Bern University of Applied Sciences

BTI7311p - Informatik Seminar

State of WebRTC and some Use Cases

State of Progress Report

Authors:
Frédéric N. Lehmann

Tutor:
Dr. Simon Kramer

March 17, 2020

Abstract

WebRTC is a web API for real-time communication on a peer-to-peer basis. It's implemented by browsers and made available for web developers through dedicated interfaces. Like all standards, the WebRTC standard changes over time to cover new use cases and remove unused or deprecated use cases. There are some moving parts involved in the way the API is used which one needs to be aware of before using the API. In this paper we will explore the state of WebRTC and some of its use cases.

Contents

List of Figures

List of Tables

Listings

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.

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 we can use the CanIUse [2] site.

1.2 Signaling Server

Although the WebRTC is a peer-to-peer communication API it can't fully function without a server. It needs a signaling server to resolve the connection between peers. After the peers have established a connection 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 Architecture

WebRTC uses different networking techniques to create a connection between peers. We will explore those techniques and technologies in this chapter.

2.1.1 Network Address Translation (NAT)

Devices in a network need a public IP address assigned so traffic from outside the network can be routed to the correct destination, this is done by using NAT. The router will translate requests from a source's private IP to to the routers 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)

STUN is used to find the public IP of a peer to which will be connected later on. It also can determine if there are any network restrictions in place which would prevent a connection, such as 'Symmetric NAT'.

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

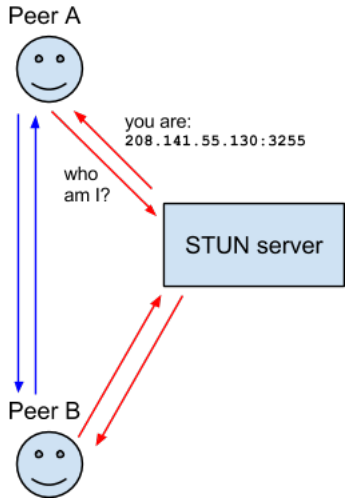

Figure 2.1: STUN communication schema, `https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Protocols`

Here are some public available example STUN servers:

- 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 connection then will relaying all information through that server.

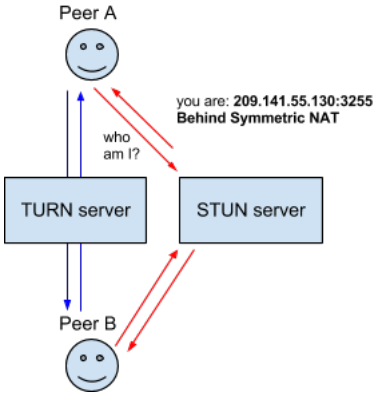

Figure 2.2: TURN communication schema, `https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Protocols`

There are open TURN servers available, for example provided by google. But this will mean all communication is going unprotected through 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 and its fallbacks.

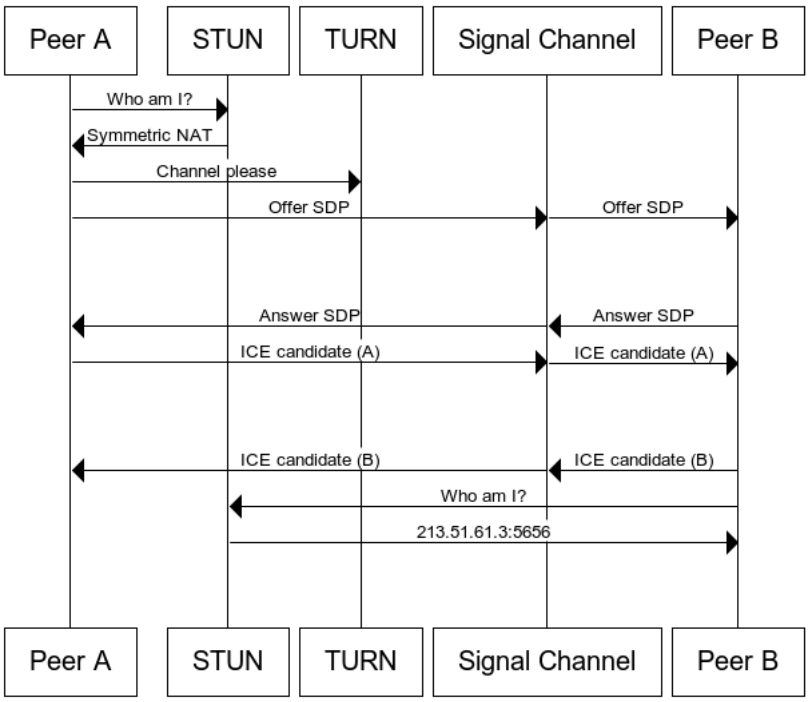

Figure 2.3: WebRTC Complete communication schema, `https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Connectivity`

2.2 Security

Generally WebRTC traffic is encrypted using Datagram Transport Layer Security (DTLS). Your data will be as secure as using any standard TLS 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. [4]

2.3 Basic Example Implementations

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 Client

This examples shows in a minimal way, how to create a WebRTC 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.

```
1  // Create connections
2  const sender = new RTCPeerConnection();
3  const receiver = new RTCPeerConnection();
4
5  // Set ICE candidate
6  sender.onicecandidate = e =>
7    !e.candidate || receiver.addIceCandidate(e.candidate);
8  receiver.onicecandidate = e =>
9    !e.candidate || sender.addIceCandidate(e.candidate);
10
11 // Create offer and set description
12 const offer = await sender.createOffer();
13 await sender.setLocalDescription(offer);
14 await receiver.setRemoteDescription(offer);
15
16 // Create answer from receiver and set description
17 const answer = await receiver.createAnswer();
18 await receiver.setLocalDescription(answer);
19 await sender.setRemoteDescription(answer);
```

Listing 2.1: Connect to a client

2.3.2 Disconnect Client

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

```
1  sender.close();
```

```
2 receiver.close();
```

Listing 2.2: Disconnect from a client

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.

```
1  let senderChannel;
2
3  async function send() {
4    // Send data
5    senderChannel.send(document.querySelector("#senderArea").value);
6  }
7
8  async function init() {
9    let sender = new RTCPeerConnection(null);
10   senderChannel = sender.createDataChannel("sendDataChannel");
11   let receiver = new RTCPeerConnection(null);
12
13   // listen on data received
14   receiver.ondatachannel = e => {
15     e.channel.onmessage = event => {
16       document.querySelector("#recieverArea").value = event.data;
17     };
18   };
19
20   const offer = await sender.createOffer();
21   sender.setLocalDescription(offer);
22   receiver.setRemoteDescription(offer);
23
24   const answer = await receiver.createAnswer();
25   receiver.setLocalDescription(answer);
26   sender.setRemoteDescription(answer);
27 }
```

Listing 2.3: Sending data

2.3.4 Video Chat

Client Media Check

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

```
1 function hasUserMedia() {
2   return !!(navigator.mediaDevices
3     && navigator.mediaDevices.getUserMedia);
4 }
```

Listing 2.4: Check client media

Accessing Client Media

In this example we see how client media, in this case video and audio, can be accessed. This action needs the users permission, which will be asked for by the browser automatically. In a real world example the application would need to handle the access denied case.

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

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

Listing 2.5: Access client media

Send Media

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

```
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
}
```

Listing 2.6: Send media

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.

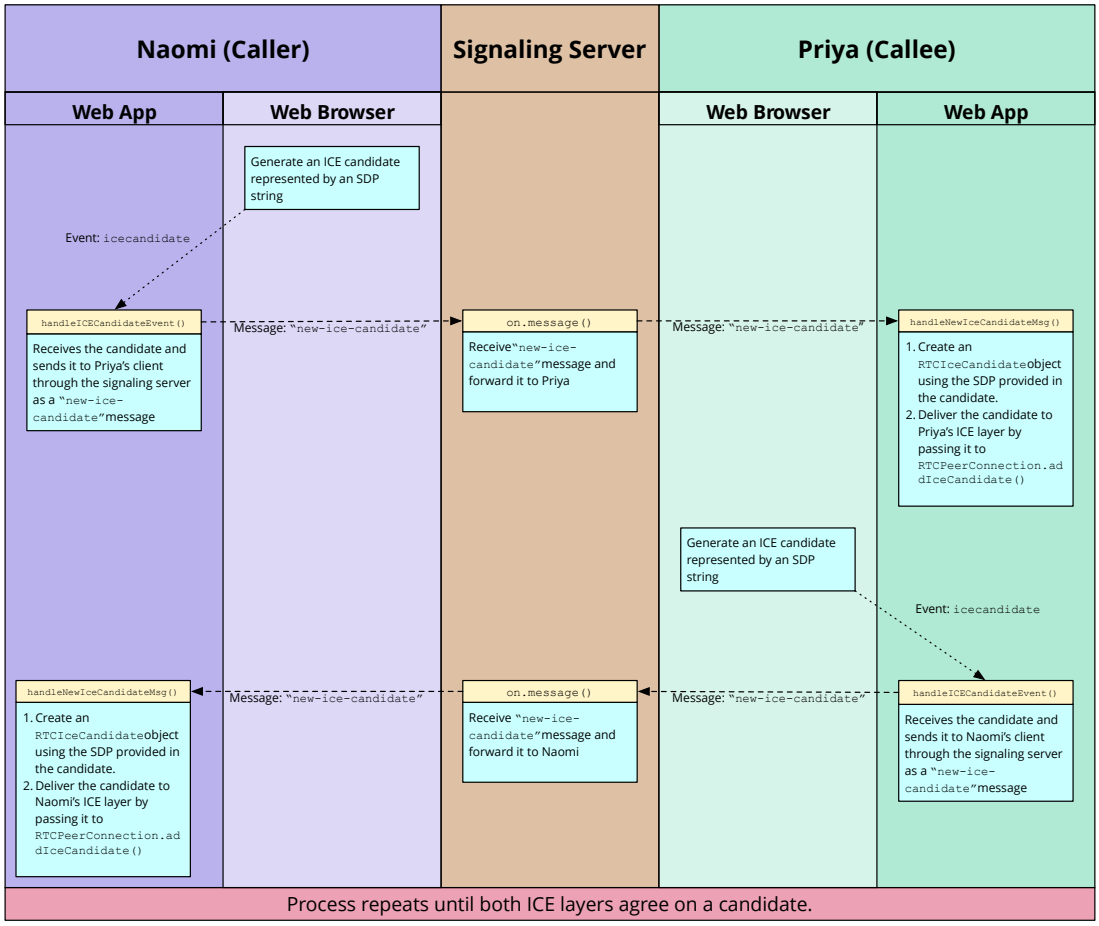

Figure 2.4: ICE communication schema, `https://mdn.mozillademos.org/files/12365/WebRTC-ICECandidateExchange.svg`

```
const servers = {
  iceServers: [{ urls: 'stun:stun.l.google.com:19302' }]
};

const connection = new RTCPeerConnection(servers);
connection.onicecandidate = e => {
  if (e.candidate) {
    // share candidate with the other peer
  }
};

connection.onmessage = receivedString => {
```

```
13    const message = JSON.parse(receivedString);
14    if (message.ice) {
15      connection.addIceCandidate(message.ice);
16    }
17  };
18
19  // create offer
20  const stream = await navigator.mediaDevices.getUserMedia({ audio: true })
        ;
21  stream.getTracks().forEach(track => connection.addTrack(track, stream));
22
23  const offer = await connection.createOffer({ offerToReceiveAudio: 1 });
24  connection.setLocalDescription(offer);
```

Listing 2.7: ICE Candidates

2.4 Signaling Server

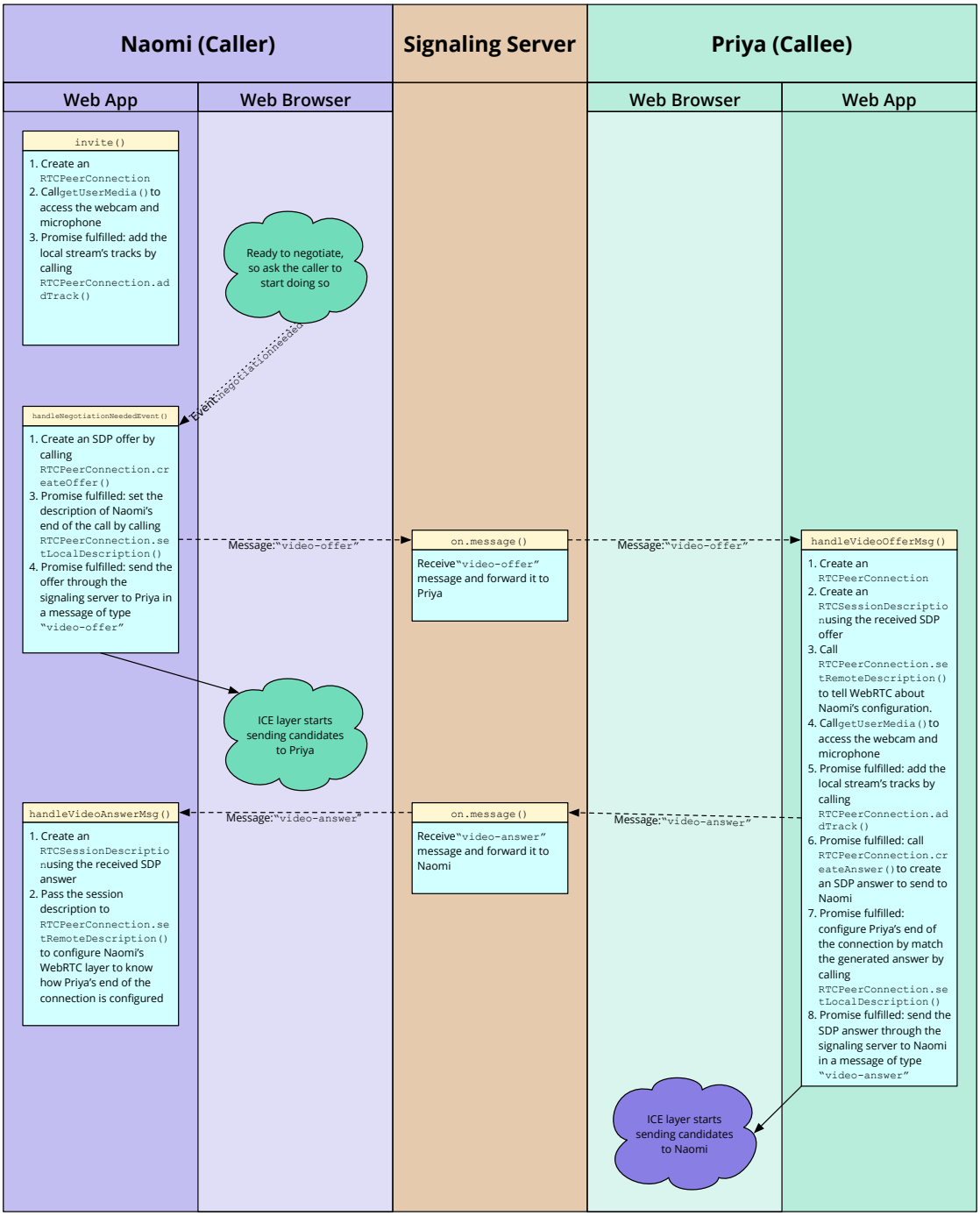

Figure 2.5: Signaling Diagram, `https://mdn.mozillademos.org/files/12363/WebRTC-SignalingDiagram.svg`

2.5 Applications of WebRTC

Application	Description	Pros and Cons	Security	Ownership
Jitsi	Jitsi is an open	Open Source	Good Security	Test

3 Conclusion

3.0.1 STUN / TURN Server

To not be dependent from open STUN/TURNTURN 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. Videobridge 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.

4 Outlook

5 Independence declaration

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

Index