# **REAL TIME** **SECURED COMMUNICATION SYSTEM USING WEBRTC**

**Peter Chidubem Atunalu - 144013**

Submitted to the

Graduation Projects Examination Jury

in partial fulfillment of the requirements for the degree of

Bachelor of Science

In

Computer Engineering

European University of Lefke

May 2018

**Approval of the Computer Engineering Department**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Assoc. Prof. Dr. Hüseyin Ademgil

Chairman/Director

This is to certify that we have read this graduation project and that in our opinion it is fully adequate, in cope and quality, as an Undergraduate Project.

…………………………….

Assist. Prof. Dr. Zefer Ernel

Supervisor

Members of the examining committee

Name Signature

1. Assoc. Prof. Dr. Hüseyin Ademgil …….………………………

2. Assist. Prof. Dr. Cem B. Kalyoncu …….………………………

3. Assist. Prof. Dr. Ferhun Yorgancıoğlu …….………………………

4. Assist. Prof. Dr. Vesile Evrim …….………………………

5. Dr. Ersin Çağlar …….………………………

Date: …….………………………

# Abstract

**REAL TIME SECURED COMMUNICATION SYSTEM USING WEBRTC (RTConn)**

by

Peter Chidubem Atunalu

Computer Engineering Department

European University of Lefke

Supervisor: Assist. Prof. Dr. Zafer Erenel

In a world of computers and cell phones, the need for effective and rapid communication has never been stronger than it is today. RTConn is a real time secured communication system that connects multiple users across the globe using native web API (WebRTC). RTConn allow users connect with each other through video, text, files and screen sharing in real time. Allowing collaboration and connection with friends, family, coworker and any one of interest. By eliminating the need of a server during communication and the requirement to download an external plugin, RTConn increases efficiency, speed and security of data transmitted from one user to another by connecting users’ device together in a pair to pair network.

**Keywords:** WebRTC, VOIP, Internet programming, socket programming, Node.Js, MongoDB, Redis, Video Chat, NoSQL

# Acknowledgments

I would like to thank all those who made this project a success, through their teachings and advice. To all my colleges in school, my family and friends. I want to specially thank my project supervisor Asst. Prof. Dr. Zefer Ernel, and all instructors at the faculty of engineering whose teaching has made it possible for me to complete this project; Asst. Prof. Dr. Ferhun Yorgancıoğlu, Asst. Prof. Dr. Cem Kalyoncu and also my course advisor Asst. Prof. Dr. Vesile Evrim.

# Table of Contents

[**REAL TIME** **SECURED COMMUNICATION SYSTEM USING WEBRTC** 2](#_Toc515011971)

[Abstract 5](#_Toc515011972)

[Acknowledgments 6](#_Toc515011973)

[Table of Contents 7](#_Toc515011974)

[List of Figures 9](#_Toc515011975)

[List of Tables 10](#_Toc515011976)

[1. INTRODUCTION 11](#_Toc515011977)

[1.1 Definition 11](#_Toc515011978)

[1.1.1 Classic VoIP 11](#_Toc515011979)

[1.1.2 WebRTC 13](#_Toc515011980)

[1.2 Goal 16](#_Toc515011981)

[1.3 Required software 18](#_Toc515011982)

[2. LITERATURE SURVEY 20](#_Toc515011983)

[3. BACKGROUND INFORMATION 22](#_Toc515011984)

[3.1 Overview 22](#_Toc515011985)

[3.2 The general flow 22](#_Toc515011986)

[3.2.1 Connect peers 22](#_Toc515011987)

[3.2.2 Initiate signals 23](#_Toc515011988)

[3.2.3 Find candidates 24](#_Toc515011989)

[3.2.4 Negotiate media sessions 24](#_Toc515011990)

[3.2.5 Start RTCPeerConnection streams 25](#_Toc515011991)

[3.3 Signaling 25](#_Toc515011992)

[3.1 STUN server 26](#_Toc515011993)

[3.2 TURN Server 27](#_Toc515011994)

[3.3 Using WebSockets 28](#_Toc515011995)

[3.4 WebRTC APIs 31](#_Toc515011996)

[3.4.1 GetUserMedia 31](#_Toc515011997)

[3.4.2 MediaStream 31](#_Toc515011998)

[3.4.3 RTCPeerConnection 32](#_Toc515011999)

[3.4.4 DataChannel 32](#_Toc515012000)

[4. IMPLEMEMTATION DETAILS 34](#_Toc515012001)

[4.1 Setup page 34](#_Toc515012002)

[4.1.1 Step 1: Adding the HTML 35](#_Toc515012003)

[4.1.2 Step 2: Prefix getUserMedia API to work on all browser 36](#_Toc515012004)

[4.1.3 Step 3: Obtain user MediaDevices 37](#_Toc515012005)

[4.1.4 Step 4: Obtain user media stream 37](#_Toc515012006)

[4.2 Connect Peers 38](#_Toc515012007)

[4.2.1 Join socket.io rooms 38](#_Toc515012008)

[4.2.2 Initiate peer connection and dataChannel 40](#_Toc515012009)

[4.2.3 Handling signal messages 41](#_Toc515012010)

[4.2.4 Attach stream to video element source 42](#_Toc515012011)

[4.3 Add video and audio media streams 43](#_Toc515012012)

[4.4 Add instant text messaging using dataChannel 44](#_Toc515012013)

[4.5 Transferring Files 48](#_Toc515012014)

[4.6 Screen sharing 50](#_Toc515012015)

[Chrome 50](#_Toc515012016)

[4.7 Conferencing 51](#_Toc515012017)

[5. CONCLUSION AND FUTURE RECOMMENDATION 53](#_Toc515012018)

[5.1 Benefits 53](#_Toc515012019)

[5.2 Future works 54](#_Toc515012020)

[6. REFERENCES 55](#_Toc515012021)

[7. Appendix A. WebRTC 1.0 APIs 56](#_Toc515012022)

[State Definition 58](#_Toc515012023)

[Peer-to-Peer Data API 60](#_Toc515012024)

[Interface RTCDataChannel Interface Methods 60](#_Toc515012025)

[RTCDataChannel Interface Attributes 61](#_Toc515012026)

# List of Figures

[Fig 1.1: Classic VoIP Architecture 12](#_Toc515008502)

[Fig 1.2: WebRTC Trapezoid 1 WebRTC Trapezoid 14](#_Toc515008503)

[Fig 1.3: The WebRTC Triangle 1 15](#_Toc515008504)

Fig 3.1 RTConn Peer Connection exchanging signaling message …………………………..24

Fig 3.2: RTCPeerConnection Ready To Stream Live Data …………………………………25

Fig 3.3: RTCPeerConnection discovering it IP and Port for a STUN serve ………….……..27

Fig: 3.4: WebRTC Connected Using a TURN Server……..…………………………………28

Fig 3.5: Standard WebRTC architecture..…………………………………………………….30

Fig 4.1: Setup Page HTML (code)...…….……………………………………………………35

Fig 4.2: Make getUserMedia work same on all browser (code)……………………………...36

Fig 4.3: Get user media devices and append to select option (code) …………………………37

Fig 4.4: Obtain User webcam and audio function (code)……………………………………..37

Fig 4.5: getUserMedia Permission box (demo)………………………………………………37

Fig 4.6: Setting Room Name and Initiating Socket.io (code)……….………………………..39

Fig 4.7: Starting the calling Process (Code)………………………………………………….39

Fig 4.8: Initializing the PeerConnection and DataChannel (code)……………...…………….40

Fig 4.9: Handling Signal Message (code)………………………………………………..…...41

Fig 4.10: Server On Signal Message Handler (code)…………………………………………42

Fig 4.11: Attach the stream to the setup video src (code)……………………………………...42

Fig 4.12: Setup page after user video and audio stream has been attach (demo)…………….43

Fig 4.13: Adding Remote Media Stream to the Remote Video Stream (code)………………44

Fig 4.14: Peer-to-Peer video chat (demo)…………………………………………………….44

Fig 4.15: Chat and File transfer HTML (code)…………………………………………….…45

Fig 4.16: Sending instant text message with the dataChannel (code)…………………………46

Fig 4.17: Handling dataChannel incoming messages and files (code)……………………….47

Fig 4.18: Chat messaging (demo)…………………………………….………………………48

Fig 4.19: WebRTC File Transfer……………………………………………………………..49

Fig 4.20: Slicing and sending files via dataChannel (code)…………………………………50

# List of Tables

Table 1.1: Classic VoIP or WebRTC ?....................................................................................16

Table 3.1: Comparison of TCP vs UDP vs SCTP……………………………………………32

Table A-1. RTCConfiguration dictionary members……………………………………….....56

Table A-2. RTCIceServer dictionary members………………………………………………56

Table A-3. RTCIceTransports enumeration values………………………………………..…57

Table A-4. RTCPeerConnection attributes…………………………………………………..57

Table A-5. RTCSignalingState………………………………………………………………58

Table A-6. RTCIceGatheringState…………………………………………………………...59

Table A-7. RTCIceConnectionState………………………………………………………….59

Table A-8. RTCDataChannel Methods……………………………………………………....61

Table A-9. Attributes…………………………………………………………………………61

Table A-10. RTCDataChannelInit dictionary………………………………………………..62

Table A-11. RTCDataChannelState enumeration values…………………………………….63

# INTRODUCTION

## Definition

### Classic VoIP

Voice over Internet Protocol (VoIP) is a form of communication that allows you to make phone calls over a broadband internet connection instead of typical analog telephone lines. Basic VoIP access usually allows you to call others who are also receiving calls over the internet. Interconnected VoIP services also allow you to make and receive calls to and from traditional landline numbers, usually for a service fee. Some VoIP services require a computer or a dedicated VoIP phone, while others allow you to use your landline phone to place VoIP calls through a special adapter. VoIP is becoming an attractive communications option for consumers. Given the trend towards lower fees for basic broadband service and the brisk adoption of even faster internet offerings, VoIP usage should only gain popularity with time. However, as VoIP usage increases, so will the potential threats to the typical user. While VoIP vulnerabilities are typically similar to the ones users face on the internet, new threats, scams, and attacks unique to IP telephony are now emerging.

Voice over Internet Protocol (VoIP) is a way of voice communication using the Internet Protocol (IP) for the transport. Public Switched Telephone Networks (PSTN) is the protocol used by traditional phone networks (see Fig 1.1), and it uses circuit-switching. Information are reserved in the whole communication channel for the duration of the call in Circuit-Switching,. But in packet-switching which is used by the Internet Protocol (IP), data is divided into one or more packets and transmitted digitally. In packet switching every packet knows its origin and destination, and it may travel to it’s destionation via different paths over the network. Developing VoIP requires so many protocols, ranging from those needed to do signaling for call initialization and more, to the transportation of real-time voice across the network, and down to quality-of-service-aware routing, and much more.



Fig 1.1: Classic VoIP Architecture

VoIP protocols usually uses Real-time Transport Protocol (RTP) as the protocol for media and speech stream. Real-time Transport Protocol (RTP) uses User Data-gram Protocol as the transport protocol. Voice signaling data are usually built using Transmission Control Protocol (TCP) as the transport protocol. Routing and network-level addressing is been provided by the IP layer, while the data-link layer protocols direct and govern the transmission of the data over the physical layer. When choosing between VoIP, VoATM transport, and Layer 2 VoFR integration with other voice or multimedia applications is the major factor of making a decision. When it comes to voice-over-packet in the industry today, and to implementing voice applications VoIP is the predominant form. The complete intranet IP IP infrastructure and internet is used for routing, thus the design any type of calling in a VoIP network becomes simpler. The major problem with VoIP is that it is expensive as compared to Web Real-Time Communication (WebRTC). In this project we will discover the WebRTC API and how it can be larveage to build a real-time communication system.

The appearance of VoIP comes at a juncture when telecommunications system has already turned into a large-scale, complex system with multiple, competing infrastructures. VoIP, however, greatly augments the nested complexity by affording a technology that enables multiple architectures and business models for delivering the same voice (and often converged voice and data) service, while remaining agnostic to the underlying infrastructure. The VoIP-enabled architectures have very different capabilities and costs from one another. Many do not – or cannot – support social regulations such as emergency 911, wiretapping and disability access. Most exploit the economic arbitrage opportunities by evading access charges and universal service contributions. Added to this is the combination of reduced asset specificity due to VoIP’s layered architecture and a global standard based ubiquitous IP technology that frees the service providers of the need to own the delivery infrastructure, and enables them to offer service from anywhere globally. Such a misalignment – between regulatory obligations and technical capabilities – has the potential to incubate large-scale systemic failures due to lack of coordination between the local optimization focused private markets and the highly compartmentalized public institutions.

### WebRTC

Web Real-Time Communication (WebRTC) is a new web API which allows browsers to communicate with each other in real time, using a peer-to-peer architecture. WebRTC is a consent-based, audio, video, and data secure peer-to-peer communication API between modern browsers. It enables, organizations, businesses and web developers to develop multimedia real-time applications with no need for external plug-ins; this has never been possible before. WebRTC is an evolution in web applications development, it combines wo historically separated systems together VoIP and web development.

The World Wide Web Consortium (W3C) and the Internet Engineering Task Force (IETF) are jointly defining the JavaScript APIs (Application Programming Interfaces) available at <http://www.w3.org/2011/04/webrtc/> and <https://datatracker.ietf.org/wg/rtcweb/documents/>, the standard HTML5 tags, and the underlying communication protocols for the setup and management of a reliable communication channel between any pair of next-generation web browsers. The standardization aim to define an API that allows a web application that runs on any device, to interchange media and data with a remote party in real-time and in a peer-to-peer fashion through a secure access to the input peripherals of the device (like the webcams and microphones). This will allow for development of several types of application, ranging from a simple audio communication to video-conference with multiple people, providing these functions “out of the box” as part of the basic capabilities of the browser.

In the classic web architecture, an HTTP (Hypertext Transfer Protocol) request for content is sent by the client (browser) to the server, and the server will then respond with a payload that includes all the information the client requested. Uniform Resource Identifier (URI) or Uniform Resource Locator (URL) entity is closely related to the resources provided by the server. The server may include some JavaScript code/file in the web application scenario, when an HTML page is sent to the client. The JavaScript code can interact with the browsers via a standard JavaScript APIs. WebRTC extends the client-server semantics by introducing a peer-to-peer communication paradigm between browsers. The most general WebRTC architectural model draws its inspiration from SIP (Session Initiation Protocol) Trapezoid (shown in Fig 1.2).

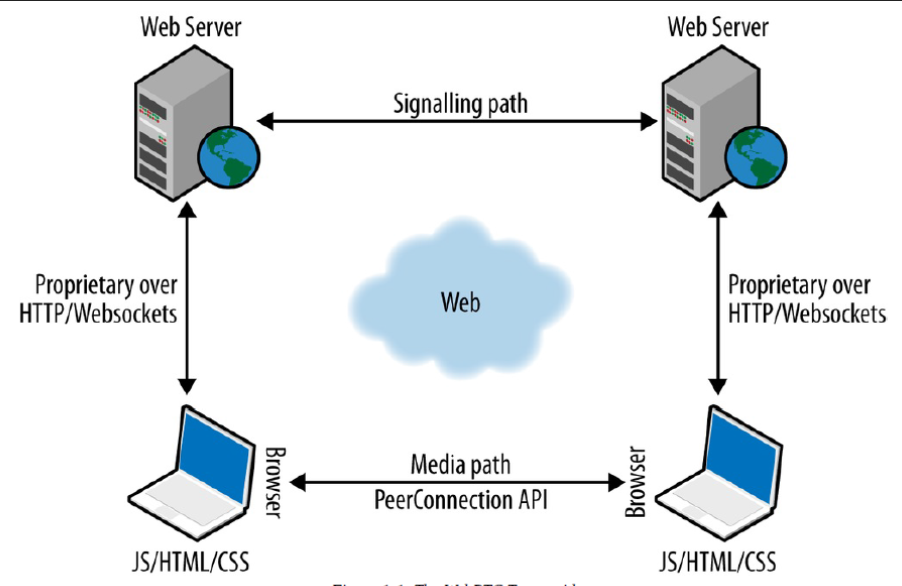


Fig 1.2: WebRTC Trapezoid 1 WebRTC Trapezoid

In the WebRTC Trapezoid model, the web application is been run by the browsers, that is been downloaded from different or same web server. Signaling messages are transported using HTTP or WebSocket protocol via web servers. As needed the server is used to initialize, modify, translate, or manage, and close communications.

WebRTC does not standardized the signaling method to be used between browser and server, leaving developers to implement signaling as part of the application as they see fit. Multimedia flows directly from and to browsers without any intervening servers via the **RTCPeerConnection** API. Direct peer-to-peer connections provides lower latency, thereby making video streaming, text messaging, sensor data feeds, and so on, appear faster. Using a standard signaling protocol such as SIP, XAMP or Jingle (XEP-0166), the two web servers can communicate with each other. Otherwise, they can use a proprietary signaling protocol.

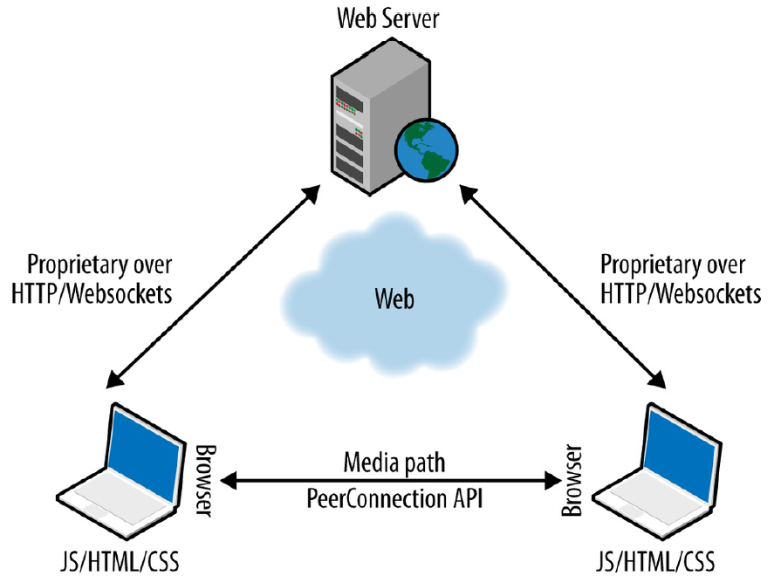


Fig 1.3: The WebRTC Triangle 1

A WebRTC web application (written as a mix of HTML and JavaScript) interacts with web browsers through the standardized WebRTC API, which allows it to properly discover and control the real-time functions of the browser. RTConn interacts with the browser, using both WebRTC and other standardized APIs, both proactively (e.g., to check browser capabilities) and reactively. WebRTC API provides a wide set of functions, like connection management (in a peer-to-peer architecture), media encoding/decoding, session negotiations, media control, firewall and Network Address Translator (NAT), etc.

Table 1.1: Classic VoIP or WebRTC?

|  |  |  |
| --- | --- | --- |
|  | VoIP | WebRTC |
| **Signaling** | SIP or H.323 in most cases | Undefined |
| **Media Transport** | RTP/RTCP | RTP/RTCP |
| **Security** | SRTP in SIP,H.235 in H.323 | SRTP |
| **NAT traversal** | STUN/TURN/ICE in SIP,H.450.x in H.323 | STUN/TURN/ICE |
| **Video codecs** | H.263, H.264 | VP8 |
| **Voice codecs** | G.7xx series of codecs, and some more | G.711, iLBC, iSAC |

## Goal

The aim of this project is to build a real-time communication system that can be deployed on cheap consumer device, while enhancing latency, speed, and security. Now more than ever been able to connect with friends, family, coworkers, instructors, students, and business partners in real-time has become more important than ever before. We were all once together, celebrating life and love, then distance came in and set us apart from our family, friend and work. With so many social media applications like Facebook, Instagram and Whatsapp most of the problem that arise from connect with those we love have been solved. But in other to use this services one most have to sign up for service and be connected with the person you wish to speak with. In other to deliver a real time system that can transmit video companies usually need spend a huge sum of money on hardware, and for this reason most organization depends on other large organization for real time communication.

Some time ago, expensive and complex equipment and expertise is required to develop a video calling or conferencing application. As of today the whole infrastructure required to connect with some remotely anywhere from in the world can be carried in the pocket. With standard Internet connectivity and basic hardware you can easily participate, and host very nice video conferencing sessions using your smart phone and mobile device including your computer. Due to the advent and development of Voice over IP (VoIP), Video conferencing become more common and more accessible to everyone. Video data packets along with voice packets and other types of data (Packet), are transmitted via the Internet, therefore it made voice and video communication free.

The huge benefits of connecting through video calling is a better and efficient meetings with a stronger sense of connection among contacts and the exchange of non-verbal expression and communications, both within and between friends and family, customers, as well as with companies. Face-to-face connection adds non-verbal communication to meetings on a more personal level, the exchange permits participants to build a stronger sense of familiarity with each other, even though they may never be able to meet in person.

Previously, when it came to live video broadcasting, only very well-funded companies where able afford the cost to produce and distribute a working system. This companies usually need need a large audience to make ends needs, due to high production and distribution costs. Therefore they have to broadcast contents for the lowest common denominator; for this reason, regular-season like Little League Baseball are not usually broadcast live — yet. When live video broadcasting was introduced to mobile devices, it revolved the way we develop real-time solution. Bringing down the costs of real-time system to the expense of having a smart phone and an Internet connection, changes people broadcast and what they choose to watch? If we fast-forward to ten years, there will be huge increase in number of services that offers live video broadcasting services; this days there are people on the internet that are sharing everything from cute puppies to books, and other things. A lot of viewers spent more than a billion minutes in total watching “Twitch plays Pokémon” stream.

We are witnessing an increasing number of live broadcasts, breaking news and angles that were never previously available to us. Today even war are streamed live for the world to view. Individuals who are able to creating their own news and sharing it globally are now competing with traditional media companies, with the world as their audience. Broadcasters and publishers have grown interest in mining live video streaming platforms for content they could never have captured themselves, since the advent of consumer live streaming. They mine content created by people like you and me, via modern consumer tech and shared to the world in real-time as an interactive video.

## Required software

WebRTC**:** WebRTC is a free, open project that provides browsers and mobile applications with Real-Time Communications (RTC) capabilities via simple APIs. The WebRTC components have been optimized to best serve this purpose.

**USE:** WebRTC is the tech and backbone of the communication been built in this project.

Node.js:Node.js is an open source,server-side platform built on Google Chrome's JavaScript Engine (V8 Engine). Ryan Dahl developed Node.js in 2009. Node.js is a run-time server-side and networking applications used for development of cross-platform application. Node.js applications can be run on any operating that has a Node.js runtime installed and are written in JavaScript.

**USE:** Node.js is used as the server and signaling server for this application. It contains the business logic used in connecting to separate used at remote place apart from each other.

MongoDB: MongoDB is a free and open-source no realtion database (NoSQL) and distributed database at its core, with high availability, horizontal scaling, and geographic distribution are built into it.

**USE:** Used to store user and system data, like username and password.

Socket.io:It is a websocket library written in JavaScript for active bidirectional networking or communication between the server-side (Node.js) and the browser. It abstracts away websockets communication schemes, based upon the capabilities of the browser. With also some convenient features like broadcasts and multicasts, that are beyond the features of the plain html5 websockets.

**USE:** Socket.io is used in this project to transmit signaling message across to all users of RTConnm, and to logically separate user in into rooms and namespace.

Amazon EC2:Amazon Elastic Compute Cloud (Amazon EC2) is a cloud computing service from Amazon. It provides on-demand and salable computing capacity in the Amazon Web Services (AWS) cloud ecosystem. Amazon EC2 cancels out the need for upfront investment in hardware, so development and deployment of applications can become faster and affordable. Amazon EC2 can be used to launch as many or as few virtual servers as need, configure security and networking, and manage compute storage. Amazon EC2 allows applications to scale up or down in order to handle changes in requirements or spikes in popularity, eliminating the need to forecast traffic.

**USE:** EC2 provides the necessary infrastructure need for the back-end and signaling server.

Docker:Docker is a container tool, that makes it easier to design, create, deploy, and run applications. Using containers tool like docker, developers can package up application with all of the parts needed, such as system libraries and application dependencies, and ship it all out as one package. Thanks to the container, we can be rest assured that the application will run on any other Linux machine without any regard to any customization that machine might have and is different from the machine used for writing and testing the application. The statement “it works on my machine” has been eliminated by docker once and for all.

**USE:** Docker help package the application for continuous delivery and deployment pipeline, and makes it easy to deploy the application.

# LITERATURE SURVEY

If you look at the browser, with a few exceptions it currently replaces all of the other platforms and environments for the applications we use. One such exception are applications that require bidirectional voice and video calling capabilities. While you can get these by way of a Flash plugin, this has its drawbacks. WebRTC is a free, open project, supported by Google, Mozilla, Opera and others.

In the beginning of 2010, Google finalized its acquisition of On2, a video codec company that has developed the VP series of codecs, with the latest one being VP8. On2 has always positioned its codecs as a patent free replacement to the H.26x series of codecs, which were standardized, patented and widely used. It then went about opening On2’s technologies to the world and open sources VP8 under the name of WebM. The idea was to replace H.264 for web videos and by that, reduce patent costs for everyone – especially Google itself.

Google went on and during 2010 acquired Global IP Solutions (GIPS), a company known for their media frameworks – a piece of technology that makes developing VoIP and video calling applications easier. At the time, GIPS had a large market share in VoIP, which caused most of the industry to scurry and search for alternative solutions. As with On2, Google took GIPS’ assets and open sourced them. This time, with an interesting twist: they threw out all voice and video codecs that had patent owners and added an additional layer – a JavaScript API as an integration layer to web browsers. The idea? Have bidirectional media processing and media coding technologies available in every browser. It then went on to push it as a standard at the IETF and W3C, where such standards live. It is now called WebRTC.

VOIP Techonology

There are two fundamental technologies that are necessary for the existence of VoIP. The first, and most widely used, is the telephone. The second technology is the Internet. The telephone was as direct result of the (independent) work of Alexander Gram Bell and Elisha Gray in the 1870s . The first regular telephone exchange was established in New Haven in 1878. Early telephones were leased in pairs to subscribers.

The subscriber was required to put up their own line to connect with another. In 1889, Almon B. Strowger, a Kansas City undertaker, invented a switch that could connect one line to any of 100 lines by using relays and sliders. This switch became known as "The Strowger Switch" and was still in use in some telephone offices well over 100 years later. To make a call, the user needed to push a button on their phone the required number of times to dial the receivers phone number. This button was replaced in 1896. Interestingly, Philadelphia was the last city to give up the dual service (rotary and button) in 1943. About the same time the transistor was invented, mathematician Dr. Claude Shannon published "A Mathematical Theory of Communication," which promoted the concept of communicating in binary code. Dr. Shannon's paper formed the basis of the entire digital communications revolution, from cell phones to the Internet.

15 years later, in 1963, AT&T used Dr. Shannon’s ideas and created “TouchTone” dialing. This evolution of technology allowed calls to be switched digitally and, later, enabled all manner of automated menus and functionality that eliminated the need for human operators. In 1984 the US government broke apart AT&T – allowing home users to stop leasing their phones from AT&T and allowed them to purchase their own phones. These changes lead to a wave of new designs and functions for the home phone.

In 1968 the Internet was first developed by ARPANET (Advanced Research Projects Agency Network), founded by the U.S. Department of Defense in 1957. ARPANET was developed to provide a decentralized communications network that would not be disrupted by a potential global war. Developed in the 1970s, and in parallel to the Internet, were time-share computer networks owned by large companies who would rent out their large mainframe computers during the evening and weekends when they lay virtually unused. In 1979 CompuServe started a time-share computer service to consumers during these evening downtimes.

As the PC became popular, online service companies (e.g. Prodigy and AOL) formed to provide proprietary information and email services. Subscribers would dial into the network with their telephone lines would pay an hourly fee to receive the services offered

# BACKGROUND INFORMATION

## 3.1 Overview

Trade of self-assertive paired information can be exchanged between peers utilizing the **RTCDataChannel** interface. DataChannels can be utilized for back-channel data, metadata trade, diversion status bundles, document exchanges, or even as an essential channel for information exchange. Utilizing JavaScript‐based APIs that empower equipment control and correspondence through the program, engineers can fabricated a WebRTC application.

## 3.2 The general flow

There are a good vary of situations, starting from single web content demos running on one device to complicated distributed multi-party conferencing with a mix of media relays and archiving services. To begin, we are going to concentrate on the foremost common flow, that covers 2 web browsers exploitation WebRTC to set up a straightforward video call between them. The following is that the outline of this flow:

• Connect peers

• Initiate signals

• Find candidates

• Negotiate media sessions

• Start RTCPeerConnection streams

### 3.2.1 Connect peers

To connect users collectively as a peer, the best way of doing that is to have each them visit same web page. This web page can then discover all browsers and join each of them to a shared signaling server, through the use of some thing like WebSocket. This webpage often, assigns a unique token that may be used to hyperlink the verbal exchange between these two browsers.

You can think of this token as a room or verbal exchange ID. In this task, the primary user visits https://rtconn.tk, and is then create a unique room (if he create the room eul\_comp the URL for the room might be https://rtconn.Tk/eul\_comp). This first consumer then sends this precise URL to the second one user, and once they each have this web page open at the identical time the first step is entire.

### 3.2.2 Initiate signals

Now that both users are in same page, they can now exchange signaling messages to negotiate the setup of their peer connection. In this context, "signaling messages" are simply any form of communication that helps these two browsers establish and control their WebRTC communication as show in Fig 3.1. The WebRTC standards don't define exactly how this has to be completed. This is a benefit, because it leaves this part of the process open for innovation and evolution. It is also a challenge as this uncertainty often confuses developers who are new to RTC communication in general. RTConn application implements this using socket.io. This could, just as easily, be any other approach such as XHR polling, Server-Sent Events (<http://www.html5rocks.com/en/tutorials/eventsource/basics/>),or any combination of these, you feel comfortable with.

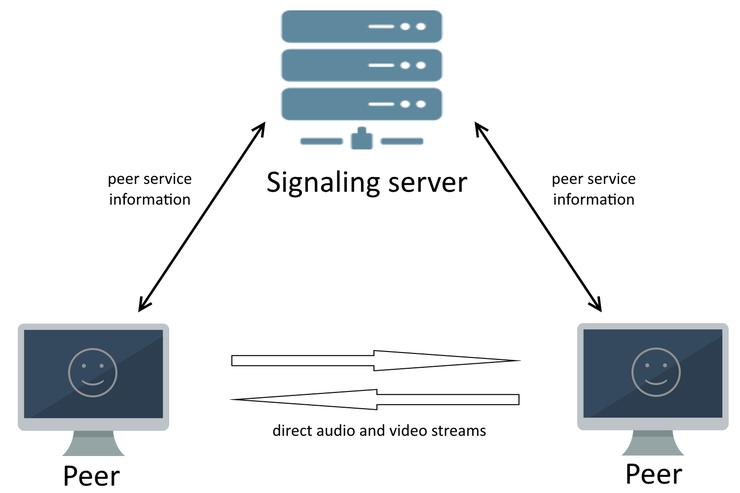


Fig 3.1 RTConn Peer Connection exchanging signaling message

### 3.2.3 Find candidates

The next step is for the two browsers to exchange information about their networks, and how they can be contacted, this process is commonly described as "finding candidates". In this stage a directly accessible network interface and port id established at the end each browser. Each browser is likely to be sitting behind a router that may be using Network Address Translation (NAT) to connect the local network to the internet. Their routers may also impose firewall restrictions that block certain ports and incoming connections. NAT Traversal is critical to establishing a WebRTC communication; NAT is finding a way to connect through these types of routers with firewalls that blocks it from accessing the internet [3]. A common and easy way to achieve this is to use a Session Traversal Utilities for NAT (STUN) server [4], which simply helps to identify how you can be contacted from the public internet and then returns this information in a useful form. There are a range of people that provide public STUN servers. RTConn uses one provided by Google and Numb.

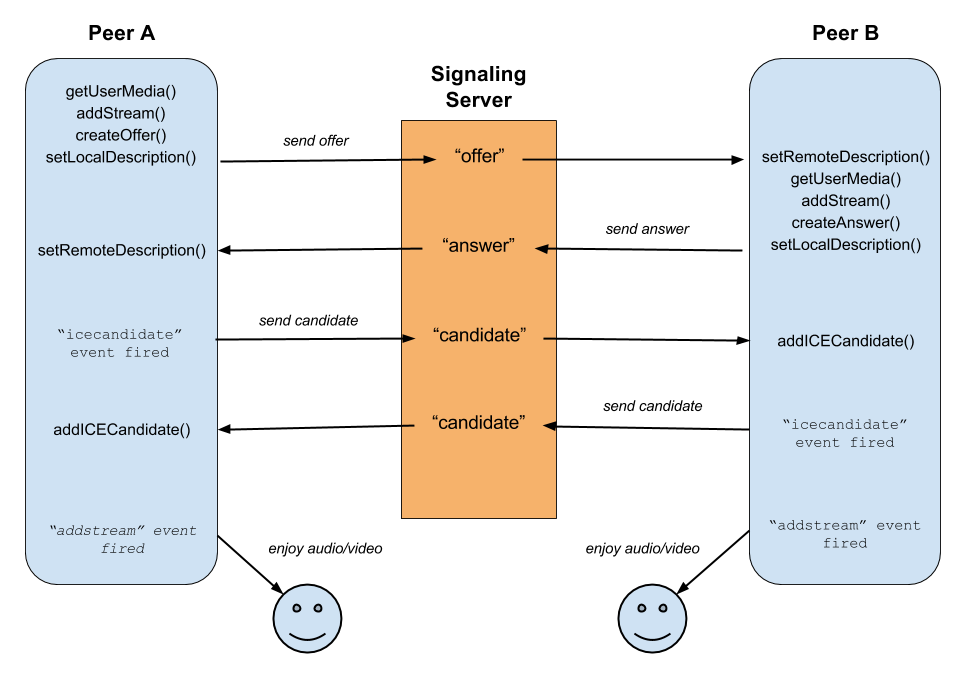
If the STUN server cannot find a way to connect to the browser from the public internet, there is no other option than to fall back to using a solution that relays your media, such as a Traversal Using Relay NAT (TURN) server [5]. This effectively takes you back to a non peer-to-peer architecture, but in some cases, where you are inside a particularly strict private network, this may be your only option. Within WebRTC, this whole process is usually bound into a single Interactive Connectivity Establishment (ICE) framework [6] that handles connecting to a STUN server and then falling back to a TURN server where required.

### 3.2.4 Negotiate media sessions

Now that both the browsers know how to talk to each other as they have gain the public information of each other, they now must decide on the type, format codec, resolution, bitrate, etc. of media that will exchanged. This is usually negotiated using an offer/answer based model, built upon the Session Description Protocol (SDP) [7]. This has been defined as the JavaScript Session Establishment Protocol (JSEP) defined by the IETF [8].

### 3.2.5 Start RTCPeerConnection streams

Once this has all been completed, the browsers can finally start streaming media to each other, either directly through their peer-to-peer connections or via any media relay gateway they have fallen back to using.

Fig 3.2: RTCPeerConnection Ready To Stream Live Data

At this stage show in fig 3.2, the browsers can continue to use the same signaling server solution for sharing communication to control this WebRTC communication. They can also use a specific type of WebRTC data channel to do this directly with each other.

## Signaling

The general idea behind the design of WebRTC has been to fully specify how to control the media plane, while leaving the signaling plane as much as possible to the application layer. The rationale is that different applications may prefer to use different standardized signaling protocols (e.g., SIP or eXtensible Messaging and Presence Protocol [XMPP]) or even something custom. Session description represents the most important information that needs to be exchanged. It specifies the transport (and Interactive Connectivity Establishment [ICE]) information, as well as the media type, format, and all associated media configuration parameters needed to establish the media path. Since the original idea to exchange session description information in the form of Session Description Protocol (SDP) “blobs” presented several shortcomings, some of which turned out to be really hard to address, the IETF is now standardizing the JavaScript Session Establishment Protocol (JSEP). JSEP provides the interface needed by an application to deal with the negotiated local and remote session descriptions (with the negotiation carried out through whatever signaling mechanism might be desired), together with a standardized way of interacting with the ICE state machine. The JSEP approach delegates entirely to the application the responsibility for driving the signaling state machine: the application must call the right APIs at the right times, and convert the session descriptions and related ICE information into the defined messages of its chosen signaling protocol, instead of simply forwarding to the remote side the messages emitted from the browser.

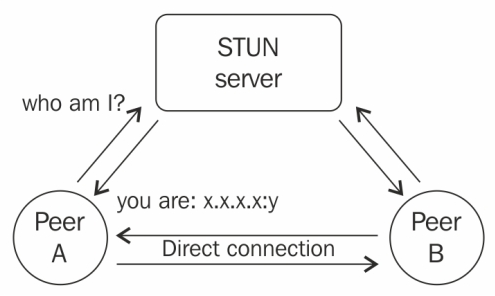
### 3.1 STUN server

Network Address Translation provides a device with an IP address for use within a private local network, but this address cannot be used externally and without a public address there is no way for WebRTC peers to communicate. To get around this problem WebRTC uses Session Traversal Utilities for NAT (STUN) servers, to try and get an external address to a peer. In a simple world, every WebRTC application would be able to learn its external address which it could exchange to other peers in order to communicate directly. In reality most devices exist behind one or more layers of NAT, firewall, proxies and anti-virus software which may block certain addresses, ports and protocols. STUN is a tool that helps protocols dealing with NAT traversal, it may be used by a device to determine the IP address and port allocated to itself by its NAT. It may also be used to check connectivity between two endpoints, and as a keep-alive protocol to maintain NAT bindings.

A STUN server have one simple task, to check the IP and port of an incoming request from application that is running behind a NAT and send that address back as a response. WebRTC applications can use a STUN server to discover the browsers IP and port from a publi

perspective. This enables a peer to get its own publicly accessible address and then pass it on to another peer via a signaling server in order to set up a direct link.

According to Webrtcstats and Bistri statistics that was published in June 2014 1, 92% of all calls successfully made a P2P connection. That means that 8% of the traffic had to be relayed through a TURN server.

Fig 3.3: RTCPeerConnection discovering it IP and Port for a STUN server

### 3.2 TURN Server

Traversal Using Relays around NAT (TURN) servers is the last resort when trying to create a P2P connection with WebRTC. If a host is located behind proxies, firewalls or strict NAT’s and STUN fails to get the public IP of a peer then it is impossible to set up a P2P connection. When this happens, instead of having the connection fail, WebRTC will fall back to use TURN servers to relay data between the two hosts.

If WebRTC needs to use a TURN server to relay the data, the communication will not be P2P but by using it as fallback WebRTC increases the odds to successfully establish connections for a wide variety of devices, as seen in the illustration shown in Figure 3.4

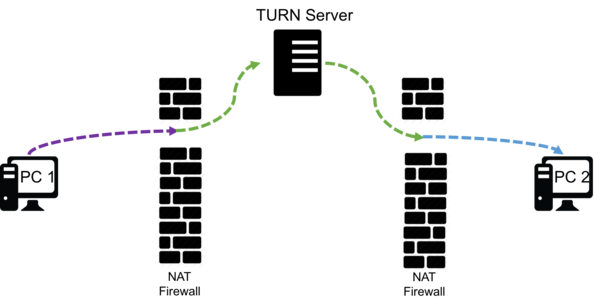


Fig: 3.4: WebRTC Connected Using a TURN Server

TURN servers have a public address so they can be contacted by peers even if the peer is behind a firewall or proxy. They have a theoretically simple task, to relay data between two peers. But unlike STUN servers they will have a huge bandwidth load which results in them having to be sturdier than STUN servers. The TURN servers to be used by the application is specified in the IceConfiguration to the PeerConnection object.

### 3.3 Using WebSockets

The WebSocket API makes it easy for web developers to utilize bidirectional communication within their web applications. You simply create a new connection using the var connection = new WebSocket(url); constructor, and then create your own functions to handle when messages and errors are received. And sending a message is as simple as using the connection API,  **send(message);**  method.

The key benefit here is that the messaging is truly bidirectional, fast, and lightweight.

This means the WebSocket API server can send messages directly to your browser whenever it wants, and you receive them as soon as they happen. There are no delays or constant network traffic as it is in the XHR polling or long-polling model, which makes this ideal for the sort of offer/answer signaling dance that's required to set up WebRTC communication.

The WebSocket API server can then use the unique room or conversation token, previously described, to work out which of the WebSocket API clients messages should be relayed to. In this manner, a single WebSocket API server can support a very large number of clients. And since the network connection setup happens very rarely, and the messages themselves tend to be small, the server resources required are very modest.

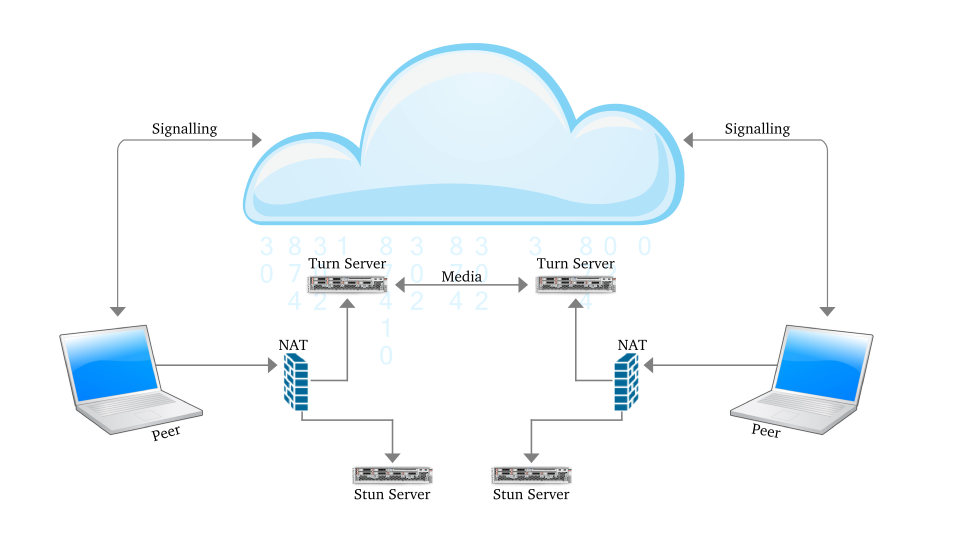
There are WebSocket API libraries available in almost all major programming languages, and since Node.js is based on JavaScript, it has become a popular choice for this type of implementation. Libraries such as socket.io ( http://socket.io/ ) provide a great example of just how easy this approach can really be.

The task of getting the initial signaling data from one peer to another seems like it should be a simple process. Perhaps in a perfect world, a WebRTC signaling mechanism would be able to connect peers directly, without any detours or sidetracking. But the modern internet is structured in such a way that makes this sort of easy relay impossible. NATs of all varieties, and firewalls on many devices, will often erroneously filter packets that are not primed to deal with ALGs and other protective measures. Outside of generating the SDP packet itself, the signaling mechanism is also crucially responsible for ensuring that these signaling messages can be shared between peers in the first place.

So, how does a WebRTC signaling mechanism negotiate the perilous maze of the internet? The answer is simple in theory: it utilizes a versatile framework known as ICE. The efficiency of ICE allows it to calculate, with a mere three methods, the quickest and easiest NAT traversal route for a packet to reach its destination. The first method used, and the least likely to occur, is when ICE tries to make a UDP connection using the host address obtained from a device’s operating system and network card. This will inevitably fail on devices behind NATs, and so there are two remaining methods for ICE to employ: a STUN server or a TURN relay server.

86% of all WebRTC calls are established via STUN servers 1. A STUN server operates STUN servers check the IP address and port of incoming requests, and it then sends that address back to the device’s WebRTC application as a response. The WebRTC application thus uses a STUN server to ascertain its own IP port address from a public perspective. This allows the application to offer a publicly accessible address, which is then passed to another WebRTC-enabled peer via the signaling mechanism.

If both methods fail, the final method employed by ICE is a TURN relay server. TURN servers are used to stream audio, video, and other real-time data between peers. Technically speaking, it does not relay signaling information, because it enables actual real-time data exchanges between peers. TURN servers have publicly available addresses, so peers can connect to them even if they are behind NATs and firewalls. TURN servers are costlier to maintain than STUN servers, because they are actually streaming media rather than connecting peers.

Fig 3.5: Standard WebRTC architecture

A fully functioning WebRTC application (Fig 3.5) requires all of ICE’s capabilities to operate smoothly and effectively. But purchasing and maintaining numerous servers at a significant cost is simply not a feasible option for developers who are looking to make sound economic and personnel decisions.

## 3.4 WebRTC APIs

### 3.4.1 GetUserMedia

The MediaDevices getUserMedia() method prompts the user for permission to use a media input which produces a MediaStream with tracks containing the requested types of media. That stream can include, for example, a video track (produced by either a hardware or virtual video source such as a camera, video recording device, screen sharing service, and so forth), an audio track (similarly, produced by a physical or virtual audio source like a microphone, A/D converter, or the like), and possibly other track types [10].

### 3.4.2 MediaStream

A MediaStream is an abstract representation of an actual stream of data of audio and/or video. It serves as a handle for managing actions on the media stream, such as displaying the stream’s content, recording it, or sending it to a remote peer. A MediaStream may be extended to represent a stream that either comes from (remote stream) or is sent to (local stream) a remote node. A LocalMediaStream represents a media stream from a local media-capture device (e.g., webcam, microphone, etc.). To create and use a local stream, the web application must request access from the user through the getUserMedia() function. The application specifies the type of media—audio or video—to which it requires access. The devices selector in the browser interface serves as the mechanism for granting or denying access. Once the application is done, it may revoke its own access by calling the stop() function on the LocalMediaStream. Media-plane signaling is carried out of band between the peers; the Secure Real-time Transport Protocol (SRTP) is used to carry the media data together with the RTP Control Protocol (RTCP) information used to monitor transmission **Error! Bookmark not defined.**management.

### 3.4.3 RTCPeerConnection

RTCPeerConnection (simply called PeerConnection) is a component that handles multimedia communication between two peers, making sure the streams are stable and efficient. It is an API that contains functions for encryption and bandwidth management. When setting up a connection, both peers need to initialize their PeerConnection object. The configuration parameter is optional and it contains information to find the servers used by Interactive Connectivity Establishment (ICE). There may be multiple servers of each type2 and if no configuration is set, the default settings will be used [2].

### 3.4.4 DataChannel

RTCDataChannel (simply DataChannel) enables exchange of arbitrary data between two peers with customizable delivery properties of the underlying transport. It is a bi-directional channel and resides as a component of the PeerConnection API [2]. Each application using the channel can configure it to provide the following:

* Reliable or partially reliable delivery of sent messages
* In-order or out-of-order delivery of sent messages

Table 3.1: Comparison of TCP vs UDP vs SCTP

|  |  |  |  |
| --- | --- | --- | --- |
|  | TCP | UDP | SCTP |
| Reliability | Reliable | Unreliable | Configurable |
| Delivery | Ordered | Unordered | Configurable |
| Transmission | Byte-Oriented | Message-Oriented | Message-Oriented |
| Flow Control | Yes | No | Yes |
| Congestion Control | Yes | No | Yes |

The standardization work within the IETF has reached a general consensus on the usage of the Stream Control Transmission Protocol (SCTP) encapsulated in DTLS to handle non media data types. The encapsulation of SCTP over DTLS over UDP together with ICE provides a NAT traversal solution, as well as confidentiality, source authentication, and integrity protected transfers. Moreover, this solution allows the data transport to interwork smoothly with the parallel media transports, and both can potentially also share a single transport-layer port number. SCTP has been chosen since it natively supports multiple streams with either reliable or partially reliable delivery modes. It provides the possibility of opening several independent streams within an SCTP association towards a peering SCTP endpoint. Each stream actually represents a unidirectional logical channel providing the notion of in sequence delivery. A message sequence can be sent either ordered or unordered. The message delivery order is preserved only for all ordered messages sent on the same stream. However, the DataChannel API has been designed to be bidirectional, which means that each DataChannel is composed as a bundle of an incoming and an outgoing SCTP stream. The DataChannel setup is carried out (i.e., the SCTP association is created) when the CreateDataChannel() function is called for the first time on an instantiated PeerConnection object. Each subsequent call to the CreateDataChannel() function just creates a new DataChannel within the existing SCTP association

# IMPLEMEMTATION DETAILS

WebRTC lets in you to initialize peer-to-peer connections with different browsers speedy and without problems. Build such an software from scratch, a wealth of frameworks and libraries coping with regular troubles like data loss, connection dropping, and NAT traversal, will have to be taken care of manually. With WebRTC, all of this comes built-in into the browser out-of-the-box. This era of WebRTC, no plugins or third-celebration software program is required to develop a real-time system. It is open-sourced and its source code is freely available at http://www.webrtc.org. The WebRTC consists of media caption, encoding and decoding of audio and video, transportation layer, and consultation management APIs.

## Setup page

This page get the user ready to join a call session. We will start by capturing the Media Stream and giving the user the opportunity to select their desire camera for video stream and microphone for audio stream, and an input area to input his/her name. Figure 4.1 shows the necessary HTML for the page.

### Step 1: Adding the HTML

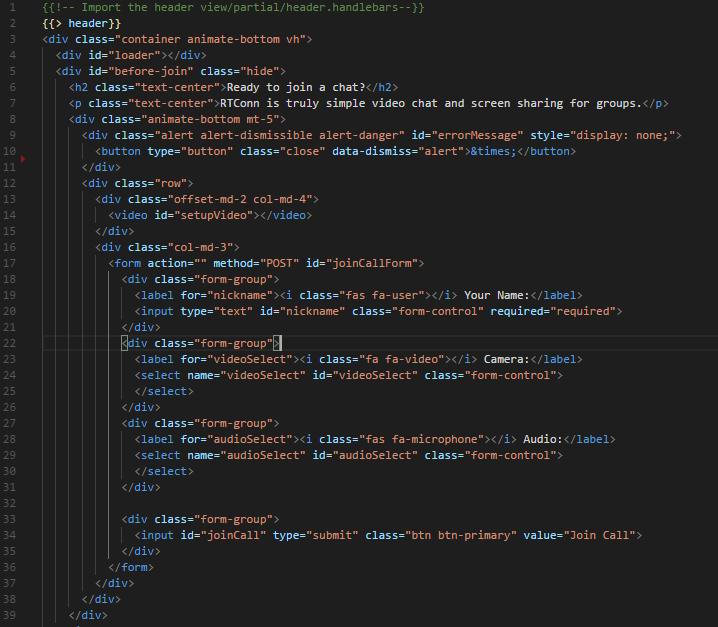


Fig 4.1: Setup Page HTML (code)

### Step 2: Prefix getUserMedia API to work on all browser

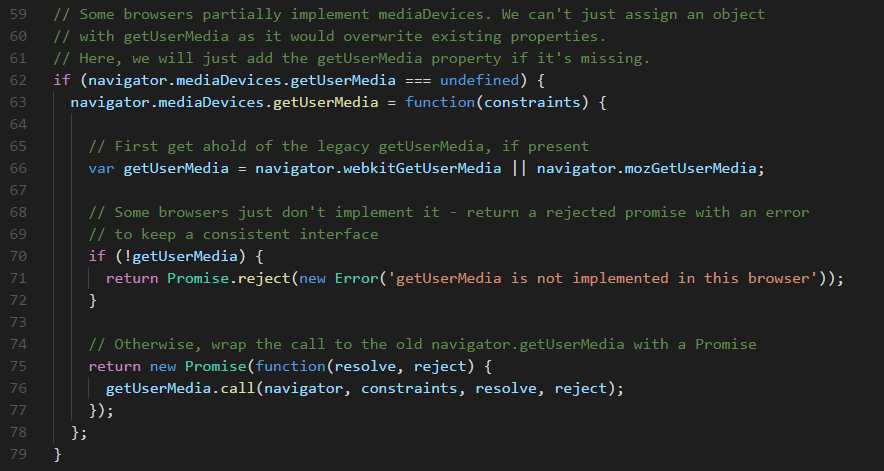


Fig 4.2: Make getUserMedia work same on all browser (code)

The code above (Fig 4.2) ensure the navigator.mediaDevices.getUserMedia have same API interface regardless of the browser the user is using. This is needed because all browser do not implement this API. First we check to see if it is define, when it is not define the we set it using the legacy getUserMediam if present. We then return a Promise object, on Error or Success.

### Step 3: Obtain user MediaDevices

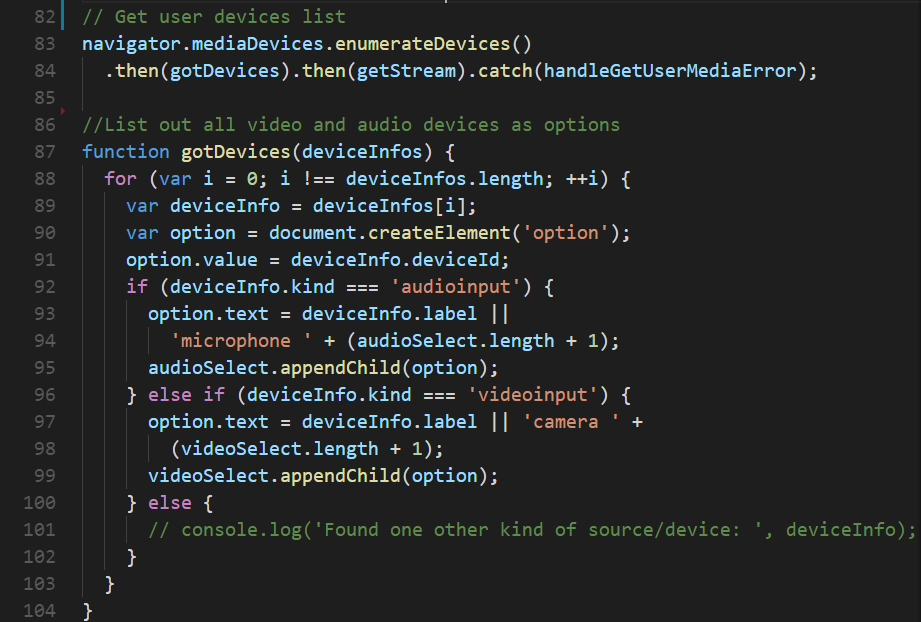


Figure 4.3: Get user media devices and append to select option (code)

The function above (Fig 4.3) Obtains the list of all media devices, if media type is audio it adds it to the audio option and if it is video it is added to the video option. The value of the option is set to the device ID and is used in the ***getVideoConstrains*** function.

### Step 4: Obtain user media stream

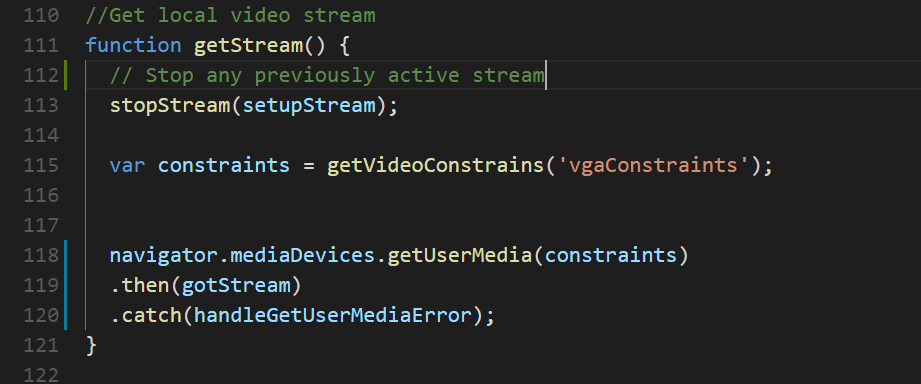


Fig 4.4: Obtain User webcam and audio function (code)

The ***getStream*** function (Fig 4.4) fires every time the select video or audio select box on the setup page changes. First any previously selected stream is stopped, and then a new constrain for the stream is obtained. Once the stream is stop and the new constrain is obtained we the call the ***getUserMedia*** function passing the new constrain, this function may requires user permission Fig 4.5 shows this in a Chrome browser.

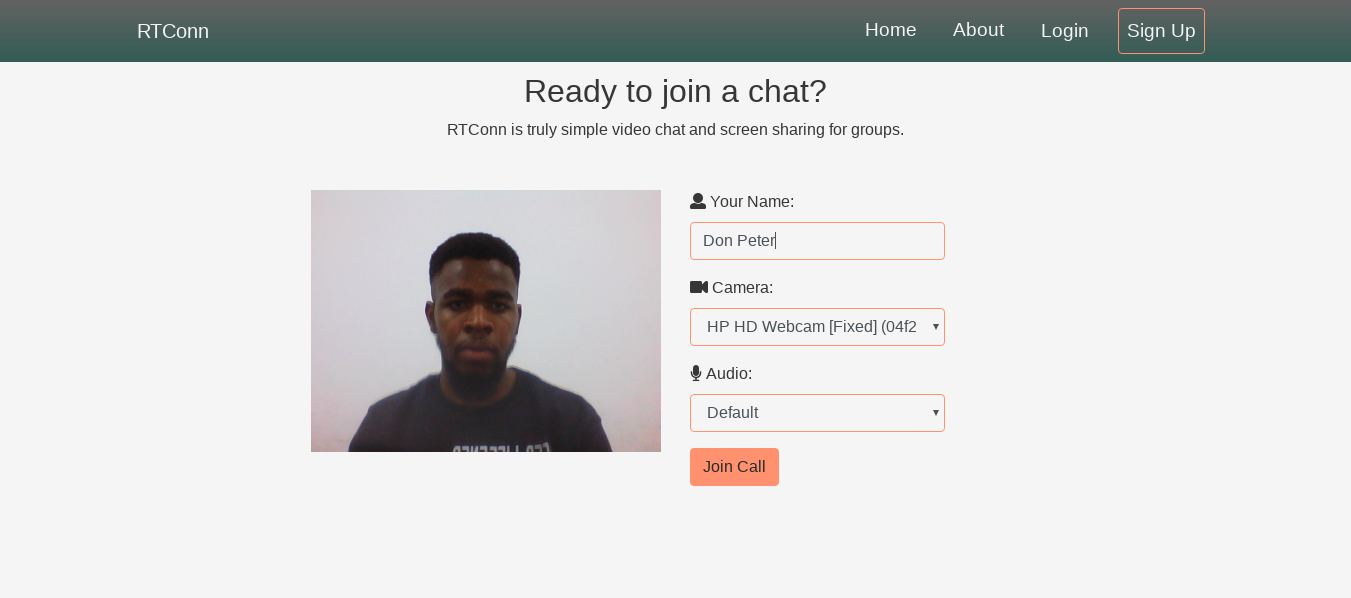


Fig 4.5: getUserMedia Permission box (demo).

## Connect Peers

### Join socket.io rooms

The very first step to take is to join the chat socket.io room through which all signaling message will be exchanged among peers. The below listed rooms will be joined

* Chat Room *(URL-chat-room)*
* File Room  *(URL-file-room)*
* Signal Room  *(URL-signal-room)*

The name of all the rooms is prefixed with a namespace that is the URL path. So if the URL is <https://www.rtconn.tk/eul_comp>, then the namespace will be eul\_comp; thereby setting the rooms to be joined as eul\_comp-chat-room, eul\_comp-file-room and eul\_comp-signal-room. The code in fig 4.6 sets the value for the rooms apporiately as descriped.

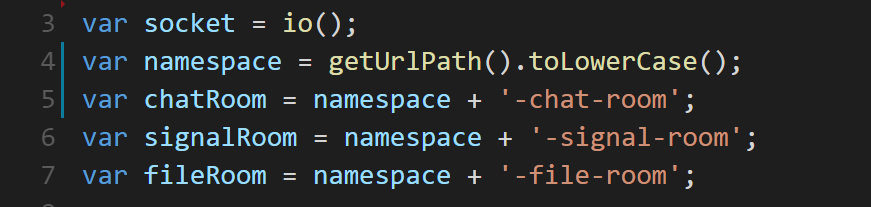


Fig 4.6: Setting Room Name and Initiating Socket.io (code)

With the value of the rooms set, the function ***startCall*** (see in fig 4.7) is called. The very first thing the function does is to join the socket.io rooms and then obtain a new Media Stream which will be referred to as localStream (this object is available global as it is attached to the window). It then emit the very first signal message of type “user\_here”, this message initiate the call, by checking if there is a user already in the room.

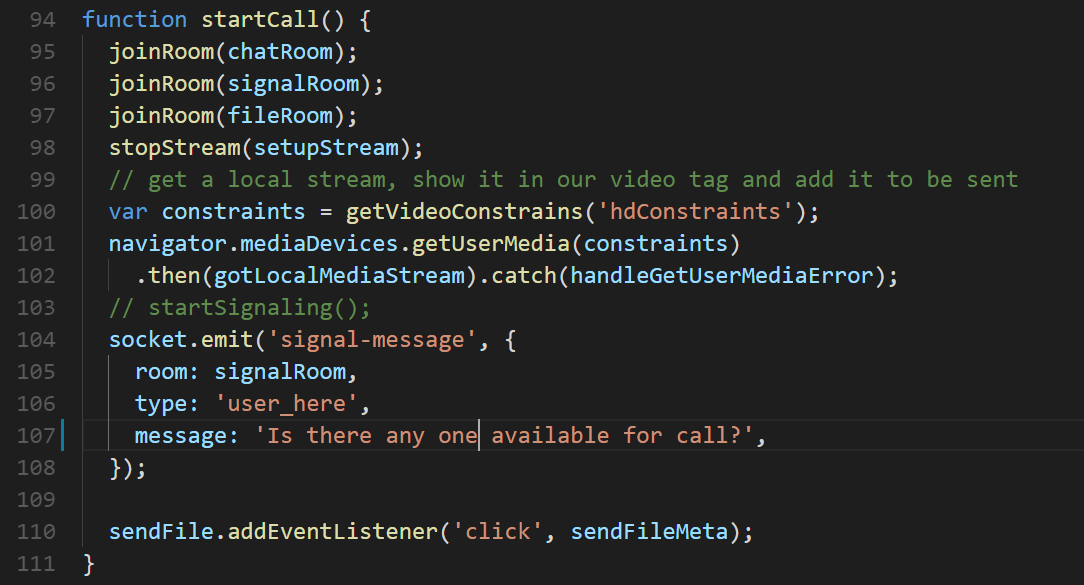


Fig 4.7: Starting the calling Process (Code)

### Initiate peer connection and dataChannel

Once we have successfully join the signaling room, we then initialize the WebRTC connection object by calling the new ***RTCPeerConnection*** construct. The PeerConnection object is then used to create a DataChannel. Now an EventHadler is set for the following event on the PeerConnection object ***onicecandidate***, ***onnegotiationneeded*** and ***ontrack.*** The code for this is shown in Fig 4.8.

Onicecandidate is an event handler that fires every time the ICE agent wishes to deliver a message to the peer via the signaling server. Thiereby letting the ICE agent perform negotiation with the remote peer, and without the browser itself desiring to realize any specifics about the technology used for signaling; implement this method to use whatever messaging technology you choose to send the ICE candidate to the remote peer[11].

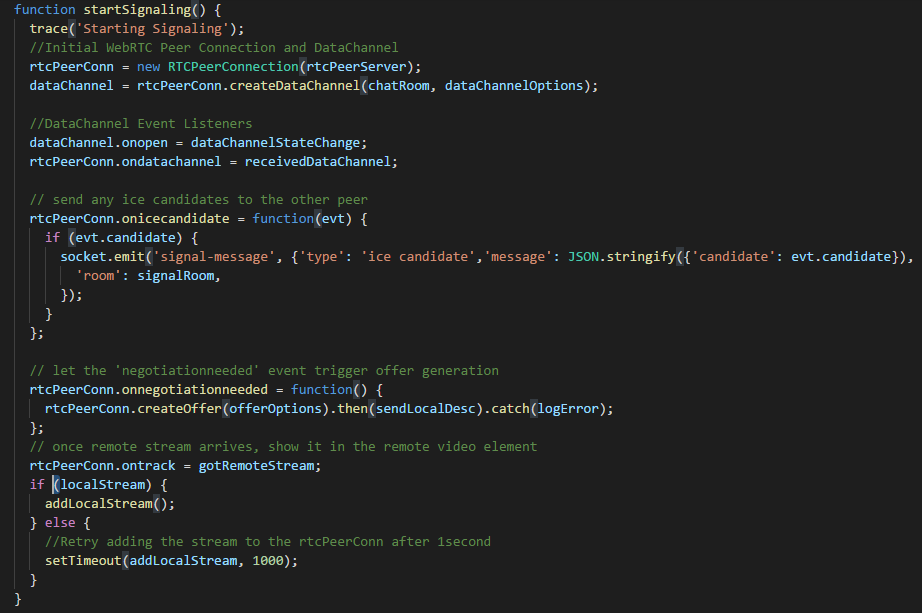


Fig 4.8: Initializing the PeerConnection and DataChannel (code)

The onnegotiationneeded property is an event handler that specifies the function that is called to handle the negotiationneeded event once it happens on the RTCPeerConnection instance. This event is fired when a change has occurred which requires session negotiation.This event is emitted once a modification which needs session negotiation occurrs. This negotiation should be carried out by the caller, because some session changes cannot be negotiated as the answerer. The negotiationneeded event is commonly fired after a send track is brought to the RTCPeerConnection. If the session is changed in a manner that requires negotiation even as a negotiation is already in progress, no negotiationneeded event will fire until negotiation completes, and then if negotiation is still needed [12].

### Handling signal messages

In other to initiate a peer-to-peer connection browser will have to pass several signal message to one another, and this messages have to be processed by the frontend (Fig 4.9) and backend.



Fig 4.9: Handling Signal Message (code)

Upon receiving a signal message we first check if the PeerConnection object has been defined, if it hasn’t the startSignaling function is called. The startSignaling method setup the PeerConnection object.

On the server side once a signal message is receive it send it to all other users on the signal-room expect from the sender. Fig 4.10 is the code for the server on signal-message handler.

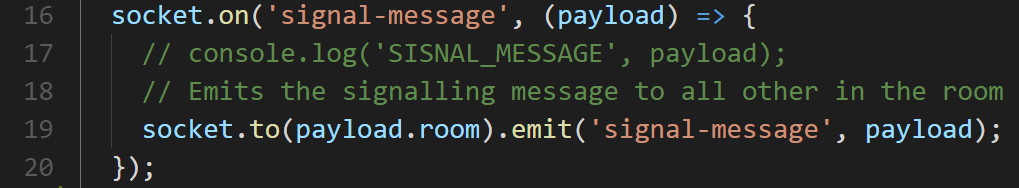


Fig 4.10: Server On Signal Message Handler (code)

As show from the **socker.on**(‘signal-message’) handler in Fig 4.10 all signal message is directly emitted to all other clients without modifying or transforming it.

### Attach stream to video element source

Once we obtain user media stream next we have to attach it to our local video stream. Figure 4.6 show the code for this and figure 4.11 show the page after it has been attached.

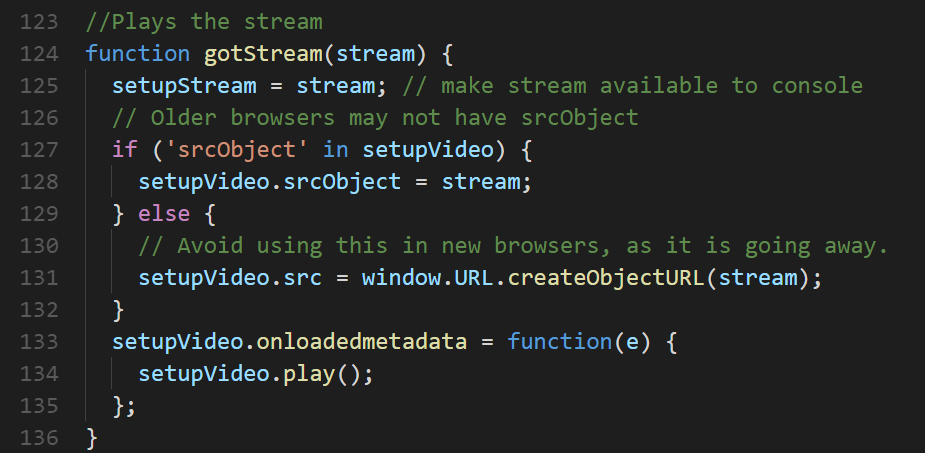


Fig 4.11: Attach the stream to the setup video src (code)

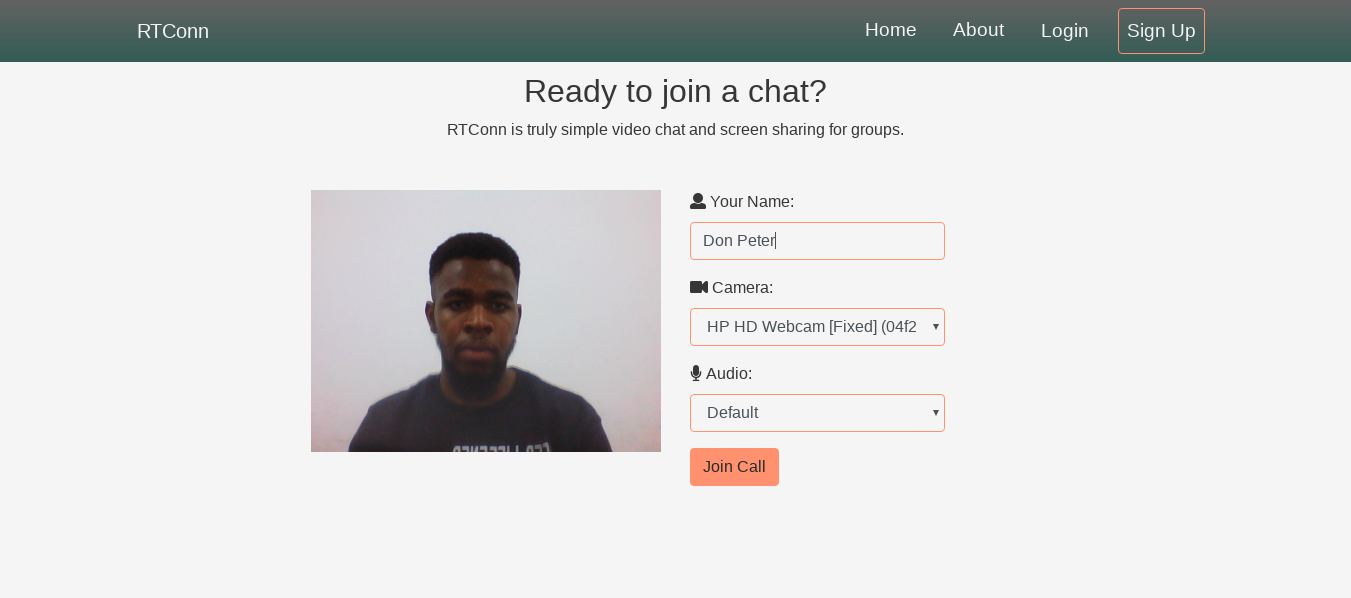


Fig 4.12: Setup page after user video and audio stream has been attach (demo)

After the user desire audio and video is ready and fully setup, the user is now ready to join the call. Once the user click on the “Join Call” button (see Fig 4.12), then the value nickname, selected video and audio will be passed to the controller “main.js” and will be used to initiate the call.

## Add video and audio media streams

Now that the peers are connected and both browser now know how to communicate with each other directly or via a relay (if TURN server is been used). Each browser will need to add his local media stream; which will be used by the other peer as the remote steam and then attach it to the ***remoteVideo*.*src*** (code shown in Fig 4.13)***.***

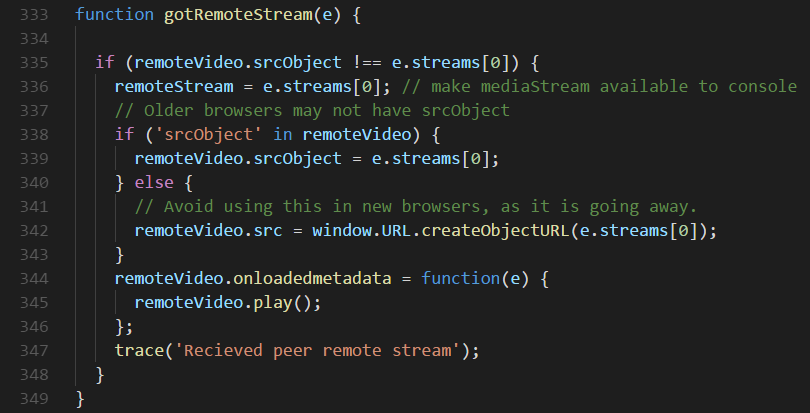


Fig 4.13: Adding Remote Media Stream to the Remote Video Stream (code)

After successfully attaching the remote media stream to the remote video, we have fully setup a live video chat between both peers (as shown in Fig 4.14). A real time video stream from one end to the other directly peer-to-peer, with no work load on the server. The browser does the encoding and decoding of the video streams.



Fig 4.14: Peer-to-Peer video chat demo

## Add instant text messaging using dataChannel

Adding instant text messaging functionality to the system is half way setup at this stage. As shown in *Fig4.10* when we initialize the RTCPeerConnection object, we also created a DataChannel through which we will forward all chat message and file transfer via the RTCPeerConnection.

Firstly, we need to create the necessary HTML element in other to be able to send message from one peer to the other peer the code in Fig 4.15 show the HTML code used in the project.

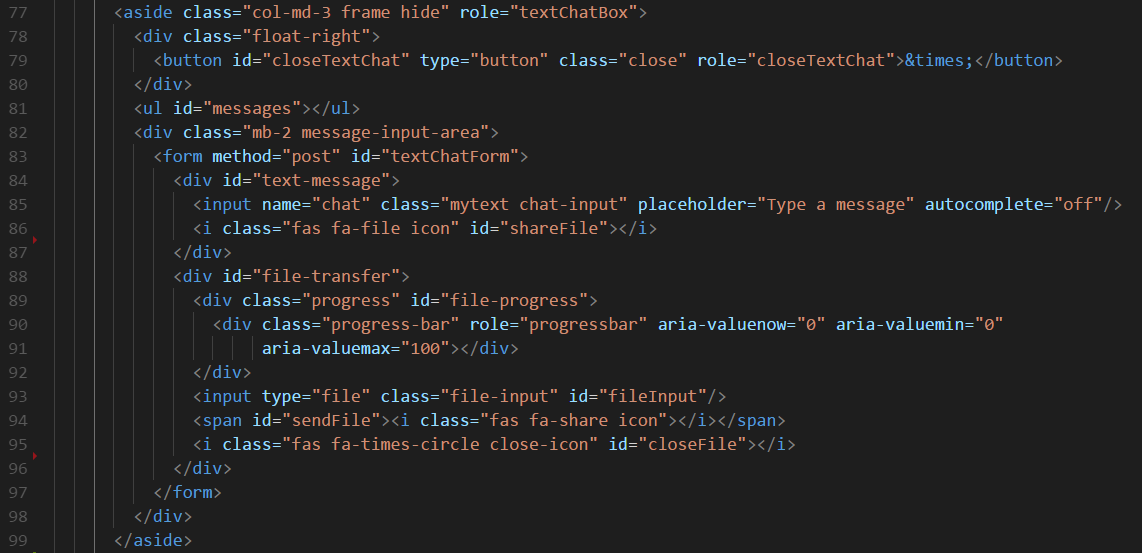


Fig 4.15: Chat and File transfer HTML (code)

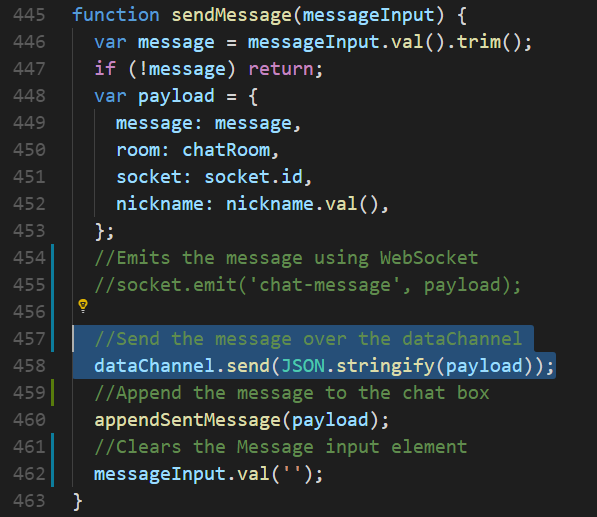


Fig 4.16: Sending instant text message with the dataChannel (code)

The send() method of the RTCDataChannel interface (shown in Fig 4.16) sends data across the data channel to the remote peer. This can be done any time except during the initial process of creating the underlying transport channel. Data sent before connecting is buffered if possible (or an error occurs if it's not possible), and is also buffered if sent while the connection is closing or closed [14].

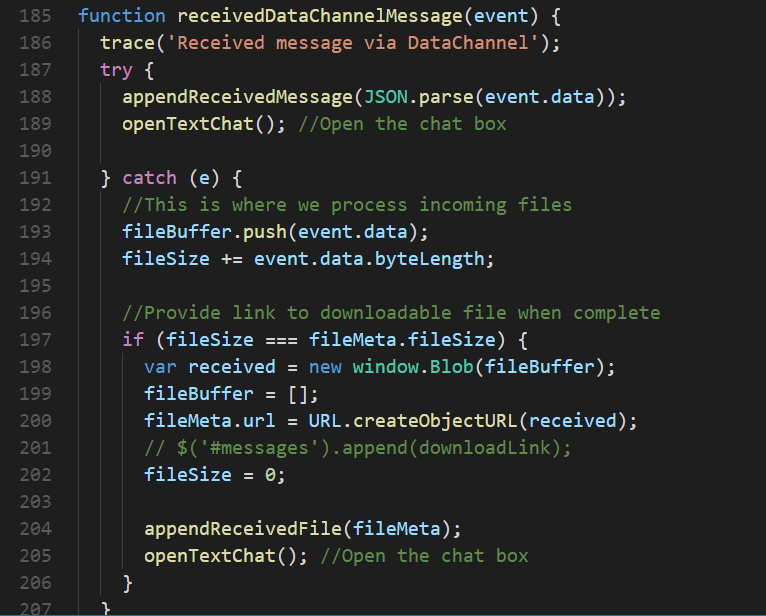


Fig 4.17: Handling dataChannel incoming messages and files (code)

The RTCDataChannel.onmessage property is an Event Handler which specifies a function to be called when the message event is fired on the channel. This event is represented by the MessageEvent interface. This event is sent to the channel when a message is received from the other peer [15]. Fig 4.17 shows the function definition for the onmessage event handler, usually different channel is used for chat message and file transfer, but this project uses on channel for both. The try section of the function handle chat messages, but if an error is thrown then it must a file transfer message.

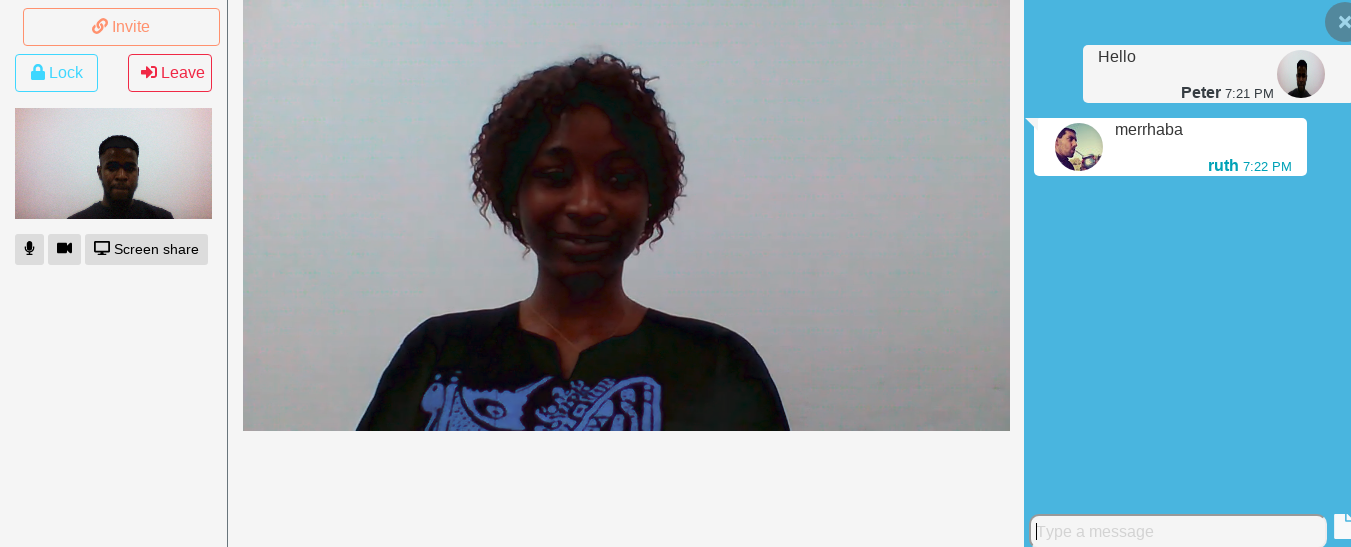


Fig 4.18: Chat messaging (demo)

## Transferring Files

WebRTC Data (aka DataChannels / RTCDataChannel) API is enables developers transmit arbitrary information at once among two users (P2P) in ultra-low latency (Shown in Fig four.19). This has not really been possible, up until lately, and it’s a tech changer. Why? Because in the following couple of years it will likely be an important building block for constructing web programs.

People are obviously talking about applications like video-conference and file-sharing. Traditionally, file-sharing and video chat were built the use of P2P technologies that trusted proprietary clients (BitTorrent, Skype). Due to several boundaries, it was the quality (or the most effective) desire for the mainstream. But as soon as WebRTC matures, the natural, lengthy-awaited evolution (now not revolution in my opinion) will be successful and prevail over traditional technologies.

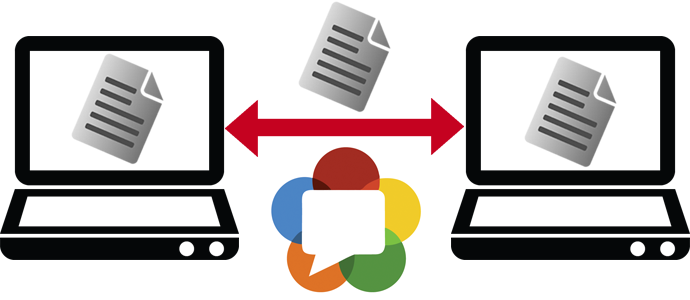


Fig 4.19: WebRTC File Transfer

Being on the web brings so many powerful advantages. The web is standard, more updatable, accessible, searchable, embeddable and is just plain awesome. To accomplish this in an interoperable way, the file is split into chunks (Fig)which are then transferred via the dataChannel. The dataChannel is reliable and ordered by default which is well-suited to file transfers.

At the receiver, the file is reassembled using the Blob API and made available for download. RTConn uses websocket to send metadata about the file (such as the filename, type, size, last modification date, hash, ...).

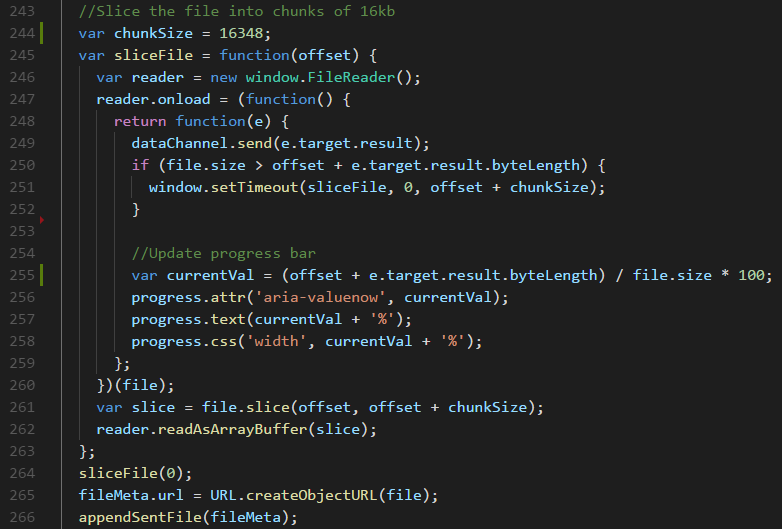


Fig 4.20: Slicing and sending files via dataChannel (code)

## Screen sharing

After voice, video, instant messaging, presence and sending files; the only missing piece for collaboration would be screen sharing. Both Chrome and Firefox currently have built-in support for screen capturing in the desktop versions. Screen sharing is not supported on mobile, as at the time of writing this (mobile can still receive screen shares). In both chr, screen capturing is achieved through the MediaDevices.getUserMedia() (gUM) interface. gUM can be called once to get an user audio / video stream, and a second time to get a screen stream. The details are sadly not common between browsers and seem poorly documented. Note that we are also using adapter.js.

### Chrome

Chrome has built in support for screen capture, however to gain permission to use this functionality, an application must use a Chrome Extension. (Unless of course you are Google and kindly white-list yourself, as they have done for Hangouts) The extension uses chrome.desktopCapture.chooseDesktopMedia() to return a sourceID. The sourceID is then used as part of the gUM constraints.

Chrome Extension architecture from https://developer.chrome.com/extensions/overview

Google provides extensive documentation for extension development, which I won’t repeat all of, nor claim to be an expert on. However in short, a simple extension for screen sharing will have a content script which runs in the context of your web page and a background script running in a seperate extension context. The content script can communicate with your web app by sending messages to window or via DOM manipulation, whereas the background script can not. The background script can access all Chrome extension api’s, but the content script can not. The content script and background script can communicate with each other via chrome.runtime.connect().

## Conferencing

In a WebRTC conferencing scenario (or N-way call), each browser has to receive and handle the media streams generated by the other N-1 browsers, as well as deliver its own generated media streams to N-1 browsers (i.e., the application-level topology is a mesh network). While this is a quite straightforward scenario, it is nonetheless difficult to manage for a browser and at the same time calls for linearly increasing network bandwidth availability.

For these reasons, video conferencing systems usually rely upon a star topology where each peer connects to a dedicated server that is simultaneously responsible for:

* Negotiating parameters with every other peer in the network
* Controlling conferencing resources
* Aggregating (or mixing) the individual streams
* Distributing the proper mixed stream to each and every peer participating in the conference

Delivering a single stream clearly reduces both the amount of bandwidth and amount of CPU (and possibly GPU [Graphics Processing Unit]) resources required by each peer involved in a conference. The dedicated server can be either one of the peers or a server specifically optimized for processing and distributing real-time data. In the latter case, the server is usually identified as a Multipoint Control Unit (MCU).

The WebRTC API does not provide any particular mechanism to assist the conferencing scenario. The criteria and process to identify the MCU are delegated to the application. However, this is a big engineering challenge because it envisages the introduction of a centralized infrastructure in the WebRTC peer-to-peer communication model. The upside of such a challenge clearly resides in the consideration that being capable of establishing a peer connection with a proxy server adds to the benefits offered by WebRTC through the additional services offered by the proxy server itself.

# 5. CONCLUSION AND FUTURE RECOMMENDATION

## 5.1 Benefits

* ***Utilise every minute:*** *There's no need to waste money travelling up and down the country for a meeting when you have iMeet. iMeet brings you video chat and screen share 24/7 so you can hold meetings face-to-face without any hassle.*
* ***Wherever, whenever:*** *With iMeet you can work wherever you like, whether you're in the office, or on-the-go, it works wherever you are whilst still delivering the same amazing experience. Best yet you can even swap between devices seamlessly; computer, smartphone or tablet, the choice is yours.*
* ***Take control:*** *As host, you're in control of who is on the main stage. Is it your department's turn to speak? Then put your cube front and centre and bring co-presenters on when you are ready. You can add multiple people to the main stage as you move through your meeting.*
* ***Take notes:*** *iMeet is the only online meeting tool that allows you to take notes in real time during the meeting. You can even link your notes to Evernote, allowing you to easily share your notes with other participants.*
* ***Improve Mobile Experience:*** *WebRTC-enabled technologies allow Mobile Network Operators to create a more compelling customer experience on mobile devices.*
* ***Humanize Your Conversation:*** *Take this point in contrast with voice communication or email correspondence. Video is moving pictures, which are worth more than a million words. By showing yourself and seeing others, you can work the charms of body language, which is so important in business and other activities involving human interaction. Also, seeing someone while talking to them completely changes the nomenclature of a conversation, be it for business or in a personal relationship.*
* ***Learn and Teach Online:*** *There are great courses being offered and great teachers teaching, but most of them may be far, very far from you. If you are a teacher or trainer, your market may be lying far from where you are. This app is a great way of acquiring and sharing knowledge beyond hurdles. While it will not be like being physically present, the interaction is adequate. You will be able to use multimedia facilities like online interactive whiteboards, and you can use online collaboration tools. Courses can take place with the teacher and each different student being in a different location, and that would constitute a real video conferencing session. Alternately, the teacher would be in one location and the class in another, with all students looking at one same screen.*

*Some time ago, video calling or conferencing was a luxury and required expensive and (then) complex equipment and expertise. Today, you literally carry it in your pocket. You can participate in and host video conferencing sessions on your smartphone and mobile device as well as on your computer with basic hardware and adequate Internet connectivity.*

## 5.2 Future works

The ideal that lead to this project , was one that I had two year ago during a church service, where the preacher talked about checking out an event that was announced on Facebook, and seeing so many religious activities and post on my Facebook timeline. It felt a bit off as Facebook didn’t seem to be a fitting community for such. Then I realized that if a new community can be built for the church, with the right requirement, a kind of social network with religious body as it niche.

Two year ago, I have no knowledge on how to build a server/client web application, but today I have a very good understanding and knowledge of the required component required to build the social network I plan on building in the future.

Completing of this project means I have nearly (around 90%) completed the list of required knowledge I set out two years ago. I plan on making the application as integrateable as possible, so that I and other developers out there trying to build an application that connect people together in real time, will have no difficulty in doing so. Therefore this project will continuously be extended and improve, to work as a black box to empowering real-time system.

# 6. REFERENCES

*Books:*

[1] Real-Time Communication with WebRTC, Salvatore Loreto and Simon Pietro Romano, 2014 pp. 15–25.

[2] Peer-to-peer communication in web browsers using WebRTC, 2015, pp. 15–19.

Webs:

[3] <http://en.wikipedia.org/wiki/NAT_traversal>

[4] <http://en.wikipedia.org/wiki/Session_Traversal_Utilities_for_NAT>

[5] <http://en.wikipedia.org/wiki/Traversal_Using_Relay_NAT>

[6] <http://en.wikipedia.org/wiki/Interactive_Connectivity_Establishment>

[7] <http://en.wikipedia.org/wiki/Session_Description_Protocol>

[8] <http://tools.ietf.org/html/draft-ietf-rtcweb-jsep-00>

[9] <https://www.onsip.com/blog/webrtc-signaling>

[10] <https://developer.mozilla.org/en-US/docs/Web/API/MediaDevices/getUserMedia>

[11] <https://developer.mozilla.org/en-US/docs/Web/API/RTCPeerConnection/onicecandidate>

[12] <https://developer.mozilla.org/en-US/docs/Web/API/RTCPeerConnection/onnegotiationneeded>

[13] <https://developer.mozilla.org/en-US/docs/Web/API/RTCPeerConnection/ontrack>

[14] <https://developer.mozilla.org/en-US/docs/Web/API/RTCDataChannel/send>

[15] <https://developer.mozilla.org/en-US/docs/Web/API/RTCDataChannel/onmessage>

[16] Chris Ward, Implementing WebRTC Screen Sharing in a web app, late 2016, <https://medium.com/@chris_82106/implementing-webrtc-screen-sharing-in-a-web-app-late-2016-51c1a2642e4>

# 7. Appendix A. WebRTC 1.0 APIs

This Appendix provides a summary of the W3C WebRTC APIs.

RTCPeerConnection API

Table A-1. RTCConfiguration dictionary members

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Type | Default | Description |
| iceServers | sequence<RTCIceServer> |  | An array containing URIs of servers available to be used by ICE, such as STUN and TURN servers. |
| iceTransports | RTCIceTransports | “all” | Indicates which candidates the ICE engine is allowed to use. |
| requestIdentity | RTCIdentityOption | “ifconfigured” | See the requestIdentity member of the RTCOfferAnswerOptions dictionary. |

**Table A-2. RTCIceServer dictionary members**

| Name | Type | Description |
| --- | --- | --- |
| credential | DOMString | If this RTCIceServer object represents a TURN server, then this attribute specifies the credentials to use with that TURN server. |
| urls | (DOMString or sequence<DOMString>) | STUN or TURN URI(s) as defined in [STUN-URI] and [TURN-URI] or other URI types. |
| username | DOMString | If this RTCIceServer object represents a TURN server, then this attribute specifies the username to use with that TURN server. |

**Table A-3. RTCIceTransports enumeration values**

| Value | Description |
| --- | --- |
| none | The ICE engine must not send or receive any packets at this point. |
| relay | The ICE engine must only use media relay candidates such as candidates passing through a TURN server. This can be used to reduce leakage of IP addresses in certain use cases. |
| all | The ICE engine may use any type of candidates when this value is specified. |

**Table A-4. RTCPeerConnection attributes**

| Access property | Type | Name |
| --- | --- | --- |
| readonly | RTCSessionDescription | remoteDescription |
| readonly | RTCSignalingState | signalingState |
| readonly | RTCIceGatheringState | iceGatheringState |
| readonly | RTCIceConnectionState | iceConnectionState |
|  | EventHandler | onnegotiationneeded |
|  | EventHandler | onicecandidate |
|  | EventHandler | onsignalingstatechange |
|  | EventHandler | onaddstream |
|  | EventHandler | onremovestream |
|  | EventHandler | oniceconnectionstatechange |

## State Definition

**Table A-5. RTCSignalingState**

| Value | Description |
| --- | --- |
| stable | There is no offer/answer exchange in progress. This is also the initial state in which case the local and remote descriptions are empty. |
| have-local-offer | A local description, of type “offer,” has been successfully applied. |
| have-remote-offer | A remote description, of type “offer,” has been successfully applied. |
| have-local-pranswer | A remote description of type “offer” has been successfully applied and a local description of type “pranswer” has been successfully applied. |
| have-remote-pranswer | A local description of type “offer” has been successfully applied and a remote description of type “pranswer” has been successfully applied. |
| closed | The connection is closed. |

**Table A-6. RTCIceGatheringState**

| Value | Description |
| --- | --- |
| new | The object was just created, and no networking has occurred yet. |
| gathering | The ICE engine is in the process of gathering candidates for this RTCPeerConnection. |
| complete | The ICE engine has completed gathering. Events such as adding a new interface or a new TURN server will cause the state to go back to gathering. |

**Table A-7. RTCIceConnectionState**

| Value | Description |
| --- | --- |
| new | The ICE Agent is gathering addresses and/or waiting for remote candidates to be supplied. |
| checking | The ICE Agent has received remote candidates on at least one component, and is checking candidate pairs but has not yet found a connection. In addition to checking, it may also still be gathering. |
| connected | The ICE Agent has found a usable connection for all components but is still checking other candidate pairs to see if there is a better connection. It may also still be gathering. |
| completed | The ICE Agent has finished gathering and checking and found a connection for all components. |
| failed | The ICE Agent is finished checking all candidate pairs and failed to find a connection for at least one component. Connections may have been found for some components. |
| disconnected | Liveness checks have failed for one or more components. This is more aggressive than failed, and may trigger intermittently (and resolve itself without action) on a flaky network. |
| closed | The ICE Agent has shut down and is no longer responding to STUN requests. |

## Peer-to-Peer Data API

The Peer-to-Peer Data API lets a web application send and receive generic application data peer-to-peer. The API for sending and receiving data models the behavior of WebSockets.

RTCDataChannel

createDataChannel(...)

### Interface RTCDataChannel Interface Methods

**Table A-8. RTCDataChannel Methods**

| Return type | Name |
| --- | --- |
| void | close() |
| void | send(DOMString data) |
| void | send(Blob data) |
| void | send(ArrayBuffer data) |
| void | send(ArrayBufferView data) |

### RTCDataChannel Interface Attributes

**Table A-9. Attributes**

| Access property | Type | Name |
| --- | --- | --- |
| readonly | DOMString | label |
| readonly | boolean | ordered |
| readonly | unsigned? | maxRetransmitTime |
| readonly | unsigned? | maxRetransmits |
| readonly | DOMString | protocol |
| readonly | attribute | negotiated |
| readonly | unsigned short | id |
| readonly | RTCDataChannelState | readyState |
| readonly | unsigned long | bufferedAmount |
|  | EventHandler | onopen |
|  | EventHandler | onerror |
|  | EventHandler | onclose |
|  | EventHandler | onmessage |
|  | DOMString | binaryType |

**Table A-10. RTCDataChannelInit dictionary**

| Name | Type | Description |
| --- | --- | --- |
| id | unsigned short | Overrides the default selection of ID for this channel. |
| maxRetransmitTime | unsigned short | Limits the time during which the channel will retransmit data if not successfully delivered. |
| maxRetransmits | unsigned short | Limits the number of times a channel will retransmit data if not successfully delivered. |
| negotiated | boolean | Defaults to false. The default value of false tells the user agent to announce the channel in-band and instruct the other peer to dispatch a corresponding RTCDataChannel object. If set to true, it is up to the application to negotiate the channel and create an RTCDataChannel object with the same ID as the other peer. |
| ordered | boolean | Defaults to true. If set to false, data is allowed to be delivered out of order. The default value of true guarantees that data will be delivered in order. |
| protocol | DOMString | Defaults to "". Subprotocol name used for this channel. |

**Table A-11. RTCDataChannelState enumeration values**

| Value | Description |
| --- | --- |
| connecting | The user agent is attempting to establish the underlying data transport. This is the initial state of an RTCDataChannel object created with createDataChannel(). |
| open | The underlying data transport is established and communication is possible. This is the initial state of an RTCDataChannel object dispatched as a part of an RTCDataChannelEvent. |
| closing | The procedure to close down the underlying data transport has started. |
| closed | The underlying data transport has been closed or could not be established. |