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 22, 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.

WebSocket

Since WebRTC needs to establish a connection to a peer there is a need for a intermediate to create this connection. Often such a signaling server is using WebSocket since it provides a bidirectional communication.

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.

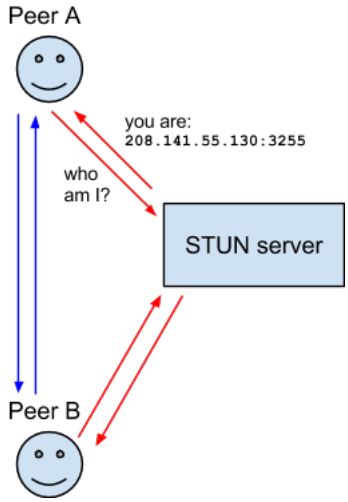

Figure 2.1: STUN communication schema, `https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Protocols`

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.

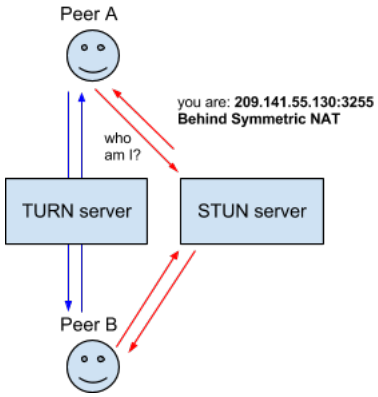

Figure 2.2: TURN communication schema, `https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Protocols`

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.

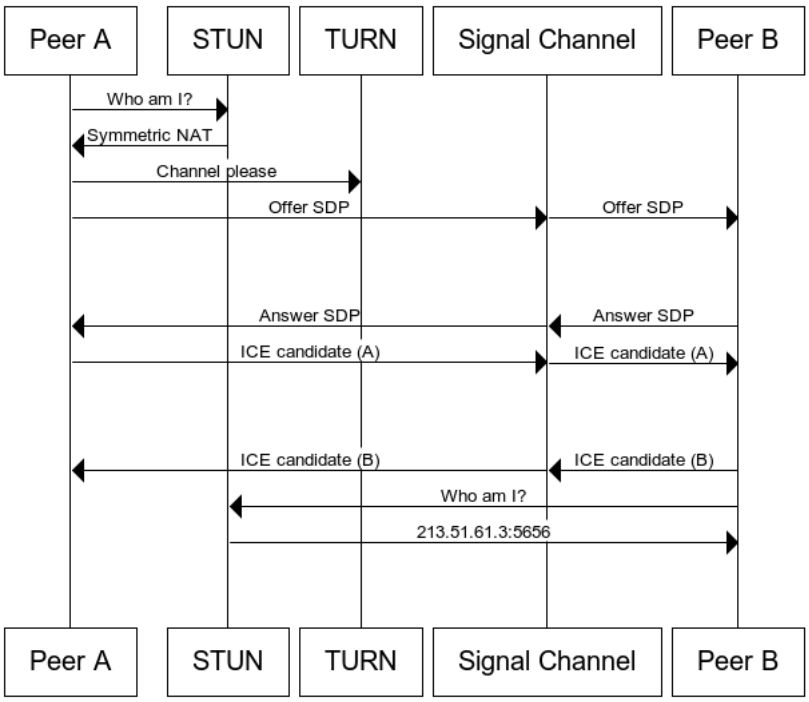

Figure 2.3: WebRTC Complete communication schema, `https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Connectivity`

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.

```
sender.close();
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.

```
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);
}
```

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.

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

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 find 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.

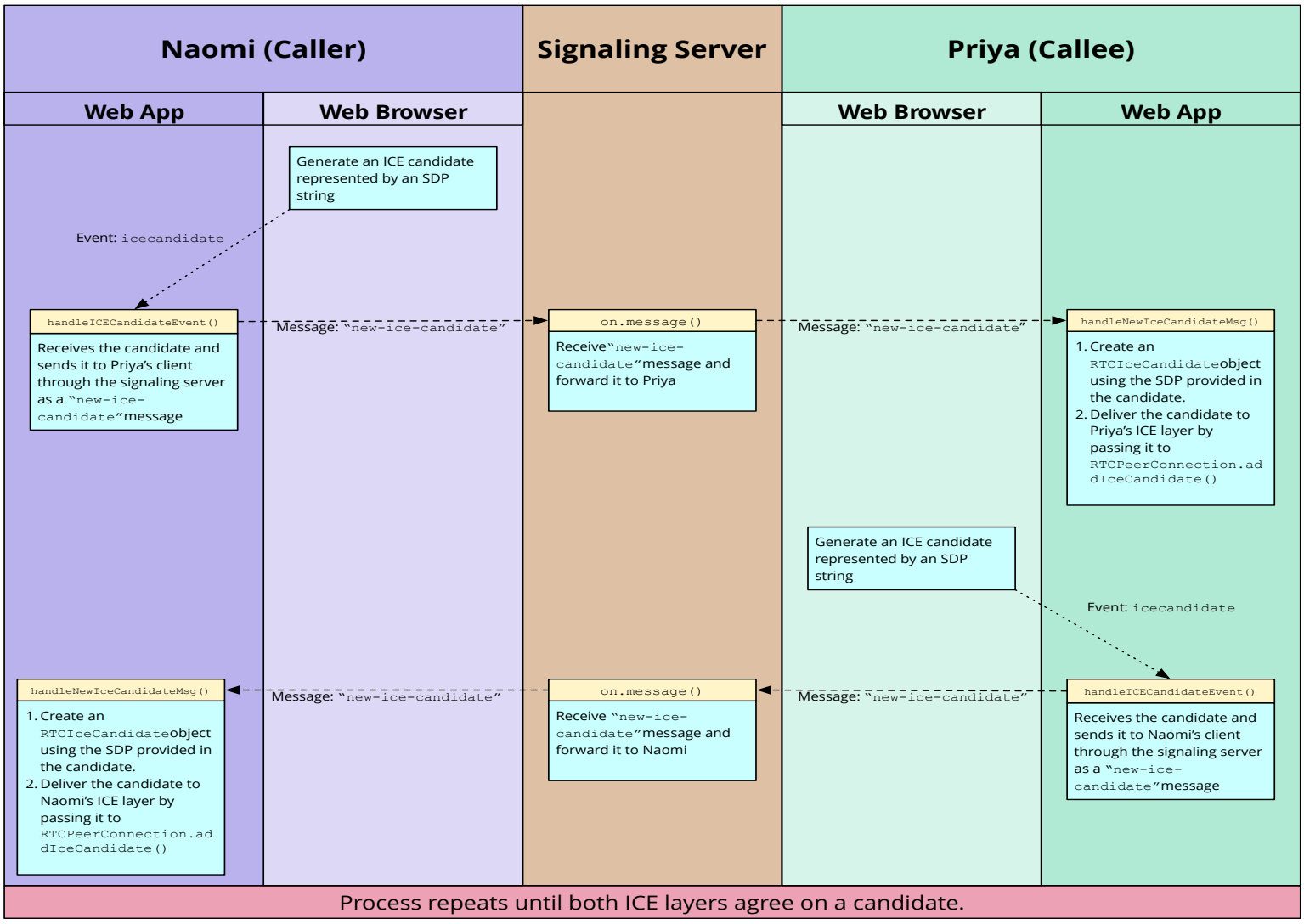

Figure 2.4: ICE communication schema, `https://mdn.mozillademos.org/files/12365/WebRTC-ICECandidateExchange.svg`

This is a short code example of such an ICE candidate exchange.

```
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 => {
  const message = JSON.parse(receivedString);
  if (message.ice) {
    connection.addIceCandidate(message.ice);
  }
};

// create offer
const stream = await navigator.mediaDevices.getUserMedia({ audio: true })
    ;
stream.getTracks().forEach(track => connection.addTrack(track, stream));

const offer = await connection.createOffer({ offerToReceiveAudio: 1 });
connection.setLocalDescription(offer);
```

Listing 2.7: ICE Candidates

2.4 Signaling Server

Although WebRTC is a peer-to-peer communication technology it requires some sort of signaling server to connect peers.

The following figure shows a complete communication for a video conference.

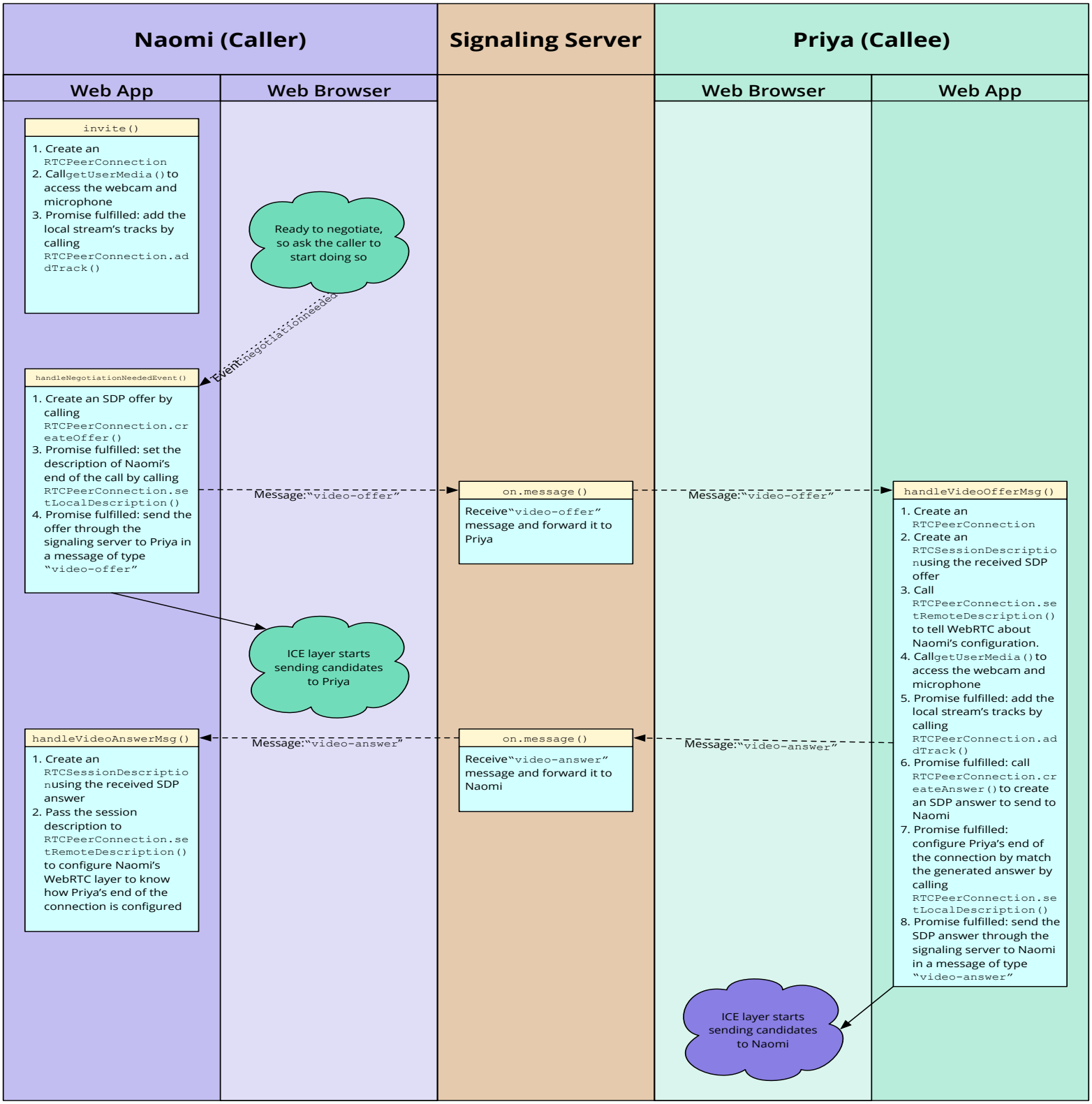

Figure 2.5: Signaling Diagram, `https://mdn.mozillademos.org/files/12363/WebRTC-SignalingDiagram.svg`

There are multiple signaling solutions available, but there are also frameworks available to create custom ones.

For a custom solutions it's handy to use WebSocket since it's creating a bidirectional connection between server and client. A handy solution for this use case would be a framework like socket.io [3] since it's a JavaScript framework it'll work the same on a node server and client.

2.5 Jitsi

Well known implementations of WebRTC are the free and open source software solutions from Jitsi. Jitsi provides multiple products for video conferencing, for example the Jitsi Videobridge which is a server application used to build scalable multiparty video applications or Jisti Meet which is a complete video conferencing solution.

There is a variety of conferencing solutions with or without WebRTC:

Source: Wikipedia [5]

2.5.1 Jitsi Meet

Program	Jitsi Meet
License	Apache License
Capacity	Unknown
Encrypted	✓
Pros	• Is fully free • It's open source • Focus on Web Application • Can be used without installation (https://meet.jit.si/)
Cons	• Depending on browser • Not all features are supported by major browsers

Table 2.1: Jitsi Meet

2.5.2 Apache OpenMeetings

Program	Apache OpenMeetings
License	Apache License
Capacity	1-125
Encrypted	✓
Pros	• Is fully free • It's open source
Cons	• Server needs to be provided • Low capacity

Table 2.2: Apache OpenMeetings

2.5.3 BigBlueButton

Program	BigBlueButton
License	LGPL
Capacity	1-100
Encrypted	✓
Pros	• Is fully free • It's open source • Is designed for teaching / online classrooms
Cons	• Low capacity • Is designed for teaching / online classrooms

Table 2.3: BigBlueButton

2.5.4 Zoom

Program	Zoom
License	Proprietary
Capacity	1000 videos participants & 49 videos on Screen
Encrypted	✓
Pros	• Can be used for free on a small scale
Cons	• Priced if used for larger scale • Proprietary

Table 2.4: Zoom

2.5.5 TeamViewer

Program	TeamViewer
License	Proprietary
Capacity	1-25
Encrypted	✓
Pros	• Focus on live support • Sophisticated controls of viewed source
Cons	• Very low capacity • Proprietary • Not primarily designed as conferencing tool

Table 2.5: TeamViewer

2.5.6 Skype

Program	Skype
License	Proprietary
Capacity	1-25
Encrypted	✓
Pros	• Focus on telephony
Cons	• Is not fully free • Proprietary • Very low capacity

Table 2.6: Skype

2.5.7 Microsoft Teams

Program	Microsoft Teams
License	Proprietary
Capacity	1-25
Encrypted	✓
Pros	• Integration with Microsoft collaboration tools (SharePoint) • Strongly supports chats • Good live collaboration integration
Cons	• Not primarily designed as conferencing tool • Proprietary

Table 2.7: Microsoft Teams

2.5.8 Google Meet

Program	Google Meet
License	Proprietary
Capacity	1-10
Encrypted	✓
Pros	• Integration with other google services
Cons	• Proprietary • Not fully free

Table 2.8: Google Meet

2.5.9 StarLeaf

Program	StarLeaf
License	Proprietary
Capacity	1-25
Encrypted	✓
Pros	• Integration with business applications (Outlook, Slack) • Provides API's for custom integrations
Cons	• Proprietary • Not fully free

Table 2.9: StarLeaf

3 Conclusion

The development of WebRTC has proceeded very well, to a point where one can create complete conferencing tools solely on a browser without the need of a installed software for the client. The example of Jitsi Meet impressively showcases the abilities of such browser only solutions. The common browsers not only implements WebRTC in a way that it can used fairly simple, but also that the accommodating API's and technologies are in a well developed state.

One pain point left is the need of a signaling server. WebRTC still needs a signaling server to find peers. There are open STUN and TURN servers but it's rather easy to install FOSS solutions or use existing libraries.

There are still limitations to what can de done on a peer-to-peer basis, especially with video conferencing. The limits of a browser or a network can be reached quite easily with video data in a high quality. There are solutions to fix that issue, like the Jitsi Videobridge, but they require even more servers rather then just a STUN/TURN server.

4 Outlook

In a further project one could take a look on a more complete solution. How would the architecture of a fully working video conferencing application look, how's the signaling implemented, for example with socket.io [3], what are we using as STUN/TURN servers etc.. I think it would be quite interesting to see a complete solution to a conferencing tool.

5 Statement of Authorship

I hereby declare that I am the sole author of this work and that I have not used any sources other than those listed in the bibliography and identified as references.

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] *Socket.io*. URL: https://socket.io/ (visited on 03/21/2020).

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

[5] *Wikipedia: Comparison of web conferencing software*. URL: https://en.wikipedia.org/wiki/Comparison_of_web_conferencing_software (visited on 03/22/2020).

Index