WebRTC

tutorialspoint

SIMPLYEASYLEARNING

www.tutorialspoint.com

https://www.facebook.com/tutorialspointindia

https://twitter.com/tutorialspoint

About the Tutorial

With Web Real-Time Communication (WebRTC), modern web applications can easily stream audio and video content to millions of people. In this tutorial, we would explain how you can use WebRTC to set up peer-to-peer connections to other web browsers quickly and easily.

Audience

This tutorial is going to help all those developers who would like to learn how to build applications such as real-time advertising, multiplayer games, live broadcasting, e-learning, to name a few, where the action takes place in real time.

Prerequisites

WebRTC is a powerful tool that can be used to infuse Real-Time Communications (RTC) capabilities into browsers and mobile applications. This tutorial covers only the basics of WebRTC and any regular developer with some level of exposure to real-time session management can easily grasp the concepts discussed here.

Disclaimer & Copyright

© Copyright 2015 by Tutorials Point (I) Pvt. Ltd.

All the content and graphics published in this e-book are the property of Tutorials Point (I) Pvt. Ltd. The user of this e-book is prohibited to reuse, retain, copy, distribute, or republish any contents or a part of contents of this e-book in any manner without written consent of the publisher.

We strive to update the contents of our website and tutorials as timely and as precisely as possible, however, the contents may contain inaccuracies or errors. Tutorials Point (I) Pvt. Ltd. provides no guarantee regarding the accuracy, timeliness, or completeness of our website or its contents including this tutorial. If you discover any errors on our website or in this tutorial, please notify us at contact@tutorialspoint.com

Table of Contents

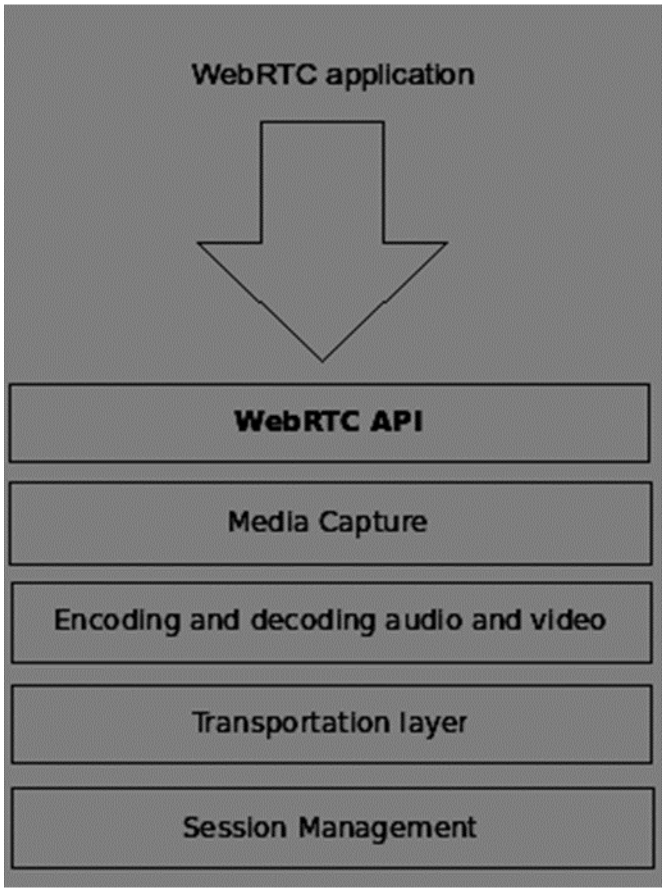
1. Web RTC – Overview

The Web is no more a stranger to real-time communication as **WebRTC (Web Real-Time Communication)** comes into play. Although it was released in May 2011, it is still developing and its standards are changing. A set of protocols is standardized by *Real-Time Communication in WEB-browsers Working group* at http://tools.ietf.org/wg/rtcweb/ of the **IETF (Internet Engineering Task Force)** while new sets of APIs are standardized by the *Web Real-Time Communications Working Groupe* at http://www.w3.org/2011/04/webrtc/ of the **W3C (World Wide Web Consortium)**. With the appearance of WebRTC, modern web applications can easily stream audio and video content to millions of people.

Basic Scheme

WebRTC allows you to set up peer-to-peer connections to other web browsers quickly and easily. To build such an application from scratch, you would need a wealth of frameworks and libraries dealing with typical issues like data loss, connection dropping, and NAT traversal. With WebRTC, all of this comes built-in into the browser out-of-the-box. This technology doesn't need any plugins or third-party software. It is open-sourced and its source code is freely available at http://www.webrtc.org/.

The WebRTC API includes media capture, encoding and decoding audio and video, transportation layer, and session management.



Media Capture

The first step is to get access to the camera and microphone of the user's device. We detect the type of devices available, get user permission to access these devices and manage the stream.

Encoding and Decoding Audio and Video

It is not an easy task to send a stream of audio and video data over the Internet. This is where encoding and decoding are used. This is the process of splitting up video frames and audio waves into smaller chunks and compressing them. This algorithm is called **codec**. There is an enormous amount of different codecs, which are maintained by different companies with different business goals. There are also many codecs inside WebRTC like H.264, iSAC, Opus and VP8. When two browsers connect together, they choose the most optimal supported codec between two users. Fortunately, WebRTC does most of the encoding behind the scenes.

Transportation Layer

The transportation layer manages the order of packets, deal with packet loss and connecting to other users. Again the WebRTC API gives us an easy access to events that tell us when there are issues with the connection.

Session Management

The session management deals with managing, opening and organizing connections. This is commonly called **signaling.** If you transfer audio and video streams to the user it also makes sense to transfer collateral data. This is done by the **RTCDataChannel** API.

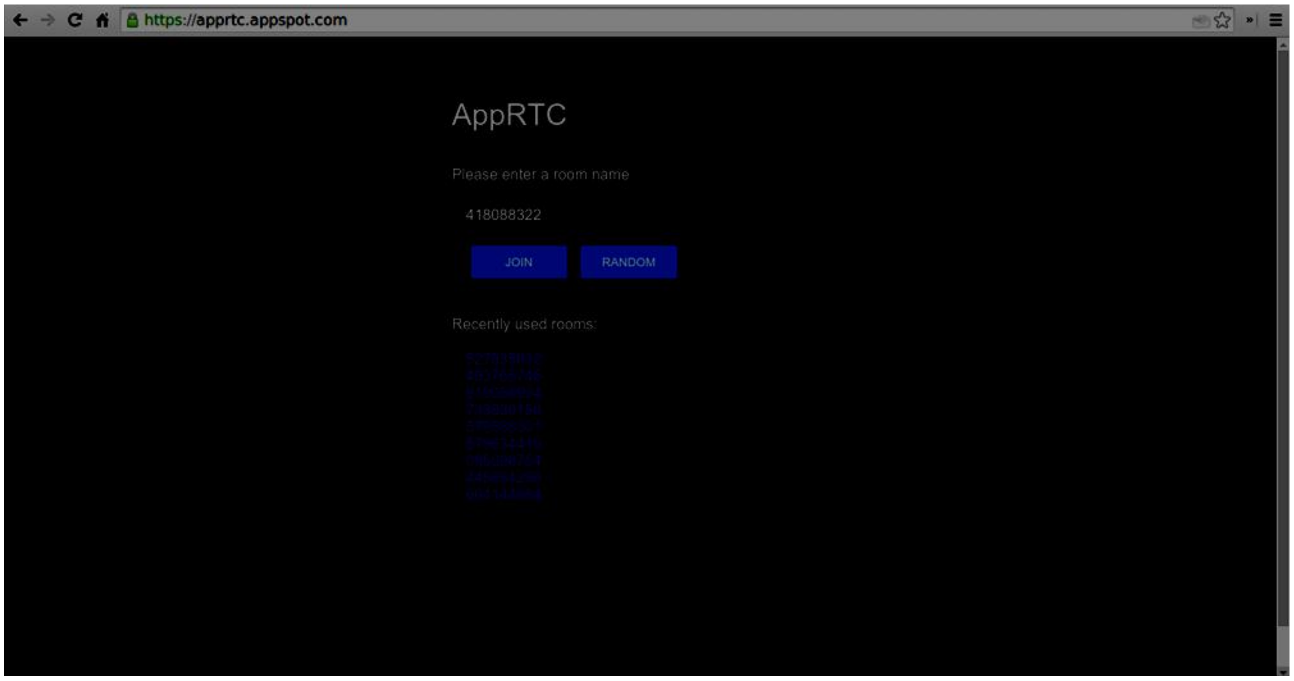
Engineers from companies like Google, Mozilla, Opera and others have done a great job to bring this real-time experience to the Web.

Browser Compatibility

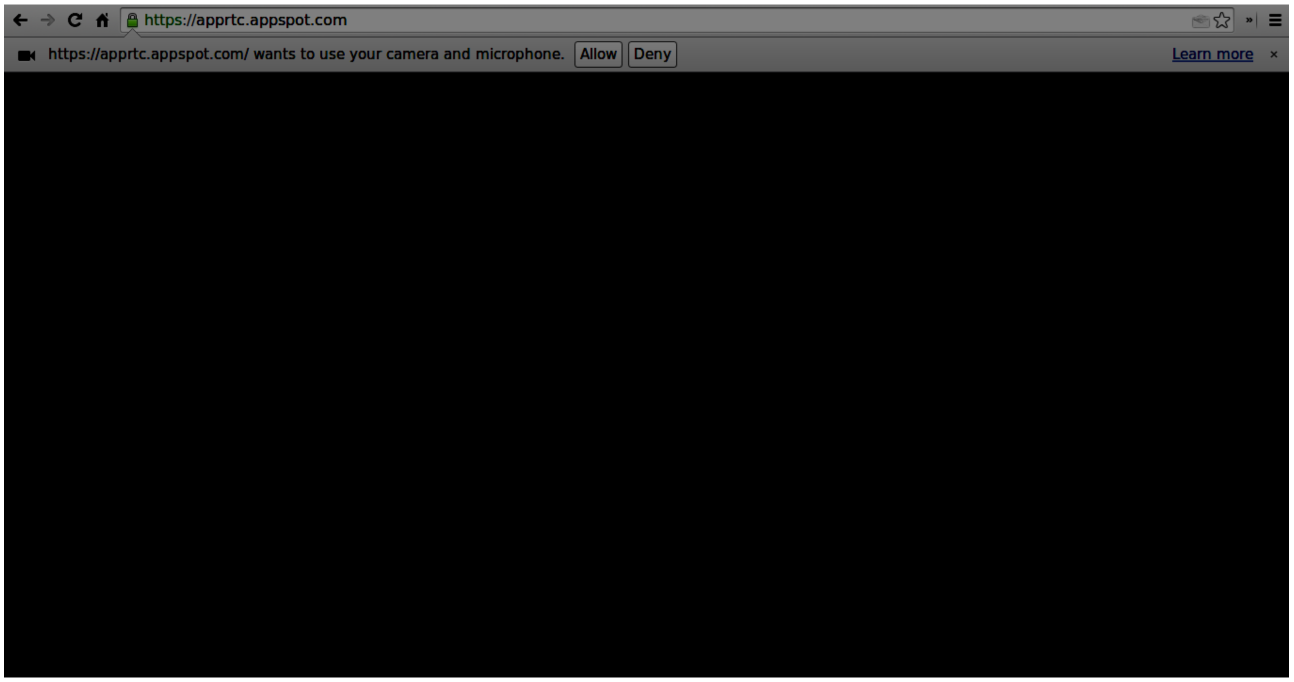
The WebRTC standards are one of the fastest evolving on the web, so it doesn't mean that every browser supports all the same features at the same time. To check whether your browser supports WebRTC or not, you may visit http://caniuse.com/#feat=rtcpeerconnection. Throughout all the tutorials, I recommend you to use Chrome for all the examples.

Trying out WebRTC

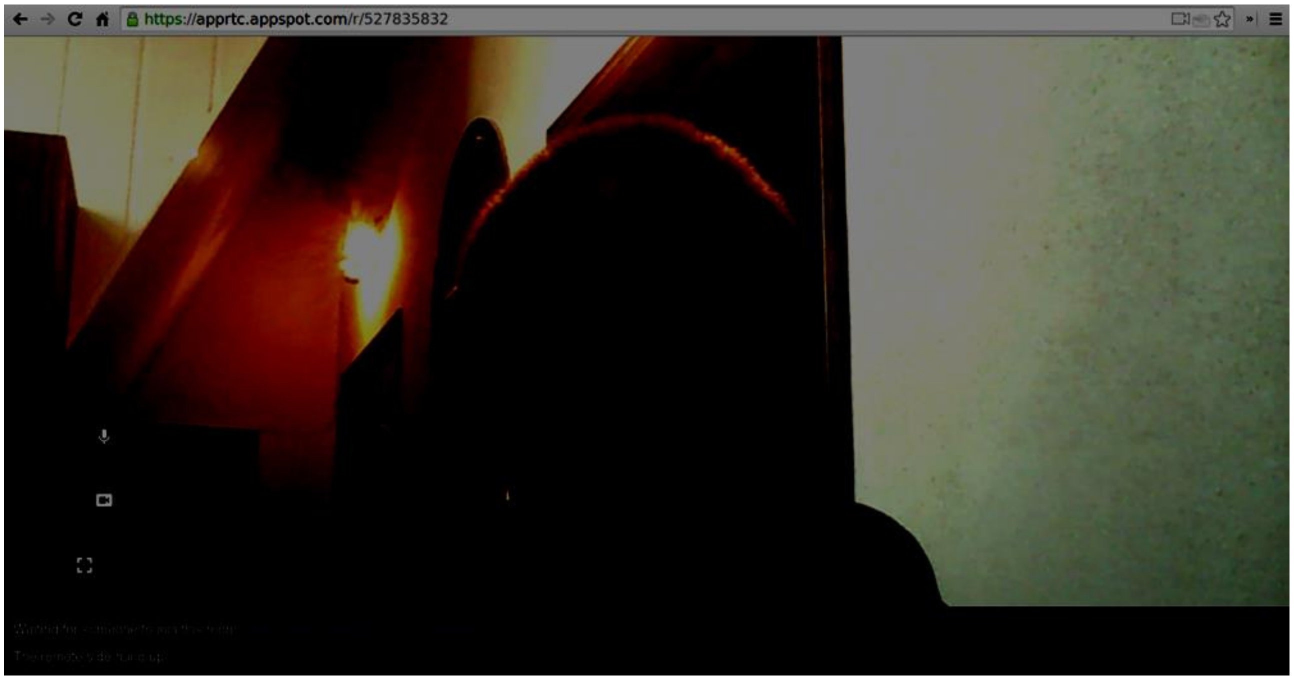
Let's get started using WebRTC right now. Navigate your browser to the demo site at https://apprtc.appspot.com/



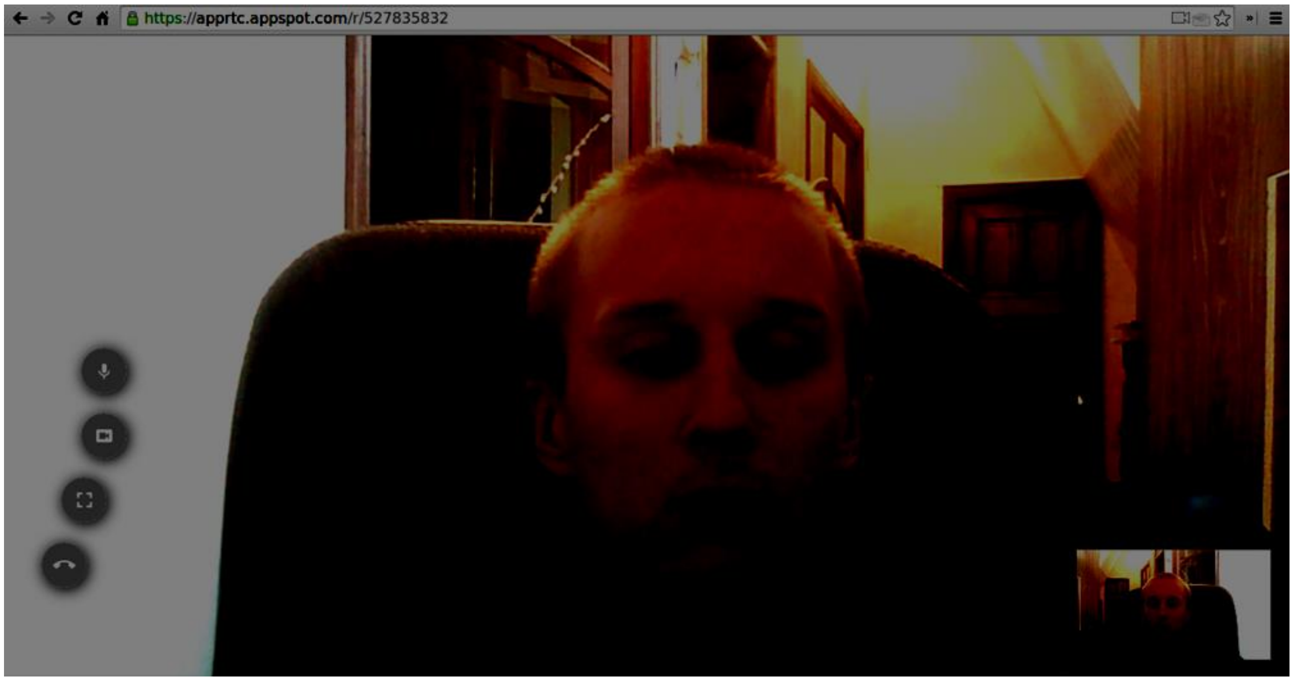
Click the "JOIN" button. You should see a drop-down notification.


https://apprtc.appspot.com
https://apprtc.appspot.com/ wants to use your camera and microphone.
Allow
Deny
Learn more

Click the "Allow" button to start streaming your video and audio to the web page. You should see a video stream of yourself.



Now open the URL you are currently on in a new browser tab and click on "JOIN". You should see two video streams – one from your first client and another from the second one.



Now you should understand why WebRTC is a powerful tool.

Use Cases

The real-time web opens the door to a whole new range of applications, including text-based chat, screen and file sharing, gaming, video chat, and more. Besides communication you can use WebRTC for other purposes like:

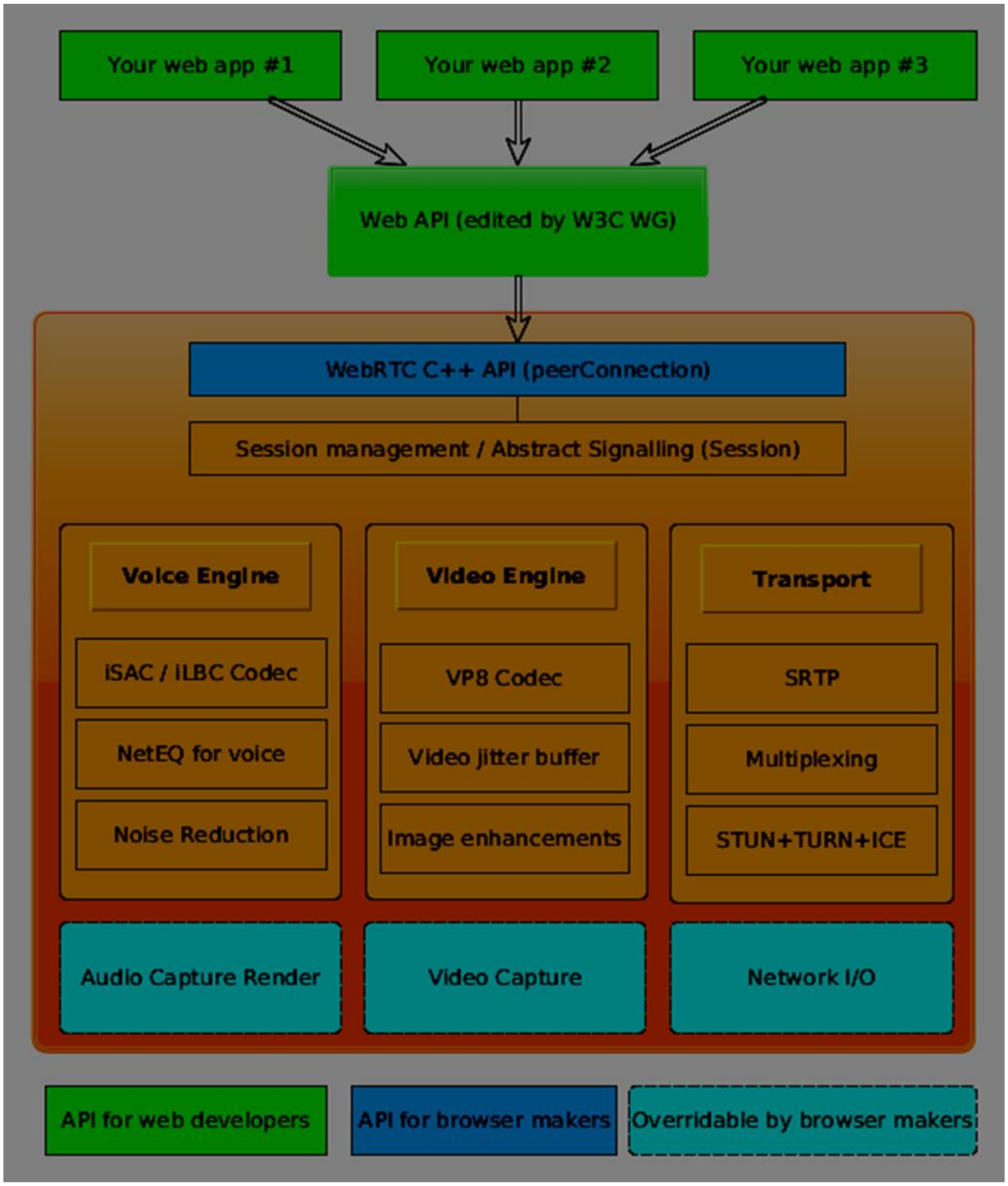
- real-time marketing
- real-time advertising
- back office communications (CRM, ERP, SCM, FFM)
- HR management
- social networking
- dating services
- online medical consultations
- financial services
- surveillance
- multiplayer games
- live broadcasting
- e-learning

Summary

Now you should have a clear understanding of the term WebRTC. You should also have an idea of what types of applications can be built with WebRTC, as you have already tried it in your browser. To sum up, WebRTC is quite a useful technology.

2. WebRTC – Architecture

The overall WebRTC architecture has a great level of complexity.



Here you can find three different layers:

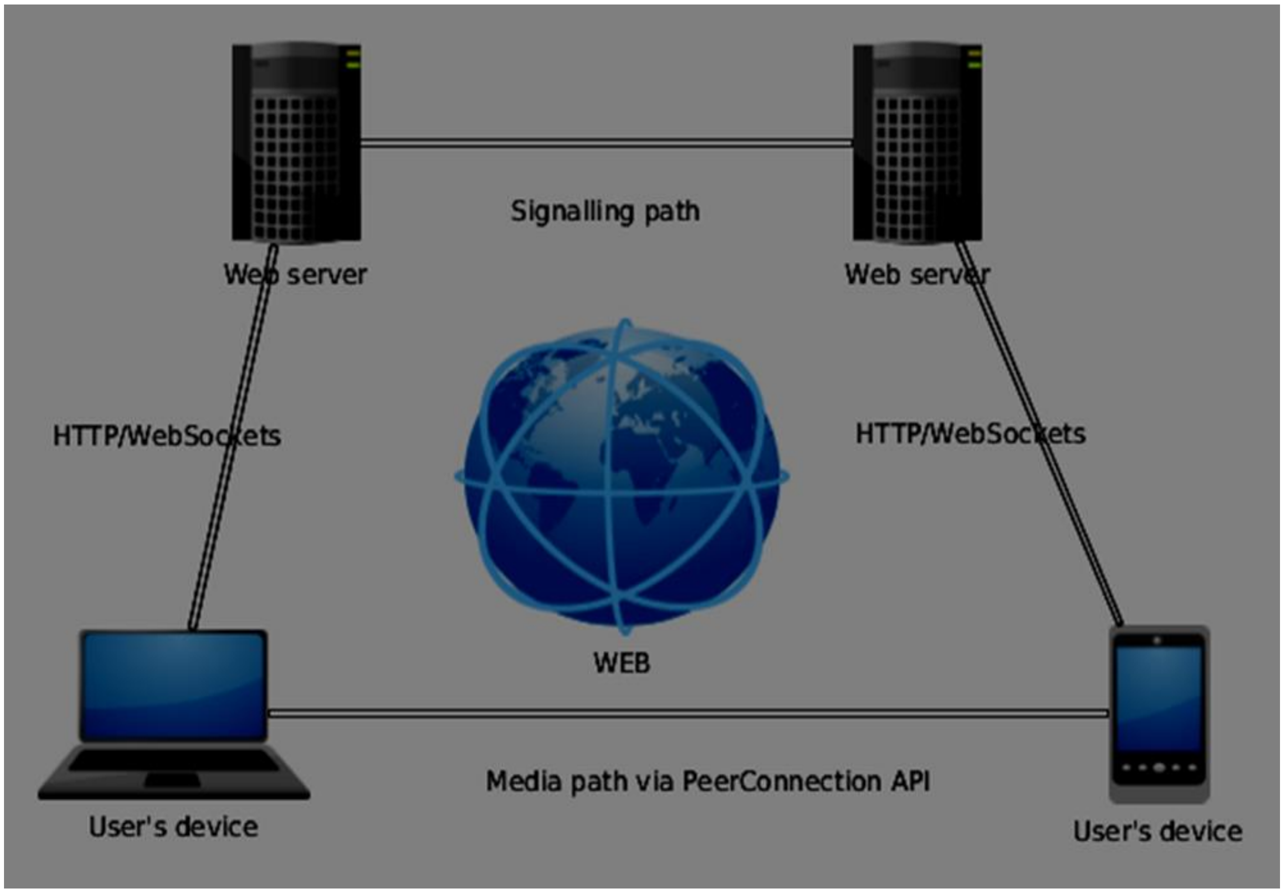
- API for web developers – this layer contains all the APIs web developer needed, including RTCPeerConnection, RTCDataChannel, and MediaStrean objects.

- API for browser makers

- Overridable API, which browser makers can hook.

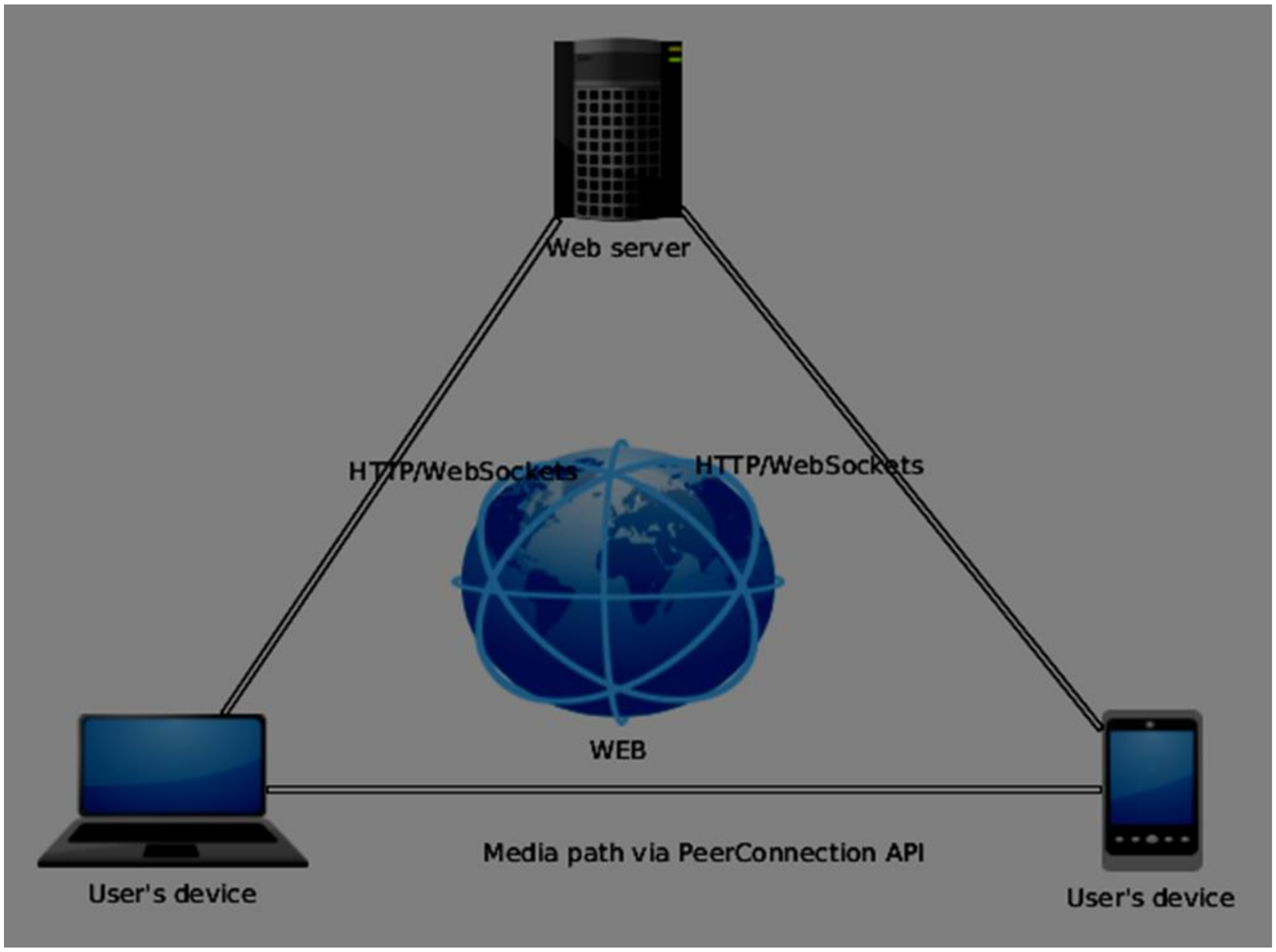
Transport components allow establishing connections across various types of networks while voice and video engines are frameworks responsible for transferring audio and video streams from a sound card and camera to the network. For Web developers, the most important part is WebRTC API.

If we look at the WebRTC architecture from the client-server side we can see that one of the most commonly used models is inspired by the SIP(Session Initiation Protocol) Trapezoid.



In this model, both devices are running a web application from different servers. The RTCPeerConnection object configures streams so they could connect to each other, peer-to-peer. This signaling is done via HTTP or WebSockets.

But the most commonly used model is Triangle:



In this model both devices use the same web application. It gives web developer more flexibility when managing user connections.

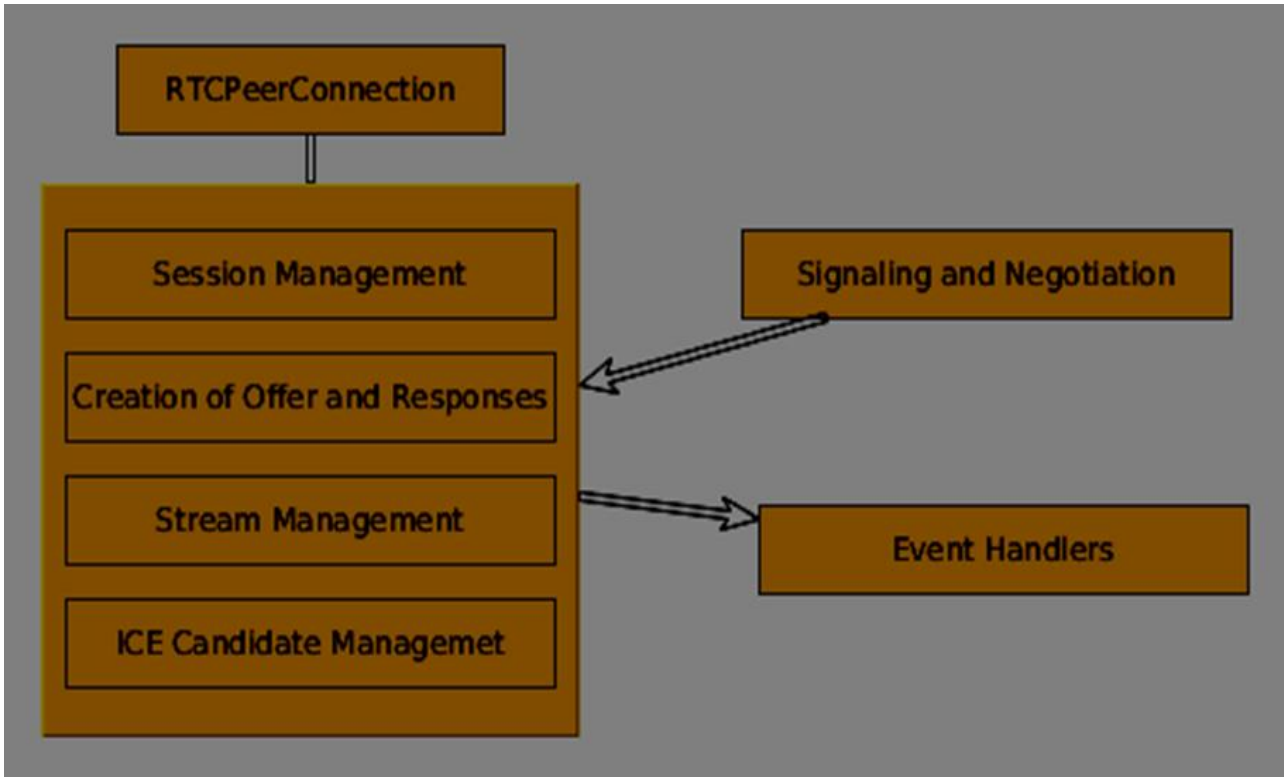
The WebRTC API

It consists of a few main javascript objects: RTCPeerConnection, MediaStream, and RTCDataChannel.

The RTCPeerConnection object

This object is the main entry point to the WebRTC API. It helps us connect to peers, initialize connections and attach media streams. It also manages a UDP connection with another user.

The main task of the RTCPeerConnection object is to setup and create a peer connection. We can easily hook keys points of the connection because this object fires a set of events when they appear. These events give you access to the configuration of our connection:



The RTCPeerConnection is a simple javascript object, which you can simply create this way:

```
[code]
var conn = new RTCPeerConnection(conf);
conn.onaddstream = function(stream){
    // use stream here
};
[/code]
```

The RTCPeerConnection object accepts a *conf* parameter, which we will cover later in these tutorials. The *onaddstream* event is fired when the remote user adds a video or audio stream to their peer connection.

MediaStream API

Modern browsers give a developer access to the *getUserMedia* API, also known as the *MediaStream* API. There are three key points of functionality:

- It gives a developer access to a *stream* object that represent video and audio streams

- It manages the selection of input user devices in case a user has multiple cameras or microphones on his device
- It provides a security level asking user all the time he wants to fetch s stream

To test this API let's create a simple HTML page. It will show a single <video> element, ask the user's permission to use the camera and show a live stream from the camera on the page. Create an *index.html* file and add:

```
[code]
<html>
    <head>
        <meta charset=”utf-8”>
    </head>
    <body>
        <video autoplay></video>
        <script src=”client.js”></script>
    </body>
</html>
[/code]
```

Then add a *client.js* file:

```
[code]
//checks if the browser supports WebRTC
function hasUserMedia(){
    navigator.getUserMedia = navigator.getUserMedia ||
navigator.webkitGetUserMedia
        || navigator.mozGetUserMedia || navigator.msGetUserMedia;
    return !!navigator.getUserMedia;
}
if (hasUserMedia()) {
  navigator.getUserMedia = navigator.getUserMedia ||
navigator.webkitGetUserMedia || navigator.mozGetUserMedia ||
navigator.msGetUserMedia;
  //get both video and audio streams from user's camera
  navigator.getUserMedia({ video: true, audio: true }, function (stream) {
    var video = document.querySelector('video');
    //insert stream into the video tag
    video.src = window.URL.createObjectURL(stream);
  }, function (err) {});
} else {
  alert("Error. WebRTC is not supported!");
}
[/code]
```

Now open the *index.html* and you should see the video stream displaying your face.

But be careful, because WebRTC works only on the server side. If you simply open this page with the browser it won't work. You need to host these files on the Apache or Node servers, or which one you prefer.

The RTCDataChannel object

As well as sending media streams between peers, you may also send additional data using *DataChannel* API. This API is as simple as MediaStream API. The main job is to create a channel coming from an existing RTCPeerConnection object:

```
[code]
var peerConn = new RTCPeerConnection();
//establishing peer connection
//...
//end of establishing peer connection
var dataChannel = peerConnection.createDataChannel("myChannel",
dataChannelOptions);
// here we can start sending direct messages to another peer
[/code]
```

This is all you needed, just two lines of code. Everything else is done on the browser's internal layer. You can create a channel at any peer connection until the *RTCPeerConnectionobject* is closed.

Summary

You should now have a firm grasp of the WebRTC architecture. We also covered MediaStream, RTCPeerConnection, and RTCDataChannel APIs. The WebRTC API is a moving target, so always keep up with the latest specifications.

3. WebRTC – Environment

Before we start building our WebRTC applications, we should set our coding environment. First of all, you should have a text editor or IDE where you can edit HTML and Javascript. There are chances that you have already chosen the preferred one as you are reading this tutorial. As for me, I'm using WebStorm IDE. You can download its trial version at https://www.jetbrains.com/webstorm/. I'm also using Linux Mint as my OS of choice.

The other requirement for common WebRTC applications is having a server to host the HTML and Javascript files. The code will not work just by double-clicking on the files because the browser is not allowed to connect to cameras and microphones unless the files are being served by an actual server. This is done obviously due to the security issues.

There are tons of different web servers, but in this tutorial, we are going to use Node.js with node-static.:

1. Visit https://nodejs.org/en/ and download the latest Node.js version.

2. Unpack it to the /usr/local/nodejs directory.

3. Open the /home/YOUR_USERNAME/.profile file and add the following line to the end: *export PATH=$PATH:/usr/local/nodejs/bin*

4. The you can restart your computer or run *source /home/YOUR_USERNAME/.profile*

5. Now the *node* command should be available from the command line. The *npm* command is also available. NMP is the package manager for Node.js. You can learn more at https://www.npmjs.com/ .

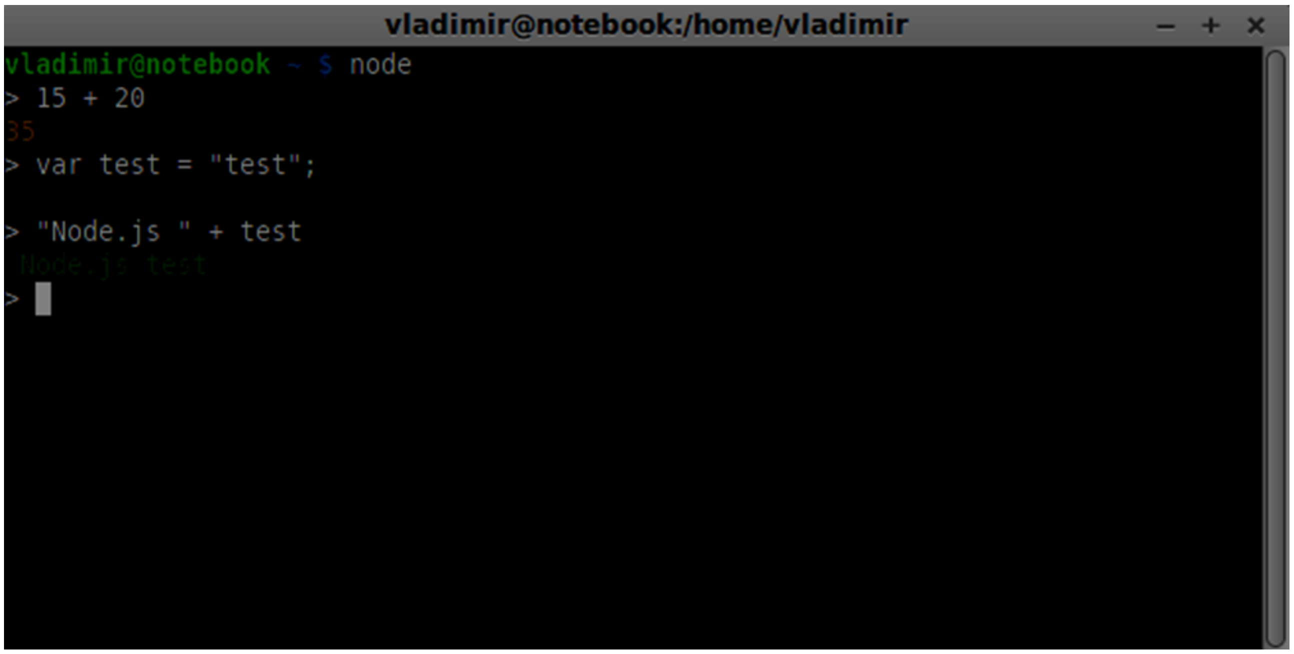
6. Open up a terminal and run *sudo npm install -g node-static.* This will install the static web server for Node.js.

7. Now navigate to any directory containing the HTML files and run the *static* command inside the directory to start your web server.

8. You can navigate to http://localhost:8080 to see your files.

There is another way to install nodejs. Just run *sudo apt-get install nodejs* in the terminal window.

To test your Node.js installation open up your terminal and run the *node* command. Type a few commands to check how it works:



vladimir@notebook:/home/vladimir

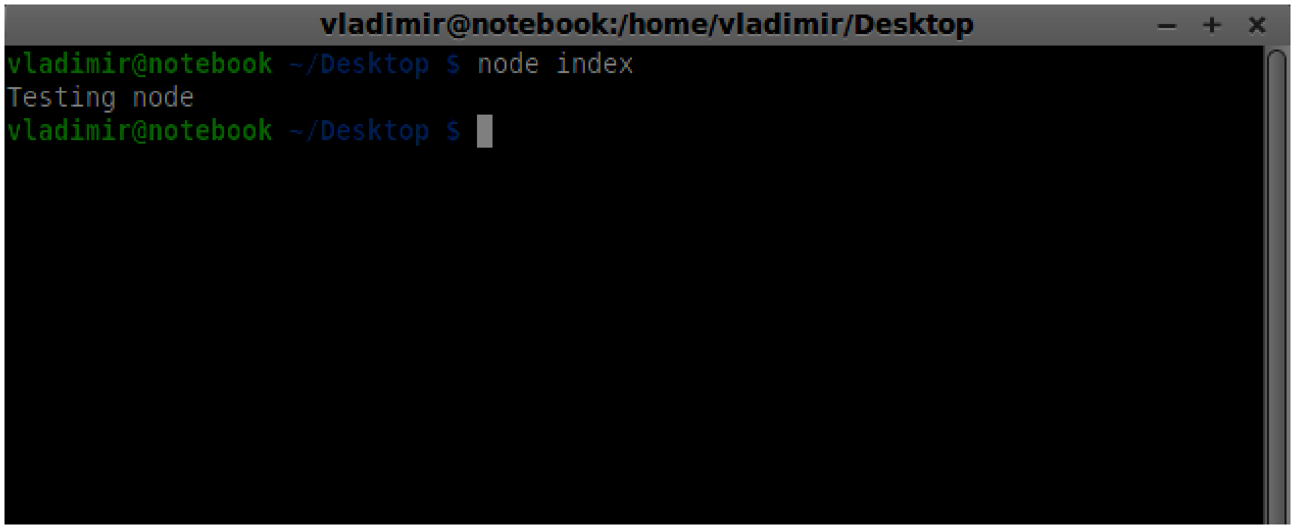
```
vladimir@notebook ~ $ node
> 15 + 20
35
> var test = "test";

> "Node.js " + test
 Node.js test
> 
```

Node.js runs Javascript files as well as commands typed in the terminal. Create an *index.js* file with the following content:

```
console.log("Testing Node.js");
```

Then run the *node index* command. You will see the following:



vladimir@notebook:/home/vladimir/Desktop

```
vladimir@notebook ~/Desktop $ node index
Testing node
vladimir@notebook ~/Desktop $ 
```

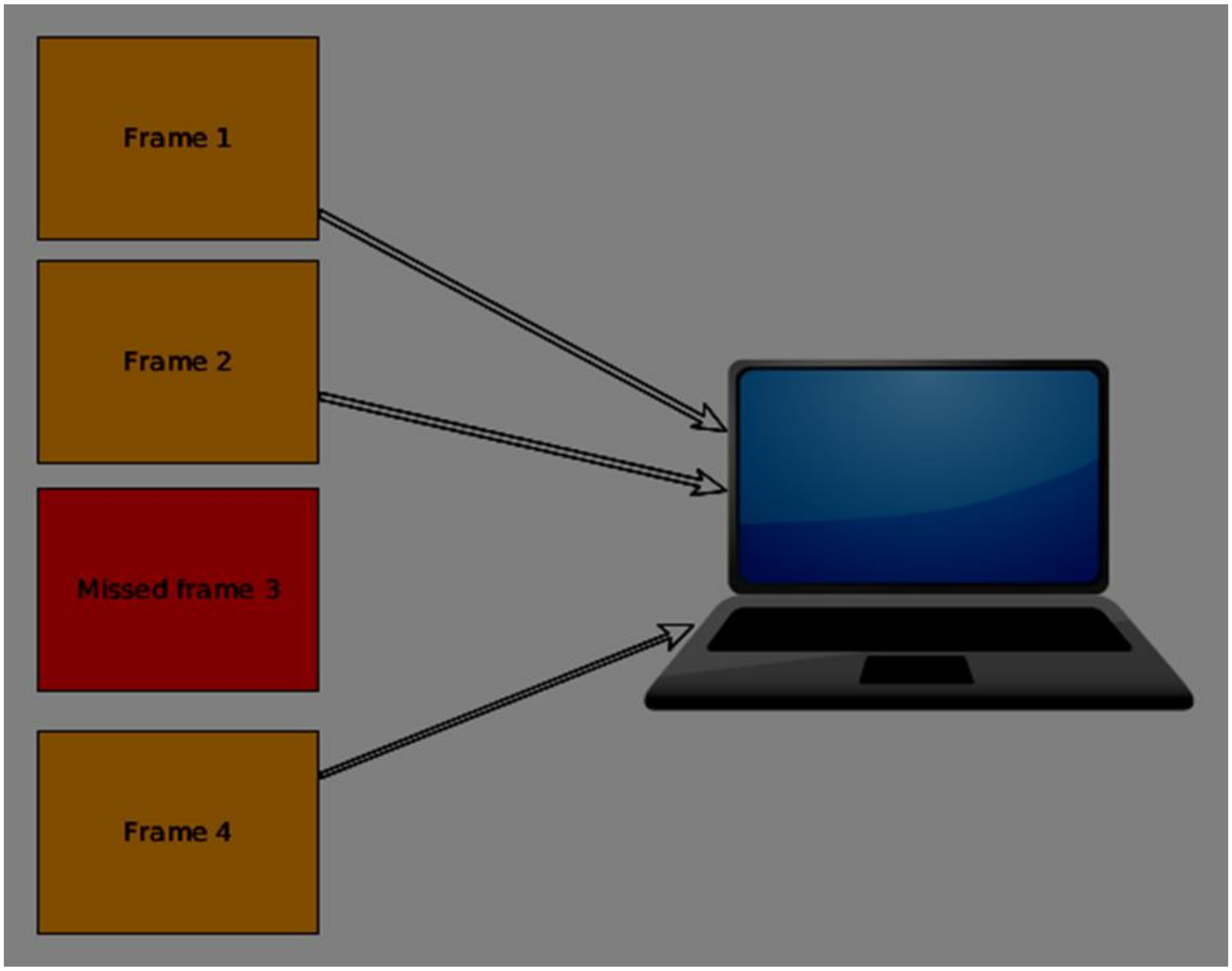
When building our signaling server we will use a WebSockets library for Node.js . To install in run *npm install ws* in the terminal.

For testing our signaling server, we will use the wscat utility. To install it run *npm install -g wscat* in your terminal window.

WebRTC Protocols

Real-time data communication means a fast connection speed between both user's devices. A common connection takes a frame of video or audio and transfers it to another user's device between 30 and 60 times per second in order to achieve a good quality. So it is important to understand that sending the latest frame of data is more crucial than making sure that every single frame gets to the other side. That is why WebRTC applications may miss certain frames in order to keep a good speed of the connection.

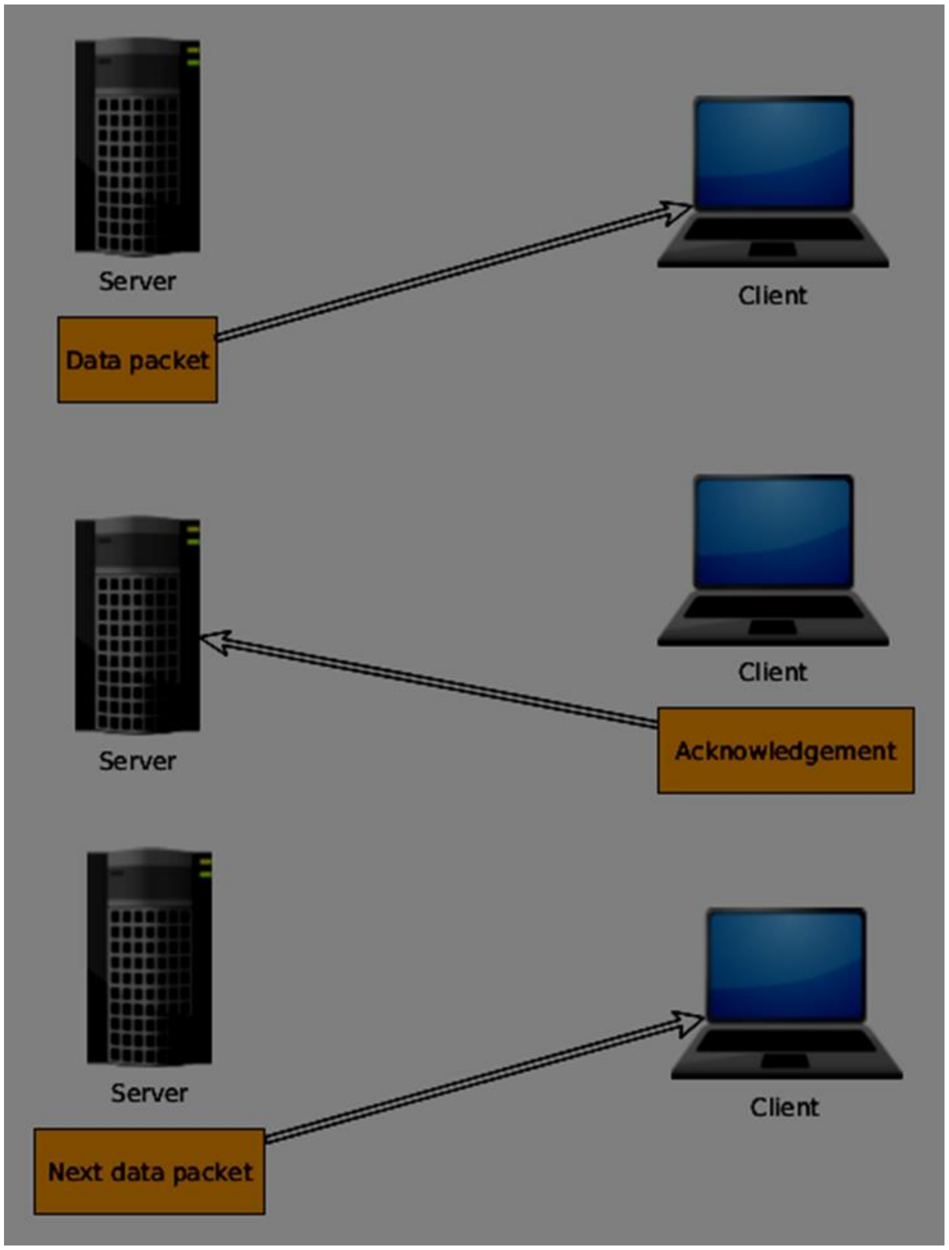
You may see this effect almost in any video-playing application nowadays. Video games and video streaming apps can afford to lose a few frames of video because our mind try to fill these spaces as we always visualize what we are watching. If we want our application to play 50 frames in one second and we miss frames 15, 25, and 38, most of the time, the user won't event notice it. So for video streaming applications there is a different set of requirements:



This is why WebRTC applications use UDP (User Datagram Protocol) as the transport protocol. Most web applications today are built with the using of the TCP (Transmission Control Protocol) because it guarantees that:

- any data sent will be marked as received
- any data that does not get to the other side will be resent and sending of other data will be temporarily terminated
- any data will be unique without duplicates on the other side

You may see why TCP is a great choice for most web applications today. If you are requesting an HTML page, it makes sense to get all the data in the right order. But this technology can not fit for all use cases. If we take, for example, a multiplayer game, the user will be able to only see what has happened in the last few seconds and nothing more which may lead to a large bottleneck when the data is missing:



The audio and video WebRTC connection is not mean to be the most reliable, but rather to be the fastest between two user's devices. So we can afford losing frames, which means that UDP is the best choice for audio and video streaming applications.

UDP was built to be a less reliable transport layer. You can not be sure in:

- the order of your data
- the delivery status of your data
- the state of every single data packet

Nowadays, WebRTC sends media packets in the fastest way possible. WebRTC can be a complex topic when concerning large corporate networks. Their firewalls can block UDP traffic across them. A lot of work have been done to make UDP work properly for wide audience.

Most Internet traffic today is built on TCP and UDP, not only web pages. You can find them in tablets, mobile devices, Smart TVs, and more. So it is important to understand how these technologies work.

The Session Description Protocol

The SDP is an important part of the WebRTC. It is a protocol that is intended to describe media communication sessions. It does not deliver the media data but is used for negotiation between peers of various audio and video codecs, network topologies, and other device information. It also needs to be easily transportable. Simply put we need a string-based profile with all the information about the user's device. This is where SDP comes in.

The SDP is a well-known method of establishing media connections as it appeared in the late 90s. It has been used in a vast amount of other types of applications before WebRTC like phone and text-based chatting.

The SDP is string data containing sets of key-value pairs, separated by line breaks:

```
key=value\n
```

The *key* is a single character that sets the type of the *value*. The *value* is a machine-readable configuration value.

The SDP covers media description and media constraints for a given user. When we start using *RTCPeerConnection* object later we will be able easily print this to the javascript console.

The SDP is the first part of the peer connection. Peers have to exchange SDP data with the help of the signaling channel in order to establish a connection.

This is an example of an SDP offer:

```
v=0

o=- 487255629242026503 2 IN IP4 127.0.0.1

s=-
```

```
t=0 0
a=group:BUNDLE audio video
a=msid-semantic: WMS 6x9ZxQZqpo19FRr3Q0xsWC2JJ1lVsk2JE0sG
m=audio 9 RTP/SAVPF 111 103 104 9 0 8 106 105 13 126
c=IN IP4 0.0.0.0
a=rtcp:9 IN IP4 0.0.0.0
a=ice-ufrag:8a1/LJqQMzBmYtes
a=ice-pwd:sbfskHYHACygyHW1wVi8GZM+
a=ice-options:google-ice
a=fingerprint:sha-256
28:4C:19:10:97:56:FB:22:57:9E:5A:88:28:F3:04:DF:37:D0:7D:55:C3:D1:59:B0:B2:
81:FB:9D:DF:CB:15:A8
a=setup:actpass
a=mid:audio
a=extmap:1 urn:ietf:params:rtp-hdrext:ssrc-audio-level
a=extmap:3 http://www.webrtc.org/experiments/rtp-hdrext/abs-send-time
a=sendrecv
a=rtcp-mux
a=rtpmap:111 opus/48000/2
a=fmtp:111 minptime=10
a=rtpmap:103 ISAC/16000
a=rtpmap:104 ISAC/32000
a=rtpmap:9 G722/8000
a=rtpmap:0 PCMU/8000
a=rtpmap:8 PCMA/8000
a=rtpmap:106 CN/32000
a=rtpmap:105 CN/16000
a=rtpmap:13 CN/8000
a=rtpmap:126 telephone-event/8000
a=maxptime:60
a=ssrc:3607952327 cname:v1SBHP7c76XqYcWx
```

```
a=ssrc:3607952327 msid:6x9ZxQZqpo19FRr3Q0xsWC2JJ1lVsk2JE0sG 9eb1f6d5-c3b2-
46fe-b46b-63ea11c46c74

a=ssrc:3607952327 mslabel:6x9ZxQZqpo19FRr3Q0xsWC2JJ1lVsk2JE0sG

a=ssrc:3607952327 label:9eb1f6d5-c3b2-46fe-b46b-63ea11c46c74

m=video 9 RTP/SAVPF 100 116 117 96

c=IN IP4 0.0.0.0

a=rtcp:9 IN IP4 0.0.0.0

a=ice-ufrag:8a1/LJqQMzBmYtes

a=ice-pwd:sbfskHYHACygyHW1wVi8GZM+

a=ice-options:google-ice

a=fingerprint:sha-256
28:4C:19:10:97:56:FB:22:57:9E:5A:88:28:F3:04:DF:37:D0:7D:55:C3:D1:59:B0:B2:
81:FB:9D:DF:CB:15:A8

a=setup:actpass

a=mid:video

a=extmap:2 urn:ietf:params:rtp-hdrext:toffset

a=extmap:3 http://www.webrtc.org/experiments/rtp-hdrext/abs-send-time

a=sendrecv

a=rtcp-mux

a=rtpmap:100 VP8/90000

a=rtcp-fb:100 ccm fir

a=rtcp-fb:100 nack

a=rtcp-fb:100 nack pli

a=rtcp-fb:100 goog-remb

a=rtpmap:116 red/90000

a=rtpmap:117 ulpfec/90000

a=rtpmap:96 rtx/90000

a=fmtp:96 apt=100

a=ssrc-group:FID 1175220440 3592114481

a=ssrc:1175220440 cname:v1SBHP7c76XqYcWx

a=ssrc:1175220440 msid:6x9ZxQZqpo19FRr3Q0xsWC2JJ1lVsk2JE0sG 43d2eec3-7116-
4b29-ad33-466c9358bfb3
```

```
a=ssrc:1175220440 mslabel:6x9ZxQZqpo19FRr3Q0xsWC2JJ1lVsk2JE0sG

a=ssrc:1175220440 label:43d2eec3-7116-4b29-ad33-466c9358bfb3

a=ssrc:3592114481 cname:v1SBHP7c76XqYcWx

a=ssrc:3592114481 msid:6x9ZxQZqpo19FRr3Q0xsWC2JJ1lVsk2JE0sG 43d2eec3-7116-
4b29-ad33-466c9358bfb3

a=ssrc:3592114481 mslabel:6x9ZxQZqpo19FRr3Q0xsWC2JJ1lVsk2JE0sG

a=ssrc:3592114481 label:43d2eec3-7116-4b29-ad33-466c9358bfb3
```

This is taken from my own laptop. It is complex to understand at first glance. It starts with identifying the connection with the IP address, then sets up basic information about my request, audio and video information, encryption type. So the goal is not to understand every line, but to get familiar with it because you will never have to work with it directly.

The following is an SDP answer:

```
v=0

o=- 5504016820010393753 2 IN IP4 127.0.0.1

s=-

t=0 0

a=group:BUNDLE audio video

a=msid-semantic: WMS

m=audio 9 RTP/SAVPF 111 103 104 9 0 8 106 105 13 126

c=IN IP4 0.0.0.0

a=rtcp:9 IN IP4 0.0.0.0

a=ice-ufrag:RjDpYl08FRKBqZ4A

a=ice-pwd:wSgwewyvypHhyxrcZELBLOBO

a=fingerprint:sha-256
28:4C:19:10:97:56:FB:22:57:9E:5A:88:28:F3:04:DF:37:D0:7D:55:C3:D1:59:B0:B2:
81:FB:9D:DF:CB:15:A8

a=setup:active

a=mid:audio

a=extmap:1 urn:ietf:params:rtp-hdrext:ssrc-audio-level

a=extmap:3 http://www.webrtc.org/experiments/rtp-hdrext/abs-send-time

a=recvonly

a=rtcp-mux
```

```
a=rtpmap:111 opus/48000/2

a=fmtp:111 minptime=10

a=rtpmap:103 ISAC/16000

a=rtpmap:104 ISAC/32000

a=rtpmap:9 G722/8000

a=rtpmap:0 PCMU/8000

a=rtpmap:8 PCMA/8000

a=rtpmap:106 CN/32000

a=rtpmap:105 CN/16000

a=rtpmap:13 CN/8000

a=rtpmap:126 telephone-event/8000

a=maxptime:60

m=video 9 RTP/SAVPF 100 116 117 96

c=IN IP4 0.0.0.0

a=rtcp:9 IN IP4 0.0.0.0

a=ice-ufrag:RjDpYl08FRKBqZ4A

a=ice-pwd:wSgwewyvypHhyxrcZELBLOBO

a=fingerprint:sha-256
28:4C:19:10:97:56:FB:22:57:9E:5A:88:28:F3:04:DF:37:D0:7D:55:C3:D1:59:B0:B2:
81:FB:9D:DF:CB:15:A8

a=setup:active

a=mid:video

a=extmap:2 urn:ietf:params:rtp-hdrext:toffset

a=extmap:3 http://www.webrtc.org/experiments/rtp-hdrext/abs-send-time

a=recvonly

a=rtcp-mux

a=rtpmap:100 VP8/90000

a=rtcp-fb:100 ccm fir

a=rtcp-fb:100 nack

a=rtcp-fb:100 nack pli

a=rtcp-fb:100 goog-remb

a=rtpmap:116 red/90000
```

```
a=rtpmap:117 ulpfec/90000

a=rtpmap:96 rtx/90000

a=fmtp:96 apt=100
```

You can find more SDP examples at https://www.rfc-editor.org/rfc/rfc4317.txt as well as more detailed specification at http://tools.ietf.org/html/rfc4566 .

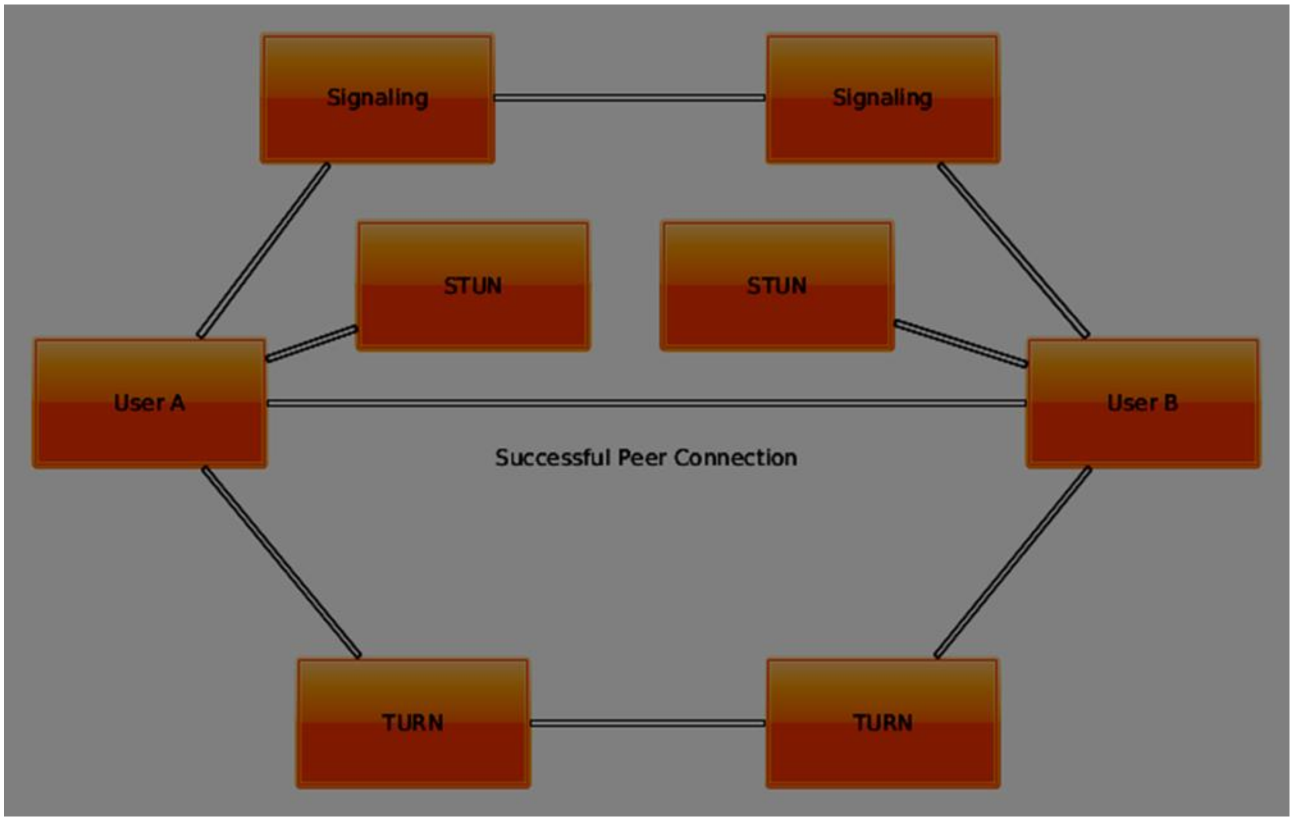
To sum up, the SDP acts as a text-based profile of your device to other users trying to connect to you.

Finding a Route

In order to connect to another user, you should find a clear path around your own network and the other user's network. But there are chances that the network you are using has several levels of access control to avoid security issues. There are several technologies used for finding a clear route to another user:

- STUN (Session Traversal Utilities for NAT)
- TURN (Traversal Using Relays around NAT)
- ICE (Interactive Connectivity Establishment)

To understand how they work, let's see how the layout of a typical WebRTC connection looks like:

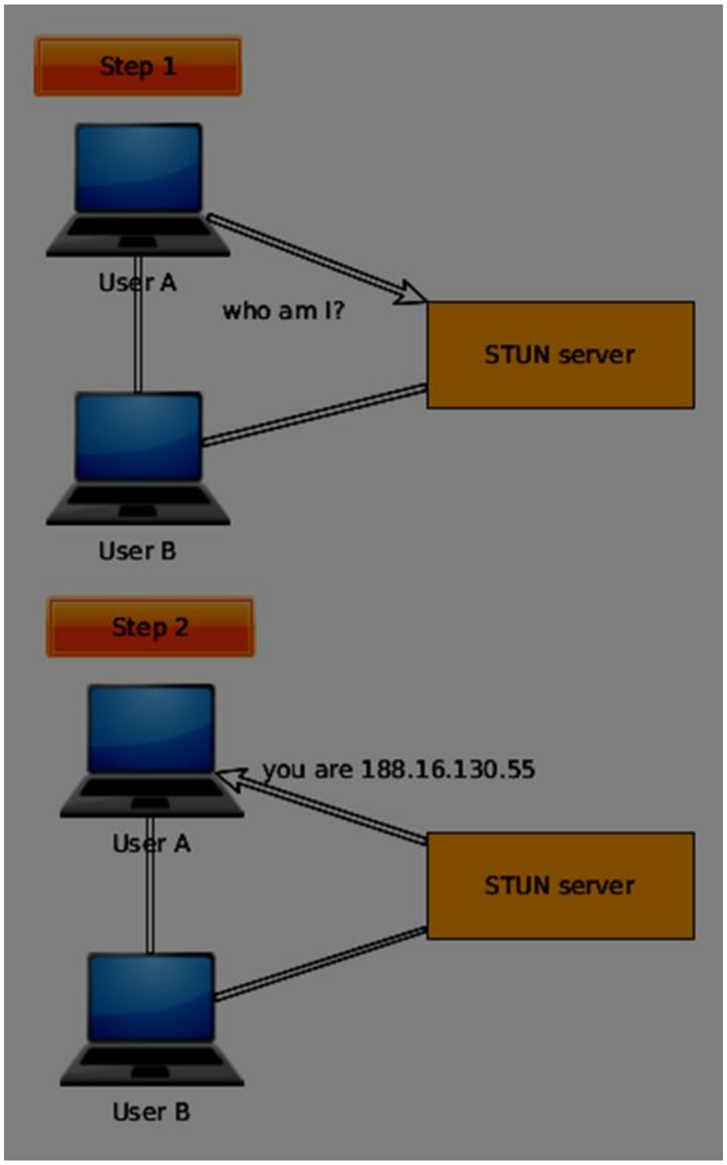


The first step is finding out your own IP address. But there's an issue when your IP address is sitting behind a network router. To increase security and allow multiple users to use the same IP address the router hides your own network address and replaces it with another one. It is a common situation when you have several IP addresses between yourself and the public Web.

STUN

STUN helps to identify each user and find a good connection between them. First of all it makes a request to a server, enabled with the STUN protocol. Then the server sends back the IP address of the client. The client now can identify itself with this IP address.

So basically there are two steps:



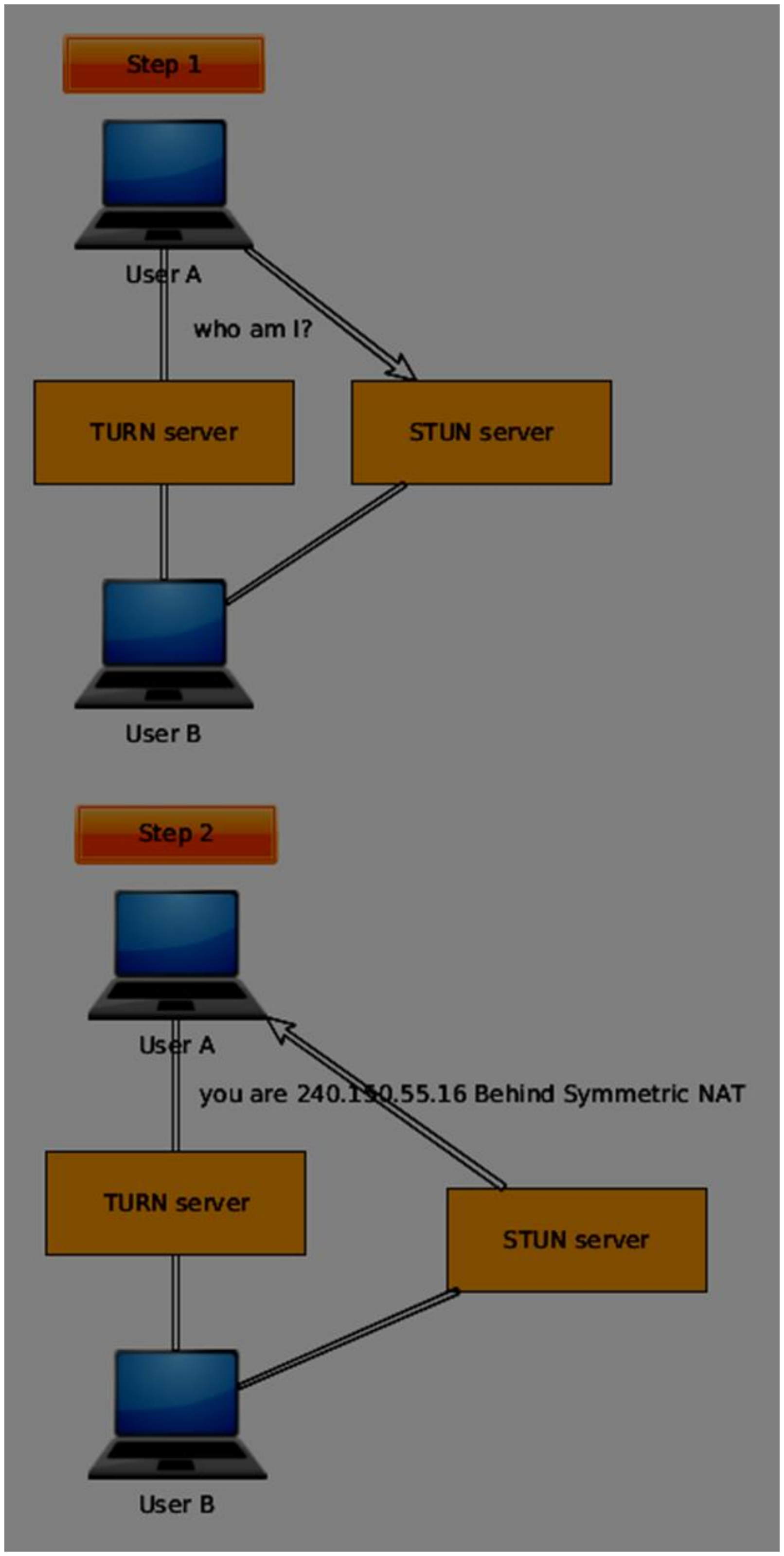
For using this protocol, you need a STUN-enabled server to connect to. The great thing is that Chrome and Firefox provide default servers out-of-the-box for you to test things out.

For application in a production environment, you will need to deploy your own STUN and TURN servers for your clients to use. There are several open-sourced services providing this today.

TURN

Sometimes there is a firewall not allowing any STUN-based traffic to the other user. For example in some enterprise NAT. This is where TURN comes out as a different method of connecting with another user.

TURN works by adding a relay in between the clients. This relay acts as a peer to peer connection on behalf the users. The user then gets its data from the TURN server. Then the TURN server will obtain and redirect every data packet that gets sent to it for each user. This is why, it is the last resort when there are no alternatives.


Step 1
User A
who am I?
TURN server
STUN server
User B
Step 2
User A
you are 240.150.55.16 Behind Symmetric NAT
TURN server
STUN server
User B

Most of the time users will be fine without TURN. When setting up a production application, it is a good idea to decide whether the cost of using a TURN server is worth it or not.

ICE

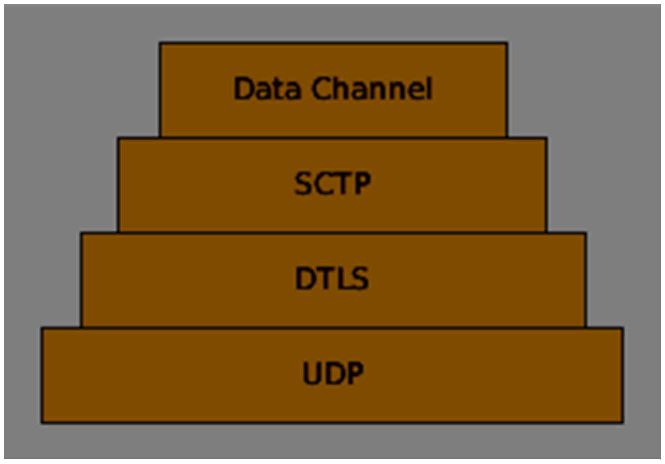
Now we can learn how STUN and TURN are all brought together through ICE. It utilizes STUN and TURN to provide a successful peer to peer connection. ICE finds and tests in sorted order a range of addresses that will work for both of the users.

When ICE starts off it doesn't know anything about each user's network. The process of ICE will go through a set of stages incrementally to discover how each client's network is set up, using a different set of technologies. The main task is to find out enough information about each network in order to make a successful connection.

STUN and TURN are used to find each ICE candidate. ICE will use the STUN server to find an external IP. If the connection fails it will try to use the TURN server. When the browser finds a new ICE candidate, it notifies the client application about it. Then the application sends the ICE candidate through the signaling channel. When enough addresses are found and tested the connection is established.

Stream Control Transmission Protocol

With the peer connection, we have the ability to send quickly video and audio data. The SCTP protocol is used today to send blob data on top of our currently setup peer connection when using the RTCDataChannel object. SCTP is built on top of the DTLS (Datagram Transport Layer Security) protocol that is implemented for each WebRTC connection. It provides an API for the data channel to bind to. All of this sit on top of the UDP protocol which is the base transport protocol for all WebRTC applications.

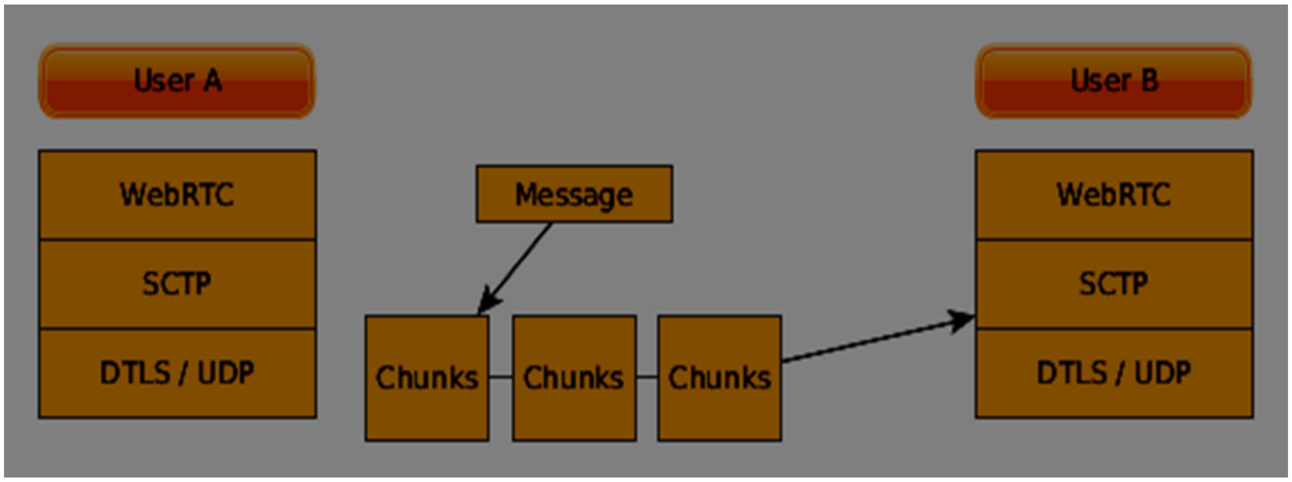


The developers of WebRTC knew that every application would be unique when using the data channel. Some might want the high performance of UDP while others might need the reliable delivery of TCP. That is why the created the SCTP protocol. These are the features of SCTP:

- There are two modes of the transport layer: reliable and unreliable

- The transport layer is secured
- When transporting data messages, the are allowed to be broken down and reassembled on the other side
- There are two order modes of the transport layer: ordered and not ordered
- Flow and congestion control are provided through the transport layer

The SCTP protocol uses multiple endpoints (number of connections between two IP locations), which sends messages broken down through chunks (a part of any message).



So you must understand that the data channel uses a completely different protocol than the other data-based transport layers in the browser. You can easily configure it up to your needs.

Summary

In this chapter, we covered several of the technologies that enable peer connections, such as UDP, TCP, STUN, TURN, ICE, and SCTP. You should now have a surface-level understanding of how SDP works and its use cases.

End of ebook preview
If you liked what you saw…
Buy it from our store @ **https://store.tutorialspoint.com**