NTNU – Trondheim
Norwegian University of
Science and Technology

Unified Communication and WebRTC

Xiao Chen

Submission date: May 2014
Responsible professor: Mazen Malek Shiaa, ITEM
Supervisor: Mazen Malek Shiaa, ITEM

Norwegian University of Science and Technology
Department of Telematics

Title: Unified Communication and WebRTC
Student: Xiao Chen

Problem description:

Web Real-Time Communication (WebRTC) offers application developers the ability to write rich, real-time multimedia application (e.g. video chat) on the web, without requiring any plugins, downloads or installations. WebRTC is also currently the only existing soon-to-be standardized technology on the market to create horizontal cross-platform communication services, encompassing smartphones, tablets, PCs, laptops and TVs, which adds value for both consumers and enterprises. WebRTC gives operators the opportunity to offer telephony services to more devices, such as PCs, tablets and TVs. This thesis considers how WebRTC can enhance the existing echo-systems for telephony and messaging services by providing the end-user rich application client.

It will also covers research about different solutions to implement WebRTC to cooperate with existing telephony services like hosted virtual Private Branch Exchange (PBX) services.

A prototype of WebRTC deployment based on different rich communication scenarios will be implemented along with this thesis. Some corresponding test and evaluation will be fulfilled in this prototype.

Research about advanced WebRTC usability in telephony and messaging services will be covered in this thesis by the feedback of the WebRTC prototype

Responsible professor: Mazen Malek Shiaa, ITEM
Supervisor: Mazen Malek Shiaa, ITEM

Abstract

For the development of traditional telephony echo-systems, the cost of maintenance traditional telephony network is getting higher and higher but the number of customer does not grow rapidly any more since almost every one has a phone to access the traditional telephony network. WebRTC is an Application Programming Interface (API) definition drafted by the World Wide Web Consortium (W3C) that supports browser-to-browser applications for voice calling, video chat, and Peer-To-Peer (P2P) file sharing without plugins.[Wik14y] "This technology, along with other advances in HyperText Markup Language 5 (HTML5) browsers, has the potential to revolutionize the way we all communicate, in both personal and business spheres."[JB13a]

As network operators aspect, WebRTC provides many opportunities to the future telecommunication business module. For the users already have mobile service, operator can offer WebRTC service with session-based charging to the existing service plans. Messaging APIs can augment WebRTC web application with Rich Communication Services (RCS) and other messaging services developers already know and implement. Furthermore, since WebRTC is a web based API, then the implementation of Quality of Service (QoS) for WebRTC can provide assurance to users and priority services (enterprise, emergency, law enforcement, eHealth) that a WebRTC service will work as well as they need it to. WebRTC almost provide network operator a complete new business market with a huge amount of end-users.

As an end-user aspect, WebRTC provides a much simpler way to have real-time conversation with another end-user. It is based on browser and internet which almost personal or enterprise computer already have, without any installation and plugins, end-user can have exactly the same service which previous stand-alone desktop client provides. By the system this thesis will cover, the end-user even can have the real-time rich communication service with multiple kinds of end-users.

This thesis will cover the research about how to apply WebRTC technology with existing legacy Voice over Internet Protocol (VoIP) network.

Keywords : WebRTC, AngularJs, Nodejs, SIP, WebSocket, Dialogic XMS

Preface

WebRTC is quite popular topic in the web development filed since the massive usage and development of HTML5 web application on the internet. The initial purpose of this web API is to provide the browser client the ability to create real-time conversation between each other. After many WebRTC based application come out the market, it is quite normal to think about how to integrate these kind of web application with the current legacy telephony network as the next big step for this technology. The requirement of this process is not only from the traditional telephony operator but also the normal end-users. The approach to achieve this goal is the man purpose of this thesis.

Research about current WebRTC technology usage and development of a WebRTC prototype system are the two main parts of this thesis. The prototype system is implemented by regarding to the research of WebRTC integrated with legacy telephony network.

Current status of WebRTC technology, WebRTC business use cases, analysis of different possible WebRTC implement solutions and WebRTC system architecture will be covered in this thesis. Some research regarding with the development of WebRTC prototype system will be covered in this thesis as well.

The prototype described in this thesis is implemented to cooperate with existing legacy VoIP network services through Session Initiation Protocol (SIP) server and PBX[1] service. It will provide most of essential functions which are included in the legacy telephony business, besides other communication functions used on web. Moreover, some analysis and discussion about the feedback of the prototype will be covered in this thesis.

The prototype will be implemented in programming language Javascript for both client font-end and server back-end by using the AngularJs framework and Nodejs framework mainly. The approach and reason to choose these framework and programming language will be expounded in the later chapter in this thesis.

[1]Users of the PBX share a certain number of outside lines for making telephone calls external to the PBX.[Web14c]

Acknowledgment

Written by Xiao Chen in Trondheim in May 2014

Thanks for Mazen Malek Shiaa, ITEM

Frank Mbaabu Kiriinya, Gintel AS

Roman Stobnicki, Dialogic, the Network Fuel company

Special thanks for Gintel AS

Contents

List of Figures

List of Tables

List of Code Snippets

List of Acronyms

AJAX Asynchronous JavaScript and XML.

API Application Programming Interface.

CSS Cascading Style Sheets.

DOM Document Object Model.

DTLS Datagram Transport Layer Security.

EJS Embedded JavaScript templates.

GIPS Global IP Solutions.

HTML HyperText Markup Language.

HTML5 HyperText Markup Language 5.

HTTP Hypertext Transfer Protocol.

HTTPS Hypertext Transfer Protocol over Secure Socket Layer.

ICE Interactive Connectivity Establishment.

IETF Internet Engineering Task Force.

IMS IP Multimedia Subsystem.

IO Input/Output.

IP Internet Protocol.

JAIN Java APIs for Integrated Networks.

JEE Joint Entrance Examination.

JSLEE JAIN Service Logic Execution Environment.

JSON JavaScript Object Notation.

JSONP JSON with padding.

JSR Java Specification Requests.

MPBX Multimedia Private Branch Exchange.

MVC Model–View–Controller.

NAT Network Address Translator.

NIO Non-Blocking I/O.

OAuth Open standard for Authorization.

P2P Peer-To-Peer.

PBX Private Branch Exchange.

PHP PHP: Hypertext Preprocessor.

PSTN Public Switched Telephone Network.

QoS Quality of Service.

RCS Rich Communication Services.

RTC Real-Time Communication.

RTP Real-time Transport Protocol.

SDP Session Description Protocol.

SIP Session Initiation Protocol.

SLEE Service Logic Execution Environment.

SMS Short Message Service.

SRTP Secure Real-time Transport Protocol.

SSL Secure Sockets Layer.

STUN Session Traversal Utilities for NAT.

TCP Transmission Control Protocol.

TLS Transport Layer Security.

TOR The Onion Router.

TURN Traversal Using Relays around NAT.

UA User Agent.

UAC User Agent Client.

UAS User Agent Server.

UDP User Datagram Protocol.

UI User Interface.

URI Uniform Resource Identifier.

URL Uniform Resource Locator.

VM Virtual Machine.

VoIP Voice over Internet Protocol.

W3C World Wide Web Consortium.

WebRTC Web Real-Time Communication.

XMPP Extensible Messaging and Presence Protocol.

Chapter 1
Introduction

In this Chapter, introduction of WebRTC and SIP network will be covered. SIP is one of the VoIP signaling protocols widely used in current internet telephony service which is also the target telephony network integrated with WebRTC application system in this thesis.

1.1 WebRTC

Gmail[1] video chat became popular in 2008, and in 2011 Google introduced Hangouts[2], which use the Google Talk service (as does Gmail). Google bought Global IP Solutions (GIPS), a company which had developed many components required for Real-Time Communication (RTC), such as codecs and echo cancellation techniques. Google open sourced the technologies developed by GIPS and engaged with relevant standards bodies at the Internet Engineering Task Force (IETF) and W3C to ensure industry consensus. In May 2011, Ericsson built the first implementation of WebRTC.

1.1.1 What is WebRTC ?

WebRTC is an industry and standards effort to put real-time communications capabilities into all browsers and make these capabilities accessible to web developers via standard HTML5 tags and JavaScript APIs. For example, consider functionality similar to that offered by Skype[3]. but without having to install any software or plug-ins. For a website or web application to work regardless of which browser is used, standards are required. Also, standards are required so that browsers can

[1] Gmail is a free , advertising-supported email service provided by Google.

[2] Google Hangouts is an instant messaging and video chat platform developed by Google, which launched on May 15, 2013 during the keynote of its I/O development conference. It replaces three messaging products that Google had implemented concurrently within its services, including Talk, Google+ Messenger, and Hangouts, a video chat system present within Google+.

[3] Skype is a freemium voice-over-IP service and instant messaging client, currently developed by the Microsoft Skype Division.[Wik14v]

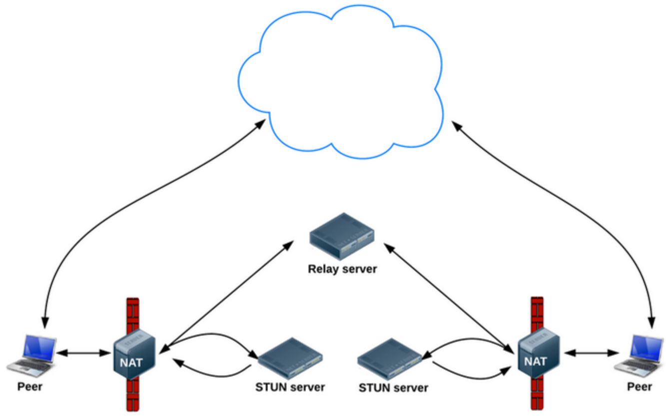

Figure 1.1: WebRTC Network: Finding connection candidates[Dut14]

communicate with non-browsers, including enterprise and service provider telephony and communications equipment[JB13d].

With the rapidly development of internet, more and more communication traffic is moving to web from the traditional telephony network. And in the recent decade, VoIP network services are growing to the peek of the market capacity. Solution to integrate WebRTC and existing VoIP network is the right approach the trend of the internet communication requirement.

1.1.2 WebRTC Network Structure

In the Figure1.1[Dut14] showing how the Interactive Connectivity Establishment (ICE) framework[4] to find peer candidate through Session Traversal Utilities for NAT (STUN) server and its extension Traversal Using Relays around NAT (TURN) server.

Initially, ICE tries to connect peers directly, with the lowest possible latency, via User Datagram Protocol (UDP). In this process, STUN servers have a single task: to enable a peer behind a Network Address Translator (NAT) to find out its public address and port. If UDP fails, ICE tries Transmission Control Protocol (TCP): first Hypertext Transfer Protocol (HTTP), then Hypertext Transfer Protocol over Secure Socket Layer (HTTPS). If direct connection fails—in particular, because of enterprise NAT traversal and firewalls—ICE uses an intermediary (relay) TURN server. In other words, ICE will first use STUN with UDP to directly connect peers and, if that fails, will fall back to a TURN relay server. The expression 'finding candidates' refers to the process of finding network interfaces and ports.[Dut14]

[4]ICE is a framework for connecting peers, such as two video chat clients.[Wik14l]

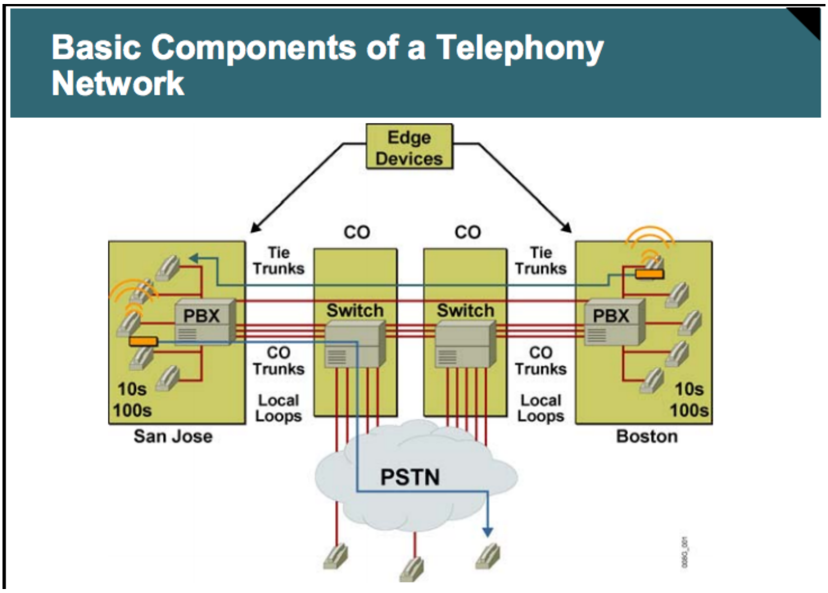

Figure 1.2: Traditional Telephony Network

The difference and usage of STUN server and TURN server will be discussed more detail in Chapter 5.

WebRTC needs server to help users discover each other and exchange 'real world' details such as names. Then WebRTC client applications (peers) exchange network information. After that, peers exchange data about media such as video format and resolution. Finally, WebRTC client applications can traverse NAT gateways and firewalls.

Compare to the traditional telephony network which is shown in Figure1.2[Inc05], the main difference between these two communication network is that WebRTC is P2P communication in STUN server scenario, after the signaling between end-peers, the media data are exchanged directly between tow peers. However, in the traditional telephony, all the media data are transferred to PBX and switches regarding to Public Switched Telephone Network (PSTN)[5] then reach the other side of the peer. Even in TURN server scenario for WebRTC, the media stream is only relaying to the TURN then directly transfer to another peer, no switches involved.

[5]The PSTN consists of telephone lines, fiber optic cables, microwave transmission links, cellular networks, communications satellites, and undersea telephone cables, all interconnected by switching centers, thus allowing any telephone in the world to communicate with any other. Originally a network of fixed-line analog telephone systems, the PSTN is now almost entirely digital in its core network and includes mobile and other networks, as well as fixed telephones.[Wik14q]

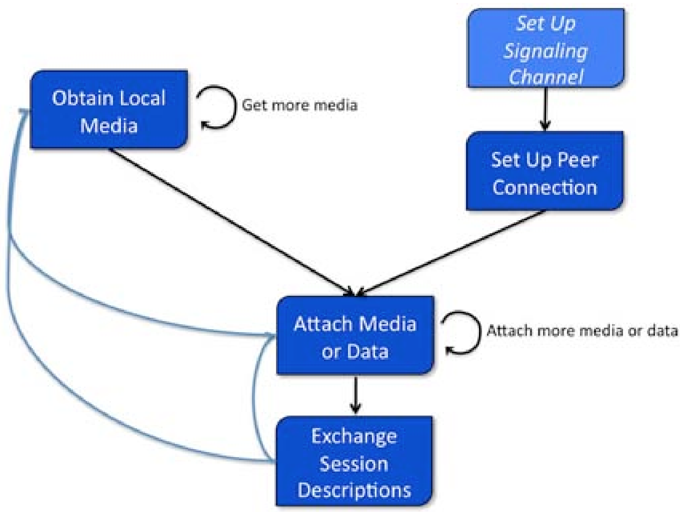

Figure 1.3: WebRTC API View with Signaling[JB13b]

1.1.3 WebRTC Implementation Steps

There are four main steps to implement a WebRTC session shown in Figure 1.3. The browser client need to obtain local media first, then set up a connection between the browser and the other peer through some signaling, after that attach the media and data channels to the connection, afterwards exchange the session description from each other. Finally the media stream will automatically exchange through the real-time peer to peer media channel.

Each step shown in the Figure 1.3 is implemented by some WebRTC APIs. More detail about how to use WebRTC APIs to implement these steps will be covered in Chapter 4. The WebRTC architecture is shown in Figure 1.4, the main focus in this thesis will be Web API part and transport part because Web API is the tool to implement the WebRTC application and transport part is the key for WebRTC application to communicate with application server, media server and other end peer in the system.

Besides WebRTC APIs, signaling is the other important factor in the system. WebRTC uses *RTCPeerConnection* (more about this API will be discussed in Chapter 4) to communicate streaming data between browsers, but also needs a mechanism to coordinate communication and to send control messages, a process known as signaling. Signaling methods and protocols are not specified by WebRTC by Google purpose, so signaling is not part of the *RTCPeerConnection* API.

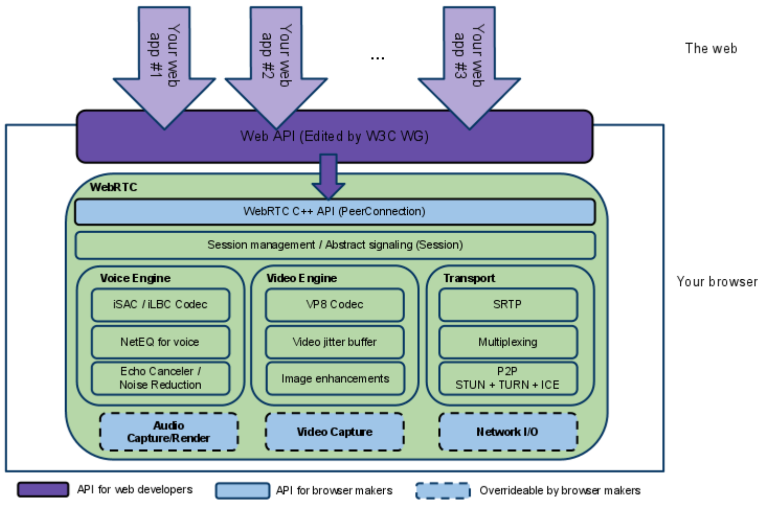

Figure 1.4: WebRTC architecture [Goo12]

Instead, WebRTC app developers can choose whatever messaging protocol they prefer, such as SIP or Extensible Messaging and Presence Protocol (XMPP), and any appropriate duplex (two-way) communication channel. The prototype application in this thesis will use WebSocket[6] as signaling between WebRTC browser end point and keep use SIP as signaling for SIP end point (mobile/fixed phone based on PSTN in this case).

Signaling is used to exchange three types of information[Dut14]:

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

The exchange of information via signaling must have completed successfully before peer-to-peer streaming can begin. For the prototype application in this thesis, the signaling has two mechanisms, one is for WebRTC browser clients and the other is for SIP clients, it will be explained in Chapter 4.

1.2 SIP

The prototype application in this thesis will be integrated with PSTN through SIP server. Therefore the application server implemented in this system will use SIP

[6] WebSocket is a protocol providing full-duplex communications channels over a single TCP connection.[Wik14z]

signaling to communicate with SIP server to handle the signaling configuration with mobile/fixed phone end-point.

1.2.1 What is SIP?

The SIP is a signaling communications protocol, widely used for controlling multimedia communication sessions such as voice and video calls over Internet Protocol (IP) networks.

The protocol defines the messages that are sent between endpoints which govern establishment, termination and other essential elements of a call. SIP can be used for creating, modifying and terminating sessions consisting of one or several media streams. SIP can be used for two-party (unicast) or multiparty (multicast) sessions. Other SIP applications include video conferencing, streaming multimedia distribution, instant messaging, presence information, file transfer, fax over IP and online games.[Wik14u]

SIP works in conjunction with several other application layer protocols that identify and carry the session media. Media identification and negotiation is achieved with the SDP. It is different key filed format than the WebRTC SDP. For the transmission of media streams (voice, video) SDP typically employs the Real-time Transport Protocol (RTP) or Secure Real-time Transport Protocol (SRTP). For secure transmissions of SIP messages, the protocol may be encrypted with Transport Layer Security (TLS).

1.2.2 SIP Network Elements

In normal SIP network, SIP defines user-agents as well as several types of server network elements. Two SIP endpoints can communicate without any intervening SIP infrastructure. However, this approach is often impractical for a public service, which needs directory services to locate available nodes on the network. In the system implemented of this thesis, the application server will play as 'User Agent', 'Registrar' and 'Gateway' elements in the network.

User Agent[Wik14u]:

A SIP User Agent (UA) is a logical network end-point used to create or receive SIP messages and thereby manage a SIP session. A SIP UA can perform the role of a User Agent Client (UAC), which sends SIP requests, and the User Agent Server (UAS), which receives the requests and returns a SIP response. These roles of UAC and UAS only last for the duration of a SIP transaction.

Registrar[Wik14u]:

A registrar is a SIP endpoint that accepts REGISTER requests and places the information it receives in those requests into a location service for the domain it handles. The location service links one or more IP addresses to the SIP Uniform Resource Identifier (URI) of the registering agent. The URI uses the sip: scheme, although other protocol schemes are possible, such as tel:. More than one user agent can register at the same URI, with the result that all registered user agents receive the calls to the URI.

Gateway[Wik14u]:

Gateways can be used to interface a SIP network to other networks, such as the PSTN, which use different protocols or technologies. In the prototype application, the application server is the gateway to interface a WebRTC WebSocket network. The working process will be covered in Chapter 4.

1.2.3 SIP messages

Since the application server in this system will be used as SIP UA and SIP Gateway, it will send SIP message request to SIP server and receive SIP message request from the SIP server.

One of the wonderful things about SIP is that it is a text-based protocol modeled on the request/response model used in HTTP. This makes it easy to debug because the messages are easy to construct and easy to see. Contrasted with H.323[7], SIP is an exceedingly simple protocol. Nevertheless, it has enough powerful features to model the behavior of a very complex traditional telephone PBX.[Wor04]

There are two different types of SIP messages: requests and responses. The first line of a request has a method, defining the nature of the request, and a Request-URI, indicating where the request should be sent.The first line of a response has a response code.

For sip requests, regarding to RFC 3261[Soc02], the application server in the system will use following SIP messages:

- **REGISTER:** Used by a UA to indicate its current IP address and the Uniform Resource Locator (URL)s for which it would like to receive calls.
- **INVITE:** Used to establish a media session between user agents.
- **ACK:** Confirms reliable message exchanges.
- **CANCEL:** Terminates a pending request.

[7]H.323 is a recommendation from the ITU Telecommunication Standardization Sector (ITU-T) that defines the protocols to provide audio-visual communication sessions on any packet network. The H.323 standard addresses call signaling and control, multimedia transport and control, and bandwidth control for point-to-point and multi-point conferences.[Wik14k]

- **BYE:** Terminates a session between two users in a conference.

The SIP response types defined in RFC 3261 will be listened by application server in the following response codes[Wik14n]:

- **100 Trying:** Extended search being performed may take a significant time so a forking proxy must send a 100 Trying response.
- **180 Ringing:** Destination user agent received INVITE, and is alerting user of call.
- **200 OK:** Indicates the request was successful.
- **400 Bad Request:** The request could not be understood due to malformed syntax.
- **401 Unauthorized:** The request requires user authentication. This response is issued by UASs and registrars.
- **408 Request Timeout:** Couldn't find the user in time. The server could not produce a response within a suitable amount of time, for example, if it could not determine the location of the user in time. The client MAY repeat the request without modifications at any later time.
- **480 Temporarily Unavailable:** Callee currently unavailable.
- **486 Busy Here:** Callee is busy.

By listening these SIP response, the application will send request to either WebRTC browser client or SIP client to play as the gateway role in the system. This gateway mechanism will be introduced in Chapter 3.

1.3 Prototype System Working Flow

The main purpose of this thesis is to make unified communication solution with WebRTC technology.

To connect with the traditional telephony network, the VoIP system bridges the PSTN and the IP network. VoIP systems employ session control and signaling protocols to control the signaling, set-up, and tear-down of calls. They transport audio streams over IP networks using special media delivery protocols that encode voice, audio, video with audio codecs, and video codecs as Digital audio by streaming media. In this prototype, SIP signaling is used because of its widely usage and current target PSTN has SIP server support.

The Figure 1.5 shows the basic working flow of the prototype system. The Web Server is the application server in the system, it mainly bridges the WebRTC browser client with other WebRTC clients and the SIP network. The SIP server bridges the SIP network and PSTN network or traditional telephony network. And also the Media Relay server relay all the media stream from different end clients, in the prototype system, it is a media server provided by Dialogic, the Network

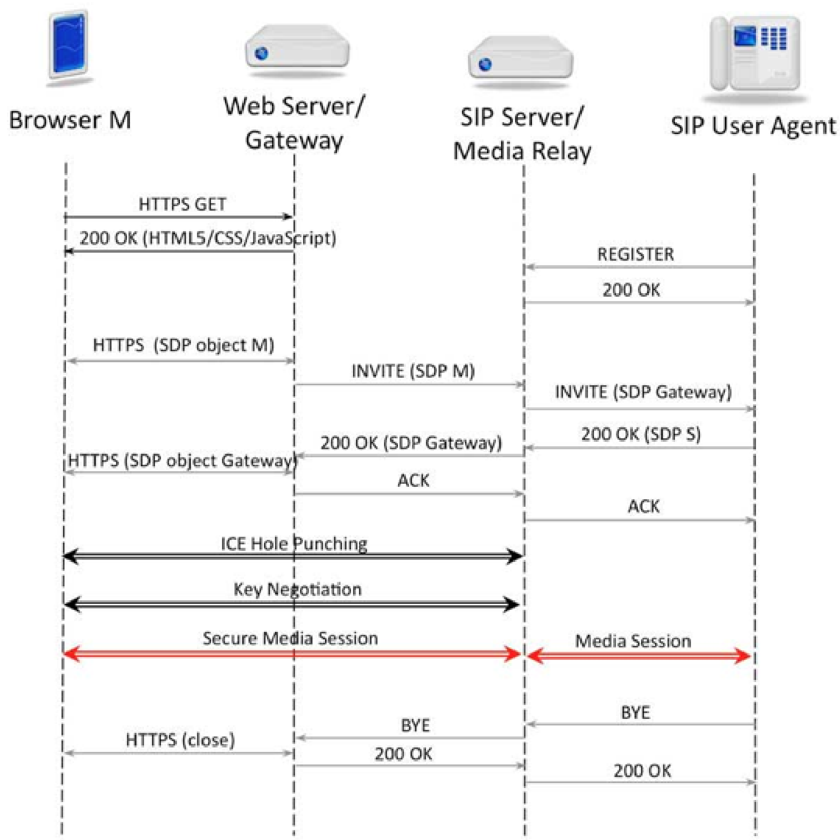

Figure 1.5: Prototype System Working Diagram [JB13c]

Fuel company, which is called PowerMedia XMS v2.1[8] PowerMedia XMS acts as a WebRTC Media Gateway to mediate WebRTC media-plane differences from those of typical existing VoIP networks including encryption interworking, transcoding, and client-based NAT traversal support. The reason to use this media server is to avoid hard-code transition between WebRTC SDP and SIP SDP. Then the end client no matter it is WebRTC client or SIP client, they will communicate with the same signaling client for their aspect.

Moreover, since the media server is used in this case, during the multiple end-point conversation, each end-point will only exchange their media stream to the single end-point on the media server (PowerMedia XMS server), it will make light client

[8]PowerMedia XMS is pre-integrated with a variety of application servers and signaling gateways with HTTP-to-SIP (H2S) functionality and rapidly integrates with others using its web API or standard interfaces.

and centralized media server control. The benefit of this system architecture will be discussed more in the Chapter 3.

Therefore, in the Figure 1.5, all the end point keep using their own original signaling protocol to communicate with different server in order to reach different scope end point.

Chapter 2
Preliminary Studies

In this chapter, some preliminary studies of WebRTC business cases and prototype working scenario will be covered. The prototype working scenario is designed by considering different WebRTC usage cases.

2.1 WebRTC Usage Cases

In May 2011, Google released an open source project for browser-based real-time communication known as WebRTC. This has been followed by ongoing work to standardise the relevant protocols in the IETF and browser APIs in the W3C. Then more and more web application are using it in different ways. There are mainly two part of the WebRTC APIs could be used separately or cooperatively in the different web application.

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

Because most of the application need to get the user's camera view and microphone sound, the *MediaStream* API is used always in real-time communication application. Normally *MediaStream* API will be used along with *RTCPeerConnection* for showing remote peer media source content. The following business usage cases, 'Tropo' and 'Uberconference', are in this category.

2.1.1 Tropo

Tropo is an application platform that enables web developers to write communication applications in the languages they already use, Groovy[1], Ruby[2], PHP: Hypertext Preprocessor (PHP)[3], Python[4] and JavaScript[5], or use a Web API which will talk with an application running on your own server through the use of HTTP and JavaScript Object Notation (JSON), feeding requests and processing responses back and forth as needed. Tropo is in the cloud, so it manages the headaches of dealing with infrastructure and keeping applications up and running at enterprise-grade. With Tropo, developers can build and deploy voice and telephony applications, or add voice to existing applications.[Cru14a]

It has some advanced features, like 'Phone numbers around the world', 'Text messaging', 'Transcription', 'Call Recording', 'Conferencing', 'Text to Speech' and 'Speech Recognition'. The prototype system in this thesis will provide similar functions like 'Text messaging' and 'Conferencing'. Since Tropo is a cloud application platform, it generates its own scripts based on programming language to provide developer possibility to easily use WebRTC to communicate with other kinds of network rather than IP network. The functions Tropo provided is implemented in application server in the prototype, the application server will handle both the SIP stack and WebRTC stack in the system. For the client scripts will be host on the same application server for browser user to access and use.

2.1.2 Uberconference

UberConference fixes all the broken and outdated aspects of traditional conference calling, making it a more productive business tool, and transforming an industry that hasn't seen real innovation in decades. UberConference gives a visual interface to every conference call so callers can know who's on a call and who's speaking at any time, in addition to making many other features, such as Hangouts[6] integration and screen sharing, easy-to-use with the click of a button. Built by the teams that brought Google Voice[7] and Yahoo! Voice to tens of millions of users, UberConference launched in 2012 and is funded by Andreessen Horowitz and Google Ventures.[Cru14b]

---

[1] Groovy is an object-oriented programming language for the Java platform. It is a dynamic language with features similar to those of Python, Ruby, Perl, and Smalltalk.[Wik14j]

[2] Ruby is a dynamic, reflective, object-oriented, general-purpose programming language. It was designed and developed in the mid-1990s by Yukihiro "Matz" Matsumoto in Japan.[Wik14s]

[3] PHP is a server-side scripting language designed for web development but also used as a general-purpose programming language.[Wik14p]

[4] Python is a widely used general-purpose, high-level programming language.[Wik14r]

[5] JavaScript (JS) is a dynamic computer programming language.[Wik14m]

[6] Google Hangouts is an instant messaging and video chat platform developed by Google, which launched on May 15, 2013 during the keynote of its I/O development conference.[Wik14h]

[7] Google Voice (formerly GrandCentral) is a telecommunications service by Google launched on March 11, 2009.[Wik14i]

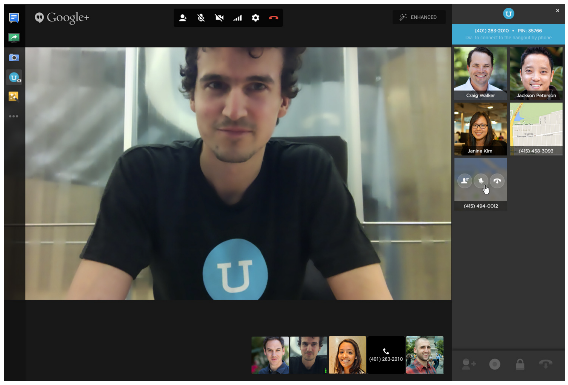

Figure 2.1: UberConference integrate with Hangouts Screen shot[Web14a]

The prototype system in this thesis is ideally to provide same rich media communication platform as the service provided by UberConference. In February of 2014, UberConference release the new feature which allow user to call into a Google Hangouts session with their mobile phone. The feature is shown in Figure 2.1, Once you have installed the UberConference app in Hangouts, people can join your call via phone with the help of a dedicated number. The prototype system will provide the same real-time communication service, but allow the user to create a video conference based on WebRTC on browser by their mobile phone number and communicate with audio only mobile phone user as well. It will be more easier for user since they just need to remember their user credential related to their mobile phone number in order to use the prototype application rather than register another service user binding with private telephone number. During the real-time conversation, the prototype application will provide user cooperation tools like instance message and file sharing in this development phase.

2.1.3 Cube Slam

However, there is another important API, *RTCDataChannel* , can be used more creatively by the developer to build web applications. The experiment usage cases, 'Cube Slam' and 'Webtorrent', are in this category which is using *RTCDataChannel* to build P2P data sharing without data going though the server to dispatch to other peers. It works more efficiently to handle the synchronization problem.

Cube Slam (shown in Figure 2.2) is a Chrome Experiment built with WebRTC , play an old-school arcade game with your friends without downloading and installing any plug-ins. Cube Slam uses *getUserMedia* to access user's webcam and microphone ,*RTCPeerConnection* to stream user video to another user, and *RTCDataChannel* to transfer the bits that keep the gameplay in sync. If two users are behind firewalls, *RTCPeerConnection* uses a TURN relay server (hosted on Google Compute Engine) to make the connection. However, when there are no firewalls in the way, the entire game happens directly peer-to-peer, reducing latency for players and server costs for developers.[Blo14]

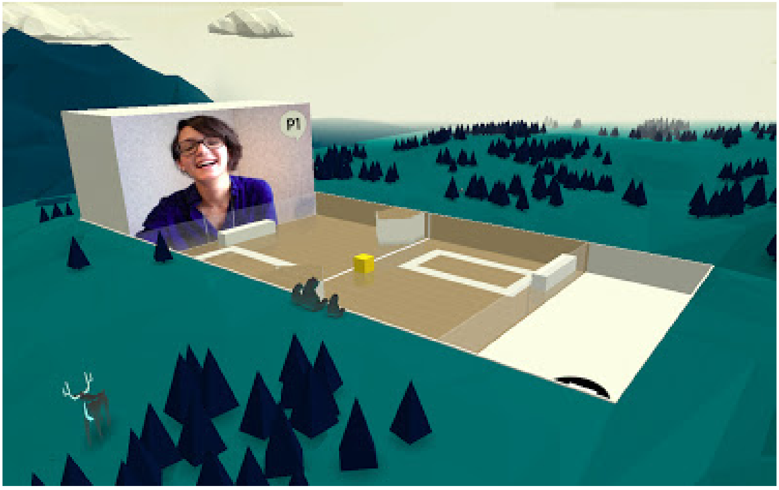

Figure 2.2: Cube Slam Game Over Screen

The idea behind the Cube Slam is that use *RTCDataChannel* to sync the player data in real-time to reduce the latency by peer to peer. *RTCDataChannel* sends data securely, and supports an "unreliable" mode for cases where you want high performance but don't care about every single packet making it across the network. In cases like games where low delay often matters more than perfect delivery, this ensures that a single stray packet doesn't slow down the whole app. The prototype application in this thesis will still use WebSocket for data sharing instead of *RTCDataChannel* because the media server using in this system is not support *RTCDataChannel* yet, so it is not possible to create peer to peer session regarding to this issue. This case about using *RTCDataChannel* in prototype application will be discussed in Chapter 6.

2.1.4 Webtorrent

The goal of project Webtorrent is to build a browser BitTorrent client that requires no install, no plugin, no extension, and fully-interoperates with the regular BitTorrent network. It uses WebRTC Data Channels for peer-to-peer transport. Since WebTorrent is web-first, it's simple for users who do not understand .torrent files, magnet links, NATs, etc. By making BitTorrent easier, it will be accessible to new swathes of users who were previously intimidated, confused, or unwilling to install a program on their machine to participate.[Abo14]

Since WebRTC is usually used for peer to peer communication, the *RTCDataChannel* can be used in more creative way like Webtorrent. Although it need to keep the browser up and running on both ends, then there will be no asynchronous nature into it, it does reduce the bandwidth required and it adds privacy as to who has access to the file being shared. Since the application can reach direct between browsers, it can use the data channel to create a low latency network, where data is shared directly without going through servers on the way. It is lower cost for the developer and more secure on this case. For example, doing the same using a drastically larger number of web browser nodes as The Onion Router (TOR)[8], increases the chance of privacy.This can reduce the need for “real” web servers to run services, and use those only as points of access into the dynamic network that is created ad-hoc.

2.2 Prototype Working Scenario

The prototype system in this thesis will pay more attention on the real-time communication usage of WebRTC. The main purpose of the system is to combine internet browser user and traditional telephony user without complicate instillation, plugin and extension. There are two typical working scenarios of the prototype system will be described below.

2.2.1 Advanced ’one-number’ communication platform

Adam is a typical Facebook[9] user and he does synchronize his contact list through Google Contacts[10] by his smart phone. Now his operator provides user credential from his telephone number to him. Then Adam just login on his operator ’FellowPhone’ web page, now he can import his contacts list through his Google contact list. After that, he can see if his contact person is online by using the same web application ’FellowPhone’ or not. He can also import his Facebook friends list and fulfill the friends list with his contacts list information. Therefore, Adam can see if his facebook friends online or not. If his facebook friends/ Google contacts are online and use ’FellowPhone’ web application from their operator, Adam can invite them have a video conference otherwise his friends are not online then he can still invite them into the video conference but though his friends mobile phone with only audio sound.

During the video conference, Adam can send his online friends files and instance messages (website links, video links and so on). Moreover, his offline friends in the same conference will get the same information as text Short Message Service (SMS).

[8]Tor (previously an acronym for The Onion Router) is free software for enabling online anonymity and censorship resistance. Tor directs Internet traffic through a free, worldwide, volunteer network consisting of more than five thousand relays to conceal a user’s location or usage from anyone conducting network surveillance or traffic analysis.[Wik14x]

[9]Facebook is an online social networking service.

[10]Google Contacts is Google’s contact management tool that is available in its free email service Gmail, as a standalone service, and as a part of Google’s business-oriented suite of web apps Google Apps.[Wik14g]

Adam can reach his friends wherever they are and no matter if they are online or not as long as they have their mobile phone.

2.2.2 Multiple doctors consultation room

Eve is a 80-year-old lady, she lives with her children in their house. But at the day time, her children go to work, she need take care of herself. She has appointment with her doctor about her backache. But she can not go to hospital or family doctor office. Then she use her mobile phone to call her family doctor. Her family doctor, Isak using the prototype service from his company and operator. When Eve call to her doctor, Isak, for help, Isak answered her phone and try to get her previous medical information from other system. Then he found out that Eve had other doctor about her back treatment before. He can just login in the prototype system and find out if the other doctor is at work (online in the system). The other doctor, Stella, she has the treatment log about Eve. She got invitation to join the current conversion with Isak and Eve. She can send message to Isak and share the treatment log with Isak if necessary. She can listen to the talk between Isak and Eve about the new update of the treatment to give suggestion. Isak can ask for more different doctors in the system for advice and consultation to help Eve case.

But in Eve aspect, she only calls doctor Isak, and she can got help from more than one doctor at the same time. If it is necessary, she can use the computer to login the same system to have video conference with different doctors for her case. The only thing required for her is a telephone number and a mobile phone.

Chapter 3
Prototype System Design

In this chapter, it will cover system design progress of the prototype system along with explanation and analysis. The prototype system is designed based on preliminary studies from previous chapter. There will be different implementation solutions to the prototype working scenario discussed and evaluated in this chapter. After evaluating these solutions, it will come up with the fit solution to the prototype working scenario.

3.1 Prototype System Network

In the original WebRTC application implementation, it uses mesh network because WebRTC means to be the peer to peer communication method bypass the third party server. However, the prototype system will use centralized server network to control and route the communication channels between different types end points. In this section, it will describe the reason to use centralized server network rather than mesh network.

3.1.1 Mesh Network

A mesh network is a network topology in which each node (called a mesh node) relays data for the network, the illustration of the network is shown in Figure 3.1. All nodes cooperate in the distribution of data in the network. When WebRTC designed, it considered as mesh network using and take the advantages of the mesh network. Mesh network provides point-to-point line configuration makes identification and isolation of faults easy. The messages travel through a dedicated line in the mesh network, directly to the intended recipient. More privacy and security are thus enhanced. If a fault occurs in a given link of the network, only those communications between that specific pair of devices sharing the link will be affected.[Wik13d]

However, with the design of mesh network, the more extensive the network, in terms of scope or of physical area, the greater the investment necessary to build it

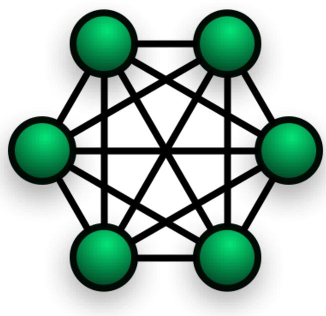

Figure 3.1: Illustration of a Mesh Network [Wik13d]

will be, due, among other considerations, to the amount of cabling and the number of hardware ports it will require. Every device must be connected to every other device, installation and re-connection are difficult. The huge bulk of the wiring can often be greater than the available space in the ceiling or under floors can accommodate.

Considering the prototype system case, a real-time communication system, the scaling problem in the feature will eventually be the top priority issue. With the mesh network, it is difficult and impossible to scale the system with the control since the network scales by the unknown end points.There is a similar production application called *appear.in*. It is a video conversations application with up to 8 people in the browser. *appear.in* uses peer-to-peer communication, meaning that the video streams are sent directly between the browser clients. Nothing is stored on the server and all the communication is encrypted over SSL. But the limit of 8 clients in one conversation is mainly because the client browser it self can not handle too many peer connections. Because according to mesh network, every client in the conversation would set up one unique WebRTC *RTCPeerConnection* object and one unique media stream exchange channel on the client, it consumes client computer resources a lot. Thus, the prototype system will not use mesh network as the system network architecture in order to avoid the future scaling problem. The advantages of the mesh network is well implemented in the WebRTC api, then the prototype system will keep these advantages to keep the point-to-point lines isolated with each other and keep the point-to-point communication more private and secure.

3.1.2 Centralized Network

Centralized network is a type of network where all users connect to a central server, which is the acting agent for all communications. This server would store both the communications and the user account information. Most public instant

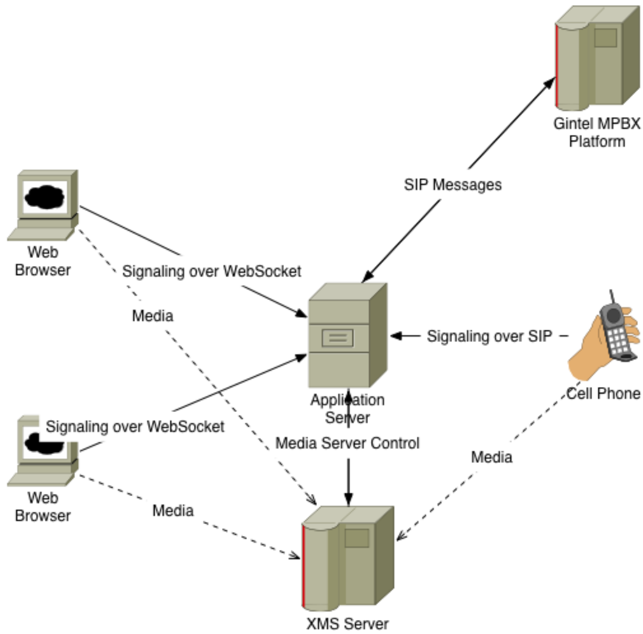

Figure 3.2: Prototype System Network

messaging platforms use a centralized network. It is also called as centralized server-structure.[Web14b] It is similar network architecture shown in Figure 3.2.

The advantages of centralized server network are centralized control of the system, centralized observation of the system and light requirement for the client . In the prototype system, there are application server and media server to handle the application logic business and media stream exchange business(see in Figure 3.2). Although every clients communicate with application to do the WebRTC signaling, the media stream is not go through the application server, it goes through the media server only. Furthermore, every client creates single WebRTC connection pair with the one resource on XMS media server, the advantage of point-to-point line configuration is still kept in the system. As client aspect, it still makes peer-to-peer media stream connection based on Secure Sockets Layer (SSL). The function of XMS media server is to combine more than two of the peer resources into one conference resource in order to set up the multimedia conference channel. More detail about XMS media server handling will be covered in Chapter 4.

The other important advantage of centralized server network is that the application server and media server can observe the condition and quality of the real-time conversation to administrate the routing and quality improvement process. For this reason, the media stream quality on every end point will be more stable and better quality control. Since the prototype application is to integrate with traditional telephony network, it is important to provide similar quality control and fault tolerant mechanism in the prototype system.

The disadvantage of centralized server network would be the application server and media server itself as well. During the development of the prototype system, it is easy to figure out that the machine to host the application server and media server is not powerful enough to handle too much client connection and media stream connection. When it meets the scaling issue, the application server and media server need to be distributed in multiple server host and consume powerful server machines. The cost of the entire system is higher than the mesh network solution.

As a conclusion of these two types network architecture, for this prototype system, it will be centralized server network, Figure 3.2, to be implemented because it is more suit to the goal of this thesis to be integrated with traditional telephony network usage.

3.2 Prototype Implementation Framework

Since WebRTC is a web API, the prototype application will be a web application. There are many different web application framework nowadays to provide rich-client web application. In this section, some of the web application framework will be discussed to figure out which framework is best solution to the prototype scenario. Furthermore, application server will be discussed with different implementation solutions since it does signaling and bridge the SIP network and clients.

3.2.1 Client Implementation Framework

To choose web application framework to implement the client application in this thesis scenario, the main fact is that if the web application framework is fit to the real time communication application and if the framework has the ability to integrate with WebRTC API. After research about these kinds of web application framework, it narrows down to three main framework to discuss.

AngularDart :

AngularDart is a framework for building web-apps in Dart. Dart is an open-source Web programming language developed by Google. It is a class-based, single inheritance, object-oriented language with C-style syntax. It supports interfaces, abstract classes, reified generics, and optional typing. Static type annotations do not affect the runtime semantics of the code. Instead, the type annotations can provide documentation for tools like static checkers and dynamic run time checks.[Wik14d]

Because most of the script language is not type strict, it is easy to mess up the code and value type in script language. Moreover, Dart has Dart-to-JavaScript compiler,dart2js, it makes Dart can be used in client and server both. Addition to AngularJs framework in Dart, it provide a professional web application structure to the developer to implement. More about AngularJs notable features will be covered in the later AngularJs solutions.

The WebRTC implementation in Dart is in this repository: https://github.com/br1anchen/AngularDart_webRTC. The Code Snippet C.1 shows the main controller in AngularDart. The line 5 is to import WebRTC client class *speack_client.dart*, the class has all the WebRTC APIs implemented in Dart. Line 23 is to initialize the *SpeakerClient* object and set the arguments WebSocket url and room name. They are used for signaling in WebSocket Protocol.

However, after implementation of client application and server back-end in Dart. There is a critical bug in the current Dartium browser. The Dart SDK ships with a version of the Chromium web browser modified to include a Dart Virtual Machine (VM). Dartium browser can run Dart code directly without compilation to JavaScript. It is intended as a development tool for Dart applications, rather than as a general purpose web browser. When embedding Dart code into web apps, the current recommended procedure is to load a bootstrap JavaScript file, "dart.js", which will detect the presence or absence of the Dart VM and load the corresponding Dart or compiled JavaScript code, respectively, therefore guaranteeing browser compatibility with or without the custom Dart VM.[Wik14d]

The issue noticed as **RtcPeerConnection.addIceCandidate results in a NotSupportedError: Internal Dartium Exception** in the Dart Google Project issues.[Iss14] The sample code in the WebRTC Dart implementation shown in Code Snippet 3.1, line 1 is to create *RTCPeerConnection* object. From line 5 to line 13 is to send message to server when *RTCPeerConnection* object get *onIceCandidate* event witch ICE candidate information. Line 17 is to bind the message listener event to Dart function *onCandidate.listen*. From line 21 to line 30 is the Dart function to create *RTCIceCandidate* object and add to *RTCPeerConnection* object. The bug issue happens on line 27, when the *RTCPeerConnection* call *addIceCandidate* function, it is not allowed to have callback function in current version Dartium.

Code Snippet 3.1: Add IceCandidate in Dart

```
1 var pc = new RtcPeerConnection(_iceServers, _dataConfig);
2
3 ....
4
5     pc.onIceCandidate.listen((e){
6       if (e.candidate != null) {
```

```
        _send('candidate', {
          'label': e.candidate.sdpMLineIndex,
          'id': id,
          'candidate': e.candidate.candidate
        });
      }
    });

...

get onCandidate => _messages.where((m) => m['type'] == 'candidate');

...

onCandidate.listen((message) {
  var candidate = new RtcIceCandidate({
    'sdpMLineIndex': message['label'],
        'candidate': message['candidate']
    });

    _connections[message['id']].addIceCandidate(candidate,(){},(e){
        print('add ice candidate error');
    });
});

...
```

There is a work around solution in one Stack Overflow[1] answer: http://stackoverflow.com/questions/20404312/how-to-call-addicecandidate-in-dart. The fix method is to use *js-interop* library to use pure JavaScript code in Dart to call the WebRTC Web API instead of Dart WebRTC interface.

Mozilla's Brendan Eich, who developed the JavaScript language, stated that:

"I guarantee you that Apple and Microsoft (and Opera and Mozilla, but the first two are enough) will never embed the Dart VM. So 'Works best in Chrome' and even 'Works only in Chrome' are new norms promulgated intentionally by Google. We see more of this fragmentation every day. As a user of Chrome and Firefox (and Safari), I find it painful to experience, never mind the political bad taste."[Wik14d]

[1] Stack Overflow is a privately held website, the flagship site of the Stack Exchange Network, created in 2008 by Jeff Atwood and Joel Spolsky, as a more open alternative to earlier Q&A sites such as Experts Exchange.

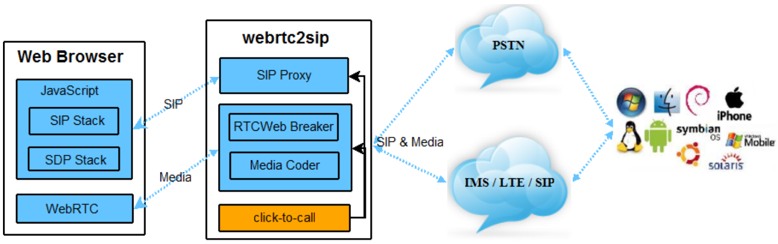

Figure 3.3: Sipml5 and webrtc2sip Network

Since Dart in not support to most modern web browser like FireFox, will not be used in this prototype.

Sipml5 + webrtc2sip:

Sipml5 is the world's first open source HTML5 SIP client entirely written in JavaScript for integration in social networks (FaceBook, Twitter, Google+), online games, e-commerce websites, email signatures. The media stack rely on WebRTC. The client can be used to connect to any SIP or IP Multimedia Subsystem (IMS) network from your preferred browser to make and receive audio/video calls and instant messages.[Tel14a]

Sipml5 provides whole client solution to communicate with other kind of signaling real-time communication network. The SIP and SDP stacks are entirely written in JavaScript and the network transport uses WebSockets as per draft-ibc-sipcore-sip-websocket. However the community of sipml5 is not so active, the issues and source code on sipml5 source code project website https://code.google.com/p/sipml5/ are not updated regularly. Like the Figure 3.3 showing, it works with media gateway webrtc2sip.

webrtc2sip is a smart and powerful gateway using WebRTC and SIP to turn your browser into a phone with audio, video and SMS capabilities. The gateway allows your web browser to make and receive calls from/to any SIP-legacy network or PSTN. The gateway contains four modules: SIP Proxy, RTCWeb Breaker, Media Coder, Click-to-Call.[Tel14b]

In the prototype working scenario, it is necessary to have media gateway to communicate with SIP-legacy network. Since the current PSTN using in this prototype go through Gintel Multimedia Private Branch Exchange (MPBX) Platform, it is necessary to use RTCWeb Breaker to be able to connect the browser to a SIP-legacy endpoint.

Therefore, the test for Sipml5 and webrtc2sip solution is based on the live demo http://sipml5.org/call.htm. But even with the RTCWeb Breaker, the test is still failed to call any number through the target PSTN. Since most of the source code of these two framework are hidden from the encapsulation, it is impossible to debug which part of the testing system is the problem. In the test, the registration for SIP client is successful, but there are 'too long message' in the SIP error message got from the SIP server. It means that the sipml5 and webrtc2sip network architecture is not compatible with the target PSTN through the Gintel MPBX Platform. This solution can not be used in the prototype system.

AngularJs + Socket.IO:

AngularJS is built around the belief that declarative programming should be used for building user interfaces and wiring software components, while imperative programming is excellent for expressing business logic. The framework adapts and extends traditional HyperText Markup Language (HTML) to better serve dynamic content through two-way data-binding that allows for the automatic synchronization of models and views. As a result, AngularJS de-emphasizes Document Object Model (DOM) manipulation and improves testability. Angular follows the Model–View–Controller (MVC) pattern of software engineering and encourages loose coupling between presentation, data, and logic components. Using dependency injection, Angular brings traditional server-side services, such as view-dependent controllers, to client-side web applications. Consequently, much of the burden on the backend is reduced, leading to much lighter web applications.[Wik14a]

AngularJs is perfect for single-page web application, the framework features provide developer a professional way to structure the web application in JavaScript. Moreover, the developer community of AngularJs is quite active, there are a lot of different Angular module services to provide the different interfaces against different web APIs.In the prototype, there will be several third party Angular module library to be used in order to integrate with some advanced JavaScript library or web APIs in Angular style.

Socket.IO is a JavaScript library for realtime web applications. It has two parts: a client-side library that runs in the browser, and a server-side library for node.js. Both components have a nearly identical API. Socket.IO primarily uses the WebSocket protocol, but if needed can fallback on multiple other methods, such as Adobe Flash sockets, JSON with padding (JSONP) polling, and Asynchronous JavaScript and XML (AJAX) long polling, while providing the same interface. Although it can be used as simply a wrapper for WebSocket, it provides many more features, including broadcasting to multiple sockets, storing data associated with each client, and asynchronous I/O.[Wik14w] In the prototype application, Socket.IO is used in WebSocket protocol because the WebSocket protocol provides full-duplex communications channels over a single TCP connection. Then the communication channel will be active and real time between the clients and server during the whole connecting procedure. It fits the real time communication application requirement.

After test demo client application implemented in AngularJs and Socket.IO frameworks, it works fine with the basic WebRTC functions and simple SIP registration against SIP server to target PSTN. The final decision of the client implementation framework of prototype system will be AngularJs and Socket.IO.

3.2.2 Server Implementation Framework

Since the client side will use Socket.IO as communication protocol library, the server back-end in the prototype system will use Node.js as server implementation framework. In thist section, more detail about comparison and differences of Node.js against traditional web service back-end (in Java, ASP .NET[2] or PHP) will be covered.

Node.js:

Node.js is a software platform for scalable server-side and networking applications. Node.js applications are written in JavaScript, and can be run within the Node.js runtime on Mac OS X, Windows and Linux with no changes. Node.js applications are designed to maximize throughput and efficiency, using non-blocking I/O(Input/Output) and asynchronous events. Node.js applications run single-threaded, although Node.js uses multiple threads for file and network events. Node.js is commonly used for real time applications due to its asynchronous nature.[Wik14o]

At high levels of concurrency server needs to go to asynchronous non-blocking, otherwise there will be blocking Input/Output (IO) on the server to delay other IO process. The issue is that if any part of the server code blocks, on the traditional server framework, it is going to need a thread. And at these levels of concurrency, it can't keep creating threads for every connection. Then the whole codepath needs to be non-blocking and synchronized, not just the IO layer. This is where Node excels, shown in Figure 3.4. The main difference between Figure 3.4 and Figure 3.5 is the way of server to handle the requests. On Node.js server, it handles all the requests in asynchronous threads after the requests are delegated from event loop. But on multiple threaded server, programming language used on these server mostly does not support for the async pattern. Then it would not matter whether raw Non-Blocking I/O (NIO) performance is better than Node or any other benchmark result.

[2] ASP.NET is a server-side Web application framework designed for Web development to produce dynamic Web pages. It was developed by Microsoft to allow programmers to build dynamic web sites, web applications and web services.[Wik14c]

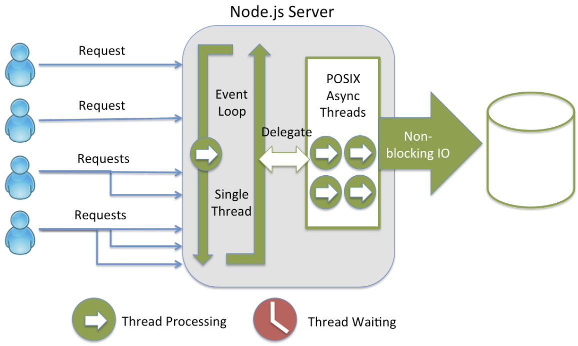

Figure 3.4: Node.js Non-blocking I/O[Rot14]

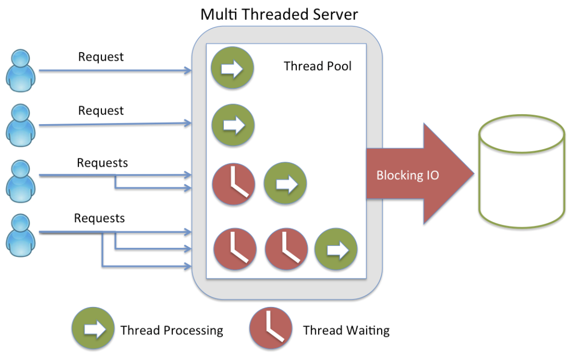

Figure 3.5: Multiple Threaded Server[Rot14]

Since the prototype is a real-time communication application, it is better to use Node.js as back-end server rather than multiple threaded server. Moreover, the WebSocket protocol framework used on client side has good server side solution based on Node.js, it makes the development of the prototype system much easier to implement. The prototype system is a centralized server network, the communication between application server to XMS server will be hold on normal HTTP/HTTPS protocol, Node.js provides these protocol communication as well, no need to host any additional web server software such as Apache[3].

For the other part of the prototype, SIP network, there is a existing Node.js module can be used as SIP stack on Node.js server. sip.js is a SIP stack for node.js. It implements tranaction and transport layers as described in RFC3261[4].[kir14] Although sip.js is not production framework yet, it is one of the few SIP stack

[3]The Apache HTTP Server, commonly referred to as Apache, is a web server application notable for playing a key role in the initial growth of the World Wide Web.[Wik14b]

[4]SIP: Session Initiation Protocol

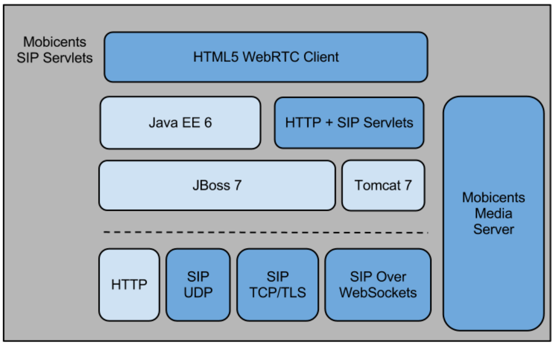

Figure 3.6: Mobicents SIP Servlets[Tel14c]

library in Node.js. It provides SIP message parser, UDP/TCP/ TLS based transport transactions and digest authentication. These features are quite fit to the prototype requirement and quite handy to implement.

There will be more detail about SIP implementation on sip.js library on Node.js in the later chapter. Since it is not mature library, there are quite few stuff need to be fixed through the development.

Mobicents Sip Servlets

Mobicents SIP Servlets delivers a consistent, open platform on which to develop and deploy portable and distributable SIP and Converged Joint Entrance Examination (JEE) services. It is the first open source certified implementation of the SIP Servlet v1.1 (Java Specification Requests (JSR) 289 Spec) on top of Tomcat[5] and JBoss[6] containers and strive to feature best performances, security, foster innovation and develop interoperability standards between SIP Servlets and JAIN Service Logic Execution Environment (JSLEE) so that applications may exploit the strengths of both. The Java APIs for Integrated Networks (JAIN)[7]-SIP Reference implementation is leveraged as the SIP stack and Mobicents JAIN Service Logic Execution Environment (SLEE)[8] is used as the SLEE implementation.

[5] Apache Tomcat (or simply Tomcat, formerly also Jakarta Tomcat) is an open source web server and servlet container developed by the Apache Software Foundation (ASF).[Wik13a]

[6] WildFly, formerly known as JBoss AS, or simply JBoss, is an application server authored by JBoss, now developed by Red Hat. WildFly is written in Java, and implements the Java Platform, Enterprise Edition (Java EE) specification. It runs on multiple platforms.[Wik13f]

[7] Java APIs for Integrated Networks (JAIN) is an activity within the Java Community Process, developing APIs for the creation of telephony (voice and data) services.[Wik13c]

[8] An accelerated development and deployment environment of new IP Multimedia Subsystem (IMS) services for convergent fixed- mobile network environments.[Bah14]

The architecture of the Mobicents SIP Servlets is shown in Figure 3.6. As it described, Mobicents SIP Servlets provide multiple transport protocol include HTTP, UDP, TCP and WebSocket. These transport protocols are fit the prototype requirements, but on the application layer, it has two application server need to be host, one is JBoss and the other is Tomcat 7, JBoss is support for all the SIP stack transport and Tomcat 7 is support for HTTP requests. JBoss is the gateway to communicate with SIP network and Tomcat host the application server to communicate with media server to handle the real-time multimedia stream.

It is quite nice system architecture to work with, but it need powerful server machine to host two web application server on it. Considering Node.js solution, it is not easy to maintain the system since developer need to configure on two different web application server to handle different protocol transportation and client and server are implemented in different programming languages.

After implemented one test application by Mobicents SIP Servlets framework, it is hard for developer to program the lower level source codes, for example SIP message headers field modification and WebSocket transport template. The test application successes to set up conversation session between WebRTC browser client and SIP client. But when the media stream exchange on XMS server(media server), there is only one way audio(from browser to phone) in the conversation. Because the SIP transport layer is encapsulated in the Mobicents SIP Servlets framework, it is hard to modify it to do more test if the bug is in the transport layer of the framework. The source code of this test application is owned by Gintel AS.

As a conclusion, the prototype system will use Node.js as server side back-end to communicate with AngularJs client application on Socket.IO protocol.

Chapter 4

Prototype System Implementation

In this chapter, it will cover development implementation progress of the prototype system along with explanation and analysis. The prototype system is implemented based on system design from chapter 3.

4.1 WebRTC APIs Implementation

WebRTC components are accessed with JavaScript APIs. Currently in development are the Network Stream API, which represents an audio or video data stream, and the PeerConnection API, which allows two or more users to communicate browser-to-browser. Also under development is a DataChannel API that enables communication of other types of data for real-time gaming, text chat, file transfer, and so forth. Because the media server used in prototype system is not support for DataChannel yet, the DataChannel API will not be covered in this section.

4.1.1 MediaStream API

The MediaStream API represents synchronized streams of media. For example, a stream taken from camera and microphone input has synchronized video and audio tracks. In order to obtain local media, the start step for both peers in Figure 4.1 which is a communication process to set up call process from caller peer, the WebRTC APIs provide *navigator.getUserMedia()* function to get the video and audio stream from user. For privacy reasons, a web application's request for access to a user's microphone or camera will only be granted after the browser has obtained permission from the user. Each MediaStream has an input, which might be a MediaStream generated by *navigator.getUserMedia()*, and an output, which might be passed to a video element or an *RTCPeerConnection*.

The *getUserMedia()* method takes three parameters:

– A constraints object.
– A success callback which, if called, is passed a MediaStream.

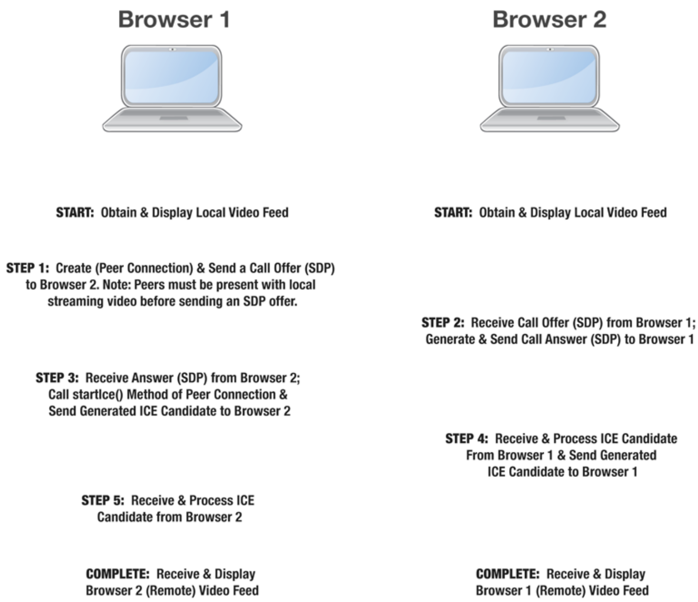

Figure 4.1: WebRTC two peer communication process[Net14]

– A failure callback which, if called, is passed an error object.

The Code Snippet 4.1 shows that how the prototype application implements *getUserMedia()* function, it is encapsulated in *WebRTCService* (service is a reusable business logic independent of views in prototype application regarding to AngularJs framework[1]).For the constraints object in parameters, the prototype application set 'audio' and 'video' value to true because it is necessary for the real-time communication application to have video and audio stream both.

Code Snippet 4.1: Get User Media Stream function

```
var media_constraints = {audio: true,video: true};

function _setMediaStream(){
  WebRTCService.getUserMedia(media_constraints,
                             _handleUserMedia,
                             _handleUserMediaError);
```

[1]AngularJS is an open-source web application framework, maintained by Google and community, that assists with creating single-page applications, one-page web applications that only require HTML, CSS, and JavaScript on the client side.[Wik14a]

```
    console.log('Getting user media with constraints',
          media_constraints);
}
```

getUserMedia() function is currently available in Chrome, Opera and Firefox. Almost all of the WebRTC APIs are slightly different based on different browsers implementation. In the Code Snippet A.2, from line 40 to line 103 is to make all the set up process for FireFox and from line 106 to line 175 is to make the same set up process for Google Chrome. Because WebRTC is not standard Web API yet, so the implementation on different browsers are different and the WebRTC APIs names are slightly different in some browsers. For example, in the Code Snippet A.2 showing, the *RTCPeerConnection* API in Firefox is *mozRTCPeerConnection* but in Google Chrome it is *webkitRTCPeerConnection*. In order to make the WebRTC application works on more browsers, the client side need to figure out which kind of browser is using on the machine then call the corresponding WebRTC APIs. Google provides a JavaScript shim called *adapter.js*. It is maintained by Google, it abstracts away browser differences and spec changes. For Angularjs framework used by prototype application, then the *WebRTCService* is implemented to be integrated with *adapter.js* function to achieve the goal of compatibility.

However, the prototype application in this thesis will only focus on Google Chrome browser[2] to simplify the development process because WebRTC lower level implementation on different browser s are different and hard to track the issues. Then most of the results in this thesis is based on the application performance of Google Chrome browser. The reason to choose Google Chrome browser rather than other browser because WebRTC is the technology rapidly pushed by Google and Google Chrome browser has the most market share in the world. As of March 2014, StatCounter estimates that Google Chrome has a 43% worldwide usage share of web browsers, making it the most widely used web browser in the world.[Wik14f] However, Google changes a lot to improve the performance of WebRTC on Google Chrome browser, then it makes the WebRTC APIs work different on different version of Google Chrome browser. In the Code Snippet A.2, from line 124 to line line 136 is the sample case to distinguish the difference among different version of Google Chrome to handle the *RTCPeerConnection* ICE server constraint implementation.

Since WebRTC APIs is not standard API yet, the prototype application in this thesis will not pay too much work-load on compatibility for different browsers platform. More detail about this issue will be discussed in the Chapter 6.

[2]Google Chrome is a freeware web browser developed by Google. It used the WebKit layout engine until version 27 and, with the exception of its iOS releases, from version 28 and beyond uses the WebKit fork Blink.[Wik14f]

4.1.2 RTCPeerConnection API

To set up peer connection, the *RTCPeerConnection* API sets up a connection between two peers. In this context, “peers” means two communication endpoints on the World Wide Web. Instead of requiring communication through a server, the communication is direct between the two entities. In the specific case of WebRTC, a peer connection is a direct media connection between two web browsers. This is particularly relevant when a multi-way communication such as a conference call is set up among three or more browsers. Each pair of browsers will require a single peer connection to join them, allowing for audio and video media to flow directly between the two peers.

To establish peer connection, it requires a new *RTCPeerConnection* object. The only input to the *RTCPeerConnection* constructor method is a configuration object containing the information that ICE, will use to “punch holes” through intervening NAT devices and firewalls. The Code Snippet 4.2 shows the create *RTCPeerConnection* object and set three listener (*onicecandidate*,*onaddstream*,*onremovestream*) to trigger the handlers to deal with the ICE candidate event and remote stream add/remove events.

The *RTCPeerConnection* API has two arguments to set, one is configuration object for peer connection and the other is constraint object (set transparent protocol and encryption) for peer connection, these value are shown in Code Snippet 4.2 line 1 to line 10. In the showing case, the prototype is using STUN servers for different browser aspect, and set the RTC channel encryption protocol to Datagram Transport Layer Security (DTLS)[3] and enable the RTC DataChannel.

Because in Firefox, WebRTC media transparent channel is only based on DTLS protocol, and in latest version Google Chrome, it is support, then in the prototype application, it will use DTLS protocol to exchange the media stream.

There are two APIs to handle the *IceCandidate* object which contains ICE information data. One is *onicecandidate* listener to trigger the function to handle the new *IceCandidate* data object. The other one is *addIceCandidate* function, which is shown in the Code Snippet 4.3, to add the new *IceCandidate* data object to the remote/local peer connection session description field.

Code Snippet 4.2: Create Peer Connection function

```
1 pc_config = WebRTCService.webrtcDetectedBrowser() === 'firefox' ?
2         {'iceServers':[{'urls':'stun:stun.services.mozilla.com'}]} :
```

[3]In information technology, the Datagram Transport Layer Security (DTLS) protocol provides communications privacy for datagram protocols. DTLS allows datagram-based applications to communicate in a way that is designed to prevent eavesdropping, tampering, or message forgery.[Wik14e]

```
        {'iceServers':[{'urls': '
            stun:stun.l.google.com:19302'}]};

pc_constraints = {
        'optional': [
          {'DtlsSrtpKeyAgreement': true},
          {'RtpDataChannels': true}
        ]
      };

function _createPeerConnection(){

  try {
    pc = WebRTCService.peerConnection(pc_config,
        pc_constraints);
    pc.onicecandidate = _handleIceCandidate;
    console.log('Created RTCPeerConnnection with:\n' +
          '  config: \'' + JSON.stringify(pc_config) + '\';\
             n' +
          '  constraints: \'' + JSON.stringify(
             pc_constraints) + '\'.');
  } catch (e) {
    console.log('Failed to create PeerConnection, exception:
        ' + e.message);
    alert('Cannot create RTCPeerConnection object.');
    return;
  }
  pc.onaddstream = _handleRemoteStreamAdded;
  pc.onremovestream = _handleRemoteStreamRemoved;

}
```

Code Snippet 4.3: Add Remote IceCandidate function

```
var candidate = WebRTCService.RTCIceCandidate({
                sdpMLineIndex:data.content.label,
                sdpMid:data.content.id,
              candidate:data.content.candidate
        });
pc.addIceCandidate(candidate);
```

In the step 2 of Figure 4.1, after the caller *RTCPeerConnection* run *createOffer()* function to send offer to callee through signaling channel, the callee need run *createAnswer()* function to ask the STUN/TURN server to find the path for each other peer and create the answer with SDP content. SDP is intended for describing multimedia communication sessions for the purposes of session announcement, session invitation, and parameter negotiation. SDP does not deliver media itself but is used for negotiation between end points of media type, format, and all associated properties.[Wik14t] Before *RTCPeerConnection* use *createOffer()* function to send a WebRTC offer to the callee, it is required to be present with local streaming video, like Figure 4.1 mentioned.

The sample SDP from the prototype application is shown in Code Snippet 4.4. Line 2 in Code Snippet 4.4 is the field 'o', it describes originator, session identifier, username, id, version number and network address. It usually means that where this package comes from. Line 7 and line 17 are field 'm', it describes media name and transport address. And line 11,12 and line 27,28 are the relevant lines for audio and video media field, they describes media filed 'candidate' attributes, in the sample case of Code Snippet 4.4, they are the ICE candidate from the STUN/TURN server. These are important fields regarding to the prototype system because they are used in XMS server and application server of the prototype system.

Code Snippet 4.4: Sample WebRTC Answer SDP

```
1  sdp: v=0
2  o=xmserver 1399363527 1399363528 IN IP4 10.254.9.135
3  s=xmserver
4  c=IN IP4 10.254.9.135
5  t=0 0
6  a=ice-lite
7  m=audio 49152 RTP/SAVPF 0 126
8  a=rtpmap:0 PCMU/8000
9  a=sendrecv
10 a=rtcp:49153
11 a=candidate:1 1 UDP 2130706431 10.254.9.135 49152 typ host
12 a=candidate:1 2 UDP 2130706430 10.254.9.135 49153 typ host
13 ...
14 a=acfg:1 t=1
15 a=rtpmap:126 telephone-event/8000
16 a=fmtp:126 0-15
17 m=video 57344 RTP/SAVPF 100
18 b=AS:1000
19 a=rtpmap:100 VP8/90000
20 a=fmtp:100 max-fr=30; max-fs=1200
21 a=sendrecv
22 a=rtcp:57345
23 a=rtcp-fb:100 ccm fir
24 a=rtcp-fb:100 nack
25 a=rtcp-fb:100 nack pli
26 a=rtcp-fb:100 goog-remb
```

```
a=candidate:2 1 UDP 2130706431 10.254.9.135 57344 typ host
a=candidate:2 2 UDP 2130706430 10.254.9.135 57345 typ host
...
```

In the step 3 of Figure 4.1, the caller will receive the answer from callee and process it by adding the remote SDP to *RTCPeerConnection*, like the Code Snippet 4.3. By the meantime, the step 4 of Figure 4.1, the callee will receive the SDP from caller with the ICE candidate information data, and process it the same way as caller does, add some to *RTCPeerConnection* object by *addIceCandidate()* function.

WebRTC clients (known as peers) also need to ascertain and exchange local and remote audio and video media information, such as resolution and codec capabilities. Signaling to exchange media configuration information proceeds by exchanging an offer and an answer using the SDP. The *createOffer()* function and *createAnswer()* function both have callback function to handle the SDP either to call *setLocalDescription()* by caller or call *setRemoteDescription()* by callee when callee gets the caller's SDP from WebRTC offer. The Code Snippet4.4 shown is the WebRTC answer SDP from the callee when the callee end-point decide to accept this conversion session.

Once the *RTCPeerConnection* is established, the client need configure where the media or data to store and display if it is necessary. In the prototype application of this thesis, media stream will be displayed in a HTML5 tag called *<video>*. It will only be shown when there is media stream in *<video>* tag source.

4.2 AngularJs framework Implementation

As it described about AngularJs in Chapter 2, there are three layer components in the framework, view, controller and service. The files structure is shown in Appendix D.1. Application has two main pages, *login* page and *phone* page. There are *chatboard*,*contacts list*, *contacts table*, *dialpanel* and *notification* user interface component block in *phone* page. For each part of the application block, it has controller Javascript file and service Javascript file. Controller and service scripts are working with the HTML view scripts. In this section, there will be one sample part of the prototype application client explained to understand how the AngularJs is used in prototype application.

The *app.js* script shown in Code Snippet A.1 is the bootstrap script for AngularJs framework. It initializes the application module of AngularJs framework and declare the dependencies which will be used in the application.

The contact table component in *phone* page of the application is structured in four scripts, *contactTable.jade* script in Code Snippet A.3, *ContactTableDirective.js* script in Code Snippet A.4, *ContactsCtrl.js* script in Code Snippet A.5 and *GoogleAPIService.js* script in Code Snippet A.6. It provides the application contacts information in advanced functioning table and search function in text input filed.

4.2.1 app.js Script (AngularJs Bootstrap)

The *app.js* script is shown in Code Snippet A.1, it declares the application level module which depends on different filters, modules and services. The modules *webrtcDemo.services*, *webrtcDemo.controllers*, *webrtcDemo.directives* and *webrtcDemo.filters* are the customized modules implemented for prototype application. The rest of the module which are included as dependencies are third party AngularJs modules used in the prototype application. AngularJs developer community is quite active community, there are many useful open sourced projects or modules can be just included like the line 6 to line 15 in Code Snippet A.1.

From line 22 to line 41 is the configuration for the application level module *webrtcDemo*, in the prototype application code, it used to set the application routing map. There are two main pages, one is *login* page with "/login" URL(line 24,25) and the other one is *phone* page with "/chat" URL(line 28,29). The Angular controllers which are bind with these page view are also declared in *$routeProvider* service in line 26 and line 30. And the default URL is set to "/login" to make sure if user has not logged in the system, he need to input the user credential to log himself.

From line 43 to line 49 in the Code Snippet A.1 are the modules declaration part for four different customized modules used in application.

4.2.2 contactTable.jade Script (View)

The *contactTable.jade* script is the view component of the AngularJs. It is a Jade[4] script file. The template engine used on Node.js in prototype application is Jade which provides more clear way to program HTML node template scripts. In the Code Snippet A.3, Jade has the same node name as normal node template engine Embedded JavaScript templates (EJS) and some Angular directives in the template. For example, at line 2 in Code Snippet A.3, the *angucomplete-alt* directive is a third party Angular directive to provide auto-completion features in HTML *<input>* text tag. The different attributes in the *angucomplete-alt* node is to set some configuration to this directive, like the attribute files *local-data* is the array data to search for content as auto-complete reference.

Moreover, AngularJs itself provides native Angular directive as well. For instance, at line 17 in Code Snippet A.3, the attribute *ng-class* is a native Angular directive

[4]Jade is a high performance template engine heavily influenced by Haml and implemented with JavaScript for node.[vis14]

attribute, it provides the Cascading Style Sheets (CSS)[5] change to some specific CSS class name when some certain value matches in AngularJs expression. At line 17, the *<tr>* tag's CSS attributes will be success class only if the boolean value of *item.online* is *true*.

AngularJs provides two-way data module binding in the template and controller. Line 63 in the Code Snippet A.3, *{{item.number}}* is the Angular template to display the *number* property value of *item* object in the HTML template. And line 21 is the example of Angular template integrated with Angular filter, the third-party filter *iif* here is the filter to check the *{{item.online}}* value if it is *true* or *false*. If it is *true* then it will show *Online* string text in the HTML template otherwise it will show *Offline* string text. The syntax here is quite similar to any other programming language.

4.2.3 ContactTableDirective.js Script (Customized Directive)

After creating the view of contact table component, it is necessary to make a customized directive to bind controller to the view. It is called *Directive* in AngularJs, the *ContactTableDirective.js* script is shown in Code Snippet A.4. From line 10 to line 20 is the directive declaration, it sets the *templateUrl* to 'partials/contactTable' which is the view component of contact table file path and binds the controller which name *ContactsCtrl* to the view component. The *restrict* filed in the directive is to set the template type for *ContactTableDirective*, in the Code Snippet A.4 line 13, it means this directive is a HTML element template, it can be used as normal HTML element by using name 'contact-table'.

From line 22 to line 27 is the Angular filter declaration, there is one filter name *iif*, the only function it does is to check the *input* value and return *trueValue* if *input* is *true* otherwise return *falseValue*. The usage is described in previous section in line 21 of the Code Snippet A.3.

4.2.4 ContactsCtrl.js Script (Controller)

The controller in AngularJs is to control the user interface logic and bridge the data business logic from the services to the user interface views. The example controller in Code Snippet A.5 controls the contactTable view directive and get data from GoogleAPIService. In the line 10 of Code Snippet A.5, in the controller construction function, there are several services arguments. They are the services this controller will use in the application, one of them is *GoogleAPIService* which is related to the contacts information data. The contactTable view directive need contacts information

[5]Cascading Style Sheets (CSS) is a style sheet language used for describing the look and formatting of a document written in a markup language.[Wik13b]

data to show in the HTML template. And *storage* is another service provides textitlocalstorage function in HTML5 application. This service is used to store the contacts information data locally to make user no need to import his Google contacts information all the time. This function is implemented at line 82 of the Code Snippet A.5.

At line 56 of the Code Snippet A.5, it is the function *$scope.importContacts*, the reason this function is under *$scope* object is because this function is directly triggered by one User Interface (UI) button. In this function, there are two Javascript promise function from the *GoogleAPIService*. One is *GoogleAPIService.oAuth()* function which is to ask user to get Google API permission to query the Google Contacts API. The other one is after get the Google API permission to query the contacts information data by Google Contacts API.

Promise object is the new concept in the Javascript and AngularJs. The core idea behind promises is that a promise represents the result of an asynchronous operation. A promise is in one of three different states:[pro14]

– **Pending** - The initial state of a promise.
– **Fulfilled** - The state of a promise representing a successful operation.
– **Rejected** - The state of a promise representing a failed operation.

It is the very cool concept in the AngularJs. Since everything in Javascript is asynchronous operation, then promise object function is used to deal with the function calling after previous asynchronous operation success. The implementation of these two promise functions will be covered in the next section.

From line 60 to line 78 is the process to filter out the useful information from the response data to get the correct contact information into *contact* object, then push them one by one into a *contact* object array in order to be used by contact table view component.

4.2.5 GoogleAPIService.js Script (Service)

AngularJs service provides most of the business logic of the application. Like the sample code shown in Code Snippet A.6, it provides interfaces of Google API to the controller. There are two interfaces in the *GoogleAPIService.js* script. One is Google authorization login and get the user permission, the other one is fetching Google contacts information from the Google Contacts API.

From line 14 to line 30 in Code Snippet A.6, it is the promise function, *_authLogin()*, to get Google authorization token in order to call any Google API later. It uses *$q* service from AngularJs to provide *deferred* API and *prmoise* API. The purpose of the deferred object is to expose the associated Promise instance as well as APIs that can be used for signaling the successful or unsuccessful completion, as well as the status of the task.The purpose of the promise object is to allow for interested parties to get access to the result of the deferred task when it completes.[ang14] At line 15 and line 29 is the code to create a new instance of deferred and a new promise instance. From line 17 to 20, it is the configuration object for Google API authorization. The *gapi* object is from the Google API Javascript client script included in *index.jade* shown in Code Snippet 4.5.

Code Snippet 4.5: Include Google API Javascript file in Index.iade

```
script(src='https://apis.google.com/js/client.js' type='text
    /javascript')
```

Since application only need to get permission form user Google Contacts, then the scope is set to https://www.google.com/m8/feeds and the client_id is got from the Google App Engine (https://console.developers.google.com). Developer need to create his own Google App project then set the APIs which the project will ask user permission to use and the credentials used for client or web service. In the prototype system, it is the web application client to use the Google Contacts API then there is a client Open standard for Authorization (OAuth) 2.0[6] credential created on Google App project.

Then the *gapi* object call *auth.authorize()* function with the configuration object to get authorization token. At line 25, when the asynchronous process is finished, *deferred* object to call *resolve* function to send the token object back to the promise *then* function at line 58 in Code Snippet A.5 which is mentioned at previous section.

From line 32 to line 45 in Code Snippet A.6 is another promise function, *_fetch-Contacts()* , to fetch the Google contacts information data after getting user permission to use their Google service data. This function makes a HTTP request in JSONP to fetch all the contacts information from Google Contacts API. JSONP is a communication technique used in JavaScript programs running in web browsers to request data from a server in a different domain, something prohibited by typical web browsers because of the same-origin policy. JSONP takes advantage of the fact that browsers do not enforce the same-origin policy on *<script>* tags. Th reason the application uses JSONP in HTTP request is that web application is host in one origin domain and Google API server is in another origin domain, it is cross domain request when prototype application request for data from Google API server. And Google API server does not support cross domain request, but with JSONP it is allowed to have cross origin domain resources sharing.

[6] OAuth is an open standard for authorization. OAuth provides client applications a 'secure delegated access' to server resources on behalf of a resource owner.OAuth 2.0 is the next evolution of the OAuth protocol and is not backwards compatible with OAuth 1.0. OAuth 2.0 focuses on client developer simplicity while providing specific authorization flows for web applications, desktop applications, mobile phones, and living room devices.[Wik13e]

The *_fetchContacts()* function uses the same mechanism as *_authLogin()* function described above to make promise function, it returns contacts information data from Google Contacts API.

4.3 Socket.IO Implementation

In the prototype system, web application client and application server are communicating over WebSocket shown in Figure 3.2. There are two main intentions to have the signaling channel over WebSocket. One is to signaling for WebRTC ICE candidate exchange and the other one is to exchange the communication data (text message, files). Unlike HTTP, WebSocket provides for full-duplex communication. Additionally, Websocket enables streams of messages on top of TCP. TCP alone deals with streams of bytes with no inherent concept of a message. Before WebSocket, port 80 full-duplex communication was attainable using Comet channels; however, Comet implementation is nontrivial, and due to the TCP handshake and HTTP header overhead, it is inefficient for small messages. WebSocket protocol aims to solve these problems without compromising security assumptions of the web.[Wik14z]

4.3.1 Server Side Implementation

The Code Snippet B.1, it is implementation of Socket.IO on the application server. From line 1 to line 13, they are the intialization process of the Socket.IO on Node.js. At line 10, it means that when the client binds with the application server through WebSocket, the listener start in handler function *_handlerSocket()*. The WebSocket channels and usage is shown in Table 4.1.

4.3.2 Client Side Implementation

Since Socket.IO library is a library to make the communication channel between server and client, besides server side implementation, there is client side implementation(shown in Code Snippet A.7) which is correspond to the server side implementation.

Table 4.1: : Socket.IO Listening Channels

WebSocket Channel	Message Data Type	System Function
SIP	register(line 30 - 37)	Web application login page SIP registeration message to SIP server
	invite(line 38 - 107)	Web application client invite SIP client message
	answerInvite(line 108 - 167)	Web application client get INVITE SIP message from SIP client and answer it
WebRTC	register(line 248 - 283)	Web application client finish login with SIP credential and get user permission to use *getUserMedia()* function and register client itself on application server for WebSocket use
	offer(line 218 - 247)	Web application client send offer message to appliction server to create call resource on XMS media server
	answerInvite(line 173 - 194)	Web application client get INVITE message from WebRTC client and answer it
	endCandidate(line 284 - 317)	Web application client finish get ICE candidate from STUN/TURN server then application send HTTP request to XMS media server with final SDP
	hangup(line 321 - 551)	Web application client send hangup messge to hangup iteself from the current conference
message	Instance Message (IM)(line 566 - 582)	Web application send instance message to application server in order to broadcast to all the clients in current conference
	SMS(line 584 - 594)	Web application client send SMS to SIP client
disconnect	*(line 604 - 639)	Web application client disconnect from the application server

4.4 SIP Implementation on Application Server

4.5 XMS Media Server Integration on Application Server

4.6 Advanced Communication Function Implementation

4.6.1 SMS Messaging

4.6.2 Files Sharing

4.6.3 Google Contacts Import

Chapter 5
Prototype System Deployment

Chapter 6

Future Work

6.1 RTCDataChannel usage

6.2 Browser Compatibility

6.3 Media Server Performance

6.4 Object RTC (ORTC) API for WebRTC

6.5 Advanced function for telecommunication

References

[Abo14] Feross Aboukhadijeh. Webtorrent - streaming torrent client for node and the browser, 2014. [Online; accessed 9-May-2014].

[ang14] angularjs.org. $q - service in module ng, 2014. [Online; accessed 23-May-2014].

[Bah14] Bruce Bahlmann. Slee - service logic execution environment, 2014. [Online; accessed 20-May-2014].

[Blo14] The Chromium Blog. Play cube slam, a real-time webrtc video game, 2014. [Online; accessed 8-May-2014].

[Cru14a] Crunchbase. Tropo, 2014. [Online; accessed 8-May-2014].

[Cru14b] Crunchbase. Uberconference, 2014. [Online; accessed 8-May-2014].

[Dut14] Sam Dutton. Getting started with webrtc — html5rocks, 2014. [Online; accessed 2-May-2014].

[Goo12] Google. General overview — webrtc.org, 2012. [Online; accessed 7-May-2014].

[Inc05] Cisco Systems Inc. Differences between traditional telephony and voip, 2005. [Online; accessed 2-May-2014].

[Iss14] Dart Issues. Issue 15008: Rtcpeerconnection.addicecandidate results in a not-supportederror: Internal dartium exception (webrtc/dartium), 2014. [Online; accessed 15-May-2014].

[JB13a] Alan B Johnston and Daniel C Burnett. *WebRTC: APIs and RTCWEB Protocols of the HTML5 Real-Time Web*, chapter Preface, page 12. Digital Codex LLC, second edition, 2013.

[JB13b] Alan B Johnston and Daniel C Burnett. *WebRTC: APIs and RTCWEB Protocols of the HTML5 Real-Time Web*, chapter How to Use WebRTC, page 33. Digital Codex LLC, second edition, 2013.

[JB13c] Alan B Johnston and Daniel C Burnett. *WebRTC: APIs and RTCWEB Protocols of the HTML5 Real-Time Web*, chapter How to Use WebRTC, page 48. Digital Codex LLC, second edition, 2013.

[JB13d] Alan B Johnston and Daniel C Burnett. *WebRTC: APIs and RTCWEB Protocols of the HTML5 Real-Time Web*, chapter Introduction to Web Real-Time Communications, page 15. Digital Codex LLC, second edition, 2013.

[kir14] kirm. sip.js — github, 2014. [Online; accessed 20-May-2014].

[Net14] Mozilla Developer Network. Peer-to-peer communications with webrtc, 2014. [Online; accessed 12-May-2014].

[pro14] promisejs.org. What is a promise?, 2014. [Online; accessed 22-May-2014].

[Rot14] Issac Roth. What makes node.js faster than java?, 2014. [Online; accessed 20-May-2014].

[Soc02] The Internet Society. Sip: Session initiation protocol — rfc 3261, 2002. [Online; accessed 7-May-2014].

[Tel14a] Doubango Telecom. sipml5.org, 2014. [Online; accessed 15-May-2014].

[Tel14b] Doubango Telecom. webrtc2sip.org, 2014. [Online; accessed 15-May-2014].

[Tel14c] Telestax. sipservlets - leading html5 webrtc compliant sip/ims application server, 2014. [Online; accessed 20-May-2014].

[vis14] visionmedia. Jade — github, 2014. [Online; accessed 22-May-2014].

[Web14a] The Next Web. Uberconference turns google hangouts into a conference calling system, 2014. [Online; accessed 8-May-2014].

[Web14b] Webopedia. Centralized network, 2014. [Online; accessed 22-May-2014].

[Web14c] Webopedia. Pbx - private branch exchange — webopedia, 2014. [Online; accessed 2-May-2014].

[Wik13a] Wikipedia. Apache tomcat — Wikipedia, the free encyclopedia, 2013. [Online; accessed 20-May-2014].

[Wik13b] Wikipedia. Cascading style sheets — Wikipedia, the free encyclopedia, 2013. [Online; accessed 22-May-2014].

[Wik13c] Wikipedia. Java apis for integrated networks — Wikipedia, the free encyclopedia, 2013. [Online; accessed 20-May-2014].

[Wik13d] Wikipedia. Mesh networking — Wikipedia, the free encyclopedia, 2013. [Online; accessed 22-May-2014].

[Wik13e] Wikipedia. Oauth — Wikipedia, the free encyclopedia, 2013. [Online; accessed 22-May-2014].

[Wik13f] Wikipedia. Wildfly — Wikipedia, the free encyclopedia, 2013. [Online; accessed 20-May-2014].

[Wik14a] Wikipedia. Angularjs — Wikipedia, the free encyclopedia, 2014. [Online; accessed 12-May-2014].

[Wik14b] Wikipedia. Apache http server — Wikipedia, the free encyclopedia, 2014. [Online; accessed 20-May-2014].

[Wik14c] Wikipedia. Asp.net — Wikipedia, the free encyclopedia, 2014. [Online; accessed 20-May-2014].

[Wik14d] Wikipedia. Dart (programming language) — Wikipedia, the free encyclopedia, 2014. [Online; accessed 15-May-2014].

[Wik14e] Wikipedia. Datagram transport layer security — Wikipedia, the free encyclopedia, 2014. [Online; accessed 13-May-2014].

[Wik14f] Wikipedia. Google chrome — Wikipedia, the free encyclopedia, 2014. [Online; accessed 12-May-2014].

[Wik14g] Wikipedia. Google contacts — Wikipedia, the free encyclopedia, 2014. [Online; accessed 15-May-2014].

[Wik14h] Wikipedia. Google hangouts — Wikipedia, the free encyclopedia, 2014. [Online; accessed 8-May-2014].

[Wik14i] Wikipedia. Google voice — Wikipedia, the free encyclopedia, 2014. [Online; accessed 8-May-2014].

[Wik14j] Wikipedia. Groovy (programming language) — Wikipedia, the free encyclopedia, 2014. [Online; accessed 8-May-2014].

[Wik14k] Wikipedia. H.323 — Wikipedia, the free encyclopedia, 2014. [Online; accessed 7-May-2014].

[Wik14l] Wikipedia. Interactive connectivity establishment — Wikipedia, the free encyclopedia, 2014. [Online; accessed 2-May-2014].

[Wik14m] Wikipedia. Javascript — Wikipedia, the free encyclopedia, 2014. [Online; accessed 8-May-2014].

[Wik14n] Wikipedia. List of sip response codes — Wikipedia, the free encyclopedia, 2014. [Online; accessed 7-May-2014].

[Wik14o] Wikipedia. Node.js — Wikipedia, the free encyclopedia, 2014. [Online; accessed 20-May-2014].

[Wik14p] Wikipedia. Php — Wikipedia, the free encyclopedia, 2014. [Online; accessed 8-May-2014].

[Wik14q] Wikipedia. Public switched telephone network — Wikipedia, the free encyclopedia, 2014. [Online; accessed 2-May-2014].

[Wik14r] Wikipedia. Python (programming language) — Wikipedia, the free encyclopedia, 2014. [Online; accessed 8-May-2014].

[Wik14s] Wikipedia. Ruby (programming language) — Wikipedia, the free encyclopedia, 2014. [Online; accessed 8-May-2014].

[Wik14t] Wikipedia. Session description protocol — Wikipedia, the free encyclopedia, 2014. [Online; accessed 13-May-2014].

[Wik14u] Wikipedia. Session initiation protocol — Wikipedia, the free encyclopedia, 2014. [Online; accessed 7-May-2014].

[Wik14v] Wikipedia. Skype — Wikipedia, the free encyclopedia, 2014. [Online; accessed 2-May-2014].

[Wik14w] Wikipedia. Socket.io — Wikipedia, the free encyclopedia, 2014. [Online; accessed 16-May-2014].

[Wik14x] Wikipedia. Tor (anonymity network) — Wikipedia, the free encyclopedia, 2014. [Online; accessed 12-May-2014].

[Wik14y] Wikipedia. Webrtc — Wikipedia, the free encyclopedia, 2014. [Online; accessed 29-April-2014].

[Wik14z] Wikipedia. Websocket — Wikipedia, the free encyclopedia, 2014. [Online; accessed 7-May-2014].

[Wor04] Network World. What is sip? — network world, 2004. [Online; accessed 7-May-2014].

Appendix A

A.1 Application Script

Code Snippet A.1: app.js in application client

```
'use strict';

// Declare app level module which depends on filters, and services

angular.module('webrtcDemo', [
  'ui.bootstrap',
  'btford.socket-io',
  'angularLocalStorage',
  'angular-underscore',
  'ngRoute',
  'ui.keypress',
  'angularMoment',
  'angularFileUpload',
  'angular-table',
  'angucomplete-alt',
  'webrtcDemo.services.values',
  'webrtcDemo.services',
  'webrtcDemo.controllers',
  'webrtcDemo.directives',
  'webrtcDemo.filters'
]).
config(function ($routeProvider, $locationProvider, $httpProvider) {
  $routeProvider.
    when('/chat', {
      templateUrl: 'partials/phoneView',
      controller: 'PhoneViewCtrl'
    }).
```

```
    when('/login',{
      templateUrl: 'partials/login',
      controller: 'LoginViewCtrl'
    }).
    otherwise({
      redirectTo: '/login'
    });

  $locationProvider.html5Mode(true);

  $httpProvider.defaults.useXdomain = true;
  delete $httpProvider.defaults.headers.common['
      X-Requested-With'];
  $httpProvider.defaults.withCredentials = true;
});

angular.module('webrtcDemo.services', ['
    webrtcDemo.services.values']);

angular.module('webrtcDemo.controllers',['
    webrtcDemo.services.values']);

angular.module('webrtcDemo.directives',['
    webrtcDemo.services.values']);

angular.module('webrtcDemo.filters',['
    webrtcDemo.services.values']);
```

A.2 WebRTCService Script

Code Snippet A.2: WebRTCService.js in application client

```
'use strict';

/**
*  services Module
*
* WebRTCService with browser adapter.js function
*/

angular.module('webrtcDemo.services').
  factory('WebRTCService',function () {
    var _ws;//websocket obj

    var _RTCPeerConnection;
    var _RTCSessionDescription;
```

```
    var _RTCIceCandidate;
    var _getUserMedia;
    var _createIceServer;
    var _attachMediaStream;
    var _reattachMediaStream;
    var _webrtcDetectedBrowser;
    var _webrtcDetectedVersion;

    function _initWebRTC () {
      _RTCPeerConnection = null;
      _RTCSessionDescription = null;
      _RTCIceCandidate = null;
      _getUserMedia = null;
      _createIceServer = null;
      _attachMediaStream = null;
      _reattachMediaStream = null;
      _webrtcDetectedBrowser = null;
      _webrtcDetectedVersion = null;

      _setRTCElement();
    }

    function _setRTCElement() {

      if(navigator.mozGetUserMedia){
        console.log("This appears to be Firefox");

        _webrtcDetectedBrowser = "firefox";
        _webrtcDetectedVersion = parseInt(
            navigator.userAgent.match(/Firefox\/([0-9]+)\./)
            [1], 10);

        _RTCPeerConnection = mozRTCPeerConnection;
        _RTCSessionDescription = mozRTCSessionDescription;
        _RTCIceCandidate = mozRTCIceCandidate;
        _getUserMedia = navigator.mozGetUserMedia.bind(
            navigator);

        // Creates iceServer from the url for FF.
        _createIceServer = function(url, username, password)
             {
          var iceServer = null;
          var url_parts = url.split(':');
          if (url_parts[0].indexOf('stun') === 0) {
            // Create iceServer with stun url.
```

```
    iceServer = { 'url': url };
  } else if (url_parts[0].indexOf('turn') === 0) {
    if (_webrtcDetectedVersion < 27) {
      // Create iceServer with turn url.
      // Ignore the transport parameter from TURN url for FF version <=27.
      var turn_url_parts = url.split("?");
      // Return null for createIceServer if transport=tcp.
      if (turn_url_parts.length === 1 ||
          turn_url_parts[1].indexOf('transport=udp') === 0) {
        iceServer = { 'url': turn_url_parts[0],
                      'credential': password,
                      'username': username };
      }
    } else {
      // FF 27 and above supports transport parameters in TURN url,
      // So passing in the full url to create iceServer.
      iceServer = { 'url': url,
                    'credential': password,
                    'username': username };
    }
  }
  return iceServer;
};

_attachMediaStream = function(element, stream) {
  console.log("Attaching media stream");
  element.mozSrcObject = stream;
  element.play();
};

_reattachMediaStream = function(to, from) {
  console.log("Reattaching media stream");
  to.mozSrcObject = from.mozSrcObject;
  to.play();
};

// Fake get{Video,Audio}Tracks
if (!MediaStream.prototype.getVideoTracks) {
  MediaStream.prototype.getVideoTracks = function() {
```

```
          return [];
        };
      }

      if (!MediaStream.prototype.getAudioTracks) {
        MediaStream.prototype.getAudioTracks = function() {
          return [];
        };
      }

    }else if(navigator.webkitGetUserMedia){
      console.log("This appears to be Chrome");

      _webrtcDetectedBrowser = "chrome";
      _webrtcDetectedVersion = parseInt(navigator.userAgent.match(/Chrom(e|ium)\/([0-9]+)\./)[2], 10);

      _RTCPeerConnection = webkitRTCPeerConnection;
      _RTCSessionDescription = RTCSessionDescription;
      _RTCIceCandidate = RTCIceCandidate;
      _getUserMedia = navigator.webkitGetUserMedia.bind(navigator);

      // Creates iceServer from the url for Chrome.
      _createIceServer = function(url, username, password) {
        var iceServer = null;
        var url_parts = url.split(':');
        if (url_parts[0].indexOf('stun') === 0) {
          // Create iceServer with stun url.
          iceServer = { 'url': url };
        } else if (url_parts[0].indexOf('turn') === 0) {
          if (_webrtcDetectedVersion < 28) {
            // For pre-M28 chrome versions use old TURN format.
            var url_turn_parts = url.split("turn:");
            iceServer = { 'url': 'turn:' + username + '@' + url_turn_parts[1],
                          'credential': password };
          } else {
            // For Chrome M28 & above use new TURN format.
            iceServer = { 'url': url,
                          'credential': password,
```

```
                              'username': username };
          }
        }
        return iceServer;
      };

      // Attach a media stream to an element.
      _attachMediaStream = function(element, stream) {
        if (typeof element.srcObject !== 'undefined') {
          element.srcObject = stream;
        } else if (typeof element.mozSrcObject !== 'undefined') {
          element.mozSrcObject = stream;
        } else if (typeof element.src !== 'undefined') {
          element.src = URL.createObjectURL(stream);
        } else {
          console.log('Error attaching stream to element.');
        }
      };

      _reattachMediaStream = function(to, from) {
        to.src = from.src;
      };

      // The representation of tracks in a stream is changed in M26
      // Unify them for earlier Chrome versions in the coexisting period
      if (!webkitMediaStream.prototype.getVideoTracks) {
        webkitMediaStream.prototype.getVideoTracks = function() {
          return this.videoTracks;
        };
        webkitMediaStream.prototype.getAudioTracks = function() {
          return this.audioTracks;
        };
      }

      // New syntax of getXXXStreams method in M26
      if (!webkitRTCPeerConnection.prototype.getLocalStreams) {
```

```
          webkitRTCPeerConnection.prototype.getLocalStreams
              = function() {
            return this.localStreams;
          };
          webkitRTCPeerConnection.prototype.getRemoteStreams
              = function() {
            return this.remoteStreams;
          };
        }

      }else{
        console.log("Browser does not appear to be
            WebRTC-capable");
      }

    }

    return {
      init : function (socket) {
        if(socket){
          //init service with websocket
          _ws = socket;
        }

        _initWebRTC();

      },

      peerConnection : function(config,constraints){
        if(_RTCPeerConnection){
          return new _RTCPeerConnection(config,constraints);
        }
        return null;
      },

      RTCSessionDescription : function(message){
        if(_RTCSessionDescription){
          return new _RTCSessionDescription(message);
        }
        return null;
      },

      RTCIceCandidate : function(options){
        if(_RTCIceCandidate){
            return new _RTCIceCandidate(options);
```

```
          }
          return null;
      },

      webrtcDetectedBrowser : function(){
        return _webrtcDetectedBrowser;
      },

      attachMediaStream : function(element, stream){
        return _attachMediaStream(element, stream);
      },

      reattachMediaStream : function(to, from){
        return _reattachMediaStream(to, from);
      },

      getUserMedia : function(constraints, handleUserMedia,
          handleUserMediaError){
        return _getUserMedia(constraints, handleUserMedia,
            handleUserMediaError);
      }

    }
  });
```

A.3 ContactTable Scripts

Code Snippet A.3: contactTable.jade in application client

```
div(id = "contactTable")
  angucomplete-alt(id="contactSearch",
    place-holder="Search Contact Name",
    pause="100",
    selected-object="selectedContact",
    local-data="contactsHolder.contacts",
    search-fields="name",
    title-field="name",
    description-field="number"
    minlength="1",
    input-class="form-control form-control-small")
  tabset
    tab(heading = "Conacts")
      table(id = "contacts", at-table, at-paginated,
          at-list="contactsHolder.contacts | orderBy:online"
          , at-config="config",class="table table-hover
          table-striped table-condensed" )
```

```
        thead
        tbody
          tr(ng-class = "{success: item.online}", ng-init = "item.hvor = false", ng-mouseenter = "contactHvor(item)", ng-mouseleave = "contactHvor(item)")
            td(at-implicit, at-sortable, at-attribute="name", width="250")
            td(at-sortable, at-attribute="online", at-title="Status", width="200", at-initial-sorting="desc")
              p(ng-hide = "item.hvor").
                {{item.online | iif : "Online" : "Offline"}}
              div(class="btn-group contactCallBtn", ng-show = "item.hvor")
                button(type="button", class="btn btn-success", ng-click = "callContact(item)").
                  {{item.online | iif: "Video Call" : "Call"}}
                button(type="button", class="btn btn-success dropdown-toggle", data-toggle="dropdown")
                  span(class="caret")
                ul(class="dropdown-menu", role="menu")
                  li
                    a(ng-click="") Instance Message
                  li
                    a(ng-click="") SMS
                  li(class="divider")
                  li
                    a(ng-click="") Edit Contact...
            td(at-title="Description")
              p
                | Telephone : {{item.number}}
      at-pagination(at-list="contactsHolder.contacts" at-config="config")
    tab(heading = "Online")
      table(id = "onlines", at-table, at-paginated, at-list="onlines", at-config="config", class="table table-hover table-striped table-condensed")
        thead
        tbody
          tr(ng-init = "item.hvor = false", ng-mouseenter = "contactHvor(item)", ng-mouseleave = "
```

```
                 contactHvor(item)")
            td(at-implicit, at-sortable, at-attribute="name", width="250", at-initial-sorting="asc")
            td(at-title="Status", width="200")
              p(ng-hide = "item.hvor").
                {{item.online | iif : "Online" : "Offline" }}
              div(class="btn-group contactCallBtn", ng-show = "item.hvor")
                button(type="button", class="btn btn-success", ng-click = "callContact(item)").
                  {{item.online | iif: "Video Call" : "Call" }}
                button(type="button", class="btn btn-success dropdown-toggle", data-toggle="dropdown")
                  span(class="caret")
                ul(class="dropdown-menu", role="menu")
                  li
                    a(ng-click="") Instance Message
                  li
                    a(ng-click="") SMS
                  li(class="divider")
                  li
                    a(ng-click="") Edit Contact...
            td(at-title="Description")
              p
                | Telephone : {{item.number}}
      at-pagination(at-list="onlines" at-config="config")
```

Code Snippet A.4: ContactTableDirective.js in application client

```
'use strict';


/**
*  Directives Module
*
* Contact Table Directive
*/
angular.module('webrtcDemo.directives').
  directive('contactTable',function () {

    return{
      restrict: 'E',
      replace: true,
```

```
      scope: true,
      templateUrl: 'partials/contactTable',
      controller: 'ContactsCtrl'
    };

  });

angular.module('webrtcDemo.filters').
  filter('iif', function () {
   return function(input, trueValue, falseValue) {
        return input ? trueValue : falseValue;
  };
});
```

A.4 ContactsCtrl Script

Code Snippet A.5: ContactsCtrl.js in application client

```
'use strict';


/**
*  Controllers Module
*
* Contacts Controller
*/
angular.module('webrtcDemo.controllers').
  controller('ContactsCtrl',function ($scope,$location,
     WebSocketService,GoogleAPIService,storage,$filter) {
    console.log('init contacts controller');

    $scope.selectedContact;

    var username = storage.get('current-user').name;
    _initContactsView();

    function _initContactsView(){

      $scope.$watch('selectedContact',function(newVal,oldVal
         ){
        if(newVal && newVal != oldVal){
          console.log($scope.selectedContact);
          $scope.outPhone.number =
              $scope.selectedContact.originalObject.number;
        }
      });
```

```

    if(storage.get('contactList-' + username)){
      $scope.contactsHolder.contacts = storage.get('contactList-' + username);
    }

    _initContactsTable();
  }

  function _initContactsTable(){

    $scope.config = {
      itemsPerPage: 13,
      fillLastPage: "yes"
    };
  }

  $scope.callContact = function(contact){
    console.log('pick number: ' + contact.number);
    $scope.outPhone.number = contact.number;
    $scope.callNumber();
  }

  $scope.contactHvor = function(contact){
    contact.hvor = ! contact.hvor;
  }

  $scope.contactListHvor = function(contact){
    contact.listHvor = ! contact.listHvor;
  }

  $scope.importContacts = function(){
    $scope.contactsHolder.contacts = [];
    GoogleAPIService.oAuth().then(function(token){
      GoogleAPIService.queryContacts(token).then(function(data){
        angular.forEach(data.feed.entry,function(person, key){
          if(person['gd$phoneNumber']){
            var contact = {
              name: person.title['$t'],
              number: person['gd$phoneNumber'][0]['$t'],
              online: false
            }

```

```
              if($scope.onlines){
                var online = _.find($scope.onlines, function (c){
                  return c.number == contact.number;
                });

                if(online){
                  contact.online = true;
                }
              }

              $scope.contactsHolder.contacts.push(contact);
            }
          });

          storage.set('contactList-' + username, $scope.contactsHolder.contacts);

        });
      });
    }

  });
```

A.5 GoogleAPIService Script

Code Snippet A.6: GoogleAPIService.js in application client

```
'use strict';



/**
*  services Module
*
*  Google API Service
*/

angular.module('webrtcDemo.services').
  factory('GoogleAPIService', function ($q,$http,storage) {

    function _authLogin(){
      var deferred = $q.defer();

      var config = {
```

```
          'client_id': 'xxxxxxxxxxxxxxx.apps.googleusercontent.com',
          'scope': 'https://www.google.com/m8/feeds'
        };
      gapi.auth.authorize(config, function() {

        console.log('login complete');
        console.log(gapi.auth.getToken());
        deferred.resolve(gapi.auth.getToken());

      });

      return deferred.promise;
    }

    function _fetchContacts(authToken){
      var deferred = $q.defer();

      var url = 'https://www.google.com/m8/feeds/contacts/default/full?access_token=' + authToken.access_token + '&alt=json&max-results=500&callback=JSON_CALLBACK';

      $http.jsonp(url).
      success(function(data, status, headers, config) {
        deferred.resolve(data);
      }).
      error(function(data, status, headers, config) {
        deferred.reject('GoogleAPIService:queryContacts:Failed');
      });

      return deferred.promise;
    }

    return{

      oAuth: function(){

        return _authLogin();

      },

      queryContacts: function(authToken){

```

```
          return _fetchContacts(authToken);

        }

      }

    });
```

A.6 PhoneViewCtrl Script

Code Snippet A.7: _setSocketListener() Function in PhoneViewCtrl.js on Application Client

```
function _setSocketListener(socket){

      socket.on('log', function (array){
        console.log.apply(console, array);
      });

      socket.on('webrtc', function (data){
        console.log('Received webrtc message:', data);
        switch(data.type){
          case 'register':
            if(data.msg == 'success'){
              offer2xms = false;
              if(data.colleagues.length != 0){
                _.map(data.colleagues,function(c){
                  var contact = _.find(
                      $scope.contactsHolder.contacts,
                      function(contact){
                    return contact.number == c.number;
                  });
                  if(contact){
                    contact.online = true;
                    $scope.onlines.push(contact);
                  }
                });
              }
              _maybeStart();
            }
            break;
          case 'online':
            $scope.contactsHolder.contacts = _.map(
                $scope.contactsHolder.contacts,function(c){
              if(c.number == data.user.number){
                c.online = true;
```

```
                $scope.onlines.push(c);
              }
              return c;
            })
            break;
          case 'join conference':

            break;
          case 'disconnect':
            $scope.contactsHolder.contacts = _.map(
                $scope.contactsHolder.contacts,function(c){
              if(c.number == data.user.number){
                c.online = false;
                var i = $scope.onlines.indexOf(c);
                if(i != -1) {
                  $scope.onlines.splice(i, 1);
                }
                //$scope.onlines.pop(c);
              }
              return c;
            })
            break;
          case 'offer':
            /*
            if (!isStarted) {
              _maybeStart();
            }
            pc.setRemoteDescription(
                WebRTCService.RTCSessionDescription(data));
            _doAnswer();
            */
            break;
          case 'answer':
            if(isStarted){
              if(data.sdp){
                pc.setRemoteDescription(
                    WebRTCService.RTCSessionDescription(data
                    ));
              }
              if(!data.self){
                socket.emit('sip',{
                  type: 'invite',
                  username: $scope.user.name,
                  content: {
                    to: $scope.outPhone.number
```

```
                }
              });
            }else{
              var channel;
              if(callDirection === 'outbound'){
                channel = 'sip';
              }else{
                channel = 'webrtc';
              }

              socket.emit(channel,{
                type: 'answerInvite',
                username: $scope.user.name,
                conf_id: conf_id
              });

              //show answer process notification
              $rootScope.$broadcast('answer-call',{number
                  : $scope.toAnswerPhone.number});

            }

          }
          break;
        case 'candidate':
          /*
          if(isStarted){
            var candidate = WebRTCService.RTCIceCandidate
                ({
              sdpMLineIndex:data.content.label,
              sdpMid:data.content.id,
              candidate:data.content.candidate
            });
            pc.addIceCandidate(candidate);
          }
          */
          break;
        case 'bye':
          if(isStarted){
            _handleRemoteHangup();
          }
          break;
        case 'createRTCoffer':
          callDirection = data.callDirection;
          conf_id = data.conf_id;
```

```
                break;
              default:
                break;
            }
          });

          socket.on('sip',function(data){
            console.log('Received sip message:',data);
            switch(data.type){
              case 'createRTCoffer':
                callDirection = data.callDirection;
                break;
              case 'busy':

                break;
              default:
                break;
            }
          });
        }
```

Appendix B

B.1 Socket.IO Script

Code Snippet B.1: socket.js on Application Server

```
SocketManager.prototype.listen = function(server){
  var self = this;
  clients = {};
  sipClients = {};
  callRequests = {};
  conversations = {};

  io = socketio.listen(server);

  io.sockets.on('connection', _handlerSocket);

  _handlerSip();
}

function _handlerSocket(socket) {
  var delivery = dl.listen(socket);

  // convenience function to log server messages on the client
  function log(){
    var array = [">>>"];
    for (var i = 0; i < arguments.length; i++) {
      array.push(arguments[i]);
    }
      socket.emit('log', array);
  }

  socket.on('sip',function (data){
    log('Got sip message:', data);
```

```
    switch(data.type){
      case 'register':
        if(data.username != ""){
          gw.register(data.content.browserClient,function(result){
            socket.emit('sip',result);
          });
        }
        break;
      case 'invite':
        var callerClient = clients[data.username];
        var calleeClient = clients[data.content.to];

        if(!callerClient.inConference){
          xmsManager.createConference({
            type: "audiovideo",
            max_p: 9,
            reserve: 0,
            layout: 0,
            caption: "yes"
          },function(conf_id){
            callerClient.conf_id = conf_id;
            conversations[conf_id] = _createConversation(conf_id);

            xmsManager.joinConference(callerClient.local_identifier,conf_id,{
              caption: "webrtc: " + data.username,
              region: 0,
              audio: "sendrecv",
              video: "sendrecv"
            },function(connected){
              console.log(data.username + ' joined conference : ' + connected);
              callerClient.inConference = true;
              conversations[conf_id].inboundContacts.push(data.username);
              socket.join(conf_id.toString());
            });

            if(calleeClient){
              callerClient.socket.emit('webrtc',{
                type: "ringing",
                number: calleeClient.number
```

```
              });

              calleeClient.callreq_id = callerClient.callreq_id;
              callRequests[callerClient.callreq_id].callee = calleeClient;

              calleeClient.socket.emit('webrtc',{
                type: "createRTCoffer",
                inComingNumber: data.username,
                callDirection: 'inbound',
                conf_id: conf_id
              });
            }else{
              _inviteOutboundCall(data.username, data.content.to);
            }

          });

        }else{
          if(calleeClient){
            callerClient.socket.emit('webrtc',{
              type: "ringing",
              number: calleeClient.number
            });

            calleeClient.callreq_id = callerClient.callreq_id;
            callRequests[callerClient.callreq_id].callee = calleeClient;

            calleeClient.socket.emit('webrtc',{
              type: "createRTCoffer",
              inComingNumber: data.username,
              callDirection: 'inbound',
              conf_id: callerClient.conf_id
            });
          }else{
            _inviteOutboundCall(data.username, data.content.to);
          }

        }
        break;
```

```
      case 'answerInvite':
        var client = clients[data.username];
        var req = callRequests[client.callreq_id];

        if(!client.inConference){
          xmsManager.createConference({
            type: "audiovideo",
            max_p: 9,
            reserve: 0,
            layout: 0,
            caption: "yes"
          },function(conf_id){
            client.conf_id = conf_id;
            conversations[conf_id] = _createConversation(
                conf_id);

            xmsManager.joinConference(
                client.local_identifier,conf_id,{
              caption: "webrtc: " + data.username,
              region: 0,
              audio: "sendrecv",
              video: "sendrecv"
            },function(connected){
              console.log(data.username + ' joined
                  conference : ' + connected);
              client.inConference = true;
              conversations[conf_id].inboundContacts.push(
                  data.username);
              socket.join(conf_id.toString());
            });

            xmsManager.joinConference(
                req.caller.remote_identifier,conf_id,{
              caption: "sip: " + unq(req.caller.number),
              region: 0,
              audio: "sendrecv",
              video: "inactive"
            },function(connected){
              console.log(req.caller.number + ' joined
                  conference : ' + connected);
              req.caller.conf_id = conf_id;
              conversations[conf_id].outboundContacts.push(
                  req.caller.number);
              callRequests[client.callreq_id].end = true;
            });
```

```

          });
        }else{

          xmsManager.joinConference(req.caller.remote_identifier,client.conf_id,{
            caption: "sip: " + unq(req.caller.number),
            region: 0,
            audio: "sendrecv",
            video: "inactive"
          },function(connected){
            console.log(req.caller.number + ' joined conference : ' + connected);
            req.caller.conf_id = client.conf_id;
            conversations[client.conf_id].outboundContacts.push(req.caller.number);
            callRequests[client.callreq_id].end = true;
          });

        }
        break;
      default:
        break;
    }
  });

  socket.on('webrtc', function (data) {
    log('Got webrtc message:', data);

    switch(data.type){
      case 'answerInvite':
        var client = clients[data.username];

        xmsManager.joinConference(client.local_identifier,data.conf_id,{
          caption: "webrtc: " + data.username,
          region: 0,
          audio: "sendrecv",
          video: "sendrecv"
        },function(connected){
          console.log(data.username + ' joined conference : ' + connected);
          client.inConference = true;
          client.conf_id = data.conf_id;
```

```
          conversations[data.conf_id].inboundContacts.push(data.username);
          socket.join(data.conf_id.toString());
          socket.broadcast.to(data.conf_id.toString()).emit('webrtc',{
            type: 'join conference',
            colleague: client.username,
            conf_id: data.conf_id
          });
          callRequests[client.callreq_id].end = true;
        });
        break;
      case 'answer':
      /*
        clients[data.username].role = 'callee';
        var call = clients[data.username];

        if(call.endIceCandidate){
          xmsManager.updateLocalSDP(data.content.sdp,call,function(){

            xmsManager.joinXMSCall(call,function(rs){
              console.log("join call response: \n" + rs);

              gw.sendAnswer(data.username,
                call.sipInviteRequest,
                call.remote_xmsSDP,
                function(){
                  console.log('200 ok answer sent');
                });

            });
          });
        }
      */
        break;
      case 'offer':
        var client = clients[data.username];

        if(!data.content.self){
          var id = uuid.v1();
          client.callreq_id = id;

          callRequests[id] = _createCallRequest(id);
          callRequests[id].caller = client;
```

```

        }else{

          /*
          gw.sendAnswer(data.username,
            clients[data.username].sipInviteRequest,
            clients[data.username].remote_xmsSDP,
            function(){
              console.log('200 ok answer sent');
          });
          */
        }

        if(client.inConference){
          socket.emit('webrtc',{
            type: "answer",
            self: data.content.self
          });
        }

        break;
      case 'register':
        clients[data.username] = {};
        _resetClient(data.username,data.host,socket,delivery
            );

        socket.join(data.host);

        var colleagues = _und.map(clients,function(client,
            key){
          if(client.host == data.host && key !=
              data.username){
            return {
              name: client.username,
              number: client.number,
              online: true
            };
          }
        });

        colleagues = _und.filter(colleagues,function(client)
            {
          return client != null;
        });

```

```
        socket.emit('webrtc',{
          type: 'register',
          msg: 'success',
          colleagues: colleagues
        });

        socket.broadcast.to(data.host).emit('webrtc', {
          type: 'online',
          user: {
            name: data.username,
            number: clients[data.username].number,
            online: true
          }
        });

        break;
      case 'endCandidate':
        var client = clients[data.username];
        client.endIceCandidate = true;

        var role = callRequests[client.callreq_id].
            caller.number == data.username ? 'caller' : '
            callee';

        console.log('webrtc:endCandidate: ' + role + ' sdp:
            ' + data.content.sdp);

        if(client.local_xmsSDP == '' ||
            client.local_identifier == ''){
            xmsManager.createXMSCall({
            callType: 'webrtc',
            sdp: data.content.sdp
          },function(xmsSdp,id){
            client.local_xmsSDP = xmsSdp;
            client.local_identifier = id;

            socket.emit('webrtc',{
              type: "answer",
              sdp: xmsSdp,
              self: data.content.self
            });
          });
        }else{
          xmsManager.updateLocalSDP(data.content.sdp,client,
              function(sdp){
```

```
            client.local_xmsSDP = sdp;
            socket.emit('webrtc',{
              type: "answer",
              sdp: sdp,
              self: data.content.self
            });
          });
        }

        break;
      case 'candidate':

        break;
      case 'hangup':

        try{
          var client = clients[data.username];
          var numberClient = clients[data.pairNumber] ? clients[data.pairNumber] : sipClients[data.pairNumber];

          if(!numberClient){

            gw.sendCancel(data.username,data.pairNumber,function(){
              console.log(data.username + 'cancel the call');

              callRequests[client.call_req].end = true;

            });

            var conf = conversations[client.conf_id];

            if(!client.inConference || conf.inboundContacts + conf.outboundContacts <= 1){

              xmsManager.deleteXMSCall(client.local_identifier,function(rs){
                if(rs.statusCode === 204){
                  console.log('hangup: ' + client.username);
                  client.socket.emit('webrtc',{ type: 'bye' });

```

```
                  _resetClient(client.username,client.host,
                      client.socket,client.delivery);

                }
              });

            }

          }else{

            if(numberClient.type === 'webrtc' && data.status
                === 'on'){
              var conf = conversations[client.conf_id];
              var i = conf.inboundContacts.indexOf(
                  client.username);
              if(i != -1) {
                conf.inboundContacts.splice(i, 1);
              }
              //conf.inboundContacts.pop(client.username);

              xmsManager.deleteXMSCall(
                  client.local_identifier,function(rs){
                if(rs.statusCode === 204){
                  console.log('hangup: ' + client.username);
                  client.socket.emit('webrtc',{ type: 'bye'
                      });

                  try{
                    if(conf.inboundContacts.length +
                        conf.outboundContacts.length > 1){
                      socket.broadcast.to(client.conf_id).
                          emit('webrtc',{
                        type: 'forward',
                        number: client.number,
                        newNumber:
                            conf.outboundContacts.length !=
                            0 ? conf.outboundContacts[0] :
                            conf.inboundContacts[0],
                        webrtcOnly:
                            conf.outboundContacts.length ==
                            0 ? true : false
                      });
                    }
                  }catch(e){
```

```
                    console.log('RUNTIME ERROR:webrtc:hangup:forward: ', e);
                  }

                  _resetClient(client.username,client.host,client.socket,client.delivery);

                }
              });

              if(conf.outboundContacts.length == 0 && conf.inboundContacts.length == 1){
                xmsManager.deleteXMSCall(numberClient.local_identifier,function(rs){
                  if(rs.statusCode === 204){
                    console.log('hangup to delete: ' + numberClient.username);
                    numberClient.socket.emit('webrtc',{
                      type: 'bye',
                      number: client.number
                    });
                    _resetClient(numberClient.username,numberClient.host,numberClient.socket,numberClient.delivery);
                    var i = conf.inboundContacts.indexOf(numberClient.username);
                    if(i != -1) {
                      conf.inboundContacts.splice(i, 1);
                    }
                    //conf.inboundContacts.pop(numberClient.username);
                    xmsManager.deleteXMSConference(conf.id,function(rs){
                      if(rs.statusCode === 204){
                        console.log('webrtc:hangup:conference resource: ' + conf.id + ' delete success.');
                        delete conf;
                      }
                    });
                  }
                });
```

```
            }else if(conf.outboundContacts.length == 1 && conf.inboundContacts.length == 0){
              var sipNo = conf.outboundContacts[0];
              var sipClient = sipClients[sipNo];

              xmsManager.deleteXMSCall(sipClient.remote_identifier,function(rs){
                if(rs.statusCode === 204){
                  var i = conf.outboundContacts.indexOf(sipClient.number);
                  if(i != -1) {
                    conf.outboundContacts.splice(i, 1);
                  }
                  //conf.outboundContacts.pop(sipClient.number);

                  var call_req = _und.find(callRequests, function(q){//TODO: first one match bug
                    if(q.caller.type == 'webrtc'){
                      return q.callee.number == sipClient.number;
                    }else{
                      return q.caller.number == sipClient.number;
                    }
                  });

                  var pair = call_req.caller.type == 'webrtc' ? call_req.caller : call_req.callee;

                  gw.sendBye(pair.username,function(){
                    console.log('bye to number: ' + sipClient.number);
                  });

                  xmsManager.deleteXMSConference(conf.id, function(rs){
                    if(rs.statusCode === 204){
                      console.log('webrtc:hangup:conference resource: ' + conf.id + ' delete success.');
```

```
                      delete conf;
                    }
                  });
                }
              });
            }

          }else if(numberClient.type === 'webrtc' && data.status === 'off'){
            console.log('busy: ' + client.username);

            var conf = conversations[numberClient.conf_id];

            numberClient.socket.emit('webrtc',{
              type: 'busy',
              source: client.username
            });

            if(conf.outboundContacts.length == 0 && conf.inboundContacts.length == 1){
              xmsManager.deleteXMSCall(numberClient.local_identifier,function(rs){
                if(rs.statusCode === 204){
                  console.log('busy to delete: ' + numberClient.username);
                  numberClient.socket.emit('webrtc',{type: 'bye'});
                  _resetClient(numberClient.username, numberClient.host, numberClient.socket, numberClient.delivery);
                  var i = conf.inboundContacts.indexOf(numberClient.username);
                  if(i != -1) {
                    conf.inboundContacts.splice(i, 1);
                  }
                  //conf.inboundContacts.pop(numberClient.username);

                  xmsManager.deleteXMSConference(conf.id, function(rs){
                    if(rs.statusCode === 204){
```

```
            console.log('webrtc:hangup:conference resource: ' + conf.id + ' delete success.');
            delete conf;
          }
        });
      }
    });
  }

}else if(numberClient.type === 'sip' && data.status === 'on'){
  var conf = conversations[client.conf_id];
  var i = conf.inboundContacts.indexOf(client.username);
  if(i != -1) {
    conf.inboundContacts.splice(i, 1);
  }
  //conf.inboundContacts.pop(client.username);

  xmsManager.deleteXMSCall(client.local_identifier,function(rs){
    if(rs.statusCode === 204){
      console.log('hangup: ' + client.username);
      client.socket.emit('webrtc',{ type: 'bye' });

      if(conf.inboundContacts.length + conf.outboundContacts.length > 1){
        socket.broadcast.to(client.conf_id).emit('webrtc',{
          type: 'forward',
          number: client.number,
          newNumber: conf.outboundContacts.length != 0 ? conf.outboundContacts[0] : conf.inboundContacts[0],
          webrtcOnly: conf.outboundContacts.length == 0 ? true : false
        });
      }

      if(conf.inboundContacts.length == 0){
```

```
                    xmsManager.deleteXMSCall(numberClient.remote_identifier, function(rs){
                      if(rs.statusCode === 204){
                        var i = conf.outboundContacts.indexOf(numberClient.number);
                        if(i != -1) {
                          conf.outboundContacts.splice(i, 1);
                        }
                        //conf.outboundContacts.pop(numberClient.number);

                        var call_req = _und.find(callRequests,function(q){//TODO: first one match bug
                          if(q.caller.type == 'webrtc'){
                            return q.callee.number == numberClient.number;
                          }else{
                            return q.caller.number == numberClient.number;
                          }
                        });

                        var pair = call_req.caller.type == 'webrtc' ? call_req.caller : call_req.callee;

                        gw.sendBye(pair.username,function(){
                          console.log('bye to number: ' + numberClient.number);
                        });

                        xmsManager.deleteXMSConference(conf.id,function(rs){
                          if(rs.statusCode === 204){
                            console.log('sip:hangup:conference resource: ' + conf.id + ' delete success.');
                            delete conf;
                          }
                        });
```

```
                    }

                  });
                }

                _resetClient(client.username,client.host,
                    client.socket,client.delivery);

              }
            });

          }else if(numberClient.type === 'sip' &&
              data.status === 'off'){

            gw.sendBye(client.username,function(){
              console.log('bye to number: ' +
                  numberClient.number);
            });

          }

        }

      }catch(e){
        console.log('RUNTIME ERROR:sip:hangup: ', e);
      }

      break;
    default:

      break;
  }
});

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

  log('Got text message:', data);

  var client = clients[data.source];

  try{

    if(client.conf_id && data.type === 'IM'){
```

```
        socket.broadcast.to(client.conf_id.toString()).emit(
            'message',data);

        var conference = conversations[client.conf_id];
        if(conference.outboundContacts.length != 0){
          _und.each(conference.outboundContacts,function(
              sipNumber,key){
            data.toNumber = sipNumber;
            _sendSMS(data,function(result){
              client.socket.emit('message',{
                type: 'IM Report',
                status: result,
                content: data.content,
                toNumber: sipNumber
              });
            });
          });
        }

      }else if(data.type === 'SMS'){
        _sendSMS(data,function(result){
          client.socket.emit('message',{
            type: 'SMS Report',
            status: result,
            content: data.content,
            toNumber: data.toNumber,
            comment: data.comment
          });
        });
      }

    }catch(e){
      console.log('RUNTIME ERROR:message: ', e);
    }

  });

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

    var lostClient = _und.find(clients,function(client,key){
      return client.socket.id == socket.id;
    });

    if(lostClient){

```

```
      var conf = lostClient.conf_id ? conversations[lostClient.conf_id] : undefined;
      if(conf){
        var i = conf.inboundContacts.indexOf(lostClient.username);
        if(i != -1) {
          conf.inboundContacts.splice(i, 1);
        }
        //conf.inboundContacts.pop(lostClient.username);
      }

      if(lostClient.local_identifier){
        xmsManager.deleteXMSCall(lostClient.local_identifier,function(rs){
          if(rs.statusCode === 204){
            console.log('disconnect: ' + lostClient.username);
          }
        });
      }

      delete clients[lostClient.username];

      socket.broadcast.to(lostClient.host).emit('webrtc',{
        type: 'disconnect',
        user: {
          name: lostClient.username,
          number: lostClient.number,
          online: false
        }
      });

      gw.unregister(lostClient.username);
    }

  });

  delivery.on('receive.success',function(file){

    var sendingClient = _und.find(clients,function(client,key){
      return client.socket.id == socket.id;
    });

    fs.writeFile(file.name,file.buffer, function(err){
```

```
      if(err){
        console.log('File could not be saved.');
      }else{
        console.log('File saved.');
        _und.each(clients,function(client,key){
          //if(client.conf_id == sendingClient.conf_id && client.username != sendingClient.username){
          if(sendingClient.conf_id && client.conf_id == sendingClient.conf_id && client.username != sendingClient.username){

            client.socket.emit('message',{
              content: 'Sharing file: ' + file.name + ' ...',
              source: sendingClient.username,
              number: sendingClient.number,
              action: false
            });

            client.delivery.send({
              name: file.name,
              path : './' + file.name
            });

          }
        });

        fs.unlink('./' + file.name);
      };
    });
  });

}
```

Appendix C

C.1 WebRTC in Dart

Code Snippet C.1: WebRTCCtrl in Dart application client

```
library webRTCCtrl;

import 'package:angular/angular.dart';
import 'dart:html';
import 'package:webrtcDemo/speaker/speack_client.dart';

@NgController(
  selector : '[webrtc-ctrl]',
  publishAs : 'ctrl'
)

class WebRTCCtrl {

  static const String SERVER_URL = "ws://127.0.0.1:3001";

  String websocketUrl = SERVER_URL;

  WebRTCCtrl() {
    _initConnection();
  }

  void _initConnection(){
    var speaker = new SpeakerClient(websocketUrl, room: 'room');

    speaker.createStream(audio: true, video: true ).then((stream) {
      var video = new VideoElement()
        ..autoplay = true
```

```
          ..src = Url.createObjectUrl(stream);

        document.body.append(video);
      });

      speaker.onAdd.listen((message) {
        var video = new VideoElement()
          ..id = 'remote${message['id']}'
          ..autoplay = true
          ..src = Url.createObjectUrl(message['stream']);

        document.body.append(video);
      });

      speaker.onLeave.listen((message) {
        document.query('#remote${message['id']}').remove();
      });
    }
  }
```

Appendix D

D.1 AngularJs Files Structure

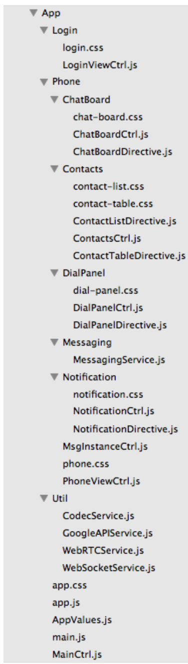

Figure D.1: Prototype Application AngularJs Files