Development of a videoconference application with speech recognition features using HTML5 APIs

Luis Villaseñor Muñoz
A20297686
May 12th, 2014
Abstract

This paper describes the development process of a real-time videoconference application with features related to speech recognition including real-time captioning, transcription storage and instant translation.

Among other things, this paper include details on how the WebRTC web API was used for developing a multi-user videoconferencing application and how the HTML5’s Web Speech API was used for providing real-time captioning.
# Table of Contents

Abstract.............................................................................................................................................. 2  
Table of Contents.................................................................................................................................. 3  
1. Introduction....................................................................................................................................... 4  
2. Project description ............................................................................................................................ 5  
2.1. Goal ............................................................................................................................................... 5  
2.2. Milestones ..................................................................................................................................... 5  
3. Requirements .................................................................................................................................... 6  
3.1. Equipment ..................................................................................................................................... 6  
3.2. Node.js ......................................................................................................................................... 7  
3.3. Google Chrome ............................................................................................................................. 7  
4. Development ..................................................................................................................................... 8  
4.1. Web server ..................................................................................................................................... 8  
4.2. index.html ..................................................................................................................................... 9  
4.3. Multi-user WebRTC application .................................................................................................... 9  
4.4. Real-time captioning ..................................................................................................................... 17  
4.5. Transcription storage ................................................................................................................... 19  
4.6. Instant translation .......................................................................................................................... 20  
4.7. Spoken translated subtitles .......................................................................................................... 22  
4.8. Appearance ................................................................................................................................... 22  
5. Next steps ......................................................................................................................................... 24  
6. Conclusions ...................................................................................................................................... 26  
Acknowledgements ................................................................................................................................. 27  
References .............................................................................................................................................. 28  
Appendix ............................................................................................................................................... 30  
A. HTTP server and HTTPS server implementation code ................................................................. 30
1. Introduction

Web technologies have experienced a major development during the last years. The power and versatility demonstrated by these technologies point the browser as the appropriate platform for the development of a new breed of applications. No need of installation, always updated and available worldwide, are some of the advantages inherent to this kind of applications.

Included as one more API of the HTML5 specification, the WebRTC web API enables web browsers with real-time communications capabilities using simple JavaScript. Although WebRTC is still under development, it has already shown a great ability establishing high quality videoconferences. In May 2014, WebRTC is compatible with some of the major browser vendors: Google Chrome, Mozilla Firefox and Opera. In addition, the Android version of these web browsers are also WebRTC capable.

With many WebRTC applications online nowadays, the need of a real-time captioning solution for these applications in order to be used by hearing impaired people was the first motivation for the initiation of this project. Once we explored the Web Speech API and its possibilities, the goals went further, aiming other possible uses as, for example, instant translation.
2. Project description

The project exposed in this paper combines the WebRTC web API, the Web Speech API and other HTML5 APIs in order to obtain a multi-user videoconferencing application able to provide real-time caption and other features related to speech recognition.

2.1. Goal

“To develop a WebRTC multi-user video conference application with some extra features based on speech recognition as real-time captioning, instant translation and transcription storage.”

2.2. Milestones

In order to manage the application development in an efficient way, the next milestones were established:

- Multi-user videoconferencing application
- Real-time captioning
- Transcription storage
- Instant translation

The development of each of these milestones will be explained later in this document.
3. Requirements

The next requirements are needed in order to use the application.

3.1. Equipment

Since the project is a web application, it needs to be hosted in a web server. A computer with internet connection and a public IP are needed.

These are the characteristics of computer used:

<table>
<thead>
<tr>
<th>Processor:</th>
<th>Intel® Core™2 CPU <a href="mailto:6320@1.86GHz">6320@1.86GHz</a> x 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory:</td>
<td>3.8 GiB</td>
</tr>
<tr>
<td>Operating system:</td>
<td>Ubuntu Release 12.04 (precise) 64-bit</td>
</tr>
<tr>
<td></td>
<td>Kernel Linux 3.11.0-20-generic</td>
</tr>
<tr>
<td></td>
<td>GNOME 3.4.2</td>
</tr>
</tbody>
</table>

We chose Ubuntu Desktop 12.04 LTS as operating system. This operating system fits our need of a free operating system with no major compatibility issues and an easy to use interface. Instructions about the download and the installation of this operating system can be found in the official Ubuntu web page[1] listed in the references of this document.

This computer used one connection of the VoIP lab’s 109 network. We used the IP 64.131.109.59, whose host name is dixie11.rice.iit.edu. You can easily know the IP of a Linux machine by executing the `ifconfig` command. The hostname of an IP address can be found out by executing the `host <IP address>` command.
3.2. **Node.js**

The server part of the application will use Node.js, a software platform for server-side and networking applications. These applications are written in JavaScript. Since the WebRTC web API is accessed using Javascript, choosing Node.js gives us the opportunity of write everything in Javascript making easier the communications between client and server. Node.js is available for download at the Node.js website[2].

In addition, we have used socket.io, a Node.js module that enables WebSockets. We use WebSockets for the message exchange between clients and server. Instructions of installation and usage can be found at the Socket.IO webpage[3].

3.3. **Google Chrome**

The clients must use Google Chrome[4] in order to being able to use all the implemented features. Although other browsers are WebRTC capable, Google Chrome is the only one that has implemented the WebSpeech API so far. The WebSpeech API will be the cornerstone for all the speech recognition features.
4. Development

The next sections explain in detail the development process of every functional element that was implemented for the completion of the project.

4.1. Web server

In this section we will explain how the HTTP and HTTPS request are handled. The code of the implementation of both servers can be found in the appendix section A. In the appendix sections B and C there are explained some tools that make easier managing the server.

4.1.1. HTTP server

All the requests received by the HTTP server will be redirected to the HTTPS server. A 301 Moved Permanently response will be provided. The HTTP server will take into consideration the relative path the request was trying to access redirecting the client to the same relative path in the HTTPS server.

- Example:
  
  http://dixie11.rice.iit.edu/room=room1 \rightarrow https://dixie11.rice.iit.edu/room=room1

4.1.2. HTTPS server

The advantage of using a HTTPS server for serving the application is that the application doesn't need to ask the user for permission to use his camera and microphone every time they had to be used. The application only needs to be granted access once to use them every time they are needed.

The HTTPS server will behave as related next:

- If the relative path of the requested address is compliant with the application’s syntax (/room=<room> or /room=<room>&user=<user>) the HTTPS server will provide the room.html file where the WebRTC application will start.
- If the requested file doesn’t exist or it is one of the protected files (SSL private key and SSL certificate) the server will provide a 404 Not Found response and the user will be redirected to an error page.

- Every other existing file requested will be provided along with a 200 OK response.

In order to run the HTTPS server, a SSL certificate is needed. Information about how to get one can be found in appendix section D.

### 4.2. index.html

The index.html page gives the user the opportunity of indicate his username and the name of the room he wants to join with a form. The initial.js script will verify that these fields are properly filled and then it will redirect the user to the room he has asked.

The initial.js script also contains a listener for the enter key so the user can submit the form by pressing this key.

### 4.3. Multi-user WebRTC application

For achieving this milestone we have used the MediaStream API and the RTCPeerConnection API included in the WebRTC web API.

#### 4.3.1. Connection handling

Once the user has downloaded the room.html the main.js script will execute. This script contains the client part of the WebRTC application. The typical application flow will be as follows:

1. The application will check if the user has already specified his username. If it is not specified, the user will be asked for it.
2. The user will establish a WebSocket connection with the server and will request to join the room. All the message exchange between server and clients is made through WebSockets.
3. The server will handle the user request:
   3.1. If the username is already in use in the requested room the user will be asked for a different username. The server keeps a list with every user in every existing room.
   3.2. If the username is not in use and the room doesn’t exist, the room will be created.
   3.3. If the username is not in use and the room exists, the user will join the room and the rest of users in the room will be notified that a new user has joined the room.

4. When a user joins a room the application will proceed to get his local stream using the MediaStream API.

5. When a user receives notice of a new user, he will create a new peer connection element, he will attach his local stream to it, and will wait for the other user to start the offer/answer exchange.

6. After joining a room and getting his local stream ready, if there are other users in the room, the user will create a peer connection element for every one of them and will start the offer/answer protocol in order to establish a peer to peer connection with each of them.

The following ladder diagram exemplify the most important points of the application flow for establishing a call between two users:
Once the connection phase is over there will be a peer to peer connection between each pair of users present in one room, resulting in a mesh network. In addition, we can have several different rooms simultaneously, so we can have more than one mesh. The next figure exemplify the situation of two rooms with 6 users in each room.

- **Limits:**

  Although proper measures of this fact are needed, we have observed that very few resources were used by the server for maintaining the WebSocket connections alive. Neither the processor usage, the memory usage or the bandwidth usage are important, so we think that a great number of WebSocket connections can be maintained at the same time with our current server.

### 4.3.2. MediaStream API

We use the MediaStream API in order to gain access to the user's camera and microphone. As stated in the W3C Editor's Draft titled Media Capture and Streams[5], the MediaStream interface is used to represent streams of media data, typically (but not necessarily) of audio and/or video content.
• **Usage:**

For obtaining the user's MediaStream we use the following code:

```
navigator.getUserMedia(constraints, successCallback, errorCallback);
```

Where:

- **constraints:** This variable allow us to indicate constraints on the MediaStreamTracks we want to obtain. In our case the value of this variable is: `{video: true, audio: true}`

- **successCallback:** This parameter indicates the function that will be called if the getUserMedia request is successful. In our case the local video will be attached to the HTML video element located in the local user area with that purpose and the user will start calling the rest of the users that have already joined the room.

- **errorCallback:** This parameter indicates which function will be called if the getUserMedia request fails. In this case, the user will be alerted about the error.

If the request is successful, we will obtain a MediaStream object as the one represented in the picture below. All the MediaStreamTracks inside a MediaStream object are automatically synchronized.

![MediaStream Diagram](image_url)

*[6] Figure by Justin Uberti and Sam Dutton.*
4.3.3. RTCPeerConnection API

We use the RTCPeerConnection API for establishing peer to peer connections between users. This API is specifically designed for establishing audio and video conferences. It is almost transparent for the programmer. Some of their built-in duties are:

- Connecting to remote peers using NAT-traversal technologies such as ICE, STUN, and TURN.
- Managing the media engines (codecs, echo cancelation, noise reduction...).
- Sending the locally-produced streams to remote peers and receiving streams from remote peers.
- Sending arbitrary data directly to remote peers.
- Taking care of the security, using the most appropriate secure protocol for each of the WebRTC tasks. It uses HTTPS for the signaling, Secure RTP for the media and DTLs for data channel.

A diagram showing some of the RTCPeerConnection features and how they are accessed is next:

[7] Figure from WebRTC.org.
More technical details about this API can be found in the next documents:
- WebRTC 1.0: Real-time Communication Between Browsers[8].
- Javascript Session Establishment Protocol[9].

**Usage:**

During the connection phase the user will create an RTCPeerConnection object for every user in the room. All these objects are stored in a JSON object using as key the username of the user for who the object has been created.

When creating the RTCPeerConnection objects we specify the application's behavior for each of the next events:

- **onaddstream**: when the remote stream is added we will create dynamically all the HTML objects required for displaying the remote user's video and his subtitles. We will assign custom HTML id tags to each of these elements so we can recover them again when necessary.

- **onremovestream**: when remote stream is removed we will recover the HTML elements that were used for displaying this stream using the custom id assigned in the onaddstream event and we will remove them from the view.

- **onicecandidate**: when a user receives a new ICE candidate it will be sent to the remote user though the signaling server using WebSockets.

- **ondatachannel**: if the data channel is created by the remote user this event will be triggered. The local user will set the data channel up and store it for later use. We will send the subtitles through the data channel.

In addition, when creating the RTCPeerConnection objects we also specify, if any, the STUN and TURN servers the application will be using for ICE. In our case, depending on which browser is the user using we will choose what server to use between Google's STUN, Mozilla's STUN and VoIP lab's STUN. In any case, we will also use the VoIP lab's TURN for solving difficult connectivity issues caused by NATs.
4.3.4. ICE / STUN / TURN

ICE, STUN and TURN are different mechanisms for obtaining possible addresses where a peer can contact another peer. As stated in the RFC 5245[12], ICE is an extension to the offer/answer model, and works by including a multiplicity of IP addresses and ports in SDP offers and answers, which are then tested for connectivity by peer-to-peer connectivity checks. The IP addresses and ports included in the SDP and the connectivity checks are performed using the Session Traversal Utilities for NAT (STUN) protocol and its extension, Traversal Using Relay NAT (TURN). ICE can be used by any protocol utilizing the offer/answer model, such as the Session Initiation Protocol (SIP).

While STUN works most of the times, in some very difficult situations TURN is the only option. TURN enables the communication between two users that can’t find each other because of NATs by relaying their media. This is very expensive in system resources. In addition, there are some security flaws.

- **TURN server’s performance**

These are the characteristics of the computer used as TURN server:

<table>
<thead>
<tr>
<th>Processor:</th>
<th>Intel® Core™2 Quad CPU <a href="mailto:Q9300@2.5GHz">Q9300@2.5GHz</a> x 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory:</td>
<td>7.6 GiB</td>
</tr>
<tr>
<td>Operating system:</td>
<td>CentOS Release 6.3 (Final)</td>
</tr>
<tr>
<td></td>
<td>Kernel Linux 2.6.32-279.el.x86_64</td>
</tr>
<tr>
<td></td>
<td>GNOME 2.28.2</td>
</tr>
</tbody>
</table>

Although the computer used as TURN server is much better than the one used as web server, we have observed that due to the amount of load the TURN server has to deal with, makes this computer insufficient. For instance, if 2 clients need the TURN server for communicating between them the TURN server will be dealing with 4 video streams (1 upstream and 1 downstream for each client). If 3 clients need to use the TURN server for communicating between them, the TURN server will be dealing with 12 video streams (2 upstream and 2 downstream for each client). This is too much load and only 3 clients are using it. The video the clients received from the TURN server will be low quality and it will freeze. The TURN server doesn’t scale well.
• **TURN server’s security**

  In order to get access and use the TURN server a password is needed. Since it is the client the one who should use the TURN server for relaying his media, this password should be in the client side. That means that the password can be easily found by any user that enters the application. This fact is a major security flaw. The TURN server password should be assigned dynamically somehow in order to improve the security.

• **Reused code:**

  I would like to put on record that I have used code from the WebRTC tutorial[13] written by Sam Dutton for the process of the SDPs during the offer/answer exchange. I followed his tutorial when I started with WebRTC and it didn't make much sense to me to re-implement this part in a different way since there are no many other different ways of implementing that. The functions taken from that code are: mergeConstraint, preferOpus, extractSdp, setDefaultCodec and removeCN.

### 4.3.5. Disconnection handling

The server knows at every moment the state of every client's connection thanks to Socket.IO. In case a user closes the application's tab, the server will be informed and it will alert about it to the rest of the users that remains in the room in which the user that closed the application was.

When a client is alerted about another client's disconnection, all the HTML elements that were used for displaying the disconnected user are removed from the application view dynamically. In addition, all the variables related to the disconnected user are removed: RTCPeerConnection object and RTCDataChannel object.
4.4. **Real-time captioning**

For achieving the Real-time captioning we will use the WebSpeech API for converting the user's voice into text and the WebRTC's data channel for sending the text (subtitles) to the remote user that is requesting them.

Although the last WebSpeech API specification[10] dates October 2012, Google Chrome browser is the only browser that supports it. All the features related to speech recognition exposed in this paper don't work in any other browser at the moment. Chrome uses the same speech recognition service that other Google’s products as the Android devices or the Google Glasses use.

4.4.1. **SpeechRecognition interface**

The WebSpeech API is composed by 2 interfaces: the SpeechRecognition interface, used for converting speech to text, and the SpeechSynthesis interface, used for turning text to speech. The SpeechRecognition interface will be the cornerstone of the real-time captioning feature implemented in our application.

- **Usage:**

  As soon as a user joins a room, the application will request his permission for accessing the camera and the microphone. Since the application is hosted in a HTTPS server the application won't need to ask the user for permission again. This enable us to switch on or switch off the speech recognition feature without requesting the user for permission to access the microphone again.

  We will use the WebSocket connections to redirect remote users' requests for subtitles to the local user. Once a request for subtitles is received, the speech recognition will be switched on without the local user intervention. In order to save system resources and bandwidth, the subtitles will only be generated if a remote user is requesting them.

  The application will take the browser's default language as the default language for speech recognition. The user can modify the language used for speech recognition selecting the desired language inside the dropdown located at the left of the screen. Only some of the most common world languages have been included in the list in order to simplify the implementation. More languages can be added easily.
After the speech recognition is turned on, the user's voice will be automatically sent to Google's speech recognition service.

The application requests interim results. This means that the user will start receiving his transcribed speech even before of finishing the current phrase. Thanks to this set up, the remote user will feel the subtitles as real-time.

The speech recognition feature is event driven. The onresult event will handle the results of the speech transcription. The results are JSON objects with a list of possible matches. We will take the most probable of these possible matches (the first one), and we will send it to the users that are requesting subtitles using the data channel.

The results obtained contain an isFinal property that indicates that the phrase is completed. The application sends this property along with the subtitles to the receiving user in order to let him know if it is a final result or if it is just another interim result. The subtitle's text and the isFinal property are encapsulated in a JSON object in order to be sent as text though the data channel.

The application has been implemented to keep the speech recognition alive while someone is requesting subtitles, keeping track of all the users that are requesting subtitles anytime. Although the speech recognition has been set up for requesting continuous speech recognition (recognition.continuous = true), Google's server will end the speech recognition eventually. The onend event defined in the application will call the keepSpeechRecognitionAliveIf Needed() function which will switch on the speech recognition again if needed.

### 4.4.2. RTCDataChannel

At the same time that the application created a RTCPeerConnection element for every user in the room, an RTCDataChannel element was also created for each of them. All these elements are also stored in a JSON object using the remote user's username as key.

We send text though the data channel using the next syntax:

```javascript
dataChannel.send('text');
```

The RTCDataChannel interface is also event driven. The onmessage event will be triggered when a subtitle is received. Then, application will write the subtitle in the caption space located at the inferior part of the remote user's video element.
4.4.3. Architecture

The next diagram represent the typical situation in which a user (User A) request subtitles from other user (User B):

4.5. Transcription storage

The aim of this feature is to store locally, in text format, anything said by anybody in a room. We use the IndexedDB API for achieving this. The IndexedDB API is also part of HTML5. We can create simple and easy to use databases using this API. For our application, when the users use it for the first time, a database with the next columns is created:

| id | date | room | user | text |

The database can easily be reviewed under the Resources tab of the Google Chrome's Developer Tools when accessing the application.
• **Usage:**

The user can turn on or off the transcription storage feature by clicking the On and Off buttons placed at the left of the screen. When the transcription storage feature is turned on the application will request subtitles to all the users in the room and it will also start the local user speech recognition.

When receiving subtitles from the remote users the application will check if the transcription storage is enabled. If it is enabled, the application will check if the received subtitle is a final subtitle before saving it in the database along with the user that said it, the room, the date and a unique id that is used as the database's primary key. Checking the isFinal subtitle's property makes possible to store once every phrase instead of saving each interim result.

The transcription of the local speech goes through a similar process when it is received from the speech recognition service.

For retrieving the stored transcriptions, the user have to click the "Browse stored transcription link". This link will open in a new tab so, in case there is any conference in progress, the conference won't finish. The transcriptionStorage.js script will display all the data stored inside the local database in a table.

**4.6. Instant translation**

The application will translate the requested subtitles from the originating user's language to the terminating user's language using an online translation service.

**4.6.1. Microsoft Translator**

Translation APIs are not free. Microsoft Translator[15] is the only one which offers some characters for free. However, since these free characters are limited to 2.000.000 per month, we decide to only translate the final results of the speech recognition service. This decision make the translation feature to be slower and we cannot considered it real-time anymore. However, if we request translation for the interim results we will obtain a really good user experience in terms of quickness and we could consider it is real-time translation.
In order to use the Microsoft Translator API I had to register a developer account. They gave me a password for using the translation service. In order to keep this password secret, it is stored in the server side. Because of this, all the translations request must go through the Node JS server. So, in case of requesting translated subtitles, they will go through the server instead of going through the data channel. A figure explaining this scenario is included in section 4.6.3.

4.6.2. Node module

In order to simplify the server side code, since there is no official Javascript API for Microsoft Translator, I have used a node JS module developed by Kenan Shifflett called mstranslator[11] and that works as a Javascript API for Microsoft Translator.

4.6.3. Architecture

The next figure illustrates the situation in which User A request translated subtitles from User B. Notice that in order to translate the subtitles from the originating user’s language to the terminating user’s language we need to specify these language in some of the messages exchanged between clients and server.
4.7. Spoken translated subtitles

Once we have the subtitles translated, the next step will consist in saying them aloud using the text to speech feature included in the WebSpeech API.

4.7.1. SpeechSynthesis interface

Using a similar procedure than the one we use for the transcription storage, if the spoken translated subtitles feature is enabled, the application will check the isFinal property of the subtitles it receives. If the isFinal property is true, the application will read the subtitle aloud.

Chrome version 34, the most up-to-date Chrome's version right now, have 9 built-in speech synthesis voices. The languages supported are English, Spanish, French, Italian, German, Japanese, Korean and Chinese. In case the local user’s language is not among the supported ones the subtitles wouldn’t be read.

A well explained tutorial called Web apps that talk - Introduction to the Speech Synthesis API[16] about how to use the SpeechSynthesis interface can be found between the rest of the references of this document.

4.8. Appearance

CSS3 was used for obtaining the appearance of the application. However, I would like to remark a couple of facts.

4.8.1. Uikit front-end framework

A front-end framework was used in order to obtain good looking and functional user interface elements easily. More information about how to obtain and use this framework is available at the Uikit webpage[17].

There are a lot of front-end frameworks freely available on the Internet. A list[18] of them can be found in the references section.
4.8.2. `manageRemoteAreasClassNames()` function

Every time a user connects or disconnects from a room, the `manageRemoteAreasClassNames()` function will be called. This function is in charge of adapting the size of the video elements in which the remote users are displayed depending on how many of them are actually in the room.
5. Next steps

5.1. Solve TURN server issues

The security issue with the TURN server should be solved. Maybe a system that provides dynamic access to the TURN server using tokens exchange can solve the problem. The users that can access the TURN server should be created and destroyed dynamically.

Concerning the TURN server’s performance I think the only way of improving it is using more powerful computers and/or more than one computer and balance the traffic between the available resources. However, all these solutions are expensive and they don’t scale well anyway.

5.2. Measure application performance

Having data about the application performance would be really useful. Some of these measurements could be:

- Maximum number of concurrent WebSocket connections that the web server can handle.

- TURN server’s resources used per client and maximum number of clients that the TURN server can handled without degrading the conference quality too much.

- Maximum number of concurrent users in a conference that can be supported by an average client computer.
5.3. Develop related applications

Other applications related to speech recognition can be developed. Some ideas for this applications could be:

- Connect the application to the PSTN in order to receive calls from the PSTN and generate subtitles for them.

- Connect the application to other lab's testbeds as the NG911 testbed.

- Develop other ideas to make this technology useful. For example, an online platform where people that is learning languages can meet for practicing each other's language with the help of real-time captioning.
6. Conclusion

The project result is pretty nice. All the goals were achieved and the application can actually be used by real users. Right now it is accessible at https://dixie11.rice.iit.edu.

However, the application is not perfect. Some of its more important flaws are the TURN server limitations, the speech recognition service accuracy and the translation feature's delay.

But we have to take into consideration that this project was developed by a single information technology student, using free resources and technologies that are still under development. I’m sure about that if we give these technologies a little more time and if we use the proper resources we will be enjoying real-time captioning for videoconference really soon and with a really good quality.
Acknowledgements

I would like to thank the next people for their help with this project. It wouldn’t have been possible without them.

Carol Davids, for her wise guidance and advice. Also, for giving me the opportunity to present this project at the Student Presentations and Demonstrations event that was held on April 30th, 2014.

Tania Arenas de la Rubia, for her cool designs, including BaBL’s background and logo.

Javier Conde Monteoliva and Miguel Camacho Ruiz, for their priceless help about programming and WebRTC.

Elias Yousef, for letting me use the TURN server he was working on.

Don Monte and Nishant Agrawal, for the help with the hardware and the network connection.

My family and friends, for their support.

Thank you.
References


Appendix

A. HTTP server and HTTPS server implementation code

////////////////////////////////////////////////////////////////////////////////////////
//WEB SERVER:

// DEPENDENCIES:
var http = require('http');
var https = require('https');
var fs = require('fs');
var static = require('node-static');
var file = new static.Server();

//SSL CERTIFICATE:
var httpsOptions = {
    key: fs.readFileSync('dixie11.key'),
    cert: fs.readFileSync('dixie11_rice_iit_edu.crt')
};

//HTTPS WEB SERVER:
var HTTPSWebServer = https.createServer(httpsOptions, function(req, res) {
    //Serve room requests
    if (req.url.substring(1, 6) === 'room=' && req.url.indexOf('&username=') !== 6) {
        file.sendFile('/room.html', 200, {}, req, res);
    //Serve error404.html for hidden files
    } else if (req.url === '/dixie11.key' || req.url === 'dixie11_rice_iit_edu.crt') {
        file.sendFile('/error404.html', 404, {}, req, res);
    //Serve the rest of the files and handles 404 errors
    } else {
        file.serve(req, res, function(error, errorRes) {
            if (error && (error.status === 404)) {
                file.sendFile('/error404.html', 404, {}, req, res);
            }
        });
    }
}).listen(443);
// HTTP WEB SERVER: Redirects all traffic to HTTPS server
var HTTPWebServer = http.createServer(function(req, res) {
    res.writeHead(301, {'Location': 'https://dixie11.rice.iit.edu' + req.url});
    res.end();
}).listen(80);

**B. Forever**

Forever is the name of a Node JS module that will restart the application in case it is closed unexpectedly.

In order to enable it we start our application using the next command in the folder where the server.js file is located:

```bash
sudo forever start -l babl.log server.js
```

babl.log will be log file and it will stored all the console's outputs of the application. It is located inside the .forever directory located at the user's home directory.

More information about installation and usage of Forever can be found at its repository in GitHub[22].

**C. Remote access**

For accessing and managing the server remotely we used the next tools:

- Putty for Windows.
- WinSCP for Windows.
- JuiceSSH for Android.

**D. SSL certificate**

In order to run a HTTPS server we need to obtain a SSL certificate. Good information about how create it and the possible options to sign it can be found in the next article:

- Creating SSL keys, CSRs, self-signed certificates, and .pem files[23].

For our application, we used a Comodo’s 90 days free trial in order to avoid the browser’s warning about being a non-trusted application during this period of time.
E. Application screenshot