Universidade de Aveiro
2013

Departamento de Eletrónica,
Telecomunicações e Informática
Elaborada na Universidade Técnica de Hamburgo
Instituto de Telemática

Francisco
de Gouveia

Transmissão e apresentação de conteúdos de sensores médicos

Transmission and presentation of medical sensor-data

Übertragung und Darstellung von medizinischen Sensor-Daten

erasmus

TUHH
Technische Universität Hamburg-Harburg

Universidade de Aveiro
2013

Departamento de Eletrónica,
Telecomunicações e Informática
Elaborada na Universidade Técnica de Hamburgo
Instituto de Telemática

Francisco
de Gouveia

Transmissão e apresentação de conteúdos de sensores médicos

Transmission and presentation of medical sensor-data

Übertragung und Darstellung von medizinischen Sensor-Daten

Dissertação apresentada à Universidade Técnica de Hamburgo ao abrigo do programa ERASMUS para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Sistemas de Informação na Universidade de Aveiro, realizada sob a orientação científica do Doutor (Prof. Dr. Volker Turau), Professor do Instituto de Telemática da Universidade Técnica de Hamburgo, de Martin Ringwelski, investigador assistente do Instituto de Telemática da Universidade Técnica de Hamburgo e do Prof. Dr. José Maria Fernandes, Professor auxiliar do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro.

Declaration of Authorship

I declare that this thesis and the work presented in it are my own and have been generated as the result of my own original research. Each significant contribution to it and quotation from the work of other people has been attributed and referenced. This thesis has not been previously submitted in whole or in part for the award of any degree.

Date: 08.10.2013 Signature:

o júri / the jury

presidente / president	Prof. Dr. Volker Turau Professor at Hamburg University of Technology
vogais / examiners committee	Prof Dr. José Maria Fernandes Professor at University of Aveiro

Palavras Chave

transmissão de dados, monitorizaçao de sensores, sensores médicos, aa4r, vitalresponder

Resumo

Este documento apresenta o design, implementação e avaliação de um sistema que recebe, processa e apresenta emissões instantâneas de dados vitais de sensores ligados a uma pessoa ou ao meio em que esta se insere. Este é usado para prevenção, análise e/ou acção perante uma situação de emergência. Dados os cenários críticos one o sistema pode ser usado, este é composto por uma arquitectura distribuída, com o intuito de reduzir o risco de o sistema parar por alguma falha, e dar a possibilidade de expandir ou reduzir a capacidade de processamento de acordo com a necessidade de utilização. Além disso, é também um sistema completamante modular e suporta o desevolvimento de módulos com novas funcionalidades ou suporte para diferentes tipos de sensores. A sua interface Web permite o acesso ao sistema, independentemente da plataforma utilizada, desde que esta tenha um browser. Está preparada para ter um design responsivo, de acordo com o tamanho do ecrã do dispositivo, seja um telemóvel, um tablet ou um computador de mesa. Dada a maturidade das aplicações e serviços Web disponíveis, é fácil extender também a interface para suportar novoso tipos de visualizações de informação.

Keywords data transmission, sensor monitoring, medical sensors, aa4r, vitalresponder

Abstract

This document covers the design, implementation and evaluation of a system that receives, processes and presents live streams of vital signs from sensors attached to a person's body or in his surrounding environment. This is used either to prevent, analyse and/or act upon a critical scenario of emergency. Due to this critical scenarios where the system can be used, a distributed approach is implemented. Its aim is to reduce the risk of failure and give the possibility of transparent resource scaling according to the needs. Moreover, it is fully modularized for feature extensability and multiple sensor type support. Its Web interface is meant to provide a multi-platform access to the system, as long as the platform has a browser installed. It has a responsive design, according to the screen size of the client device, be it a smartphone, a tablet or a desktop computer. Given the maturity of Web applications and services available, it is easy to add the support for different visualization frameworks or services.

Keywords

Datenübertragung, Sensor Überwachung, medizinische Sensoren, aa4r, vital-responder

Abstract

Diese Arbeit behandelt Design, Implementierung und Evaluation eines Systems, das live übertragene Vitalparameter von Sensoren empfängt, verarbeitet und darstellt, die an einem menschlichen Körper angebracht sind oder in seiner Umgebung. Es wird genutzt um kritischen Unfallszenarien vorzubeugen, sie zu analysieren und/oder auf sie zu reagieren. Aufgrund dieser kritischen Szenarien, in denen das System genutzt werden kann, wird ein verteilter Ansatz implementiert. Das Ziel ist es die Fehlerrate zu reduzieren und, bedarfsabhängig, die Möglichkeit zur tranparenten Skalierung der Ressourcen zu geben. Desweiteren ist das System voll modularisiert, um Erweiterbarkeit und die Unterstützung vieler Sensortypen zu gewährleisten. Das Webinterface bietet Zugang von verschiedensten Plattformen, solange ein Browser installiert ist. Es hat ein responsives Webdesign, dass sich and die Bildschirmgröße jedes Nutzergerätes anpasst, sei es ein Smartphone, Tablet oder Desktop Computer. Der gegebene Reifegrad von Webapplikationen und -diensten ermöglicht die Unterstützung verschiedener Visualisierungsframeworks oder -dienste.

Contents

List of Figures

Listings

Chapter One

Introduction

1.1 Motivation

The ability to remotely monitor the vital signs of a person and the surrounding environmental conditions can be used in different scenarios. One of this cases is the remote healthcare. Allowing a non-critical patient to be monitored from home can be beneficial, both to the hospital and to the patient. Considering a patient who doesn't require a skilled nurse to accomplish his treatments, one might benefit from the comfort of home, but also from the help and constant support of the family. On the other hand, the hospital will be able to host other patients with critical needs, while keeping track of the condition of patients that are recovering at home. Another advantage is the ability to inform the patient about his condition and advice the right medicine according to the situation, as it would be happening at the hospital's facilities.

A different scenario is to monitor old adults who live alone. In Portugal, it was reported that back in 2011, 2872 old citizens were found dead alone in their homes [1]. Those citizens were only discovered thanks to first responders breaking into their places [1], after reports regarding the missing person. With the monitoring of vital signs, it is possible to know when something is wrong with a person and have an appropriate response to the case.

Currently, there is a project starting that focus on safe rehabilitation, called Ambient Assistance for Recovery (AA4R) [1]. The idea behind is divided into three parts. One of them is the monitoring of the patient's status and its surrounding environment. Another part is the medical server, whom is receiving the monitored data from the patient and sending the processed information to the third entity, for instance, the hospital. The hospital, in turn, will analyse the information and give feedback directly to the patient. Refer to chapter 2.1

[1]Ambient Assistance for Recovery project (`http://www.aa4r.org/`). It involves nine different research institutes from the Hamburg University of Technology (TUHH), including the Institute of Telematics (`http://www.ti5.tuhh.de/`), in cooperation with medical and industrial partners.

for more information about this project.

The previous scenarios are focused on the monitoring of the patients that are either recovering or just being checked at home from the hospital. A different case scenario, where the monitoring of vital signs can be also applied, is the one where people are taken into dangerous extreme situations. For instance, first responders face high risk missions in critical scenarios of emergency which may expose them, among other dangers, to high temperatures and constant stress. Taking in consideration that both of this factors combined can lead to a rapid increase of body core temperature, which is dangerous to human organism [2, p. 70], it is essential to keep constant track of vital signs of a first responder. Moreover, incidents during their missions may cause depression, sleepless or loss of appetite [2, p. 70], which may lead to fatigue during their missions. With a fast response to an anomaly in the vital signs, such as the replacement of a first responder in a bad condition, can prevent serious illnesses or even the death of this professionals.

The project named Vital Responder[2], aims to provide a first response monitoring system [3, p. 397] for extreme scenarios, such as the one previously described. It is constituted by different components, including a wearable device equipped with sensors, that transmits data to a smartphone. This data is then processed with the smartphone application called DroidJacket. This application's functionality is to reduce the content size by using data down sampling, present optionally a visualization on the screen and automatically detect possible alarms. The processed data is then relayed to an external client [3, p. 399] to process the detected alarms and present a visualization of the data streamed from the sensors. Currently, the external client is a base station application called iVital developed for iPad. It allows to visualize the individual status of elements of a team, with up to four elements at once [3, p. 400]. Since this devices are not meant to be used by users experienced on technology, there is an effort to keep the interfaces and interaction easily accessible and intuitive [3, p. 400]. Refer to chapter 2.2 for detailed information about this project.

1.2 Objectives

The aim of this Master's Thesis is to analyse the requirements of both AA4R and VitalResponder projects, regarding the modelling and development of an information system as a solution to remotely monitor patient's or first responder's vital signs and surrounding environmental conditions. The system design should take in consideration the minimization of risk of failure and give the possibility to scale the processing capacity, according to the

[2]Vital Responder project (`http://www.vitalresponder.pt/`) is organized in a consortium, with Electronics and Telematics Engineering Institute of Aveiro (Electronics and Telematics Engineering Institute of Aveiro (IEETA) `http://www.ieeta.pt/`), from University of Aveiro, as the main institution, in partnership with Telecommunications Institute (IT `http://www.it.pt/`), BioDevices S.A. (`http://www.biodevices.pt/`) and Center for Sensed Critical Infrastructure Research (CenSCIR), from Carnegie Mellon University.

quantity of patients and sensors. Furthermore, the system must be able to present to the final user a clear visualization of the information obtained from the sensors' data, and display alerts when they occur. The client implementation should be prepared for multi-platform.

Additionally, a smartphone application will be developed with the purpose of creating a gateway between the sensors and the information system. Currently, the VitalResponder project has its own implementation, the DroidJacket, which obtains the data from the sensors through Bluetooth connection and transmits it to the base station with TCP. For the AA4R project, the implementation should transmit the data with UDP instead.

The UDP implementation will have to take in consideration the unreliability of the protocol. Latencies, packet loss detection and recovery will be analysed with the UDP protocol. Also, the delay between the time of reading values from the sensors up to the presentation to the client interface, should be taken into analysis.

1.3 Chapter structure

The next chapter will introduce into more detail the projects above mentioned (AA4R and Vital Responder) and how this Master's thesis applies to both of them. Requirements are analysed and documented for both cases. Also, the research results on relevant projects in the field are presented.

Later, on the chapter 3, requirements will be merged, conflicts determined and analysed for a final solution. The solution to the requirements will be then documented together with a system design, based on the decision taken.

Before deeping into implementation details, the technologies and tools used for the development are presented in the chapter 4

Regarding the implementation of the information system, chapter 5 will cover the smartphone application development, the system responsible for data receiving, process and storage, as well as the platform-independent client application, meant to present the information to the user. In this chapter, technologies used for the development can be found, with guidelines to the system usage and possibilities for further implementations.

The results of the development of the system will be analysed and documented on chapter 6.

Chapter Two

Background

In this chapter, the ongoing projects are introduced and contextualized in this Master's Thesis. Those projects are the AA4R and the Vital Responder. Additionaly, a research on other relevant projects was made and documented.

2.1 Ambient Assistance for Recovery

As mentioned before, the Ambient Assistance for Recovery is a project starting at the Hamburg University of Technology, involving nine different research institutes, in cooperation with medical and industrial partners. Each of those institutes plays a different role in this project, according to their field of expertise.

The aim of AA4R is to give to the patients the ability to recover at home, while their vital signs are being remotely monitored in the health care institution. Also, the health care institution should be able to give back feedback to the patients or even control remotely the patient's actuators to improve their healing process. This is possible by merging the existent technology with a system that allows to communicate with remote devices, actuators and sensors.

As the figure 2.1 illustrates, there are three main points of communication. One is composed by the entire set of instruments installed at the patient's location or connected at his/her body. They are used for localized tracking of environmental state and tracking of patient's vital signs, respectively. This point is where all the data is gathered and transmitted to another point of communication: the medical server.

At the medical server, the data is processed, recorded and possibly transformed for a further analysis from a specialized team. This is also the point where the healthcare institution (the third point of communication) can access the information and send feedback to the patient.

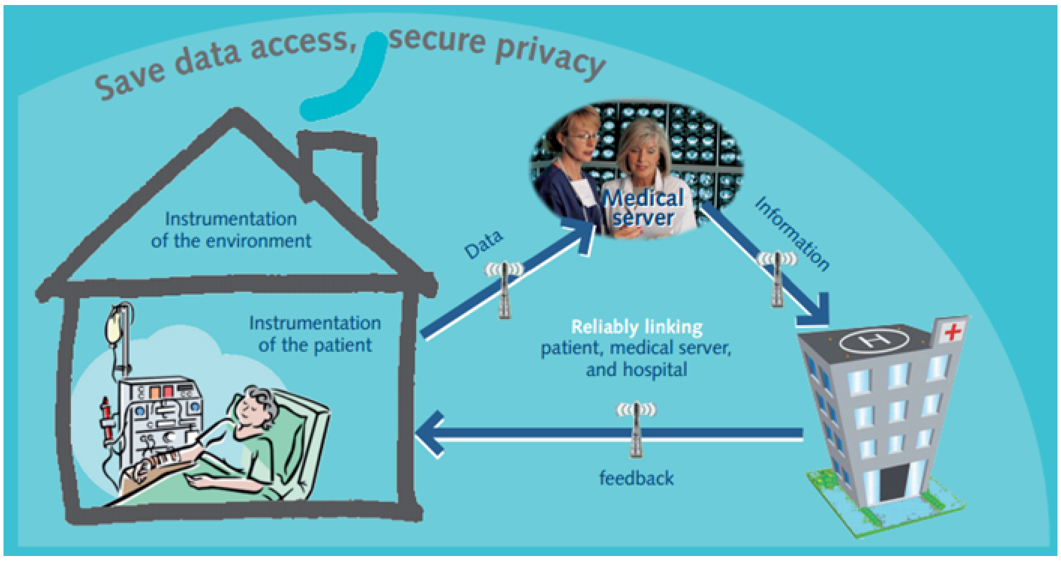

Save data access, secure privacy
Instrumentation of the environment
Instrumentation of the patient
Data
Medical server
Information
Reliably linking patient, medical server, and hospital
feedback

Figure 2.1: Ambient Assistance for Recovery [4]

There are many cases where this could be applied. On orthopaedics, as an example, devices such as intelligent implants with sensors, actuators and a radio frequency module can track and optimize the healing process of a fractured bone. An existent product that could be applied in this case is the iOS/ Intelligent Implant for Osteosynthesis, from livetec, winner of the Innovation Competition of 2007 for the promotion of medical technology. It measures the stiffness of the bone with non-invasive means and transmits it through Radio-frequency transmitter (RF) to an external device [5].

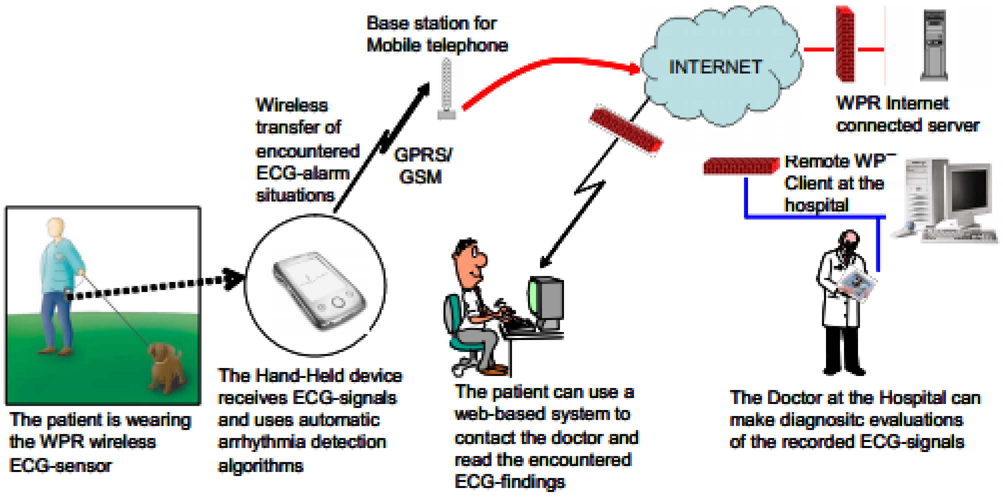

Base station for Mobile telephone
INTERNET
WPR Internet connected server
Wireless transfer of encountered ECG-alarm situations
GPRS/ GSM
Remote WP Client at the hospital
The patient is wearing the WPR wireless ECG-sensor
The Hand-Held device receives ECG-signals and uses automatic arrhythmia detection algorithms
The patient can use a web-based system to contact the doctor and read the encountered ECG-findings
The Doctor at the Hospital can make diagnositc evaluations of the recorded ECG-signals

Figure 2.2: ECG Tele-home-care solution [6]

Cardiology is another use case. In 2005, a paper was published with a concept for a continuous remote recording of Electrocardiogram (ECG) signals, as the figure 2.2 illustrates. It consists on a sticky wireless sensor attached to the patient's chest, continuously transmitting read data to a Hand Held Device (HHD) , via RF. Installed in the HHD, the implemented software automatically connects the device to a GPRS/GSM mobile network and transmits the necessary data to a web server, through the Internet [6]. On this HHD, the doctor can define alarm criteria, so that in case of anomaly detection, 1 minute of ECG is recorded and sent to the server. The doctor, then, has access to this data and can leave feedback in the same system. The patient has also access to the system through a web-interface, where he/she can retrieve the ECG-findings and contact the doctor [6].

2.1.1 Scope in this Master's Thesis

The design and development of an information system that can record, process and show data from different devices, located remotely at each patient's location, was the topic proposed for this Master's Thesis. This information system covers the three communication points described before (see figure 2.1).

Similarly to the concept described on in [6], an application for a HHD will be developed to receive the data from the different sensors (instead of a single ECG sensor), via Bluetooth, and relay them to the web server, using the protocol UDP/IP, via 802.11 or the mobile's broadband connection. This application will be running on the patient's side, and will serve as a Gateway to the patient's sensors and actuators.

For the third communication point, the healthcare institute, a web-interface will be implemented with live data from sensors being presented in the form of charts.

2.1.2 Requirements

For the HHD, an Android application should be developed. It will be responsible for receiving data from the sensors, via Bluetooth, and relaying the data to the web server, via 802.11 or mobile's broadband connection.

On the medical server side, the system should be able to connect to the HHD, receive the UDP/IP data, interpret it, process it and store it in a database. Considering that a single system will be receiving data from different patients, and that each patient might have more than one sensor being monitored and that there are sensors transmitting data with an high-frequency rate (such as ECG), the system should be prepared to handle the high number of connections and packets received. It is convenient to have a system prepared to be scalable, for the case of having a growing number of patients and greater number of sensors per patient in the future. Also, taking in consideration that the health of the patients might be in risk, single points of failure in the system should be avoided or its risk of failure reduced.

Regarding the data transmission, to avoid missing data, a protocol that prevents packet loss should be developed. Also, it is important to implement a protocol that synchronizes the clock of the HHD with the server time. For that, the latency should also be taken into consideration.

Regarding the web interface, it should list the patients of an authenticated doctor. When a patient is selected, his sensors' live data should be presented. For the consistent visualization of the data, the order of the values must be taken into consideration.

2.2 Vital Responder

Thought to be used by first responders during their missions under adverse conditions, Vital Responder project is a set of components that help this professionals to understand, in real time, the human stress and fatigue of their team and detect anomalies with their vital signs. It is a research and development project, lead by Institute of Electronics and Telematics Engineering of Aveiro, from University of Aveiro, in partnership with Instituto de Telecomunicações, BioDevices S.A. and Center for Sensed Critical Infrastructure Research, from Carnegie Mellon University.

The challenges of this project are mainly related with real-time constant information gathering of body signals (which is hard to achieve under the adverse conditions first responders have to face), indoor location tracking of this professionals (which is not possible to achieve with Global Positioning System (GPS) based location systems) and inter-disciplinary research in a team with engineers and clinicians on the origins and nature of physiological stress [7].

2.2.1 Components

The current implementation of Vital Responder project [7] is composed by four main components, introduced and explained in the following sub-sections.

Vital Jacket

Designed to be used in many critical scenarios that require movement, Vital Jacket® was a project for a vital signs' monitoring system. It was meant to be wearable, while maintaining the high quality of vital signs' tracking with continuous or high frequent sampling [8].

The concept of having an wearable device was based on former work made by the IEETA - R&D non-profitable organization, from University of Aveiro [8][9]. The institute developed all the microelectronics, informatics and mobile communications. The textile work was developed together with the Technological Centre for the Textile and Clothing Industries of Portugal (CITEVE) , a non-profilable association from the Portuguese textile industry.

The prototype of the project is a jacket with sensors attached to it. It has a pocket where the device which receives and relays the data from all the sensors will be inserted (see section 2.2.1). The prototype can be seen in the picture a) at the figure 2.3.

Back in 2007, the project development of Vital Jacket® was licensed to a biomedical engineering spin-off company named BioDevices, S.A.. This company further developed the project, reducing the size of the jacket, turning it into a t-shirt, as can be seen in the b) picture at the figure 2.3. The t-shirt is able to monitor the ECG, hearth rate and body position, using ECG leads and a 3-axis accelerometer.

This is a certified product used for sports, clinical scenarios and emergency situations [10]. It is integrated into Vital Responder project to monitor the first responders' vital signs.

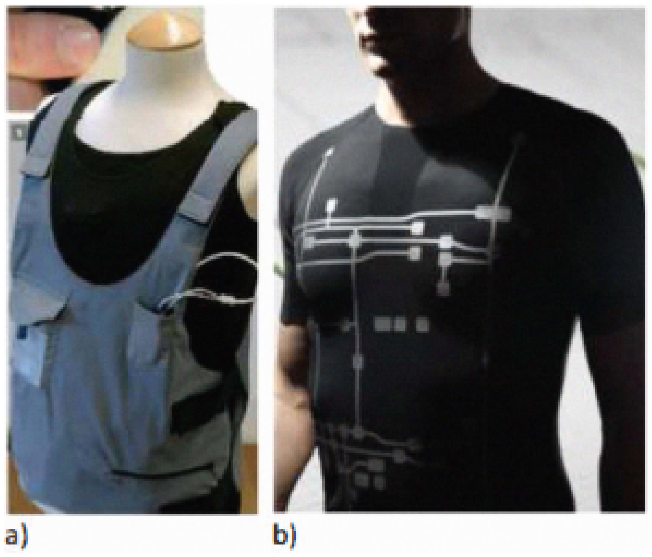


Figure 2.3: Vital Jacket prototype (a) and commercial version (b) [8]

Vital Jacket Box

The Vital Jacket®'s sensors are connected through wires to a single device, named Vital Jacket®Box (see figure 2.4). This device relays the received data from the sensors to another device through Bluetooth connection. Additionally, it can store the received data into a SD-Card for further offline analysis.

Figure 2.4: Vital Jacket Box [11]

Droid Jacket

To receive the data sent from the Vital Jacket®Box, an Android HHD is used. This device has an application called Droid Jacket installed. This application has several components, which include user management (for team support), data processing, information visualization and data server.

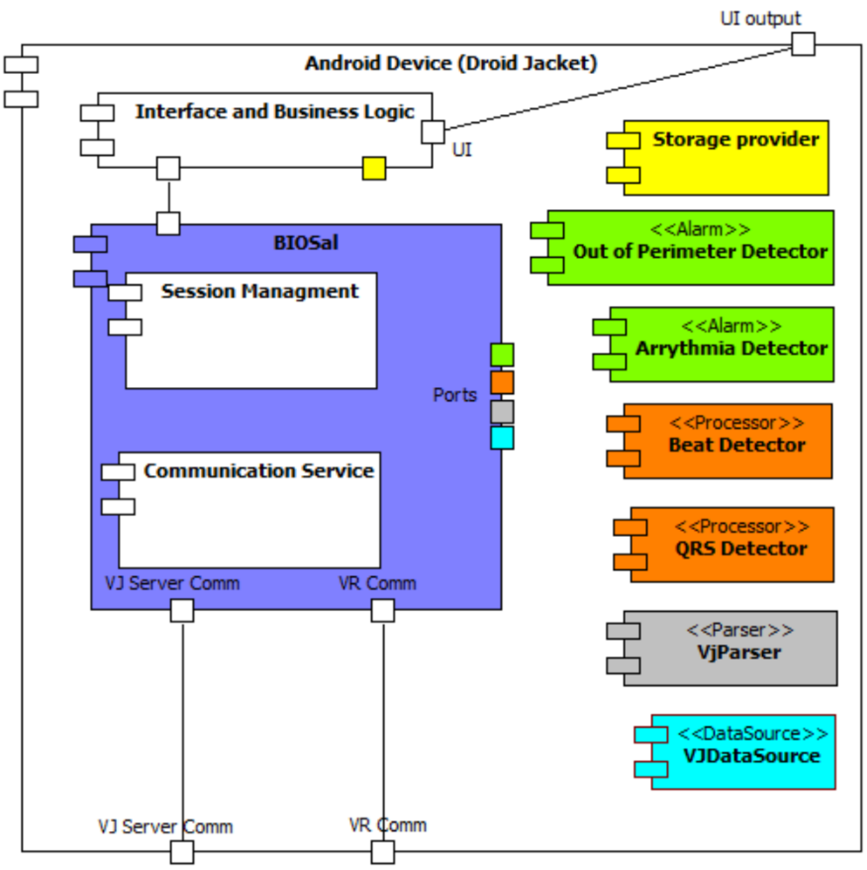


Figure 2.5: Vital Jacket Box architecture [11]

The application was designed as a modular system, managed by the implemented framework named Biological Signal Acquisition Layer (BIOSal) . Different data sources, parsers, processors and alarms can be implemented and integrated into the application (see figure 2.5).

As the data is received, it is parsed, processed, checked for alarming values and, when in server mode, relayed to the connected clients. The processing modules can be implemented to detect patterns in the data, such as heart beats detection. Alarms can be created, for the case of alarming values or alarming patterns' detection, such as arrhythmia.

Regarding the user interface, the application provides access to user management, to a real-time monitoring visualization of the processed data, to a map with the spatial positioning of the other users and gives the ability to the user to start the server mode and change application's configuration.

At the figure 2.6, a real-time visualization of the processed data can be seen. In this case, the information is being presented in a form of a chart. The figure 2.7.b shows a visualization in form of a map, with the spatial positioning of the team members. The screen in the figure 2.7.d shows Android's notification list with an alert from the application, alerting for arrhythmia detection.

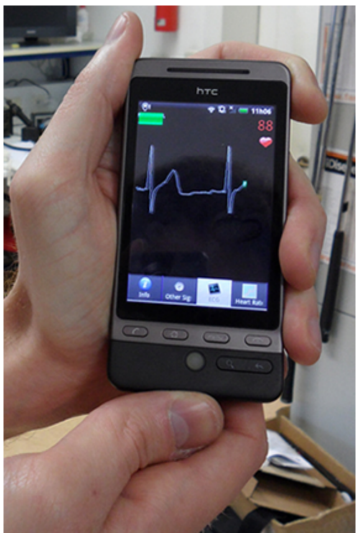
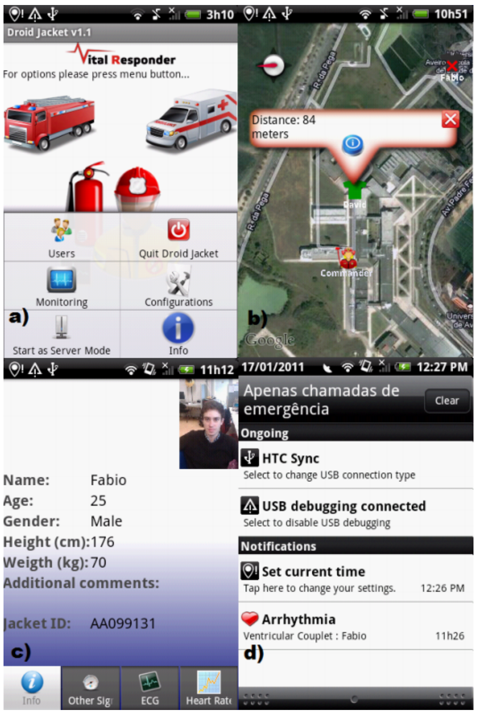


Figure 2.6: Droid Jacket [11]

When the application is set to server mode, it starts listening for external TCP connections. Once connected, the processed data and alarms are relayed to the connected device. In the case of Vital Responder project, the device being connected to receive this data is an iPad with an application called iVital (see section 2.2.1). When connected to iVital, the Droid Jacket application starts running in the background, as any interaction from the first responders with the HHD is not desirable during the performance of their tasks [3].



Name: Fabio
Age: 25
Gender: Male
Height (cm):176
Weigth (kg):70
Additional comments:
Jacket ID: AA099131

Apenas chamadas de emergência
Ongoing
HTC Sync
Select to change USB connection type
USB debugging connected
Select to disable USB debugging
Notifications
Set current time
Tap here to change your settings. 12:26 PM
Arrhythmia
Ventricular Couplet : Fabio 11h26

Figure 2.7: Droid Jacket screens [11]

iVital

As it is not convenient for the first responders to interact with the mobile device during their tasks, a separate mobile team coordination station was implemented. This base station, a proof-of-concept in form of an iPad application dubbed iVital, has the capability to connect via 802.11 to 4 devices (but, technically scalable up to 12) with Droid Jacket running in server mode (see section 2.2.1). It is a mobile solution, used by the team coordinator. It allows to have an overview status of the team in action, as the figure 2.8 illustrates.

On the screen, the intervention area is presented. In this area, the team members are spatially located with color representation, when the GPS positioning is available. On the bottom, a list of team members is shown, with a color representations consistent to their position on the map. By pressing one of the team members, the bottom screen shows his/hers vital signs and alarms. As for the environmental condition, on the top of the screen it is possible to read information, such as humidity, temperature and wind.

The implemented alarm system warns the user with audio or/and visual signs whenever it detects that an individual is in an alarming situation, with the pre-defined alarm implementations (see BIOSal, in section 2.2.1).

Another important feature implemented in the iVital, is the feature "search and rescue". This feature gives the ability to send help to an injured person, by sending a message to the nearest first responder with the location where that person is. The message sent contains the

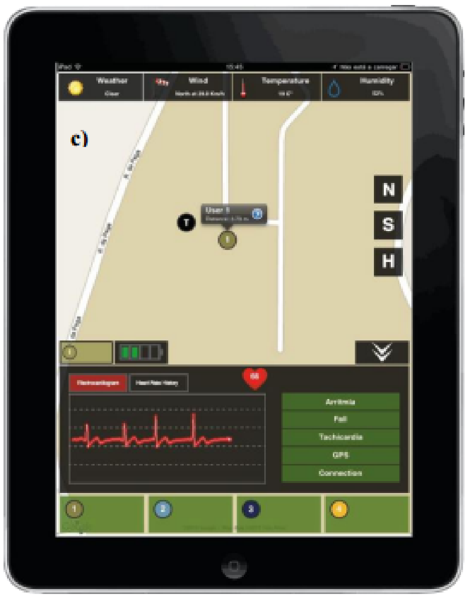


Figure 2.8: iVital [3]

coordinates of the person, which DroidJacket will parse and then guide the first responder to the injured individual [3].

The communication with the Droid Jacket is made with a tag-oriented protocol. The datagrams are composed by a common header and the data. The common header contains the timestamp tag and the timestamp value. The data is composed by the data type tag and the data values [3].

At this device, the data received is aggregated, processed and then presented to the user in form of visualization, such as the ones described above.

2.2.2 Communication

As previously described, the Vital Responder project is composed by a wearable device Vital Jacket together with Vital Jacket Box for sensor data relay, an Android application dubbed Droid Jacket for data processing, alarm detector, visualization and relaying and the iVital, used for aggregated data processing, alarm detection and visualization for team coordination.

The Vital Jacket's sensors communicate with the Vital Jacket box through wires. The latter device relays the data via Bluetooth and can optionally save it into a SD-Card for further analysis.

The Android device, with the Droid Jacket application installed, receives the Bluetooth streamed data from a Vital Jacket Box, as can be seen in the figure 2.9. When set into server mode, the Droid Jacket application listens to TCP/IP connections. When a connection is established, it is registered as a listener for the processed data.

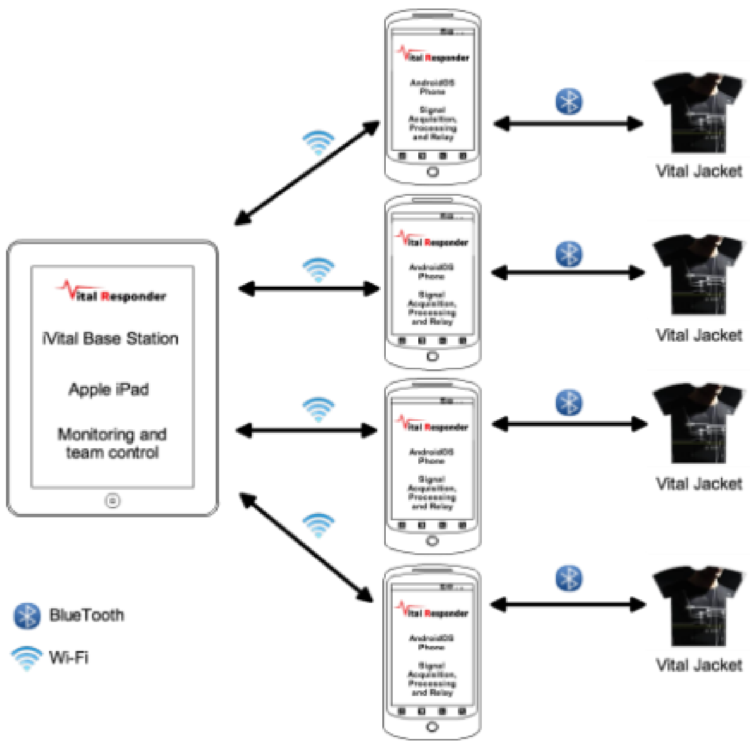


Figure 2.9: Vital Responder architecture [3]

Currently, the iVital application is the one establishing connection to several Droid Jacket's devices (up to 4, but technically scalable to 12). The connection is made through wireless connection with the IEEE 802.11 protocol. Unfortunately, an operational limitation exists, as the connection cannot be established directly to the Android device. This is due to the fact that the operative system do not support ad-hoc connections [3].

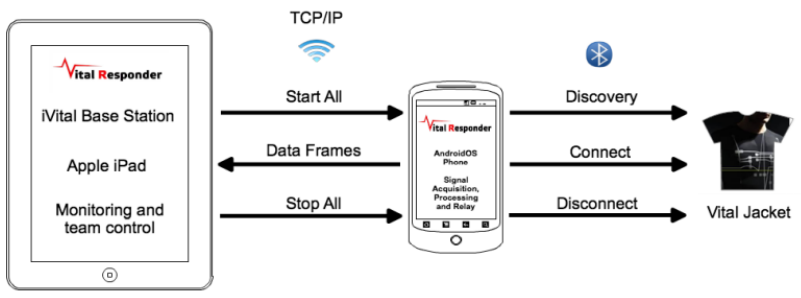


Figure 2.10: Vital Responder communication [3]

The figure 2.10 describes the communication steps between the above described devices.

First, the Droid Jacket application starts the Bluetooth device discovery. Then, when the correct Vital Jacket Box is found and selected, it establishes a connection and keeps it alive until a disconnect request. The physiological data is transmitted from the Vital Jacket Box while connected to the Droid Jacket's HHD. The transmitted data includes 1 lead ECG at 500Hz, accelerometers sampled at 10Hz and GPS location at 1Hz [3]. When set into server mode, the Droid Jacket application listens to TCP/IP connections. This connection is used to relay the processed data to other devices.

In Vital Responder context, the connection to the Droid Jacket is done from the iPad's application iVital. This application establishes a connection to the Android HHD device and sends a command representing a "Start All". Once this command is received on the Droid Jacket, the iVital's connection is registered as a listener to the processed data. It means that, as soon as the data is processed, it is sent to the iVital application.

2.2.3 Scope in this thesis

The aim for this Master's Thesis will be to develop a multi-platform information system. This information system should receive the data from the Droid Jacket and present it live at the client application interface with appropriate visualizations. The idea is to have a Web alternative to the currently developed iVital application.

2.2.4 Requirements

As mentioned in the subsection 2.2.3, one of the objectives is to design and develop a multi-platform information system. It should be possible to access this system in a computer, a tablet or a HHD, taking in consideration the different screen sizes. In all of those cases, the user interface should be intuitive, to be used by professionals with no experience on computer software usage.

The information shown on the screen should be presented as soon as the data is available. A perfect scenario would be to have the information displayed in real-time.

Regarding the connection, it should be established from the information system to the Droid Jacket applications in server mode, via TCP/IP. The communication should be bi-lateral to allow the sending of commands.

2.3 Research on relevant projects

In this section, the research on other related work is documented. It includes a Brazilian project [12], a telemedicine solution for senior citizens and patients with cronic deseases, a Norwegian system for electrocardiogram data transmission [6] and an American project presenting a heterogeneous wireless network to support a home health system [13].

2.3.1 Mobile Telemedicine System

The Brazilian telemedicine project's aim is the constant assistance of senior citizens and patients with cronic deseases. It consists of a home care system that supports medical monitoring standards, as well as telecommunication standards.

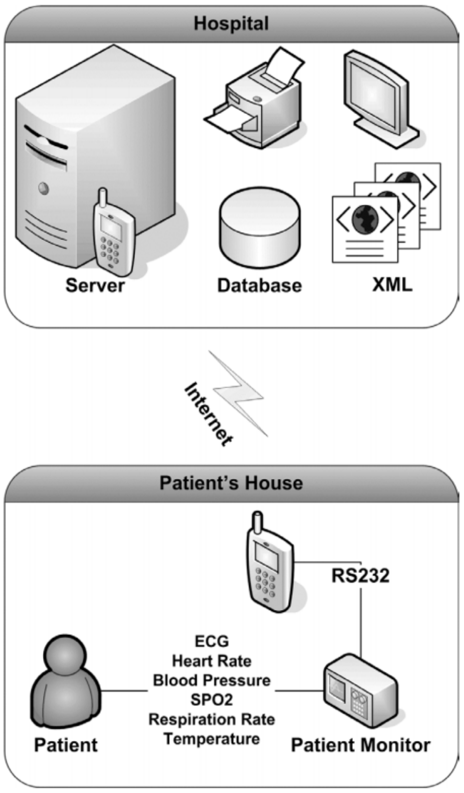


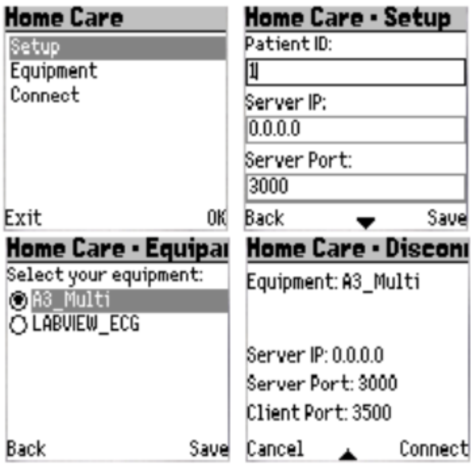
Figure 2.11: Mobile Telemedicine System: System schematic [12]

As the figure 2.11 presents, the implemented system uses mobile telephony for a telemedicine system. The mobile device is connected to a sensor gateway through a RS-232 standard serial interface. This sensor gateway is a radio-frequency receiver that obtains the data wirelessly transmitted by the sensors.

The authors of [12] describe a mobile phone application with a simple to use interface (see figure 2.12). With few commands and basic options, a patient will have decreased difficulties interacting with the application. Its purpose is to upload the monitored data, the clients vital signs, to a server via TCP/IP. This server is, typically, located in a hospital and stores the data in a relational database. For the connection to the monitors, only a generic driver is modeled. The implementation of drivers for many different signals from several available equipements is out of the scope of the project.

For the data to be evaluated by a doctor, a server application is provided. This application aims to receive, store and distribute data. The implemented features allow to display patient lists with patient data, visualization of vital signs, data export in eXtensible Markup Language eXtensible Markup Language (XML) format and printouts as PNG-images. The XML export makes the system compatible to other analyzing software.

Both applications in the client-server architecture, the client application on the mobile phone



Patient ID: 1
Server IP: 0.0.0.0
Server Port: 3000

Select your equipment:
● A3_Multi
○ LABVIEW_ECG

Equipment: A3_Multi
Server IP: 0.0.0.0
Server Port: 3000
Client Port: 3500

Figure 2.12: Mobile Telemedicine System: Mobile application's user interface [12]

and the server application on a hospital's computer, are implemented in Java.

According to the authors of [12] the system archieves a guaranteed transmission of a packet each 600 ms. Depending on the data size a transmission lasts 2 to 30 seconds at a baud rate of 3400 bps. Lost packets are tracked by Cyclic Redundancy Code (CRC) .

2.3.2 Wearable ECG-recording System

In 2005, a complete system for ECG data transmission was in trial in Norway's hospitals. Rare occurences of cardiac arrhythmias of patients in daily life are aimed to be detected.

Critical patients get wireless ECG sensors which they can stick to their chest (see figure 2.13). Each sensor contains a battery that lasts for several days and a RF transmitter. The RF radio receiver and respective microcontroller are connected to a standard Personal Digital Assistant (PDA) , via RS-232. This HHD can be set-up by the doctor who defines alarm criteria. While the sensor continously sends data, the HDD detects arrhytmias in it. Only if the patient is in an alarming or abnormal state, the PDA sends the last minute of recorded ECG data via GPRS to a server. As the data is not constantly transmitted to the server, this is not a remote monitoring solution with live data. However, for each minute, the HHD calculates average values of the monitored sensor data, such as heart rate, and stores it in a status file. This data is sent regulary to the server as an XML file by using the File Transfer Protocol (FTP) protocol. It is mentioned that no technical skills are required to use the system.

The server is located in a Local Area Network (LAN) of a hospital. It runs Microsoft Server 2003, an Structured Query Language (SQL) database and a Microsoft .Net application. The client and doctor can only access the server through a Virtual Private Network (VPN) . The application is accessed by using a browser. The patient can see his own data, read messages and drug prescriptions from the doctor and can send him questions. In such way,

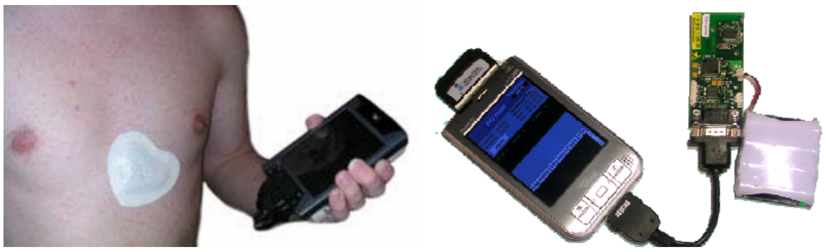


Figure 2.13: Wearable ECG-recording System: Devices [6]

a fast feedback can be provided. On the hospitals' side, a remote clinical application is installed. Alarm records and their time are displayed as well as a vizualization of the ECG data. Furthermore, printouts are possible and a textbox for comments is available.

2.3.3 ITALH and SensorNet

The focus of the Information Technology for Assisted Living at Home (ITALH) project lies on privacy and security of the users. The target group are elder citizens who would have to live in group care facilities if they would not have the system. The main differences to other systems are the Home Health System gateway which replaces the server of other systems and the fact that it is located at the user's home.

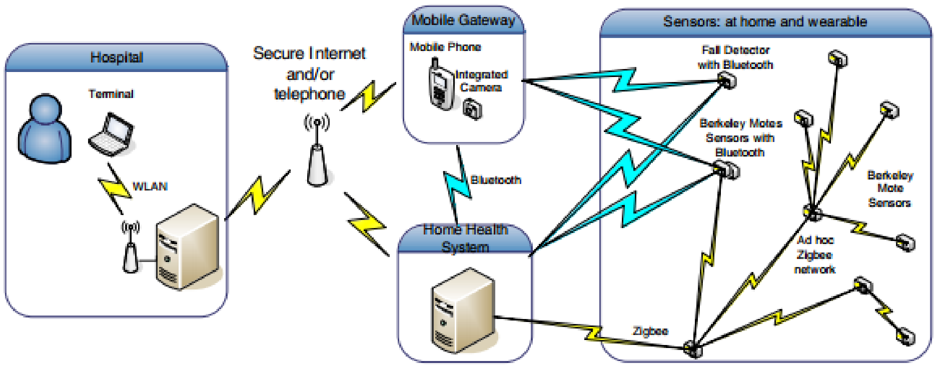


Figure 2.14: SensorNet: Wireless connectivity [13]

In order to provide privacy, the system is encapsulated. The sensors contain embedded processing capability. Only on significant events, the data is transmitted to the central server. The data transmitted contains the sensor status and notification messages. The SensorNet is the main part of the system. It is an ad hoc heterogeneous wireless network, based on Bluetooth and Zigbee (IEEE 802.11.4) technology (see figure 2.14). All kinds of sensors are connected by this network. They measure data about the patients state, his movements and his environment, to detect events, such as if he falls. All the sensors have processors which analyse the data immediatly without sending it to the network. Only in case of emegency, or if an authorized user connects to them, data will be sent to the Home Health System or an additional mobile gateway. This saves local bandwidth and data protection is higher. Furthermore, transmitted data is encrypted.

There are two kinds of gateways possible in the SensorNet. The fixed one, a Windows XP computer and a mobile one, which is a Symbian mobile phone. They are also communicating among each other, if needed. In 2005, when the project started, a video conference with the mobile phone was only possible over the bluetooth connection to the fixed Home Health System gateway.

The task of the Home Health System is to decide if further help is needed and request it in that case. It is securely connected to the global internet and telephone system. If a sensor detects abnormal behaviour, the system will send a notification to the user to confirm his status. If no response is obtained back in a (short) period of time, an emergency signal is created and the access to sensor data is automatically authorized without the user consent. Many different proceedings are possible. Nevertheless, the system will notify a doctor or relative or establishes a phone call if needed. Authorized users can then login and connect to some sensors, for example a camera, to check the users state. A similar usage is imaginable when the user has an online appointment with a doctor.

The great advantage is that the Home Health System is installed at home and no private data is sent, unless necessary. There is no constant or unsecure streaming of data, as it happens in in other systems. With the mobile gateway it is even possible to collect data in the Home Health System when the user is not at his place.

This project from the United States and Denmark provides remote monitoring in case of an accident or acute illness if needed, but also protects the privacy of the clients. The gateway interfaces can be configured variabily, depending on the technical or medical knowledge of the user.

Chapter Three

Design

In this chapter are documented the analysis of conflicting requirements and the documentation of the system design.

3.1 Analysis of conflicting requirements

Taking in consideration the requirements mentioned in the chapter 2, from both AA4R and Vital Responder projects, an information system solution was designed. As the following requirements were conflicting, some decisions had to be taken during the design process. This section is meant to identify the conflicting requirements.

3.1.1 User interface

The iVital application is meant to show in real time the positioning of the team-members on a map. Each team member is represented by a marker uniquely identified by a color. Individual vital signs' visualization can be done without losing the map visualization (see figure 2.8). It would be desirable for the information system to replicate this feature. Unfortunately, in the context of patient monitoring, having a user interface that is showing each patient position in real time can be a privacy issue. Only in some extreme emergency scenarios, a visualization with a map showing the patient's position would become useful, but still, can be seen as a privacy concern.

3.1.2 Communication protocols

The Droid Jacket application uses TCP/IP protocol for the incoming connections, when in server mode. This mode is meant to relay the processed data to another device. Currently this is used by the iVital application. The aim of this Thesis is to create a web alternative to the iVital application, which is able to communicate to the Droid Jacket as it is.

Additionally, as a requirement for the AA4R project, a similar application will be developed. For scientific purposes, the use of UDP/IP protocol must be used instead of TCP/IP. This protocol is lightweight, comparing to TCP/IP, allowing it to be faster at the cost of its reliability [14]. Due to this, a protocol to prevent packet loss should be designed.

So, with this addition in mind, the system which will communicate with the devices will have to be able to communicate with both TCP/IP and UDP/IP protocols.

3.1.3 Adaptability on demand

While on Vital Responder project, the system would be used mainly during the intervention of the first responders' teams, the AA4R project is meant for a long-term usage. It means that, in the former scenario, the system would have peaks of usage during many simultaneous interventions, and fortunate moments when the system is barely used. On the latter scenario, the constant monitoring of the patients is fundamental. Also, the number of patients can grow with the adoption of the system, as well as the quantity of sensors and, possibly, their increasing data frequency rates for a most precise sampling.

3.2 Proposed design

With the analysed requirements in mind, the system can be decoupled into three distinct projects. Those projects are organized into three main functionalities: sensor data relay, sensor data processing and client interface for live visualization of sensor data.

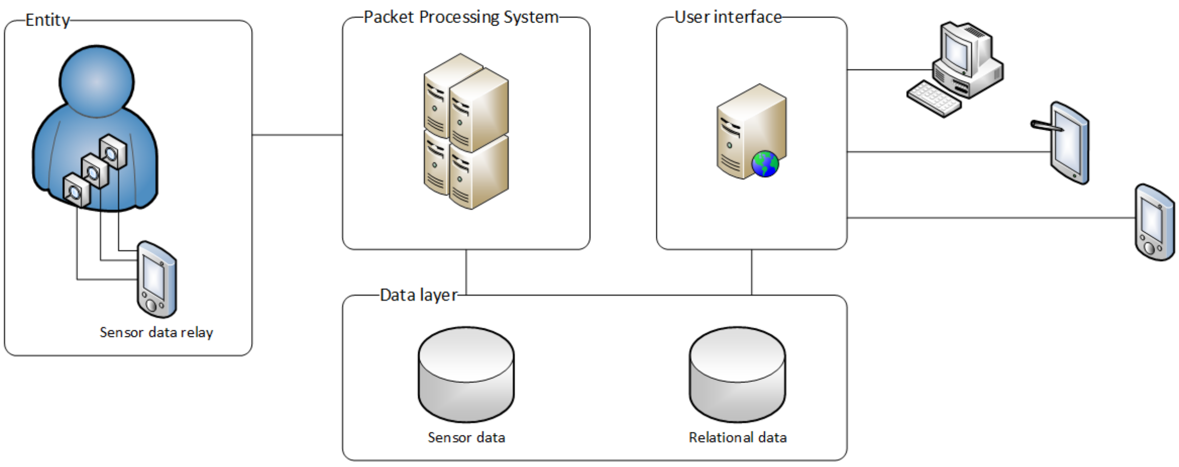


Figure 3.1: Overall system architecture

The figure 3.1 gives an overview of the decoupled projects and the data layer used by them. It also illustrates their relations in the communication.

An entity is defined as a set of sensors connected to a single sensor data relay device and the device itself. It represents a person and/or location, depending on which sensors are connected to the device. From this section on, when the word entity is used, it is being refered to this meaning, unless stated otherwise.

The packet processing system is the set of components responsible for data gathering from the entities, followed by data processing and storage.

The data layer is composed by two databases meant to store sensor data, as well as user and sensor management data. Both are used by the client interface to support the information organization and data visualization, while the packet processing system solely uses the former to store the processed sensor data.

This chapter introduces the design of the components and protocols that compose the final solution:

- Sensor data relay application
- Packet processing system
  - Entity communication point
  - Queue
  - Scallable packet hander
- Client interface

3.2.1 Sensor data relay application

As one of the functionalities of the Droid Jacket application, the aim of the SDR application is to receive data from a set of sensors and relay it to the connected remote systems. The connection to the sensors is made through a gateway, which retrieves the data from the connected sensors (see figure 3.2). The data gatherer (see chapter 3.2.2) connects then to the SDR to obtain the data retrieved from the gateway.

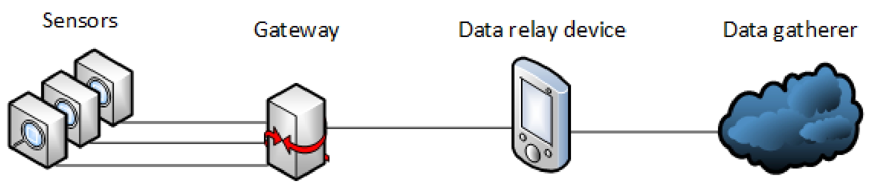


Figure 3.2: Entity components

The selection of the sensor gateway is made by the user, when the SDR application starts, through a list of nearby devices. When the user selects the gateway, the connection to it is established. Once the connection is established, the gateway starts streaming data. At the same time, the SDR application starts listening for connections from data gatherers. At this point, if successfully connected, the user is able to see information relative to Bluetooth and UDP/IP connection, including the local IP address. Using this IP address, the remote system should be able to connect to the SDR application. Appart from Bluetooth and UDP/IP connection information, the user is also able to see statistics, such as, number of packets received and transmitted. The use case diagram of the figure 3.3 describes this interactions with the application.

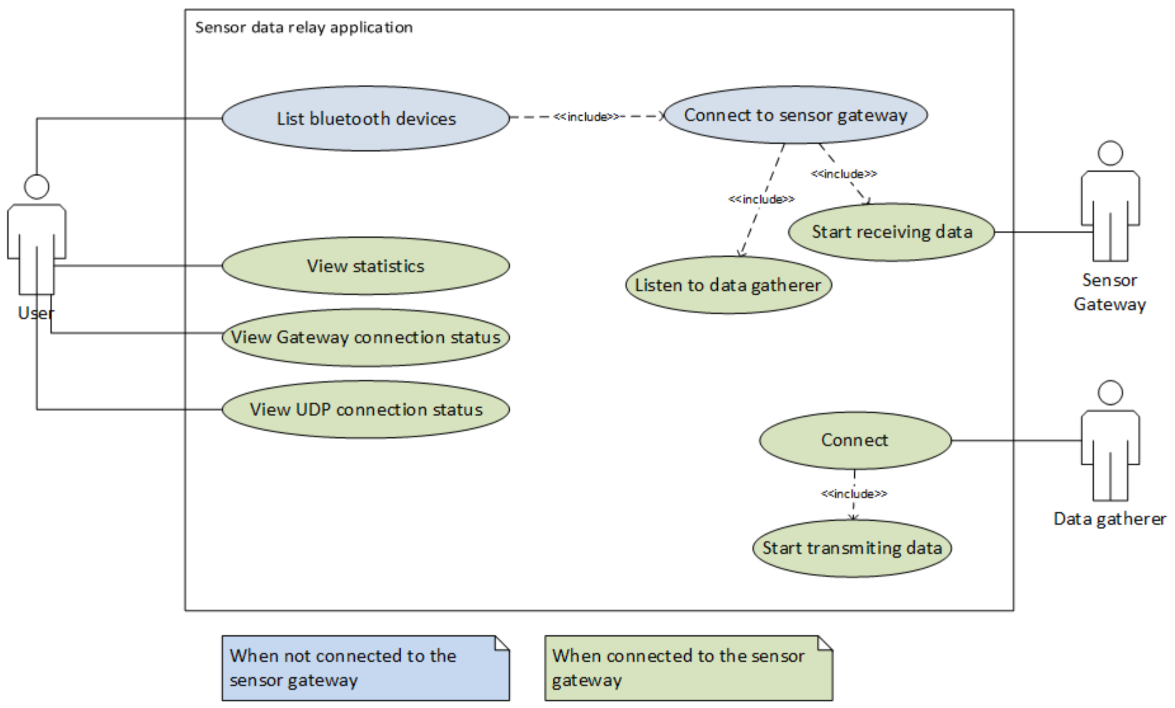

Sensor data relay application
List bluetooth devices
<<include>>
Connect to sensor gateway
<<include>>
<<include>>
Start receiving data
Listen to data gatherer
View statistics
View Gateway connection status
View UDP connection status
Connect
<<include>>
Start transmiting data
User
Sensor Gateway
Data gatherer
When not connected to the sensor gateway
When connected to the sensor gateway

Figure 3.3: Use case diagram: Entity

As the diagram illustrates, only the use cases with blue color should be visible and used when the device is not connected to a gateway. On the other hand, only the green use cases should be visible and used when a connection to the gateway is successfully established.

No other use cases exist, because it is not desirable that the user is required to interact with the application. When the application is connected, it should perform its tasks as a background process with no user interface. The user interface to show Bluetooth and UDP/IP status and packets' statistics should only be shown on user request.

Regarding the connection between the data gatherer and the SDR application, the UDP/IP protocol is to be used. As this protocol does not guarantee the order and the delivery of all the packets [14], a solution should be designed to keep data consistency through the time and a protocol to prevent that the data is not being lost.

To prevent data's consistency loss due to the stream via UDP/IP, the timestamp is sent together with each sensor value. But, this raises a new issue. If the time at the device is not correct, the data will be wrongly shifted in time. An attempt to solve this problem is to synchronize the time on the device with the data gatherer. The data gatherer sends a packet with its current timestamp, the SDR receives it and stores the difference in time between both. Everytime a timestamp is generated on the SDR side, the time difference is taken into account. Unfortunately, this is not enough.

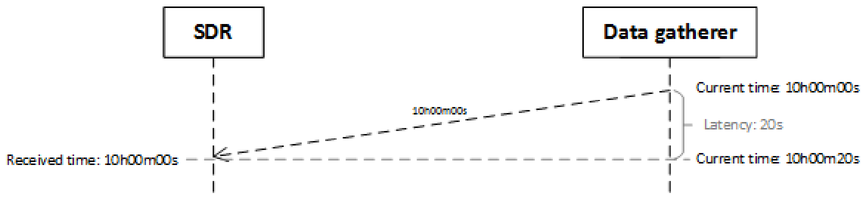


Figure 3.4: Time synchronization - latency issue

As illustrated on the figure 3.4, only using the server time is not enough. The time taken for the packet to reach the SDR application should be considered as well. Analysing the example of the figure, the 20 seconds it takes for the packet with the timestamp to reach the target will result in a wrong time synchronization with 20 seconds of delay. To solve it, it is required to know what is the latency of the packets. As the clock time should not be taken into consideration for the latency calculation (the purpose of calculating the latency is to know the correct time after all), sending the timestamp of one side to another will not add any relevance to the result. But, if the time sent by the data gathered is sent back from the SDR application, as it is, then the sum of the latency of both directions can be measured. In this case, the timestamp can be used as it will be calculated by the same source. So, to calculate the latency of both directions, the difference between the reception time and the packet's timestamp is calculated. An estimate of how much latency exists on one direction can be calculated by dividing this time difference by two.

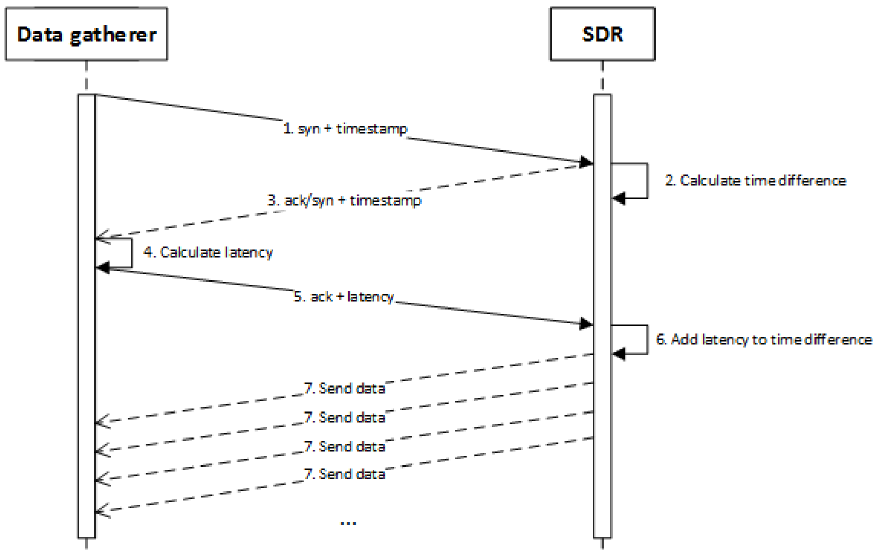


Figure 3.5: Time synchronization and hand-shake

This process can be done during the connection establishment. The protocol illustrated on the figure 3.5 defines a 3-way hand-shake procedure (ideologically based on the one from TCP/IP [15, p. 31]).

At the step 1., a synchronization packet is sent, together with the data gatherer timestamp. At step 2., the SDR application calculates the difference between its time and the timestamp received. At step 3., the packet is sent back as an acknowledge and synchronization packet, together with the original timestamp. At step 4., the data gatherer calculates the difference between its current time and the timestamp received. At step 5., an acknowledge packet is sent, together with the estimated latency (time difference divided by two). At step 6., to the time difference calculated previously, is added the estimated latency. Once the SDR receives the acknowledge, the connection is considered established. At this point, the SDR application starts streaming the data from the sensors.

Unfortunately, this method does not consider time drifting. One possible solution for time drifting, is to do a periodic time synchronization with the Entity Communication Point (ECP) .

Regarding the unreliability of the protocol UDP/IP [14], a protocol should be defined to detect and recover the packets that are lost during the sensor data transmission. As the timestamps are being sent from the SDR application to the data gatherer, they can be used to detect missing packets.

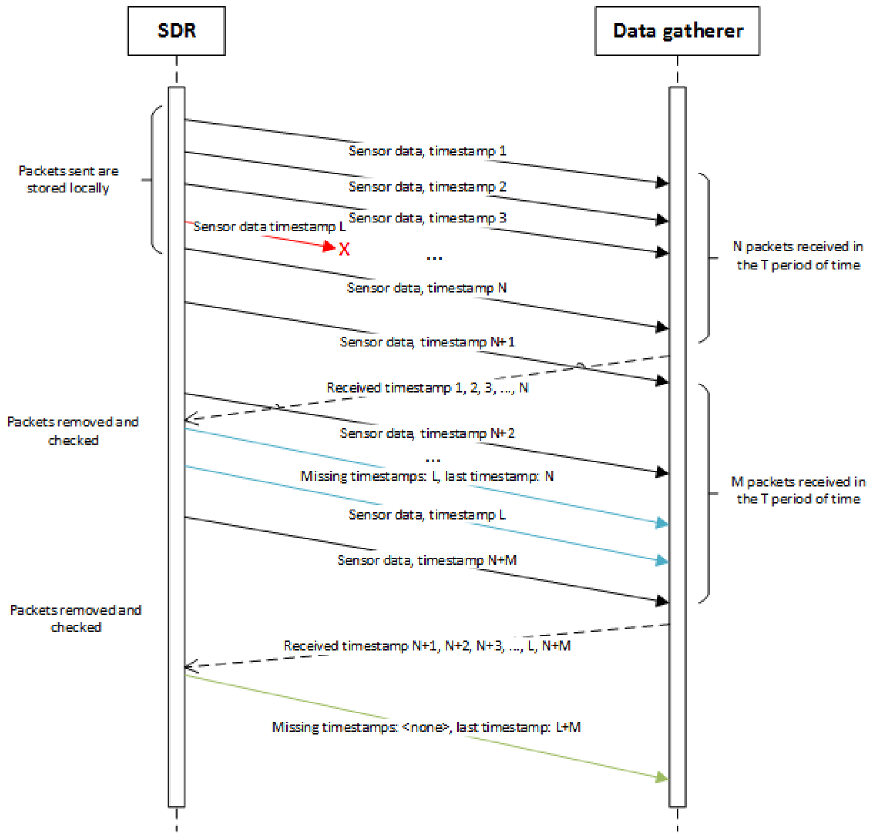

SDR
Data gatherer
Packets sent are stored locally
Sensor data, timestamp 1
Sensor data, timestamp 2
Sensor data, timestamp 3
Sensor data timestamp L
X
...
Sensor data, timestamp N
N packets received in the T period of time
Sensor data, timestamp N+1
Received timestamp 1, 2, 3, ..., N
Packets removed and checked
Sensor data, timestamp N+2
...
Missing timestamps: L, last timestamp: N
Sensor data, timestamp L
Sensor data, timestamp N+M
M packets received in the T period of time
Packets removed and checked
Received timestamp N+1, N+2, N+3, ..., L, N+M
Missing timestamps: <none>, last timestamp: L+M

Figure 3.6: Packet loss detection and recovery

The figure 3.6 shows the designed protocol. The SDR application stores locally each packet sent. With a fixed period of time T, the data gatherer sends back all the timestamps of sensor data received. The SDR then removes all the entries with the corresponding timestamps, and checks for entries with the timestamp lower than the highest timestamp acknowledged. All existent entries are referent to lost packets and are sent again to the data gatherer. While this process is performed, the current sensor data is still being transmitted without interruption.

Between the first N packets in the figure 3.6, one packet is lost (red arrow, packet L). After T period of time, a packet is sent back with the received timestamps. In this packet, the timestamp L is not included, as it was not received. The SDR application removes all the entries with timestamps 1, 2, 3, ..., N. Then, a query is made to the stored packets. Assuming N is the highest timestamp received, all the entries found with timestamp lower than N will be considered as not successfully sent. The first blue arrow in the figure represent the acknowledgement of lost packets and the highest timestamp. The second blue arrow is the lost packet being sent again. In the second T period of time, the packet with timestamp L is acknowledged as received. It is sent together with the other timestamps received in the meantime. The green arrow acknowledges that no packet was lost, together with the highest acknowledged timestamp.

3.2.2 Packet processing system

The packet processing system is responsible for:

- Connection to the entities and sensor data retrieval
- Sensor data processing
- Sensor data storage

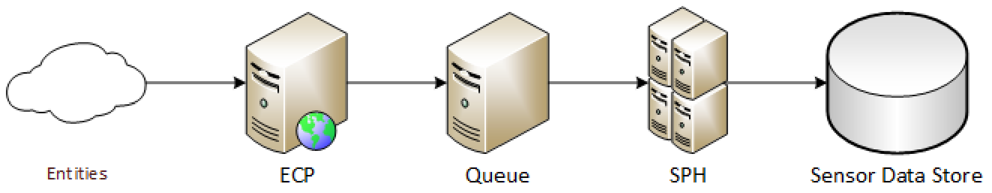


Figure 3.7: Packet processing system

Since the system usage is variable, depending on the case scenario and on the occasion, a scalable system is taken into account during the design. Moreover, considering the importance of a reliable system in patients or first responder's health monitoring, the tasks are decoupled into different steps in a distributed system. By doing so, the different components can be installed in separate physical machines, preventing that the processing of one functionality competes with another in the same processor. Additionaly, by dividing the complexity through the components, the maintenance is easier, the development more clear and the risk of failure can be reduced.

Illustrated in the figure 3.7, the system is decoupled into an ECP, a Queue and a Scalable Packet Handler (SPH) . The processed data, from the SPH, is stored in a data store, such as a non-relational database.

The following sections describe the design of each component of the distributed system:

- Entity Communication Point
- Queue
- Scalable Packet Handler

Entity Communication Point

The ECP is the component which manages connections to the entities. It connects to the remote SDR applications, obtains the sensor data and sends it to a queue for further processing. This component must be able to establish connections, using different protocols, on demand.

This component should not realize unstable operations, such as data interpretation and processing, as it is a single point of failure in the distributed system. If this system fails, no sensor data is received and the other components will not be able to accomplish their tasks without data. Still, some basic operations should be still done, such as adding a timestamp to the packet and sending it to a queue. The operations in a packet should happen in a separate thread to the packet listener. In such way, if a task takes longer to accomplish, the packet gathering will not be compromised. Also, the packet listener usually is a blocking operation. Separating it from the processing will allow the application to run tasks in parallel, while the listener waits for the next packet.

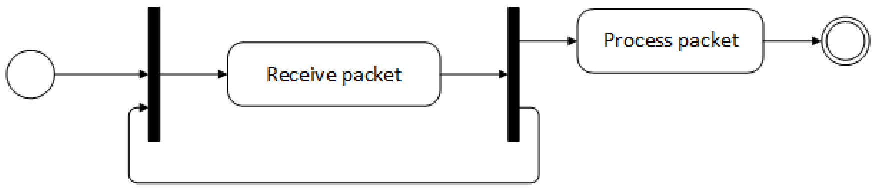


Figure 3.8: Activity diagram: Entity communication Point - Packet reception

The figure 3.8 is an activity diagram with the process of packet listening and processing. As it is represented, each time a new packet is received, a new thread is created to process this packet, while at the same time, the packet listener waits for another packet.

Also important is the fact that each connection's listener should run in a different thread. Otherwise, only one connection could stay actively listening for new packets.

Queue

The Queue is a component that receives messages, store and allow their retrieval on demand. When a message is received it is placed into a queue. In the context of this project, a message will serve the purpose to buffer the received packets for further processing. By buffering the messages, the processing system is not overloaded with packets. Instead, it consumes them as there are resources available [16, p. 11-12]. This queue provides a similar workflow than the Amazon Simple Queue Service [17, p. 6][18, p. 2], but without the advantage of a redundant and distributed queueing system . Still, this architecture contributes to a system more resilient to failure [17, p. 6][16, p. 10].

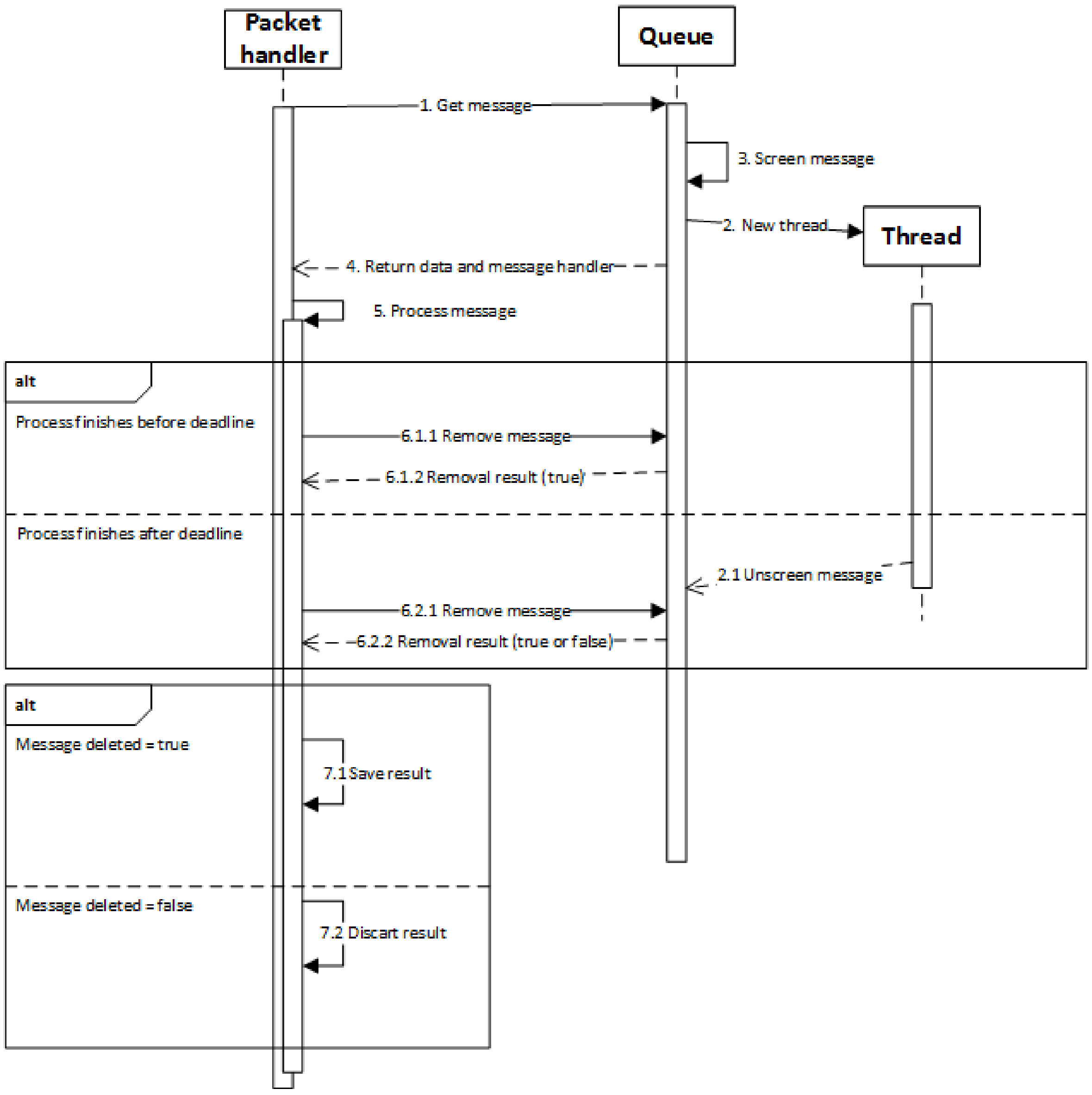


Figure 3.9: Sequence diagram: Queue - Message handling

The figure 3.9 illustrates the sequence of a message request by a packet handler. When a packet handler requests a message, if the queue is empty, the connection is kept active until a new message arrives (1). If the queue contains at least one message, the oldest message is screened from the other packet handlers (2) and retrieved to the requesting one (4). A new thread is created with a timeout of a time T (3). When this timeout reaches the deadline, the message is unscreened from the other packet handlers, unless it has been removed before. A packet is removed with success when no other packet handler finished the same job before (6). If a packet handler takes longer than the deadline, then another packet handler can read and process the same message. If, by any chance, a second packet handler obtains the message after the deadline and is able to finish before the current task, then the second packet handler will successfully remove the message. Then the first packet handler will atempt to remove the same message, a negative result will be given. In this case, the processed data should be discarded (7.2). If a packet is not successfully removed from the queue because it doesn't exist anymore, then it means that another packet handler have already processed the message. For this reason, the storage of the results of the processed packet should only occur after a successfully removed packet from the queue (7.1).

Scalable Packet Handler

A packet handler is a component that retrieves messages from the queue, processes and stores them into a data store. As the values are independently processed, many instances can run parallelly[16, p. 14-15] (see figure 3.10). This makes the system scalable, as new instances and/or machines can be started to consume and process more data, when necessary. On the other hand, it is also possible to reduce the number of instances when the workload is lower. Such scalable system allows the adaptability of resources, depending on the required usage, solving the problem mentioned on the section 3.1.3.

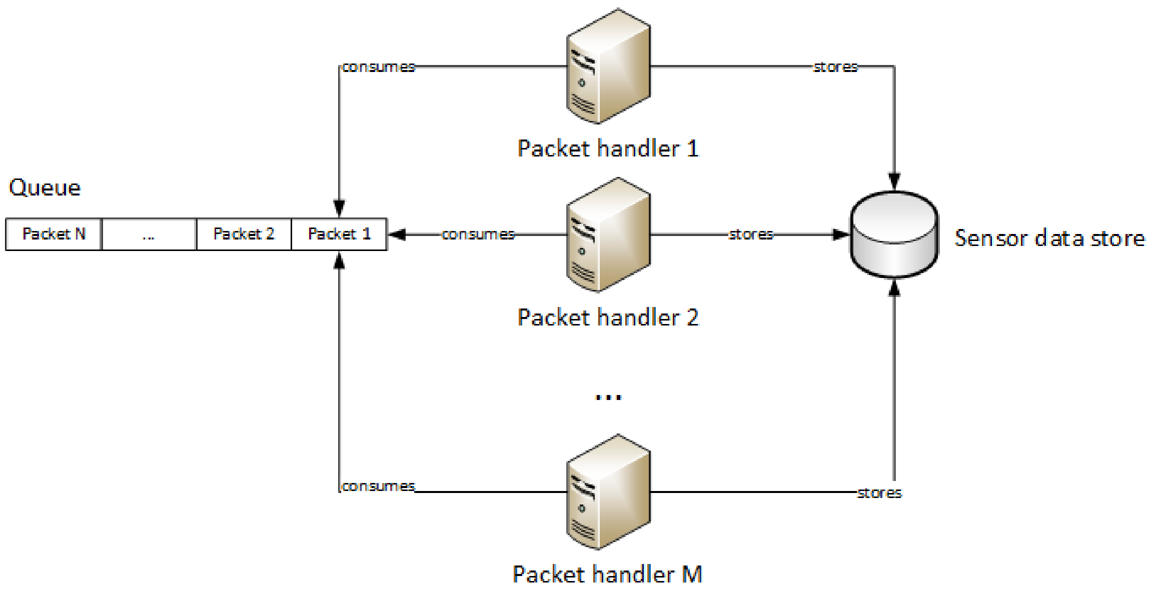


Figure 3.10: Scalable packet hander

If the quantity of messages being stored in the queue have a constant increase, it means that an additional packet handler is needed to avoid delays in the packet processing. If this delay occurs, the client interface will not be able to reproduce a reliable live data visualization of the streamed sensor data. On the other hand, if the queue is constantly empty and the packet handler needs to wait to obtain the next message, then too many packet handlers are running and at least one can be safely turned off, unless only one exists.

3.2.3 Client interface

The client interface serves the purpose of showing historical data, as well as live data being transmitted from the entities. A user is able to authenticate and then list its asscoiated entities. When chosing an entity, the user can see a set of widgets, each one representing one sensor of this entity. The visualization should be generated according to the data type and needs.

It should be able to adapt to the user screen, being it a smartphone, a tablet or a computer desktop.

Chapter Four

Framework

In this chapter, some of the technologies used in this project will be introduced:

- Java platform
  - Differences between JSE, JEE and JME
  - Glassfish, a Java's web application server
  - Brief introduction to Glasfish's HTTP Server, Enterprise Java Beans, Java Server Pages, Java Server Faces
- Relational database MySQL
- NoSQL database MongoDB
- Web frameworks
  - jQuery
  - Twitter's Bootstrap
  - Sammy.js

4.1 Platform

The following subsections will introduce the technologies and tools that will be used during the implementation of the project.

4.1.1 Java

Apart from being a general-purpose, concurrent, class-based and object-oriented language [19, p. 1], Java is not only a programming language. It is also a platform that runs over other platforms, as an abstraction layer. It allows the same Java application to run over different operative systems, such as Microsoft Windows and Linux, without the need of a different compilation[20, p. 6]. The platform is composed by a Virtual Machine (VM) and an Application Programming Interface (API) .

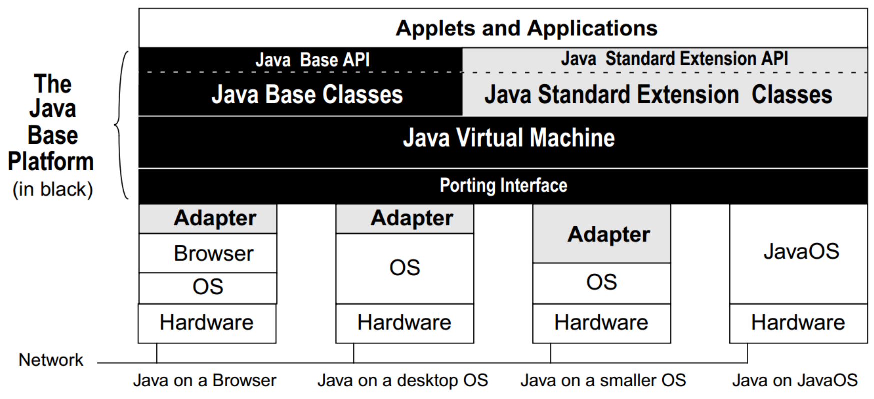


Figure 4.1: The Java Base Platform [20, p. 14]

To program an application, the Java Platform provides the Java Base API, together with a Java Standard Extension API. The former API is the one containing classes that the programmer can take for granted for development, depending on the Java Platform, but independently of the underlying operative system. The sub-sections will introduce some of the available Java Platforms. Regarding the latter, the Java Standard Extension API, contains classes that extend the capabilities of the Java beyond the Java Base API. Those classes may eventually migrate to the Base API, but are still under development or are waiting for review and feedback before being finalized [20].

An application written in Java code is compiled into bytecoded instructions and binary, defined by [19]. This bytecode is not executed directly on a physical machine, but on a Java Virtual Machine (JVM) instead.

The JVM is responsible for interpreting Java Bytecode and translating it, through a Porting Interface, into something that can be understood by the underlying platform (see figure 4.1), at run-time. The underlying platform can be a desktop, server, mobile operative system or even a browser. This means that the JVM is an abstract computing machine [21]. It doesn't recognize the Java programming language, only the bytecode from the compiled *class* files. With that in mind, a different programming language can be used, as far as it is compiled at the end to valid *class* bytecode files [21].

In the figure 4.2 can be seen the processes of compilation and execution. The *java* source files, containing Java programming language code, are compiled into Java bytecode *class* files. As mentioned before, these files contain the bytecode understood by the Java platform, and have no relation to Java programming language. All classes used in the compiled files are loaded from libraries, at run-time. If the refered classes are not found in the libraries, a run-time exception occurs. Otherwise, the JVM interprets the Java bytecode and compiles it according to the underlying platform, where is then executed.

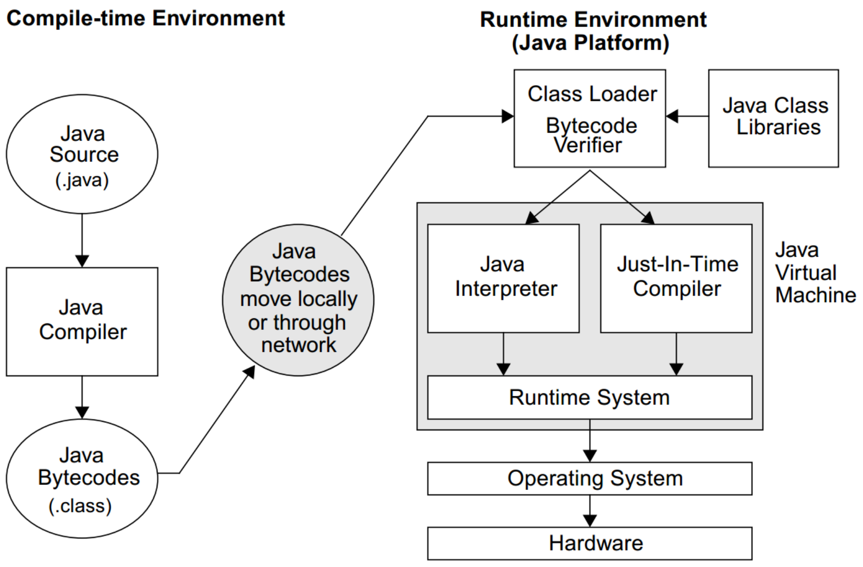


Figure 4.2: Compilation and execution [20, p. 24]

Java Standard Edition

The Java Standard Edition (JSE) is a set of libraries and a platform essential for the core functionality of a Java application. It contains the basic types and objects, in addition to high-level classes meant to deal with networking, security, database access, Graphical user interface (GUI) and XML parsing [22]. It is used to develop and run command-line or GUI applications.

Java Enterprise Edition

Built on top of JSE, Java Enterprise Edition (JEE) is a platform and a set of libraries meant to run and develop large-scale, multi-tiered, scalable, reliable and secure network applications [22]. Those applications run on an application server, such as Glassfish (see section 4.1.2) or Tomcat. As it is built on top of JSE, this API contains all of the JSE features.

Java Micro Edition

Built for small devices, such as mobile phones, Java Micro Edition (JME) is a lightweight platform and a subset of libraries present in JSE[22]. The applications developed for this platform are usually client applications of JEE applications.

4.1.2 Glassfish

Glassfish is an open-source Java application server. Its development is based on the Reference Implementation (RI) of JEE[23, p. 34], with the purpose of running JEE applications. It is built on top of Open Services Gateway initiative (OSGi) , a service platform for Java which gives the capability to install, start, stop and uninstall components in run-time, without the need to restart the system.

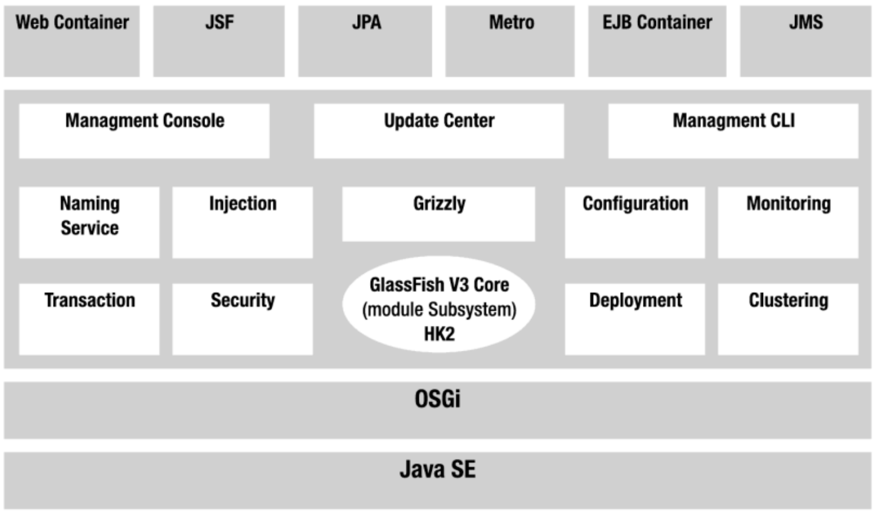


Figure 4.3: Glassfish architecture [23, p. 35]

As the figure 4.3 expresses, this application server contains several decoupled features that can be used by the JEE application or directly by the server administrator. It has a command-line and a Web Interface to configure and monitor the application server. This interfaces also allow the deployement and undeployement of applications and components in run-time.

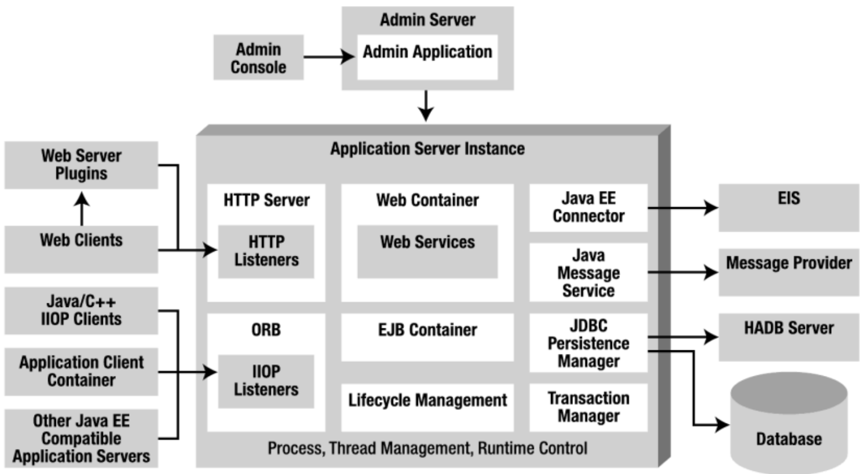


Figure 4.4: Functional parts of Glassfish [23, p. 36]

The figure 4.4 shows the application server splitted into different modules. Not all the modules are going to be used in this thesis. For that reason, a short introduction will be made on the ones considered relevant in this context.

HTTP Server

One of the possible interfaces for a JEE application is through the HTTP protocol. The application server manages a listener on the configured ports (usually 8080 and secure 8181 for development, or 80 and secure 443 for production). When a request is made, Glassfish parses its HTTP header and forwards the request to the right action: be it a HTML page, a resource file or a Web Service. When the request leads to nowhere, a 404 error (page not found) is returned to the client.

Web Container

As the figure 4.5 illustrates, the JEE application server provides a Web Container and an Enterprise Java Beans (EJB) container (see section 4.1.2 for the latter)[24].

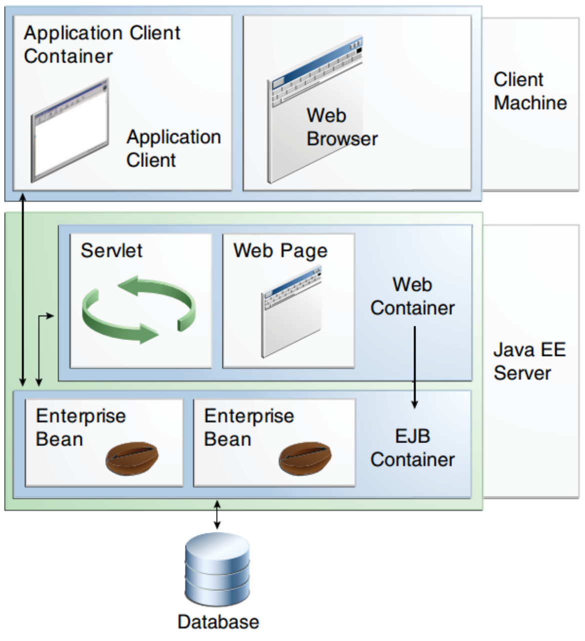


Figure 4.5: Java EE Containers [24, p. 48]

The Web Container manages the execution of web pages, servlets and local EJB interface components. When dynamic content needs to be rendered into the HTML pages, technologies such as Java Server Pages (JSP) , Java Server Faces (JSF) or servlets can be used.

Servlets are modules dynamically loaded on a Web Server. It handles the received requests and, according to the implementation, responses are generated. A HTTP Servlet is an extension to the servlets. It has implemented all the necessary methods to handle Hypertext Transfer Protocol (HTTP) requests headers. As an example, its interface has the methods *doPost* and *doGet* to handle *POST* and *GET* requests, respectively. Those methods are automatically called according to the request, and the developer only has to implement their logic, without the need to handle and filter HTTP headers.

```
<html>
<head><title>First JSP</title></head>
<body>
  <%
    double num = Math.random();
    if (num > 0.95) {
```

```
  %>
      <h2>You will have a luck day!</h2><p>(<%= num %>)</p>
  <%
    } else {
  %>
      <h2>Well, life goes on ... </h2><p>(<%= num %>)</p>
  <%
    }
  %>
  <a href="<%= request.getRequestURI() %>"><h3>Try Again</h3></a>
</body>
</html>
```

Listing 1: JSP Example [25]

JSP complements servlets by generating code inside text-based documents, such as Hypertext Markup Language (HTML) pages or XML documents, in a *jsp* file. This file contains a mixture of static content (such as HTML code) and scriplets. Similarly to PHP, the scriplets are enclosed by delimiters. In JSP case, the delimiters are <% %>. All the Java code that is written inside the scriplets delimiters is rendered on the server side, while the rest of the content of the *jsp* file is kept intact and sent to the client as it is. The listing 1 illustrates a small example of a JSP page.

JSF is a framework for building Web Applications [24, p. 59], that separates the logic from the presentation [24, p. 105]. It is composed by an API for event-handling, validation, data-conversion, internationalization and accessibility, and a set of tag libraries to bind components to their representation in the web page. The components implemented can be reusable and their functionality extended. The binding between the components and their representation is made through managed beans. Managed beans are Plain Old Java Objects (POJO) that support resource injection, lifecycle callbacks and interceptors [24, p. 104]. As managed beans are stateful objects, their lifecycle depends on the scope defined during the implementation. Since JEE 6, the scope of the managed beans can be set by using annotations. There are three possible scopes: Request, Session and Application. With the request scope (defined with @RequestScoped) the managed bean is alive only during the request processing time, being discarted as soon as the response is sent to the client. With the session scope (defined with @SessionScoped), it is loaded during all user session, keeping state between different requests. With the application scope (defined with @ApplicationScoped), it is kept loaded during the total execution of the application, regardless of the active sessions. Its state is persisted among all the users.

Like the JSP, JSF is built on top of Java servlets (see figure 4.6).

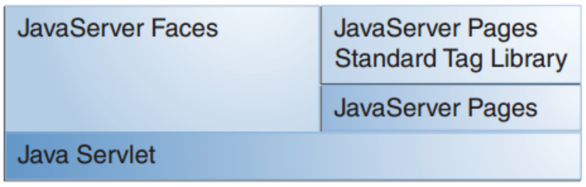


Figure 4.6: Java Server Faces [24, p. 105]

Enterprise Java Beans Container

As the figure 4.5 illustrates, the JEE application server provides a Web Container and an EJB container (see section 4.1.2 for the former)[24].

An EJB is a component that encapsulates the internal logic of an application [24, p. 435], denominated as *BusinessLogic*. The task of the EJB container is to control system-level services, such as object lifecycle, transaction management and security[24, p. 436].

As the logic is decoupled from the presentation, the client applications are lightweight and the client's development is mainly focused on the application presentation. Also, as EJB are portable components, the same implementation can be used by different applications.

There are three ways to access an EJB, as the figure 4.5 describes. If the client application is running on the client's machine, the EJB can be accessed through remote interfaces. The remote interface contain part of or all methods of the EJB implementation. If the client application runs on the server side (such as a Web Application), the EJB can be accessed either by the local interface or directly in the implementation methods. The difference between those two is that, on the latter case, the EJB implementation should part of the same Web Application project. The former case is when the same EJB is shared between applications in the same Web Application Server.

Such as managed beans, EJB Session Beans also have lifecycles declared with annotations. There are three different type of Session Beans: Stateful Session Bean, Stateless Session Bean and Singleton. The Stateful Session Bean (declared with @Stateful) is loaded and persists its state during a user session. Its state is not shared among clients, and it is discarted as soon as the session is terminated. The Stateless Session Bean (declared with @Stateless) should only keep the client's state during its invocation. When the invocation is finished, all the related data should be discarted. The reason for this is that the Stateless Session Beans are pooled in EJB container, and they are used according to their usage. Also, they are shared among users. During two consecutive invocations, by the same user, to the same Stateless Session Bean, it is not guaranteed that the same instance will be invoqued. Regarding the Singleton EJB (declared with @Singleton), it works the same way as the latter Session Bean, with the difference that the EJB Container assures that only one instance of this bean exists during all application lifecycle. It has transactional features and manages concurrent access by clients [24, p. 438].

4.2 Storage

The following subsections will introduce a relational and a non-relational database.

4.2.1 MySQL

MySQL is a free Open Source Relational Database Management System (RDMS) . It works as a server and can be managed either via command line or via client applications. As a relational database, the data is organized into different tables. In each table, the information is stored as entries (rows in a table). Each entry can be related to many entries in the same table or other different tables. The relationship between entries is made through a foreign key. A foreign key is an unique identifier of an entry in a table.

Queries to the MySQL database are done with SQL. SQL is a standardized language, used to query relational databases.

4.2.2 MongoDB

Meant for schema free, JSON-format document based storage, MongoDB is an Open Source non-relational database. Its syntax is similar to the Javascript Object Notation (JSON) format and it fully supports indexing on attributes.

4.3 User Interface

The following subsections will introduce libraries to support the development of the presentation layer of Web Applications.

4.3.1 jQuery

jQuery is a javascript library that supports the work of the developer to build web applications with Javascript. It works on the client side, in a web browser.

It is lightweight and wraps common tasks in functions, which allows the code to remain cleaner and more readable to the developer. It also simplifies the Assynchronous Javascript And XML (AJAX) request calls and message handling. Regarding the elements in the Document Object Model (DOM) , the creation, filtering and manipulation of elements is made trivial with this framework. Furthermore, it is extensible as it has plugins' support.

4.3.2 Twitter's Bootstrap

This Twitter framework solves many issues related with cross-browser visualization and provides functionality to build a clean, responsive and interactive web interface. When the responsive design is being used, its elements change form according to the screen size, allowing the interface to adapt accordingly to stay usable.

4.3.3 Sammy.js

The Sammy.js helps to keep track of page navigation flow in Single Page Interface (SPI) . It interprets the URL and redirects the calls to the appropriate functions to show content in the pages, without the need to refresh them.

Chapter Five

Implementation

In this chapter, the implementation of the designed solution will be described. It contains the following items:

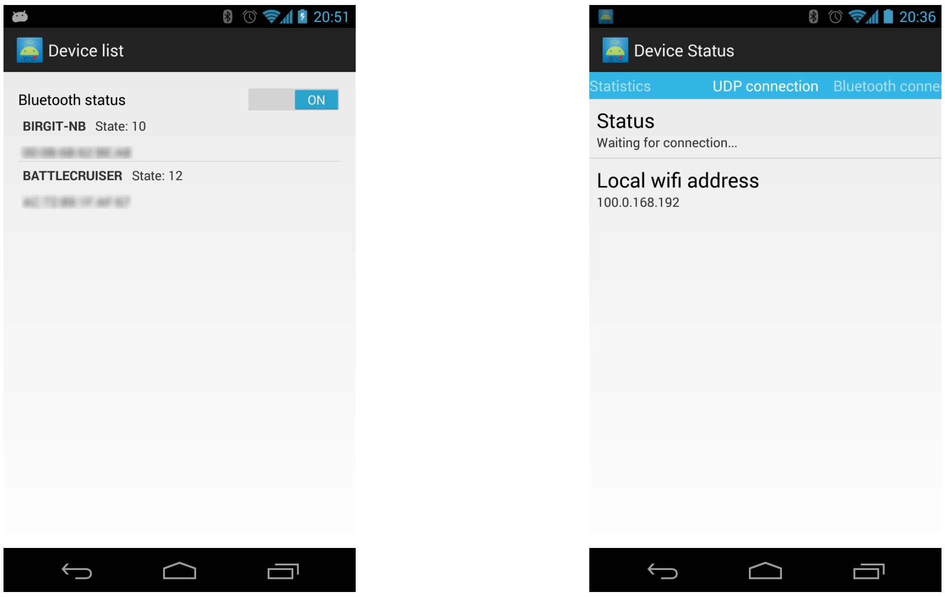
- Sensor Data Relay application
- Packet Processing System
- Client interface

5.1 Sensor Data Relay application

The SDR application is implemented as an Android application. It is composed by two *Activities* and a *Service*. The two *Activites* are referent to two user interface screens.

One *Activity* shows the list of Bluetooth devices nearby (see figure 5.1a). This is the activity that is presented to the user when the connection to the sensor gateway is not established. It allows turning on and off bluetooth. When turned on, the application makes a discovery for devices that are nearby and discoverable. Once the user selects the device, the connection is established and the user is asked to uniquely identify the entity. Then, the second *Activity* is presented and a *Service* is started in the background.

The second *Activity* contains a three-tab layout that shows statistics, UDP connection information and Bluetooth connection information, in each respective tab (see figure 5.1b). This screen is presented only when the application has a connection established to the sensor gateway. It gives connection information, such as the the local IP address. This information is important to establish a connection from the Packet Processing System to the SDR application.


20:51

Device list

Bluetooth status ON

BIRGIT-NB State: 10

[redacted]

BATTLECRUISER State: 12

[redacted]

(a) Devices nearby

20:36

Device Status

Statistics UDP connection Bluetooth conne

Status

Waiting for connection...

Local wifi address

100.0.168.192

(b) UDP connection status

Figure 5.1: Sensor Data Relay: User Interface

The *Service* running in the background has the purpose to do all the logic and connection management. It means that even if the application is closed, it keeps running in the background. When the *Service* is created, it starts listening to UDP packets, in a parallel thread.

There are three ways to communicate with a *Service* from an *Activity* [26]:

- Extending the Binder class - If the service is private to the own application;
- Using a Messenger - If the communication is made across different processes. Messages are buffered in a queue and processed one by one;
- Using Android Interface Definition Language (AIDL) - Used also for interprocess-communication. The difference to the Messenger is that multiple messages are processed at the same time.

As the *Service* in this application is meant to be used solely by the application itself, the first approach is used. To do so, the Binder interface is implemented, containing a reference to the instance of the *Service*. This *Binder* instance is returned by the service's *onBind* method. To have access to the *Binder* instance, the activity should call the *onServiceConnected* callback method. This method returns a *ServiceConnection* instance with the reference to the service's *Binder* instance. See figure 5.2 to understand how they are connected.

The protocol for connection establishment and clock synchronization is implemented in this

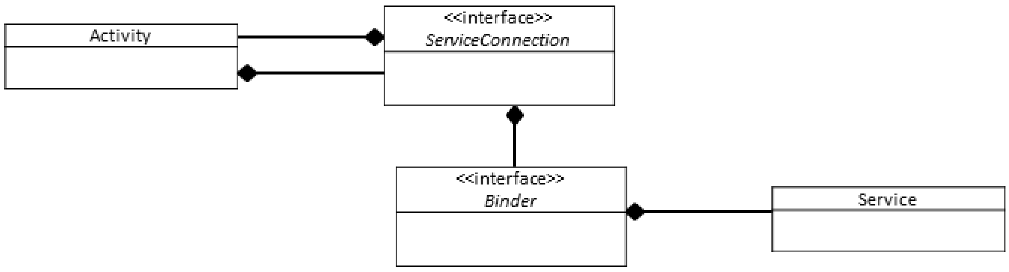


Figure 5.2: Class diagram: Android communication between Activity and Service

service. The figures 5.3a, 5.3a and 5.3a illustrate the packet structure of the three-way handshake protocol from the figure 3.5.

MSG Type 1 byte (0x01)	Timestamp 4 bytes

(a) SYN packet

MSG Type 1 byte (0x02)	Timestamp 4 bytes

(b) SYN/ACK packet

MSG Type 1 byte (0x03)	Latency 4 bytes

(c) ACK packet

Figure 5.3: Sensor Data Relay: Hand-shacke packet structure

When the three-way handshake successfully happens, the connection is considered established. Once it happens, the data that is received from the sensor gateway starts to be streamed. To prevent a too high number of packets being sent, a buffer is developed. This buffer keeps the received sensor data until the next packet is built and sent. To build a packet, all the data is loaded from the buffer, the buffer is cleared and the loaded data is then merged. The packet sending is defined to happen each 500 ms. In such way, two packets are sent per second, independently of the number of sensor values in the buffer.

Sensor ID 2 bytes	Sensor Type 1 byte	Value 2 bytes	Timestamp 4 bytes

(a) Data chunk

MSG Type 1 byte (0x11)	Entity ID 1 byte	Timestamp 4 bytes	Data 1 9 bytes	Data 2 9 bytes	...	Data N 9 bytes

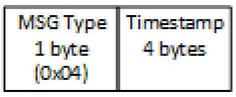
(b) Entity data packet, with N sensor values (data chunks)

Figure 5.4: Sensor Data Relay: Data packet structure

The figure 5.4a represents a chunk of data. It contains the respective sensor identifier, sensor type, a timestamp of the reading and the value. The figure 5.4b represents the complete data packet sent from the entity to the ECP with N sensor values.

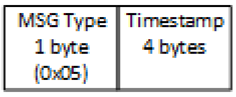
In all the packets transmitted between the entity and the ECP, there is a message type. The message type identifies the type of message being sent.

Additionally to the protocols previously designed and documented, a protocol to constantly estimate the latency between the SDR and the ECP is developed. The timestamp is sent to the SDR (figure 5.5a), which in return sends the timestamp unchanged (figure 5.5b). The latency is estimated by the time it took for the packet to travel in both directions, divided by two.



MSG Type 1 byte (0x04)	Timestamp 4 bytes

(a) Ping request packet



MSG Type 1 byte (0x05)	Timestamp 4 bytes

(b) Ping response packet

Figure 5.5: Sensor Data Relay: Ping packet structure

At the ECP, the average latency is constantly calculated. Thus, the cumulative average formula is used to avoid the need to store all the latency values received (see figure 5.6).

$CA_{i+1} = \frac{iCA_i + x_{i+1}}{i+1}$, where CA is the cumulative average, x_{i+1} the current latency and i the number of latency values received.

Figure 5.6: Cumulative average formula

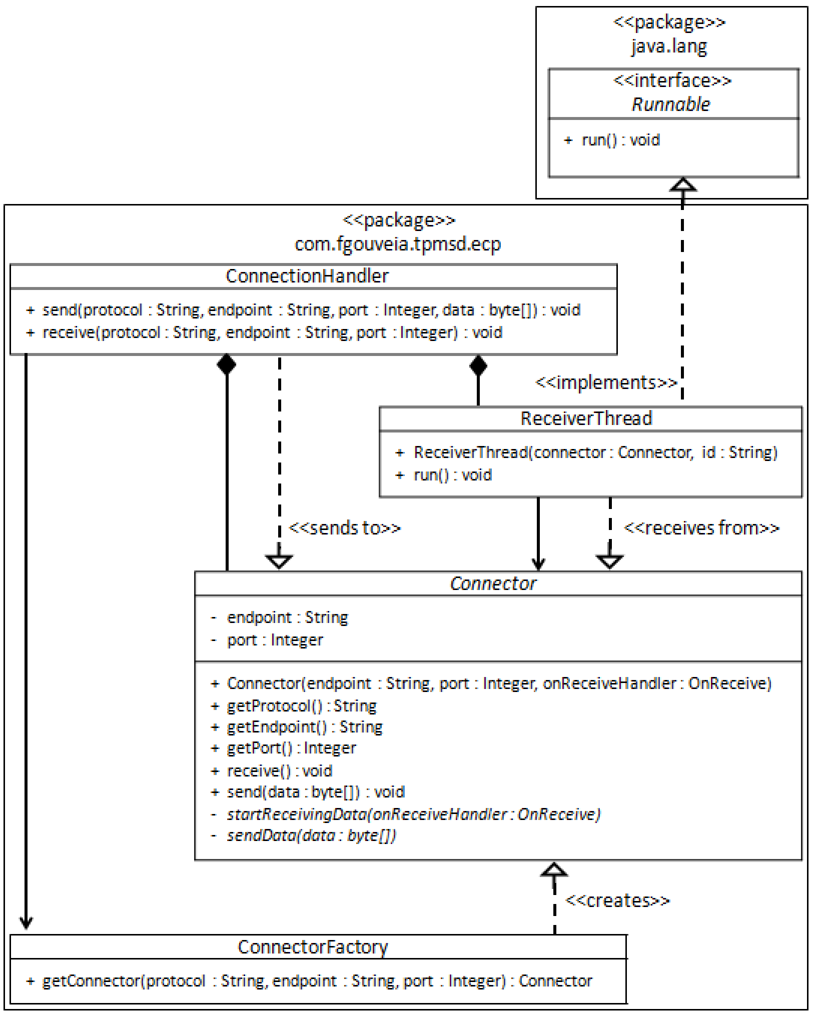
This formula is also used in order to calculate other cumulative averages, such as the average number of sensor values received per packet.

5.2 Packet Processing System

As designed, the packet receiving process is decoupled from the packet processing and packet storage. This decouple is made by the separation of each process into different components (see figure 3.7).

5.2.1 Entity Communication Point

The component responsible for handling the connections to the entities is the ECP. It establishes the connections to the devices and receives the data from them. Due to the fact



<<package>>
java.lang
<<interface>>
Runnable
+ run() : void
<<package>>
com.fgouveia.tpmsd.ecp
ConnectionHandler
+ send(protocol : String, endpoint : String, port : Integer, data : byte[]) : void
+ receive(protocol : String, endpoint : String, port : Integer) : void
<<implements>>
ReceiverThread
+ ReceiverThread(connector : Connector, id : String)
+ run() : void
<<sends to>>
<<receives from>>
Connector
- endpoint : String
- port : Integer
+ Connector(endpoint : String, port : Integer, onReceiveHandler : OnReceive)
+ getProtocol() : String
+ getEndpoint() : String
+ getPort() : Integer
+ receive() : void
+ send(data : byte[]) : void
- *startReceivingData(onReceiveHandler : OnReceive)*
- *sendData(data : byte[])*
<<creates>>
ConnectorFactory
+ getConnector(protocol : String, endpoint : String, port : Integer) : Connector

Figure 5.7: Class diagram - Entity Communication Point - Connection handler

that the system should be able to receive data from different protocols, a modular approach was taken into consideration (see figure 5.7).

To allow the creation of customized connectors, a *Connector* interface is implemented. It is an abstraction of a connection to a remote device. It contains the methods to obtain the connection's information and also methods to communicate with the device. Independently of the implementation of this interface, calling the same method in two different implementations should result in a similar behaviour, in what the communication to devices is concerned.

The ECP has a connection handler (*ConnectionHandler*) with two methods (*send* and *receive*). It uses a connection factory (*ConnectionFactory*) to instantiate the connectors (*Connector*), depending on the protocol to be used. Once the connector is created and connection established, it is able to send data to the remote devices. Also, when the receive method is called, a new thread is created to wait for incoming data (*ReceiverThread*). If no thread would be created at this point, either the programmer would be responsible for the non-blocking listener implementation or the system would stop each time a listener waits for the next packet.

As the packet listening runs assynchronously in a different thread, the system does not wait for its data to arrive. Instead, a data handler (*OnReceive*) is passed as an argument in the *listen* method. Each time a packet arrives, the implementation of the *Connector* should use the *OnReceive* handler to call the *processData* method, with the data as argument. This method will call the configured packet processor's modules (implementation of *DataProcessor*).

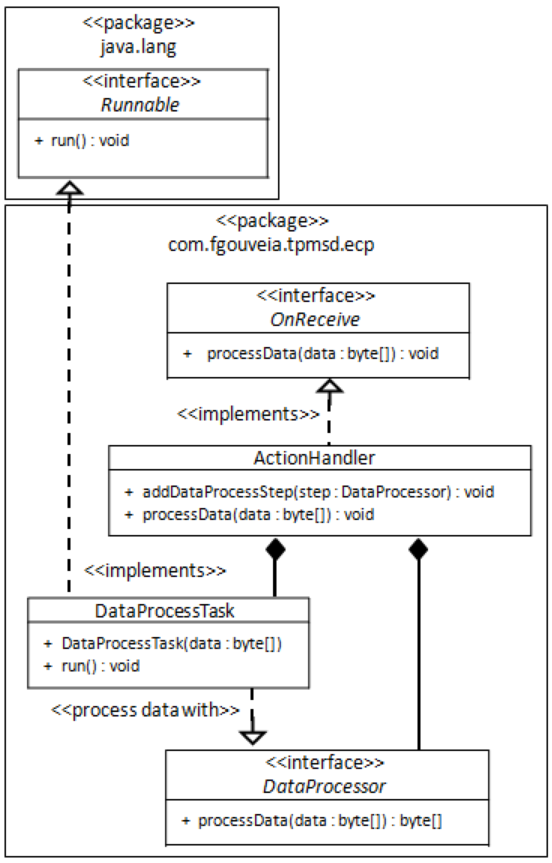


Figure 5.8: Class diagram - Entity Communication Point - OnReceive module

The *DataProcessor* is an interface with a method to process the data (*processData*). The implementation of this interface receives the data as an input argument and should return the processed data as its output. The order of the configured *DataProcessor*s is important as the output of one module will be the input of the next module, as a chain of data processors. An example of an implementation is the one that adds the ECP timestamp to the data received (see listing 2).

```
public class AddTimestamp implements DataProcessor{

    @Override
    public byte[] processData(byte[] data) {
        Long timestamp = System.currentTimeMillis();

        ByteBuffer newData = ByteBuffer.allocate(data.length +
            Constants.TIMESTAMP_SIZE);
        // Add timestamp to the beggining
        newData.putLong(timestamp);
        // Add data to the end
        newData.put(data);

        return newData.array(); // Return the timestamp together with the data
            in the same binary array
    }
}
```

Listing 2: Data processor implementation: Add timestamp

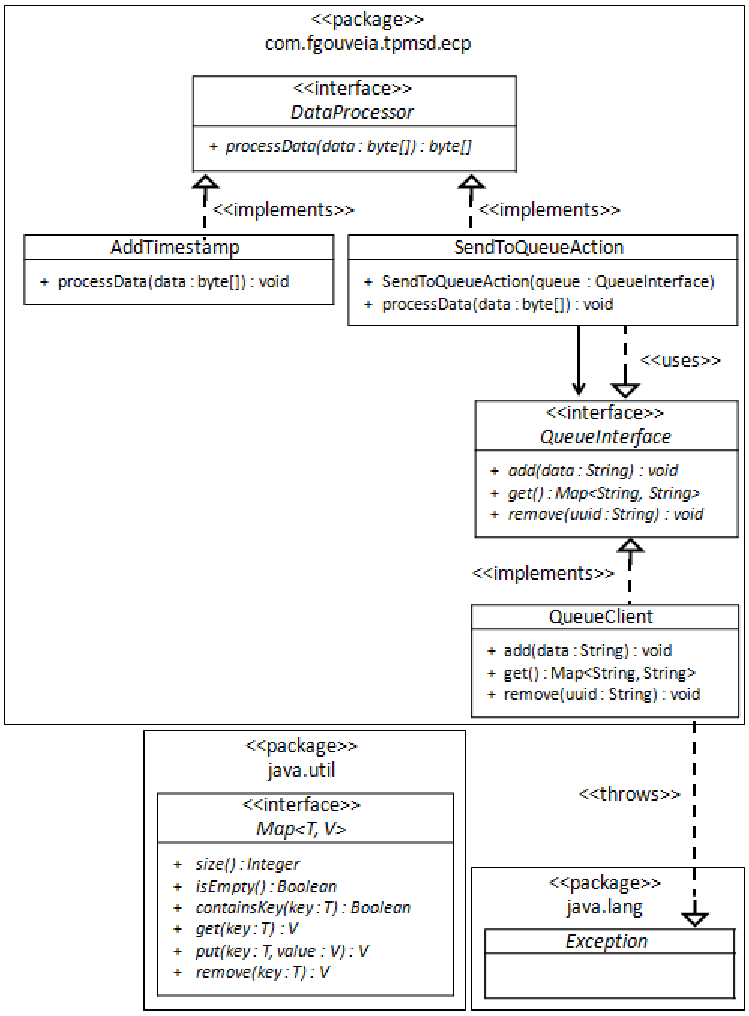
It is not mandatory to modify the data received. One implementation where it happens is the one that sends the received data to a queue. In this case, after the data is sent, the method returns the original data, received initially as an input argument (see listing 3).

```
public class SendToQueueAction implements DataProcessor{
    private final QueueInterface queue;
    // ... Constructor with the QueueInterface initialization

    @Override
    public byte[] processData(byte[] data) {
        // Send to the queue
        queue.add(Base64CoDec.Base64Encode(data));

        // No changes were made to the data, returning original
        return data;
    }
}
```

Listing 3: Data processor implementation: Send to queue

The class diagram of the figure 5.9 represents how the extensions to the *DataProcessor* are implemented.

The implementation of the *OnReceive* interface is the *ActionHandler*. It creates a new thread (*DataProcessTask*) to execute the chain of packet processors. It is used each time



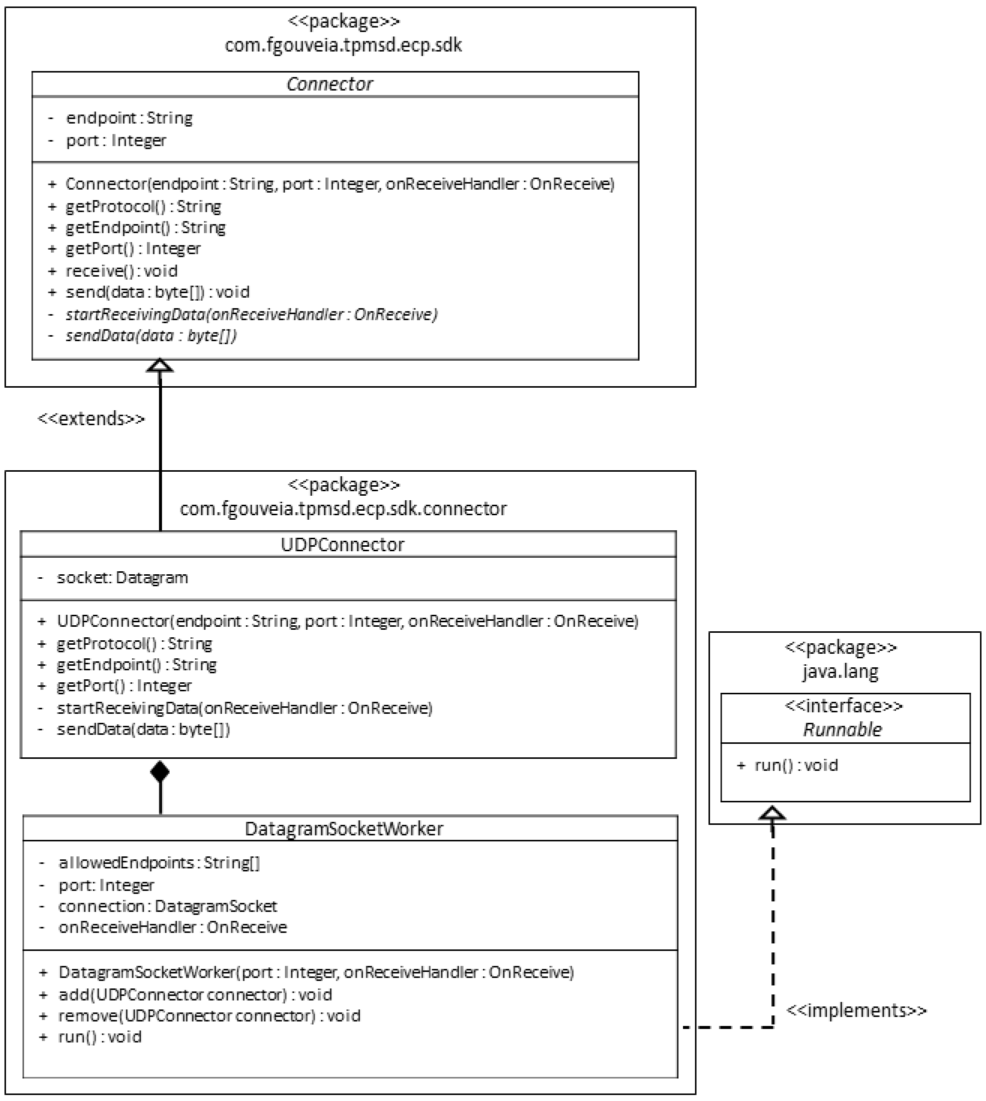
<<package>>
com.fgouveia.tpmsd.ecp
<<interface>>
DataProcessor
+ processData(data : byte[]) : byte[]
<<implements>>
<<implements>>
AddTimestamp
+ processData(data : byte[]) : void
SendToQueueAction
+ SendToQueueAction(queue : QueueInterface)
+ processData(data : byte[]) : void
<<uses>>
<<interface>>
QueueInterface
+ add(data : String) : void
+ get() : Map<String, String>
+ remove(uuid : String) : void
<<implements>>
QueueClient
+ add(data : String) : void
+ get() : Map<String, String>
+ remove(uuid : String) : void
<<package>>
java.util
<<interface>>
Map<T, V>
+ size() : Integer
+ isEmpty() : Boolean
+ containsKey(key : T) : Boolean
+ get(key : T) : V
+ put(key : T, value : V) : V
+ remove(key : T) : V
<<throws>>
<<package>>
java.lang
Exception

Figure 5.9: Class diagram - Entity Communication Point - DataProcessor module implementations

the method *processData* is called. If, otherwise, the processing would be running in the same thread, the connector would be waiting for the processing chain to finish before listening to the next packet.

Furthermore, by using a thread to process each packet, more than one packet processor runs in parallel. Additionally, if a packet processing fails for some reason, the processing chain might be interrupted, but the system continues working. It is important to guarantee that, as a single point of failure, the ECP has a reduced risk of failing due to processing problems.

To complete the topic, the *Connector* implementations to establish the connections are presented. Currently, the UDP connector is implemented with the basic functionality of sending packets and receiving packets. This module is implemented in a way that only packets from requested addresses are accepted. If a packet arrives with an invalid source address, the packet is discarted. As the system can be connected to different devices, and the receiving port is the same for all of them, an object to manage the UDP connections' addresses is implemented (see figure 5.10, *DatagramSocketWorker*). It is important to state that the same UDP implementation is being reused for different connections.

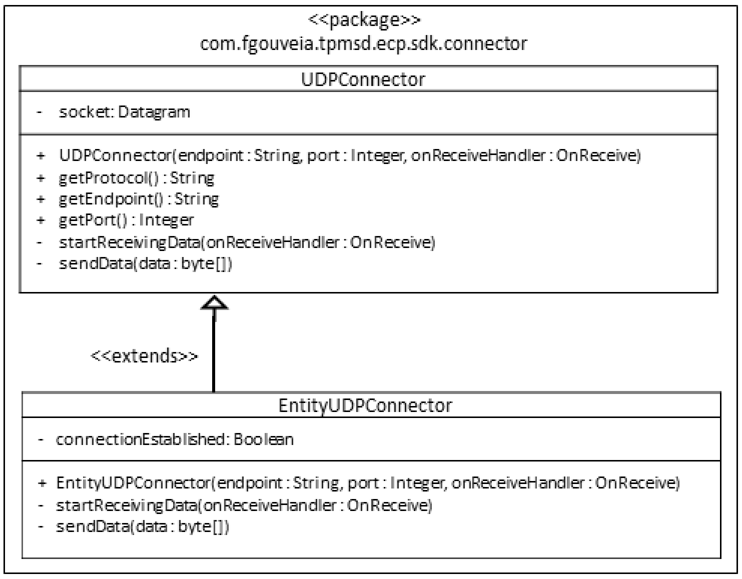


<<package>>
com.fgouveia.tpmsd.ecp.sdk
Connector
- endpoint : String
- port : Integer
+ Connector(endpoint : String, port : Integer, onReceiveHandler : OnReceive)
+ getProtocol() : String
+ getEndpoint() : String
+ getPort() : Integer
+ receive() : void
+ send(data : byte[]) : void
- startReceivingData(onReceiveHandler : OnReceive)
- sendData(data : byte[])
<<extends>>
<<package>>
com.fgouveia.tpmsd.ecp.sdk.connector
UDPConnector
- socket: Datagram
+ UDPConnector(endpoint : String, port : Integer, onReceiveHandler : OnReceive)
+ getProtocol() : String
+ getEndpoint() : String
+ getPort() : Integer
- startReceivingData(onReceiveHandler : OnReceive)
- sendData(data : byte[])
DatagramSocketWorker
- allowedEndpoints : String[]
- port: Integer
- connection : DatagramSocket
- onReceiveHandler : OnReceive
+ DatagramSocketWorker(port : Integer, onReceiveHandler : OnReceive)
+ add(UDPConnector connector) : void
+ remove(UDPConnector connector) : void
+ run() : void
<<package>>
java.lang
<<interface>>
Runnable
+ run() : void
<<implements>>

Figure 5.10: Class diagram - Entity Communication Point - UDP Connector

The *DatagramSocketWorker* contains a list of allowed addresses that can be accepted as packet sources. When a new device is connected, the address is added to the list. When a packet arrives, it is checked for its validity.

The *EntityUDPConnector* represented in the figure 5.11 is an extension to the UDP connector. The protocol of communication defined in the design for the three-way handshake and packet loss prevention is built on top of the existent UDP implementation. By reusing the implementation of the *UDPConnection* module, the complexity of sending and receiving packets is reduced to the use of the two methods *send* and *receive*.



<<package>>
com.fgouveia.tpmsd.ecp.sdk.connector
UDPConnector
- socket: Datagram
+ UDPConnector(endpoint : String, port : Integer, onReceiveHandler : OnReceive)
+ getProtocol() : String
+ getEndpoint() : String
+ getPort() : Integer
- startReceivingData(onReceiveHandler : OnReceive)
- sendData(data : byte[])
<<extends>>
EntityUDPConnector
- connectionEstablished: Boolean
+ EntityUDPConnector(endpoint : String, port : Integer, onReceiveHandler : OnReceive)
- startReceivingData(onReceiveHandler : OnReceive)
- sendData(data : byte[])

Figure 5.11: Class diagram - Entity Communication Point - Entity UDP Connector

To manage the connections and their actions at the ECP, such as command sending to entities or establishment of a new connection, REST services were implemented. They are meant to be used by the client interface application, according to the user request.

The URL `http://<ecp_endpoint>/ECP/webresources/connector/execute?command=receive&endpoint=192.168.0.100&port=1234&protocol=tpmsd.entity` is an example of a service endpoint to establish connection to the address 192.168.0.100, on the remote port 1234 and using the protocol *tpmsd.entity*. Currently, the protocol names are hardcoded in the *ConnectorFactory*'s implementation.

5.2.2 Queue

The Queue is implemented as a buffer to the packet handlers, as described in the system design. It is implemented with three basic methods: *add*, *get* and *remove*. Those methods are accessible through a REST web services' interface. The result of the service calls is in JSON format [27]. To simplify the transfer of data from and to the queue through HTTP requests, the messages that the queue handles are in the String format. To avoid characters in the messages that would interfer with the JSON format and to have a standardized[28] mechanism to convert binary data into and from String format, the Base64 is used.

Base64 is an encoding mechanism which uses 6-bits to represent a character in the list of 64 characters (+1 for padding) of the Base64 alphabet. Each composition of four characters in Base64 format (24-bit) represents a block that can be splitted into 3 bytes [28, p. 6]. The Base64 alphabet is composed by lower-case and upper-case letters, a plus sign and a forward-slash. If the binary data does not fill the last 24-bit block, the equal sign is used for padding. This range of characters do not interfer in the JSON syntax.

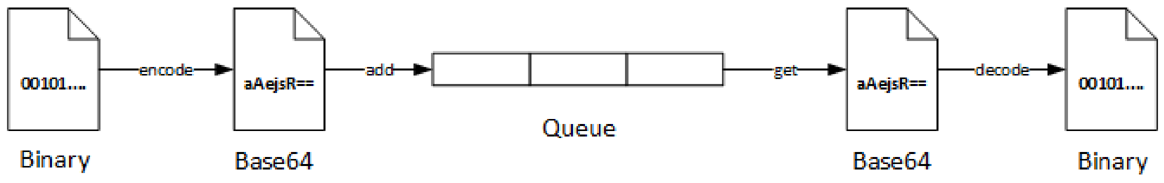


Figure 5.12: Queue: Base 64 encoding and decoding

To add a message to the queue, the message needs to be encoded first into Base64 (see figure 5.12). Then, a HTTP Post request to the URL `http://<queue_endpoint>/Queue/webresources/queue/add` is done, with the encoded message in the body. The expected result is a JSON map with a key-value pair representing the success of the operation (see listing 4). If the result is *true*, the message is added with success.

```
{
    "result":"true|false"
}
```

Listing 4: Queue add method's response

To get a message from the queue, the URL `http://<queue_endpoint>/Queue/webresources/queue/get` is used. It retrieves the oldest unscreened message with a message handler. The message handler serves to identify the message uniquely. This handler is later used to remove the message. If there are no messages in the queue, the request connection stays waiting until the new message arrives. This process of waiting is done with Java monitor. When a message is retrieved, it is screened from the buffer, not being possible to retireve it in a defined interval of time. If after the interval of time the message is not removed, it is put back into the queue (see figure 3.9). The message comes enconded in Base64 format. For that reason, a decoding needs to be done to obtain the original message (see figure 5.12). The listing 5 is an example of a get method's response.

```
{
    "content":"AAABQXWsCyVFbnRlciBkYXRhIHRvIHNlbmQuLi4=",
    "uuid":"e5894b3e-99bd-41f8-9410-ec86dc597caa"
}
```

Listing 5: Queue get method's response example

To remove a message, the URL `http://<queue_endpoint>/Queue/webresources/queue/remove?uuid=<messagehandler>` is used, with the *uuid* parameter obtained in the *get* method. If the message is removed successfully, the service returns a positive answer (see listing 6 to see the remove method's response format). Otherwise, if the service returns a negative answer, it means that some other process obtained the same message and processed it faster (refer to chapter 3 for details).

```
{
    "result":"true|false"
}
```

Listing 6: Queue remove method's response

5.2.3 Packet Handler

The packet handler obtains the packets from the queue, processes them and stores the processed values into a NoSQL database. It is implemented as a JSE application, to avoid the need of a web application server. As a Daemon, it runs without the need of user interaction. It can run in multiple cores, as the number of running Daemons can be defined when starting the application (see listing 7). By default, four Daemons are initialized.

```
{
    java -jar Daemon.jar <num_daemons>
}
```

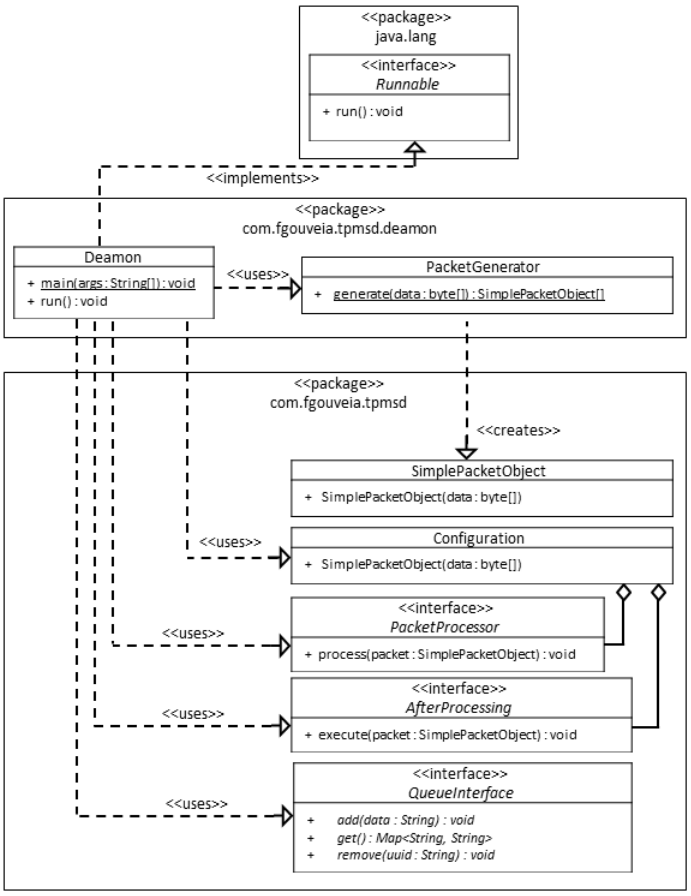
Listing 7: Packet handler initialization

To not be dependent on the implementation, the Daemon's components are modular.

The interface to obtain the message from the queue is the same as in the ECP (the $QueueInterface$ on the figure 5.13 is the same as on the figure 5.9).

When the data is received, it is decoded from Base64 to the original binary data. The class *SimplePacketObject* is an implementation with the base functionality to interpret the information in the binary data. This class can be extended to interpret different data types and provide an interface to edit the values. An implemented extension is the *SimpleDataObject*, which reads the data from the sensors and allows the data to be edited later in the processing steps.

There are two processing interfaces that can be implemeted: *PacketProcessor* and $AfterProcessing$. The former serves the purpose to modify the read data. An example where it can be useful is when the floating point values from the sensors are calculated to be



<<package>>
java.lang
<<interface>>
Runnable
+ run() : void
<<implements>>
<<package>>
com.fgouveia.tpmsd.deamon
Deamon
+ main(args : String[]) : void
+ run() : void
<<uses>>
PacketGenerator
+ generate(data : byte[]) : SimplePacketObject[]
<<package>>
com.fgouveia.tpmsd
<<creates>>
SimplePacketObject
+ SimplePacketObject(data : byte[])
<<uses>>
Configuration
+ SimplePacketObject(data : byte[])
<<uses>>
<<interface>>
PacketProcessor
+ process(packet : SimplePacketObject) : void
<<uses>>
<<interface>>
AfterProcessing
+ execute(packet : SimplePacketObject) : void
<<uses>>
<<interface>>
QueueInterface
+ add(data : String) : void
+ get() : Map<String, String>
+ remove(uuid : String) : void

Figure 5.13: Class diagram: Scallable Packet Handler

a short number, so that it can be placed in packet's data placeholder - e.g.: a sensor reading the value 19.20, transforms by multiplying the number by 100 and sends it as 1920. When it arrives to the packet hander's processor, the reverse is done to obtain the original value, by dividing it by 100 (see example in listing 8).

```
public class TemperaturePacketProcessor implements PacketProcessor{
    private static final int PACKET_TYPE_TEMPERATURE = 1;

    @Override
    public void process(SimplePacketObject packet) {
        for(SimpleDataObject obj : packet.getSensorsData()){
            // Checks if it is a temperature packet
            if(obj.getSensorTypeID() == PACKET_TYPE_TEMPERATURE){
                obj.updateValue(obj.getValue() / 100); // Calculates back the
                    original number
            }
        }
    }
}
```

Listing 8: Packet handler processor example

The latter interface, *AfterProcessing*, is meant to perform tasks with the data after being processed, without changing it. One possible action is the storage of the values. The example in the listing 9 is the implementation of this interface to store data in the MongoDB NoSQL database.

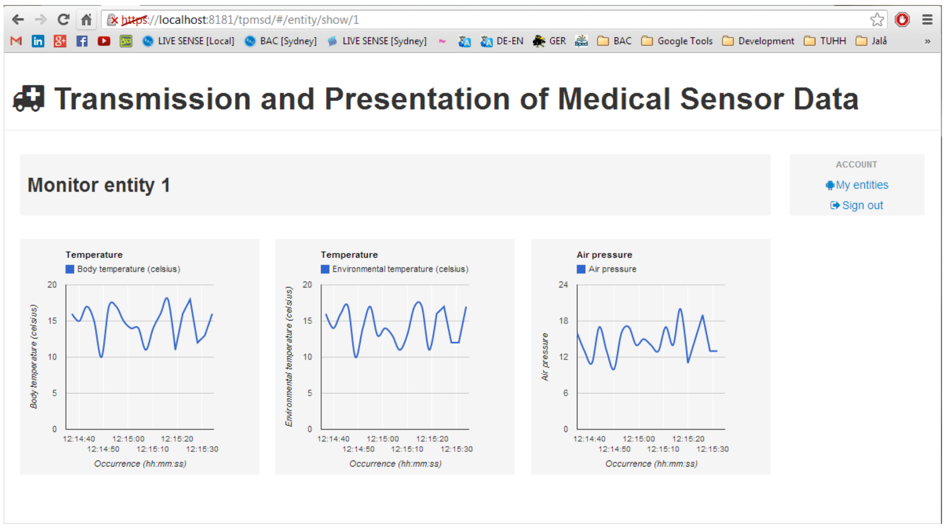
```
public class StoreToMongoDB implements AfterProcessing {
    private DBCollection dbCollection;
    // ... Construtor with DBCollecton initialization
    @Override
    public void execute(SimplePacketObject packet, Debug debug) {
        List<DBObject> list = new LinkedList<>();

        for (SimpleDataObject sdo : packet.getSensorsData()) {
            BasicDBObject o = new BasicDBObject("sensor",
                    new BasicDBObject("sensorID", sdo.getSensorID())
                    .append("sensorTypeID", sdo.getSensorTypeID())
                    .append("entityID", packet.getEntityID()))
                    .append("timestamp", sdo.getTimestamp())
                    .append("value", sdo.getValue())
                    .append("ecpTimestamp", packet.getEcpTimestamp());

            list.add(o);
        }
        dbCollection.insert(list);
    }
}
```

Listing 9: Packet handler after processing example

As it happens in the ECP, the processing instances run sequencially in a chain.

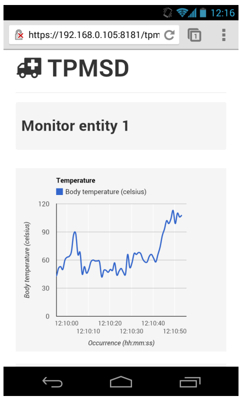
5.3 Client interface

For the client application of this project, a web-based application is developed. The choice of a web-based application is based on the need of a multi-platform client interface that can be easily adapted to different screen sizes.



https://localhost:8181/tpmsd/#/entity/show/1
Transmission and Presentation of Medical Sensor Data
Monitor entity 1
ACCOUNT
My entities
Sign out
Temperature
Body temperature (celsius)
Environmental temperature (celsius)
Air pressure
Occurrence (hh:mm:ss)

(a) Client interface: Desktop interface



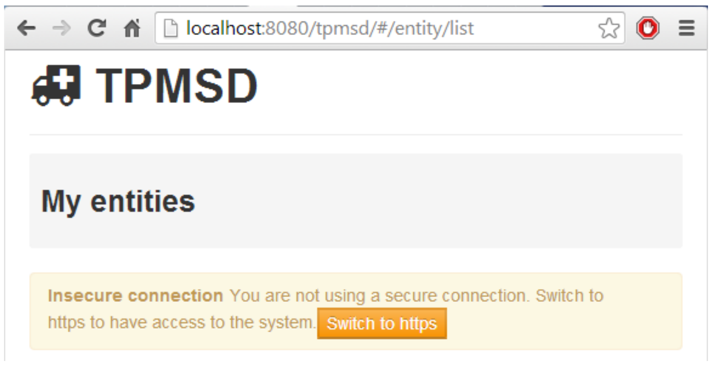
(b) Client interface: Mobile interface

Figure 5.14: Client interface: Responsive design

The application is built as a SPI. It is based on a single HTML page and AJAX requests in order to update page information and presentation. The AJAX requests are made to REST webservices created at the web interface server side application. The REST webservices provide information, such as the user's entities and sensor values' history. The REST webservices do not keep the state between requests. In order to use the webservices, an authentication token is required. In order to generate this token, an authentication is required in the authentication service. The use of such an implementation allows the development of native applications which can use the system by making requests to the webservices in order to obtain information.

To aid the development of the web interface application, the jQuery framework is used. To manage page Uniform Resource Locator (URL) s with a non-refreshing page, the library Sammy.js is used. For a responsive user interface, adaptable to user's screen size, the Twitter Bootstrap is used (see figures 5.14a and 5.14b, for desktop and mobile interface, respectively).

For security reasons, the page and the webservices cannot be used with a connection without security. If the page is loaded with an `http://<endpoint>/tpmsd` prefix, an error is displayed and a button to switch to `https://<endpoint>/tpmsd` (see figure 5.15).



Insecure connection You are not using a secure connection. Switch to https to have access to the system.

Figure 5.15: Client interface: Secure connection required

In order to support the management of the information in the application, the MySQL relational database is used. The figure 5.16 contains the database architecture.

The *user* table contains the username and a password hash generated with a random *salt*. By using *salt* in the password hash, the difficulty is increased to guess passwords that fit to the hash [29], in case the database is compromised.

Related to the *user* table, there is an *usersession*. This table stores temporary valid tokens of user's authentication. A token authentication fails in the services if a non-existent token or a token after its expiration date is used.

Each *user* is related to entities. Each *entity* is composed by sensors. Each *sensor* has a type. The ID fields in the *entity*, *sensor* and *sensortype* tables are defined in the hardware. With those identifiers, it is possible to query the sensor data database in order to obtain the correct sensor data.

Even though the *user* is related to entities, the table *usersensordashboard* defines which sensors are shown in the user's dashboard, and with which visualization type. Currently,

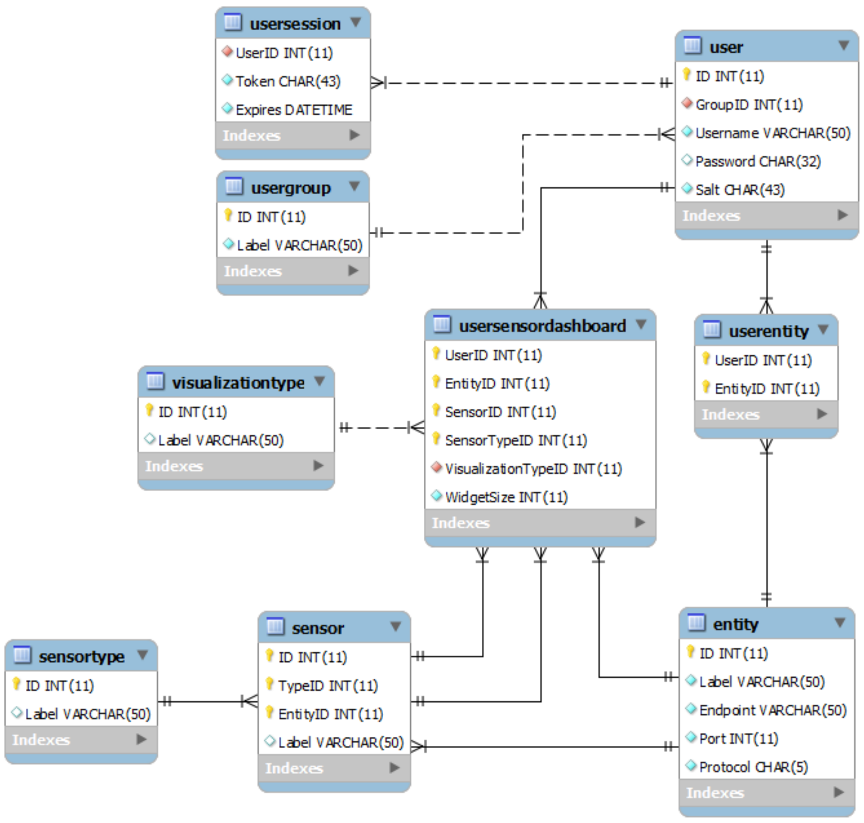


Figure 5.16: Client interface: Database architecture

only chart visualization type is implemented in the javascript, but can be extended to other types.

Chapter Six

Results

In order to analyse the performance of the system, the execution time of the components is measured in a cumulative average (see chapter 5).

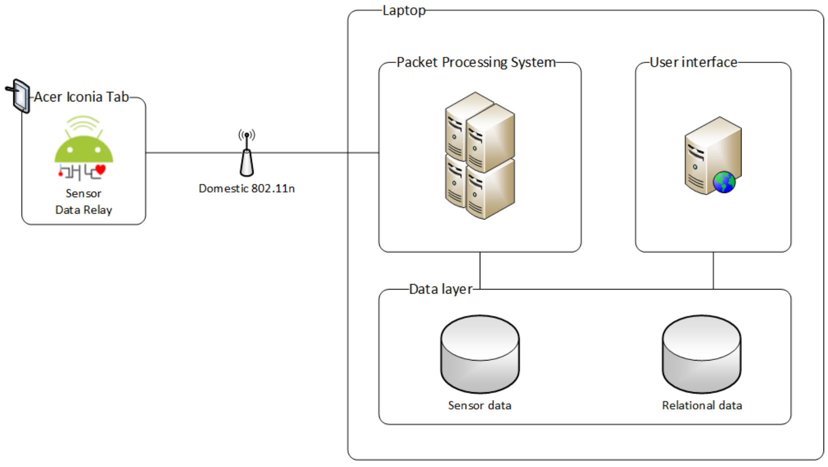


Figure 6.1: Wearable ECG-recording System: Devices [6]

First of all, it is important to mention how the system is deployed during these tests. As the figure 6.1 illustrates, the mobile application SDR is running on an Acer Iconia Tab A200 tablet running Android 4.0.3. On the other side, the ECP, the queue, the SPH and the Web interface are runnning on the same laptop computer. The relevant hardware to be mentioned is an Intel Core$^{\text{TM}}$i7 processor, 8 GB of DDR3 1333MHz memory, a primary SSD drive running the operative system Microsoft ®Windows 7, a secondary hard drive with 7200rpm where both MySQL and MongoDB databases are stored and a 802.11n connection.

In this tests, the time measured between the SDR and the rest of the system is an estimative. It is not guaranteed that the time a packet needs to be transmitted from the ECP to the SDR will be the same as in the other way around. Still, the estimation of latency is done with the average of the two ways.

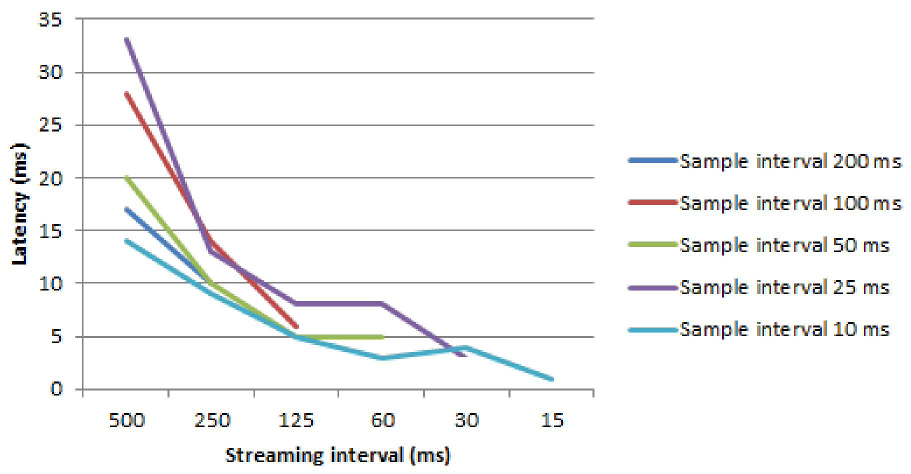


Figure 6.2: Results: Latency

Taking in consideration the sensors' sample interval, as well as the streaming interval to the ECP, tests were made with different combinations. All the tests were made twice to the UDP protocol with samples of, at least, 1000 packets, each. The sample interval is the difference of time between the reading of two sensor values. The streaming interval is the time difference between two packet sendings from the SDR. Considering, as an example, a sample interval of 200ms, which is equivalent to 5 reads per second, and a streaming interval of 500ms (2 packets sent per second), each packet contains in average 2.5 sensor values.

Not all the lines reach the end of the charts. The reason is that if the sample interval is bigger than the streaming interval, then empty packets are transmitted. In these cases, the tests were not considered.

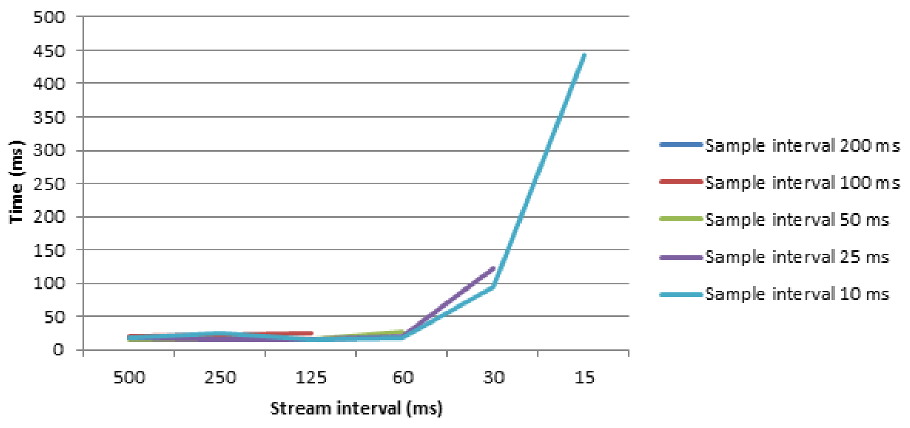


Figure 6.3: Results: Processing time in the packet processing system

By measuring the latency it is possible to conclude that, as the number of packets sent per second increase, the latency seems to decrease. This is an interesting result, as the latency is not calculated based on the sensor data packets. Also interesting is the fact that the latency decreases with the increasing number of received packets per second. The reason for this event could not be found and the implementation of the latency calculation should not have any influence on the data received and vice versa.

The figure 6.3 shows that the processing time required, from the ECP up to the end of the Packet Handler processing, increases as the number of packets sent per second increases as well.

Both of the last measures influence the total time the packets need to be processed until they reach the database. By analysing the total time required for the packets to be processed, the conclusion is that the system, running in the conditions stated above, have the most efficient performance when the packets are streamed each 125ms (see figure 6.4).

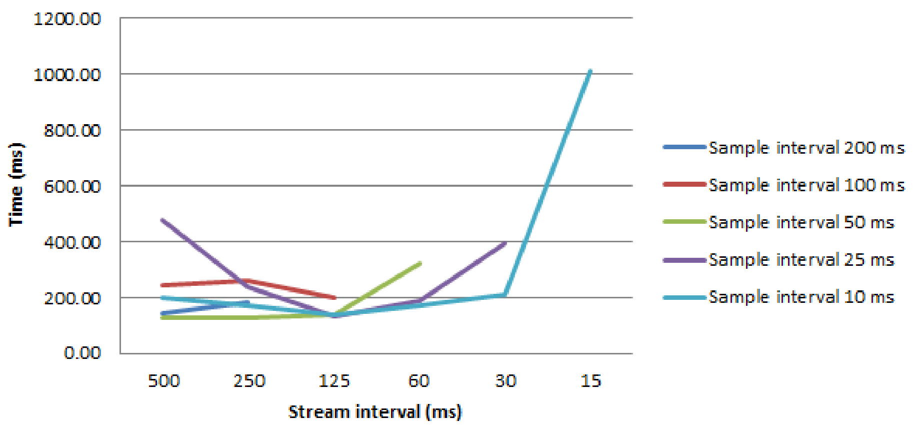
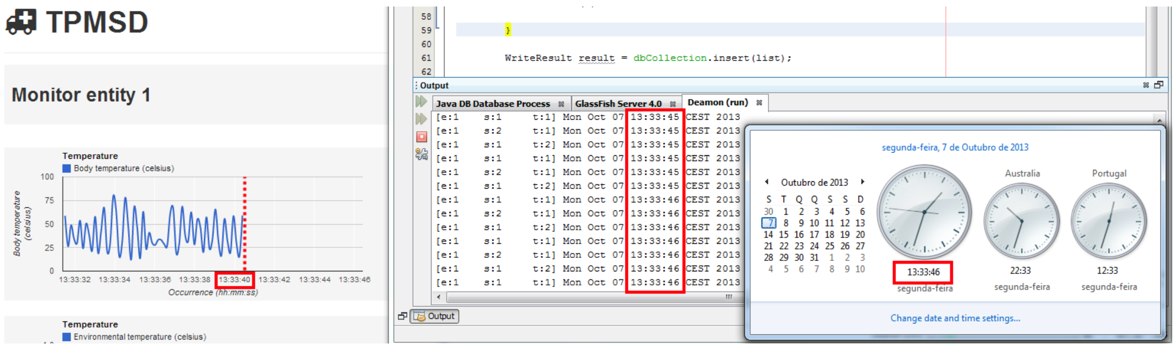


Figure 6.4: Results: Total processing time, starting on the packet sending of SDR until the storage in the database

However, the database seems to struggle retrieving the most recently added results. The figure 6.5 is a screenshot which shows that the data is sent to the database with a delay of less than one second. On the user interface, it is presented with a delay of 6 seconds. In this example, the data is tested with a sample interval of 150ms and a streaming interval of 200ms.



```
        }

        WriteResult result = dbCollection.insert(list);
```

```
[e:1    s:1     t:1] Mon Oct 07 13:33:45 CEST 2013
[e:1    s:2     t:1] Mon Oct 07 13:33:45 CEST 2013
[e:1    s:1     t:2] Mon Oct 07 13:33:45 CEST 2013
[e:1    s:1     t:1] Mon Oct 07 13:33:45 CEST 2013
[e:1    s:2     t:1] Mon Oct 07 13:33:45 CEST 2013
[e:1    s:1     t:2] Mon Oct 07 13:33:45 CEST 2013
[e:1    s:1     t:1] Mon Oct 07 13:33:46 CEST 2013
[e:1    s:2     t:1] Mon Oct 07 13:33:46 CEST 2013
[e:1    s:1     t:2] Mon Oct 07 13:33:46 CEST 2013
[e:1    s:1     t:1] Mon Oct 07 13:33:46 CEST 2013
[e:1    s:2     t:1] Mon Oct 07 13:33:46 CEST 2013
[e:1    s:1     t:2] Mon Oct 07 13:33:46 CEST 2013
[e:1    s:1     t:1] Mon Oct 07 13:33:46 CEST 2013
```

Figure 6.5: Results: Delay in live data presentation

Chapter Seven

Conclusion and future work

In this work, a scalable distributed system applied in a remote vital signs monitoring scenario is presented. The decouple of main functionalities of a system into separate autonomous components makes their development less complex and easier to test. Moreover, the tasks can be distributed to replicas of components, such as the scalable packet handler allowing the data processing being executed in parallel. The queue system is implemented in such a way that prevents the loss of data due to packet processing failures. If a packet is processed by a packet handler which stops working, the packet will be put back into the queue after some time, so that another packet handler restarts the unfinished job. Therefore, the risk of such critical system to fail is reduced. Furthermore, the parallelization of the components opens the possibility to scale the system according to the usage scenario. On top of that, this can be done while the system is running by just stating a new instance of the component. This means that the system does not need to be restarted and the change is transparent to the user, unless the increase or decrease of performance is apparent. Also important to mention is the extensability of the system. The fully modular design allows implementation of key components to support new functionalities and different devices.

The system is constituted by a HHD application, a packet receiver component, a decoupled queue system, a set of packet processing modules and a web interface. The storage of the sensor data is made in a NoSQL database, while the administration part of the web interface is stored in a relational database.

The implemented system is a fully working solution as shown in chapter 5. However, some adjustments and additional features can improve the system functionality and performance.

One of the features is the implementation of different connectors to support different devices. A connector, that is not implemented, yet, is the Droid Jacket's connection through TCP/IP protocol. Currently, UDP is implemented and an extension to the procotol exists to support the SDR.

Additionally, further packet processing modules' development may be taken into consideration. Appart from just re-calculating values, as the listing 8 shows, a module can be implemented to detect alarming situations and act accordingly.

Regarding performance, a direct connection from the packet handlers to the web interface's server can be taken into consideration. This is due to the time required for the database to retrieve instantaneously the last stored data and a solution to reduce the need of reads of the database. By sending the data directly to the web server, and using a WebSocket from the web server to the client's browser, the data can be streamed as soon as it is processed by the packet handlers without additional queries to the database (see figure 7.1).

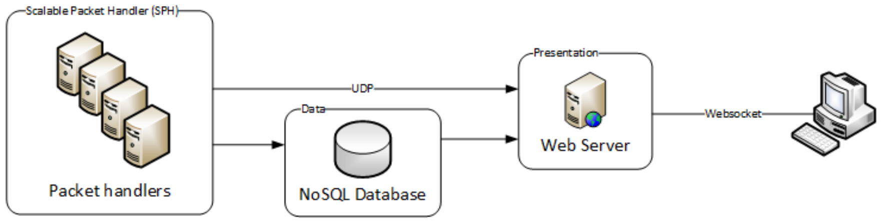


Figure 7.1: Direct connection from packet handlers to the web server and Websocket connection to clients' browser

Stress tests to the system on a real distributed environment are needed to understand how the components act in physically separated machines and to understand when it is necessary to increase or reduce the scalable packet handler. It is also important to evaluate the transmittion of data with a less reliable and slower network connection, such as mobile broadband connection in zones where its coverage is reduced.

A missing mandatory feature in the SDR application is the bluetooth connection to a sensor gateway. Without it, the data cannot be obtained from the sensors. Currently, the Android application serves as a simulator with user interaction.

Finally, the web interface needs further design and implementation regarding the management of users, entities and visualizations. Currently, the web interface allows the reading of information from the existent database. Besides, there is one kind of visualization implemented in the Javascript, with the Google Chart API. More realisations are possible.

Glossary

AA4R	Ambient Assistance for Recovery
AIDL	Android Interface Definition Language
AJAX	Assynchronous Javascript And XML
API	Application Programming Interface
BIOSal	Biological Signal Acquisition Layer
CITEVE	Technological Centre for the Textile and Clothing Industries of Portugal
CRC	Cyclic Redundancy Code
DOM	Document Object Model
ECG	Electrocardiogram
ECP	Entity Communication Point
EJB	Enterprise Java Beans
FTP	File Transfer Protocol
GPS	Global Positioning System
GUI	Graphical user interface
HHD	Hand Held Device
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IEETA	Electronics and Telematics Engineering Institute of Aveiro
ITALH	Information Technology for Assisted Living at Home
JEE	Java Enterprise Edition
JME	Java Micro Edition
JSF	Java Server Faces
JSON	Javascript Object Notation
JSP	Java Server Pages
JSE	Java Standard Edition
JVM	Java Virtual Machine
LAN	Local Area Network
OSGi	Open Services Gateway initiative
PDA	Personal Digital Assistant
POJO	Plain Old Java Objects
RDMS	Relational Database Management System
RF	Radio-frequency transmitter
RI	Reference Implementation
SDR	Sensor Data Relay
SPH	Scalable Packet Handler
SPI	Single Page Interface
SQL	Structured Query Language
URL	Uniform Resource Locator
VM	Virtual Machine
VPN	Virtual Private Network
XML	eXtensible Markup Language

References

[1] K. Catulo, "Descobrir os velhos que se tornaram invisíveis", *Jornal i*, pp. 28–29, Feb. 2012.

[2] A. Bitner, P. Zalewski, J. Klawe, K. Gorynski, M. Zawadka, and J. Pawlak, "Heat exposure effects and kinds of illnesses among firefighters - review", *Medical and Biological Sciences*, vol. 26, no. 2, pp. 69–72, Oct. 2012. DOI: `10.2478/v10251-012-0034-6`.

[3] D. Teles, M. Colunas, J. Fernandes, I. Oliveira, and J. Cunha, "Ivital: a real time monitoring system for first response teams", in *Mobile Networks and Management*, ser. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, K. Pentikousis, R. Aguiar, S. Sargento, and R. Agüero, Eds., vol. 97, Springer Berlin Heidelberg, 2012, pp. 396–404, ISBN: 978-3-642-30421-7. DOI: `10.1007/978-3-642-30422-4_29`. [Online]. Available: `http://dx.doi.org/10.1007/978-3-642-30422-4_29`.

[4] *AA4R Fail Safety*, `http://www.aa4r.org/files/Fail_Safety_in_AA4R-121101.pdf`, [Online; accessed 8-April-2013].

[5] *Intelligent Telemetry for Implants*, `http://www.compamed.de/cipp/md_compamed/custom/pub/content,oid,19798/lang,2/ticket,g_u_e_s_t/~/Intelligent_Telemetry_for_Implants.html`, [Online; accessed 10-July-2013], Nov. 2010.

[6] R. Fensli, E. Gunnarson, and T. Gundersen, "A wearable ecg-recording system for continuous arrhythmia monitoring in a wireless tele-home-care situation", in *Computer-Based Medical Systems, 2005. Proceedings. 18th IEEE Symposium on*, 2005, pp. 407–412. DOI: `10.1109/CBMS.2005.22`.

[7] *Vital Responder Project*, `http://www.vitalresponder.pt/index.php?option=com_content&view=article&id=2&Itemid=3`, [Online; accessed 5-April-2013].

[8] J. P. S. Cunha, B. Cunha, A. Pereira, W. Xavier, N. Ferreira, and L. Meireles, "Vital-jacket: a wearable wireless vital signs monitor for patients' mobility in cardiology and sports", in *Pervasive Computing Technologies for Healthcare (PervasiveHealth), 2010 4th International Conference on-NO PERMISSIONS*, 2010, pp. 1–2. DOI: `10.4108/ICST.PERVASIVEHEALTH2010.8991`.

[9] A. Trigo, C. J. P. S, C. M. B., W. Xavier, and F. N. S., Eds., *Wireless bedside vital signs monitoring unit*, Luxemburg: Med-e-Tel, 2004.

[10] M. Colunas, J. Fernandes, I. Oliveira, and J. P. S. Cunha, "Droidjacket: an android-based application for first responders monitoring", in *Information Systems and Technologies (CISTI), 2011 6th Iberian Conference on*, 2011, pp. 1–4.

[11] M. Colunas, J. Fernandes, I. Oliveira, and J. Cunha, "Droid jacket: using an android based smartphone for team monitoring", in *Wireless Communications and Mobile Computing Conference (IWCMC), 2011 7th International*, 2011, pp. 2157–2161. DOI: `10.1109/IWCMC.2011.5982868`.

[12] M. Figueredo and J. Dias, "Mobile telemedicine system for home care and patient monitoring", in *Engineering in Medicine and Biology Society, 2004. IEMBS'04. 26th Annual International Conference of the IEEE*, IEEE, vol. 2, 2004, pp. 3387–3390.

[13] J. M. Eklund, T. R. Hansen, J. Sprinkle, and S. Sastry, "Information technology for assisted living at home: building a wireless infrastructure for assisted living", in *Engineering in Medicine and Biology Society, 2005. IEEE-EMBS 2005. 27th Annual International Conference of the*, IEEE, 2006, pp. 3931–3934.

[14] J. Postel, "User datagram protocol", *Isi*, 1980.

[15] ——, "Rfc 793: transmission control protocol, september 1981", *Status: Standard*, 2003.

[16] J. Varia, "Architecting for the cloud: best practices", *Amazon Web Services*, 2010.

[17] ——, "Cloud architectures", *White Paper of Amazon, jineshvaria. s3. amazonaws. com/public/cloudarchitectures-varia. pdf*, 2008.

[18] *Amazon Simple Queue Service - Developer Guide API version 2012-11-05*. 2012.

[19] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley, "The java tm language specification java se 7 edition", 2012.

[20] D. Kramer, "The java platform", *White Paper, Sun Microsystems, Mountain View, CA*, 1996.

[21] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, *The Java Virtual Machine Specification, Java SE 7 Edition*. Pearson Education, 2013, ISBN: 9780133260465. [Online]. Available: `http://www.google.de/books?id=95HzjxTELRkC`.

[22] *Your first cup: An introduction to the java ee platform*, `http://docs.oracle.com/javaee/6/firstcup/doc/p1.html`, [Online; accessed 12-August-2013], Apr. 2012.

[23] A. Goncalves, *Beginning Java EE 6 with GlassFish 3*. Apress, 2010.

[24] E. Jendrock, R. Cervera-Navarro, I. Evans, D. Gollapudi, K. Haase, M. William, and C. Srivathsa, *The java ee 6 tutorial*, `http://docs.oracle.com/javaee/6/tutorial/doc/index.html`, [Online; accessed 12-August-2013], Jan. 2013.

[25] *Java Server-side Programming: Getting started with JSP by Examples*, `http://www.ntu.edu.sg/home/ehchua/programming/java/JSPByExample.html`, [Online; accessed 12-September-2013], Oct. 2012.

[26] *Bound-services, Android Developers*, `http://developer.android.com/guide/components/bound-services.html#Binder`, [Online; accessed 27-September-2013].

[27] D. Crockford, "The application/json media type for javascript object notation (json)", 2006.

[28] S. Josefsson, "The base16, base32, and base64 data encodings", 2006.

[29] A. Baldwin, "Enhanced accountability for electronic processes", in *Trust Management*, Springer, 2004, pp. 319–332.

[30] *Bluetooth, Android Developers*, `http://developer.android.com/guide/topics/connectivity/bluetooth.html`, [Online; accessed 10-April-2013].

[31] H. Weberpals, *Parallel programming primer*, `http://www.tuhh.de/parallel/cw/lecture.html`, [accessed December-2012; Computational Web lecture at Technical University of Hamburg-Harburg], Dec. 2012.

Appendix

Contents

- This Master's Thesis document in pdf format
- Complete source code of the developed project
  - Sensor Data Relay mobile application
  - Packet Processing System
    - Entity Communication Point
    - Queue
    - Packet Handler
- Software deployment manual

CD