Development of a WebRTC application for Firefox/Chrome

Alex Albàs & Gerard Auguets
Universitat Politècnica de Catalunya (UPC)

Outline

Outline

Motivation I

- The application that is going to be developed is web-oriented and focused on real time communications.
- This kind of technology has been developed by Google in order to use on its browsers, thing that make easier the fact no plugins and extensions are required to install in order to use it.
- To start, we will have to treat many aspects:
  1. WebRTC API's and technologies.
  2. HTML language.
  3. JavaScript language.
  4. Callback concept.
  5. Server made with Node.js technology.
  6. Signaling process.
  7. Latex.

Motivation II

- Other related knowledges are needed, those that have been acquired during this years at the university, thing that will make easier all the absorption of concepts above:
  1. Object Oriented Programming.
  2. Multimedia coding.
  3. Audio/video concepts.
  4. Telematic concepts.
  5. Unix/Linux environment.
- The main goal of this project is help us to get started with WebRTC by researching and understanding this technology and finally trying to develop a simple WebRTC site from scratch.

Outline

Objectives

- Develop a web application through an expanding technology as WebRTC to:
  1. Establish multimedia communication between peers.
  2. Room treatment.
  3. Text chat communication.
  4. File sharing.
- The application should have the following objectives:
  1. A Node.js Server.
  2. Web-page made with html and JavaScript code.
  3. Handle streams.
  4. HTTPS secure connection.
  5. File Sharing.

Outline

Outline

History

- The web evolution...
  1. 1990:Navigation by HREF-based hyperlinks.
  2. 2000: XMLHttpRequest (XHR), no need to update their content. Allow server-based web services like Facebook, Gmail, etc..
  3. Nowadays: NO servers required to transmit data between peers.

Overview

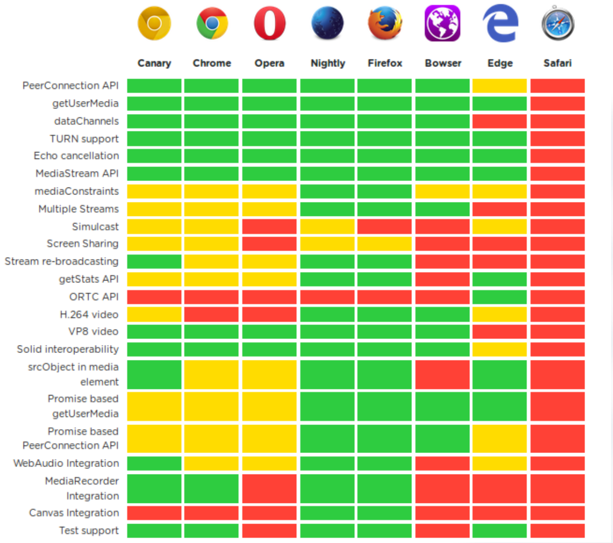
- Direct peer to peer communications should provide:
  1. Lower latency.
  2. Multiplayer gaming.
  3. Video/audio streaming.
  4. Sensor data feeds.
  5. Secure P2P connections.
- So it allows the fact of exchanging information privately without being managed by intermediary servers. It introduces a way to create new types of services and applications.
- This is just a brief overview of how WebRTC can be used.

Outline

Trade Off

- Advantages:
  1. No plugins required.
  2. Only a handshaking process (Signaling).
  3. No server required to exchanging data.
  4. File sharing or text chat (Data channel).
  5. Better encryption.
- Drawbacks:
  1. Vulnerable on the signaling process.
  2. Not supported for all browsers.
  3. Mobiles are not supported yet.
  4. Hard to scaling.

Browser support



	Canary	Chrome	Opera	Nightly	Firefox	Bowser	Edge	Safari
PeerConnection API	🟩	🟩	🟩	🟩	🟩	🟩	🟨	🟥
getUserMedia	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟥
dataChannels	🟩	🟩	🟩	🟩	🟩	🟩	🟥	🟥
TURN support	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟥
Echo cancellation	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟥
MediaStream API	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟥
mediaConstraints	🟨	🟨	🟨	🟩	🟩	🟨	🟨	🟥
Multiple Streams	🟨	🟨	🟨	🟩	🟩	🟩	🟥	🟥
Simulcast	🟨	🟨	🟥	🟨	🟥	🟥	🟨	🟥
Screen Sharing	🟨	🟨	🟥	🟨	🟨	🟥	🟥	🟥
Stream re-broadcasting	🟩	🟨	🟨	🟩	🟩	🟥	🟥	🟥
getStats API	🟨	🟨	🟨	🟩	🟩	🟥	🟩	🟥
ORTC API	🟥	🟥	🟥	🟥	🟥	🟥	🟩	🟥
H.264 video	🟨	🟥	🟥	🟩	🟩	🟩	🟨	🟥
VP8 video	🟩	🟩	🟩	🟩	🟩	🟩	🟥	🟥
Solid interoperability	🟩	🟩	🟩	🟩	🟩	🟩	🟨	🟥
srcObject in media element	🟩	🟨	🟨	🟩	🟩	🟥	🟩	🟥
Promise based getUserMedia	🟨	🟨	🟨	🟩	🟩	🟩	🟩	🟥
Promise based PeerConnection API	🟨	🟨	🟨	🟩	🟩	🟩	🟨	🟥
WebAudio Integration	🟩	🟨	🟨	🟩	🟩	🟥	🟨	🟥
MediaRecorder Integration	🟩	🟩	🟥	🟩	🟩	🟥	🟥	🟥
Canvas Integration	🟥	🟥	🟥	🟩	🟩	🟥	🟥	🟥
Test support	🟩	🟩	🟥	🟩	🟩	🟥	🟩	🟥

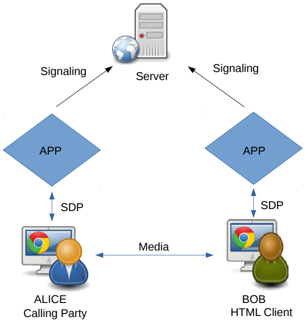
Outline

Signaling Overview

- Analogy:
- Private Branch exchange
- Signaling is an extremely important part of webRTC. In order for a WebRTC application to set up a 'call', its clients need to exchange information:
  1. Session control messages used to open or close communication.
  2. Error messages.
  3. Media meta data such as codecs and codec settings, bandwidth and media types.
  4. Key data, used to establish secure connections.
  5. Network data, such as a host's IP address and port as seen by the outside world.

Signaling I

- Signaling methods and protocols are not defined on the WebRTC standards in order to maximize compatibility with established technologies.
- The JavaScript Session Establishment Protocol (JSEP) defines an approach for the fully application but excludes the signaling methods to use.



Signaling II

- The offer/Answer contains the Session Description Protocol (SDP) that features all the characteristics to the data (video & audio) to be exchanged:
  1. Media (Audio & Video) Codecs.
  2. Resolution format.
  3. Transport protocol used.
  4. Endpoint IP address and port.
  5. Other info to describe a media transfer endpoint.

SDP

- SDP Components:
  1. Session Description
  2. Time Description
  3. Media Description
- An SDP session description consists of a number of lines of text of the form: `<type>=<value>`.
- where:
  1. Type must be exactly one case-significant character
  2. Value is structured text whose format depends on <type>. In general, <value> is either a number of fields delimited by a single space character or a free format string.
  3. No white spaces allowed on both sides of "=" sign.
  4. Optional items are marked with a "*".

SDP Example

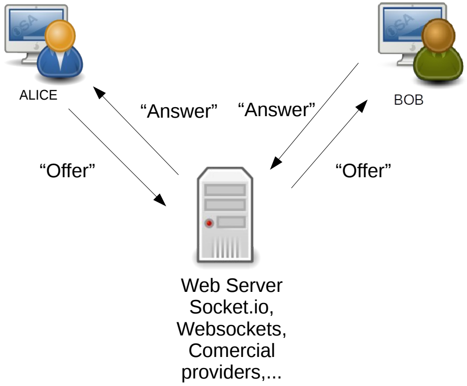
- SDP example:

```
v=0;
o= jdoe 2890844526 2890842807 IN IP4 10.47.16.5;
s= SDP WebRTC Alex—Gerard;
i= A example of SDP used in WebRTC;
u=http://Alex—Gerard—EetWebRTC—PFG.com;
e=j.eetwebrtc@gmail.com (Alex Albas, Gerard Auguets);
c=IN IP4 224.2.17.12/127;
t=2873397496 2873404696;
m= audio 54609 UDP/TLS/RTP/SAVPF;
a=rtpmap:0 PCMU/8000;
a=rtcp—mux; \\Alice can perform RTP/RTCP Mux
a=rtcp:54609 IN IP4 24.23.204.141 \\Port for RTCP data
a=ptime:60   \\Audio packetization of 60 ms
a=sendrecv \\Alice can send and receive audio
a=setup:actpass \\Alice can perform DTLS before answer arrives
a=fingerprint:sha—1 99:41:49:83:4a:97:0e:1f:ef:6d:f7:c9:c7:70:
9d:1f:66:79:a8:07 \\DTLS Fingerprint for SRTP
a=ice—ufrag:074c6550 \\ICE user fragment
a=ice—pwd:a28a397a4c3f31747d1ee3 \\ICE password
a=candidate:0 1 UDP  2122194687
192.168.1.4 54609 typ host \\RTP Host Candidate
a=candidate:0 2 UDP 2122194687
192.168.1.4 54609 typ host \\RTCP Host Candidate
a=candidate:1 1 UDP  1685987071 24.23.204.141 64678 typ srflx
addr 192.168.1.4 rport 54609  \\RTP Server Reflexive ICE Candidate
a=candidate:1 2 UDP  1685987071 24.23.204.141 64678 typ srflx
raddr 192.168.1.4 rport 54609\\RTCP Server Reflexive Candidate
a=rtcp—fb:109 nack \\Indicates NACK RTCP feedback support
a=rtcp—rsize Alice intends to use reduced size RTCP for this session
```

Example

- The initial signaling steps:
  1. Alice wants to reach Bob's browser, so an Offer is created and sent to the Server.
  2. Alice adds its offer to setLocalDescription().
  3. Bob reach the server and receive the offer generated by Alice.
  4. Bob adds to its setRemoteDescription() the received offer and create an answer to the server.
  5. Bob adds its answer to its setLocalDescription().
  6. Alice receive the answer from Bob and adds it to its setRemoteDescription().

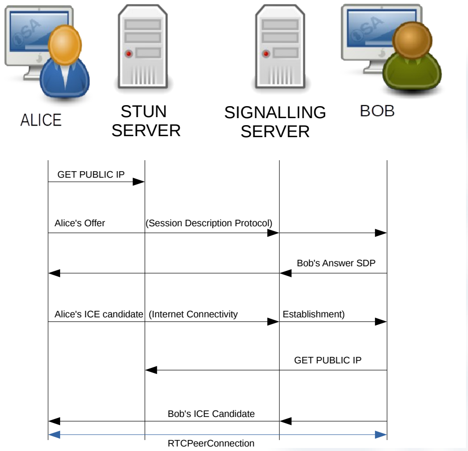
Initial Scenario



ICE Candidates

- Peers must exchange information about the network connection. This is known as an **ICE Candidate** and it details the available methods that the peer is able to communicate (directly or through a TURN server).
- In other words, look for the best way to reach the other part through:
  1. Stun Server.
  2. Turn Server.
- The claim of ICE candidates is discover which transport address are valid, so **ICE framework** will try to find the best ICE candidate until works.
- Once Offer/Answer have been sent:
  1. ICE Candidate is going back and forward between Alice and Bob by the Signaling Server.
  2. Generally both could get an public IP, but if there is a firewall in the middle, other IP is given.
  3. When an ICE candidate is ready, it is possible to establish an **RTCPeerConnection**.

Initial Scenario



Outline

WebRTC API's

- There are three main categories of API that exist in WebRTC.
  - Acquiring audio and video
  - Communicating audio and video
  - Communicating arbitrary data
- Because this three categories, WebRTC has three primary objects in order to acces to this components.
  - MediaStream API
  - RTCPeerConnection API
  - RTCDataChannel API

Outline

getUserMedia Method

- This API is the responsible for managing all the media sources of the laptop, for example the video of the webcam and the audio of the microphone.
- Thanks to the MediaStream API the user can get access to all the laptop devices using the getUserMedia() method.

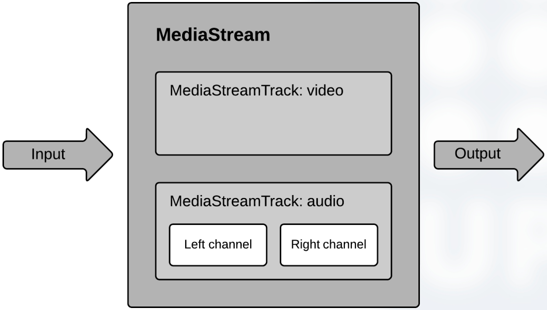
Example

- The following code shows a simple getUserMedia implementation:

```
var constraints = {video: true};

function succesCallback(stream) {
var video = document.querySelector("video");
video.src = window.URL.createObjectURL(stream);
}

function errorCallback(error){
console.log("navigator.getUserMedia error ", error);
}

navigator.getUserMedia(constraints,succesCallback,
    errorCallback);
```

Mediastream

- A MediaStreamTrack is a single source media element with an unique ID for example the audio stream provided by the microphone.
- A MediaStream Represents a single source of synchronized audio, video or both. Each MediaStream contains one or more MediaStream tracks.



Outline

Introduction

- RTCPeerConnection:
- The interface represents a WebRTC connection between peers, maintain and monitor the connection, and close the connection once it's no longer needed.
- This API doesn't handle the signaling process
- The main functionality of this application is to transmit the acquired flow by the getUserMedia API and send it to the other peer.
- Management of all the parameters necessary to carry out transmission of multimedia data stream.

RTCPeerConnection handling

It is responsible for managing the following:

- Signal processing and voice engine to remove noise from audio and video data and provide echo cancellation.
- Codec handling and video engine.
- Compression and decompression of audio and video.
- Transport and Peer to peer communication in order to find the optimal route through firewalls, NATs and relays.
- Security, encrypting the data to guarantee secure communications.
- Bandwidth management.

RTCPeerConnection flow

1. RTCPeerConnection API only has to send and receive MediaStreamTracks over a peer-to-peer connection.
2. The encoding of MediaStreamTracks is managed by objects called RTCRtpSenders.
3. The reception and decoding of MediaStreamTracks is managed through objects called RTCRtpReceivers.
4. When a MediaStreamTrack is attached to a RTCPeerConnection, a RTCRtpSenders object is created to be associated with this MediaStreamTrack.
5. RTCRtpReceivers are created when remote signaling indicates that there are new tracks available.
6. The MediaStreamTrack and its associated RTCRtpReceivers are surfaced to the application.

Outline

Introduction

- Sending information is present in many areas like communications, file transfer and video games. This requires fast communications, low-latency and guaranteeing the security of the data.
- Currently there are other data channels as Websockets, Ajax and many more, but these technologies are designed for client/server communications.
- RTCDataChannel works directly with the RTCPeerConnection API.
- RTCDataChannel is designed to replicate exactly the Websockets technology.

Transmission Modes

1. **Unreliable mode**: This mode does not guarantee the correct transmission of all the packets because works through UDP protocol, but eliminates the overhead so it is a faster mode, especially useful for gaming applications.
2. **Reliable mode**: This mode guarantee the transmission and order of all the packets because works through TCP protocol, but adds overhead in order to handling the control and error flow.

RTCDataChannel Example

```
var peerConnection = new RTCPeerConnection();
var dataChannel =
  peerConnection.createDataChannel("STUN SERVERS", dataChannelOptions);

dataChannel.onerror = function (error) { //Manages error handling
  console.log("Data Channel Error:", error);
};
dataChannel.onmessage = function (event) { //Specifies a function to be called when the message event is fired on the channel.
  console.log("Got Data Channel Message:", event.data);
};
dataChannel.onopen = function () { //Manages the channel aperture
  dataChannel.send("Hello World!");
};
dataChannel.onclose = function () {
  console.log("The Data Channel is Closed"); //Manage the channel closure
};
```

Outline

Introduction

- WebRTC implementations can transfer sensitive content.
- Ensuring communications security is a fundamental requirement for this type of applications.
- The WebRTC architecture assumes from a security perspective that network resources exist in a hierarchy of trust.
- The browser's job is to enable access to the Internet providing adequate security protections to the user.
- The browser is the portal through which the user accesses all WebRTC applications and content.
- The level of trust provided to the user by WebRTC is directly influenced by the user's trust in the browser.

WebRTC Security Mechanisms

- WebRTC is not a plugin.
- The browser can access local resources. WebRTC combat this by requiring the user to give explicit permission to the user.
- When either the microphone or camera is being used the client UI is required to expressly show the user that the microphone or camera are being operated.

HTTPS

- HTTPS (Hypertext transfer protocol secure) is a communication protocol that uses the HTTP (Hypertext transfer protocol secure) and the SSL/TLS (Secure sockets layer/Transport layer security) protocols to provide encrypted communication and secure identification of a Web Server.
- This technology consist of communication over HTTP within a connection encrypted by transport layer security or secure sockets layer.
- It creates a secure channel over an insecure network.
- HTTP operates at the application layer.

HTTPS encryption & DTLS

- Full encryption P2P.
- The encryption protocol used depends on the channel type.
  1. Data streams are encrypted using Datagram Transport Layer Security (DTLS).
  2. Media streams are encrypted using Secure Real-time Transport Protocol (SRTP).
- DTLS is a standardized protocol used in email and voip communications to encrypt information.
- Offers full encryption with asymmetric cryptography methods,data authentication and message authentication.
- SRTP is used for the mediaStreams encryption because is a lighter-weight option than DTLS.
- SRTP uses AES,as a symmetric-key algorithm, meaning the same key is used for both encrypting and decrypting.

Outline

Outline

Server Side

- In the following code all the modules needed are imported using the require() method.

```
//Getting the modules
var https = require('https');
var fs = require('fs');
var path = require('path');
var express = require('express.io');
```

Server Side

- In order to enable the server to handle HTTPS connections, the server need to read the keys and certificates created through OPENSSL based on Linux.

```
  $ openssl genrsa 1024 > file.pem //Generating RSA
       KEY.
  $ openssl req -new -key file.pem -out csr.pem
  //Input data in order to generate crs.pem
  $ openssl x509 -req -days 365 -in csr.pem
      -signkey file.pem -out file.crt
  //SSL certificate saved
```

- Uses of .pem & .crt using relative paths.

```
  //Key & Certificate for https server.
  var options = {
key:fs.readFileSync('./cert/file.pem'),
cert:fs.readFileSync('./cert/file.crt')
  };
```

Server Side

- HTTPS server creation, request and response.

```
//Server Creation
app.https(options).io();

//Defined HTTPS server port
var PORT = 8102;
console.log('server started and listen to port: ' + 
     PORT);

//Ask for static files on the public directory
app.use(express.static(path.join(__dirname, 'public
    ')));

//When a request comes index.ejs is provide to the
    client
app.get('/', function(req, res){
    res.render('index.ejs');
});
```

Server Side

- Every time that a client connect/disconnect to the server a message on the console of the server will be printed.

```
//Display on the console connected/disconnected
    users
app.io.on('connection' , function(socket){
    console.log("A user connected!!");
    socket.on('disconnect', function() {
    // make your disconnection actions
    console.log("User disconnected!!");
    })
})
```

Server Side

- Joining into the room introduced by the user at the beginning of the connection.

```
//message other clients when some new user enter to
     the chatroom
app.io.route('ready', function(req){
    //req.data bring the name of the room (for
        textchat and signaling room)
    req.io.join(req.data.chat_room);
    req.io.join(req.data.signal_room);
    req.io.join(req.data.files_room);
    //Send a broadcast message to the room with
        name req.data
    app.io.room(req.data).broadcast('announce', {
        message:'New client in the ' + req.data + '
            room.'
    });
})
```

Server Side

- This function set up the listening port able to receive connections:

```
app.listen(PORT);
```

- Similar to text-chat Implementation. Key difference:The sender will not receive his own message because of using req() method instead off app.io.room.

```
//Signaling process
app.io.route('signal', function(req){
    req.io.room(req.data.room).broadcast('
        signaling_message', {
        type:req.data.type,
        message:req.data.message});
})
```

Server Side

- For the text chat Broadcast message to all the clients (sender included), that are connected in to the same room.

```
//Sending text messages to the connected clients
app.io.route('send', function(req){
    app.io.room(req.data.room).broadcast('message',
           {
        message:req.data.message,
        author:req.data.author});
        })
```

Outline

Client Side I

Main Aspects:

- The Core: Creating The RTCPeerConnection on the signaling process:

```
//Setting up the RTC Peer Connection object
if (!rtcPeerConn) {
console.log("Start Signaling?");
startSignaling();
console.log('Yes, Signalingng has been started!!');
}
if (data.type != "user_here") {
var message = JSON.parse(data.message);
  if (message.sdp) {
    rtcPeerConn.setRemoteDescription(new
        RTCSessionDescription(
  message.sdp), function () {
  //if we have received an offer, we need to answer
  if (rtcPeerConn.remoteDescription.type == 'offer'
      ) {
  rtcPeerConn.createAnswer(sendLocalDesc, logError)
      ;
  }
```

Client Side II

```
  }, logError);
  } else {
    rtcPeerConn.addIceCandidate(new RTCIceCandidate
        (message.candidate));
  }
  }
});
```

StartSignaling Method I

- Starting the signaling process:

```
function startSignaling() {
displaySignalMessage("starting signaling...");
rtcPeerConn = new RTCPeerConnection(configuration,
    null);
//Send any ice candidates to the other peer
rtcPeerConn.onicecandidate = function (evt) {
if (evt.candidate) {
io.emit('signal', {
"type": "ice candidate",
"message": JSON.stringify({
'candidate': evt.candidate
}),
"room": SIGNAL_ROOM
});
displaySignalMessage("Completed that ICE
candidate");
}
};
//let the 'negotiationneeded' event trigger offer
generation (SDP offer)
```

StartSignaling Method II

```
rtcPeerConn.onnegotiationneeded = function () {
displaySignalMessage("on negotiation called");
rtcPeerConn.createOffer(sendLocalDesc, logError);
};

//Once remote stream arrives, show it in the remote
video element
rtcPeerConn.onaddstream = function (evt) {
displaySignalMessage("going to add their stream...
");
theirVideoArea.src = URL.createObjectURL(
evt.stream);
};
```

GetUSerMedia

- Getting the media when the Signaling process has been completed.

```
//This goes inside startSignaling() function.
navigator.getUserMedia({
'audio': true,
'video': true
//Once the stream is created, display in myVideo
tag, and add our stream as a source of audio/
video to our RTCPeerConn
}, function (stream) {
displaySignalMessage("going to display my
stream...");
//Adding the local stream to the video tag.
myVideoArea.src = URL.createObjectURL(stream);
//
rtcPeerConn.addStream(stream);
}, logError);
```

Outline

Testing

- LET'S TRY

Outline

Testing

- Satisfactory results for the project development and the system implementation.
- Academic-oriented.
- Successfully technical knowledge.
- WebRTC fulfill generated expectations.