[GSOC 2016] webrtc implementation

Table of contents:

General parameters for webrtc app

First of all I define a "webrtc" term. Webrtc is a set of APIs for work with:

- Capturing user media - get stream from camera or microphone via browser
- RTCPeerConnection that manage video/audio calls
- RTCDataChannel that allows to transmit custom data between peers if they already have established connection

Secondly I describe two specials servers for establishing connection between peers.

Mostly devices in the Internet have gray IP addresses. They connect to the Internet via provides.

Providers have a lot of clients and small pool of public (white) IP addresses therefore they assign one public IP to one active client.

It can be done in various ways. This process is called NAT - network address translation. There are two big type of NAT - symmetric and asymmetric.

Both can translates IP address and ports. One assigns IP for a long time but second do it for each request.

There are two servers - STUN and TURN. First is necessary for discovering your public IP address if you have gray address.

For example - you send some special message to the STUN and it responses with your NAT.

In order that establish connection between peers in the Internet you must:

1. discover public IP address via STUN server
2. if your NAT is dynamic thus STUN server doesn't help you and you must use TURN server that can transmit your data over the net
3. create RTCPeerConnection and etc (see Our signal server on Java paragraph)

There are many public STUN/TURN servers - https://gist.github.com/zziuni/3741933.

Also there are many open source java implementation of STUN/TURN protocols and servers.

I'm using a public google stun server for demo app. It doesn't support TURN mode but work fine for my local provider.

To sum up for our webrtc scheme you need have two servers - STUN and TURN.

Our signal server on Java

Establishing connection between peers demands one more public server - a Signal server.

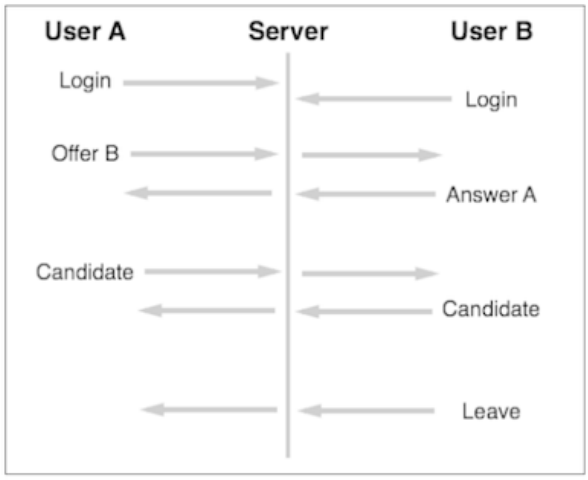
It uses for transmit auxiliary data. The following steps are typical that connect two peers:

Alice is going to call (video/audio) to Bob.

1. Alice create RTCPeerConnection object.
2. Then she calls RTCPeerConnection::CreateOffer(). Create offer method discovers Alice's public API via STUN server, gather her video/audio settings, pack this information and form a SDP packet.
3. Further Alice must transmit the sdp offer to Bob but she doesn't know Bob's IP address therefore she add pair "name=Bob" in the packet and send it ot the Signal server.
4. Signal server unpack the packet and retransmit offer to Bob.
5. Bob receives this packet and calls RTCPeerConnection::SetRemoteDescription(offer) on his RTCPeerConnection objects.
6. Then he creates an answer via RTCPeerConnection::CreateAnswer, append the pair "name=Alice" and sends it to the signal server.
7. Server handles answer from Bob and retransmit answer to Alice.
8. Alice receives answer from Bob and calls RTCPeerConnection::SetRemoteDesription(answer)
9. Also both Alice and Bob calls RTCPeerConnection::SetLocalDescription() with theirs offers (or in Bob's case answer).
10. Now they can calls RTCPeerConnection::onTrack(stream) and this stream will sends directly to another peer.

This is a short description how to connect two peers because client must exchange ice candidates too but it's similar on offer/answer exchanging.

The following picture illustrates typical work-flow.



Our signal server.

Our signal server uses JSON format for message and work via web-socket's protocol.

Why have I used JSON and web-sockets? Because JSON is a native format for JS in browsers and the web-socket is the fastest protocol for reliable connections.

Also web-sockets are integrated into JS and are supported by all modern browsers.

A structure of particular packet you can see in the code. It's attached to the this documentation.

Our signal server works under red5 server and uses tomcat-embedded web-sockets.

The RTCPeerConnection API is oriented on p2p connections but our signal server also supports room.

You can create a new room just via sending createRoom message and you can connect to the room by id.

I've taken care about the flood attack on server so when you connect to the room you shouldn't create RTCPeer object and allocate any resources.

Signal server transmits your connecting message to all other clients in the room and they send your offer later.

A few words about demo client in browser - it's written on JS and supports our signal server.

It also can be improved via using some cross-browser library for RTCPeerConnection (https://github.com/otalk/rtcpeerconnection).

You also can test signal server - all instructions are in README file.

Native client

As I mention in Our signal server chapter the RTCPeerConnection API is oriented on p2p connections but It allows to create client-server solutions.

I need to implement server-to-client solution because OM works on server-client architecture.

For this purposes I've invented the next solution.

The solution is to add our native client into communication and dump video stream from native client on hard disk.

This will work because we use our signal server that can do anything for us.

I've tryed to create RTCPeerConnection on java but there aren't any java api for this purposes.

There are only js api and some abandoned demos in the Internet but I've found the repository with native (c++) webrtc peerConnection api for browsers.

This native implemenation includes java api for android. But It also broken, you can see explanation here - https://groups.google.com/forum/#!topic/discuss-webrtc/BUPq2UxIyq8.

I've implementing RTCPeerConnection api for java. I used the native webrtc source and JNI because implementing webrtc requires a lot of protocols (SRTP, SCTP, RTP etc).

I would have done this JNI stuff but I hadn't been stuck with one problem. I couldn't glue JNI and multi-threading native client. I've received JVM errors about damaged frame stack.

So I've implemented demo native client that works via console as signaling. The native client's code is attached to the this doc.

To build and test it you must do the following tasks:

1. Go to the https://webrtc.org/native-code/development/ and install depot tools
2. Then checkout chromium webrtc sources. Instructions are here - https://webrtc.org/native-code/development/. Check-outing will borrow a lot of time ~ 1 hour for me.
3. Then copy my native client code to src/webrtc/examples/peerconnection/linux (or windows if you use it)
4. Then you must install ninja build system and run
5. `$ ninja -C out/Debug`
6. in your src directory
7. Finally the peer_connection_client must arrive in your out/Debug directory
8. Then just execute it and test. It provides testing instructions.

It can connect to the browser RTCPeerConnection and dump stream into hard disk.

Links

OMStreamSaver-master.zip

signalling_server-master.zip

https://github.com/Dima00782/signalling_server

https://github.com/Dima00782/OMStreamSaver