Bern University of Applied Sciences

BTI7311p - Informatik Seminar

State of WebRTC and its use cases

User Manual

Authors:
Frédéric N. Lehmann

Tutor:
Dr. Simon Kramer

March 13, 2020

Abstract

In this paper we will explore the state of WebRTC and its use cases. WebRTC is a web API for real-time communication on a peer to peer basis. Goal of this work will be to document the state of WebRTC and its real world applications. We will explore those possibilities on the basis of example applications and their implementations.

Contents

List of Figures

1 Introduction

WebRTC is short for web real-time communication, it is an API that modern browser support and can be used by web developers to implement a peer to peer communication. It can be used to capture and stream audio and/or video data, as well as to exchange arbitrary data between browsers. This technology does not require an intermediary.

1.1 Web Browser Support

All major browser support WebRTC in its newest release. Older versions might not, or only partially, implement this API so the Adapter.js [1] project should be considered for productive solutions. For detailed information on supported browsers use caniuse [2].

1.2 Signaling server

Although the WebRTC is a peer to peer communication API it can not fully function without a server. It needs a signaling server to resolve how to connect peers over the internet. The signaling server is an intermediary, so two peers find each other and can establish a connection. After the peers have found each other and have exchanged their negotiation messages they don't need the signaling server anymore.

1.3 Related API's

There are multiple related topics to WebRTC. In this section we'll try to give a quick overview over the most important ones.

Media Capture and Streams API

This API is heavily related to WebRTC, it provides support for streaming audio and video data. Provided are interfaces and methods for working with the success and error callbacks when using the data asynchronously and the events that are fired during the process, as well as the constraints associated with data formats.

2 WebRTC

2.1 Session Establishment

The session establishment uses different network methods to create connection to a peer. This also includes substitutions for situations where the default connection can not be established.

2.1.1 Network Address Translation (NAT)

Is used to give devices in a network a public IP address. This is achieved by translating requests from the device's private IP to the router's public IP with a unique port. The goal is to not need a unique public IP for each device.

2.1.2 Session Traversal Utilities for NAT (STUN)

This protocol is used to discover the public address of the peer. It also will determine any restrictions that would prevent a direct connection with a peer.

The peer sends a 'who am i' request to a STUN server which responds with the public address of the peer.

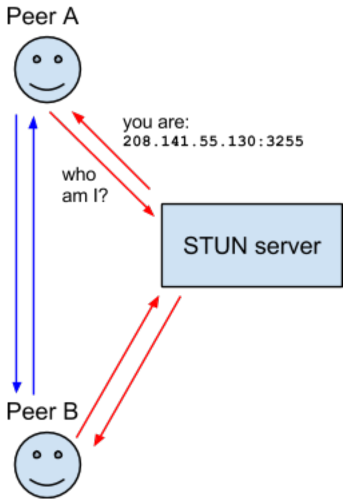

Figure 2.1: STUN communication schema, https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Protocols

There are open STUN servers available (list might not complete):

- stun.l.google.com:19302
- stun[1-4].l.google.com:19302
- stunserver.org
- stun.schlund.de
- stun.voipstunt.com

2.1.3 Traversal Using Relays around NAT (TURN)

If STUN can't be used, because for example 'Symmetric NAT' is employed in the network, TURN will be used as fallback. This is achieved by opening a connection with a TURN server, this server then will relaying all information through that server.

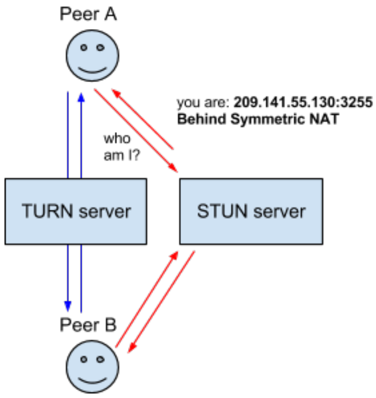

Figure 2.2: TURN communication schema, https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Protocols

There are open TURN servers, for example provided by google. But this will mean all communication is going threw a foreign server which might not be acceptable.

2.1.4 Session Description Protocol (SDP)

This standard describes the multimedia content of a connection. This includes a resolution, formats, codecs, encryption, etc. basically it is the metadata describing the content not the content itself.

2.1.5 Interactive Connectivity Establishment (ICE) candidates

Peers have to exchange information about the network connection, this is known as an ICE candidate. Each peer proposes its best candidate, and will work down to the worst candidate until they agree on a common candidate.

2.1.6 Complete communication schema

The following figure gives an overview over the complete communication mechanism. They also include the fallback mechanisms in case the default is not acceptable.

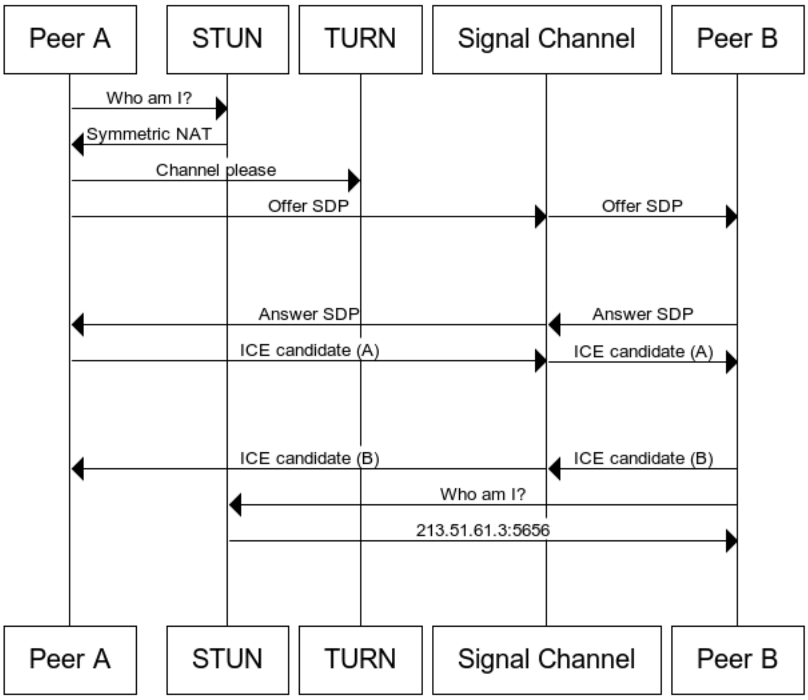

Figure 2.3: WebRTC Complete communication schema, `https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Connectivity`

2.2 Security

Generally WebRTC traffic is encrypted using Datagram Transport Layer Security (DTLS). Your data will be as secure as using any standard SSL based connection. Traffic that is relayed over a TURN server on the other hand is not necessarily end-to-end encrypted.

Confidentiality for the application data relayed by TURN is best provided by the application protocol itself, since running TURN over TLS does not protect application data between the server and the peer. If confidentiality of application data is important, then the application should encrypt or otherwise protect its data. For example, for real-time media, confidentiality can be provided by using SRTP. [6]

2.3 example applications

In this chapter we'll show some example applications and implementations. These are in a minified version and do not necessarily represent the best practices that should be applied. Where ever possible sources will be provided where those best practices can be read on.

These examples show versions where the sender and receiver are the same client. For a real world application those code parts would be separated. The needed communication information, like offer, answer or ICE candidate, would be transferred through a signaling service which is not part of the WebRTC specs. Additionally the ICE candidates need to be negotiated between the peers. This will be explored in a following section.

2.3.1 Connect

This examples shows in a minimal way, how to create a RTC connection. Important in this case is that, the sender creates an offer which is then used to create the `localDescription` for the sender and the `remoteDescription` for the receiver. The receiver on the other hand creates an answer which is used to set the `localDescription` of the receiver and the `remoteDescription` of the sender.

Since sender and receiver are the same client we can set the ICE candidate in a minimal way. These process would be more complex in a real world application.

```
// Create connections
const sender = new RTCPeerConnection();
const receiver = new RTCPeerConnection();

// Set ICE candidate
sender.onicecandidate = e =>
  !e.candidate || receiver.addIceCandidate(e.candidate);
receiver.onicecandidate = e =>
  !e.candidate || sender.addIceCandidate(e.candidate);

// Create offer and set description
const offer = await sender.createOffer();
await sender.setLocalDescription(offer);
await receiver.setRemoteDescription(offer);

// Create answer from receiver and set description
const answer = await receiver.createAnswer();
await receiver.setLocalDescription(answer);
await sender.setRemoteDescription(answer);
```

Working source: [7]

2.3.2 Disconnect

The connection can be closed by simply call the close function of the RTC connection.

```
sender.close();
receiver.close();
```

Working source: [8]

2.3.3 Sending Data

In this section we will showcase the ability to transfer arbitrary data between peers. In our case we will send text data, but it could be data in any format.

The code only contains the necessary lines. Code that does not provide insights on the topic got removed.

```
let senderChannel;

async function send() {
  // Send data
  senderChannel.send(document.querySelector("#senderArea").value);
}

async function init() {
  let sender = new RTCPeerConnection(null);
  senderChannel = sender.createDataChannel("sendDataChannel");
  let receiver = new RTCPeerConnection(null);

  // listen on data received
  receiver.ondatachannel = e => {
    e.channel.onmessage = event => {
      document.querySelector("#recieverArea").value = event.data;
    };
  };

  const offer = await sender.createOffer();
  sender.setLocalDescription(offer);
  receiver.setRemoteDescription(offer);

  const answer = await receiver.createAnswer();
  receiver.setLocalDescription(answer);
  sender.setRemoteDescription(answer);
}
```

Working source: [9]

2.3.4 Video chat

Accessing client media - Feature check

We need to check if the current environment supports the needed API's. Here is an example of such a check.

```
function hasUserMedia() {
  return !!(navigator.mediaDevices
    && navigator.mediaDevices.getUserMedia);
}
```

Working source: [5]

Accessing client media - Access media

For accessing client media, such as audio and video, the users consent is needed. This permission request is handled by the browser automatically. The application on the other hand needs to handle the case of not getting the permissions needed.

In this example we see how client media, in this case video and audio, can be accessed.

```
const constraints = {
  audio: true,
  video: true
};

navigator
  .mediaDevices
  .getUserMedia(constraints)
  .then(stream => {
    // use the stream, for example to present to the user
  });
```

Working source: [4]

Send Media

In the following example we will take a look on how media can be send between peers.

The code only contains the necessary lines. Code that does not provide insights on the topic got removed.

```
async function start() {
  // Start the video / audio stream
  const stream = await startStream();

  // Create sender / receiver connection
  let sender = new RTCPeerConnection();
  let receiver = new RTCPeerConnection();

  // Listen on incoming stream
  receiver.ontrack = e => {
    document.querySelector("#remoteVideo").srcObject = e.streams[0];
  };

  // Bind stream to sender
  stream.getTracks().forEach(track => sender.addTrack(track, stream));

  // Create offer and answer and set the corresponding descriptions
}
```

Working source: [10]

2.3.5 ICE Candidates

In a real world application we need to gather possible ICE candidates for our connection. This is achieved by gathering the sending clients ICE candidates and exchange them with the ICE candidates from the receiving client. This starts with the highest priority candidates and continues to the lowest until a common candidate is found. The `onicecandidate` event handler is used to listen for incoming ICE candidates.

2.4 Signaling

Here will be a short signaling example

2.5 Signaling

Here will be a short signaling example

3 Conclusion

3.0.1 STUN / TURN Server

To not be dependent from open STUN/TURN servers one can deploy his own servers. There are open source projects covering this case. For example coturn [3].

3.0.2 Videobridge

Video conferencing can be a resource intensive task for a browser. Which leads it to be a solution that is not really scalable. Videobridges can help tackle this issue. The Jitsi videobridge is an open source example of such a videobridge. It is used to implement scalable video conferencing platforms. It is an Selective Forwarding Unit (SFU) which relays video streams between peers.

Bibliography

[1] *Adapter.js*. URL: https://github.com/webrtc/adapter (visited on 03/03/2020).

[2] *Can I Use*. 2020. URL: https://caniuse.com/.

[3] *Coturn*. URL: https://github.com/coturn/coturn (visited on 03/03/2020).

[4] *Media access*. URL: https://codesandbox.io/s/access-device-uwvby (visited on 03/11/2020).

[5] *Media feature check*. URL: https://codesandbox.io/s/feature-check-xqv8t (visited on 03/11/2020).

[6] *TURN security*. URL: https://tools.ietf.org/html/rfc5766#section-17.1.6 (visited on 03/03/2020).

[7] *WebRTC connect*. URL: https://codesandbox.io/s/open-connection-lri4p (visited on 03/11/2020).

[8] *WebRTC disconnect*. URL: https://codesandbox.io/s/close-connection-lvzcw (visited on 03/11/2020).

[9] *WebRTC send data*. URL: https://codesandbox.io/s/send-data-zy5r9 (visited on 03/11/2020).

[10] *WebRTC send media*. URL: https://codesandbox.io/s/close-connection-lvzcw (visited on 03/11/2020).

Glossary

API application programming interface. 2, 5, 11

WebRTC Web real-time communication. 2, 5, 6, 9, 10