NTNU – Trondheim
Norwegian University of
Science and Technology

Unified Communication and WebRTC

Xiao Chen

Submission date:	June 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

During the development of traditional telephony echo-systems, the cost of maintenaning 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. To the users who have already had 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 which developers have already implemented. Furthermore, since WebRTC is a web based API, then the implementation of Quality of Service (QoS) for WebRTC can provide assurance to users and prioritize services (enterprise, emergency, law enforcement, eHealth) that a WebRTC service will work as well as they need it to. WebRTC almost provides network operator a complete new business market with a huge amount of new 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 prototype system of this thesis will cover, the end-user can even 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 in the market, it is quite normal to think about how to integrate these kinds of web applications 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 main purpose of this thesis.

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

Current status of WebRTC technology, WebRTC business use cases, analysis of different possible WebRTC implementation 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 already 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 uses 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 installing 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 communicate with non-

[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]

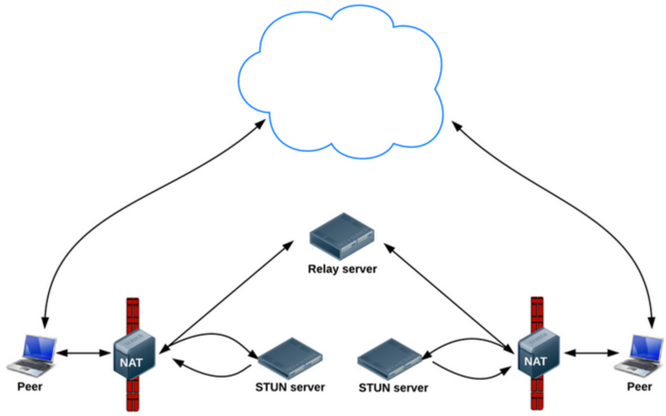

Figure 1.1: WebRTC Network: Finding connection candidates[Dut14]

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] shows 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 which is 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]

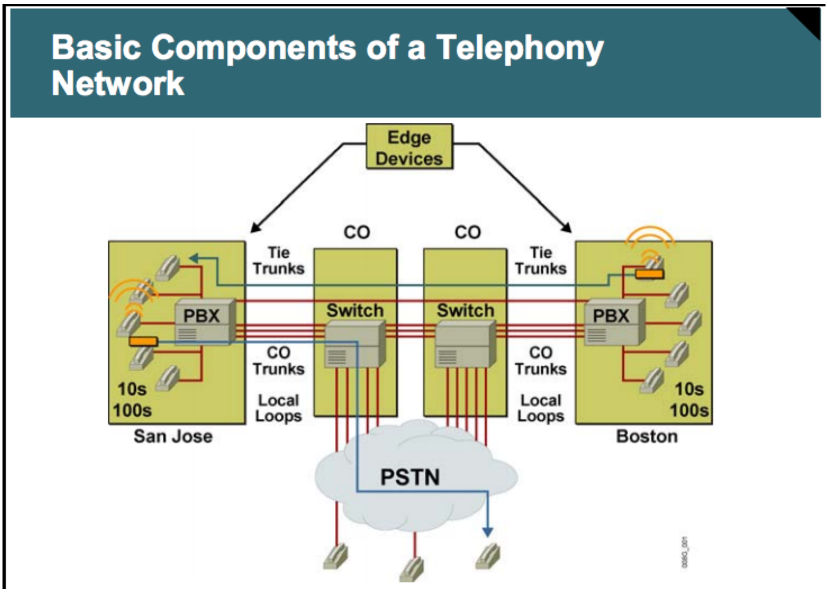

Figure 1.2: Traditional Telephony Network

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 information 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 two 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]

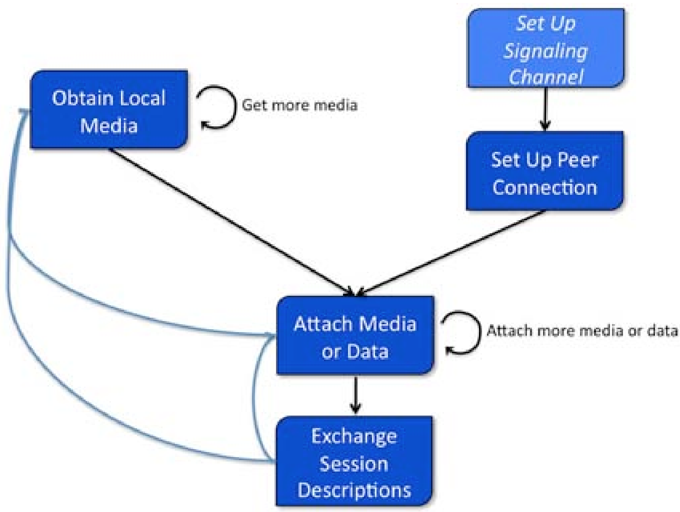

Figure 1.3: WebRTC API View with Signaling[JB13b]

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. Then 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 these 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 any 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 in purpose, then signaling is not part of the *RTCPeerConnection* API which can be decide how to implemented based on different project scenario.

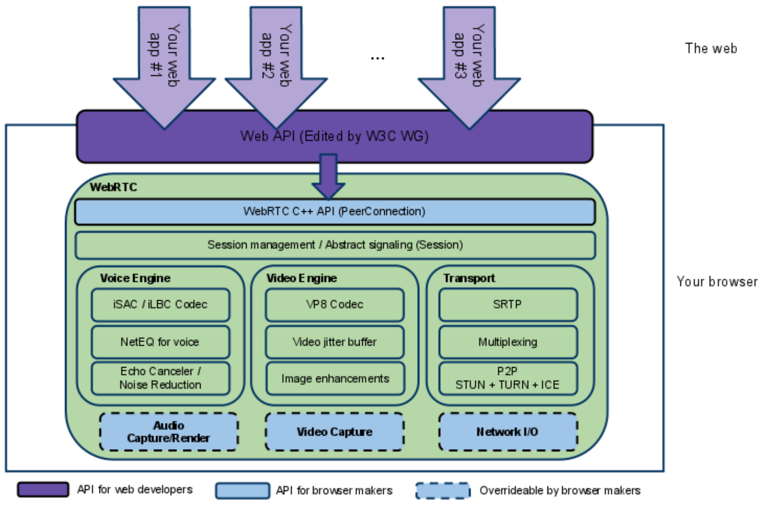

Figure 1.4: WebRTC architecture [Goo12]

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 in WebRTC[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 as

[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 communication 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 can 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 the roles as 'User Agent', 'Registrar' and 'Gateway' elements in the SIP 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 requests to SIP server and receive SIP message requests 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 requests 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 the prototype system, 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/ Gateway is the application server in the prototyep 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, there is another media server besides the media relay function provided by SIP server because the media server needs to

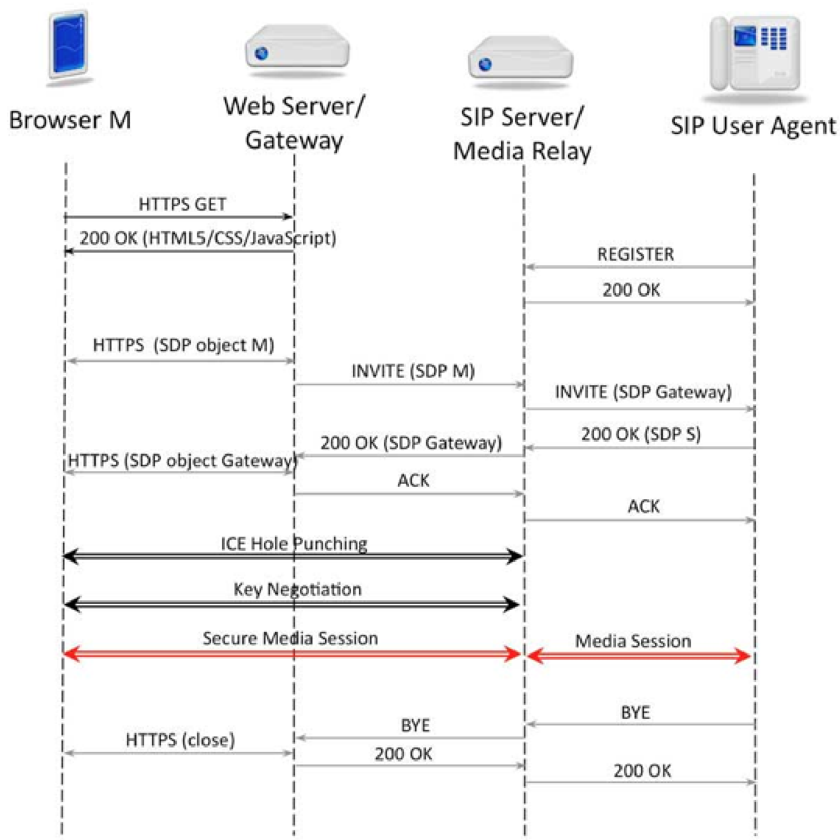

Figure 1.5: Prototype System Working Diagram [JB13c]

handle different media SDP in signalings which are WebRTC SDP and SIP SDP. The media server used in the prototype system is provided by Dialogic, the Network 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 is a WebRTC client or a SIP client, they will communicate with the same signaling client in their aspect.

Moreover, since the media server is used in this case, during the multiple end-point

[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.

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 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 servers of the prototype system 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 based on these study 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 standardize the relevant protocols in the IETF and browser APIs in the W3C. Then more and more web applications start using it in different ways. WebRTC APIs includes three important APIs, shown below. There are mainly two types of the WebRTC applications used them in separately or cooperatively way.

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

RTCPeerConnection is the foundation of all WebRTC application to establish the peer to peer connection. For showing remote peer media source content and exchange the local peer media source content, the web application need to get the user's camera view and microphone sound, the *MediaStream* API is used always in real-time communication application. 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 to access and use.

2.1.2 Uberconference

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. It is 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]

The prototype system in this thesis ideally is 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 it allows 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. And also it is more like usual telephone using because user call contacts based on their telephone number on the contact list. During the real-time conversation, the prototype application will provide user cooperation tools like instance message and file sharing in this development phase.

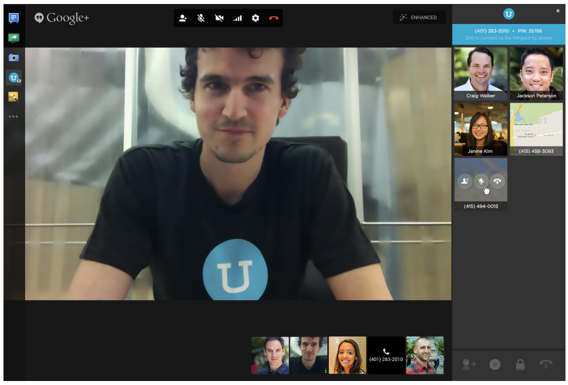

Figure 2.1: UberConference integrate with Hangouts Screen shot[Web14a]

2.1.3 Cube Slam

Moreover, there is another important API, *RTCDataChannel* , can be used more creatively by developers to build web applications. The experiment usage cases, 'Cube Slam' and 'Webtorrent', are in this category which uses *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

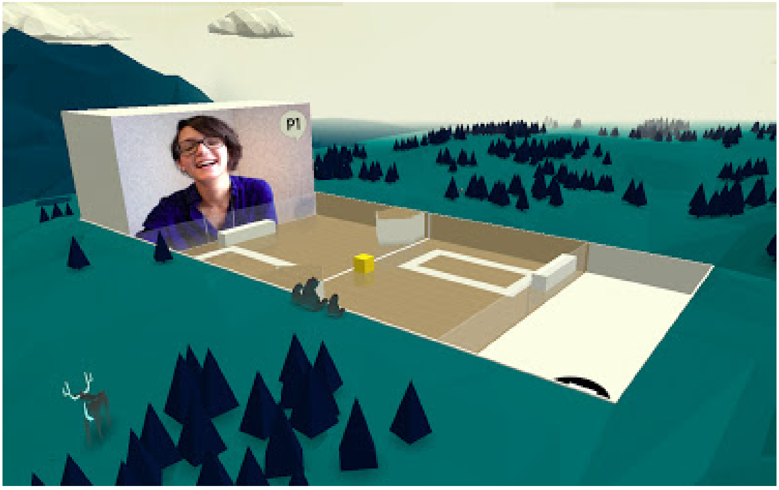

Figure 2.2: Cube Slam Game Over Screen

developers.[Blo14]

The idea behind the Cube Slam is that using *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 the 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 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. The *RTCDataChannel* solution 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 *RTCDat-*

aChannel can be used in more creative way like Webtorrent. Although it need to keep the browser up and running on both ends and 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 for the clients. 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.

This *RTCDataChannel* usage is reasonable solution to the prototype system as well. However, the main focus of the prototype system is to integrate the WebRTC multimedia type with the VoIP network against with traditional telephony network. It will not implement *RTCDataChannel* function in the system, but this topic will be discussed in chapter 6.

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

[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]

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 through 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). 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 70-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 by her own. Then she uses her mobile phone to call her family doctor. Her family doctor, Isak, uses the prototype service from his company and operator. When Eve call to her doctor for help, Isak answered her phone and tried to get her previous medical information from his working 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). Eve's previous 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 it is necessary. She can also 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 for Eve case.

In Eve aspect, she only calls doctor Isak, but 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 meants to be the peer to peer communication architecture and 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.[Wik13i]

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

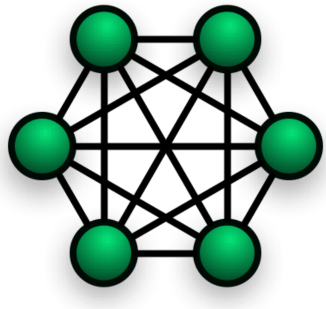

Figure 3.1: Illustration of a Mesh Network [Wik13i]

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 will eventually be the top priority issue in the future. 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

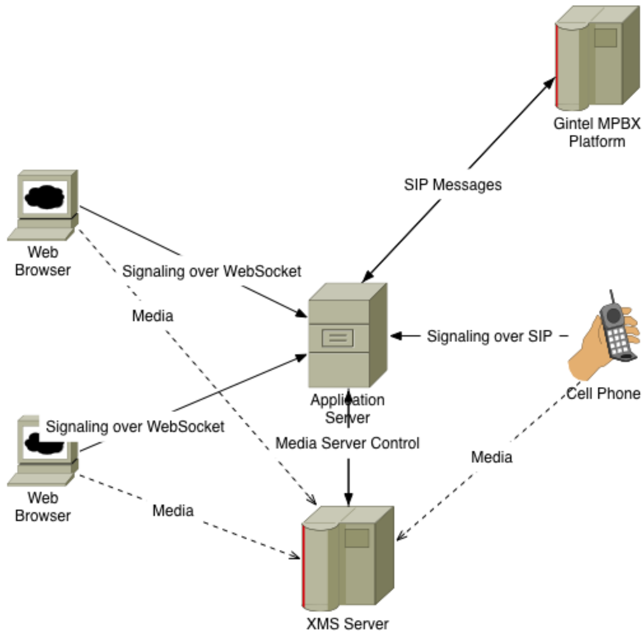

Figure 3.2: Prototype System Network

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 server to do the WebRTC signaling, the media stream is not go through the application server. Instead, it goes through the media server only. Furthermore, every client creates single WebRTC connection peer with the one call resource on XMS media server, the advantage of point-to-point line configuration is still kept in this 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 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.

Regarding to centralized server network, it is possible to use different signaling protocol for WebRTC browser clients with application server communication and SIP clients with application server communication. The benefits of having two different signaling protocols in prototype system is that it keeps the WebRTC clients and the SIP clients in their own traditional working process, there will be no compatibility issues for both sides. The application server in the prototype system will play the role as a gateway to decide which signaling protocol needs to be used when there are two different communication end clients in the conversation. Moreover, it will be easier for different existing WebRTC commercial services and SIP commercial services to integrate with the prototype system in order to communicate with each other network.

The disadvantage of centralized server network would be the application server and media server themselves. During the development of the prototype system, it is easy to figure out that the machines for hosting the application server and media server are not powerful enough to handle too many client connections and media stream exchange traffic load. When it comes to the scaling issue, the application server and media server need to be distributed in multiple server hosts on powerful server machines. The cost of the entire system will probably be 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 which is integrated with traditional telephony network.

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 frameworks nowadays which provide ways to develop a rich-client web application. In this section, some of the web application framework will be discussed to figure out which framework is the best solution to the prototype scenario. Furthermore, application server will be discussed with different implementation solutions since it does signaling and bridges the SIP network and browser clients.

3.2.1 Client Implementation Framework

To choose right web application framework to implement the application client 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 frameworks, it narrows down to three main framework to address.

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 restrict, it is easy to mess up the code and value type in script language. But Dart has the type restrict in the language with the other feature script language has. 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 A.5 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 Software Development Kit (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 is 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 9 is to send message to server when *RTCPeerConnection* object get *onIceCandidate* event witch ICE candidate information. Line 13 is to bind the message listener event to Dart function *onCandidate.listen*. From line 16 to line 19 is the Dart function to create *RTCIceCandidate* object and add to *RTCPeerConnection* object. The bug issue happens on line 20, 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      pc.onIceCandidate.listen((e){
4        if (e.candidate != null) {
5          _send('candidate', {
6            'label': e.candidate.sdpMLineIndex,
7            'id': id,
8            'candidate': e.candidate.candidate
9          });
10       }
11     });
12 ...
13 get onCandidate => _messages.where((m) => m['type'] == '
       candidate');
14 ...
15 onCandidate.listen((message) {
16   var candidate = new RtcIceCandidate({
17     'sdpMLineIndex': message['label'],
18         'candidate': message['candidate']
19     });
20     _connections[message['id']].addIceCandidate(candidate,()
           {},(e){
21         print('add ice candidate error');
22     });
23 });
24 ...
```

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

[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.

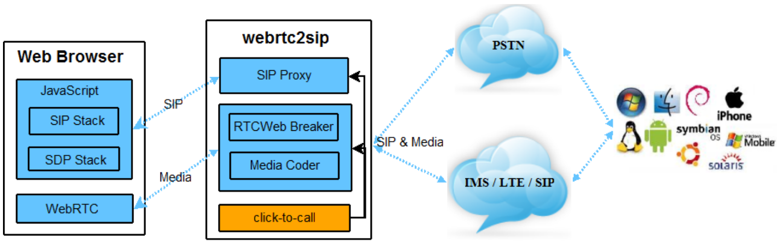

Figure 3.3: Sipml5 and webrtc2sip Network

use *js-interop* library to use pure JavaScript code in Dart to call the WebRTC Web API instead of Dart WebRTC interface.

However, 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]

Since Dart in not support to most modern web browser like FireFox, will not be used in this prototype. The prototype application need to meet the requirement of most modern web browsers.

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 entire client solution to communicate with other kinds 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. Like the Figure 3.3 showing, it works with media gateway webrtc2sip. 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.

webrtc2sip is a smart and powerful gateway using WebRTC and SIP to turn your browser into a phone with audio, video and SMS messaging capabilities. The gateway allows 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 as well.

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 causes the problem. In the test, the registration for SIP client is successful, but there are 'too long message' as 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 Angular module services to provide different interfaces against different web APIs. In the prototype application, there will be several third party Angular module libraries used in order to integrate with some advanced JavaScript library or web APIs in

Angular style.

Socket.IO is a JavaScript library for real time web applications. It has two parts: a client-side library that runs in the browser client, 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 it 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 in prototype working scenario.

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 needs to support Socket.IO framework as well. The most natural solution would be Node.js, but it is possible to work with other traditional web server solution based on WebSocket protocol still. In this 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]

[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]

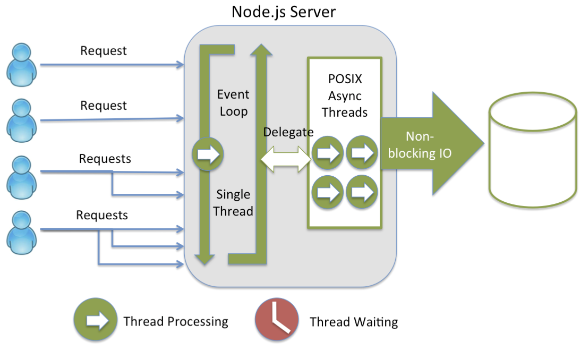

Figure 3.4: Node.js Non-blocking I/O[Rot14]

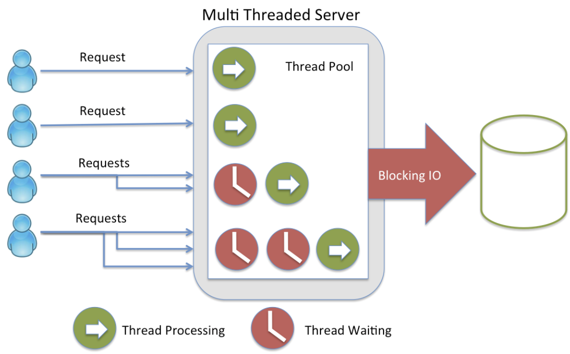

Figure 3.5: Multiple Threaded Server[Rot14]

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 code path needs to be non-blocking and synchronized, not just the IO layer. This is where Node.js 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.

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

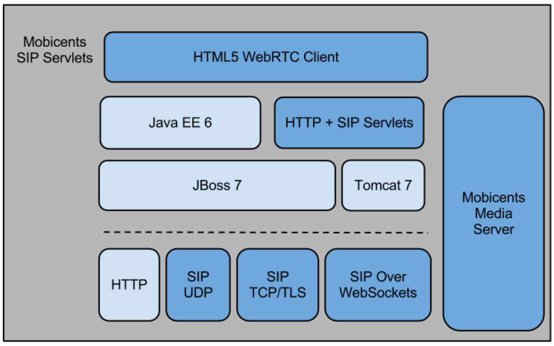

Figure 3.6: Mobicents SIP Servlets[Tel14c]

WebSocket protocol framework(Socket.IO) used on client side has good server side solution based on Node.js, it makes 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 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 chapter 4. Since it is not mature library, there are quite a 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

[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

SIP Servlet v1.1 (Java Specification Requests (JSR) 289 Spec) on top of Tomcat[5] and JBoss[6] containers and it strives 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.

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 needs 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, it makes the development harder as well.

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. But the SIP transport layer is encapsulated in the Mobicents SIP Servlets framework, it is hard to modify it in order 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.

---

[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).[Wik13b]

[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.[Wik13q]

[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.[Wik13g]

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

To sum up, 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 implementation progress of the prototype system along with explanation and analysis. The prototype system is implemented based on solution research 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 which 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. *RTCDataChannel* API will be discussed in chapter 6.

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, the WebRTC API provides *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.

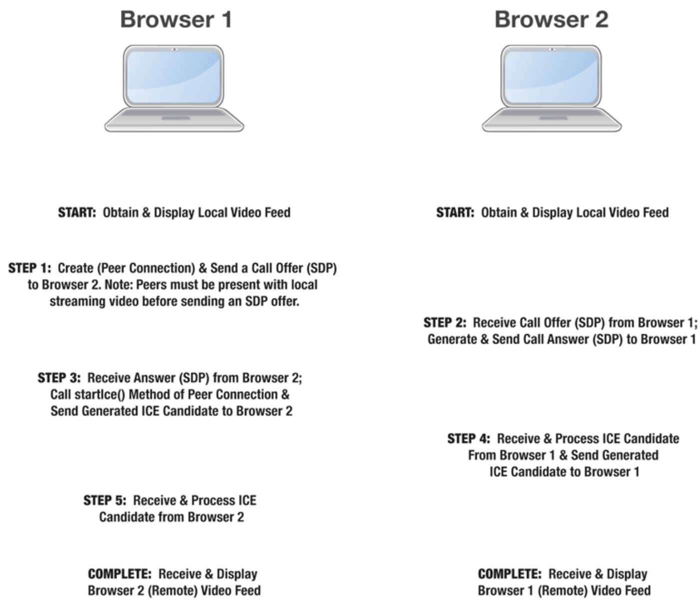

Figure 4.1: WebRTC two peer communication process[Net14b]

– 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 4.2, there are two blocks to make all the set up process for FireFox and 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 as well. For example, 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 browsers 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 differently on different version of Google Chrome browser. In the Code Snippet 4.2, from line 19 to line line 29 is the sample case to distinguish the difference among different version of Google Chrome to handle the *RTCPeerConnection* ICE server constraint implementation.

Code Snippet 4.2: WebRTCService.js in application client

```
1 angular.module('webrtcDemo.services').
2   factory('WebRTCService',function () {
3
4     ...
```

[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]

```

  function _setRTCElement() {

    if(navigator.mozGetUserMedia){
      ...
    }else if(navigator.webkitGetUserMedia){
      ...

      // Creates iceServer from the url for Chrome.
      _createIceServer = function(url, username, password) {
        ...
        if (url_parts[0].indexOf('stun') === 0) {
          ...
        } 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;
      };

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

  }

  return {
    ...
  }
});
```

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.

4.1.2 RTCPeerConnection API

The step 1 of Figure 4.1 is that the caller peer set up a communication process, it simply creates *RTCPeerConnection* then send a WebRTC offer(by calling *createOffer()* function and sending WebSocket message with SDP) with local SDP to the other peer. In the prototype system, it will send WebRTC offer to the application server, then the application server will check if the receiver is a WebRTC client or SIP to send different type of offer message(It will be covered in later of this chapter).

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 4.3 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 4.3 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, DTLS is supported, then in the

[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]

prototype application, it will use DTLS protocol to exchange the media stream.

There are two APIs to handle the *RTCIceCandidate* object which contains ICE information data. One is *onicecandidate* listener to trigger the function to handle the new *RTCIceCandidate* data object. The other one is *addIceCandidate* function, which is shown in the Code Snippet 4.4, to add the new *RTCIceCandidate* data object to the remote/local peer connection session description field. As Code Snippet 4.4 shown, the *RTCIceCandidate* object has three attributes, *sdpMLineIndex* is the media line index in the SDP, *sdpMid* is the media id which differ it is video or audio in the SDP and *candidate* is the IP address and other detail of this media source.

Code Snippet 4.3: Create Peer Connection function

```
pc_config = WebRTCService.webrtcDetectedBrowser() === 'firefox' ?
        {'iceServers':[{'urls':'stun:stun.services.mozilla.com'}]} :
        {'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 4.4: Add Remote IceCandidate function

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

There are difference between Firefox and Google Chrome to handle the *RTCIceCandidate* content in the SDP while sending the offer to the other peer. In Firefox, it waits for all the *RTCIceCandidate* data is fetched from STUN/TURN server then send it with WebRTC offer message. However, in Google Chrome, it sends the WebRTC offer message first, then update the local SDP with new coming *RTCIceCandidate* data one by one. Then in the prototype system, it needs to handle these differences by listening the *endIceCandidate* event on application. After client fetches all the *RTCIceCandidate* data, it sends an *endIceCandidate* message to the application server to send the complete WebRTC offer message to the other peer.

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.

WebRTC clients 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 Snippet4.5 shown is the WebRTC answer SDP from the callee when the callee end-point decide to accept this conversion session.

The sample SDP from the prototype application is shown in Code Snippet 4.5. Line 2 in Code Snippet 4.5 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 4.5, 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 to exchange the media stream.

Code Snippet 4.5: 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
27 a=candidate:2 1 UDP 2130706431 10.254.9.135 57344 typ host
28 a=candidate:2 2 UDP 2130706430 10.254.9.135 57345 typ host
29 ...
```

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 4.4. 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. In the prototype system, these exchange *RTCIceCandidate* process is relayed by application server to the different end points.

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 B.1. Application has two main web pages, *login* page and *phone* page. There are *chatboard*,*contacts list*, *contacts table*, *dialpanel* and *notification* user interface component blocks in *phone* page. For each part of the application blocks, it has controller script and service script. Controller and service scripts are working with the HTML view scripts. In this section, there will be one sample component block of the prototype application client explained in order to understand how the AngularJs is used in prototype application.

The *app.js* script shown in Code Snippet 4.6 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 4.7, *ContactTableDirective.js* script in Code Snippet 4.8, *ContactsCtrl.js* script in Code Snippet 4.9 and *GoogleAPIService.js* script in Code Snippet 4.11. 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 shown in Code Snippet 4.6, 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 modules 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 for using in the prototype application.

In Code Snippet 4.6, from line 8 to line 20 is implemented to set the application routing map. There are two main pages, one is *login* page with "/login" URL and the other one is *phone* page with "/chat" URL. The Angular controllers which are bind with these page view are also declared in *$routeProvider* service. 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 enter the service.

Code Snippet 4.6: app.js in application client

```
1  angular.module('webrtcDemo', [
2      ...
3    'webrtcDemo.services',
4    'webrtcDemo.controllers',
5    'webrtcDemo.directives',
6    'webrtcDemo.filters'
7  ]).
8  config(function ($routeProvider, $locationProvider,
       $httpProvider) {
9    $routeProvider.
10     when('/chat', {
11       templateUrl: 'partials/phoneView',
12       controller: 'PhoneViewCtrl'
13     }).
14     when('/login',{
15       templateUrl: 'partials/login',
16       controller: 'LoginViewCtrl'
17     }).
18     otherwise({
19       redirectTo: '/login'
20     });
21
22   ...
23   });
```

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 than normal Embedded JavaScript templates (EJS) template engine. In Code Snippet 4.7, Jade has the same node name as EJS. And there are some Angular directives in the template from Code Snippet 4.7. For example, at line 2 in Code Snippet 4.7,

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

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 field *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 11 in Code Snippet 4.7, the attribute *ng-class* is a native Angular directive attribute, it provides the Cascading Style Sheets (CSS)[5] changes to some specific CSS class name according to some certain value matches in AngularJs expression. At line 11, the *<tr>* tag's CSS attributes will be success CSS class only if the boolean value of *item.online* is *true*.

AngularJs provides two-way data module binding in the template and controller. Line 17 in the Code Snippet 4.7, *{{item.number}}* is the Angular template to display the *number* property value of *item* object in the HTML template. And line 14 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.

Code Snippet 4.7: contactTable.jade in application client

```
1  div(id = "contactTable")
2    angucomplete-alt(id="contactSearch",
3      ...
4      local-data="contactsHolder.contacts",
5      ...)
6    tabset
7      tab(heading = "Conacts")
8        table(id = "contacts", at-table, at-paginated,
             at-list="contactsHolder.contacts | orderBy:online"
             , at-config="config",class="table table-hover
             table-striped table-condensed" )
9          thead
10         tbody
11           tr(ng-class = "{success: item.online}", ng-init =
                 "item.hvor = false", ng-mouseenter = "
                 contactHvor(item)", ng-mouseleave = "
                 contactHvor(item)")
12             ...
13               p(ng-hide = "item.hvor").
```

[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.[Wik13d]

```
14                  {{item.online | iif : "Online" : "Offline"
                        }}
15                          ...
16                p
17                  | Telephone : {{item.number}}
18          ...
```

4.2.3 ContactTableDirective.js Script (Customized Directive)

After creating the view of contact table component, it is necessary to bind controller to the view and declare the component a tag name used in the HTML template. It is called *Directive* in AngularJs, and the *ContactTableDirective.js* script is shown in Code Snippet 4.8. From line 1 to line 12 is the directive declaration, it sets the *templateUrl* to 'partials/contactTable' which is the view component file path and binds the controller which named as *ContactsCtrl* to the view component. The *restrict* filed in the directive is to set the template type for *ContactTableDirective*, in the Code Snippet 4.8 line 5, it means this directive is a HTML element template, it can be used as normal HTML element by using name 'contact-table'. By using AngularJs directive, it makes the HTML view template more modularized and makes the same view component could be reused in different place in the web application.

From line 14 to line 19 is the Angular filter declaration, it is a filter named as *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 at line 14 of the Code Snippet 4.7.

Code Snippet 4.8: ContactTableDirective.js in application client

```
1  angular.module('webrtcDemo.directives').
2    directive('contactTable',function () {
3
4      return{
5        restrict: 'E',
6        replace: true,
7        scope: true,
8        templateUrl: 'partials/contactTable',
9        controller: 'ContactsCtrl'
10     };
11
12   });
13
14 angular.module('webrtcDemo.filters').
15   filter('iif', function () {
16    return function(input, trueValue, falseValue) {
17        return input ? trueValue : falseValue;
```

```
    };
});
```

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 with the user interface views. The example controller in Code Snippet 4.9 controls the contactTable view directive and get data functions from GoogleAPIService. At the line 2 of Code Snippet 4.9, 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 data to show in the HTML template. And *storage* is another service provides *localstorage* 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 22 of the Code Snippet 4.9 by calling *storage.set()* function to store the contacts data in the W3C web storage.

At line 5 of the Code Snippet 4.9, 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 functions from the *GoogleAPIService* used. 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 to query the contacts information data by Google Contacts API after get the Google API permission.

Promise object is the new concept in the AngularJs, and it will be standardized in new ECMAScript[6] 6. The core idea behind promises is that a promise represents the result of an asynchronous operation. A promise object 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 a great concept and important feature in the AngularJs. Since everything in Javascript is asynchronous operation, then promise 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 of AngularJS service.

[6]ECMAScript is the scripting language standardized by Ecma International in the ECMA-262 specification and ISO/IEC 16262. The language is widely used for client-side scripting on the web, in the form of several well-known implementations such as JavaScript, JScript and ActionScript.[Wik13f]

From line 9 to line 15 is the process to strip 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 use the contacts list in contact table view component.

Code Snippet 4.9: ContactsCtrl.js in application client

```
1  angular.module('webrtcDemo.controllers').
2    controller('ContactsCtrl',function ($scope,$location,
       WebSocketService,GoogleAPIService,storage,$filter) {
3      ...
4
5      $scope.importContacts = function(){
6        $scope.contactsHolder.contacts = [];
7        GoogleAPIService.oAuth().then(function(token){
8          GoogleAPIService.queryContacts(token).then(function(
             data){
9            angular.forEach(data.feed.entry,function(person,
               key){
10             if(person['gd$phoneNumber']){
11               var contact = {
12                 name: person.title['$t'],
13                 number: person['gd$phoneNumber'][0]['$t'],
14                 online: false
15               }
16
17               ...
18
19             }
20           });
21
22           storage.set('contactList-' + username,
               $scope.contactsHolder.contacts);
23
24         });
25       });
26     }
27
28   });
```

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 4.11, it provides interfaces of Google API to the controller. There are two interfaces in the *GoogleAPIService.js* script. One is Google authorization login to get the user permission, the other one is fetching Google contacts information from the Google Contacts API.

From line 4 to line 20 in Code Snippet 4.11, it is the promise function, *_authLogin()*, to get Google authorization token in order to call any Google APIs 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 *prmoise* 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 *prmoise* object is to allow for interested parties to get access to the result of the deferred task when it completes.[ang14] At line 5 and line 23 is the code to create a new instance of *deferred* and a new *prmoise* instance. From line 7 to 10, it is the configuration object for Google API authorization. The *gapi* object is loaded from the Google API Javascript client script included in *index.jade* shown in Code Snippet 4.10.

Code Snippet 4.10: Include Google API Javascript file in Index.iade

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

Since application only needs 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 needs to create his own Google App project then set the APIs which the project will ask user permission for 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[7] credential created on Google App project.

Then the *gapi* object call *auth.authorize()* function with the configuration object to get authorization token. At line 15, when the asynchronous process is finished, *deferred* object will call *resolve* function to send the token object back to the *promise.then* function at line 7 in Code Snippet 4.9 which is mentioned at previous section.

From line 22 to line 34 in Code Snippet 4.11 is another promise function, *_fetchContacts()* , 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

[7] 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.[Wik13k]

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[Wik13h]. The reason 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 requests for data from Google API server. And Google API server does not support cross domain request for security, but with JSONP it is allowed to have cross origin domain resources sharing.

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 as the promise function resolving data, shown at line27 in Code Snippet 4.11.

Code Snippet 4.11: GoogleAPIService.js in application client

```
1  angular.module('webrtcDemo.services').
2    factory('GoogleAPIService', function ($q,$http,storage) {
3
4      function _authLogin(){
5        var deferred = $q.defer();
6
7        var config = {
8            'client_id': '
                 xxxxxxxxxxxxxx.apps.googleusercontent.com',
9            'scope': 'https://www.google.com/m8/feeds'
10         };
11       gapi.auth.authorize(config, function() {
12
13         console.log('login complete');
14         console.log(gapi.auth.getToken());
15         deferred.resolve(gapi.auth.getToken());
16
17       });
18
19       return deferred.promise;
20     }
21
22     function _fetchContacts(authToken){
23       ...
24
25       $http.jsonp(url).
26       success(function(data, status, headers, config) {
27         deferred.resolve(data);
```

```
      }).
      error(function(data, status, headers, config) {
        deferred.reject('
            GoogleAPIService:queryContacts:Failed');
      });

      return deferred.promise;
    }

    ...
  });
```

4.3 Socket.IO Implementation

In the prototype system, web application client and application server communicate with each other over WebSocket shown in Figure 3.2. There are two main intentions to have the signaling channel over WebSocket. One is for signaling of WebRTC ICE candidate exchange and the other one is to exchange the communication data (text message, files). Unlike HTTP, WebSocket provides 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[8] 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 A.1, it is implementation of Socket.IO on the application server. From line 1 to line 6, it is the initialization process of the Socket.IO on Node.js. At line 4, it means that when the client binds with the application server through WebSocket, the listener start in handler function *_handlerSocket()*. The WebSocket channels and usages are shown in Table 4.1.

At line 11 in Code Snippet A.1, *socket* object is created by the Socket.IO framework whenever one client connects with the server through WebSocket. The *listener* function is implemented in the same pattern in *socket.on()*. There are two arguments

[8]Comet is a web application model in which a long-held HTTP request allows a web server to push data to a browser, without the browser explicitly requesting it.Comet is an umbrella term, encompassing multiple techniques for achieving this interaction. All these methods rely on features included by default in browsers, such as JavaScript, rather than on non-default plugins. The Comet approach differs from the original model of the web, in which a browser requests a complete web page at a time.[Wik13e]

Table 4.1: : Socket.IO Listening Channels in Code Snippet A.1

WebSocket Channel	Message Data Type	System Function
SIP	register	Web application login page SIP registeration message to SIP server
	invite	Web application client invites SIP client message
	answerInvite	Web application client gets INVITE SIP message from SIP client and answers it
WebRTC	register	Web application client finishes login with SIP credential and gets user permission to use *getUserMedia()* function and registers client itself on application server for WebSocket usage
	offer	Web application client sends offer message to appliction server to create call resource on XMS media server
	answerInvite	Web application client gets INVITE message from WebRTC client and answers it
	endCandidate	Web application client finishes getting ICE candidate from STUN/TURN server then application sends HTTP request to XMS media server with final SDP
	hangup	Web application client sends hangup messge to hangup itself from the current conference
message	Instance Message (IM)	Web application sends instance message to application server in order to broadcast to all the rest clients in current conference
	SMS	Web application client sends SMS to SIP client
disconnect	*	Web application client disconnects from the application server

taken by this function. The first one is the channel name, at line 11, it is *sip*, and second argument is callback function when this channel got any socket message. This callback function also takes one argument which is the message data sent by the client.

At line 16 in Code Snippet A.1, it is the implementation for server to send socket message back to the client through WebSocket in Socket.IO framework. *socket.emit()*, this function takes two arguments, the first one is the WebSocket channel name and second one is the socket message data object. All the socket message data is formatted in JSON because it is easier for both client and server side to resolve these message data.

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 which is correspond to the server side implementation.

The client side Socket.IO implementation is quite similar as the server side implementation mentioned above. The socket message event listener is implemented as the same pattern, like line 3 in Code Snippet 4.12. Moreover, at line 15, *socket.emit()* function is used for client to send socket message through WebSocket to server in Socket.IO framework. In this way, the client has same WebSocket channels listed in Table 4.1 and sends the related data type to the server to request server to run some corresponding process.

Code Snippet 4.12: _setSocketListener() Function in PhoneViewCtrl.js on Application Client

```
1  function _setSocketListener(socket){
2
3     socket.on('log', function (array){
4       console.log.apply(console, array);
5     });
6
7     socket.on('webrtc', function (data){
8
9       switch(data.type){
10        ...
11        case 'answer':
12          if(isStarted){
13            ...
14            if(!data.self){
15              socket.emit('sip',{
16                type: 'invite',
```

```
                  username: $scope.user.name,
                  content: {
                    to: $scope.outPhone.number
                  }
                });
              }else{
                ...
              }
            }
            break;
          ...
        }
      });

      ...
    }
```

4.4 SIP Implementation on Application Server

There are not many SIP stack module made on Node.js Package Manager (NPM)[9]. After a lot of research, this prototype system will use a simple SIP module(sip.js ,https://www.npmjs.org/package/sip) on Node.js. It implements tranaction and transport layers as described in RFC3261(SIP: Session Initiation Protocol, http://www.ietf.org/rfc/rfc3261.txt). This library is still maintained by its author although the developer of this library is not so active during this thesis research period. But this library is the most appropriate library for Node.js.

The examples of sip.js library usage mostly are to be implemented as a SIP registration server or proxy server. Then the most of the interfaces provided by sip.js library are design to redirect all the SIP message and SIP register request. Although sip.js library provides SIP stack for Node.js and lower layer transportation on SIP protocol interface, it is not designed for manually generating different SIP message request to SIP server. Most of the SIP implementation of prototype application server have to be implemented to handle with all different types of SIP message generation cases by its own which is shown in Code Snippet A.2. These implementation is made based on the reference of RFC3261 and Wireshark[10] trace log of the SIP soft-phone

[9]npm is the official package manager for Node.js. As of Node.js version 0.6.3, npm is bundled and installed automatically with the environment. npm runs through the command line and manages dependencies for an application. It also allows users to install Node.js applications that are available on the npm registry.[Wik13j]

[10]Wireshark is a free and open-source packet analyzer. It is used for network troubleshooting, analysis, software and communications protocol development, and education. Originally named Ethereal, in May 2006 the project was renamed Wireshark due to trademark issues.[Wik13r]

application[11] (like Zoiper).

4.4.1 SIP Request Message Implementation

As mentioned in Chapter 1, there will be *REGISTER*,*INVITE*,*ACK*,*CANCEL* and *BYE* SIP message request implemented in application server to provide normal WebRTC browser client the SIP communication ability. Fortunately, the sip.js library provides mostly used SIP response, it is no need to modified these response when application server relaies the SIP response back to client.

From line 3 to line 20 of Code Snippet A.2 , it is the code block for generating *REGISTER* SIP message request. It is implemented regarding to RFC3261. The important part of this block implementation is the header of *REGISTER* SIP message. There are *call-id*,*cseq*,*from*,*to*,*contact* fields need to be set in the header. Field *call-id* contains a globally unique identifier for this call, generated by the combination of a random string and the client's host name or IP address. The combination of the *to* tag, *from* tag, and *call-id* completely defines a peer-to-peer SIP relationship between two end points and is referred to as a dialog. Field *cseq* or Command Sequence contains an integer and a method name. The *cseq* number is incremented for each new request within a dialog and is a traditional sequence number. For the prototype application server, the *cseq* number is increased (shown at line 69 of Code Snippet A.2) when the SIP *REGISTER* request is unauthorized then application server need to send another SIP *REGISTER* request with authorization information. And the method of *cseq* is *REGISTER* in this registration implementation. This process is implemented from line 63 to line 99 in Code Snippet A.2. Moreover, since the return 200 *OK* SIP response with the limited expired time for this *REGISTER* session by Session Border Controller (SBC)[12], at line 76, the application server sets up a timer to re-register the client in the interval of expire time. Field *to* contains a display name and a SIP or SIPs URI towards which the request was originally directed. Field *from* also contains a display name and a SIP or SIPs URI that indicate the originator of the request. This header field also has a tag parameter containing a random string that was added to the URI by the application server. It is used for identification purposes. Field *contact* contains a SIP or SIPs URI that represents a direct route to contact client, usually composed of a username at a Fully Qualified Domain Name (FQDN). It is important to use application server public IP address and port number since all the client SIP request messages and SIP responses need

[11] A softphone is a software program for making telephone calls over the Internet using a general purpose computer, rather than using dedicated hardware.[Wik13o]

[12] A session border controller (SBC) is a device regularly deployed in Voice over Internet Protocol (VoIP) networks to exert control over the signaling and usually also the media streams involved in setting up, conducting, and tearing down telephone calls or other interactive media communications.[Wik13n]

to be sent to application server in order to trigger other process in the prototype system.

Code Snippet 4.13: ACK Alice -> Bob Sample [Soc03]

```
1 ACK sip:bob@client.biloxi.example.com SIP/2.0
2 Via: SIP/2.0/TCP
      client.atlanta.example.com:5060;branch=z9hG4bK74bd5
3 Max-Forwards: 70
4 From: Alice
      <sip:alice@atlanta.example.com>;tag=9fxced76sl
5 To: Bob <sip:bob@biloxi.example.com>;tag=8321234356
6 Call-ID: 3848276298220188511@atlanta.example.com
7 CSeq: 1 ACK
8 Content-Length: 0
```

During the development, there is a bug issue found in the sip.js library when it regards to implement the *ACK* SIP message if an *INVITE* SIP message got accepted (200 *OK* message). The example SIP *ACK* message is from RFC3665(SIP Basic Call Flow Examples, http://tools.ietf.org/html/rfc3665) shown in Code Snippet 4.13. When implementing this SIP message in sip.js, the URI field of SIP *ACK* message need to set the port number with it to force this SIP message sending to the correct SIP protocol port on the SIP server regarding to line 2 of Code Snippet 4.13. It is normally set *client.atlanta.example.com;transport=udp* for the URI headers filed in most SIP soft-phone client. It means that the SIP request will be sent to the URL address server on UDP/TCP protocol port. Usually SIP server will set port number 5060 as UDP protocol port as default setting. However, this URI format feature is not supported by sip.js library. Then the implementation of this process, shown from line 22 to line 25 is to check if the contact URI of the SIP response header has port number or not. If there is no port number in it, it need to set the URI with the *5060* port number which is the target SIP server UDP port with SIP protocol implementation (it is implemented in same way as most SIP server).

4.4.2 SIP Message Listener and Handler Implementation

The application server in the prototype system does not only create SIP request message and send them to SIP server, but also listens to the SIP request/response messages from the SIP server.

In Code Snippet A.2, from line 29 to line 62, it is the initialization function for SIP gateway on application server. There are two parts in this code block. The first part is from line 31 to line 39 in Code Snippet A.2, it is the initialization of the SIP stack on application server on port *5060*. It configures the SIP stack on host IP address and host port number, also initializes the registration array for sip client credentials. Then at line 206, the function *sip.start()* is to start the SIP stack listener on these SIP gateway configuration.

The second part of the code in in Code Snippet A.2, it is the SIP listener to handle different SIP requests and SIP responses. Since the communication protocol between WebRTC browser clients and application server is WebSocket not SIP. Then the SIP messages to the application server are only from the SIP server on the traditional telephony network. The *rq* object in Code Snippet A.2 is the request/response SIP stack object on application server. It is the same message object when the application send to SIP server in sip.js library. In the code block of this part, it is necessary to check the *method* parameter of *rq* object to find out which type message it is.

For example, from line 45 to line 61 in Code Snippet A.2, it is the code block when application server get SIP *INVITE* request from the SIP server. It means that there is one SIP client wants to call the other WebRTC browser client in the prototype system. The application firstly send back a *Trying* SIP response back to the SIP server (at line 47 and 48 in Code Snippet A.2) by using the *sip.makeResponse* function to notify SIP client that the application server is trying to find out if this contact number is online in the prototype system. If the contact number is online in the prototype sytem, the application stores some necessary information data from the *INVITE* request and broadcast an internal event (*SIPREMOTE*) by *EventEmmitter* (init at line 1 in Code Snippet A.2) from *events* module in Node.js framework. This event is used to let *socket* service of application server get notified by remote SIP events to send necessary WebSocket message to the client in order to notify the end point user about the SIP messages(in *INVITE* example, the socket handler function is shown in Code Snippet 4.14).

Code Snippet 4.14: SIPREMOTE event handler for INVITE message

```
function _handlerSip(){
  gw.on('SIPREMOTE', function (data) {
      switch(data.type){
          ...
          case 'INVITE':
            var client = clients[data.content.toNumber];
            if(client.inConference){
                  ...
            }else{
              ...
              xmsManager.createXMSCall({
                callType: 'sip',
                sdp: data.content.inviteRequest.content
              },function(remote_xmsSDP,remote_id){
                sip.remote_xmsSDP = remote_xmsSDP;
```

```
16                     sip.remote_identifier = remote_id;
17
18                     client.socket.emit('sip',{
19                       type: "createRTCoffer",
20                       inComingNumber: data.content.fromNumber,
21                       callDirection: 'outbound'
22                     });
23                     ...
24                   });
25                 }
26             ...
27         }
28     }
```

In Code Snippet 4.14, at line 7, it checks if the receiver of this SIP *INVITE* message is during the conversation. If not, it will send *createXMSCall* request to XMS media server to create a new call resource for the SIP client(at line 11 in Code Snippet 4.14) and send WebSocket message (from line 18 to line 22 in Code Snippet 4.14) about this invite message to the WebRTC target client. At line 21 in Code Snippet 4.14, the *callDirection* parameter of WebSocket message object is set to *outbound*, it means that this invite message request is from a SIP client(outside of the prototype system network) to a WebRTC client(inside the prototype system network). The reason for this flag is for application server to create correct call resources on the XMS media server and does correct switch routing work for either WebRTC client or SIP client. The integration with XMS media server of application server will be discussed more in the next section.

4.5 XMS Media Server Integration on Application Server

XMS media server is used as media gateway in the prototype system, the main functions of it are to create call/conference session resources for multiple clients and to convert between WebRTC SDP and SIP SDP in order to bridge the WebRTC clients with SIP clients on RTP channel.

Since the integration is only between application server and XMS media server, using Representational State Transfer (REST)ful[13] communication based on HTTP is a appropriate solution to the prototype scenario and it is supported by XMS media

[13]Representational state transfer (REST) is a software architectural style consisting of a coordinated set of architectural constraints applied to components, connectors, and data elements, within a distributed hypermedia system. REST ignores the details of component implementation and protocol syntax in order to focus on the roles of components, the constraints upon their interaction with other components, and their interpretation of significant data elements.[Wik13m]

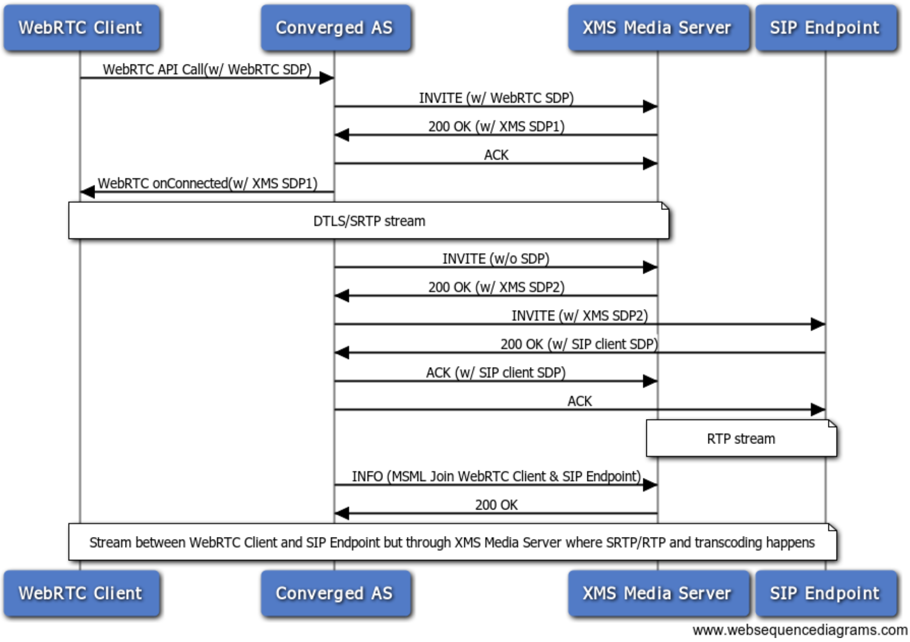

Figure 4.2: Single Call from Browser to SIP Client

server and Node.js framework on application server as well. The detail working flow of the prototype system integrated with XMS media server is shown in Figure 4.2.

During the process of single call from WebRTC client to SIP client, the application server needs to send *INVITE* message with WebRTC SDP of browser client to XMS media server(create call resrouce request) in order to request XMS media server to create WebRTC call resource and get the SDP of this newly created call resource from XMS media server. After WebRTC and newly created call resource connected in the RTP channel, application server sends *INVITE* message without SDP(create call request) to XMS media server in order to create SIP call resource. Then it sends the SIP *INVITE* message with the return SDP to SIP server to crate RTP session with SIP client. At the end, application server sends *join* request to XMS media server in order to joint these two created WebRTC call and SIP call resources, then these two different RTP session channel get connected. The process of multiple clients join a existing conference resource on XMS media server is a similar process as the single call example in Figure 4.2.

According to the documentation provided by Dialogic on XMS 2.1 RESTful API[Dia13], it is only necessary to set *encryption* field as *dtls* and *ice* as field *yes* in the SDP for WebRTC SDP otherwise not set both fields for SIP SDP when creating a call resource on XMS media server(shown at line 4 and 5 in Code Snippet A.3). It makes the interfaces on application server not necessarily be implemented differently for WebRTC and SIP clients.

In this sense, there are *createXMSCall()*, *joinXMSCall()*, *updateLocalSDP()*, *createConference()*, *joinConference()*, *deleteXMSCall()* and *deleteXMSConference()* interfaces(shown in Code Snippet A.3) implemented on application server by the reference of XMS 2.1 RESTful API User's Guide [Dia13].

Using *createXMSCall()* function as example of XMS integration implementation (from line 1 to line 37 in Code Snippet A.3), application server uses *http* Node.js module and *xml2js* Node.js module to implement this interface. Regards to XMS 2.1 RESTful API, creating call resource on XMS media server is a *POST* HTTP request with configuration Extensible Markup Language (XML) as request content. From line 2 to line 7 n Code Snippet A.3 is to generate the XML content. And at line 8, application server calls *http.request()* function with the *option* object and callback function as arguments. The request *option* object has *host*,*port*,*method*,*path*,*headers* fields need to be set. The important point is that the *Content-length* is necessary in the headers field, it is the length of the XML content. These configuration is implemented from line 9 to line 17 in Code Snippet A.3. At line 35, application server calls *req.write()* function to write XML content data in HTTP request and sends it to XMS media server. There is a callback function along with the asynchronous function (*http.request()*). In the callback function, application server needs to check the response data from the HTTP *POST* request for useful SDP data. By using *xml2js* module object *xmlparser* to parse the XML data into JSON, at line 26 and line 27, it is the process to strip the useful data SDP from the response data. It is easier to keep using JSON format for all the data object in Node.js as well as the application client. After the other process at line 28 and line 30, replacing some unsupported character in the SDP for XMS media server by *string.replace()* function, the *createXMSCall()* function will return usful SDP as result.

The rest of the interfaces for XMS media server on application server are similar with *createXMSCall()* interface. *joinXMSCall()* interface is made for single call resource join with another single call resource, it is used when a WebRTC call resource join a SIP call resource for single pair conversation in the prototype system. *updateLocalSDP()* is made to update the SDP of specific call resource on the XMS media server, but it does not work well against the current XMS media server when the WebRTC call resource is made without SDP during the test and development. For this reason, the prototype application server can not use the normal process shown in Figure 4.3 when application server got a SIP *INVITE* message. During the test, after creating the WebRTC call resource on XMS media server without SDP and updating it later with the browser client answer SDP, browser client fails to

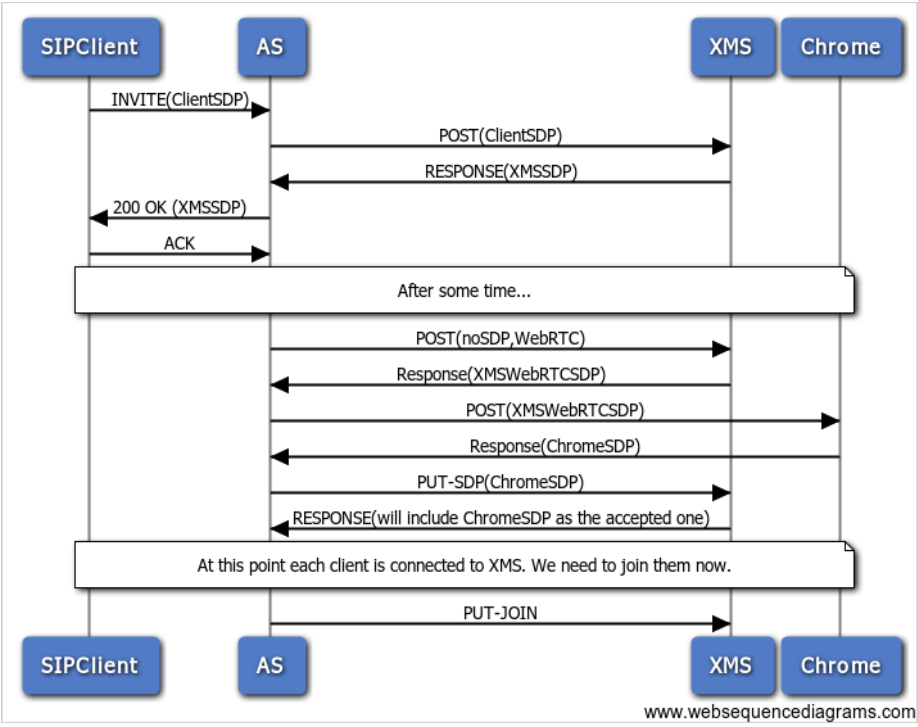

Figure 4.3: Single Call from SIP Client to Browser Client

connect with the call resource on XMS media server in the RTP session. To fix that, the application does the same process for the WebRTC part as the process shown in Figure **??**. It means that no matter the WebRTC client is a receiver or a sender of the call request during the establishment, for WebRTC client itself will treat itself as a sender all the time, then the application server can always get the correct SDP from the client to create the WebRTC call resource on XMS media server. This implementation of fixing solution is based on the WebSocket channels *answer* and *answerInvite* for both client side and server side Soket.IO code blocks with the *self* flag value to see if the client is sender or receiver of the *INVITE* message. This issue is reported to Dialogic team, hope it will be resolved in the future version of XMS media server.

createConference() and *joinConference()* are the corresponding interfaces against *createXMSCall()* and *joinXMSCall()*, they are made for conference resources usage on XMS media server. *deleteXMSCall()* and *deleteXMSConference()* are the delete functions for call resources and conference resources on XMS media server.

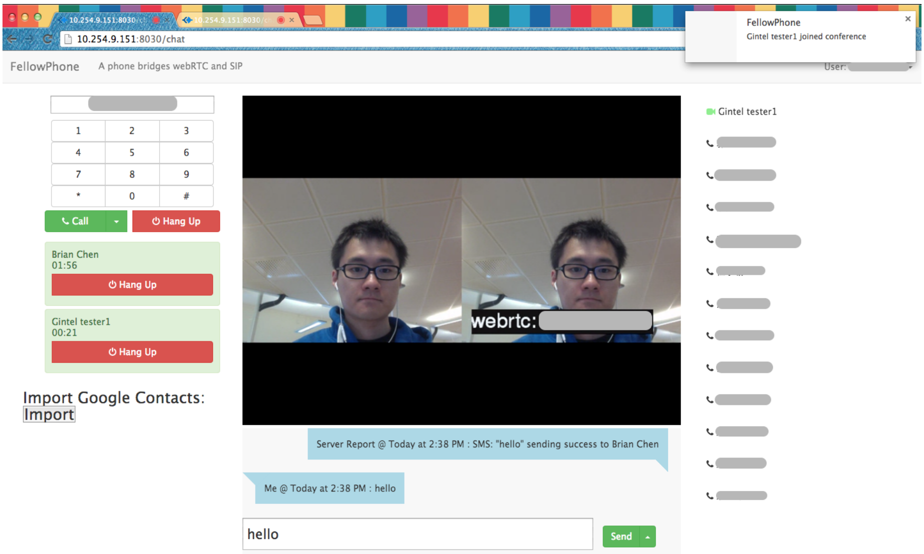

Figure 4.4: Prototype Application in Conference Sending SMS

4.6 Advanced Communication Function Implementation

The most exciting reason for combining WebRTC technology with SIP VoIP network is that there will be more advanced communication functions can be implemented under the power of web technology. There are two main advanced communication functions implemented in the prototype system.

4.6.1 SMS Messaging

SMS messaging is required for normal telephone usage. In the prototype system, it is necessary to have SMS messaging function during the conference if one of the participants is on his mobile phone only. The working prototype is shown in Figure 4.4. There are two peers in one conference, two of them are WebRTC clients(Gintel tester1, Gintel tester2) and one mobile phone client(Brian Chen). The implementation service is based on Mobile Service Gateway (MSG) provided by Gintel AS. It is a message service gateway for SIP clients to send SMS to other SIP clients or physical mobile phone. The implementation is shown in Code Snippet A.4. It uses the same HTTP Node.js module to implement the HTTP request communication with MSG server.

There are two steps to send SMS message. The first one is to get correct MSG user credential by providing the correct *loginDto* object. *loginDto* is a JSON object generated with the user name and password. From line 2 to line 23 in Code Snippet

A.4 is the implementation of this process. It uses another Node.js framework library *https.request()* function because of HTTPS protocol is used on target MSG. This library can be used in the same pattern as *http.request()* described before. However, *https.request* takes only string text in the header fields, then at line 2 in Code Snippet A.4, it converts JSON object into string. At line 13 in Code Snippet A.4, it is shown that if the credential sent to MSG server is correct, it will return a validate cookie in response data. This cookie will be used for second step to send SMS message. From line 25 to line 61 is the implementation of second step, the login credential and cookie need to be set in the header fields *login* and *Cookie*(shown at line 23 and 36). Moreover the message string is set in the HTTP request body at line 59, the *Content-Type* and *Content-Length* in the headers should be set as "application/json" and the length of message string content(it shows at line 33 and line 37).

4.6.2 Files Sharing

Because the RTP media channel is connected with XMS media server for media stream exchange, WebRTC *RTCDataChannel* can not be used in this case. However, considering the prototype system is a real-time communication platform for collaboration working scenario, it is necessary for end points to have some collaboration tools such as files sharing. The screen shoot of file sharing in prototype application is shown in Figure 4.5 and Figure 4.6.

As the screen shoot showing, when sender client uploads files to the application server, application server will directly share these files with the other clients in current conference session. The receiver client can decide whether these files need to be saved or not.

Prototype application uses Delivery.js library to do bidirectional File Transfers For Node.js via Socket.IO. Delivery.js uses Node.js and Socket.IO to make it easy to push files to the client, or send them to the server. Files can be pushed to the client as text (utf-8) or base64 (for images and binary files).[Git14] Since it is based on Socket.IO, it has the similar client and server implementation mechanism as Socket.IO. Once a WebSocket connection is established messages (frames) sent between the client and server contain only 2 additional bytes of overhead. In contrast, a traditional *POST* request, and response, may have headers totaling 871 bytes. This could be a significant addition if many files are being sent, and would be even more significant if files are being divided into batches before being sent to the server. When pushing files to the client, the overhead of traditional polling methods provides an even starker contrast to WebSockets.

At line 9 in Code Snippet A.1, it declares the *delivery* object using Delivery.js API *dl.listen()* with the Socket.IO *socket* object as the argument. From line 36 to line 64 in Code Snippet A.1 is the server side Delivery.js implementation code.

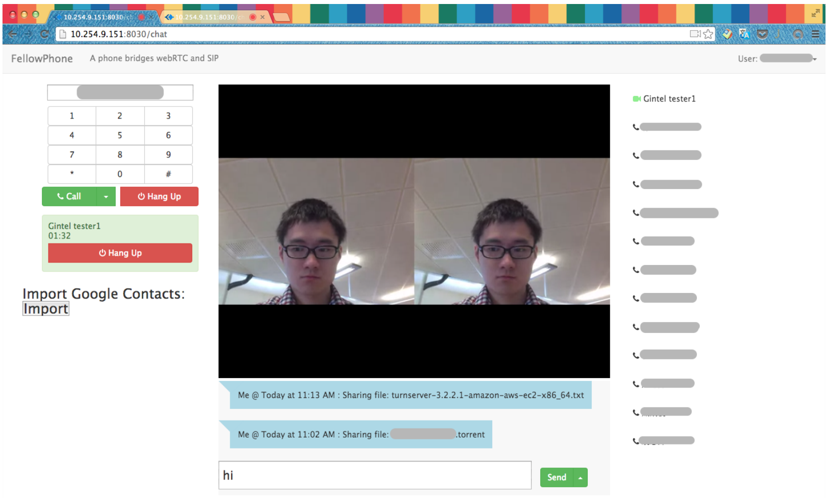

Figure 4.5: File Sharing Sender Client

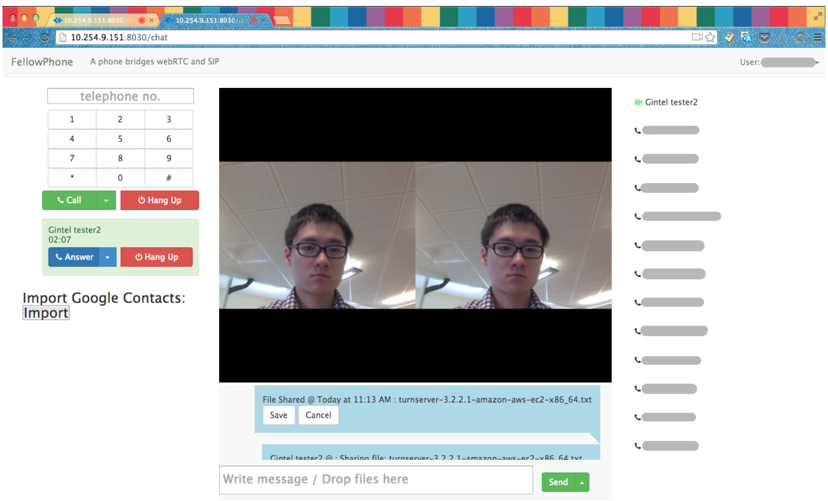

Figure 4.6: File Sharing Receiver Client

It listens to the 'receive.success' WebSocket channel, when the upload files from client are received successfully the application server will make temporary files for the upload files and push these files to every connected clients in sender's current conference session. At line 38, application server uses *fs.writeFile()* function from the Node.js framework *fs* library to write the file byte data got from the client to the application server file system, then at line 46, it uses Delivery.js *delivery.send()* function to push the file to the connected WebScoket client. For security reasons, the temporary files will be removed from the server when all the pushing process finished, it is implemented at line 54 by using *fs.unlink()* function from Node.js framework *fs* library.

After the files are pushed to the client, at line 42 in Code Snippet 4.15, the client side implementation will listen to the same WebSocket channel 'receive.success'. When there is file message from the application server to the client, the client listener will save the files in the runtime memory (it is not best solution for files sharing case, the improvement will be discussed in Chapter 6), then let the user decide if these files need to be downloaded or removed. At line 32 in Code Snippet 4.15, it is the client side sending files function (*delivery.send()*) to the server through WebSocket.

Code Snippet 4.15: Files Sharing in ChatBoardCtrl.js

```
1  angular.module('webrtcDemo.controllers').
2    controller('ChatBoardCtrl',function ($scope,$location,
       $upload,WebSocketService,storage,appId) {
3      ...
4      function b64toBlob(b64Data, contentType, sliceSize) {
5        contentType = contentType || '';
6        sliceSize = sliceSize || 512;
7
8        var byteCharacters = atob(b64Data);
9        var byteArrays = [];
10
11       for (var offset = 0; offset < byteCharacters.length;
           offset += sliceSize) {
12           var slice = byteCharacters.slice(offset, offset +
               sliceSize);
13
14           var byteNumbers = new Array(slice.length);
15           for (var i = 0; i < slice.length; i++) {
16               byteNumbers[i] = slice.charCodeAt(i);
17           }
18
19           var byteArray = new Uint8Array(byteNumbers);
20
21           byteArrays.push(byteArray);
```

```
      }

      var blob = new Blob(byteArrays, {type: contentType});
      return blob;
    }
        ...
    function _upload(files){
      if(channelReday){
        _.each(files,function(file){
          ...
          delivery.send(file);
        });
      }
    }
    ...
    function _initChatBoardView(){
      socket = WebSocketService.getCurrentSocket();
      ...
      delivery = new Delivery(socket);
      ...
      delivery.on('receive.success',function(file){
            $scope.recievedFiles.push(file);
            ...
        });
      ...
    }
    ...
    $scope.saveFile = function(msg,filename){
      var tempFile = _.find($scope.recievedFiles,function(file){
        return file.name == filename;
      });

      var fileBlob = b64toBlob(tempFile.data, tempFile.mimeType);
      saveAs(fileBlob, tempFile.name);
      ...
    }
    ...
  });
```

Because Delivery.js sends files in base64[14] encoding format, on the client applica-

[14] Base64 is a group of similar binary-to-text encoding schemes that represent binary data in an ASCII string format by translating it into a radix-64 representation. The term Base64 originates from a specific MIME content transfer encoding.[Wik13c]

tion, it is necessary to convert base64 encoding string to Web *Blob*[15] data for using the HTML5 W3C *saveAs()* function to download file at line 55 in Code Snippet 4.15. The *saveAs()* function takes Web *Blob* object and file name as the two function parameters. The converting function is implemented at line 4 to line 26 as *b64toBlob()* function in Code Snippet 4.15, it takes base64 string, content type of the file and slice size (in the prototype case it uses default 512 bit) to convert base64 string data to Web *Blob* data. The important function in this code block is *atob()*, it decodes a string of data which has been encoded using base-64 encoding, then slice the byte character data into byte array since *Blob()* function only takes array of data objects, shown at line 24.

Based on these implementation for file sharing, it is even possible to send an E-mail with the sharing files as attachments to the current mobile phone(SIP client) participant in the conference on the application server. Although the mobile phone (SIP client) can not get the sharing files directly in real time, the user can still browser these shared files during the conference to discuss the topic with other participants.

[15]A Blob object represents a file-like object of immutable, raw data. Blobs represent data that isn't necessarily in a JavaScript-native format. The File interface is based on Blob, inheriting blob functionality and expanding it to support files on the user's system.[Net14a]

Chapter 5

Prototype System Deployment

In this Chapter, there will be three main topics discussed because of deploying the prototype system to production usage. They are TURN server deployment for ICE candidate exchange, application server deployment for prototype system working logic and XMS media server deployment for RTP media communication channel.

5.1 TURN Server Deployment

During the development of the prototype system, the test based on XMS media server is not stable at the beginning. There is one way audio issue which happens in the prototype system when WebRTC client init a outbound call to a SIP client(mobile phone). Since it is working fine when the outbound SIP client call into the WebRTC client, after tracing the network log from the XMS media server, the problem is the ICE candidate information got from the original STUN server can not punch the whole on the NAT firewall of the XMS media server. It is normal to replace the STUN server as TURN server to solve this problem because if the STUN server solution is blocked during the media stream exchange from two end point, it will switch to TURN server solution to exchange the media stream through the TURN server to relay all the media traffic.

Moreover, the TURN server solution will work well in the different corporation networks scenario since there will be highly restrict corporation NAT firewall in front of the corporation network. Then TURN server can relay all the media stream to establish the peer to peer connection. After testing prototype system against TURN server instead of STUN server provided by Google (shown at line 3 in Code Snippet 4.3), the one way audio issue is solved and two end clients in different corporation network scenario works fine as well.

To set up TURN server in the production of prototype system, the prototype system uses Amazon Web Service (AWS) Amazon Elastic Compute Cloud (EC2)[1], IP address: 54.187.157.224. The open source project,*rfc5766-turn-server*, is a free open source implementation of TURN and STUN server maintained by Google. It provides the AWS EC2 hosting image, then it is only needed to configure the AWS EC2 virtual instance to open the necessary ports for the TURN server usage. It is shown in the following list:

- TCP 443
- TCP 3478-3479
- TCP 32355-65535
- UDP 3478-3479
- UDP 32355-65535

Furthermore, the TURN server can either use a flat file or a Structured Query Language (SQL) database for configuration and user information. In the prototype system, the TURN server on AWS EC2 will use a flat file for configuration and user information. It is edited in "/usr/local/etc/turnuserdb.conf" by adding an entry on its own line: "my_username:my_password".[Her13] Other configurations need to be completed by following the README file under the hosting instance image directory on AWS EC2. There are several paramters need to be set in "/etc/turnserver.conf" on TURN server.

Besides establishment for TURN server, there are some changes need to be done on the client application as well in order to use this TURN server to fetch the useful ICE candidate information during the peer to peer connection. Compare to the Code Snippet 4.3 with original Google STUN server address, in Code Snippet 5.1, prototype TURN server is set as *iceServer*. The *iceTransports* field is the parameter to force client to use TURN server if it is set to *relay*, but it is only an additional feature purposed by Google Chrome, it is not standard feature and not implemented in Google Chrome either.

Code Snippet 5.1: Using TURN Server on WebRTC Client

```
if (location.hostname != "localhost") {

        pc_config =
        {
          'iceServers': [{
            'urls': 'turn:54.187.157.224',
```

[1] Amazon Elastic Compute Cloud (EC2) is a central part of Amazon.com's cloud computing platform, Amazon Web Services (AWS). EC2 allows users to rent virtual computers on which to run their own computer applications. EC2 allows scalable deployment of applications by providing a Web service through which a user can boot an Amazon Machine Image to create a virtual machine, which Amazon calls an "instance", containing any software desired. A user can create, launch, and terminate server instances as needed, paying by the hour for active servers, hence the term "elastic". EC2 provides users with control over the geographical location of instances that allows for latency optimization and high levels of redundancy.[Wik13a]

```
              'username': 'my_username',
              'credential': 'my_password'
            }],
            'iceTransports': 'relay'
        };
    }
```

5.2 Application Server Deployment

Because the prototype application server is implemented in Node.js, there is no restrict requirements for the operation system platform to deploy the application server if the operation system can install Node.js library and can run V8 JavaScript Engine[2].

It also needs to open the *5060* port to support UDP for SIP stack usage. Then it just needs to use "*node server.js*" command to host the application server for production.

5.3 XMS Server Deployment

The XMS media server is host on a stand alone machine during the development. For deployment reason, it is necessary to map the internal IP address of XMS media server to a public IP address. And it is important to open the necessary port for the XMS media server usage. According to the documentation of Dialogic PowerMedia XMS Installation and Configuration Guide[Dia14],the default PowerMedia XMS configuration uses the following ports:

- TCP: 22, 80, 81, 443, 5060, 1080, 15001
- UDP: 5060, 49152-53152, 57344-57840

Because the application server and XMS media server in the prototype system are host in the corporation network, it only opens necessary port to the public network. During the exchange ICE candidate information for client, the XMS IP address will be the internal IP address by the rule of the corporation network. It is necessary to change the internal IP address into public IP address before pushing the ICE candidate information back to the end point client. This process is implemented at line 29 in Code Snippet A.3, it simply just replaces the internal IP address as public IP address of XMS media server in the SDP content string.

[2]The V8 JavaScript Engine is an open source JavaScript engine developed by Google for the Google Chrome web browser.V8 compiles JavaScript to native machine code (IA-32, x86-64, ARM, or MIPS ISAs) before executing it, instead of more traditional techniques such as interpreting bytecode or compiling whole program to machine code and executing it from a filesystem. [Wik13p]

Chapter 6
Future Work

In this Chapter, there are some future improvement for the prototype system will be discussed. And some future research direction of WebRTC integrated with traditional telephony network will be include as well.

6.1 RTCDataChannel usage

The *RTCDataChannel* API enables peer-to-peer exchange of arbitrary data, with low latency and high throughput.The API has several features to make the most of *RTCPeerConnection* and enable powerful and flexible peer-to-peer communication[Dut14]:

- Leveraging of RTCPeerConnection session setup.
- Multiple simultaneous channels, with prioritization.
- Reliable and unreliable delivery semantics.
- Built-in security (DTLS) and congestion control.
- Ability to use with or without audio or video.

Communication occurs directly between browsers, so RTCDataChannel can be much faster than WebSocket even if a relay (TURN) server is required when 'hole punching' to cope with firewalls and NATs fails.

Because the XMS media server handles all the media stream exchange between the end point clients and it is not support *RTCDataChannel*, the prototype application does not implement *RTCDataChannel* usage in the system. Current using Delivery.js library is good at bidirectional file sharing between clients and server through WebSocket. But is has some disadvantages still. The most apparent disadvantage would be the fact that it bypasses traditional caching methods. Instead of caching based on a file's URL, caching would be based on the content of the Web Socket's message. One possibility would be to cache a base64, or text, version of the file within Redis[1] for fast, in memory, access. And also the sharing files are uploaded

[1]Redis is an open-source, networked, in-memory, key-value data store with optional durability. It is written in ANSI C.[Wik13l]

to the server then push back to the other clients, it takes longer time to finish this process than peer to peer sharing files. Moreover, in current prototype system, the shared files will be temporary pre-stored for the client, it will cause some problem when the sharing file is very big size and it will take over all the memory resource which the client has.

One obvious solution will be implementing the *RTCDataChannel* API on each connected client and create new *RTCPeerConnection* for each pair user in mesh network for only sharing files purpose. Since these new *RTCPeerConnection* is not necessary active during the whole time of application using, they are possible to remove after it is used for sharing files to release more memory recourse for browser clients.

The other solution will be using third party peer to peer sharing services, such as Sharefest[2]. It operates on a mesh network similar to Bit-torrent network. The main difference is that currently the peers are coordinated using an intelligent server. This coordinator controls which parts are sent from A to B and who shall talk with whom. Peer5(http://peer5.com/) Coordinator (or any other solution) is used to accomplish this. Each peer will connect to few other peers in order to maximize the distribution of the file.[Pee14] In this case the client will still keep having single *RTCPeerConnection* with the *RTCDataChannel* on the client, it will fit the work scenario of the prototype system.

6.2 Browser Compatibility

The prototype system is developed on single browser (Google Chrome), it is not tested on other browser. The main reason is that the bug fixing for cross browser platform on WebRTC is too complicated. Since WebRTC is not standard Web API yet, all the browsers have their own implementation. Although most of the WebRTC API calling in the application layer are more or less the same, the issues happen in different ways and they are hard to debug.

Fortunately, Google provides the *adapter.js* script for developer to solve the cross platform issue on Google Chrome and Firefox. It is implemented in WebRTCService in prototype application client. During the test, it still happens some compatibility issues between Google Chrome and Firefox. Current version of prototype system is working fine on both Google Chrome and Firefox browser. However, there are some problem when call is made from Firefox to Google Chrome. The main reason

[2]One-To-Many sharing application. Serverless. Eliminates the need to fully upload your file to services such as Dropbox or Google Drive. Put your file and start sharing immediately with anyone that enters the page. Pure javascript-based. No plugins needed thanks to HTML5 WebRTC Data Channel API

for that, it is the SDP content generated on both platform is not compatible in this work scenario. This issue need to be fixed in the future work.

6.3 Media Server Performance

During the test of the prototype system, the XMS media server performance is quite concerned in the work scenario. The main reason is that the current XMS media server host on a normal laptop machine, it is not powerful enough for high traffic load of the media stream exchange.

The solution for that, it would be easy to host the media server on another powerful server machine. Considering the purpose of the prototype system is to build a system integrated with WebRTC and VoIP network, it is not good solution to keep updating the XMS media server machine. There will be two way to solve this issue in real time communication work scenario. One is to host XMS media server on the third party cloud service, like AWS EC2 instance. Because the third party service will handle the performance, it will rarely have the problem on performance issue. However, this solution is quite expensive when huge number of users make large amount of media stream traffic to the XMS media server. The other solution will be distribute multiple XMS media server to share the traffic load in the prototype system. Then it will be easy to control the performance of the media server but it will cost more physical machine expense.

As a result, the performance of the media server need to be considered as the cost of media server deployment and distribution together.

6.4 Object RTC (ORTC) API for WebRTC

Object RTC (ORTC) is a free, open project that enables mobile endpoints to talk to servers and web browsers with RTC capabilities via native and simple Javascript APIs. The Object RTC components are being optimized to best serve this purpose.[ort14] The mission of ORTC is to enable rich, high quality, RTC applications to be developed in mobile endpoints and servers via native toolkits, simple Javascript APIs and HTML5. It is also a mandate that Object RTC be compatible with WebRTC.

Current WebRTC client is made for browser only, only the smart phone with supported mobile web browser can use these application. According to ORTC, it is possible to make all the smart phone as a WebRTC client. Then there will be no more different signaling implementation because both end point use WebRTC SDP content and WebRTC mechanism. Only one signaling mechanism need to be implemented in this way, it will make less compatibility problem for different types end pints.

There is a related open source project, ortc-lib (https://github.com/openpeer/ortc-lib), it is ORTC C++ library wrapper for WebRTC.This SDK library implementation of the ORTC specification that will enable mobile end points to talk to a WebRTC enabled browser.

If we look at the success of apps like Whatsapp[3] , Tango[4] , Viber[5], Voxer[6], Facebook Messenger[7] etc these are all Over The Top (OTT) applications that have already won in mobile communications. Placing a phone call, is nearly the last thing a teen or twenty-something user is looking to do with their phone nowadays.[Web14d] If the concept of ORTC has been widely spread and implemented, WebRTC and ORTC will become the next generation telecommunication network.

6.5 Advanced function for telecommunication

Since the prototype system bridges the web network and telecommunication network, it is easy to think about how to implement powerful web technology with the telephony use case. For example real time translation in speaking. Translator.js is a JavaScript library built top on Google Speech-Recognition & Translation API to transcript and translate voice and text. It supports many locales and brings globalization in WebRTC.[Kha14] It uses Google Speech-Recognition API to convert user spoken sentence into text string, then uses Google's Non-Official Translation API to translate the text into target language text and use *meSpeak.js* library to play text using a robot voice.

With the social network information, it is easy to get the person profile information of the current conversation user. It is possible to visualize the social network topological diagram to show what is the relationship between two speaking user in the conversation. For the business conference using, it is possible to know the person information and company background information during the conference.

[3]WhatsApp Messenger is a proprietary, cross-platform instant messaging subscription service for smartphones that uses the internet for communication. In addition to text messaging, users can send each other images, video, and audio media messages as well as their location using integrated mapping features.

[4]Tango is third-party, cross platform messaging application software for smartphones developed by TangoME, Inc.

[5]Viber is a proprietary cross-platform instant messaging voice-over-Internet Protocol application for smartphones developed by Viber Media.

[6]Voxer is a San Francisco based mobile app development company most well known for its free Voxer Walkie Talkie app for smartphones.

[7]Facebook Messenger is an instant messaging service and software application which provides text and voice communication. Integrated with Facebook's web-based Chat feature and built on the open MQTT protocol,Messenger lets Facebook users chat with friends both on mobile and on the main website.

Furthermore, with the voice recognition on the web, it is possible to make any useful command through the video/audio conference. For example, one of users want other people to send an E-mail with some attachments to him and mentioned it during the conversation. Then the other user's application will recognize the command and generate the E-mail content at the same time and add the files from the computer as attachments. It will make the normal conference meeting more efficient and less misunderstanding and better for reminding.

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].

[Dia13] Dialogic. Powermedia xms restful api user's guide, 2013. [Online; accessed 26-May-2014].

[Dia14] Dialogic. Dialogic powermedia xms installation and configuration guide, 2014. [Online; accessed 27-May-2014].

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

[Git14] Github. Delivery.js — asynchronous bidirectional file transfers for node.js via socket.io, 2014. [Online; accessed 27-May-2014].

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

[Her13] John Hermanski. Setting up a turn server for webrtc use, 2013. [Online; accessed 27-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.

[Kha14] Muaz Khan. Translator.js, 2014. [Online; accessed 27-May-2014].

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

[Net14a] Mozilla Developer Network. Blob, 2014. [Online; accessed 27-May-2014].

[Net14b] Mozilla Developer Network. Peer-to-peer communications with webrtc, 2014. [Online; accessed 12-May-2014].

[ort14] ortc.org. Ortc (object rtc) object api for rtc – mobile, server, web, 2014. [Online; accessed 27-May-2014].

[Pee14] Peer5. Sharefest — web based p2p file sharing built on webrtc data channels api, 2014. [Online; accessed 27-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].

[Soc03] The Internet Society. Session initiation protocol (sip) basic call flow examples — rfc 3665, 2003. [Online; accessed 25-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].

[Web14d] WebRTC.is. Webrtc, ortc and ott comm, 2014. [Online; accessed 27-May-2014].

[Wik13a] Wikipedia. Amazon elastic compute cloud — Wikipedia, the free encyclopedia, 2013. [Online; accessed 27-May-2014].

[Wik13b] Wikipedia. Apache tomcat — Wikipedia, the free encyclopedia, 2013. [Online; accessed 20-May-2014].

[Wik13c] Wikipedia. Base64 — Wikipedia, the free encyclopedia, 2013. [Online; accessed 27-May-2014].

[Wik13d] Wikipedia. Cascading style sheets — Wikipedia, the free encyclopedia, 2013. [Online; accessed 22-May-2014].

[Wik13e] Wikipedia. Comet (programming) — Wikipedia, the free encyclopedia, 2013. [Online; accessed 2-June-2014].

[Wik13f] Wikipedia. Ecmascript — Wikipedia, the free encyclopedia, 2013. [Online; accessed 2-June-2014].

[Wik13g] Wikipedia. Java apis for integrated networks — Wikipedia, the free encyclopedia, 2013. [Online; accessed 20-May-2014].

[Wik13h] Wikipedia. Jsonp — Wikipedia, the free encyclopedia, 2013. [Online; accessed 2-June-2014].

[Wik13i] Wikipedia. Mesh networking — Wikipedia, the free encyclopedia, 2013. [Online; accessed 22-May-2014].

[Wik13j] Wikipedia. npm (software) — Wikipedia, the free encyclopedia, 2013. [Online; accessed 23-May-2014].

[Wik13k] Wikipedia. Oauth — Wikipedia, the free encyclopedia, 2013. [Online; accessed 23-May-2014].

[Wik13l] Wikipedia. Redis — Wikipedia, the free encyclopedia, 2013. [Online; accessed 27-May-2014].

[Wik13m] Wikipedia. Representational state transfer — Wikipedia, the free encyclopedia, 2013. [Online; accessed 26-May-2014].

[Wik13n] Wikipedia. Session border controller — Wikipedia, the free encyclopedia, 2013. [Online; accessed 2-June-2014].

[Wik13o] Wikipedia. Softphone — Wikipedia, the free encyclopedia, 2013. [Online; accessed 23-May-2014].

[Wik13p] Wikipedia. V8 (javascript engine) — Wikipedia, the free encyclopedia, 2013. [Online; accessed 27-May-2014].

[Wik13q] Wikipedia. Wildfly — Wikipedia, the free encyclopedia, 2013. [Online; accessed 20-May-2014].

[Wik13r] Wikipedia. Wireshark — Wikipedia, the free encyclopedia, 2013. [Online; accessed 23-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

Appendix A

A.1 Socket.IO Implementation Script

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

```
SocketManager.prototype.listen = function(server){
  ...
  io = socketio.listen(server);
  io.sockets.on('connection', _handlerSocket);
  _handlerSip();
}

function _handlerSocket(socket) {
  var delivery = dl.listen(socket);
    ...
  socket.on('sip',function (data){
    switch(data.type){
      case 'register':
        if(data.username != ""){
          gw.register(data.content.browserClient,function(result){
            socket.emit('sip',result);
          });
        }
        break;
      ...
    }
  });

  socket.on('webrtc', function (data) {
        ...
    });

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

```
    ...
  });

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

  delivery.on('receive.success',function(file){
    ...
    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(sendingClient.conf_id && client.conf_id ==
              sendingClient.conf_id && client.username !=
              sendingClient.username){
            ...
            client.delivery.send({
              name: file.name,
              path : './' + file.name
            });

          }
        });

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

}
```

A.2 SIP Implementation Script

Code Snippet A.2: sip.js on Application Server

```
util.inherits(SipGateway, EventEmitter);
...
function createRegister(user){
  return {
    method: 'REGISTER',
    uri: 'sip:' + user.hostname,
    headers:
    {
```

```
      'call-id': user.callid,
      cseq: {method: 'REGISTER', seq: ++user.seq},
      from: {name: '', uri: 'sip:' + user.name + '@' + user.hostname, params: { tag: user.tag }},
      to: {name: '', uri: 'sip:' + user.name + '@' + user.hostname},
      expires: 3600,
      contact:  [{
        uri: 'sip:' + user.name + '@'+ hostPublicAddress + ':' + hostPort
      }]

    }
  }
}
function createInviteACK(rs,client,sdp){
  var uri = rs.headers.contact[0].uri.split(';');
  if(uri[0].split(':').length != 3){
    uri[0] = uri[0] + ':5060';
  }
  ...
}
...
SipGateway.prototype.init = function () {
  ...
  registry = {};
  sip.start({
    port: hostPort,
    logger: {
      ...
    },
    publicAddress: hostPublicAddress,
    tcp: false
  },
  function(rq) {
    try {
      if(rq.method === 'REGISTER') {
        ...
      }
      else if(rq.method === 'INVITE') {
        ...
        var rs = sip.makeResponse(rq, 100, 'Trying');
        sip.send(rs);
        if(contact) {
          ...
```

```
          self.emit('SIPREMOTE',{
            type: 'INVITE',
            content:{
              fromNumber: sip.parseUri(rq.headers.from.uri).
                  user,
              toNumber: username,
              inviteRequest: rq
            }
          });
        }
        ...
  });
}
function _register(client,callback){
  var rq = createRegister(client);
  sip.send(rq,function(rs){
    if(rs.status === 401){
      var user = client;
      var creds = { user: user.name, password: user.password
          , realm: user.hostname };
      rq.headers['cseq'].seq++;
      ...
      digest.signRequest(creds, rq, rs, creds);
      sip.send(rq,function(rs){
        if(rs.status === 200){
          client.authorization = rq.headers.authorization;
          if(!client.registerTimer){
            client.registerTimer = setInterval(function(){
              console.log('register timer');
              _register(client);
            },parseInt(rs.headers.expires)*1000);
          }
                    ...
        }else{
          ...
        }
      });
    }else if(rs.status === 200){
      ...
    }
  });
}
SipGateway.prototype.register = function(client,callback){
    ...
  _register(registry[client.name],callback);
```

```
}
...
module.exports.SipGateway = SipGateway;
```

A.3 XMS Implementation Script

Code Snippet A.3: xms.js on Application Server

```
XmsManager.prototype.createXMSCall = function(data,callback) {
  var requestContent = "<web_service version=\"1.0\">";
  ...
  if(data.callType === 'webrtc'){
    requestContent += " encryption=\"dtls\"" + " ice=\"yes\"";
  }
  ...
  var req = http.request({
    host: xmsAddress,
    port: xmsPort,
    method: 'POST',
    path: xmsPath + 'calls?appid=' + xmsAppId,
    headers: {
      'Accept' : 'application/xml',
      'Content-Type' : 'application/xml',
      'Content-Length' : requestContent.length
    }
  }, function(res) {
    var resData = '';
    res.setEncoding('utf8');
    res.on('data', function (chunk) {
      resData += chunk;
    }).on('end', function() {
      if(resData != ''){
        xmlparser.parseString(resData,function(err,result){
          var xmsSdp = result['web_service']['call_response'][0]['$'].sdp;
          var id = result['web_service']['call_response'][0]['$'].identifier;
          var regex = new RegExp(xmsAddress,"g");
          pub_xmsSdp = xmsSdp.replace(regex,xmsPublicAddress);
          callback(pub_xmsSdp.replace(/\n/g,"\r\n"),id);
        });
      }
    });
```

```
  });
  req.write(requestContent);
  req.end();
}
...
module.exports.XmsManager = XmsManager;
```

A.4 MSG Implementation Script

Code Snippet A.4: msg.js on Application Server

```
MsgManager.prototype.login = function(loginDto,success,fail)
    {
  var loginStr = JSON.stringify(loginDto);
  ...
  var req = https.request(options, function(res) {
    var resData = '';
    res.setEncoding('utf8');
    res.on('data', function (chunk) {
      resData += chunk;
    }).on('end', function() {
      if(resData != ''){
        var jsonObject = JSON.parse(resData);
        if(jsonObject.code === 200){
          success(jsonObject,res.headers['set-cookie'][0]);
        }else{
          fail(jsonObject);
        }
      }
    });
  });
  req.write(loginStr);
  req.end();
  ...
}

MsgManager.prototype.sendSMS = function(organization,login,
    cookie,msgObj,success,fail){
  var msgStr = JSON.stringify(msgObj);
  var options = {
    host: msgRestUrl,
    path: '/your/url/here/sms',
    method : 'POST',
    headers : {
      "Accept" : "application/json",
      "Content-Type" : "application/json",
```

```
      "organization" : organization,
      "login" : login,
      "Cookie" : cookie,
      'Content-Length' : msgStr.length
    }
  };

  var req = https.request(options, function(res) {
    var resData = '';
    res.setEncoding('utf8');

    res.on('data', function (chunk) {
      resData += chunk;
    }).on('end', function() {
      if(resData != ''){
        var jsonObject = JSON.parse(resData);
        console.log('MSG:SENDSMS:jsonObject:',jsonObject);
        if(jsonObject.code === 200){//TODO: check report
            values as true
          success();
        }else{
          fail();
        }
      }
    });
  });
  req.write(msgStr);
  req.end();
}

module.exports.MsgManager = MsgManager;
```

A.5 WebRTC in Dart

Code Snippet A.5: 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 B

B.1 AngularJs Files Structure

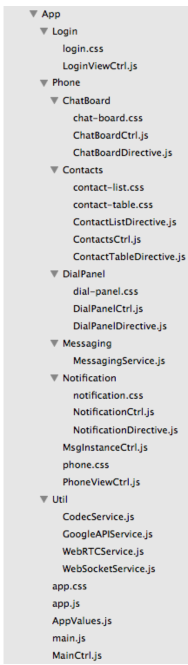

Figure B.1: Prototype Application AngularJs Files