session = headerAccessor.getSessionAttributes();
         XMPPTCPConnection connection;
         MultiUserChat muc;
         JSONObject messageJSON = new JSONObject(message);
         switch (messageJSON.getString("type")) {
```

```
            case "connect":
               connection = this.callService.login(username);
               session.put("connection", connection);
               muc = this.callService.createOrJoin(roomName,
                  username, connection, this.template);
               muc.sendMessage(message);
               session.put("muc", muc);
               messageJSON.put("type", "start");
               return messageJSON.toString();
            default:
               muc = (MultiUserChat) session.get("muc");
               muc.sendMessage(message);
               return null;
         }
      } catch (Exception e) {
         return null;
      }

   }
}
```

B.2 Service Layer

B.2.1 CallService

```
package com.aprilchun.test.service;

import com.aprilchun.test.websocket.CallMessageListener;
import com.aprilchun.test.util.Constant;
import org.jivesoftware.smack.ConnectionConfiguration;
import org.jivesoftware.smack.SASLAuthentication;
import org.jivesoftware.smack.SmackConfiguration;
import org.jivesoftware.smack.packet.Presence;
import org.jivesoftware.smack.tcp.XMPPTCPConnection;
import org.jivesoftware.smack.tcp.
    XMPPTCPConnectionConfiguration;
import org.jivesoftware.smackx.iqregister.AccountManager;
import org.jivesoftware.smackx.muc.DiscussionHistory;
import org.jivesoftware.smackx.muc.MultiUserChat;
import org.jivesoftware.smackx.muc.MultiUserChatManager;
import org.json.JSONObject;
import org.springframework.context.annotation.Scope;
import org.springframework.messaging.simp.
    SimpMessagingTemplate;
import org.springframework.stereotype.Service;

import java.util.Calendar;

@Scope("prototype")
@Service("callService")
public class CallService {

   XMPPTCPConnection connection;
   XMPPTCPConnectionConfiguration config =
       XMPPTCPConnectionConfiguration.builder()
          .setServiceName(Constant.EJABBERD_SERVER_NAME)
          .setHost(Constant.EJABBERD_HOST)
          .setPort(Constant.EJABBERD_PORT)
          .setSecurityMode(ConnectionConfiguration.SecurityMode.
             disabled)
          .setDebuggerEnabled(false)
          .build();
   MultiUserChat muc;

   public String register(String username) {
      try {
         this.openConnection();
```

```
            AccountManager accountManager = AccountManager.
                getInstance(connection);
            accountManager.createAccount(username, Constant.
                DEFAULT_PASSWORD);

            return username;
        } catch (Exception e) {
            return null;
        }
    }

    public XMPPTCPConnection login(String username) throws
        Exception {
        this.openConnection();
        connection.login(username, Constant.DEFAULT_PASSWORD);
        Presence presence = new Presence(Presence.Type.available)
            ;
        connection.sendStanza(presence);

        return connection;
    }

    public MultiUserChat createOrJoin(String roomName, String
        username, XMPPTCPConnection connection,
        SimpMessagingTemplate template) throws Exception {
        MultiUserChatManager manager = MultiUserChatManager.
            getInstanceFor(connection);
        this.muc = manager.getMultiUserChat(roomName + "
            @conference." + Constant.EJABBERD_SERVER_NAME);
        if (!this.muc.isJoined()) {
            DiscussionHistory history = new DiscussionHistory();
            history.setMaxStanzas(0);
            this.muc.createOrJoin(username, null, history,
                SmackConfiguration.getDefaultPacketReplyTimeout());
        }
        this.muc.addMessageListener(new CallMessageListener(
            roomName, username, template));

        return this.muc;
    }

    public String init() throws Exception {
        JSONObject resultJSON = new JSONObject();
        resultJSON.put("type", "init");
```

```
      return resultJSON.toString();
   }

   public void send(String msg) throws Exception {
      this.muc.sendMessage(msg);
   }

   public XMPPTCPConnection openConnection() throws Exception
       {
      if (this.connection == null) {
         this.connection = new XMPPTCPConnection(this.config);

      }
      if (!this.connection.isConnected()) {
         this.connection.connect();
      }
      return this.connection;
   }

   public void closeConnection() {
      this.connection.disconnect();
   }
}
```

B.3 WebSocket Layer

B.3.1 WebSocketConfig

```
package com.aprilchun.test.websocket;

import org.springframework.context.annotation.Configuration;
import org.springframework.messaging.simp.config.
    MessageBrokerRegistry;
import org.springframework.web.socket.config.annotation.*;
import org.springframework.web.socket.server.standard.
    TomcatRequestUpgradeStrategy;
import org.springframework.web.socket.server.support.
    DefaultHandshakeHandler;

@Configuration
@EnableWebSocket
@EnableWebSocketMessageBroker
public class WebSocketConfig extends
    AbstractWebSocketMessageBrokerConfigurer {

   @Override
   public void configureMessageBroker(MessageBrokerRegistry
       config) {
      config.enableSimpleBroker("/topic");
      config.enableSimpleBroker("/receive");
      config.setApplicationDestinationPrefixes("/app");
   }

   @Override
   public void registerStompEndpoints(StompEndpointRegistry
       registry) {
      registry.addEndpoint("/call")
            .setAllowedOrigins("*")
            .setHandshakeHandler(new DefaultHandshakeHandler(
               new TomcatRequestUpgradeStrategy()))
            .withSockJS();
      registry.addEndpoint("/message")
            .setAllowedOrigins("*")
            .setHandshakeHandler(new DefaultHandshakeHandler(
               new TomcatRequestUpgradeStrategy()))
            .withSockJS();
   }
}
```

B.3.2 CallMessageListener

```
package com.aprilchun.test.websocket;

import org.jivesoftware.smack.MessageListener;
import org.jivesoftware.smack.packet.Message;
import org.jivesoftware.smackx.muc.MultiUserChat;
import org.springframework.context.annotation.Scope;
import org.springframework.messaging.handler.annotation.
   DestinationVariable;
import org.springframework.messaging.simp.
   SimpMessagingTemplate;
import org.springframework.stereotype.Service;

public class CallMessageListener implements MessageListener
    {
   private String roomName, username;
   private SimpMessagingTemplate template;

   public CallMessageListener(String roomName, String username
       , SimpMessagingTemplate template) {
      this.roomName = roomName;
      this.username = username;
      this.template = template;
   }

   @Override
   public void processMessage(Message message) {
      String from = message.getFrom();
      String to = message.getTo();
      String body = message.getBody();
      try {
         if (!from.split("/")[1].equals(username))
         this.sendMessage(this.roomName, this.username, body);

      } catch (Exception e) {
         e.printStackTrace();
      }

   }

   public void sendMessage(@DestinationVariable String
       roomName, @DestinationVariable String username, String
       msg) throws Exception {
      this.template.convertAndSend("/receive/call/" + roomName
          + "/" + username, msg);
```

```
    }
}
```

B.3.3 CallWebSocketHandler

```
package com.aprilchun.test.websocket;

import org.springframework.messaging.simp.stomp.*;
import org.springframework.web.socket.server.standard.
    TomcatRequestUpgradeStrategy;
import org.springframework.web.socket.server.support.
    DefaultHandshakeHandler;

import java.lang.reflect.Type;

public class CallWebSocketHandler extends
    DefaultHandshakeHandler implements StompSessionHandler {
   public CallWebSocketHandler(TomcatRequestUpgradeStrategy
       tomcatRequestUpgradeStrategy) {
      super(tomcatRequestUpgradeStrategy);
   }

   @Override
   public void afterConnected(StompSession stompSession,
       StompHeaders stompHeaders) {

   }

   @Override
   public void handleException(StompSession stompSession,
       StompCommand stompCommand, StompHeaders stompHeaders,
       byte[] bytes, Throwable throwable) {
      if (throwable instanceof ConnectionLostException) {
         // if connection lost, call this
      }
   }

   @Override
   public void handleTransportError(StompSession stompSession,
        Throwable throwable) {

   }

   @Override
   public Type getPayloadType(StompHeaders stompHeaders) {
      return null;
   }

   @Override
```

```
    public void handleFrame(StompHeaders stompHeaders, Object
        o) {

    }
}
```

Chapter C
Collected Data

Table C.1: Data from Experiment 1 with Direct Connection to XMPP

Index	Total Time Cost (sec)	Total Data Size (byte)	Signalling Time Cost (sec)	Answer Size (byte)	Number of Generated Candidate	Number of Received Candidate	Candidate Size (byte)
1	11.702	717056	11.311	80232	17	4	159206
2	11.538	717006	11.118	80182	17	4	159206
3	11.492	716836	11.078	80220	17	4	159154
4	11.715	717056	11.279	80232	17	4	159206
5	11.649	716946	11.577	80226	17	4	159180
6	11.835	717056	11.763	80232	17	4	159206
7	12.132	716796	11.721	80180	17	4	159154
8	12.302	717016	11.864	80192	17	4	159206
9	12.405	717066	11.991	80242	17	4	159206
10	12.624	716896	12.208	80176	17	4	159180
Average	11.939	716973	11.591	80211.400	17	4	159190.400

Table C.2: Data from Experiment 1 with Indirect Connection to XMPP

Index	Total Time Cost (sec)	Total Data Size (byte)	Signalling Time Cost (sec)	Answer Size (byte)	Number of Generated Candidate	Number of Received Candidate	Candidate Size (byte)
1	7.306	12074	4.436	7580	17	4	1123.5
2	7.254	12064	4.374	7572	17	4	1123
3	7.241	12074	4.388	7580	17	4	1123.5
4	7.278	12074	4.434	7582	17	4	1123
5	7.298	12064	4.461	7572	17	4	1123
6	7.327	12064	4.488	7570	17	4	1123.5
7	7.363	12074	4.523	7582	17	4	1123
8	5.978	12056	4.569	7564	17	4	1123
9	7.462	12064	4.612	7572	17	4	1123
10	8.966	12060	6.132	7568	17	4	1123
Average	7.347	12066.800	4.641	7574.200	17	4	1123.150

Table C.3: Data from Experiment 2 with Direct Connection to XMPP

Index	Total Time Cost (sec)	Total Data Size (byte)	Signalling Time Cost (sec)	Answer Size (byte)	Number of Generated Candidate	Number of Received Candidate	Candidate Size (byte)
1	2.483	711662	2.033	79830	5	4	157958
2	2.489	711432	2.059	79808	5	4	157906
3	1.902	711702	1.556	79870	5	4	157958
4	2.298	711622	1.952	79790	5	4	157958
5	2.483	711592	2.096	79864	5	4	157932
6	1.915	711702	1.582	79870	5	4	157958
7	1.913	711642	1.696	79810	5	4	157958
8	2.606	711502	2.233	79774	5	4	157932
9	2.354	711552	2.194	79720	5	4	157958
10	2.48	711592	2.107	79864	5	4	157932
Average	2.292	711600	1.951	79820	5	4	157945

Table C.4: Data from Experiment 2 with Indirect Connection to XMPP

Index	Total Time Cost (sec)	Total Data Size (byte)	Signalling Time Cost (sec)	Answer Size (byte)	Number of Generated Candidate	Number of Received Candidate	Candidate Size (byte)
1	5.859	12056	4.45	7562	6	4	1123.5
2	5.939	12030	4.392	7536	6	4	1123.5
3	5.988	12056	4.624	7562	6	4	1123.5
4	5.896	12048	4.418	7554	6	4	1123.5
5	5.986	12036	4.46	7542	6	4	1123.5
6	5.944	12056	4.456	7562	6	4	1123.5
7	5.956	12046	4.521	7552	6	4	1123.5
8	7.564	12038	6.166	7544	6	4	1123.5
9	5.982	12056	4.518	7562	6	4	1123.5
10	5.982	12056	4.518	7562	6	4	1123.5
Average	6.110	12047.800	4.652	7553.800	6	4	1123.500

Table C.5: Data from Experiment 3 with Direct Connection to XMPP

Index	Total Time Cost (sec)	Total Data Size (byte)	Signalling Time Cost (sec)	Answer Size (byte)	Number of Generated Candidate	Number of Received Candidate	Candidate Size (byte)
1	7.184	447978	5.328	81090	8	3	122296
2	6.716	448018	6.271	81130	8	3	122296
3	5.047	447948	4.505	81060	8	3	122296
4	5.257	447952	4.738	81124	8	3	122276
5	5.134	447928	4.685	81040	8	3	122296
6	5.650	447978	5.130	81090	8	3	122296
7	5.325	447912	4.802	81084	8	3	122276
8	5.126	447952	4.675	81124	8	3	122276
9	5.244	447848	4.778	80960	8	3	122296
10	5.849	447938	5.383	81050	8	3	122296
Average	5.653	447945.200	5.029	81075.200	8	3	122290

Table C.6: Data from Experiment 3 with Indirect Connection to XMPP

Index	Total Time Cost (sec)	Total Data Size (byte)	Signalling Time Cost (sec)	Answer Size (byte)	Number of Generated Candidate	Number of Received Candidate	Candidate Size (byte)
1	7.591	11436	6.158	7810	6	4	906.5
2	7.591	11446	6.133	7812	6	4	908.5
3	7.460	11438	6.007	7804	6	4	908.5
4	7.460	11448	5.996	7814	6	4	908.5
5	7.473	11428	6.020	7794	6	4	908.5
6	6.009	11448	4.549	7814	6	4	908.5
7	7.506	11430	6.054	8714	6	4	679
8	7.016	11432	5.577	7798	6	4	908.5
9	8.096	11440	6.117	8724	6	4	679
10	7.769	11430	5.837	8714	6	4	679
Average	7.397	11437.600	5.845	8079.800	6	4	839.450