

Real-Time Voice Chat Realized in JavaScript

Afshin Aresh

Master of Science in Engineering Technology
Engineering Physics and Electrical Engineering

Luleå University of Technology
Department of Computer Science, Electrical and Space Engineering

Abstract

Today it is possible to run most applications directly in the browser; people spend more and more time in front of a Web browser. Since audio and video communication is part of the Internet experience, major browsers are searching for ways to standardize real-time communication in Web browsers, without the need of plug-in installation. This thesis presents the possibility for real-time communication between browsers realized in JavaScript. The goal is to combine HTML5 with Mozilla audio data API for communication between browsers. To start with, the built-in media support for audio element of HTML5 has been used to embed the audio file. Extracting the samples from the audio element is done by implementing the Mozilla audio data API. These samples are compressed with a G.711 A-law codec; the codec will compress the input data by 50%. For transferring the data to the destination client and being able to receive data from the client a two-way communication channel is required. The HTML5 WebSocket API provides full-duplex communication over a single Transmission Control Protocol (TCP). The server environment chosen to support this was Node.js. All above combined together give the possibility of sending data between browsers through a server.

Table of Contents

1. Introduction.....	1
1.1 Methods.....	1
1.2 Layout.....	1
2. JavaScript API.....	2
3. Html5 Media Elements.....	3
3.1 The video Element.....	4
4. Reading Audio Samples.....	5
4.1 Mozilla audio data API.....	5
4.2 Web audio API.....	8
5. Media Codec.....	10
5.1 media encoding.....	10
5.2 media decoding.....	10
6. Media Transport.....	13
6.1 Polling.....	13
6.2 Long polling.....	13
6.3 WebSocket.....	13
6.4 Node server.....	16
6.5 Client side.....	20
7. Results.....	21
8. Discussion and Conclusions.....	23
9. Recommendations.....	24
9.1 Stream recording.....	24
9.2 Integrate with stream API.....	25
Acknowledgments.....	27
References.....	28
Appendices A.....	29

A.1 HTTP server.....	29
A.2 WebSocket Handshake.....	29
A.3 Unmask data.....	29
A.4 Bit manipulation.....	29
A.5 Mozilla audio data capture.....	31
A.6 Node server.....	32
A.7 Client.....	33

List of tables

3.1 Audio element attributes.....	3
3.2 Audio codec support.....	4
3.3 Video codec support.....	4
5.1 Compress.....	10
5.2 Expand.....	11
6.1 Attributes.....	15
6.2 Events.....	18

List of figures

2.1 Tree of document nodes.....	2
3.1 Default audio controls.....	3
4.1 Sinusoid.....	6
4.2 Audio nodes.....	9
6.1 WebSocket handshake.....	14
6.2 WebSocket framing.....	16

6.3 Command prompt.....	17
7.1 Result.....	21
9.1 Media stream.....	24
9.2 Record Stream.....	25
9.3 Accessing microphone audio.....	26

List of examples

3.1 Add audio element.....	3
3.2 Add video element.....	4
4.1 Create media.....	5
4.2 Drawing with canvas.....	6
4.3 Event listeners.....	7
4.4 Meta data capture.....	7
4.5 Capture samples.....	7
4.6 Rendering audio from another audio source	8
4.7 Integrate with HTML5s audio and video elements.....	8
4.8 Tone generator.....	9
6.1 WebSocket connection.....	14
6.2 Send data.....	15
9.1 Prompt user.....	24
9.2 Record stream.....	25
9.3 Accessing microphone audio.....	26

1. Introduction

Ever since the beginning the Internet has always been about transferring and retrieving information. There has been a lot of development over the years and nowadays Internet is a natural part of most people's lives. It has become cheaper and easier with the advance of mobile and laptop technology to stay connected no matter where you are. Most of the communication begins with the user opening a browser. The browser developers are continuously researching and pushing the limits for new ways to transfer information fast and easy to users. Major browser developers want to take the browser use to other levels to enable real-time communication through the browser without requirement of plug-ins. Right now it is possible to do signal processing directly in the browser with faster and more effective JavaScript engines. This thesis is performed for Ericsson, with the purpose to investigate real-time voice chat realized in JavaScript.

1.1 Methods

This thesis has been performed in two parts, programming and testing. The programming language is JavaScript and all the coding has been written in the programming environment Eclipse on Microsoft Windows operating system. For the testing part, different web browsers and Node.js server are used. To be able to debug the script and monitor the HTTP headers in Firefox and Chrome browser Add-ons have been added, such as firebug and http live headers. Firebug is used for debugging and analyzing the performance of the script and live HTTP headers for analyzing the HTTP headers. Furthermore, an HTTP header add-on has been added in chrome for monitoring the HTTP headers. Produced sound is made available by using the free software program audacity that later could be tested on the various browsers.

1.2 Layout

This thesis is divided into several chapters. Chapter 2 explains the programming language in short, followed by how to embed and play audio and video using element tags of HTML5. Chapter 4 describes how to gain access to the embedded audio samples. Chapter 5-7 describes how to encode the samples and send them between clients in real-time. In the final chapter there is a discussion for future work on this thesis.

2. JavaScript API

JavaScript is a scripting language for web browsers. All the modern browsers on desktops, tablets and smart phones include JavaScript interpreters. While Hyper Text Markup Language (HTML) is a markup language meaning a set of markup tags that form the website, embedding JavaScript makes the website dynamic. The purpose of JavaScript is to turn the client side static HTML documents into user interactive applications. JavaScript can interact with everything in the HTML document and the HTML document is built as a tree of objects in Document Object Model (DOM) which is an API for representing and manipulating HTML and XML documents [1]. Figure 2.1 shows a simple node with a tree of document nodes.

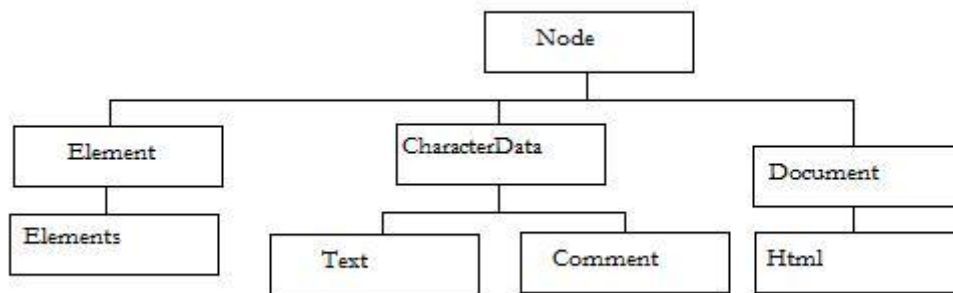


Figure 2.1 Tree of document nodes.

Figure 2.1 describes a node object with document as subtype in this case an HTML document. All characters belonging to the HTML document are stored in the subtypes of *CharacterData* depending on what kind of character it is, text or comment. The various elements added to this HTML document will be stored in subtypes of *element*. Most of the client side JavaScript programs is about manipulating the document element. There are several ways to refer to elements in the current document [1]. To refer to an element the `document.getElementById("element id")` is used in this thesis. Since the HTML document is embedded with graphical user interface it should be point out that JavaScript programs use an asynchronous event-driven programming model like all other graphical user interface (GUI) applications. Assume, for example, that we have a button element and a function which displays the current time. Whenever a user clicks on the button we want to know about it, in other words we are listening for an event to occur. When the event occurs we want to handle the user's requests by running the function, since the user obviously wants to know the time. This is just one type of event and there are others and many more are supported with arrival of new APIs. Later on in upcoming topics we will see how powerful JavaScript is combined with these APIs. Many of these new APIs require real-time processing. To perform this kind of data manipulation, all current versions of major browsers implement Just-In-Time (JIT) compilers. The JIT is a dynamic compiler, meaning that it reads byte code in chunks to compile and run the code when it is about to be executed. The JIT can also detect optimization patterns in the byte code. Different browser use different JIT compilers and the main focus are to run the code as efficiently as possible. The JIT compilers are fast because compiled code is cached to be reused when no changes have been made to the code.

3. HTML5 Media Elements

The Audio and video elements are two of HTML5's new features. To embed and play audio is as simple as adding an image and no plug-in is necessary. Example 3.1 shows how adding an audio element is done.

Example 3.1 Add audio element.

```
<audio controls>
<source src="audio.ogg" />
<source src="audio.mp3" />
</audio>
```



Figure 3.1 Default audio controls.

The `src` attribute specifies the URL where the user's audio file is located and the `controls` attribute displays the default player displayed in the webpage where a user can play or pause the audio. Figure 3.1 above shows that the players can look different depending on the current web browser. There are also other attributes of the audio element. Table 3.1 shows other useful attributes.

Table 3.1 Audio element attributes.

Attribute	Value	Function
Autoplay	Boolean	Audio plays automatically when page loaded.
Loop	Boolean	Repeat the content
Preload	None metadata auto	Preload just metadata, auto is to preload the whole file when page loads, none browser should not load the audio file.

The Audio element is supported by all major browsers and those who do not support the audio element a fallback on flash can be achieved. There is, however, one problem with the audio tag and that being there are no common codecs for the media element. Different browsers use different codecs for legal reasons. The user can get around this problem by allowing the audio tag to point at different audio formats. This can be achieved through multiple source attributes and only the supported one will be downloaded.

Table 3.2 shows the five major browsers support for audio formats. Observe that it is sufficient for the user to specify two audio sources OggVorbis and MP3 to ensure that the five major browsers will support one of the formats. It is also worth mentioning that using larger wav files should be avoided, since these are uncompressed.

Table 3.2 Audio codec support.

Format	IE	Firefox	Opera	Chrome	Safari
OggVorbis	No	Yes	Yes	Yes	No
MP3	Yes	No	No	Yes	Yes
Wav	Yes	Yes	Yes	Yes	Yes

3.1 The video element

The video element shares all the audio attributes and some other attributes that gives the user control over how the video is displayed in the browser. As mentioned before, there are no common codecs for all the browsers. Table 3.3 shows the five major browsers and the video codecs supported.

Table 3.3 Video codec support.

Format	IE	Firefox	Opera	Chrome	Safari
OggTheora	No	Yes	Yes	Yes	No
WebM	No	Yes	Yes	Yes	No
MPEG-4 H.264	Yes	No	No	Yes	Yes

The user can embed a video file in HTML5 just as easily as an audio file. Example 3.2 shows how this can be achieved.

There are two new attributes in Example 3.2.

Poster

Poster is an image or a text message the user wants to display as the first frame when the page is loaded.

Width and height

The width and the height are the size of the video player. The size can be adjusted by giving a number of pixels or by percentage of the screen the video should occupy.

Example 3.2 Add video element.

```
<video poster="image.jpg" width="pix/%" height="pix/%" controls>
  <source src="video.ogg" />
  <source src="video.mp4" />
</video>
```

4. Reading Audio Samples

The HTML5 media element API provides ways to play and get limited information about capturing the video and audio metadata, but lacks the ability to access or create media. To gain access to the HTML5s audio element samples, two different audio APIs are discussed below.

4.1 Mozilla audio data API

Mozilla have achieved the goal when it comes to capturing the raw audio data from the audio or video element. Mozilla makes the raw audio data capturing and rendering available through an event-based API known as Mozilla Audio Data API [2]. Example 4.1 shows how it is done creating a one second sinusoid signal.

Example 4.1 Create media.

```
<script>
  var audio = new Audio();
  var channels=1, sampleRate=44100;
  audio.mozSetup(channels, sampleRate);
  var buffer=new Array(44100);
  for(var i=0; i<buffer.length;i++){
    buffer[i]=Math.sin(i/10);
  }
  audio.mozWriteAudio(buffer);
</script>
```

First a new audio element is created. The audio is prepared for playback by calling the method `mozSetup` with channels and sample rate as parameters. Then an array is created and filled with 44100 samples. Finally the `mozWriteAudio` method is called with the audio samples as parameter to write the sinusoid signal. It is also possible to visualize the wave created in Example 4.1 by introducing another element of HTML5, called the canvas element. The canvas tag is used to draw graphics via scripting. Example 4.2 shows the canvas element added to Example 4.1.

Example 4.2 Drawing with canvas.

```
<canvas width="300" height="200"></canvas>
<body>
<script>
var canvas = document.getElementsByTagName("canvas")[0];
var context = canvas.getContext('2d');
var fbLength=buffer.length;
var channels=channels;
var samples = 600;
var step = (fbLength / channels) / samples;
context.fillRect(0, 0, 280, 180);
context.strokeStyle = "#FFF";
context.lineWidth = 2;
context.beginPath();
context.moveTo(0, 100-buffer[0]*100);
for(var i=1; i< samples; i++){
context.lineTo(i, 100-buffer[i*step]*100);
}
context.stroke();
</script>
```

The result of Example 4.2 is shown in Figure 4.1. First a canvas element is created with the width and height as specified, which gives the user control over this rectangular area of pixels. The reference to the canvas element is set to canvas. A rectangular 2D context object is created and given a rectangular shape of 280x180 pixels. From here on it is straight forward, the line width, color and beginning of the context are specified. The `moveTo` positions the starting point and the `lineTo` method draw a line from the context point to given point.

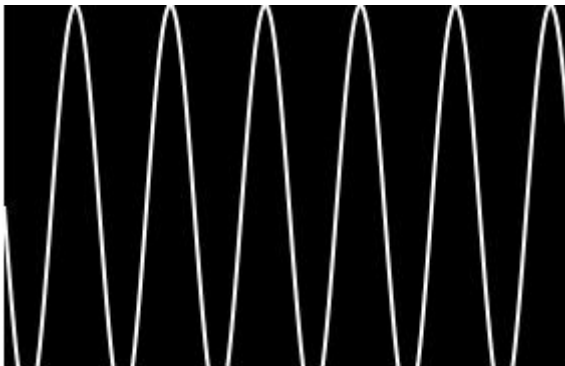


Figure 4.1 Sinusoid.

Now that it is clear how to generate sound and visualize the samples it is time to use the Mozilla Audio Data API to access the audio samples from the media source. There are two event listeners which make it possible to access various type of information about the audio data. Example 4.3 shows these two event listeners

Example 4.3 Event listeners.

```
<audio controls id="audio" src="audio"></audio>
<script>
    var audio = document.getElementById("audio");
    audio.addEventListener('MozAudioAvailable', samples, false);
    audio.addEventListener('loadedmetadata', meta, false);
</script>
```

The reference to the audio is set through `getElementById` and an event listener is attached to the audio file. The `loadedmetadata` event is a standard part of HTML5. The event occurs when useful metadata are loaded. Example 4.4 shows how to access some of the metadata such as the number of channels, the sampling rate, and the frame buffer length.

Example 4.4 Meta data capture.

```
var frameBufferLength, channels, sampleRate;
function meta() {
    frameBufferLength = audio.mozFrameBufferLength;
    channels = audio.mozChannels;
    sampleRate = audio.mozSampleRate;
}
```

The `MozAudioAvailable` event provides two pieces of information the frame buffer containing the raw audio data and the time for these samples measured from the start of the file. Example 4.5 shows how to access this data.

Example 4.5 Capture samples.

```
audio.addEventListener('MozAudioAvailable', requireSamples, false);
function requireSamples(event) {
    var buffer = event.frameBuffer;
    var time = event.time;
    for ( var i = 0; i < buffer.length; i++) {
        //have access to every sample
    }
}
```

The Event listener triggers `MozAudioAvailable` each time there is a frame buffer and the function `requireSamples` is executed. Function variable `buffer` is assigned the captured frame buffer and the time variable is assigned the time when the measurement was taken. The loop iterates through the frame buffer array containing by default 1024*channels signed 32-bit floating point raw data in form of [channel1, channel2, channel1...] and all values lie between [-1 1]. Now when the raw audio data is available it can be manipulated and called upon with `mozWriteAudio` to playback the new, by user modified audio data. The user can also attach the audio event listeners to a video element and extract the same audio data from the video media. The purpose here is to access the raw audio data for encode and decode with G.711. More about the codec implemented is discussed in chapter 5, for now Example 4.6 shows how the samples are accessed and played throw another media element.

Example 4.6 Rendering audio from another audio source.

```
<audio controls id="audio" src="Location of audio file"></audio>
<script>
    var audio1 = document.getElementById("audio");
    var audio2 = new Audio();
    var buffers = new Array();
    function loadedMetadata() {
        audio.volume = 0;
        audio2.mozSetup(audio1.mozChannels,
audio1.mozSampleRate);
    }
    audio1.addEventListener('MozAudioAvailable', requireSamples,
false);
    audio1.addEventListener('loadedmetadata', loadedMetadata,
false);

    function requireSamples(event) {
        var buffer = event.frameBuffer;
        for ( var i = 0; i < buffer.length; i++) {
            var enc = encode(buffer[i]);
            var dec = decode(enc);
            buffers[i] = dec;
        }
        var written = audio2.mozWriteAudio(buffers);
    }
</script>
```

Example 4.6 above describes the possibility of rendering audio from another audio source. First of all `audio1`'s reference is set to the audio element. Second an empty audio element (`audio2`) is created and assigned `audio1`'s channels and sample rate in the `loadedMetadata` function. All the samples are encoded and decoded in the `requireSamples` function. This means that as soon as we capture a frame from `audio1` we are doing processing and passing it directly to `audio2`. It should be noted that Mozilla data API currently only works in Firefox.

4.2 Web audio API

The Web Audio API is a W3C proposal and is in part supported in WebKit-based browsers such as Chrome and Safari. This API does also integrate with the HTML5 audio and video elements. To show the difference between these two APIs, the integration part with HTML5 is shown in Example 4.7 and Example 4.1 is then repeated in Example 4.8.

Example 4.7 Integrate with HTML5s audio and video elements.

```
<audio controls src="DTMF.wav" id="audio"></audio>
<body>
    <script>
        var context = new webkitAudioContext();
        var mediaElement = document.getElementById("dtmf.wav");
        var mediaElementSource = context.createMediaElementSource(mediaElement);
        mediaElementSource.connect(context.destination);
    </script>
</body>
```

The best way to describe Example 4.7 is by looking at the block diagram in Figure 4.2. To start off you need to create an `AudioContext` interface which represents a set of audio nodes and how they are connected. These nodes have inputs, outputs or both. There are two mandatory nodes to be able to output any sound at all, the source node which have an output, no input and the destination node that plays back the sound. All nodes in the context will be directly or indirectly connected to the destination node. Connecting two node is done by the `connect` method [3].

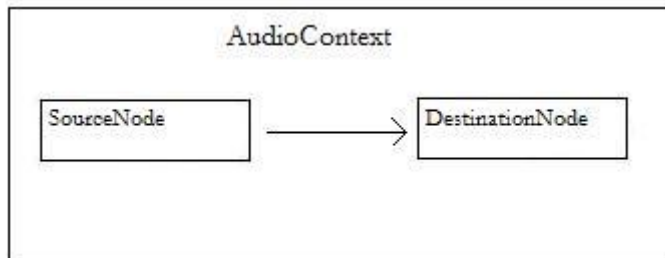


Figure 4.2 Audio nodes.

Example 4.8 shows a tone generator, where the context created owns everything inside it. In the next step a `JavaScriptNode` is added. This node can have multiple inputs and outputs, one of the inputs or outputs must be greater than zero. In this example the node has no input, one output and a buffer frame of 1024 samples which is filled with samples by calling the `onaudioprocess` event. The `JavaScriptNode` could be connected to an audio element and access the samples. This is for the time being not possible in any major browser. The difference between these Two APIs is that in the Mozilla Data API the user gains access to the samples and the limit of what can be achieved is set by the user. The Web Audio API provides objects that can be interconnected to add functionality to the media.

Example 4.8 Tone generator.

```

<script>
    var context = new webkitAudioContext();
    var node = context.createJavaScriptNode(1024, 0, 1);
    node.onaudioprocess = function(e) {
        var offset = 0;
        var buffer = e.outputBuffer;
        var left = buffer.getChannelData(0);
        var right = buffer.getChannelData(1);
        for ( var i = 0; i < buffer.length; i++) {
            left[i] = right[i] = Math.sin(i / 10);
        }
    };
    node.connect(context.destination);
</script>

```

5. Media Codec

For encoding and decoding the media the G.711 codec, also known as pulse code modulation (PCM), is chosen. It is a common and easy codec to implement. G.711 is a codec for companding audio data, meaning compression and expansion of data. There are two different algorithms for G.711; the μ -law is used in North America and Japan [4] and the A-law is used in Europe and the rest of the world [4].

5.1 Media encoding

The A-law compression takes a 16 bit input signal and produces a compressed 8-bit code. Table 5.1 shows how this is done.

Table 5.1 Compress.

16 bit signed input	8 bit signed compressed code
S0000000abcde...	S000abcd
S0000001abcde...	S001abcd
S000001abcdef...	S010abcd
S00001abcdefg...	S011abcd
S0001abcdefgh...	S100abcd
S001abcdefghi...	S101abcd
S01abcdefghij...	S110abcd
S1abcdefghijk...	S111abcd

The most significant bit (MSB) is the sign bit (S) 1 for negative and 0 for positive numbers. Note that G.711 ignores the 3 least significant bits (LSB). By applying this algorithm the user will get a 50% compression of the input value. The input value is divided into 16 segments, 8 positive and 8 negative. The three bits after the sign bit represent which segment the input signal is confined to. Each segment has 16 intervals, starting with a step size of two and doubling for each higher segment up to a step size of 128. This means that higher input values will have a bigger encoding error.

5.2 Media decoding

To reconstruct the original data, the compressed code needs to be expanded. Table 5.2 shows how to expand the compressed code. The script code can be viewed in the bit manipulation code below. Most of the code is straight forward.

Table 5.2 Expand.

8 bit signed compressed code	13 bit signed input
S000abcd	S0000000abcd1
S001abcd	S0000001abcd1
S010abcd	S000001abcd10
S011abcd	S00001abcd100
S100abcd	S0001abcd1000
S101abcd	S001abcd10000
S110abcd	S01abcd100000
S111abcd	S1abcd1000000

Bit manipulation code

```
var max = 0x7fff; // 011111111111111
var upscale = 32767;
function compress(pcm) {
  pcm = pcm * upscale;
  var signbit = 0;
  if (pcm < 0) {
    signbit = 1;
    pcm = -pcm;
  }
  if (pcm > max) { // prevent overflow
    pcm = max;
  }
  pcm = (pcm & 0xffff);
  pcm = pcm.toString(2);
  pcm = padLeft(pcm, 15);
  var msb = pcm.slice(0, 12);
  var output;
  var segment = 0;
  var intervall = 0;
  // loop throw 8 bit msb until first 1 is found or count is 0
  for ( var i = 0; i < msb.length - 4; i++) {
    if (msb[i] == 1) {
      segment = 7 - i;
      intervall = msb.slice(i + 1, i + 5);
      break;
    }
  }
  segment = segment.toString(2);
  segment = padLeft(segment, 3);
  intervall = padLeft(intervall, 4);
  signbit = signbit.toString(2);
  output = signbit.concat(segment, intervall);
  return output;
}
```

```

function expand(input) { // 8 bit input
  var signbit = input.slice(0, 1);
  var segment = input.slice(1, 4);
  var intervall = input.slice(4, input.length);
  var output;
  var mask = 0x1;
  segment = parseInt(segment, 2).toString(10);
  if (segment == 0) {
    output = intervall.concat(mask);
    output = padLeft(output, 12);
    output = signbit.concat(output);
    output = padRight(output, 16);
  } else {
    mask <= (segment - 1);
    mask = mask.toString(2);
    output = padLeft(1, (8 - segment));
    output = output.concat(intervall);
    output = output.concat(mask);
    output = padRight(output, 15);
    output = signbit.concat(output);
  }
  if (signbit == 1) {
    output = output.slice(1, output.length);
    output = -parseInt(output, 2);
  } else {
    output = parseInt(output, 2);
  }
  return output / upscale;
}

```

6. Media Transport

HTTP was first designed to be used for retrieving hypertext from a server. The client requested a document and received a response in the form of either a document or an error, if no such document existed. This meant that communication could take place one way (half –duplex). Nowadays the user is more involved and the need for real-time communication grows. There have been some application developments over the years to simulate real-time communication, which is described below.

6.1 Polling

With polling the client send a request to the server in specific time intervals asking if there is information available and the server responds immediately. If there is data available the server sends the data and if not the server sends an empty response. The client keeps repeating this procedure continuously, the smaller the interval the more fresh the data and the bigger load on the server. Longer intervals will leave the user with less up-to-date information but loads off the server.

6.2 Long polling

With long polling the user sends one request to the server asking for some data, the server keeps that request until there is data available for the client and responds the request with the data. The client must, however, immediately send another request when the response is received in order to get more data. This method is not based on a time interval but rather on when information is available. The downside of this method is that if there are a lot of requesting clients the server must hold all those requests with the entire header associated with the requests. This means that the server must save a lot of unnecessary data. There have been developments at the server side to handle this kind of requests, the most recent is the HTML5 Server-sent-events API. The client opens an event stream by creating an event source with a specified URL and listens for events. The server opens persistent connection to send data when available and a message event triggers at the client side which can be handled as desired. This API is supported by all major web browsers except Internet Explorer (IE). This kind of application is desired when it comes to broadcasting the same data to a lot of users, meaning when the communication is unidirectional. To achieve real-time bidirectional communication WebSocket API of HTML5 is currently the best solution.

6.3 WebSocket

WebSocket communicates over a single Transmission Control Protocol (TCP) socket. The client sends a regular HTTP handshake with a connection upgrade request and the server responds with a WebSocket handshake response, as shown in Figure 6.1.

Client

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Origin: http://example.com
Sec-WebSocket-Protocol: protocol1,protocol2..
Sec-WebSocket-Version: 13
```

Server

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
Sec-WebSocket-Protocol: protocol1
```

Figure 6.1 WebSocket Handshake.

The client handshake must contain all the header fields except for the sec-WebSocket-Protocol. This header is optional for client selecting one or more sub protocols. If the protocol is supported by the server then the server responds with the sub protocol selected by the server as seen from the above figure. The get header is for identifying the destination WebSocket connection and for allowing multiple connections from same IP. The origin header is for protecting against unauthorized cross-origin. The server must prove to the client that handshake has been received by taking the key provided in the client header minus any leading and trailing whitespace. This should be concatenated with the string "258EAF5E914-47DA-95CA-C5AB0DC85B11", hashed with SHA1 and finally base64 encoded. The response from the server will be the calculated value. Once all have been completed without any error the server response shown in Figure 6.1 will be echoed back to the client and there is agreement between server and client to WebSocket. Once the server and the client have agreed on an upgrade from HTTP to WebSocket, each side can initiate communication at any time. Example 6.1 shows how to create a WebSocket connection and table 6.1 shows attributes used in this paper [5].

Example 6.1 WebSocket connection.

```
var socket= new WebSocket("ws://URL");
```

Ws defines a regular connection. The secure WebSocket connection requires the prefix wss: for Transport Layer Security (TLS). The URL specifies the host to connect to port 80 for ws or port 443 for wss by default [6].

Table 6.1 Attributes.

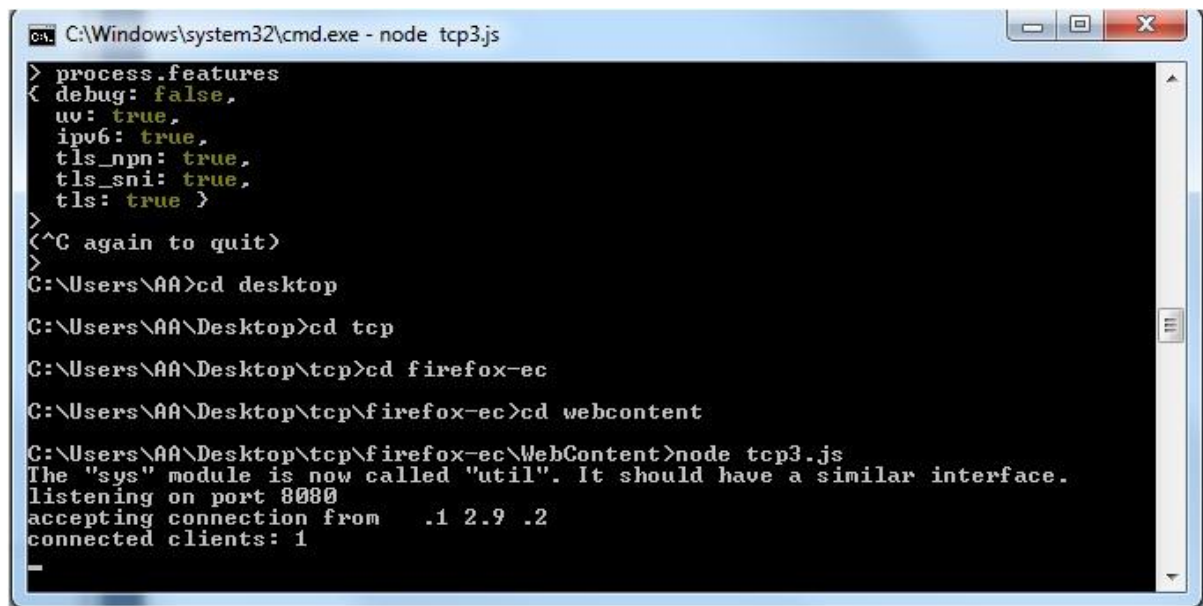
Attribute	Description
readyState	State of connection 0,1,2,3 stands for connecting open, closing and closed.
onmessage	Event handler called when message received.
onopen	Event handler called when socket is open.
onclose	Event handler called when socket is closed.
onerror	Event handler called when there is an error.
url	Returns the current URL of the socket.

Example 6.2 show how to send data to the server once connection has been established .

Example 6.2 Send data.

```
var socket= new WebSocket(ws://URL:port)
socket.send("message");
socket.close();
```

The send method accepts a string; the string is encoded using UTF-8 and transmitted with two bytes of framing according to Figure 6.2. The data is packaged as octets. The first bit in the first octet, if set this is the final frame, otherwise there is continuation. The following 3 bits are reserved and the final 4 bites of the first byte describe the type of data. How to set these bits is described in the hybi-17 draft, which is a WebSocket protocol for the current version of WebSocket. The opcode can be set to 0, 1, 2, 8, 9, 10 for continuation frame, text frame, binary frame, connection close, ping and finally pong, respectively. The masking key must be set for hybi-7 to hybi-17 protocols for security reasons. Once the masking key is set the browser must generate 4 octets of masking key and mask the data. There should be no connection whatsoever between the masking keys when masking each frame. When masked, the browser will send the masking key along with the masked payload data and the server can perform the action needed for retrieving the original information. A code example for unmasking can be found in Appendix A.3. The payload length from figure 6.2 gives the length of data. If the payload is less than 126 this means this is the actual data. If the payload is 126 the following two bytes should be read as a 16 bit unsigned integer and if the payload length is 127 the following 8 bytes should be interpreted as 64 bit unsigned integer [5].



```
C:\Windows\system32\cmd.exe - node tcp3.js
> process.features
< debug: false,
  uv: true,
  ipv6: true,
  tls_npn: true,
  tls_sni: true,
  tls: true >
>
<^C again to quit>
>
C:\Users\AAA>cd desktop
C:\Users\AAA\Desktop>cd tcp
C:\Users\AAA\Desktop\tcp>cd firefox-ec
C:\Users\AAA\Desktop\tcp\firefox-ec>cd webcontent
C:\Users\AAA\Desktop\tcp\firefox-ec\WebContent>node tcp3.js
The "sys" module is now called "util". It should have a similar interface.
listening on port 8080
accepting connection from .1 2.9 .2
connected clients: 1
```

Figure 6.3 Command prompt.

httpServer module

This module loads the http server API. Once loaded, an http server is created that listens to port 4000.

The `http.createServer(request, response)` is an event emitter and the event is triggered every time there is a request. Because nodes methods are asynchronous, this means that it cannot return the result of the request unless a function is provided when the request arrives. In this case the response is a HTML file with content. The full code of the http server can be viewed in Appendix A.1. Observe that an httpServer module is created by simply adding the last line `exports.function=function`. This will allow the user to access the module in node and keeps the code clean.

path module

This module contains methods for handling file paths. The method `path.extname("file path")` is used to identify the extension of the pathname

fs module

The fs module is the node file system API. Once the extension is identified the `fs.readFile(filepath, content)` checks if the html file exists, if so the content of the file is read and served with proper mime type.

net module

This module is uploaded to create a TCP server that listens on port 8080.

upgradeToWebSocket module

This module performs the handshake and return the server handshake header seen from Figure 6.1, the server side. The code is available in Appendix A.2. Once the handshake is approved by the server the client will have approval to send and retrieve data on port 8080.

As seen from the Node.js server code on page 19 there are various event handlers for achieving this task. Table 6.2 shows these event emitters.

Table 6.2 Events.

Event	Description
connect	Fires every time there is a connection.
data	Fires every time data is arriving to the server.
close	Fires every time a client disconnects from the server.

Connect

Emitted when there is a new WebSocket connection and the client associated with the socket is put in an array. This is to keep track of all clients. The clients IP will also be displayed on the command line together with number of clients connected to the server.

Data

The event is emitted when the server receives data from the clients. The data is inspected to see if this is a new client requesting WebSocket handshake by looking for an upgrade in the header. If there is an upgrade, the handshake is performed and the server sends the response headers back to the client. When no upgrade is found the server will send the data to all connected clients using the `socket.write("text")` method.

Close

Emitted when a client disconnects from the server. The client is located and deleted from the array of clients and the number of connected clients is displayed on the command line.

Node.js server code

```
var net = require('net');
var unmask = require("./socketcodec.js");
var upgradeToWebSocket = require("./upgradeToWebSocket.js");
var httpServer = require("./httpServer.js");
var clients = [];
httpServer.runHTTP();
var server = net.createServer(function(socket) {

    socket.on("connect", function() {
        clients.push(socket);
        console.log("client connected:" + " " + clients.length);
        console.log("connection" + " " + socket.remoteAddress);
        socket.setTimeout(0);
        // dont buffer data before sending it fire of socket.write
        socket.setNoDelay(true);
    });
    socket.on("data", function(data) {
        // obtain the upgrade header
        var change = /Upgrade: (.*)/g;
        var comp = change.exec(data);
        if (comp != null) {
            // if requesting upgrade, upgrade to WebSocket
            var response = upgradeToWebSocket.handshake(data);
            socket.write(response);
        } else {
            //var sender=socket.remoteAddress;
            var sender=clients.indexOf(socket);
            for (var i = 0; i < clients.length; i++) {
                if(clients[i]!==clients[sender]){
                    clients[i].write(data);
                }
            }
        }
    });
    socket.on("close",
        function() {
            var index = clients.indexOf(socket);
            console.log("connection: ", clients[index].remoteAddress,
                "terminated");
            clients.splice(index, index + 1);
            console.log("connected clients: ", clients.length);
        });
});

server.listen(port);
```

6.5 Client side

Most of the client code below is straight forward, a WebSocket connection is established to the server as earlier shown in Example 6.1 and listens on port 8080. Firefox version 6.0 and above use the `mozWebSocket` prefix to open a WebSocket connection and this is handled by the first script lines. In the `requireSamples` function all samples are encoded and sent as a long string through the WebSocket. When the data is received at the server side, the data event is emitted and the server sends the data to all connected clients. Once the server starts sending the data the `onmessage` event fires at the client side and the data is stored in a variable. The variable is decoded and the decoder outputs an array of 2048 samples. That array is rendered through into `audio2` with the `mozWriteAudio` method.

Client side code

```
<body>
  <script>
    if (typeof (MozWebSocket) == "function") {
      this.socket = new MozWebSocket("ws://1.1.1.36:8080/");
    } else {
      this.socket = new WebSocket("ws://localhost:8080/");
    }
    var audio = document.getElementById("audio");
    var payload;
    var audio2 = new Audio();
    function metaData() {
      audio.volume = 0;
      audio2.mozSetup(audio.mozChannels, audio.mozSampleRate);
    }
    audio.addEventListener('loadedmetadata', metaData, false);
    audio.addEventListener('MozAudioAvailable', requireSamples, false);
    function requireSamples(e) {
      var buffer = e.frameBuffer;
      var string = "";
      for ( var i = 0; i < buffer.length; i++) {
        var enc = encode(buffer[i]);
        string += enc;
      }
      socket.send(string);
    }
    socket.onmessage = function(e) {
      payload = e.data;
      var dec = decode(payload);
      var written = audio2.mozWriteAudio(dec);
    }
  </script>
</body>
```

7. Results

Figure 7.1 shows a block diagram over all parts covered this far and how they are connected.

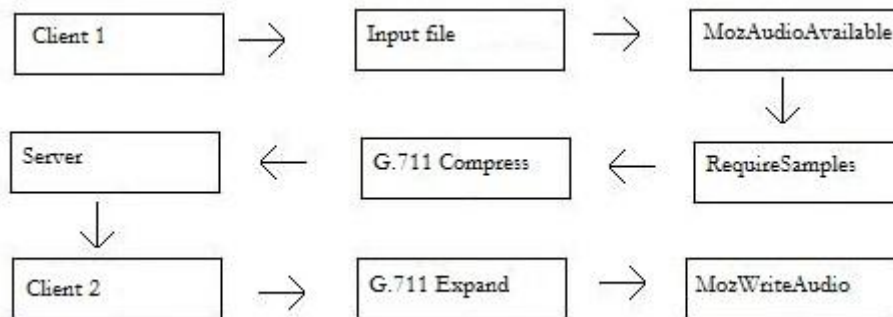


Figure 7.1 Result

Client 1

A client is served a HTML file with an audio element as content.

Input file

This is the audio file embedded in the HTML document and served to all connected clients. The clients will see a default audio player when connected to the server.

MozAudioAvailable

Event listener attached to the audio file to capture the frame buffer when available.

RequireSamples

Function used to access the frame buffers raw audio data. This function executes when `mozAudioAvailable` is emitted.

G.711 Compress

Takes a 16 bit audio data accessed in the `requireSamples` function as input and outputs an 8 bit compressed code.

Server

The server holds a TCP connection open and listens for WebSocket connections on port 8080. The server receives compressed data and send that data to the destination clients.

Client 2

A client is served with a HTML file with an audio element as content.

G.711 Expand

The client receives the compressed data from the server and expands that data to its original format for later use.

MozWriteAudio

This is a Mozilla Audio Data API method for playing an array of samples.

So this means it is possible to combine the HTML5 audio tag, WebSocket API and Mozilla audio data API to send raw audio data between browsers. So far only a file as source has been used to send media through a WebSocket to another client. To enable real-time communication the client application needs to gain access to the microphone and send that data through WebSocket. There are browser developers working on how to access the microphone through the browser with the JavaScript API at the moment and there is a stream API a W3C specification that shows how this can be done. Chapter 9 will discuss how this thesis can be adopted to W3C specification for enabling real-time communication.

8. Discussion and Conclusions

So far raw audio data have been successfully captured and sent to an echo server to simulate client to client through server communication. In this thesis it has been possible to gather some of the new browsers technology with elements and APIs of HTML5 to process raw audio data in real time. There is a lot of research in this area for enabling real-time communication between browsers without the needs of plug-ins. In latest versions of Chrome and Opera it is possible to search in Google by voice [13]. Once you're in google.com you will see a microphone icon in the search bar, to input speech you simply click on the microphone icon and start speaking. This has been possible for some time now, the Mozilla browser has an add-on called rainbow, available for Firefox nightly builds on Mac. The add-on provides video and audio capture capabilities to web pages for recording audio and video. At the present there is an open project for enabling real-time communication between browsers called webRTC. This project is supported by Google, Mozilla, Opera and Ericsson. The project is exploring the possibility for peer to peer connection where media streams from user devices are sent over user datagram protocol (UDP). While webRTC groups are working on getting a demo online, Ericsson labs have actually accomplished this task for some time now. First in 2010 the group posted a video on their blog showing how the user can choose the device input element such as webcam and then communicate through WebSocket [8]. Later in 2011, Ericsson labs published another video on their blog for real time communication, only this time modified to transport media using RTP (real-time transport protocol) over UDP[9].

9. Recommendations

As mentioned before there is an open project for enabling web real-time communication (webRTC) between browsers. Chrome is close to implementing this new feature in their browser, but at the moment this is not implemented in any major browser. This chapter describes how the current W3C specification stream API can be combined with this thesis to enable communication between browsers. To start off when there is a desire to communicate with another client through the browser, the web application needs to somehow prompt the user if it is accepted to use the microphone input. Example 9.1 shows how this can be done.

Example 9.1 Prompt user

```
navigator.getUserMedia({audio:true, video:false  
    },gotAudio,errHandler);
```

The above example asks permission to access the user microphone input. Once user accepts, the callback function `gotAudio` is invoked and if denied the `errHandler` is invoked. When the user accepts the invitation that include only audio, the media stream seen from Figure 9.1 is presented with audio stream data and there will be no video data.

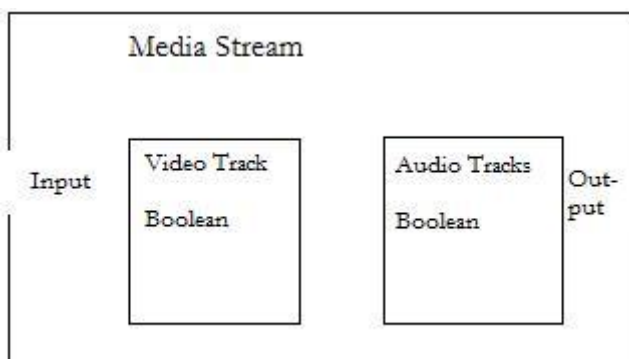


Figure 9.1 Media stream

The media stream can accept audio and video and the audio track can have multiple channels for surround sound. The input in this case is the user's microphone and the output will be the audio stream data. The stream API has some interfaces with methods for displaying information about the current stream of data, two of them that are interesting here are the `MediaStreamRecorder{} and createObjectUrl()`, which provides methods for recording and accessing the audio data [10].

9.1 Stream recording

Example 9.2 shows how one could record the audio or video message and upload it to a server.

Example 9.2 Record stream

```
<script>
navigator.getUserMedia({audio:true, video:false

    },gotAudio,errHandler);

    var stream,streamRecorder,data;

    function gotAudio(audioStream){
        stream=audioStream;
        streamrecorder=stream.record();

        setTimeout(function() {
            data = streamRecorder.getRecordedData();
            socket.send(data);
        }, 5000);
    }
</script>
```

First of all the application must send `get UserMedia` to ask for permission to use the microphone input. Once the user accepts, the audio Boolean will be set to true and the stream will now only send audio data. When the user accepts the inquiry the `gotAudio` callback function is invoked. This function will run every time user accepts a microphone invitation. Finally the recorded data can be accessed by calling the `getRecordedData()` function which will return a binary large object(Blob) of the recorded data. This blob of data could then be uploaded to a server.

9.2 Integrate with stream API

To access the captured data from users microphone it is possible to use the `createObjectURL()` method from the stream API interface URL. Example 9.3 shows how this method can be combined with the result of this thesis to simulate real-time communication over WebSocket with the client using microphone as input instead of a file. As always the user needs to be prompted to give permission for microphone communication. Once the user accepts the invitation, the callback function `gotAudio` is invoked. Inside this function the `URL.createObjectURL(stream)` can be called, this method returns a Blob URL and the source attribute of the audio element can be set to this returned value. Before an audio file was used as the URL, now the captured data from the microphone is the URL. This means that one can use either the Mozilla audio data API, as in Example 9.3, or the web audio API to capture samples from the audio element and transport over WebSocket to the destination clients audio elements for playback.

Example 9.3 Accessing microphone audio.



```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Insert title here</title>
<script type="text/javascript" src="codec3.js"></script>
<script type="text/javascript" src="functions.js"></script>
<audio controls id="audio" autoplay="true"></audio>
</head>
<body>
<script>
    if (typeof (MozWebSocket) == "function") {
        this.socket = new MozWebSocket("ws://150.132.141.36:8080/");
    } else {
        this.socket = new WebSocket("ws://150.132.141.36:8080/");
    }

    var audio = document.getElementById("audio");
    var payload;
    var audio2 = new Audio();
    var audio = document.getElementById("audio");

    function gotAudio(stream) {
        audio.src = URL.createObjectURL(stream);
    }

    function metaData() {
        audio.volume = 0;
        audio2.mozSetup(audio.mozChannels, audio.mozSampleRate);
    }

    navigator.getUserMedia({audio : true}, gotAudio, errorHandler);
    audio.addEventListener('loadedmetadata', metaData, false);
    audio.addEventListener('MozAudioAvailable', requireSamples, false);

    function requireSamples(e) {
        var buffer = e.frameBuffer;
        var string = "";
        for ( var i = 0; i < buffer.length; i++) {
            var enc = encode(buffer[i]);
            string += enc;
        }

        socket.send(string);
    }

    socket.onmessage = function(e) {
        payload = e.data;
        var dec = decode(payload);
        var written = audio2.mozWriteAudio(dec);
    }
</script>
</body>
</html>
```

Acknowledgments

This master's thesis was carried out at Ericsson AB as a part of the degree of Master of Science in Electrical Engineering at Luleå University of Technology. It has been fun and educational to work with this thesis. I would like to thank my supervisor Nicklas Sandgren at Ericsson AB for all the help and support. I would also like to thank Johan Carlson, my supervisor at the University for helping me with this thesis.

References

1. Flanagan, David(2011). JavaScript the definitive guide edition6
2. MozillaWiki, audio data API (2011). https://wiki.mozilla.org/Audio_Data_API
3. World Wide Web Consortium audio group proposal, web audio API (2010). <https://dvcs.w3.org/hg/audio/raw-file/tip/webaudio/specification.html>
4. Bellamy, John (1991). Digital telephony second edition.
5. The WebSocket protocol (2011). <http://tools.ietf.org/html/draft-ietf-hybi-thewebsocketprotocol-03>
6. World Wide Web Consortium, the WebSocket API (2011). <http://dev.w3.org/html5/websockets/>
7. Node, server (2011). <http://nodejs.org/>
8. Ericsson Labs blog WebSocket communication<https://labs.ericsson.com/developer-community/blog/beyond-html5-conversational-voice-and-video-implemented-webkit-gtk>
9. Ericsson Labs blog peer to peer communication <https://labs.ericsson.com/apis/web-real-time-communication/documentation>
10. World Wide Web Consortium, the webRTC specification (2011). <http://dev.w3.org/2011/webrtc/editor/webrtc.html>

Appendices A

A.1HTTP server

```
function runHTTP() {
  var http = require("http");
  var path = require("path");
  var fs = require("fs");
  http.createServer(function(request, response) {
    var filePath = "." + request.url;
    if (filePath == "./")
      filePath = "./client.html";
    var extension = path.extname(filePath);
    var contentType = "text/html";
    switch (extension) {
      case ".jpg":
        contentType = "img/jpg";
        break;
      case ".wav":
        contentType = "audio/wav";
        break;
      case ".ogg":
        contentType = "audio/ogg";
        break;
      case ".mp3":
        contentType = "audio/mp3";
        break;
    }
    path.exists(filePath, function(exists) {
      if (exists) {
        fs.readFile(filePath, function(error, content) {
          if (error) {
            response.writeHead(500);
            response.end("unable to upload");
          } else {
            response.writeHead(200, {
              "Content-Type" : contentType
            });
            response.end(content);
          }
        });
      } else {
        response.writeHead(404);
        response.end();
      }
    });
  }).listen(4000);
}
exports.runHTTP = runHTTP;
```

A.2 WebSocket handshake

```
var crypto = require('crypto');
function handshake(data) {
    var secKey = /Sec-WebSocket-Key: (.*)/g;
    var key = secKey.exec(data);
    var magic = '258EAF55-E914-47DA-95CA-C5AB0DC85B11';
    // create and return SHA-1 hash
    var sha1 = crypto.createHash('sha1');
    // update sha1
    sha1.update(key[1] + magic);
    // calculate the sha1 digest base 64
    var accept = sha1.digest('base64');
    var response = "HTTP/1.1 101 Switching Protocols\r\nUpgrade:
websocket\r\nConnection: Upgrade\r\nSec-WebSocket-Accept:"
        + accept + "\r\n\r\n";
    return response;
}

exports.handshake = handshake;
```

A.3 Unmask data

```
function unmask(data) {
    var l = 0x7f & data[1];
    if (l < 126) {
        var mask = data.slice(2, 6);
        var payload = data.slice(6);
    }
    var unmask = new Array(payload.length);
    for (var i = 0; i < payload.length; i++) {
        unmask[i] = payload[i] ^ mask[i % 4];
    }
    return unmask;
}

exports.unmask = unmask;
```

A.4Bit manipulation

```
var max = 0x7fff; // 0111111111111111
var upscale = 32767;
function encode(pcm) {
    pcm = pcm * upscale;
    var signbit = 0;
    if (pcm < 0) {
        signbit = 1;
        pcm = -pcm;
    }
    if (pcm > max) { // prevent overflow
        pcm = max;
    }
    pcm = (pcm & 0xffff);
    pcm = pcm.toString(2);
    pcm = padLeft(pcm, 15);
    var msb = pcm.slice(0, 12);

    var output;
    var segment = 0;
    var intervall = 0;
    // loop throw 8 bit msb until first 1 is found or count is 0
    for ( var i = 0; i < msb.length - 4; i++) {
        if (msb[i] == 1) {
            segment = 7 - i;
            intervall = msb.slice(i + 1, i + 5);
            break;
        }
    }
    segment = segment.toString(2);
    segment = padLeft(segment, 3);
    intervall = padLeft(intervall, 4);
    signbit = signbit.toString(2);
    output = signbit.concat(segment, intervall);
    return output;
}
```

```

function decode(input) { // 8 bit input
    var array = new Array(1024);
    var sub = input.match(/.{1,8}/g);
    for ( var i = 0; i < sub.length; i++) {
        input = sub[i];
        var signbit = input.slice(0, 1);
        var segment = input.slice(1, 4);
        var intervall = input.slice(4, input.length);
        var output;
        var mask = 0x1;
        segment = parseInt(segment, 2).toString(10);
        if (segment == 0) {
            output = intervall.concat(mask);
            output = padLeft(output, 12);
            output = signbit.concat(output);
            output = padRight(output, 16);
        } else {
            mask <=<= (segment - 1);
            mask = mask.toString(2);
            output = padLeft(1, (8 - segment));
            output = output.concat(intervall);
            output = output.concat(mask);
            output = padRight(output, 15);
            output = signbit.concat(output);
        }
        if (signbit == 1) {
            output = output.slice(1, output.length);
            output = -parseInt(output, 2);
        } else {
            output = parseInt(output, 2);
        }

        array[i] = output / upscale;
    }
    return array;
}

```

A.5 Mozilla audio data capture

```
<audio controls id="audio" src="Location of audio file"></audio>
<script>
    var audio1 = document.getElementById("audio");
    var audio2 = new Audio();
    var buffers = new Array();
    function loadedMetadata() {
        audio.volume = 0;
        audio2.mozSetup(audio1.mozChannels, audio1.mozSampleRate);    }
    function requireSamples(event) {
        var frameBuffer = event.frameBuffer;
        writeAudio(frameBuffer);
    }
    Audio1.addEventListener('MozAudioAvailable', requireSamples, false);
    Audio1.addEventListener('loadedmetadata', loadedMetadata, false);
    function writeAudio(soundBuffer) {
        for ( var i = 0; i < soundBuffer.length; i++) {
            //access samples.
        }
        var written = audio2.mozWriteAudio(buffers);
        if (written < buffers.length) {
            soundBuffer = buffers.slice(written);
            return;
        }
    }
</script>
```

A.6 Node Server

```
var net = require('net');
var unmask = require("./socketcodec.js");
var upgradeToWebSocket = require("./upgradeToWebSocket.js");
var httpServer = require("./httpServer.js");
var clients = [];
httpServer.runHTTP();
var server = net.createServer(function(socket) {

    socket.on("connect", function() {
        clients.push(socket);
        console.log("client connected:" + " " + clients.length);
        console.log("connection" + " " + socket.remoteAddress);
        socket.setTimeout(0);
        // dont buffer data before sending it fire of socket.write
        socket.setNoDelay(true);
    });
    socket.on("data", function(data) {
        // obtain the upgrade header
        var change = /Upgrade: (.*)/g;
        var comp = change.exec(data);
        if (comp != null) {
            // if requesting upgrade, upgrade to WebSocket
            var response = upgradeToWebSocket.handshake(data);
            socket.write(response);
        } else {
            //var sender=socket.remoteAddress;
            var sender=clients.indexOf(socket);
            for (var i = 0; i < clients.length; i++) {
                if (clients[i] != clients[sender]) {
                    clients[i].write(data);
                }
            }
        }
    });
    socket.on("close",
        function() {
            var index = clients.indexOf(socket);
            console.log("connection: ",
clients[index].remoteAddress,
                "terminated");
            clients.splice(index, index + 1);
            console.log("connected clients: ", clients.length);
        });
});

server.listen(port);
```

A.7 Client

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Insert title here</title>
<script type="text/javascript" src="codec3.js"></script>
<script type="text/javascript" src="functions.js"></script>
<audio controls src="grigor.ogg" id="audio"></audio>
</head>
<body>
  <script>
    if (typeof (MozWebSocket) == "function") {
      this.socket = new MozWebSocket("ws://1.1.1.36:8080/");
    } else {
      this.socket = new WebSocket("ws://localhost:8080/");
    }
    var audio = document.getElementById("audio");
    var payload;
    var audio2 = new Audio();
    function metaData() {
      audio.volume = 0;
      audio2.mozSetup(audio.mozChannels, audio.mozSampleRate);
    }
    audio.addEventListener('loadedmetadata', metaData, false);
    audio.addEventListener('MozAudioAvailable', requireSamples,
false);

    function requireSamples(e) {
      var buffer = e.frameBuffer;
      var string = "";
      for ( var i = 0; i < buffer.length; i++) {
        var enc = encode(buffer[i]);
        string += enc;
      }
      socket.send(string);
    }
    socket.onmessage = function(e) {
      payload = e.data;
      var dec = decode(payload);
      var written = audio2.mozWriteAudio(dec);
    }
  </script>
</body>
</html>
```

