Álvaro Bolaños Rodríguez

Cloud Communication Channel for Thermal Cameras

Helsinki Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Bachelor's Thesis

10 May 2017

Metropolia

Author(s) Title Number of Pages Date	Álvaro Bolaños Rodríguez Cloud Communication Channel for Thermal Cameras 45 pages + 3 appendices 10 May 2017
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialization option	Software Engineering
Instructor(s)	Dr. Tero Nurminen, Principal Lecturer

The objective of this project was to develop an effective bidirectional communication system for thermal sensors from the company LeViteZer using an IoT approach and connecting them through cloud services and mobile networks provided by the Nokia Innovation Platform.

In the project different methods are described to build this communication channel from a full stack point of view. The methods are grouped on different sides depending on the point of view of each element of the communication channel which are sensor, server and client side. The described sensor-side methods are different pieces of hardware to connect the sensor to the network: an Android smartphone used as a gateway, an LTE module and a Raspberry Pi with a LTE dongle. The server-side methods are different languages and frameworks to control how clients and sensor are connected to the cloud through a web API. The client-side methods are different ways to interpret the data which comes from the cloud and therefore the sensors. There is also description about other methods for testing, cloud, control version and document preparation system.

The outcome of this project was a set of pieces of applications to make the communication system using the best fitted methods for it. On the sensor-side were used a Raspberry Pi to read the sensor using Python alongside systemd services to keep the data flowing from the sensors to the network. On the server-side a Node.js application orchestrates how clients and sensors are connected. On the client-side Python and Android was used to make client applications.

This communication system as the date of the publication of this thesis was tested and met the expectations. It is being used on the Nokia Innovation Platform for further research.

Keywords	IoT, IR, thermal, sensor, cloud, full-stack, communication channel, websocket, http, LTE, Android, Python

Metropolia

Contents

Abbreviations and Terms

3G	Third Generation Mobile Network
Android	Operative System used mainly in smartphones
Apache	the most used web server software in the world
API	Application Program Interface
ASCII	American Standard Code for Information Interchange
C++	Programming language which is a superset of the C language
CPU	Central Process Unit
CVS	Control Version System
Docker	Software container platform
ER Diagram	Entity Relationship Diagram
Git	Control version software
GitHub	Hosting site for Git repositories
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	HTTP over SSL
I/O	Input/Output
IaaS	Infrastructure As A Service
IoT	Internet of Things
IP	Internet Protocol
IR	Infrared
Java	Programming language able to run on most of the Operative Systems
Java ME	Java Micro Edition: Java version for mobile or embedded devices
Javascript	Dynamic programming language used mostly in browsers although can be used in desktop and server applications
JSON	JavaScript Object Notation
Kubernetes	Automated container deployment, scaling, and management
LabVIEW	Environment for visual programming language used for instrument control
LTE	Long Term Evolution
Matplotlib	Plotting library for Python programming language
MIDlet	A MIDlet is an application that uses the Mobile Information Device Profile (MIDP) on Java ME environment
NAT	Network Address Translation
Node.js	Node.js is an open-source, cross-platform JavaScript runtime environment for developing server-side applications
NPM	Node Packet Manager
OOP	Object Oriented Programming

OTG	USB On-The-Go
PaaS	Platform As A Service
PDF	Portable Document Format
PHP	PHP Hypertext Preprocessor
Python	Dynamic typed interpreted programming language
Qt	Cross-platform application framework
RAM	Random Access Memory
RAN	Radio Access Network
RS-232	A standard for serial communications
RTP	Real Time Transport Protocol
S2I	Source-to-Image
SaaS	Software As A Service
SIM	subscriber identity module
SPI	Serial Peripheral Interface
SQL	Structured Query Language
SSH	Secure Shell
TCP	Transmission Control Protocol
TCP/IP	All the necessary layers to match the conceptual model of the Internet protocol suite
UDP	User Datagram Protocol
UI	User Interface
UML	Unified Modeling Language
URI	Uniform Resource Identifier
USB	Universal Serial Bus
VoIp	Voice over IP
VPN	Virtual Private Network
Websocket	Application protocol build on top of TCP
XML	Extensible Markup Language

1 Introduction

Thermal images have a number of advantages over conventional light-based video camera images. These thermal images can tell not only whether there are living people or animals but also whether there is any temperature anomaly on them. They can be used to make assumptions about their physical state or understand how a group of individuals moves or behaves using Computer Vision software.

Taking into account that the population in the northern European countries and especially Finland is rapidly aging, costs in elderly and general health care are becoming more expensive. Health care entities may decrease their costs by searching for new ideas in the field of IT.

Here is where the Internet of Things (IoT) comes into play. Nowadays the Internet is very accessible and fast. Almost every conceivable device such as phones, watches, televisions, speakers and cameras, can be connected, can send and can receive continuously big amounts of data.

IoT projects are usually involve on making things to be connect to Internet or any sort of network even if it is not designed for it in principle. Then to transform a regular device into an IoT device it is necessary to defining a communication channel between the IoT devices, services, clients, databases, etc. In order to manipulate, visualize, store and distribute data from them. This thesis indicates the steps that were followed to develop the solutions to create a communication channel and what methods were used to carry out the project.

Background of the case company

The thesis originates in collaboration with the case company called LeViteZer (http://www.levitezer.com/), Metropolia and Nokia. LeViteZer is a company that develops controllers for cameras for image stabilization such as gimbals [2]. They provide the thermal sensors.

Nokia with their innovation platform [3], provides the cloud environment and the network. My part as Metropolia student is develop the mentioned communication channel.

Business Challenge

The case company wants to find a system which uses thermal sensors that allow us to monitor patients while keeping their privacy along with reducing costs by using less staff and improve the service quality. Moreover, it should be possible to access data from those sensors fro anywhere which make them very versatile and portable.

With the business challenge in mind, this study aims to answer the following question:

> How to create an effective communication channel between thermal sensors and clients such as computers, laptops, etc and use it on Well-being services?

2 Theoretical Background

Communications between devices is not a new topic, there are plenty of methods and communications protocols, this can be also part of the problem: there are too many of them and sometimes this can be overwhelming. In this section it is described what is known about communication and network protocols in order to find the best way to create a communication channel as stated in the business challenge on section 1 of the introduction.

2.1 Communication Between Devices

In order to establish a communication between two ends in a computer network or on the Internet it is good to comprehend how the ”Internet Protocol Architecture” or TCP/IP stack works. Most of networks are based on it, including office and home networks [4, 9].

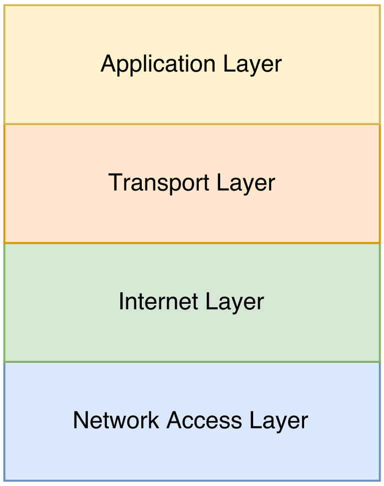

Figure 1: Transmission Control Protocol (TCP)/Internet Protocol (IP) Architecture

The TCP/IP stack is built in layers:

1. Network Access Layer: physical medium to access the network.
2. Internet Layer: handles the routing of data
3. Transport Layer: provides host to host data delivery services.
4. Application Layer: applications and process that make use of the network

Then, a way to communicate through the TCP/IP stack as shown in figure 1 must be found.

2.2 The Medium

At this point it has to be decided what physical medium to *access the network* (Network Access Layer in figure 1). It could be a simple copper cable, Wi-Fi, microwaves, laser, etc. almost any kind of electromagnetic wave. But since this is an IoT project the best approach will be using Radio Access Network (RAN) which it is accessible from cell antennas and nowadays provide great speeds and bandwidth.

As stated in the Introduction, Nokia provides access to a mobile network. This network is called NetLeap which uses Third Generation Mobile Network (3G) and Long Term Evolution (LTE) technologies (the same that use mobile phones to connect to the Internet). NetLeap is a closed network for research managed by Aalto University and Nokia [5].

2.3 Network Protocols

The Internet Layer protocols usually rely in routers and other apparatus which are out of our control, hence the only concern is about transport and application layer.

The idea is to use reliable application protocol to make the connections and this should be platform independent and a Internet standard, these protocol standards specifications are available officially in https://www.rfc-editor.org/standards. On the next sections are the protocols considered and the reasoning behind them.

2.3.1 UDP

User Datagram Protocol (UDP) is a connectionless transport protocol, it does not guaranty delivery nor order of packets which means they can get lost and will not be re-requested and might come in a different order than when they were sent [4, 18].

UDP is commonly used to provide real communication such as time video stream on protocols as Real Time Transport Protocol (RTP), it is also used in Voice over IP (VoIp) to deliver telephone calls over network. They take advantage about the connectionless nature of UDP which despite the mentioned disadvantages it has a low latency. On the other hand losing some packets during a call or video retransmission is not a big deal.

Then a UDP communication system may be suitable for this project since, what it is sending from the thermal sensors is a binary stream of images. Commands can be sent over UDP as well. This can be done sending packets between sensors and clients directly or through a server which could coordinate the data flow.

Port Issues on Remote Hosts

Routers and firewalls usually do not accept connection from ports other than 80 and 443. This is an issue when using UDP (or TCP) sockets approaches. In a local network usually there are no such issues (although firewalls could be strict, depends upon the local network administrators), so a solution can be using a Virtual Private Network (VPN) provider which allows remote computers act as if they where in a local network and increase the security as well.

Another solution is UDP hole punching technique to establish bidirectional communication between hosts that are behind Network Address Translation (NAT) routers by using a external host to keep track of the ports and addresses used in the NAT tables of both hosts routers.

2.3.2 TCP

TCP is the other transport layer protocol, unlike UDP, TCP is connection based and guaranties the delivery and order of packets. Thus protocols made on top of TCP establish a connection and have to maintain it, which increases the latency [4, 19].

Http

Hypertext Transfer Protocol (HTTP) and HTTP over SSL (HTTPS) is application layer protocol made on top of TCP. HTTP is meant to request resources such as HyperText Markup Language (HTML) documents, Extensible Markup Language (XML), JavaScript Object Notation (JSON) or plain text. Any request has a response which can contain a body of data and response code (like the famous ”404 not found”) [6].

HTTP is strongly related to the www, on most of routers and firewalls the port 80 (HTTP) and 443 (HTTPS) are allowed. In addition, the HTTP responses can be used to receive data from the other end indirectly. This could be used for a request-response sensor-client communication approach using a third party like a server.

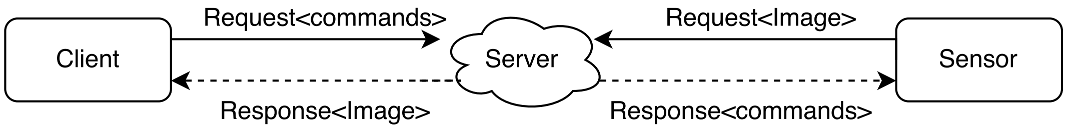

Figure 2: Http/s approach

As seen in figure 2 both sensor and client use requests to send data (which is between ”<>” signs in the figure) that other end expect to receive in the response. A server application take cares to forward requests data to responses. Thus all requests are always towards the server and must be continuous.

Websocket

Application protocol build on top of TCP (Websocket) is a relatively new protocol (end of 2011) which was meant to provide web applications with bidirectional communication without making continuous HTTP requests through techniques like XMLHttpRequest [7, 4].

As HTTP, Websocket use ports 80 and 443 (secure Websocket) by default then it can go through NAT and firewalls easily. Once a connection is established both ends can receive and send data until the connection is closed. This connection is started with a handshake in form of a HTTP GET request [7, 6].

3 Methods and Materials

The aim of this project is to design methods to transmit data from one sensor to a client application and vice-versa. Also defining a generalization to communicate from N sensors to M client applications, being either N and M arbitrary numbers.

The communication must be bidirectional since clients can send commands to sensors in order to perform operations such as calibration or delay between frames. In this communication system there are three well differentiated parts:

- Sensor side: software that connects the sensor with the server side.
- Server side: software that connects sensors and clients together.
- Client side: software that connects the user with the server side to access a sensor.

figure 3 illustrates this idea.

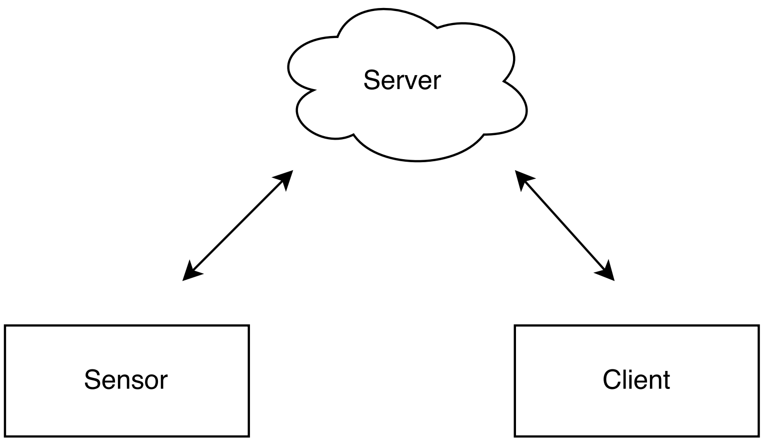

Figure 3: A communication channel

This chapter discusses the methods for each side of the communication system in detail.

3.1 The Sensor

The thermal sensor provided by LeViteZer delivers all infrared data in binary streams through the Universal Serial Bus (USB).

In order to make an image it is necessary to process those streams. Every image or frame comes in 240 rows of 80 bytes of data separated by a delimiter of 3 bytes plus an extra byte that identifies the row:

$$\text{FF FF FF 00}\,\{data\}\,\text{FF FF FF 01}\,\{data\}\,\text{FF FF FF 02}\,\{data\}\ldots \quad (1)$$

In equation 1 every byte is on hexadecimal format containing a sequence ”FF FF FF” which is at the beginning of every row. After this sequence, the 4th byte is the number which identifies the row from 0 to 240.

As equation 2 shows, the 240th (F0 in hexadecimal) and the last row provide meta-data about the frame

$$\ldots\text{FF FF FF F0}\,\{metadata\} \quad (2)$$

table 1 lists all information contained in the meta-data.

Meta-Data

Every frame comes with valuable data about its state and the sensor itself such as configuration, temperature parameters, etc.

Table 1: Meta-data and its position in the row

Meta-Data Parameter	Bytes (from 0 to 80)
Time counter	4,3,1,0
Frame counter	10,9,7,6
Frame Mean	13,12
Sensor temperature	16,15
Maximum temperature	19,18
Minimum temperature	22,21
Discarded packets count	25,24
Maximum temperature limit	28,27
Minimum temperature limit	31,30
AGC byte	34
Bit depth	35
Delay between frames	37,36

Here is a short explanation about the meta-data values:

- Time counter: Number of seconds since the sensor was power on.
- Frame counter: Number of frames since the sensor was power on.
- Frame mean: Temperature mean of the frame.
- Sensor temperature: Temperature of the sensor itself.
- Maximum temperature: Maximum temperature registered the current frame.
- Minimum temperature: Minimum temperature registered the current frame.
- Discarded packets: Packets that were not read. a great number may tell that the application is not reading the sensor fast enough.
- Maximum temperature limit: The limit set with the command for maximum limit.
- Minimum temperature limit: Same as above but with minimum temperature.
- AGC byte: Tells if the limits are set or not (useful for implementing indicators).
- Bit depth: Bit depth of the image; it can be 0, 2 or 8 (default).
- Delay between frames: If no delay is set (delay=0) then the delay is about 111 milliseconds (9 frames per second).

Note: Temperatures from sensors are not in absolute values. This is because the sensor does not detect particular values but differences in temperature.

Commands

The commands are sent over the same serial USB cable from which the frames are received. In table 2 the main commands are displayed with their binary representation: for the first byte an American Standard Code for Information Interchange (ASCII) character is used and for the data argument depends upon the command.

Note: The frame-rate is 9 frames per second, although the sensor can be configured with an arbitrary delay time between frames.

Creating the Image

Every frame has a size of 160x120 pixels, but the image data comes in a matrix of 80x239 (240 is the meta-data row) as shown in equation 1 and 2 above. Each byte of data can

Table 2: Commands accepted by the sensor

Command Name	Command (ASCII byte)	Arguments
Synchronize	S	No
Calibrate	C	No
Set Maximum Temperature Limit	H	2 bytes
Set Minimum Temperature Limit	L	2 bytes
Auto Maximum Temperature Limit	A	No
Auto Minimum Temperature Limit	a	No
Set Bit Depth	B	1 byte
Set Frame Delay	U	2 bytes

be seen as a pixel in a gray-scale image, but in order to generate the correct image the ”data matrix” must be reshaped to 160x120 as shown in figure 4. Every two rows in the data matrix make one row in the image.

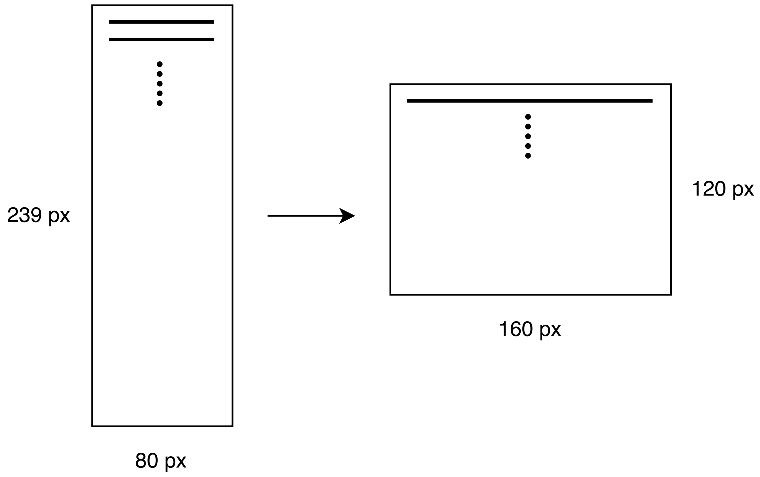

Figure 4: Reshaping data

In the future the number of pixels might change so it is better to define a generalized solution to do the reshaping

For each data row (equation 3) let n_j be the data current row number performing by the 4th byte in equation 1, then for every d_i value in the data row is possible to define every pixel p_{ij} of the image matrix (equation 4) as a coordinate pair (x, y) in the data matrix. s is the size of the data row which is 80 in this case.

$$D_{i,j} = \begin{pmatrix} n_1 & d_{1,1} & d_{1,2} & \cdots & d_{i,1} \\ n_2 & d_{2,1} & d_{2,2} & \cdots & d_{i,2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ n_j & d_{1,j} & d_{2,j} & \cdots & d_{i,j} \end{pmatrix} \tag{3}$$

$$I_{i,j} = \begin{pmatrix} p_{1,1} & p_{1,2} & \cdots & p_{1,j} \\ p_{2,1} & p_{2,2} & \cdots & p_{2,j} \\ \vdots & \vdots & \ddots & \vdots \\ p_{i,1} & p_{i,2} & \cdots & p_{i,j} \end{pmatrix} \tag{4}$$

$$x = n_j \backslash 2 \qquad y = n_j \text{ mod } s + i \tag{5}$$

$$p_{ij} = d_{xy} \tag{6}$$

As seen in equation 6 any pixel value corresponds to a x,y pair defined in equation 5. Note the "\" here is meant for integer division in equation 5, it is **not** a normal division with rational or decimal numbers. "mod" function represents the modulus operation which finds the remainder of a division.

As an example of a practical implementation in Python, see listing 1 where function `process_data_row` is called for every data row:

```python
self.frame_arr[f_row][f_col]

def process_data_row(self, row):
    n_row = row[0]

    for indx, val in enumerate(row[1:]):
        f_row = (n_row)/2
        f_col = (n_row) % 2 * 80 + indx
        self.frame_arr[f_row][f_col] = val
```

Listing 1: Simplified example of creating a frame in Python

Also note that in order to fill the "self.frame_arr" matrix the function is must be called 239 times.

3.2 Sensor Side Methods

The sensor was described in the previous section. Here different methods to read the sensor are discussed. Some worked better than others. Nevertheless all of what was tried is included.

3.2.1 Android Smartphone as Gateway

Most of Android smartphones have an USB On-The-Go (OTG) which allows to use USB peripherals in the phone with the corresponding OTG adapter. This can be used to develop an Android application to receive the data from the sensor and send it through the Internet using any protocol, in this sense the cellphone acts as a gateway to the Internet, letting the sensor access the 3G or the LTE network.

An advantage of this approach is that the smartphone (including a subscriber identity module (SIM) card), has all the hardware on it to make the communication. Hence there is only need to focus on the software part.

Note that this method requires a terminal which supports OTG.

Description of the Android Application

To create the Android application is needed:

- A minimal User Interface (UI).
- A network protocol and its implementation.
- A background process.

The UI letting us start the reading and provides an address and a port to connect. See figure 5.

Then using an Android service to keep reading in the background, which is the usual approach to deal with long-running operations on Android applications [8].

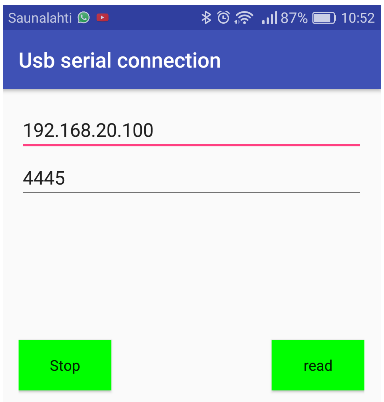

Figure 5: UI of the Android Application

The usb-serial-for-android library provides all to read the necessary data from the mini-usb port [9]. Listing 2 shows an extract of my code implementing the callback to receive data directly from the USB port.

```
1  // the parameter data is a binary string from the sensor
2  @Override
3  public void onNewData(final byte[] data) {
4      if(bufferFrames.isFull()){
5          callback.getBuffer(bufferFrames);
6          bufferFrames = new BufferFrames();
7      }else{
8          bufferFrames.addChunk(new Chunk(data));
9      }
10 }
```

Listing 2: Android: Callback to receive data from sensor

Note in line 3 that it is in reality quite simple to receive binary data in the form of a byte array.

3.2.2 LTE Module

Long Term Evolution (LTE) modules work as a phone: They need a SIM card to connect to the network and they can be integrated in a board. Typically these modules have a number of interfaces such as USB, rs-232 and Serial Peripheral Interface (SPI), to connect the peripherals.

In order to set a route between one's device and a service on the Internet or one's own server, a piece of code must be provided. It depends upon the module how can it be done. For a project like this it is interesting that the module has its own TCP/IP.

Gemalto LTE Module

The Gemalto LTE module is not a simple modem that allows other machines to be connected to the Internet. It has a complete TCP/IP stack which means that protocols of the transport layer such as TCP and UDP and application layer such as HTTP and HTTPS can be used as well, all in very small compact chip as seen in figure 6.

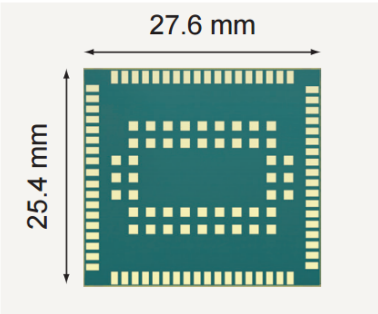

Figure 6: ELS61-E chip size

The module used was Gemalto ELS61-E, which is configured using the Hayes command set (also called AT commands) which are used on modems. See how AT commands look in listing 3.

```
AT^SMSO   # shutdown
AT+COPS   # register to network command
```

Listing 3: AT commands

The Gemalto module can be programmed using Java ME which is a Java edition for embedded devices [10]. Java Me applications use MIDlets which are modular components that allow to code the life-cycle of a single application as seen in listing 4. From a MIDlet it is possible to control the USB, SPI and RS-232 ports, use network protocols and even send AT commands.

```
import javax.microedition.midlet.*;

public class HelloWorld extends MIDlet {

    public HelloWorld() {
```

```
        System.out.println("Constructor");
    }

    /** This is the main application entry point. */
    public void startApp() throws MIDletStateChangeException {
        System.out.println("startApp");
        System.out.println("\nHello World\n");
        destroyApp(true);
    }

    /**  Called when the application has to be temporary paused. */
    public void pauseApp() {
        System.out.println("pauseApp()");
    }

    /** Here you must clean up everything not handled by the garbage collector. */
    public void destroyApp(boolean cond) {
        System.out.println("destroyApp(" + cond + ")");
        notifyDestroyed();
    }
}
```

Listing 4: MIDlet application life-cycle example

For a full list of features of the module refer to http://www.gemalto.com/brochures-site/download-site/Documents/M2M_ELS61_datasheet.pdf

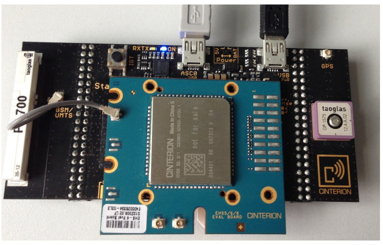

Figure 7: Lte Module on top of the board to program it

To do all the communication and configuration with the module there is a board where it can be attached proving micro-usb connectors, antenna, reset button and power among other things (see figure 7).

3.2.3 Raspberry Pi

A Raspberry Pi is a credit-card sized computer and its model 3B includes, among other things four USB-port and multi-core Central Process Unit (CPU) [11]. Considering that Raspberry Pi comes with a Linux distribution the possibilities are unlimited. For the purposes of this project it is especially convenient to have the possibility of using any programming language and have several software packets available.

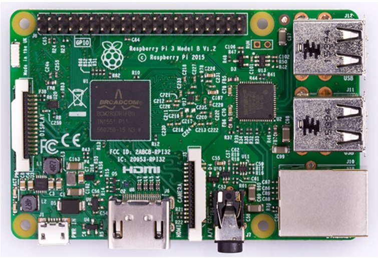

Figure 8: Raspberry Pi Physical appearance

The Reader Program

The programming language chosen to communicate with the sensor from Raspberry Pi was Python since it has a number of libraries available to use. Then it is necessary to point what features are suitable for a program which reads the sensor and opens a connection to the server side:

- Needs to communicate both ways trough USB.
- Has to open a connection to the server/cloud continuously and be able to recover. automatically from failures on the network.
- Must do all this at the same time taking advantage of Raspberry Pi's multiple cores.
- Prioritizes sensor read task over the rest; this is the most critical part and must be have a higher priority.

In order to separate tasks it is possible to use multiple threads which are supported in Python. Although this allows to run several tasks at the same time when using Python, it could be advisable to use the multiprocessing package to separate tasks into several processes and take advantage of the various CPU cores [12]. For controlling process priority the package "psutil" allows to control the "niceness" of the process [13].

In order to get access to the raspberry when it is out of reach, it can be used Secure Shell (SSH), which is a protocol to connect to remote hosts using the shell through an encrypted connection. It is also desirable to be able to do certain operations remotely without using SSH. The connection made between client-server-raspberry can be taken in advance for example to shutdown, reset or update the raspberry. For this purpose the subprocess package can be used to issue commands and other processes, as one can do in a terminal shell [14]. See listing 11 appendix to see my actual implementation.

Services

To run programs and scripts automatically from booting it is possible to add systemd-based services which can be found in most of Linux distributions. For the purposes of this sensor reader at least two services are needed: one to keep the network connection alive, and other one to keep the reader program constantly running. Listing 5 shows these two services which basically run another program and try to keep it alive.

```
[Unit]
### /lib/systemd/system/lte.service
Description=Lte module service

[Service]
ExecStartPre=ls /dev/cdc-wdm0 || echo "cannot see the lte module, retrying..."
ExecStart=/home/pi/lte-daemon
Restart=on-failure
RestartSec=5

[Install]
WantedBy=multi-user.target

[Unit]
### /lib/systemd/system/sensor.service
Description=read sensor and send data via websocket

[Service]
ExecStart=/home/pi/sensor-reader/main.py
```

```
WorkingDirectory=/home/pi/sensor-reader
Restart=always
RestartSec=5

[Install]
WantedBy=multi-user.target
```

Listing 5: Systemd services

Note that the file to be executed is defined by `ExecStart` and it is executed with root privileges.

3.3 Client Side Methods

On this section it will be described different methods to create a client able to connect to the cloud thus the sensor, present the data to the user and send commands to the sensor.

Porting the LabVIEW Application

At the beginning of the project LeViteZer provided an example application to read the sensor from a laptop made in LabVIEW. Although it works, it is not an idea platform to develop since it is strictly close-sourced software and in order to work with it an expensive license has to be paid. Then it was agreed that a ported Python version would be made from the LabVIEW client extending its capabilities beyond the laptop-sensor USB connection.

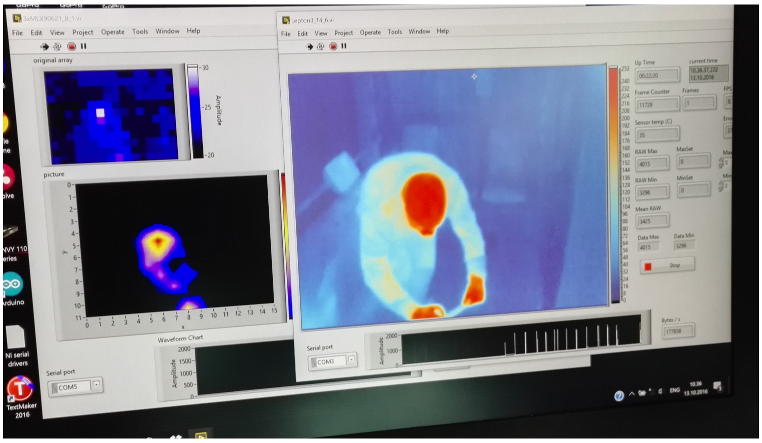

Figure 9: LabView implementation

LabVIEW programs are not written in code but using a ”visual programming language” based on diagrams similar to circuits were the data flows. It is oriented to instrument control, data acquisition and automation. [15]

Although there was no intention to use LabVIEW for the purposes of this thesis, it was used to do research about what protocols to use and testing.

3.3.1 Websocket Python Client

The websocket Python client application was intended to have the characteristics of the LabVIEW's mentioned on the previous section and provide it with connectivity to a cloud service for remote communication. The application was written in Python taking advantage of the multiple libraries that the community offers for free as open source.

Design

In a relatively complex application as this one Object Oriented Programming (OOP) is the most convenient way to go when designing the application.

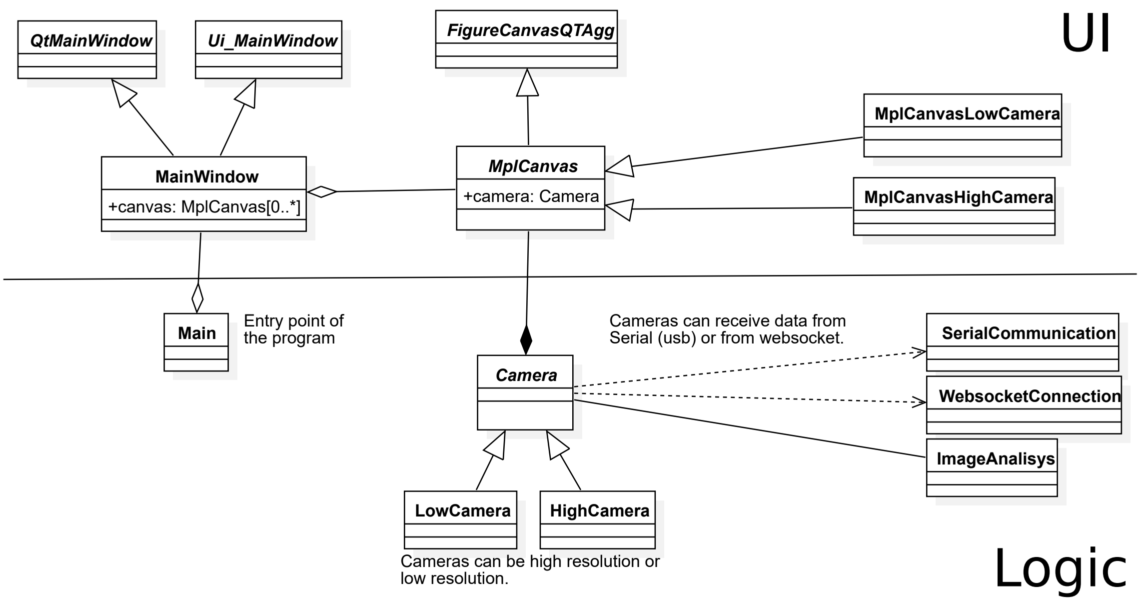

Figure 10: UML diagram (simplified)

In order to make a more flexible application, source of data (thermal image) can be either from USB or from a network, for those purposes the class "SerialConnection" will be in charge of communicate with a thermal sensor connected to the computer directly over

USB (see listing 9 in appendix). This class is the same that is used on the Raspberry Pi to read the sensor. On the other hand ”WebsocketConnection” mimics roughly the behavior of ”SerialConnection”. However it makes the connection over the Internet using the Websocket protocol.

figure 10 shows the most important components of the application and its relations. It can be noticed that there is a separation between UI components and logical ones.

User Interface

The user interface is implemented using the known Qt framework which is uses Programming language which is a superset of the C language (C++) but there is a Python bindings package to use in a Python application without writing a single line of C++ code [16].

Advantages of using qt are among others:

- Cross-platform: the same code works on any operative system where the framework is available.
- It is possible to design the interface using a designer program (see figure 11) and save it as a XML file that can be read from the application, saving much time on the development stage.
- It is a well known framework and there is plenty of information available about it.

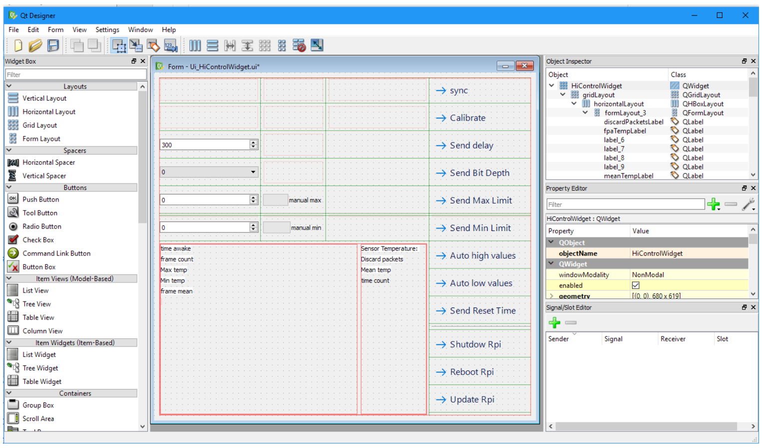

Figure 11: QT designer

The sensor image itself is made using a plotting library named Matplotlib used in quality scientific plots and animations. Matplotlib also provides a back-end to attach the graphics to Qt among other UI frameworks. This can be seen in the Unified Modeling Language (UML) diagram (figure 10) where the class ”MplCanvas” which presents the frames from the ”Camera” inherits from the Matplotlib class ”FigureCanvasQTAgg” [17].

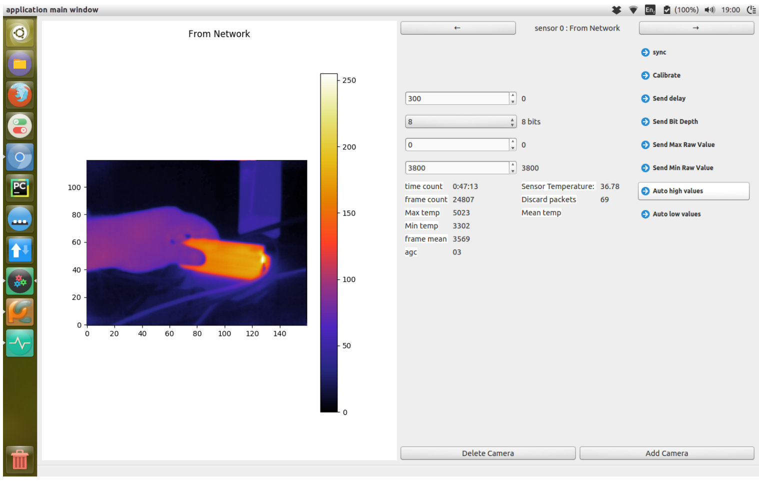

Figure 12: Python Client Application

As seen in figure 12 the application client has two well-differentiated parts. On the left is the image displayed as a colored gray-scale image in figure 13. Matplotlib let us apply color maps on the image very easily and also to add an interpolation to improve image quality, in this case there is a bicubic interpolation [18].

Figure 13: Thermal image

On the left side there is a control panel (figure 14) which displays meta-data information and buttons that can send commands to the sensor as described in section 3.1. On the right of some buttons there are input fields to enter the arguments to the commands that required it alongside the current value of the argument (current argument values are meta-data as well).

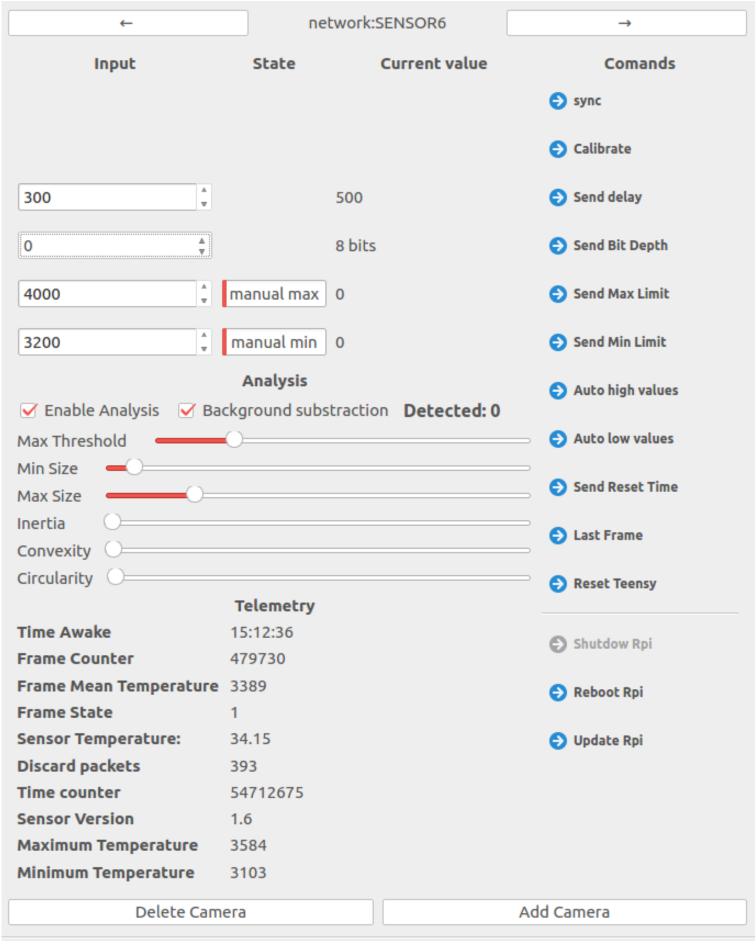

Figure 14: Control panel

There is a separate section for special buttons to control a Raspberry Pi. These commands are shutdown, reboot and update.

3.3.2 Android Client

During the development of the project It was proposed to make a mobile client application and after finishing the Python client described in the previous section I started to develop a simple but useful Android application. It lets connect to any camera already registered in the cloud (see section 3.4).

As in the Python client, it is necessary to code:

- a class to connect to server or cloud through Websocket.
- create the image from the data.
- allow to choose which camera to connect to.

Android applications are divided on different screens called Activities which contain UI elements that the user can interact with. In this case there is an activity for choosing the camera to connect, see figure 15.

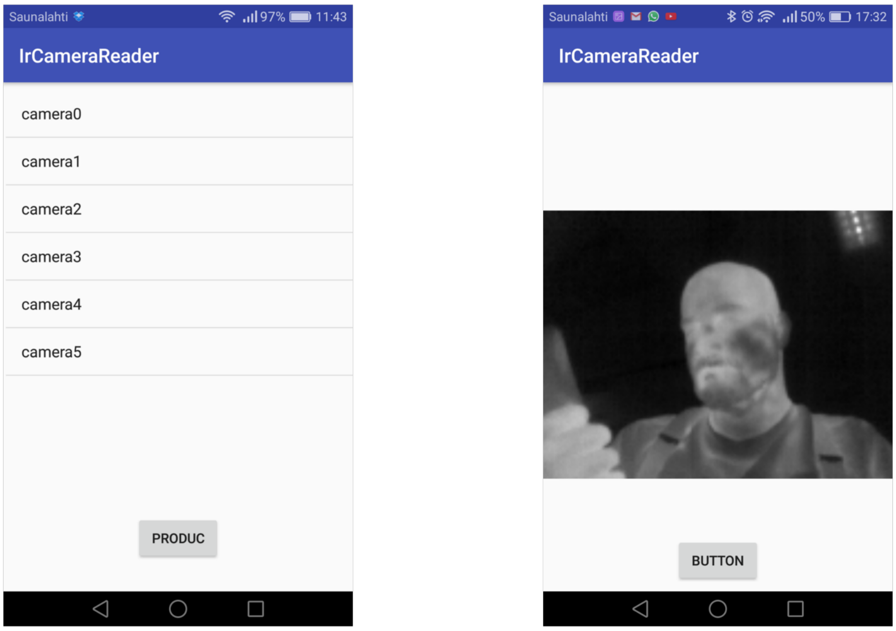

Figure 15: Android Client

The other activity makes the a Websocket connection and processes the data in real time so the different frames can be visualized in gray-scale (figure 15).

To achieve this Android allows to extend UI elements with new behavior. In this case ”ImageView” class is extended to add a Websocket connection and fill the image with an array of binary data from this connection (same as in the Python application section 3.3.1). In listing 12 the ”CameraView” class meets this behavior using ”WebsocketConnection” to get data from sensors (listing 14) and ”HighCamera” to represent a frame of the image (listing 13). All of these listings can be found in appendix .

3.4 Server Side Methods

In this section the different approaches tested to created the middle point between sensors and clients will be described. The server side application must be hosted somewhere, in this case a cloud environment described in the next section.

3.4.1 The Communication Channel

Regardless of which technology is used for coding, building and maintaining the cloud application, and before starting coding it should be defined how different components are related between each other. Let us start by defining the following entities:

- Sensor: It represents a single sensor connected to a Raspberry Pi, although it could a be normal computer. The sensor itself does not connect to networks. Thus it needs middle hardware but it is considered a whole “entity” here.
- Client: A client can be anything that can connect to a sensor by an HTTP request such as a desktop computer or a smartphone. The client must create video image from sensor and send commands using a Websocket.
- Cloud: It is what holds all the sensor entities and their clients on it. The channel should support an unlimited number of sensors which can hold an unlimited number of clients.

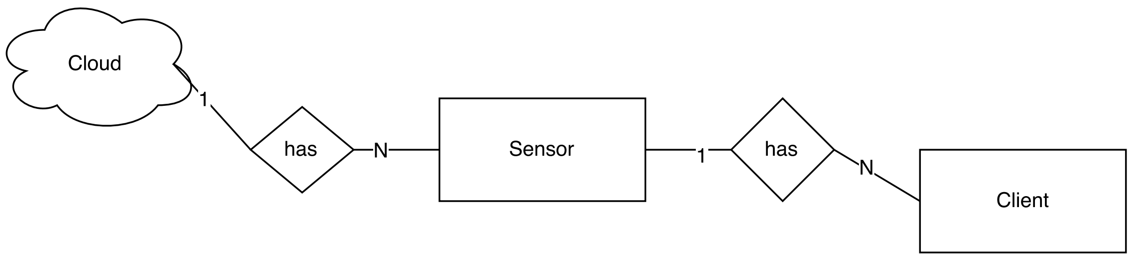

Figure 16: Entity Relationship Diagram of the communication channel

The channel should support an unlimited number of sensors which can hold an unlimited number of clients. The diagram in figure 16 represents the general view of how entities are related to each other. This helps to develop a UML Class Diagram which specifies how classes in an OOP language are related to each other. In figure 17 there is such diagram simplified.

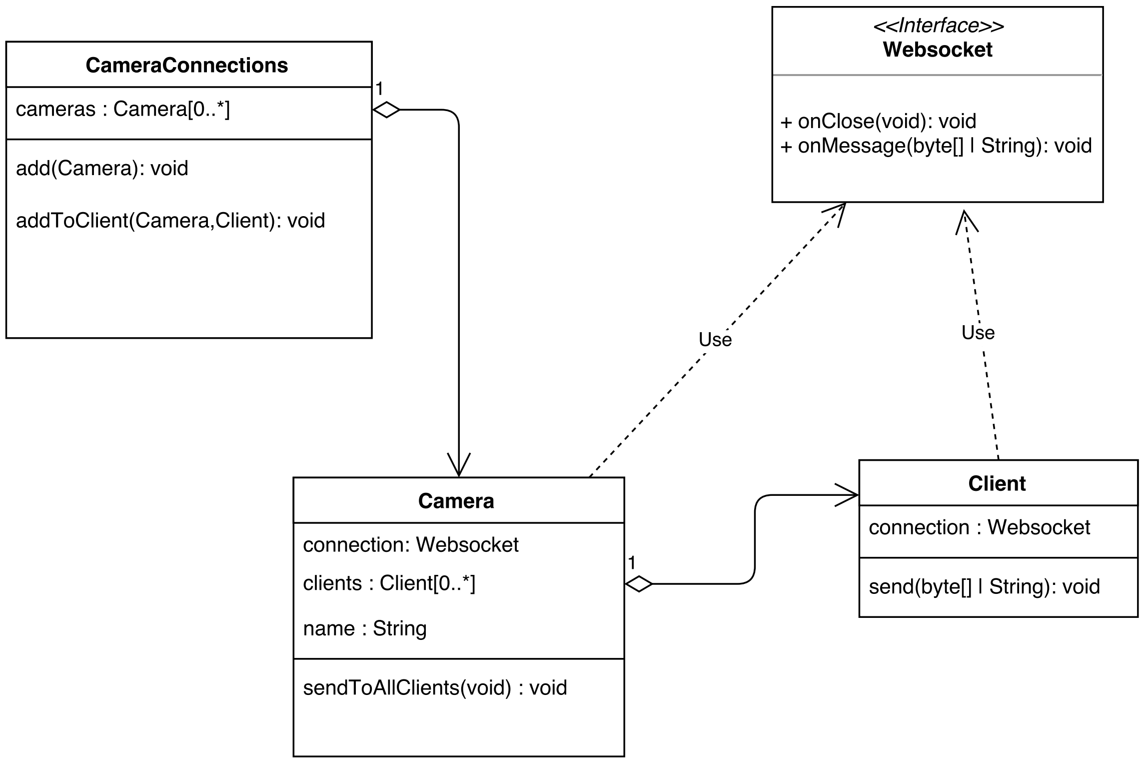

Figure 17: UML diagram

In the final implementation the server side application was written in Dynamic programming language used mostly in browsers although can be used in desktop and server applications (Javascript) which is the Language for Node.js applications.

3.4.2 Python Flask

Flask is a Python framework for web development which allows to write web application back-ends and is especially useful to create web APIs. This application uses an HTTP approach, as described in section 2.3.2, to communicate. Client-side and sensor-side applications must implement the mentioned approach as well.

The application keeps a queue buffer of image data and commands, so that both sensor and client have to request continuously even though when there is no new data. In listing 6 a simplification of this Flask application is shown, the buffer and commands are hold in the `data_queue` and `parameters`.

```
app = Flask(__name__)
parameters = {}
data_queue = Queue(5)
```

```

@app.route('/video/buffer', methods=['POST'])
def submit_buff():
    data = request.data
    logging.debug("data:%s", data)
    if data_queue.full():
        logging.debug('queue is full. dropping 1')
        data_queue.get() # drop 1 buffer

    data_queue.put(data)
    return jsonify(**parameters)


@app.route('/video/buffer', methods=['GET'])
def obtain_buff():
    global parameters
    print request.args
    for k, v in request.args.items():
        parameters[k] = v

    if data_queue.empty():
        logging.debug('queue is empty, sending 0 ...')
        return '0'
    else:
        return data_queue.get()
```

Listing 6: HTTP - buffer approach

Using the `/video/buffer` endpoint it is possible to send data to it using a 'POST' HTTP requests continuously from the sensor side and to send 'GET' requests to get image data in the client side, if a client wants to send a command (called parameter in this application) it can be sent as query parameter in the 'GET' request, for example `www.example.com/video/buffer?command0=value0&command1=value1&...`

A good thing about flask is its straightforward API as seen in listing 6, defining a route to an endpoint is very easy because takes advantage of the decorators `@app.route` and result in a quite clear code.

3.4.3 Node.js Server Application

Node.js is a Javascript runtime which allows to create server-side applications, the main reason of why one would choose Node.js over other well know options such as Apache + PHP Hypertext Preprocessor (PHP) is the non-blocking model Node.js is based on. This let make asynchronous code easily and it is quite fast [19, p.12] [20]. In addiction to this there are other options to consider working with Node.js:

- It has a big community.
- It likely has the biggest open source library on the Internet, accessible through Node Packet Manager (NPM).
- most of cloud providers offer it out of the box.
- It is very easy to start up and configure unlike options.

To work with both Websocket and HTTP protocols in a Node.js application it is possible to use these quite well know open source libraries:

- ws. It is claimed to be the fastest Websocket library https://www.npmjs.com/package/ws
- Express. Lightweight web framework for node. Probably the most used node web framework https://www.npmjs.com/package/express

Although these libraries are focused on web applications, the clients described in section 3.3 do not need HTML characteristics . However in the future a HTML client application could be developed using the API of this Node.js application.

Since both protocols use the same port, an HTTP server can be created using express library and then integrating it with the Websocket library (see listing 7).

```
const express = require('express');
const WebSocket = require('ws');
const http = require('http');

const port = process.env.PORT
const ip = '0.0.0.0';

```

```
/* http server */
const express = express();
const server = http.createServer(express);
    // ... handle http requests ...

/* websocket server extends the http server */
var wss = new WebSocket.Server({
    server: server,
    // other websocket configuration ...
});

 wss.on('connection', function connection(ws) {
    // ... handle websocket requests ...
});

server.listen(port, ip);
```

Listing 7: html and websocket server

Express and ws libraries have events to handle the connections using callbacks for any behavior one wants to add.

A complete view of the main file application can be consulted in appendix listing 15.

3.5 Testing Methods

Testing is an important part of the Software development process. It gives insight into the quality and how much the software is error-prone. However creating tests and maintaining after refactoring it is also a time-consuming practice. Since this project is carried out by one person rather than a team of developers there were added test only on the server-side of the whole communication system.

3.5.1 Unit Testing

Unit Testing is about testing modules or units which are single pieces of software that should work independently of others. In OOP this is usually a class, but it could be a single function or method. It is the developer who should define what a single unit is.

3.5.2 Acceptance Testing: Robot Framework

In order to test the server-side application with an arbitrary number of clients and sensors with different configurations and backgrounds, an acceptance testing approach is very convenient. Acceptance testing features a more structured and complex way of testing which allows an efficient way of pinpointing application failure, automation and reusability [21].

For this purpose Robot Framework is a great tool. It is a open-source for general purpose test automation. It is very flexible and it allows to create tests using human friendly-keywords and generate complete reports and logs about the test results. It can also be extended by adding existing libraries or creating them using Python or Java [22]. A simple test file can be seen in listing 8.

```
Documentation     A test suite with a single test for valid login.
Resource          resource.txt

*** Test Cases ***
Valid Login
Open Browser To Login Page
Input Username    demo
Input Password    mode
Submit Credentials
Welcome Page Should Be Open
[Teardown]    Close Browser
```

Listing 8: A example of robot framework

Using the libraries already in the client Python application (in section 3.3.1) and the Robot Framework API I created a library to test Websocket connections. Those connection are held in an array, created and deleted for every single test, see listing 18 for the whole library code in the appendix .

The library provides keywords like the ones in listing 8 but a group of test cases have to be defined in addition to some local keywords to reuse behavior. See listing 17 in appendix for all test cases defined to test the server-side application. In this file it can be noticed several sections between triple asterisks:

- Variables. To define global variables whose syntax is "`${variable_name}`".
- Settings. Import libraries, setup actions and teardown actions.
- keywords. Here custom keywords can be defined made of keywords from other libraries. They can accept arguments and return values.
- Test Cases. The last section shown every test case. In the end it will be shown if the test was passed or failed.

After the execution of tests an HTML report file will be generated (figure 18) along with a log one (figure 19).

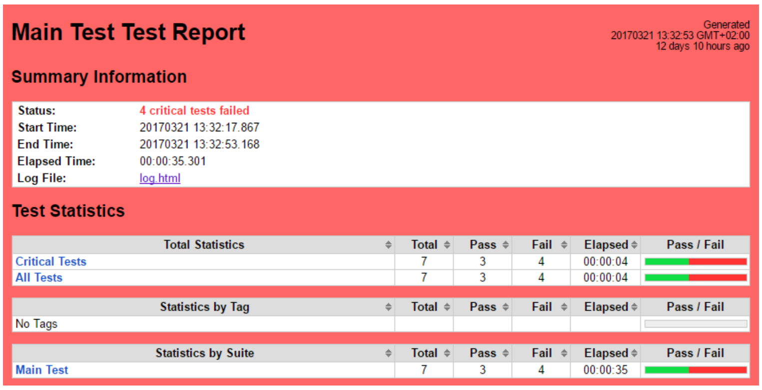

Figure 18: Test report

The report is a general view of what happened during the test execution.

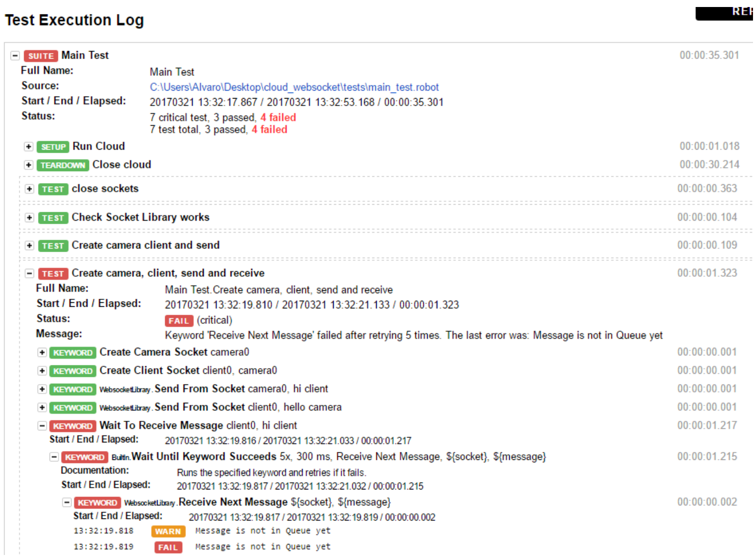

Figure 19: Test Logs

The log file contains more detailed information about the tests.

3.6 Cloud methods

While the Server Side methods (section 3.4) describes the application itself, it does not tell anything about where and how to host the code, so that it would be available everywhere on the Internet.

”The cloud” is quite a broad concept which usually includes different service models known as Software As A Service (SaaS), Platform As A Service (PaaS) and Infrastructure As A Service (IaaS).

SaaS are hosted applications accessible from client applications. On the other hand PaaS provides services for developers to create applications such as SaaS ones. However the developer does not have access to the cloud infrastructure like the servers or the network. IaaS is similar to PaaS but with more control over the platform [23].

OpenShift and Docker

OpenShift is a platform for the deployment of web applications and services. It takes advantage of technologies such as Kubernetes and Docker to run and manage application in containers [24]. Nokia (which collaborates actively in this project) provides PaaS as part of their Nokia Innovation Platform which aims to create solutions for IoT [3].

The process of updating the server side application is through an OpenShift mechanism called Source-to-Image (S2I) which allows creating containers from the application source code without using Docker files. In its simplest form a developer provides a Git repository to the platform and OpenShift takes cares of everything: building, deploying, routing, etc [24, 5].

3.7 Other Tools Used

Control Version

In any serious software development project there must be a Control Version System (CVS) which allows to keep track of any changes in the code (or other files) and reversed if necessary. In this project the popular control version tool Git was used, which has many features but the one that makes it more special is its decentralized repository model which means that every repository copy has all the history changes.

All the repositories to the applications developed in this thesis are hosted on GitHub which it is perfect for sharing or publishing open-source projects.

Latex

To write this thesis LaTeXwas used, which is a documentation preparation system rather than using the mainstream options which are LibreOffice or Office Word. The reasons for this was automation and the quality that can be achieved.

```
The process of updating the server side application is through a OpenShift mechanism called
\gls{s2i} which allow creating containers from the application source code without using
\gls{docker} files, it its simplest form a developer provides a \gls{git} repository to the
platform and OpenShift takes cares of everything: building, deploying, routing,
etc~\cite[5]{shipley2016openshift}.


\subsection{Communication sensor-client}
% TODO: \subsection{Communication sensor-client}
\putimage{communication_udp}{Diagram of the communication using the module and UDP}

\section{Other Tools used}

\subsection{Control Version}
On any serious software development project there must be a \gls{cvs} which allows keep tracking of
any changes in the code (or other files) and reversed if necessary. On this project the popular
control version tool "Git" was used, which has many features but the one that makes it more special
is its decentralized repository model which means that every repository copy has all the history
changes.

All the repositories to the applications developed in this thesis are hosted in \gls{github} which
it is perfect for sharing or publish open source projects.

\subsection{Latex}
% What is latex, what is tex
% reasons to use latex, abreviations, commands etc, show figures
To write this thesis I used \LaTeX which is a documentation preparation system rather than using
the mainstream options which are LibreOffice or Office Word. The reasons for this was automation
and the quality one can achieve. It is known that it is preferred in academic articles and theses.
```

Figure 20: Source code of the document

In figure 20 it can be observed how the code looks like. It is just raw text with no style at all and it has to be compiled to generate a Portable Document Format (PDF) document.

4 Results

The result of this thesis is a set of pieces of software which together form the communication channel. Its central part is the server-side hosted in a cloud accessible through the web API described in section 4.2.

4.1 Software

After solving some bugs on the server-side and the Raspberry Pi part the communication system can work continuously 24 hours a day with several cameras. As of 5.4.2017 the longest test in a sensor in the system has been 4 days and 19 hours. In figure 21 a client application with sensors on different locations.

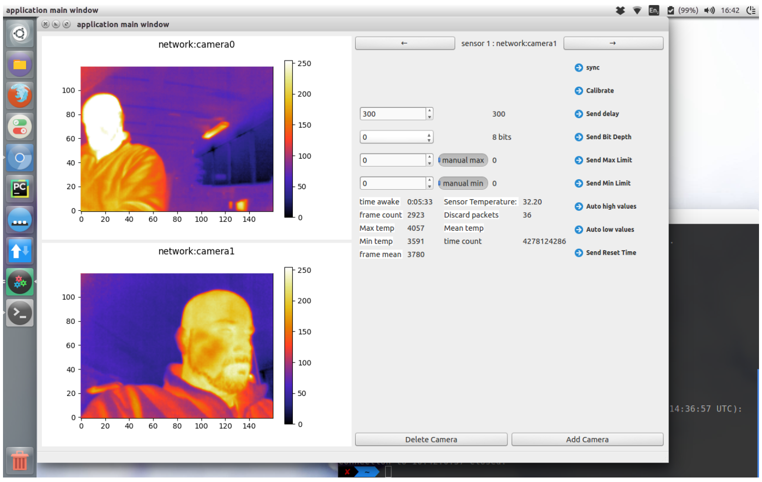

Figure 21: Multiple sensors connected to a single client

As stated in chapter 3 all the code is in GitHub repositories. Here is a list with every repository for most of the software developed in this project.

- Android client application: https://github.com/alvaro893/android-ir-sensor-client
- Server-side Node.js application: https://github.com/alvaro893/cloud_websocket
- Python client application: https://github.com/alvaro893/sensor_reader
- Sensor Reader application on the Raspberry Pi side: https://github.com/alvaro893/sensor-reader/tree/raspberry

- This Thesis: https://github.com/alvaro893/bachelors-thesis

The code shown in appendixes are in these repositories and up-to-date.

4.2 Web API

In order to provide communication with both sensors and clients a web API can be the most convenient way to expose the services of the application providing certain endpoints to register sensors, clients and to access information about the application itself, the information retrieved is in JSON format.

Every endpoint is defined as an Uniform Resource Identifier (URI). A complete URI looks like this:

```
<protocol>://<domain>:<protocol><path>?<parameters>
```

The domain is provided by the cloud service. If the server-side application is running locally on a personal computer the domain is ”localhost”. The paths accepted by the API are shown in table 3.

Table 3: Endpoints of the web API

path	protocol	description
`/client`	Websocket	Register a client
`/camera`	Websocket	Register a sensor
`/cams`	HTTP GET	receive a JSON of information

`/client` and `/camera` parameters also accept query parameters shown in table 4.

Table 4: parameters of the web API

parameter	type	description
`pass`	alphanumeric	password to access
`camera_name`	alphanumeric	camera to connect or to register

For example `ws://localhost:8080/camera?pass=d8n2d0&camera_name=corner-camera`

4.3 Discarded Methods

Android Smartphone Gateway Solution

The android smartphone gateway solution was purely to test the sensors and to seek for a suitable network protocol rather than as a serious solution for the purposes of this thesis, nevertheless the outcomes were satisfactory and opened the door to a deeper understanding on how USB communication can work using a High level language as Java.

LTE Module Solution

The LTE module solution could have been the ideal solution to read a sensor and connect it to the Internet, but there were multiple issues regarding this method:

- It turned out that the module did not support the LTE frequency bands of the mobile provider Netleap see section 2.2 of chapter 2.
- It was hard to set-up a developing environment for it.

UDP Protocols

There were many test to use UDP protocols for communications but for the issues mentioned in section 2.3.1 of chapter 2 and because Websocket worked very good. any UDP was dropped.

Python Flask Server-Side Application

At the end the Node.js application turned out to work better and faster. so the Python Flask server-side application was not further developed.

5 Discussion

This chapter includes discussion about how the results may be used, especially alongside Computer Vision. The sections here are more based on speculation rather than experimentation and proposed lines of research.

5.1 Use in the Nokia Innovation Platform

The Nokia Innovation Platform is a trial environment for teams and start-ups that are related to IoT projects especially projects that use cell-phone network like LTE. This project is part of the Nokia Innovation Platform and hence there is some ongoing use cases. Some of the sensors have been placed on different places to test their functionality and further investigate the uses of this communication channel.

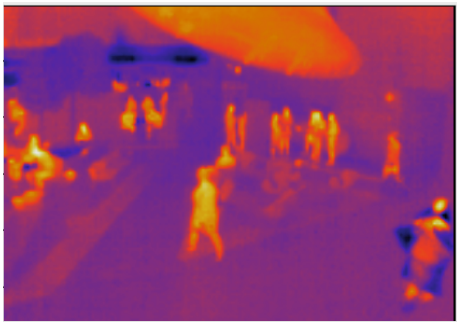

Figure 22: People in a room

In figure 22 some people can be seen clearly and in figure 23 the same situation but running computing vision software to detect how many people in the room.

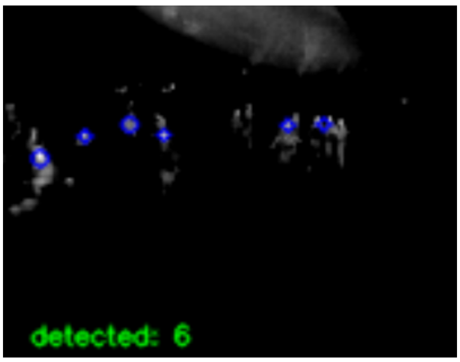

Figure 23: Analyzed image to detect heat points in the room

Computer vision can be used to track people and research their behavior storing big amounts of tracking data may allow to make many types of predictions. In the next sections some scenarios are proposed where this could be useful.

On crowded surfaces such as shopping centers, stadiums and summer festivals it may be useful to observe patterns and even detect or predict dangerous situations. This is an interesting research topic and could be a continuation of this final year project.

5.2 Use in Health Care

In this section different uses in the health care field of the sensor communication channel will be discussed. The sensors do not capture light but heat, thus making it impossible to recognize patients from a distance This can be considered an advantage since most people do not feel comfortable with the idea of being watched or video-recorded.

By using Computer Vision techniques a thermal image can be easier for a computer to ”understand”, in other words, to analyze especially when it is possible to add limits to the maximum and minimum temperature so that a range of temperature can be interesting while the rest is removed, as illustrated in figure 24.

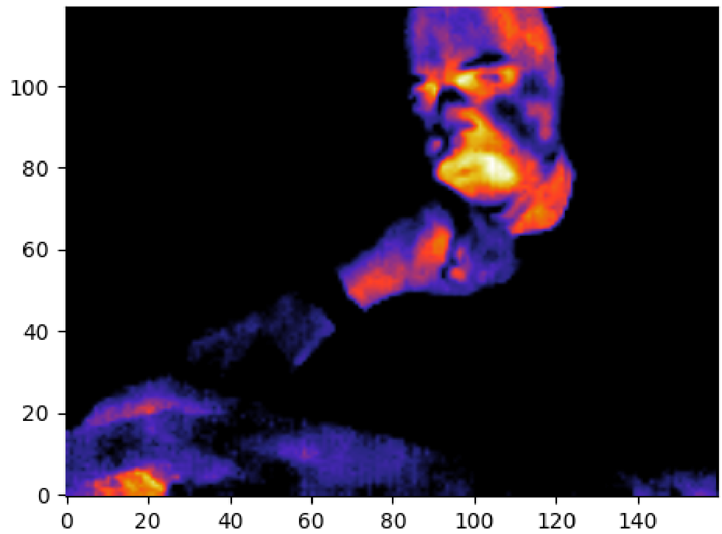

Figure 24: Minimum set to human temperature

In addition, to set the minimum and maximum temperatures it is possible to change the bit density of the image, so we get less data and probably make the image analysis a little easier, as shown in figure 25.

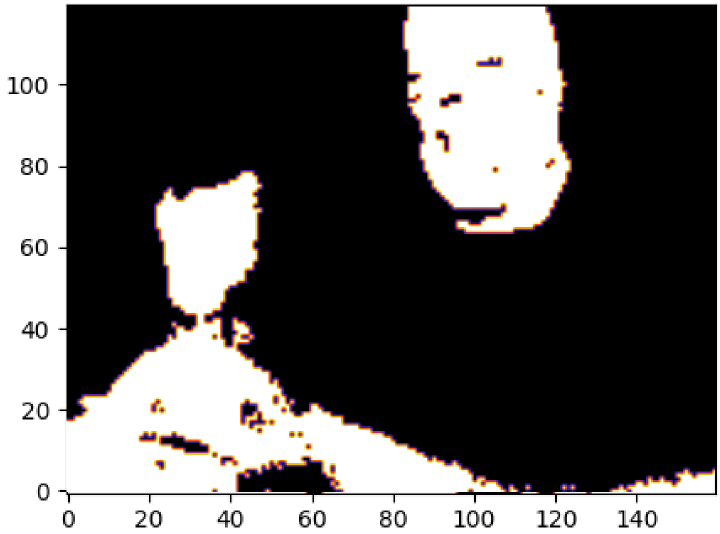

Figure 25: 2bit version of the image

Another approach could be to use thermal image to see things that usually are not visible at a glance like for example veins and arteries as seen in figure 26.

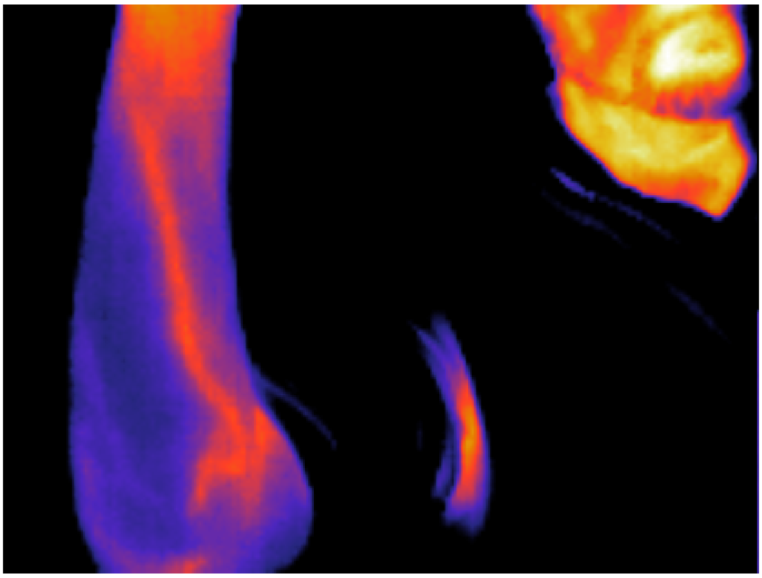

Figure 26: Arteries and veins in the human arm

Unfortunately a proper research of health care uses was beyond the scope of this thesis. Nevertheless In the following sections there are some propositions for further investigation.

5.2.1 Use in Bed Patient Care

A use case may be for monitoring patients in bed, either hospital or home bed. Using image analysis software nurses or doctors could obtain data from multiple patients at the same time. Since the sensors can detect heat, this could trigger some alarm on a high temperature or if the patient is missing for a long period of time.

Also, as discussed in previous sections it may be interesting to seek for patterns how a patients moves through time and even how long patients are not in bed.

5.2.2 Use in Nursing Homes and Psychiatric Hospitals

The same principle discussed in section 5.1 may be applied to nursing or residential homes where there is a number of people inside a room or outside in a backyard. Then trough Computer Vision it can be detected how many persons there are. If someone is too still or passed out. This detection could trigger certain alarms which will warn the caretakers in the facility.

As nursing homes, psychiatric hospitals could have this kind of system. It could detect potential dangerous situations in this case.

5.3 Other Uses

5.3.1 Self-driven Vehicles

Self-driven public transport vehicles could also benefit from this system, since there is no driver nor any other worker on the vehicle. Accessible remote thermal sensor can give valuable data to a hypothetical location in charge of the security of such vehicles and trigger an alarm in case of any dangerous situation.

Figure 27: self-driven bus in Otaniemi

Figure 27 displays the ”Robot bus” moving around the Otaniemi campus in Espoo without driver.

5.3.2 General Surveillance

Surveillance is usually carried out by regular light-based cameras plus sometimes some IR LEDs which allow cameras to see during night time. However since humans emit heat they can be easily spotted with thermal sensors instead.

5.4 Possible Additions in the Future

3D Heat Map Using Multiple Sensors

When using several sensors, it could be possible to create a 3D heat map with all the data similar to figure 28.

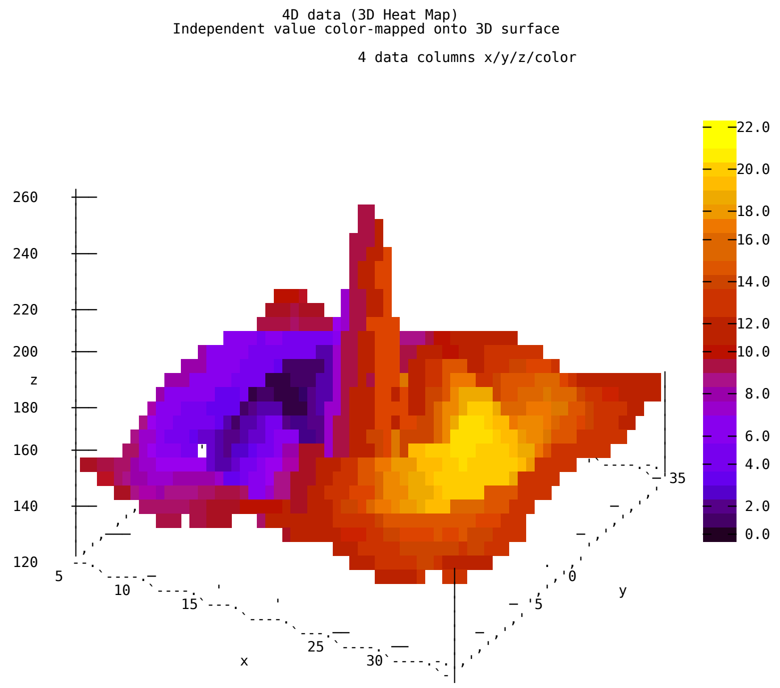

Figure 28: Heat map concept

Heat maps can be used to have a better understanding of how heat behaves in a particular area.

HTML5 Application

Since this project already uses web technologies such as Websocket it will not be difficult to create an HTML5 client application for connecting to the different sensors able to do the same as the Python client. However the user does not need to install anything to make it work, only an up-to-date Internet browser.

Using RTP to Broadcast Video to Clients

The server-side application could be extended to support UDP protocols as RTP which can be used to multi-cast video streams to a big number of clients that for example need the video image but have no control of the sensor [25, 298]. Commands can be still sent through Websocket.

6 Conclusions

As an IoT project the purpose of this thesis was to find an effective communication channel between thermal sensors and clients, in other words, to create a system where clients can control these sensors and receive data simultaneously. The result was a complete communication system:

- A sensor-side: includes a Python application to read the sensor in a Raspberry Pi and service scripts to keep alive the connection to both the network and the Sensor.
- A server-side: uses cloud technologies offered by Nokia and also includes a Node.js application that keeps sensors and clients connected.
- A client-side: includes a Python client application and an Android application able to connect to online sensors and control them.

So far there are some sensors running in the Nokia headquarters to test their usability.

Both LeViteZer and Nokia participants in the project have been satisfied with the results and are willing to continue working on this project beyond the scope of this thesis.

One of the distinctive features about the communication protocol in this project is the use of Websocket outside a web browser which has been proven to work exceptionally well, better than expected. Also with this come some limitations since Websocket is not the best way to broadcast images from sensors to big sets of clients and that is something to be improved.

It was intended to deepen more on the use cases especially on well-being services. But unfortunately there was no time for more research on this. Perhaps other students' thesis projects will continue this part.

References

1 Eurostat. Increase in the share of the population aged 65 years or over between 2005 and 2015;. Available from: http://ec.europa.eu/eurostat/statistics-explained/index.php/File:Increase_in_the_share_of_the_population_aged_65_years_or_over_between_2005_and_2015_(percentage_points)_YB16.png.

2 levitezer; 2017. Available from: http://www.levitezer.com.

3 Nokia. Nokia Innovation Platform; 2017. Available from: https://networks.nokia.com/innovation/platform.

4 Hunt C. TCP/IP Network Administration: Help for Unix System Administrators. O'Reilly Media; 2002. Available from: https://books.google.fi/books?id=A_LL2LQASdoC.

5 Timo Knuutila TK. Net Leap Network. In: Net Leap Network. http://digi.aalto.fi/en/midcom-serveattachmentguid-1e44979621ab084497911e49b7eb59a384bb1cab1ca/netleapnetworkpresentation-digibreakfast.pdf: Aalto University; 2014. p. 10.

6 Fielding R, Gettys J, Mogul J, Frystyk H, Masinter L, Leach P, et al.. RFC 2616, Hypertext Transfer Protocol – HTTP/1.1; 1999. Available from: https://www.ietf.org/rfc/rfc2616.txt.

7 Melnikov A, Fette I. The WebSocket Protocol; 2011. RFC 6455. Available from: https://tools.ietf.org/html/rfc6455.

8 Google. Android Documentation - Services;. Available from: https://developer.android.com/guide/components/services.html.

9 mik3y (https://github com/mik3y). usb-serial-for-android;. Available from: https://github.com/mik3y/usb-serial-for-android#usb-serial-for-android.

10 Oracle. JAVA PLATFORM, MICRO EDITION (JAVA ME); 2017. Available from: http://www.oracle.com/technetwork/java/embedded/javame/index.html.

11 Raspberry pi Foundation. RASPBERRY PI 3 MODEL B;. Available from: https://www.raspberrypi.org/products/raspberry-pi-3-model-b/.

12 Python Software Foundation. Python Documentation - multiprocessing;. Available from: https://docs.python.org/2/library/multiprocessing.html.

13 Rodola G. psutil documentation; 2017. Available from: https://pythonhosted.org/psutil/.

14 Python Software Foundation. Subprocess documentation; 2017. Available from: https://docs.python.org/2/library/subprocess.html.

15 Notes E. What is LabVIEW?; 2016. Available from: https://www.electronics-notes.com/articles/test-methods/labview/what-is-labview.php.

16 Riverbank Computing Limited. What is PyQt?; 2016. Available from: https://riverbankcomputing.com/software/pyqt/intro.

17 John Hunter EFMD Darren Dale, the Matplotlib development team. backend qt4agg; 2016. Available from: http://matplotlib.org/api/backend_qt4agg_api.html.

18 John Hunter EFMD Darren Dale, the Matplotlib development team. Interpolation methods; 2016. Available from: http://matplotlib.org/examples/images_contours_and_fields/interpolation_methods.html.

19 Dayley B. Node.js, MongoDB, and AngularJS Web Development. Developer's Library. Pearson Education; 2014. Available from: https://books.google.fi/books?id=8kTCAwAAQBAJ.

20 Node js Foundation. Overview of Blocking vs Non-Blocking; 2017. Available from: https://nodejs.org/en/docs/guides/blocking-vs-non-blocking/.

21 Bisht S. Robot Framework Test Automation. Packt Publishing; 2013. Available from: https://books.google.fi/books?id=_RO8AQAAQBAJ.

22 Robot Framework Foundation. Robot Framework introduction; 2017. Available from: http://robotframework.org/#introduction.

23 Mell P, Grance T. The NIST Definition of Cloud Computing. National Institute of Standards and Technology: U.S. Department of Commerce; 2011. 800-145.

24 Shipley G, Dumpleton G. OpenShift for Developers: A Guide for Impatient Beginners. O'Reilly Media; 2016. Available from: https://books.google.fi/books?id=qKvODAAAQBAJ.

25 Hardy D, Malleus G, Mereur JN. Networks: Internet · Telephony · Multimedia. Springer Berlin Heidelberg; 2013. Available from: https://books.google.fi/books?id=GZPzCAAAQBAJ.

Acknowledgments

I want to thank Jukka Honkaniemi and Tero Nurminen (Metropolia) for letting me be part of this project and for their guidance. Also I want to thanks Kim Janson (LeViteZer) for all I learned during this project.

Appendix 1: Sensor Reader Listings

```python
import thread
import logging
import psutil
from multiprocessing import Process
from serial import Serial, SerialException
from Constants import VERY_HIGH_PRIORITY, HIGH_PRIORITY

class Serial_reader(Serial):
"""" This class read data from sensor in a Thread """
    def __init__(self, pipe, port):
        Serial.__init__(self, port=port, baudrate=115200)
        self.pipe = pipe
        self._start_process()

    def _start_process(self):
        process = Process(name="SerialProcess", target=self._run, args=())
        process.daemon = True
        process.start()
        try:
            psutil.Process(process.pid).nice(VERY_HIGH_PRIORITY)
        except psutil.AccessDenied as e:
            psutil.Process(process.pid).nice(HIGH_PRIORITY)

    def _get_data(self):
        one_byte = self.read(1)
        n_bytes = self.in_waiting
        return one_byte + self.read(n_bytes)

    def _send_data(self):
        while self.is_open:
            print "waiting for commands"
            data = self.pipe.recv()
            print data
            self.write(data)

    def _run(self):
        thread.start_new_thread(self._send_data, ())
        while self.is_open:
            try:
                data = self._get_data()
                self.pipe.send(data)
            except SerialException as e:
                logging.error(e.message)
```

```
                self.stop()
                break

    def stop(self):
        if self.is_open:
        self.close()
```

Listing 9: Serial Reader Class

```
import logging
import thread
from websocket import WebSocketApp, ABNF
from Constants import URL, CAMERA_PATH, PARAMETERS

class WebSocketConnection(WebSocketApp):
    def __init__(self, pipe, url=URL + CAMERA_PATH + PARAMETERS):
    WebSocketApp.__init__(self, url,
    on_message=self.on_message,
    on_error=self.on_error,
    on_close=self.on_close,
    on_open=self.on_open)
    self.open_connection = False
    self.pipe = pipe

    def on_message(self, ws, message):
        logging.warn("received command:%s, %d bytes", message[0], len(message))
        self.pipe.send(message)

    def on_error(self, ws, error):
        logging.error(error)

    def on_close(self, ws):
        self.open_connection = False
        logging.warn("### closed ###")

    def on_open(self, ws):
        self.open_connection = True
        logging.warn("opened new socket")

        def run():
            while (self.open_connection == True):
                data = self.pipe.recv()
                self.send_data(data)

        thread.start_new_thread(run, ())

    def stop(self):
        self.open_connection = False
```

```
    def send_data(self, data):
        if self.open_connection and len(data) != 0:
        self.send(data, opcode=ABNF.OPCODE_BINARY)

    def set_pipe(self, pipe):
        self.pipe = pipe
```

Listing 10: Websocket Class

```
from thread import start_new_thread
from subprocess32 import call
import logging

def shutdown():
    run_command_async("/sbin/poweroff")

def reboot():
    run_command_async("/sbin/reboot")

def update():
    run_command_async("./build.sh")

def test():
    run_command_async("sleep", "3")

commands = {
'rs': shutdown,
'rr': reboot,
'ru': update
}

def run_command_async(*args):
    def run(*args):
        result = call(args)

    start_new_thread(run, args)


def is_raspberry_command(s):
""" Check this is a command for the raspberry rather than the sensor. First letter
     must be an 'r'"""
    if s[0] == 'r':
        command = commands.get(s)
        if callable(command):
            command()
        else:
            logging.warn("command not found: %s" % s)
```

```
            return True
    else:
        return False
```

Listing 11: Subprocess example

Appendix 2: Android Client listings

```java
package es.alvaroweb.ircamerareader.wscameraview;

import android.content.Context;
import android.graphics.Bitmap;
import android.graphics.Canvas;
import android.graphics.Color;
import android.support.annotation.Nullable;
import android.util.AttributeSet;
import android.util.Log;
import android.view.View;
import android.support.v7.widget.AppCompatImageView;
import java.util.Random;
import okio.ByteString;

/**
* Copyright (C) 2016 Alvaro Bolanos Rodriguez
*/

public class CameraView extends AppCompatImageView implements WebsocketConnection.
    OnReceiveRow, HighCamera.FrameCallback, View.OnClickListener {

    private static final String DEBUG_TAG = CameraView.class.getSimpleName();
    private static Random random = new Random();
    private Bitmap bitmap;
    private int sizex = 160;
    private int sizey = 120;
    private WebsocketConnection websocketConnection;
    private HighCamera highCamera;
    private Runnable t;
    private boolean reversed = true;


    public CameraView(Context context) {
        super(context);
        initBitmap();
    }

    public CameraView(Context context, @Nullable AttributeSet attrs) {
        super(context, attrs);
        initBitmap();
    }

    private void initBitmap() {
```

```
        bitmap = Bitmap.createBitmap(sizex, sizey, Bitmap.Config.ARGB_8888);
        highCamera = new HighCamera(this);
        this.setOnClickListener(this);
    }

    @Override
    protected void onDraw(Canvas canvas) {
        super.onDraw(canvas);
        // Bitmap has to be set here. and this callback is called from UI
        this.setImageBitmap(bitmap);
    }


    public void setImage(byte[][] array) {
        boolean dimensionMaches = array[0].length == bitmap.getWidth() &&
        array.length == bitmap.getHeight();

        if (!dimensionMaches) {
            Log.d("image", "doesn't match the dimension");
            return;
        }

        for (int i = 0; i < bitmap.getHeight(); i++) {
            for (int j = 0; j < bitmap.getWidth(); j++) {
                int pixel = convertByteToInt(array[i][j]);
                if (reversed) {
                    bitmap.setPixel(j, bitmap.getHeight() - i - 1, Color.rgb(pixel
                        , pixel, pixel));
                } else {
                    bitmap.setPixel(j, i, Color.rgb(pixel, pixel, pixel));
                }
            }
        }
    }

    public void setRandomImage(View view) {
        byte[][] arr = new byte[sizey][sizex];
        for (int i = 0; i < sizey; i++) {
            for (int j = 0; j < sizex; j++) {
                arr[i][j] = ((byte) randint(Byte.MIN_VALUE, Byte.MAX_VALUE));
            }
        }
        setImage(arr);
    }

    public void cleanImage() {
        for (int i = 0; i < bitmap.getWidth(); i++) {
            for (int j = 0; j < bitmap.getHeight(); j++) {
```

```
                bitmap.setPixel(i, j, Color.rgb(255, 255, 255));
            }
        }
    }

    private int convertByteToInt(byte b) {
        return b & 0xff;
    }

    private int randint(int min, int max) {
        return random.nextInt(max + 1 - min) + min;
    }

    public void connectTo(String uri) {
        Log.d(CameraView.class.getSimpleName(), "uri received: " + uri);
        websocketConnection = new WebsocketConnection(uri, this);
    }

    public void stopWebsocket() {
        websocketConnection.close();
    }

    @Override
    public void receiveRows(ByteString data) {
        if (data.size() < 1) {
            return;
        }
        highCamera.consumeData(data);
    }

    @Override
    public void frameReady(byte[][] frame) {
        this.setImage(frame);
    }


    @Override
    public void onClick(View view) {
        reversed = !reversed;
    }

    public interface UpdateArray {
        void updateArray();
    }
}
```

Listing 12: CameraView class, extending ImageView

```java
package es.alvaroweb.ircamerareader.wscameraview;

import android.util.Log;
import com.google.common.primitives.Bytes;
import java.util.Arrays;
import java.util.LinkedList;
import java.util.List;
import okio.ByteString;

/**
* 00 01 02 04 ------
Data is arranged on 240(0xF0) rows of 84 bytes(FF FF FF n_row and 80 of data):
FF FF FF 01 <DATA>
FF FF FF 02 <DATA>
FF FF FF 03 <DATA>
...
FF FF FF F0 <DATA>
FF FF FF <TELEMETRY> (38 Bytes)
Where the 4th byte is the number of the row
Every row of the actual picture has 2 rows of the raw data
so the image is 160 x 120 (20198 Bytes)
*/

public class HighCamera {
    private static final int TELEMETRY_ROW_NUMBER = 240;
    private static final int BYTES_IN_ROW_NUMBER = 81;
    private static final String DEBUG_TAG = HighCamera.class.getSimpleName();

    private byte[][] frame;
    FrameCallback frameCallback;
    private byte[] remains;
    private byte[] delimiter = new byte[]{-1,-1,-1};

    public HighCamera(FrameCallback callback) {
        frame = new byte[120][160];
        frameCallback = callback;
        remains = new byte[]{};
    }

    public void consumeData(ByteString data){

        List<byte[]> pieces = delimiterData(Bytes.concat(remains, data.toByteArray()), delimiter);

        int lastIndex = pieces.size() - 1;
        for(int i = 0; i < pieces.size(); i++){
            if(i == lastIndex) continue;
            processRow(pieces.get(i));
```

```
        }
        remains = pieces.get(lastIndex);
    }


    public void processRow(byte[] row){
        if(row.length < BYTES_IN_ROW_NUMBER){
            return;
        }
        int rowNumber = byteToInt(row[0]);
        Log.d(DEBUG_TAG, "rownumber:"+rowNumber);
        if (rowNumber < TELEMETRY_ROW_NUMBER){
            getFrameData(rowNumber, row);
        }else{
            getTelemetryData(row);
        }
    }

    private void getTelemetryData(byte[] row) {
        // todo telemetry
        frameCallback.frameReady(frame);
    }

    private void getFrameData(int rowNumber, byte[] row){
        for(int i = 0; i < BYTES_IN_ROW_NUMBER - 1; i++){
            int ind = i + 1;
            int frameRow = rowNumber / 2;
            int frameCol = rowNumber % 2 * (BYTES_IN_ROW_NUMBER -1)+ ind;
            try{
                frame[frameRow][frameCol] = row[ind];
            }catch (ArrayIndexOutOfBoundsException e){
                Log.e(DEBUG_TAG, e.getLocalizedMessage());
            }
        }
    }

    private int byteToInt(byte b){
        return b & 0xff;
    }


    private List<byte[]> delimiterData(byte[] array, byte[] delimiter) {
        List<byte[]> byteArrays = new LinkedList<>();
        if (delimiter.length == 0) {
            return byteArrays;
        }
        int begin = 0;

```

```
        outer:
        for (int i = 0; i < array.length - delimiter.length + 1; i++) {
            for (int j = 0; j < delimiter.length; j++) {
                if (array[i + j] != delimiter[j]) {
                    continue outer;
                }
            }
            byteArrays.add(Arrays.copyOfRange(array, begin, i));
            begin = i + delimiter.length;
        }
        byteArrays.add(Arrays.copyOfRange(array, begin, array.length));
        return byteArrays;
    }

    interface FrameCallback{
        void frameReady(byte[][] frame);
    }
}
```

Listing 13: HighCamera class, represents a frame

```
package es.alvaroweb.ircamerareader.wscameraview;

import android.util.Log;
import okhttp3.OkHttpClient;
import okhttp3.Request;
import okhttp3.Response;
import okhttp3.WebSocket;
import okhttp3.WebSocketListener;
import okio.ByteString;


/**
* Copyright (C) 2016 Alvaro Bolanos Rodriguez
*/

public class WebsocketConnection extends WebSocketListener {
    private static final String DEBUG_TAG = WebsocketConnection.class.
        getSimpleName();
    private final WebSocket webSocket;
    private final Request requestToServer;
    private OnReceiveRow callback;
    public WebsocketConnection(String serverURI, OnReceiveRow callback) {
        OkHttpClient client = new OkHttpClient();
        requestToServer = new Request.Builder().url(serverURI).build();
        webSocket = client.newWebSocket(requestToServer, this);
        this.callback = callback;
    }
```

```java

    @Override
    public void onOpen(WebSocket webSocket, Response response) {
        super.onOpen(webSocket, response);
        log("onOpen: " + response.message());
    }

    @Override
    public void onMessage(WebSocket webSocket, String text) {
        super.onMessage(webSocket, text);
        log("onMessage: " + text);

    }

    @Override
    public void onMessage(WebSocket webSocket, ByteString bytes) {
        super.onMessage(webSocket, bytes);
        log("onMessage: " + bytes.size() + "bytes received");
        callback.receiveRows(bytes);
    }

    @Override
    public void onClosing(WebSocket webSocket, int code, String reason) {
        super.onClosing(webSocket, code, reason);
        log("onClosing: " + reason + ", code:" + code);
    }

    @Override
    public void onClosed(WebSocket webSocket, int code, String reason) {
        super.onClosed(webSocket, code, reason);
        log("onClosed: " + reason + ", code:" + code);
    }

    @Override
    public void onFailure(WebSocket webSocket, Throwable t, Response response) {
        super.onFailure(webSocket, t, response);
        Log.e(DEBUG_TAG, "onFailure: " + t.getMessage());
        t.printStackTrace();

    }

    public void send(byte[] bytes){
        ByteString byteString = ByteString.of(bytes);
        webSocket.send(byteString);
    }

    public void close(){
        webSocket.close(1000, "fulfilled");
```

```java
    }

    private void log(String s){
        Log.d(DEBUG_TAG, s);
    }

    interface OnReceiveRow{
        void receiveRows(ByteString bytes);
    }
}
```

Listing 14: WebsocketConnection class, makes a websocket connection

Appendix 3: Node.js application listings

```
"use strict";

const express = require('express');
var WebsocketConnections = require('./websocketConnections');
var WebSocket = require('ws');
var url = require('url');
var http = require('http');
var params;

console.log("version 1.0");
var port = process.env.PORT || process.env.port || process.env.OPENSHIFT_NODEJS_PORT || 8080;
var ip = process.env.OPENSHIFT_NODEJS_IP || process.argv[2] || '0.0.0.0';

var PASSWORD = process.env.WS_PASSWORD;
var camDataPath = "/camera";
var clientDataPath = "/client";
var camConnections = new WebsocketConnections.CameraConnections();

/** http server: base */
const app = express();
app.get('/cams', function(req, res){
    res.send({ cams: camConnections.getInfo(), count: camConnections.count()});
});
const server = http.createServer(app);
main(server)

/** @function
*  @param {http.Server} server */
function main(server) {
    /** websocket server extends the http server */
    var wss = new WebSocket.Server({
        verifyClient: verifyClient,
        server: server
    });

    console.log("running on %s:%d", ip, port);

    wss.on('connection', function connection(ws) {
        var parsedUrl = url.parse(ws.upgradeReq.url);
        var path = parsedUrl.pathname;

        switch (path) {
```

```
            case camDataPath:  // a camera wants to register
            var camera_name = params.camera_name || undefined;
            var req = ws.upgradeReq;
            var ipAddress = req.headers['x-forwarded-for'] ||
            req.connection.remoteAddress ||
            req.socket.remoteAddress ||
            req.connection.socket.remoteAddress;
            camConnections.add(ws, camera_name, ipAddress);
            break;
            case clientDataPath:  // a client wants to register to a camera
            case "/":
            var camera_name = params.camera_name || "camera0";
            camConnections.addClientToCamera(camera_name, ws, function(err){
                if(err){ws.terminate();}
            });
            break;
            default:
            console.log("rejected: no valid path");
            ws.terminate();
            return;
        }
    });

    server.listen(port, ip);
}


function verifyClient(info) {
    var acceptHandshake = false;
    var accepted = "rejected: no valid password, use 'pass' parameter in the
        handshake please";
    var ip = info.req.connection.remoteAddress;
    var clientUrl = url.parse(info.req.url, true);
    params = clientUrl.query;

    acceptHandshake = params.pass == PASSWORD;

    if (acceptHandshake) {
        accepted = "accepted";
    }
    console.log("new client %s: %s", accepted, info.req.url);
    return acceptHandshake;
}
```

Listing 15: Main file

```
var WebSocket = require('ws');

/** A class that hold WebSocket clients for a camera
```

```
* @class
*/
function ClientConnections() {
    this.clients = [];
}
/** Lenght of the internal array of clients
* @method
*/
ClientConnections.prototype.getLength = function(){
    return this.clients.length;
};
/**@method
* @param {WebSocket} conn - client websocket connection to add
*/
ClientConnections.prototype.add = function (conn) {

    this.clients.push(conn);
    // console.log("client connections:%d", this.clients.length)
};
/**@method
* @param {WebSocket} conn - websocket connection to close
*/
ClientConnections.prototype.close = function (conn) {
    if(this.clients.length < 1){
        return;
    }
    var indx = this.clients.indexOf(conn);
    this.clients.splice(indx, 1);
};

/**send message to all websockets in the array
* @method
* @param {string} message
*/
ClientConnections.prototype.sendToAll = function (message) {
    this.clients.forEach(function (client, ind, arr) {
        checkSocketOpen(client, function(){
            client.send(message);
        });
    });
};

/**Close all clients in the array
* @method
*/
ClientConnections.prototype.closeAll = function (message) {
    this.clients.forEach(function (client, ind, arr) {
        try{
```

```
            client.close();
        }catch(err){
            console.error(err.message);
        }
    });
};


/**A class that holds CameraClient objects
* @class
*/
function CameraConnections() {
    /** @member {Array} - this an array of clientcameras, no websockets
        connections */
    this.cameras = [];
}

/**
* @method
* @return {number} - number of cameras in the connected
*/
CameraConnections.prototype.count = function() {
    return this.cameras.length;
};


/**
* @method
* @return {array} - array of names of the cameras
*/
CameraConnections.prototype.getInfo = function() {
    var cams = [];
    this.cameras.forEach(function(element, index) {
        var name = element.name;
        if (element.name === undefined){
            name = "camera"+index;
        }
        var infoObject = {name:name, ip:element.ip};
        console.log(infoObject)
        cams.push(infoObject);
    }, this);
    return cams;
};

/**
* @method
* @param {WebSocket} conn - connection to add
* @param {string} name - name of the socket
```

```
*/
CameraConnections.prototype.add = function (conn, name, ip) {
    var cname = name;
    var self = this;
    if (!cname) {
        cname = undefined; //name will be based on index
    }

    // defining the callbacks for this camera
    conn.on('message', incomingFromCamera);
    conn.on('close', closingCamera);

    var camera = new Camera(conn, cname, ip);
    this.cameras.push(camera);

    /** Callled when a connection to a camera is closed
    * @callback */
    function closingCamera(code, message) {
        try{
            camera.clients.closeAll();
            self.removeCamera(camera);
        }catch(err){
            console.error("Error on closing clients:"+err.message);
        }
        console.log("Camera %s closing connection: %d, %s", camera.name, code, message);
    }
    /** Called when data from a camera is comming
    * @callback */
    function incomingFromCamera(message, flags) {
        try {
            camera.clients.sendToAll(message);
        } catch (e) {
            console.error(e);
        }
    }
    return camera;
};
/**
* @method
* @param {(string|object|function)} cameraName - can be the name of the camera or a Camera object
* @param {WebSocket} clientConn
* @param {} callback
*/
CameraConnections.prototype.addClientToCamera = function (cameraName, clientConn, callback) {
    if(typeof cameraName === 'string'){
```

```
        this.getCamera(cameraName, tryAddClientToCamera);
    }else{
        tryAddClientToCamera(cameraName);
    }

    function tryAddClientToCamera(camera){
        if(!camera){
            var err = new Error("cannot add client to invalid camera: "+camera);
            callback(err);
            return;
        }

        // new client Callbacks
        clientConn.on('message', incomingFromClient);
        clientConn.on('close', closingClient);
        camera.clients.add(clientConn);
        callback();

        /** Called when a client sent data
        * @callback
        * @param {string} message
        * @param {object} flags
        */
        function incomingFromClient(message, flags) {
            camera.sendMessage(message);
        }
        /** Called when a connection to a client is closed
        * @callback
        * @param {number} code
        * @param {string} message
        */
        function closingClient(code) {
            camera.clients.close(clientConn);
            console.log("Closing client connection for: %s camera. info: %d, %s",
                camera.name, code);
        }

    }
};

CameraConnections.prototype.removeCamera = function(cameraClient){
    var indx = this.cameras.indexOf(cameraClient);
    this.cameras.splice(indx, 1);
};

CameraConnections.prototype.close = function(camera){
    // this will trigger closingCamera callback
    this.removeCamera(camera);
```

```
};
/**
* @method
* @param {string} name - name of the camera
* @param {} callback - callback which receives the camera object
*/
CameraConnections.prototype.getCamera = function (name, callback) {
    var cameraFound;
    this.cameras.forEach(function(c, index) {
        if (c.name == name || name == "camera"+index) {
            cameraFound = c;
        }
    }, this);

    callback(cameraFound);
};


/**@class
*  A camera client, it has a list of clients attached, and a unique name
*  @param {WebSocket} conn - connection object
*  @param {String} name - name of this camera (for identification)
*  @param {String} ip - ip address of this camera
*/
function Camera(conn, name, ip) {
    this.conn = conn;
    this.name = name;
    this.ip = ip;
    this.clients = new ClientConnections();
}

Camera.prototype.sendMessage = function (message) {
    var conn = this.conn;
    checkSocketOpen(conn, function(){
        conn.send(message);
    });
};

exports.ClientCamera = Camera;
exports.ClientConnections = ClientConnections;
exports.CameraConnections = CameraConnections;

/**
*
* @param {WebSocket} socket
* @param {} callback
*/
function checkSocketOpen(socket, callback){
```

```
    if(!socket){
        console.error('socket does not exists');
        return;
    }
    if(socket.readyState == WebSocket.OPEN){
        callback();
    }
}
```

Listing 16: implementation of server-side classes

```
*** Variables ***
${password}                 "password here"
${url}                      localhost:8080
${camera_name_param}        camera_name=
${url_params}               ?pass=${password}
${uri_client}               ws://${url}/client${url_params}&${camera_name_param}
${uri_camera}               ws://${url}/camera${url_params}&${camera_name_param}
${cloud_path}                ../
${cloud_app}                 npm start --prefix  ${cloud_path}
${outf}                     log/stdout.txt
${errf}                     log/stderr.txt


*** Settings ***
Library             lib/WebsocketLibrary.py
Library             OperatingSystem
Library             Process
Library             HttpLibrary.HTTP
Suite Setup         Run Cloud
Suite Teardown      Close Cloud
Test Teardown       sleep  200 ms

*** Keywords ***
Close cloud
    Terminate All Processes
Run Cloud
    Remove Files                ${outf}  ${errf}
    Set Environment Variable    WS_PASSWORD  30022
    Start Process               ${cloud_app}   alias=cloud_process  stdout=${outf}  stderr=${errf}  shell=True
    sleep   1
    ${is_running} =             Is Process Running  handle=cloud_process
    Should Be True              ${is_running}   msg=Cloud is not running
Wait To Receive Message
    [arguments]   ${socket}  ${message}
    Wait Until Keyword Succeeds  5x  5 ms  Receive Next Message   ${socket}  ${message}
Wait Until Queue
```

```
    [arguments]   ${socket}   ${n}
    Wait Until Keyword Succeeds   5x  50 ms  Messages In Queue Should Be   ${
        socket}   ${n}
Create Camera Socket
    [arguments]   ${name}
    create socket  ${name}  ${uri_camera}${name}
    Create Camera Socket Noname
    create socket  noname  ${uri_camera}
Create Client Socket
    [arguments]   ${name}   ${camera_socket}
    create socket  ${name}  ${uri_client}${camera_socket}
Random Message
    ${randint}  Evaluate   str(random.randint(0, sys.maxint))   modules=random,
        sys
    [Return]   ${randint}
Send Random Message From
    [arguments]  ${socket}
    ${message}   Random Message
    Send From Socket  ${socket}  ${message}
Get Cameras
    Create Http Context   ${url}   http
    GET   /cams
    Response Status Code Should Equal   200
    ${body} =   Get Response Body
    Should Start With   ${body}   {
    Log Json  ${body}
Number Of Cameras Should Be
    [arguments]   ${n}
    Create Http Context   ${url}   http
    GET   /cams
    Response Status Code Should Equal   200
    ${body} =   Get Response Body
    Json Value Should Equal  ${body}  /count  ${n}
    Log Json  ${body}


    *** Test Cases ***
    close sockets
    create Camera Socket         camera0
    Create Client Socket         client          camera0
    sleep                                100 ms
    Close Socket                 client
    Create Client Socket         client2          camera0
    sleep                                100 ms
    Close Socket                 client2


    Check Socket Library works
```

```
    Create Socket                client0  ${uri_camera}
    Do Exist Socket             client0

    Create camera client and send
    Create Camera Socket         camera0
    Create Client Socket         client0  camera0
    send From Socket             camera0  hi
    send From Socket             client0  hello

    cameras without name and with name
    create Camera Socket Noname
    create Camera Socket Noname
    create Camera Socket Noname
    create Camera Socket         1111special-cam1111
    create Camera Socket Noname
    create Camera Socket Noname
    sleep  80 ms
    Number Of Cameras Should Be    6


    5 cameras, 1 client, 5 messages
    create Camera Socket         camera0
    create Camera Socket         camera1
    create Camera Socket         camera-special
    create Camera Socket         camera3
    create Camera Socket         camera4
    Create Client Socket         client          camera-special
    sleep                                100 ms
    Get Cameras

    Send Random Message From             camera-special
    Send Random Message From             camera-special
    Send Random Message From             camera-special
    Send Random Message From             camera-special
    Send Random Message From             camera-special
    Wait Until Queue          client   5



    several cameras with several clients, bidirectional communication
    Create Camera Socket        cam
    Create Camera Socket        camf
    Create Client Socket        client       cam
    Create Client Socket        client1      cam
    Create Client Socket        client2      camf
    Create Client Socket        client3      camf
    sleep                                100 ms
    Get Cameras
```

```

    Send Random Message From            cam
    Send Random Message From            camf

    Wait Until Queue          client    1
    Wait Until Queue          client1   1
    Wait Until Queue          client2   1
    Wait Until Queue          client3   1

    Send Random Message From            client
    Send Random Message From            client1
    Send Random Message From            client2
    Send Random Message From            client3

    Wait Until Queue          cam       2
    Wait Until Queue          camf      2

    Http Server
    Get Cameras
```

Listing 17: RobotFramework test file

```python
#!/usr/bin/python
import subprocess
import sys
from Queue import Queue, Empty
from threading import Thread
import websocket
from robot.api import logger

__version__ = '0.1'
__author__ = "Alvaro Bolanos Rodriguez"


class WebsocketLibrary:
    ROBOT_LIBRARY_SCOPE = 'TEST CASE'
    ROBOT_LISTENER_API_VERSION = 2

    def __init__(self):
        self.ROBOT_LIBRARY_LISTENER = self
        # self.url = "%s:%d" % (host, port)
        self.socketDic = {}

    def _start_suite(self, name, attrs):
        print 'started suite'

    def _end_suite(self, name, attrs):
        print 'Suite %s (%s) ending.' % (name, attrs['id'])

```

```
    def _start_test(self, name, attrs):
        pass
        def _end_test(self, name, attrs):
        self.stop_all_sockets()

    def _get_socket(self, name):
        try:
            return self.socketDic.get(name)
        except Exception as e:
            logger.error(e.message)

    raise Exception("%s socket not found in list" % name)

    def create_socket(self, name, uri):
        ws = self.WebSocketConnection(uri, name=name)
        self.socketDic[name] = ws
        logger.info("created %s using %s" % (name, uri))

    def do_exist_socket(self, name):
        if self.socketDic.has_key(name):
            logger.info("'%s' exists" % name)
        else:
            raise AssertionError("'%s' does not exist" % name)

    def close_socket(self,name):
        s = self._get_socket(name)
        s.stop()

    def send_from_socket(self, socket, message):
        try:
            s = self._get_socket(socket)
            s.send_to_socket(message)
            logger.info("%s is sending '%s' message" % (s.name, message) )
        except websocket.WebSocketConnectionClosedException as e:
            logger.warn(e.message + ".Try using 'Wait to' keyword style")

    def stop_all_sockets(self):
        for name, socket in self.socketDic.items():
        socket.stop()
        self.socketDic = {}

    def receive_next_message(self, name, expected):
        ws = self._get_socket(name)
        try:
            received_message = ws.receive_next_message()
        except Empty as e:
            msg = e.message+"Message is not in Queue yet"
            logger.warn(msg)
```

```
            raise AssertionError(msg)
        if not ws:
            raise AssertionError("there is no websocket")
            if not received_message == expected:
                msg = "Messages do not match:'%s' is not '%s'" % (received_message, expected)
                logger.warn(msg)
                raise AssertionError(msg)

    def messages_in_queue_should_be(self, name, n):
        expected = int(n)
        s = self._get_socket(name)
        actual = s.in_queue.qsize()
        if actual != expected:
            raise AssertionError("number of elements does not match, was %d, expected %d" % (actual, expected))


class WebSocketConnection(Thread):
    def __init__(self, url, name):
        Thread.__init__(self, name=name)
        # websocket.enableTrace(True)
        self.url = url
        self.name = name
        self.in_queue = Queue(10)
        self.ws = websocket.WebSocketApp(self.url,
        on_message=self.on_message,
        on_error=self.on_error,
        on_close=self.on_close,
        on_open=self.on_open)
        self.setDaemon(True)
        self.start()

    def run(self):
        self.ws.run_forever()

    def on_message(self, ws, message):
        self.in_queue.put(message)
        logger.info("from %s:%s" % (self.name, message))

    def on_error(self, ws, error):
        logger.error(error)

    def on_close(self, ws):
        logger.info("closed %s" % self.name)

    def on_open(self, ws):
        logger.info("opened %s" % self.name)
```

```
    def stop(self):
        self.ws.close()

    def send_to_socket(self, data):
        self.ws.send(data)

    def receive_next_message(self):
        return self.in_queue.get(block=False)
```

Listing 18: RobotFramework websocket library