WebRTC Application
Progress Report

Agam Agarwal (CS12B1003)
Rakshit Singla (CS12B1029)
Sachin Jaiswal (CS12B1033)
Aaditya Sapkal (ES12B1016)

November 29, 2014

Contents

1 Introduction

This project aims at developing a browser based Peer-to-peer video chat application using the WebRTC APIs. The application should work irrespective of intermediate network policies, user-agent differences and aims to include some sort of IPv6 support to WebRTC.

2 What is WebRTC?

Web Real Time Communication (WebRTC) is a collection of standards, protocols, and JavaScript APIs, the combination of which enables peer to peer audio, video, and data sharing between browsers (peers). Instead of relying on third party plug-ins or proprietary software, WebRTC turns real-time communication into a standard feature that any web application can leverage via a simple JavaScript API. Delivering rich, high quality RTC applications such as audio and video teleconferencing and peer to peer data exchange requires a lot of new functionality in the browser: audio and video processing capabilities, new application APIs, and support for half a dozen new network protocols. Thankfully, the browser abstracts most of this complexity behind three primary APIs -

2.1 MediaStream API

The MediaStream Processing API, often called the Media Stream API or the Stream API, is the part of WebRTC describing a stream of audio or video data, the methods for working with them, the constraints associated with the type of data, the success and error callbacks when using the data asynchronously, and the events that are fired during the process.

2.2 RTCPeerConnection API

The RTCPeerConnection interface represents a WebRTC connection and handles efficient streaming of data between two peers. Basic RTCPeerConnection usage involves negotiating a connection between the local machine and a remote one by generating Session Description Protocol messages to exchange between the two. i.e. The caller sends an offer. The called party responds with an answer. Both parties, the caller and the called party, set up their own RTCPeerConnection objects.

The RTCPeerConnection does the following jobs:

- RTCPeerConnection manages the full ICE work-flow for NAT traversal.
- RTCPeerConnection sends automatic (STUN) keep-alives between peers.
- RTCPeerConnection keeps track of local streams.
- RTCPeerConnection keeps track of remote streams.
- RTCPeerConnection triggers automatic stream recognition as required.
- RTCPeerConnection provides necessary APIs (methods) to generate the connection offer, accept the answer, allow us to query the connection for its current state, and many more.

2.3 RTCDataChannel API

RTCDataChannel API makes use of the RTCPeerConnection to enable agents to send and recieve arbitrary application data.

Below is an overview figure of the RTC Peer Connection API

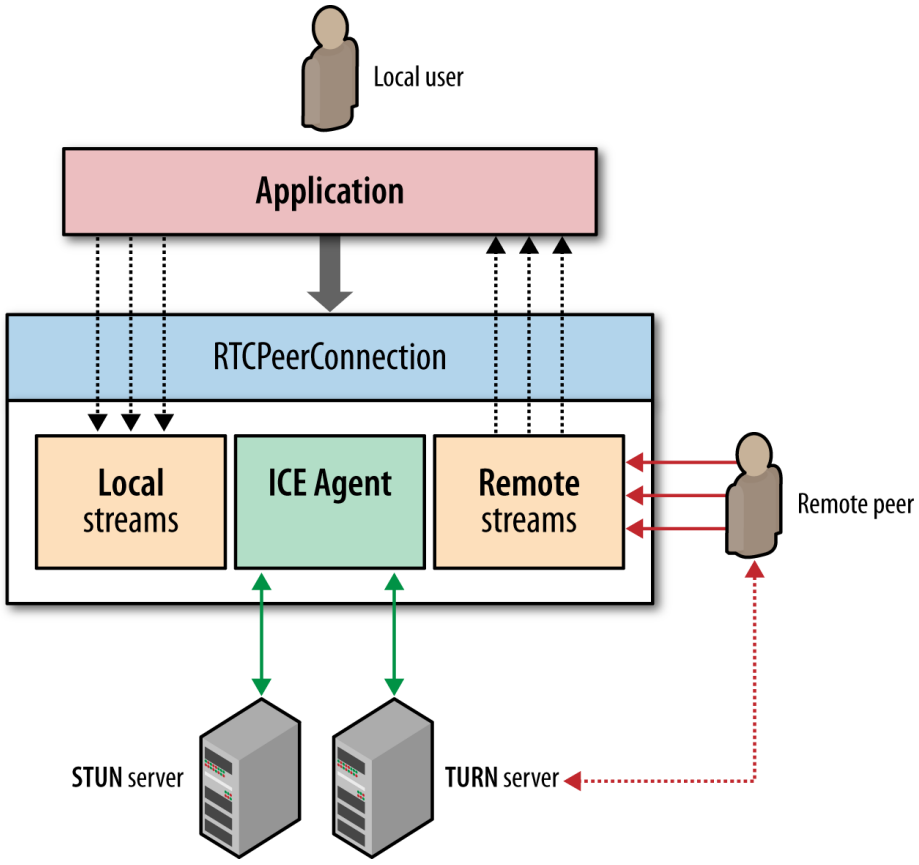

Figure 1: RTC PeerConnection API

3 Network Transport

WebRTC uses UDP as the Transport layer Protocol.Realtime communication is time sensitive and as result, audio and video streaming applications are designed to tolerate intermittent packet loss: the audio and video codecs can fill in small data gaps, often with minimal impact on the output quality.

The requirement for timeliness over reliability is the primary reason why the UDP protocol is a preferred transport for delivery of realtime data. TCP delivers a reliable, ordered stream of data: if an intermediate packet is lost, then TCP buffers all the packets after it, waits for a retransmission , and then delivers the stream in order to the application. UDP offers no promises on reliability or order of the data, and delivers each packet to the application the moment it arrives.

3.1 Making the P2P connection

WebRTC does provide P2P connection but it still needs servers for

- clients to coordinate metadata to coordinate communication, this is called signaling.
- coping with Network Address Translators(NATs).

3.2 P2P Connections and NAT

P2P connections require end-users to know each other's Transport Addresses (IP:PORT). However NATs can create problems it this process since they provide each user with an internal Transport address and translate it to an external transport address depending on the type of NAT. So, a remote peer would have to transmit data to the translated (global) address for the local peer but there was no way that a peer could know its translated transport address. To solve this problem STUN Protocol was standardised intially in RFC3489 and then revised in RFC5389.

4 Session Description Protocol (SDP [RFC4566])

When initiating multimedia teleconferences, voIP calls, streaming video, or other sessions, there is a requirement to convey media details, transport addresses, and other session description metadata to the participants.

SDP provides a standard representation for such information, irrespective of how that information is transported. SDP is purely a format for session description – it does not incorporate a transport protocol, and it is intended to use different transport protocols as appropriate.

5 Session Traversal Utilities for NAT(STUN)

STUN was originally defined in RFC 3489 as 'Simple Traversal of UDP over NAT'. That specification, sometimes referred to as "classic STUN", represented itself as a complete solution to the NAT traversal problem. In that solution, a client would discover whether it was behind a NAT, determine its NAT type, discover its IP address and port on the public side of the outermost NAT, and then utilize that IP address and port.

5.1 STUN and NATs

Clasic STUN works for simpler NAT types such as full-cone NAT and one-sided address-dependent NAT but it does not work for symmetric NAT or if both peers are behind address-dependent NATs. Since the IP address of the STUN server is different from that of the endpoint, a Symmetric NAT mapping will be different for the STUN server than for an endpoint. Thus for this and other security reasons, the classical STUN was made obsolete by the later specification (RFC5389) which describes STUN as a set of tools to be used as part of a complete NAT traversal solution.

ICE is one such complete NAT traversal solution which uses the 'offer/answer' model. WebRTC uses ICE along with STUN and TURN to make the P2P connections.

6 Traversal Using Relays around NAT (TURN)

TURN is a protocol specified in RFC5766 as an extension to STUN . It is used by clients to communicate with TURN servers. If a host is behind a NAT and it is impossible for that host to connect directly with other peers, then it has to make use of an intermediate node to relay data to the remote host.The intermediate node (TURN server) recieves data from one end-host and sends that to the other end-host. A client using TURN must have some way to communicate the relayed transport address to its peers, and to learn each peer's IP address and port.If TURN is used with ICE, then the relayed transport address and the IP addresses and ports of the peers are included in the ICE candidate information.

7 Interactive Connectivity Establishment(ICE)

ICE is defined in RFC5245 as a technique for NAT traversal for UDP-based media streams established by the offer/answer model. ICE is an extension to the offer/answer model, and works by including multiple IP addresses and ports in SDP(Session Description Protocol) offers and answers, which are then tested for connectivity by peer-to-peer connectivity checks. The IP addresses and ports included in the SDP and the connectivity checks are performed using the revised STUN specification (RFC5389). ICE also makes use of TURN, an extension to STUN.

8 System Design and Functionalities

The following APIs and services are used for developing the project -

- WebRTC - To make and manage the Peer-to-Peer Connections.
- MySQL - Used as back-end database server for signaling.
- PHP, jQuery, Bootstrap - Used to design the front-end.

8.1 Database Structure

The MySQL database used in the backend has two tables, one for the users and one for the sessions.

The user table has the details about the users and the sessions table as details like SDP offers and ICE candidates which are used for signaling and connections.

8.2 User Interface

There are two types of users in the system, Doctors and Students. When a user(say a doctor) logs in, he/she can see a list of online users of the other

type(students). The doctor can choose one of the active students and request to talk to them.

8.3 Maintaining active/busy users: Heartbeat

Every user has an active and a busy field associated with them. Active is set to true when they log in. Busy is set to true whenever that user is talking to someone. A heartbeat mechanism is employed which automatically changes these field on connection loss from the peers. If the requested user is not busy, they will be notified that someone wants to talk to them. If they accept the call, the receiver is also redirected to the chat page.

8.4 Initiating a call

As soon as a caller requests to talk, they are redirected to the chat page and they set their local Description objects. and send the SDP configuration and the ICE candidate to the signaling database. When the receiver accepts the call, they set their local objects and send the SDP and ICE candidates to the database to be set on the same session object. Both users can then set their remote Description objects and initiate the call.

8.5 Closing a call

When the call is disconnected, the session object is simply deleted from the sessions table in the database.

9 Future Works

This project is currently a work in progress. There is a lot of future work that can be done here.

9.1 IPv6 Support

Currently WebRTC works on IPv4 but since IPv4 is clearly not enough, WebRTC needs to have support for IPv6. Shifting to complete IPv6 will also eliminate the NAT traversal problem in the Peer-to-peer connectivity. Currently, Google gives IPv6 support for the Canary browser by using a pseudo constraint `googIPv6:true`. The user can control the use of IPv6 by setting `ipv6=true` but the API should handle this automatically and find the best possible way to connect two peers.

9.2 Muticast model

One possibility is to extend this peer-to-peer connection to a multicast like connection where a doctor is able to send their stream to multiple students simultaneously.

9.3 Storing session data

Another work is to store every session data(video/audio) on a central server so that it can be used later for analyses purposes. This problem deals with unstructured and big data handling and can probably make use of the DataChannel API provided in WebRTC for sending the data to the server.