Web-based Real-Time Communication for Rescue Robots

by

Akaitz Gallastegi Garcia

LIU-IDA/LITH-EX-G--14/075--SE

2014-07-03
Web-based real-time communication for rescue robots

Akaitz Gallastegi Garcia

2014-07-03

ISRN LIU-IDA/LITH-EX-G--14/075--SE
In this thesis an audio and video streaming system is implemented for its use in rescue robots. WebRTC technology is used in order to stream in real time. Implemented in an architecture based on a Web server, two pages running WebRTC and a TURN\textsuperscript{1}-STUN\textsuperscript{2} server, the system has been tested in terms of CPU and bandwidth utilization. Measurements show that when WebRTC is run in an Intel Core i3, less than 10\% of CPU is used, whereas on smaller tablets the performance is not enough for running the application with the desired quality of service.

\textsuperscript{1}Traversal Using Relays around NAT
\textsuperscript{2}Session Traversal Utilities for NAT
# Table of contents

1. **Introduction** ........................................................................................................... 6
   1.1 **Background** .......................................................................................................... 6
   1.2 **Main goal** ............................................................................................................... 7
   1.3 **Methodology** ......................................................................................................... 7
   1.4 **Structure of the report** ......................................................................................... 8
2. **Solution approach** .................................................................................................... 9
   2.1 **Technological solutions** ....................................................................................... 9
      2.1.1 **Selected technology** .................................................................................... 9
   2.2 **Proposed solutions** ............................................................................................. 11
      2.2.1 **Proposed solution 1** .................................................................................... 11
      2.2.2 **Proposed solution 2** .................................................................................... 12
      2.2.3 **Selected solution** ........................................................................................ 13
3. **Implemented solution** .............................................................................................. 14
   3.1 **Server** .................................................................................................................. 14
   3.2 **Web pages** ........................................................................................................... 15
      3.2.1 **Web pages visual design** .............................................................................. 15
      3.2.2 **Socket usage** ............................................................................................... 16
      3.2.3 **Media exchange** .......................................................................................... 18
4. **Testing** ...................................................................................................................... 20
   4.1 **Bandwidth utilization test** .................................................................................... 20
   4.2 **CPU utilization** .................................................................................................... 20
      4.2.1 **Four robots running WebRTC application and robot_agent** ......................... 20
      4.2.2 **Four robots running WebRTC application only** ............................................ 22
      4.2.3 **One robot running WebRTC application** ....................................................... 22
      4.2.4 **More powerful laptop running WebRTC application** ................................... 23
5. **Conclusions and future improvements** ................................................................... 24
6. **Appendices** .............................................................................................................. 25
   6.1 **Instructions to install the software needed to start-up the application** ................ 25
   6.2 **Instructions to run the system** ............................................................................. 26
References ........................................................................................................................ 27
List of figures
Figure 1: Project environment ................................................................. 6
Figure 2: WebRTC application ................................................................ 10
Figure 3: Proposed solution 1 ................................................................. 12
Figure 4: Proposed solution 2 ................................................................. 13
Figure 5: How WebRTC works for media exchange in the proposed solution ................................................................. 18
Figure 6: Average CPU utilization .......................................................... 21
Figure 7: WebRTC application average CPU utilization when 4 robots are running together ......................... 21
Figure 8: robot_agent average CPU utilization when 4 robots are running together ......................................... 22
Figure 9: WebRTC application average CPU utilization when 4 robots run together only WebRTC .......... 22
Figure 10: WebRTC average CPU utilization when only one robot runWebRTC application ...................... 23
Figure 11: WebRTC application CPU utilization when is run in a more powerful laptop ........................................... 23
1 Introduction
This thesis work is in partial fulfilment of a Bachelor of computer engineering (Web-based Real-Time Communication for Rescue Robots) at Mondragon University. The work was performed as a 16 ECTS point project at the Department of Computer and Information Science at Linköping University, in the context of a lab development project for real-time systems course.

In this chapter the background and the main goal of the project are presented.

1.1 Background
Nowadays robots are helpful devices in the search and rescue tasks of victims in disaster situations. These robots can be improved if they were added the capacity of sending images in real-time. These images would be transmitted to a central server controlled by a user which would be responsible for monitoring the received images. The problem in disaster areas is that the available network resources are not uniform. This is why the video quality should be adjustable, in other words, that the system must be adaptable in terms of Quality of Service (QoS). Before being able to adapt the service has to exist in a non-adaptive version.

This project has been developed in a emulated environment as it is used in real-time systems course laboratories[1]. This simulated environment is composed of:

- A number of robots that are composed of a robot chassis (in that case iRobot Create, which is a robotics development platform intended for educational use), a laptop which runs the code that controls the robot, and a Radio Frequency IDentification (RFID) reader, which reads the RFID tags that are spread over the area.

- A workstation that runs a monitor application that controls the mission and starts/stops the robots.

- A closed area with a number of RFID tags. These tags represent position markers or victims.

![Figure 1: Project environment](https://www.ida.liu.se/~TDDD07/labs/TDDD07_RoboLab_Rescue_Compendium_2013.pdf)

Each robot runs a robot agent software. This software is responsible for making the robot operate according to its specific mission and it has several tasks scheduled. In the following list the tasks that the robot agent carries out are shown:

- **Control task:** this task controls the robot and determines its current location based on few sensors. This consists of three activities, determine the location update based on odometry data, calculate a

---

3 Figure taken from [https://www.ida.liu.se/~TDDD07/labs/TDDD07_RoboLab_Rescue_Compendium_2013.pdf](https://www.ida.liu.se/~TDDD07/labs/TDDD07_RoboLab_Rescue_Compendium_2013.pdf)
new trajectory based on the current location and the information provided by the Navigate task and send the motor commands to the robot chassis.

- **Navigate task**: this task is the one responsible for determining the path in which the robot should move.
- **Mission task**: this task controls the overall mission of the robot, which is to search for victims and report found victims to the command & control monitor. It also decides when to start and stop other tasks.
- **Refine task**: this task uses the RFID reader to refine the robot positioning based on any position marker tags that may have been encountered.
- **Report task**: it recognizes a found victim noted by the refine task when encountering a RFID tag that does not have a position and deduces it is a victim. The robot uses own position to report the victim, and checks whether that victim has already been previously reported.
- **Avoid task**: this task is responsible for avoiding that the robot gets stuck when running into obstacles.
- **Communicate task**: this task is responsible for receiving messages from other robots and for transmitting messages generated by other tasks towards the command & control monitor, using a dedicated wireless communication medium (WiFi).

This software is run as a single process in the robot laptop.

Robots send messages using TDMA, a protocol that avoids collisions. Transmission time is divided into slots. Each slot is at least the amount of time needed to send a message. Each slot is assigned to a robot, and the robot can only send messages in that slot.

### 1.2 Main goal

The main goal of this project is to **develop an application that allows a robot to transmit video and voice from the rescue area to the remote command & control monitor that monitors the robot activity**. This remote server will display on a screen what it is receiving in real-time. The video stream will be sent over the same wireless channel that the rest of the scenario messages are transmitted.

### 1.3 Methodology

In the development of this project the following methodology was followed: first of all different technologies were analyzed in order to select the most suitable one for the lab environment. Due to limited time to perform the project only 3 possible technologies were selected to study. Those are:

- **LiveStream**
- **UStream**
- **WebRTC**

LiveStream and UStream technologies were selected because they are commonly used in streaming applications and WebRTC because is a new technology with promising remarks on the Internet.

After that, different solutions were proposed and the most suitable one given the limitations the project was selected. This solution has been developed in a emulated environment.

---

4 Time Division Multiple Access
Finally after the implementation, the solution has been tested in order to know whether the project lab requirements are satisfied, that is if the solution works satisfactory given the available network resources.

1.4 Structure of the report
The rest of the report is structured as follows. In chapter 2 different solution approaches are identified. Chapter 3 presents how the implemented solution is structured. Chapter 4 shows the results of different tests that the implemented solution was subject to. Finally, chapter 5 shows the main conclusions and the future work of the project.
2 Solution approach
In this chapter different solutions are proposed and compared in order to choose the most suitable one for this project. First off all, different technologies are analyzed in order to choose the best one for the project context. Afterwards the selected technology will be described more deeply. Finally according to the selected technology different solution approaches will be proposed for the project and the most suitable one will be selected.

2.1 Technological solutions
For the purpose of this project there exist some video transmission technologies that may be relevant to consider, such as:

- **WebRTC**: API framework for real-time communication in Web applications.  
- **LiveStream**: live streaming video platform that allows users to view and broadcast content through Internet. It offers APIs so that Web applications can use its services.  
- **UStream**: as LiveStream, this is a live streaming video platform and it also offers APIs to use its services in Web applications.

In table 1 each technology's advantages and disadvantages are listed:

<table>
<thead>
<tr>
<th></th>
<th>WebRTC</th>
<th>LiveStream</th>
<th>UStream</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Advantages</strong></td>
<td>- Open source</td>
<td>- Service gives support</td>
<td>- Service gives support</td>
</tr>
<tr>
<td></td>
<td>- No need of having Internet connection to use the application</td>
<td>- Good audio streaming (320kbps) even with poor video quality</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- No need of registration in an online service for using it</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Disadvantages</strong></td>
<td>- Not supported by all web browsers (only Google Chrome, Firefox and Opera)</td>
<td>- Internet connection is required to use the application</td>
<td>- Internet connection is required to use the application</td>
</tr>
<tr>
<td></td>
<td>- It is still in a development phase, so there is not so much support</td>
<td>- Registration is needed in order to use it</td>
<td>- Registration is needed in order to use it</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Based on Flash so it is not supported by all Web browsers</td>
<td>- Based on Flash so it is not supported by all Web browsers</td>
</tr>
</tbody>
</table>

Table 1: Advantages and disadvantages of streaming technology

2.1.1 Selected technology
After analysing each technology's advantages and disadvantages, it has been decided that WebRTC is the most suitable technology for this project. These are the main reasons:

- It's Open Source
- It is not necessary to register in an online service for using it
- There is no need of having Internet connection to use the application. This is a very important reason, as this project is going to be developed in an emulated environment with a dedicated Wi-Fi without connection to the external Internet. This is needed for controlled studies of resources used by the students in the course.

---

7 [http://www.ustream.tv/](http://www.ustream.tv/)
WebRTC is an API definition drafted by the World Wide Web Consortium (W3C) to enable end-to-end browser communication without using any plug-in. This browser-to-browser connection can be comprised of an audio stream, video stream and/or data channel. WebRTC uses SRTP\(^8\) for media transmission and ICE\(^9\) for traversal through NAT’s and firewalls\([2][3]\). Some more features of this technology are explained in\([4]\):

- It provides APIs and access rules for end-user devices such as microphones, cameras etc.
- An end-to-end security architecture and protocol is given. It uses SRTP.
- NAT transversal techniques for peer connectivity are implemented.
- Signaling mechanisms for setting up, updating and tearing down the sessions.
- Support for different media types is given.
- Media transport requirements.
- Quality of Service, congestion control and reliability requirements for the session over the Best-Effort Internet is provided.
- Identity architecture and mechanisms for peer identification are provided.
- Codec for audio and video compression.
- HTML and JavaScript APIs for use by application developers are provided.

Note that all network users obtain best-effort service, meaning that they obtain unspecified variable bit rate and delivery time, depending on the current traffic load and there are no guarantees regarding packet delivery. This service is the best one for real-time applications because the loss of small percentage of packets is tolerable. \([5]\)

For a full overview of WebRTC, the reference\([6]\) could be visited.

Figure 2 shows how a WebRTC application works:

\(^8\) Secure Real-Time Transport Protocol
\(^9\) Internet Communications Engine
In this project, the *getUserMedia* and *PeerConnection APIs* are useful. These two API give the programmer the following functionalities:

- **getUserMedia API**: defines application requirements to access end-users media sources.
- **PeerConnection API**: specifies Session Description Protocol (SDP)-based session description APIs and the state machine to update session setup and tear-down between peers.

In our context, *DataChannel API* is not useful because this API provides functionalities to exchange data between peers.

In the case of this project the peers are the robots and the command & control monitor.

### 2.2 Proposed solutions

#### 2.2.1 Proposed solution 1

The basic idea for this solution is to create a new process that runs in parallel to the process that is already implemented in the robot. This process will take control of both the camera and the microphone, and stream the audio and video that it gets while working. This will be implemented in native C++ using the WebRTC suite.

The process will communicate with the command & control monitor. This command & control monitor will have a browser application, with the objective of reproducing the video and audio that the robot sends. The browser will be implemented as a WebRTC application and will be developed in Qt\(^{11}\), in the same way as the monitor. QT supports Webkit\(^{12}\) (an API for developing browsers) and WebRTC.

This way the robot will transmit video and audio, and the command & control monitor will receive and show this video and audio.

Figure 3 shows the main idea of this solution proposal. Robot agent software in each robot and the monitor application in the command and control monitor are already implemented so then red boxes containing WebRTC processes and the Web browser have to be implemented.

---

\(^{11}\)http://qt-project.org/

\(^{12}\)http://www.webkit.org/
2.2.2 Proposed solution 2

The basic idea for this solution is to build-up a server with a web page that will be accessed by both the robot and the command & control monitor using the web-browser installed on the same machine.

This web page will run the webRTC APIs (as is stated above, getUserMedia and PeerConnection) and will show the media sent from the robots and show it to the user that is responsible for the command & control monitor.

The server will be built-up with node.js\(^\text{13}\), socket.IO\(^\text{14}\) and node-static. Node.js is a software platform for scalable server-side and networking applications. The applications are written in JavaScript. Socket.IO is a JavaScript library for real-time web applications and it has two parts: a client-side library which runs in the browser and a server-side library for node.js. Node-static is a HTTP static-file server module for node.js.

Since this project will be developed in an emulated environment with a dedicated Wi-Fi without connection to the external Internet there is a need to setup a TURN-STUN server for the ICE candidate exchange which is compulsory in WebRTC. A TURN-STUN Server is a VoIP media traffic NAT traversal server and gateway, and ICE candidate provides the information about the IP address and port from where the data is going to be

\(^{13}\text{http://nodejs.org/}\)
\(^{14}\text{http://socket.io/}\)
exchanged. This TURN-STUN server will be rfc5766-turn-server\textsuperscript{15}, a TURN-STUN server developed by Google. This server will be installed with the Web server in the same machine, in the command & control monitor.

Figure 4 shows the main idea of this solution. Robot Agent Software in each robot, the monitor application in the command and control monitor, and the web browser in both of them are already implemented, so red boxes containing web server and TURN-STUN server have to be implemented. The web server will include the WebRTC APIs that capture the video and audio of the robots.

\textbf{2.2.3 Selected solution}

As both of the solutions offer the same functionality and taking into account the duration of the project (10 weeks) and the knowledge about programming in C++ (language in which the WebRTC native application is built), the chosen solution for this project is the second one.

\textsuperscript{15}https://code.google.com/p/rfc5766-turn-server/
3 Implemented solution

In this chapter the implementation of the selected solution will be explained. First of all the server that will be used is going to be described and after that the developed Web pages will be explained. The interaction between robots and command & control monitor will be described in detail while explaining Web pages. Also how the WebRTC acts will be described.

In Section 3.1 the Web server in the red box allocated in the command & control monitor will be explained, and in Section 3.2 the Web pages that resides in this Web server will be explained.

Note that the listings in this chapter are adaptations of open source code from [7] and reproduced here to make the solution understandable.

3.1 Server

As it is stated in chapter 2 the Web server of the project will be build-up using node.js, socket.IO and node-static. This will be an HTTP static-file server that will be listening in a socket on port 2200[8]. Below is the code for creating and starting-up the project’s server:

```javascript
var static = require('node-static');
var http = require('http');
var server = new(static.Server)();
//Server startup
var service = http.createServer(function (req, res){
  server.serve(req,res);
}).listen(2200);
//Socket creation
vario = require('socket.io').listen(service);
```

This socket will be listening to events sent by the client sockets. Below the events that this server is going to handle are listed:

- **Create or join a room:**
  First of all it is necessary to explain what a room is. Room is the concept of socket.IO that allows simple partitioning of the connected clients. In this project context, rooms are used to create like chat rooms, so the clients joined to that room could talk to each other (in this case send or receive audio and video from each other). Each room can only accept two clients: one robot and the command and control monitor. So the command and control monitor has to connect to several rooms to receive each robot's video and audio. The reason that each room can only accept 2 clients is that doing this we open a dedicated channel for exchanging media between command & control monitor and each robot.
  When the server receives create or join event it checks how many clients are connected to this room. If the number of connected clients is 0, the server joins the client to that room and sends a created event to the client socket. However, if the number of clients is 1, it joins the client to the room, it sends a join event to the other client connected to the room and it sends a joined event to the client. And finally if the number of connected clients is bigger than one it sends a full event to the client.

- **Message:** when the server receives a message event it sends the message attached to the event to all clients connected to the room except to the sender.

Below is the code for handling events in the server:

```javascript
//Socket event handling
io.sockets.on('connection', function (socket) {
```

function log() {
    var array = [">>>Message from server: "];  
    for (vari = 0; vari < arguments.length; i++) {
        array.push(arguments[i]);
    }  
    socket.emit('log', array);
}

// When it receives a message, it resends that message to all clients connected to that socket
socket.on('message', function (message, room) {
    log('Got message: ', message);
    io.sockets.in(room).emit('message', message);
});

// When it receives an event to create or join to a room, it looks if the room is created or if the number of clients in that room is exceeded.
socket.on('create or join', function (room) {
    var numClients = io.sockets.clients(room).length;
    log('Room ' + room + ' has ' + numClients + ' client(s)');
    log('Request to create or join room', room);

    // If the room is not created, it joins to it and it sends a created event.
    if (numClients == 0) {
        socket.join(room);
        socket.emit('created', room);
        // If the room is created, it joins to it and it sends a joined event
    } else if (numClients == 1) {
        io.sockets.in(room).emit('join', room);
        socket.join(room);
        socket.emit('joined', room);
        // If the number of clients is exceeded, it sends a full event
    } else {
        socket.emit('full', room);
    }

    socket.emit('emit(): client ' + socket.id + ' joined room ' + room);
    io.sockets.in(room).emit('broadcast(): client ' + socket.id + ' joined room ' + room);
});

3.2 Web pages
This section will be divided in 3 parts: the first one for explains the visual part of the Web pages, the second explains socket usage in the clients and the final one explains how WebRTC works in these pages.

3.2.1 Web pages visual design
The video streaming function has 2 Web pages:

- **index.html**: a Web page that will be accessed by the robot in order to send the video and audio to the command and control monitor.
• **main.html**: a Web page that will be accessed by the command and control monitor to monitor the media sent by the robots. In this Web page the screen is divided in 8 cells (as the number of robots of the lab environment is currently 8) and in each cell a button is located. When this button is pressed a frame is loaded in the cell, which will show the media sent by the robot.

### 3.2.2 Socket usage

Each client connects to the Web page using websockets; in that case client-side socket.IO is used. This socket is used to connect to a room, to send messages to the server and to handle events. Below are listed the events that the client socket handles:

- **Created**: this event will tell the client that is the creator of a room that the room was successfully created.
- **Full**: this event will tell the client that the room that it wants to join does not allow more clients.
- **Join**: this event will tell the client already connected to the room that a new client has connected to the room.
- **Joined**: this will tell the client applying to join the room that it is connected to the room.
- **Message**: this will tell the client that it has received a new message. This message could be of different types, such as:
  - **Got user media**: when the client receives this type of message, it will check if it achieves some conditions are fulfilled (the channel is ready, the connection is not started and the local stream is defined) and then it will start the connection with the other peer.
  - **Offer**: when it receives an offer first of all it checks if the connection is ready. If not it will start-up the connection. It then sets the remote description to the Session Description that it has received from the other peer and it makes and sends the answer to the other peer. The Session Description has the information necessary to establish the connection between 2 peers.
  - **Answer**: when it receives an answer and the connection is started, it sets the remote description to the Session Description that it has received from the other peer.
  - **Candidate**: when it receives a candidate type message and the connection is started, it creates an ICE candidate from the message and adds to the peer.
  - **Bye**: when it receives a bye message it closes and tears down the connection to the other peer and it initializes the variable that tells whether the connection is started.

Below is the code for handling events and sending messages:

```javascript
//Create socket and join to the room to start the connection between peers
var socket = io.connect();

//Create or join to the room.
if (room !== '') {
    console.log('Create or join room', room);
    socket.emit('create or join', room);
}

//Socket event handling
socket.on('created', function (room) {
    console.log('Created room ' + room);
    isInitiator = true;
});
```
socket.on('full', function (room) {
    console.log('Room ' + room + ' is full');
});

socket.on('join', function (room) {
    console.log('Another peer made a request to join room ' + room);
    if (isInitiator) {
        console.log('This peer is the initiator of room ' + room + '!');
    }
    isChannelReady = true;
});

socket.on('joined', function (room) {
    console.log('This peer has joined room ' + room);
    isChannelReady = true;
});

// Message sending and reception
function sendMessage (message) {
    console.log('Peer sending message: ', message);
    socket.emit('message', message, room);
}

socket.on('message', function (message) {
    console.log('Peer has received message:', message);
    if (message === 'got user media') {
        maybeStart();
    } else if (message.type === 'offer') {
        if (!isInitiator && !isStarted) {
            maybeStart();
        }
        pc.setRemoteDescription(new RTCSessionDescription(message));
        doAnswer();
    } else if (message.type == 'answer' && isStarted) {
        pc.setRemoteDescription(new RTCSessionDescription(message));
    } else if (message.type === 'candidate' && isStarted) {
        var candidate = new RTCIceCandidate({
            sdpMLineIndex: message.label,
            candidate: message.candidate
        });
        pc.addIceCandidate(candidate);
    } else if (message === 'bye' && isStarted) {
        handleRemoteHangup();
    }
});
3.2.3 Media exchange

Figure 7 shows the sequence that the peers follow in the establishment of the connection in order to exchange the media.

The sequence is as follows:

1. First of all, caller and callee get the user media (camera and audio got form the webcam and the microphone of the laptop in the case of robots, and audio in the case of the command & control monitor) and create a new RTCPeerConnection. When you use WebRTC each peer needs to send some media to the other peer and in that case as the robot is not going to use the video sent from the command & control monitor it was decided to send to command & control only the audio which needs less bandwidth than video.
2. The initiator of the room creates an offer with its own Session Description, it sends an offer to the TURN-STUN Server.
3. This server will redirect the offer to the callee.
4. When the callee receives the offer, it sets the Remote Session Description to the Session Description of the offer.
5. Then it creates the answer with its own Session Description and it sends it to the TURN-STUN Server.
6. This server will redirect the answer to the caller.
7. When the caller receives the answer, it sets the Remote Session Description to the Session Description of the answer.

Figure 5: How WebRTC works for media exchange in the proposed solution
8. The exchange of ICE candidates starts. This exchange is done through the TURN-STUN Server.
9. When the exchange is finished, the media exchange starts.
4 Testing
This chapter explains how the implemented solution was tested to know if the requirements mentioned in chapter 1 are achieved. First of all bandwidth utilization test will be explained and after that CPU utilization calculation will be explained. Finally is important to tell that almost all tests done in this project were done using Aspire One D270-1375 laptops, the laptops that are shown in Figure 1. This laptop uses an Intel Atom N2600 CPU (1.6 GHz, 1 MB L2 cache), which is a dual core CPU.

4.1 Bandwidth utilization test
In order to find whether the developed system is adapted to available network resources in the lab environment, one has to create experiments that measures the bandwidth utilization so that bandwidth utilization can be measured and related to the requirements on QoS.

This test is based on the work done by Singh et al. [9]. In the case of this project the test was done with 4 robots running the robot_agent and the Web page with the WebRTC application at the same time. For measuring the bandwidth WebRTC statistics API was used.

The experiments were carried out as follows. In each robot browser the Web page with the WebRTC application was run. After that each robot was started and in the command & control monitor browser the Web page to visualize the video sent from the robot, and a video stream from the scene ahead of it was initiated. Then the bandwidth utilization was measured. This experiment was done for 1, 2 and 4 robots working at the same time, each experiment was done 10 times.

The total bandwidth consumption was:

- 1 robot: 18 kbps on average over 10 tests.
- 2 robots: 22 kbps on average over 10 tests.
- 4 robots: 80 kbps on average over 10 tests.

Taking into account that the bandwidth of the WiFi access point used in the lab environment used is 1167 Mbps the bandwidth utilization of the WebRTC application is not significant.

It is necessary to mention that the video received by the command & control monitor got frozen at some point, which might have influenced the test results. This "freezing issue" appeared in all the test cases (1, 2 and 4 robots).

4.2 CPU utilization
Due to the frozen video issue with the video received by the command & control monitor, and having a low bandwidth utilization, we decided to calculate the CPU use of the robots as a supplementary measure. To calculate the CPU utilization the Linux command top was used. This command helps us to monitor the system. It logs the total CPU usage of a process since the last screen update. In the case of these tests the command was used to log the CPU utilization 100 times per second.

These experiments were carried out as follows. For each test the system was restarted and the robots placed in different places in the lab area.

4.2.1 Four robots running WebRTC application and robot_agent
This test was done with 4 robots running the robot_agent and the Web page with the WebRTC application at the same time. The average CPU utilization of each robot in each test is shown in Figure 6:
Figure 6 shows that each robot CPU utilization is higher on average than 100% in each test, so they utilize both of the cores of the CPU.

As consequence, it was decided to calculate the CPU utilization of each application in each test to know which application uses more CPU. Figure 7 and Figure 8 show the average CPU utilization of each application:

Figure 7: WebRTC application average CPU utilization when 4 robots are running together
As it is shown in both of the Figures WebRTC is the application which consumes the highest amount of CPU, on average more than 100% in each test, while the robot_agent consumes less than 10%. The peaks shown for the Robot 1 in tests 3 and 4 are due to an isolated malfunction of the robot, which is out of the scope of this thesis. As a conclusion, the WebRTC application utilizes both of the cores of the CPU.

### 4.2.2 Four robots running WebRTC application only

In order to understand if the video got frozen because of the WebRTC, it was decided to run only the WebRTC at the same time in 4 robots. Figure 9 shows the CPU utilization of the WebRTC application in the 4 robots in 10 tests.

As is shown the Figure 9 the CPU utilization doesn’t decrease comparing with when the WebRTC is run together with robot_agent.

### 4.2.3 One robot running WebRTC application

After that, it was decided to run the WebRTC application only in one robot, to know if the problem was focused in the WebRTC application. Figure 10 shows the CPU utilization of the WebRTC application in several tests:
Web-based real-time communication for rescue robots

As it is shown in the Figure 10 the WebRTC application still utilizes both of the cores of the CPU. So, as a conclusion, the freezing problem might happen because of the high CPU use of the WebRTC in current lab environment laptops.

4.2.4 More powerful laptop running WebRTC application

Finally, to study the high CPU utilization problem was due to the WebRTC application being inefficient or because of the limited resources in the laptops that are used in the lab environment, it was decided to run the WebRTC in a more powerful laptop. The laptop that was used in this test was a Samsung R540. This laptop uses an Intel Core i3 M 350 2,27 GHz. This CPU also has 2 cores. And the software used to log the CPU utilization was Process Monitor\[^{16}\]. Figure 11 shows the average CPU utilization of the WebRTC in this laptop.

As it is shown in the plot the WebRTC application utilizes less than 10 % of the CPU. This test result demonstrates the problem is likely to be the low computing power resources of the previous laptops.

5 Conclusions and future improvements

In this project a Web application was developed for sending soft real-time video and audio from robots to a command & control monitor using WebRTC technology. This audio and video sending does not ensure real-time delivery but has a best-effort character. Functional tests showed that the main objective of the project i.e realising a first prototype was accomplished. However, the tests indicate that the system did not satisfy QoS requirements and some steps to examine the reasons for this were initiated in the project.

Below the main conclusions of this project are listed:

- From a usability (ease of integration) perspective WebRTC is a very useful technology to use in rescue robots because it does not need to install any plug-in for using it, and it is not need any connection to the outside Internet.
- The laptops used in the lab environment are not powerful enough to run the WebRTC application developed in this project because this application utilizes almost all of the CPU capacity of the laptop to run the application. Because of that reason the video received by the command & control monitor gets frozen.
- WebRTC does not consume so much bandwidth (estimated to be around 80 kbps in the case of 4 robots working together), so it is a useful technology in areas that the bandwidth is limited, such as disaster areas, thus satisfying low bandwidth requirements.

To continue and improve this work the following items may be considered:

- Implement the solution using ffmpeg\textsuperscript{17}. This is a multiplatform Open Source video and audio streamer that can be used in a very small computer like Raspberry Pi\textsuperscript{18} and Arduino\textsuperscript{19}. So it needs less CPU than WebRTC.
- Find a way to make WebRTC utilize less CPU. One way can be not showing the video taken from the webcam of the robot laptop in index.html when that Web page is accessed from the robots. Instead one could show the video in the Web page accessed from the command & control monitor.

\textsuperscript{17}https://www.ffmpeg.org/
\textsuperscript{18}http://www.raspberrypi.org/
\textsuperscript{19}http://www.arduino.cc/
6 Appendices

Below is the list of appendices of this project:

6.1 Instructions to install the software needed to start-up the application

First of all how to install and configure the TURN-STUN server will be explained. Before installing the server, the latest version of libevent library should be downloaded, built and installed. Libevent could be downloaded from:


After that logged as root, with the next commands libevent will be installed and built:

```
tarxvfz libevent-2.0.21-stable.tar.gz
cd libevent-2.0.21-stable
./configure
make
make install
```

Before installing and configuring the TURN-STUN server is necessary to ensure that universe repository is enabled. This could be done typing the next command:

```
sudogedit /etc/apt/sources.list
```

Ensure that the next line is included:

```
deb http://us.archive.ubuntu.com/ubuntu saucy main universe
```

After any changes is needed to update system using the next command:

```
sudo apt-get update
```

Finally the TURN-STUN server should be installed using the next command:

```
sudo apt-get install rfc5766-turn-server
```

After installing, the TURN-STUN server port 3478 (STUN) should be opened for TCP and UDP.

Finally, the TURN-STUN server should be configured. For it is necessary to open /usr/local/etc/turnuserdb.conf file and add the next line:

```
rescuerobot:rescuerobot
```

After installing and configuring the TURN-STUN server, it is necessary to install node.js, socket.IO and node-static in order to run the web server. For that first of all it is necessary to move to the folder were the server code is located. After it is necessary to install node.js typing these commands:

```
sudo add-apt-repository ppa:chris-lea/node.js
sudo apt-get installnodejs
```

After that socket.IO and node-static should be installed using these commands:

```
 npm install socket.io
 npm install node-static
```
Finally is necessary to open the port used by the web server (TCP: 2200).

### 6.2 Instructions to run the system

For running the system is necessary to run the TURN-STUN server. For that is necessary to move into the folder that contains the web server and run these commands:

```
 turnserver -L <public_ip_address> -o -a -b turnuserdb.conf -f -r rescuerobot.com
 node server.js
```
References


