Tatiana Tassi

Implementation of an Educational Wireless Biopotential Recorder Application

Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

14 September 2016

Author(s) Title Number of Pages Date	Tatiana Tassi Wireless Biopotential Recorder 32 pages + 3 appendices September 2016
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Software Engineering
Instructor(s)	Antti Piironen, Principal Lecturer Sakari Lukkarinen, Lecturer

The goal of the project was to implement a system to record electroencephalography biopotential measurements, and to send them wirelessly to a cloud storage service. Moreover, the application should allow the user to view and manage the recorded data using a browser. The application was planned to be used for educational purposes as a part of courses at Metropolia University of Applied Sciences. The EEG measurements were recorded using an OpenBCI 32bit board while the software application was implemented using Python 2.

The measurements consisted of a potential difference between each of the input electrodes and a reference electrode. The voltages were produced by the activity of the neurons closest to the area where the electrode was positioned. Displaying the measured voltages from each individual channel over time allowed the user to detect patterns in the signals produced by brain activity. In particular, patterns within defined frequency ranges correlated to specific types of brain activities.

The result of the project was an application which allows users to easily record EEG signals, which are then transmitted to and stored in the Metropolia cloud. The recordings can be viewed using the web application implemented as part of the project. In addition to displaying the recordings stored in the cloud, the application allows user to manage and organize the recordings based on parameters such as the patient from which they have been measured, the course of which the recording session is a component, and the instructor responsible for each course.

The initial goal of the project was achieved successfully. However, several improvements are possible as regards the security and the quality of the user experience of the application. Moreover, the system requires to be adapted to eventual future changes to the Metropolia cloud infrastructure. Nevertheless, the current state of the project can be useful as an educational tool and as a foundation for future development.

Keywords	OpenBCI, electroencephalography, Python, brain-computer interfaces

Contents

Abbreviations and Terms

BCI	Brain-computer interface
CNS	Central nervous system
ATP	Adenosine triphosphate
ECG	Electrocardiogram
EEG	Electroencephalogram
EMG	Electromyogram
GUI	Graphical user interface
TCP	Transmission control protocol
UDP	User datagram protocol
EOT	End of transmission
API	Application programming interface
ER model	Entity-relationship mode
HTML	Hypertext markup language
VM	Virtual machine

1 Introduction

The goal of the project is to implement a system to record electroencephalography biopotential measurements and to send them wirelessly to a cloud storage service. The server should also allow the user to retrieve and display the stored data. The final goal of the project is to provide an educational tool to be employed in courses related to healthcare and medical instrumentation. For this reason, the project includes the development of an application which allows the user to view the recorded samples as line charts. Moreover, the application will include tools to organize the stored sessions based on courses, instructors, and patients. Finally, the application has to be implemented to run on the Metropolia educational cloud.

The biopotential measurements will be recorded and handled using an OpenBCI 32-bit board. OpenBCI is an open-source platform aimed at measuring, recording, and processing biopotential measurements, focusing mainly on brain-computer interfacing. The platform includes both hardware and software tools, such as the now deprecated OpenBCI 8-bit board, the 32-bit board, the Daisy module, and the Ganglion bio-sensing device which is currently under development.

In order to provide a better understanding of the phenomena underlying the measurements gathered by the previously mentioned hardware, this paper will include an overview of the structure and physiology of the nervous system. Moreover, the processes involved in signal reception, processing, and transmission in neurons will be described. Finally, the topic of the hardware and software tools employed in the project will be analysed.

2 Theoretical Background

2.1 Introduction

The present chapter will review the theoretical topics on which the project implementation is based. The considered topics include the physiology of the nervous system, the components of the medical instrumentation used in encephalography, the OpenBCI tools used in the project, and the programming language and libraries used in the project.

In the next subchapter, the physiological mechanisms which produce the signals measured through electroencephalograms will be described. The structure of the nervous system will be described, with particular focus on the central nervous system (CNS). Moreover, neurons and their functioning will be described in further detail. Thirdly, an overview of brain waves will be provided. Finally, the concepts of biopotential measurements and electroencephalography will be discussed.

2.2 Physiology

2.2.1 Nervous System

Electroencephalography involves the measurement of biopotentials generated by the brain [1]. Biopotentials are electrical signals produced by body cells and tissues in connection with biochemical activity [2]. In order to achieve a clearer understanding of the physiology underlying these biopotentials, the following section will describe the nervous system. The main functions of the nervous system include the following:

- control of skeletal muscles thus allowing motion
- perception and interpretation of stimuli received by the body, such as sound and temperature
- management of automatic functions in other body systems, such as circulation and digestion, consciousness, thought, emotions.

The nervous system performs the previously listed functions through exchanges of signals between neurons and with other body cells. [3;4.]

From a physiological point of view, the nervous system can be divided into the central nervous system and the peripheral nervous system. The central nervous system is composed of two interconnected parts: the brain and the spinal cord. The main components of the brain include the cerebrum, the cerebral cortex, and the brain stem which connects the brain and the spinal cord. [3.]

Figure 1 illustrates the structure of the nervous system.

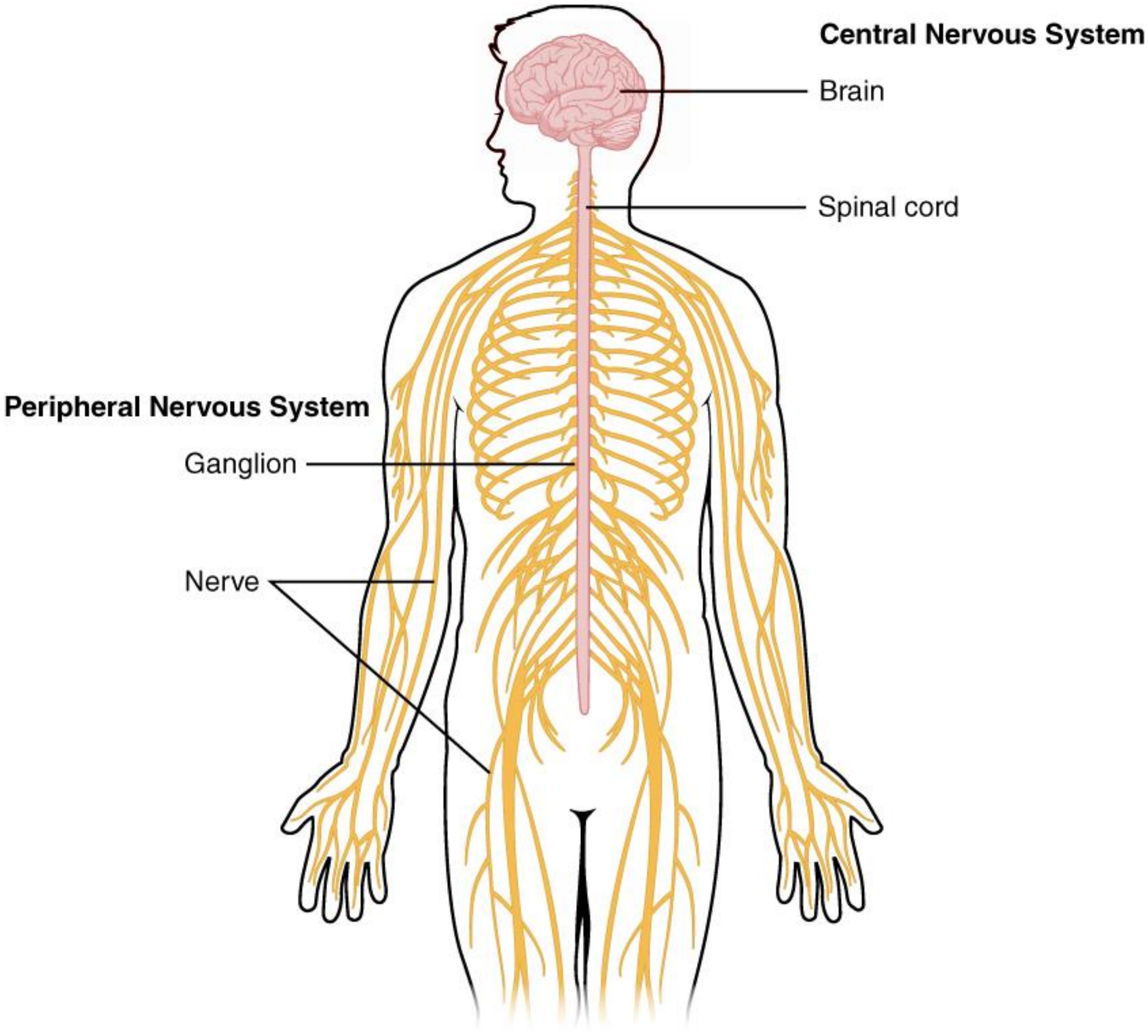


Figure 1. Human nervous system. Reprinted from OpenStax College (2013). [5.]

The nervous system is composed of neural cells, which can be divided into neurons or nerve cells, and glia or glial cells. Both subsystems include neurons, but the types of glial cells present in each subsystem are different. Neurons have the function of receiving, processing and transmitting information, while glia cells are located between nerve cells and have a function of the support and protection of nerve cells. [3;4.]

2.2.2 Neurons

Neurons are characterized by a body called soma and by a varying number of processes. Two types of neuron processes exist: dendrites and axons. Each neuron has several dendrites, which are shorter than the axons and feature several small ramifications. Neurons usually have only one axon, which splits into branches in some types of nerve cells, and has the function of carrying information from the neuron towards other nerve cells or target organs through output terminals. Neurons receive information from other cells through synapses, which are located on dendrites, soma, and axon hillock. Figure 2 shows the general structure of neuronal and glial cells. [3,4]

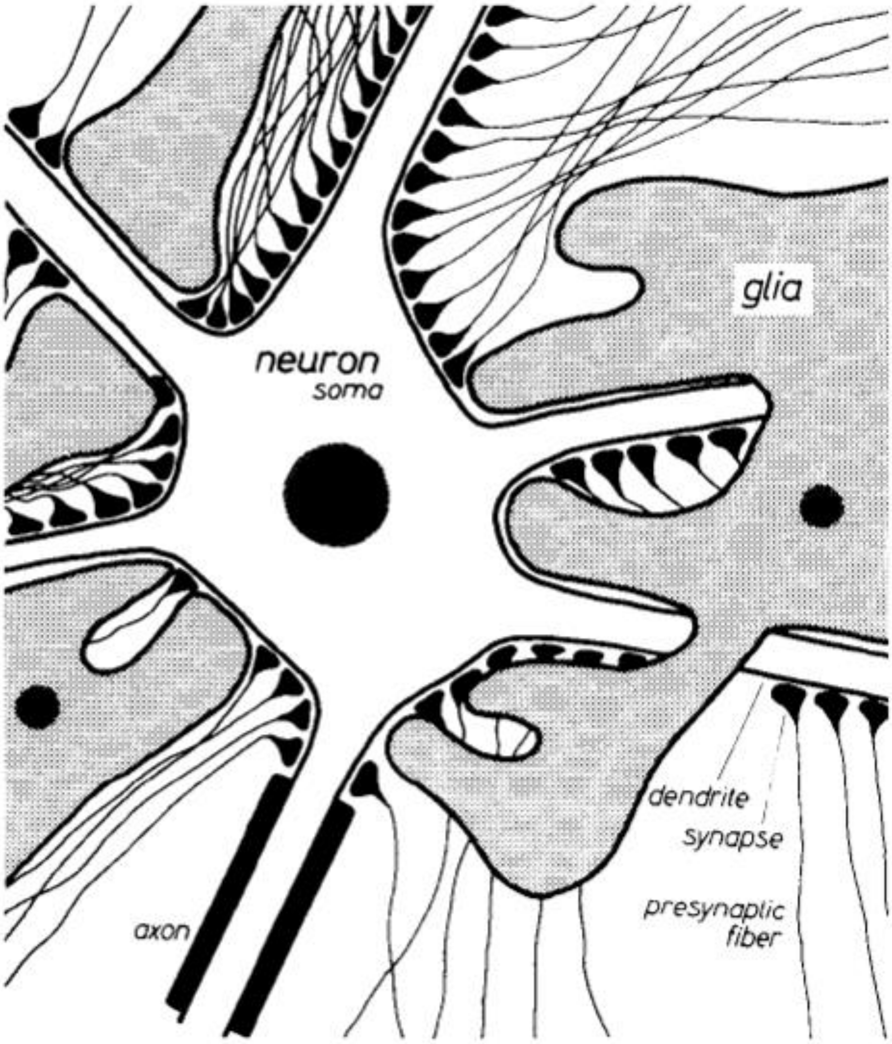


Figure 2.Morphology of neuronal and glial elements. Reprinted from Niedermeyer [4].

Among the variety of existing types of neurons, one of the main differences is the direction of information transmission between central and peripheral nervous system. Afferent neurons carry information about stimuli to the central nervous system. Efferent neurons carry signals away from the central nervous system, and are divided into motor cells, which bring information to skeletal muscles, and autonomic cells, which bring information to smooth muscles, cardiac muscles and gland cells. An additional category of neuron exists: interneurons are located exclusively in the central nervous system, and are connected to other neurons. [3,4.]

2.2.3 Neuronal Signal Transmission

In the next section the processes of signal reception and transmission in nerve cells will be discussed. When neurons are not receiving any input, the inside of the cell is more negative than the outside, with resting membrane potential around -75 mV [1]. Positive potassium and sodium ions are present both outside and inside the cell in different concentration, and the concentration of positive charges is higher in the extracellular medium. The tendency of positive ions to move towards the more negative side causes potential gradients across the cell membrane. Moreover, neuronal cells have a higher concentration of potassium ions and a lower concentration of sodium ion in respect to the extracellular medium. The difference in concentration, in conjunction with the higher permeability to potassium of the cell membrane, causes part of the potassium ions to cross the membrane and move to the extracellular medium. [1;3;4.]

The sodium-potassium pump, shown in figure, 3, is one of the main mechanisms involved in neuronal signal transmission, as it contributes to maintaining the sodium and potassium ions concentrations required for the resting potential state.

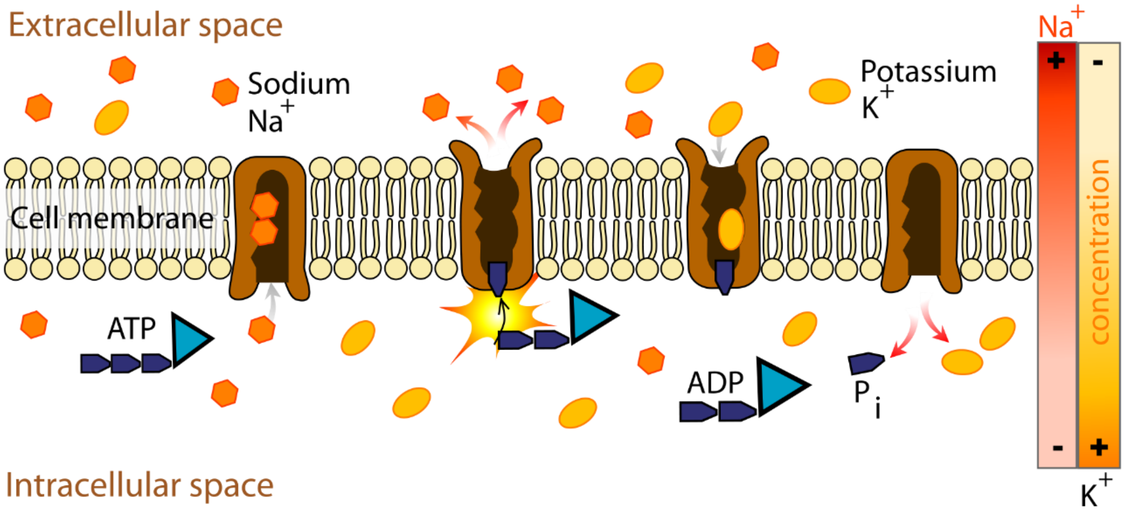


Figure 3. Sodium-potassium pump. Reprinted from Mariana Ruiz Villarreal. [6.]

The function of the pump consists of moving sodium ions located inside the neuron to the extracellular medium, and transferring potassium ions from the outside of the cell to the intracellular medium. When neurons are in an inactive state, the sodium-potassium pump binds with adenosine triphosphate (ATP) and with sodium ions. ATP hydrolysis provides energy that causes the shape of the pump to change into the active configuration, in which the bond of the pump with the sodium ions weakens. The sodium ions are then released outside the cell membrane, against charge and concentration potentials. Moreover, while the pump is in the activated state, potassium ions located outside the cell can bind to it, which causes the pump to revert the previous inactive configuration. The state change causes the potassium ions to be released inside the cell, countering the outward flow due to potassium concentration and cell membrane permeability. [3.]

When an input is received by a neuron through its synapses from other cells or from physical stimuli, a change in the membrane potential occurs. The magnitude and duration of the graded potential are dependent on the received input. Once the resulting potential on the axon hillock exceeds a specific threshold, an action potential sends a signal across the cell membrane through the neuron axon. The produced signal has high intensity and brief duration. While the magnitude of the graded potential depends on the triggering input, the magnitude of the action potential is fixed and dependent on the type of cell. Once the signal reaches the axon terminal, it will cross over to the target cell in form of neurotransmitter molecules which bind to receptors on the target cell and influence its behaviour. [3;4.]

Synaptic potentials are divided into excitatory postsynaptic potentials and inhibitory postsynaptic potentials, depending on the direction of the potential flow between the intracellular and extracellular medium. These potentials have longer duration than action potentials, and constitute the main source of extracellular current flow measured through electroencephalography, and, together with analogous ones occurring in other body systems, can be used to assess the status and help detecting issues in brain, heart and muscles.

Table 1 provides a list of the most common biosignal types and the corresponding sensor measurements. [1]

Table 1. Common biosignals. Data collected from Niedermeyer. [4].

Phenomena	**Biosignal measurement**
Electrical signals from heart activity	Electrocardiogram (ECG)
Surface signals from central nervous system activity	Electroencephalogram (EEG)
Electrical signals from muscle activity	Electromyogram (EMG)

As the electric signals generated by cell activity are typically characterized by small amplitudes, the instrumentation used needs to amplify the signals in order for analysis, recording, and display to be possible [3].

2.2.4 Brain Rhythms

Brain waves are mostly produced by both thalamus and cerebral cortex, in different ratios depending on the type of rhythm. Different types of waves may present overlapping frequency ranges. Nevertheless, waves with the same or similar frequency ranges do not necessarily share a similar origin and function. The three main types of neurons involved in the production of brain waves are thalamic neurons with cortical projections, thalamic reticular neurons, and cortical neurons. [4.]

Alpha waves have a frequency between 8 and 13 Hz. In normal EEG alpha waves occur mostly during relaxed wakefulness and eye closure, while they are weakened during eye opening and mental and visual effort. Alpha waves present symmetric voltage on both sides, and origin from the posterior area of the head. Mu waves present a frequency similar to that of alpha waves, between 8 to 10 Hz. Mu waves originate in the central part of the head, and are possibly asymmetric in voltage. Moreover, they relate to the sensorimotor cortex in the resting state, and are weakened by motor activity. The areas of origin and the frequencies of Alpha and Mu waves may overlap. In such a case, eye closing and opening may be used to identify the wave type in a clinical setting, as these phenomena block Alpha rhythms but not Mu. [1; 7.] Figure 4 shows a graphical display of some of the most common brain rhythms.

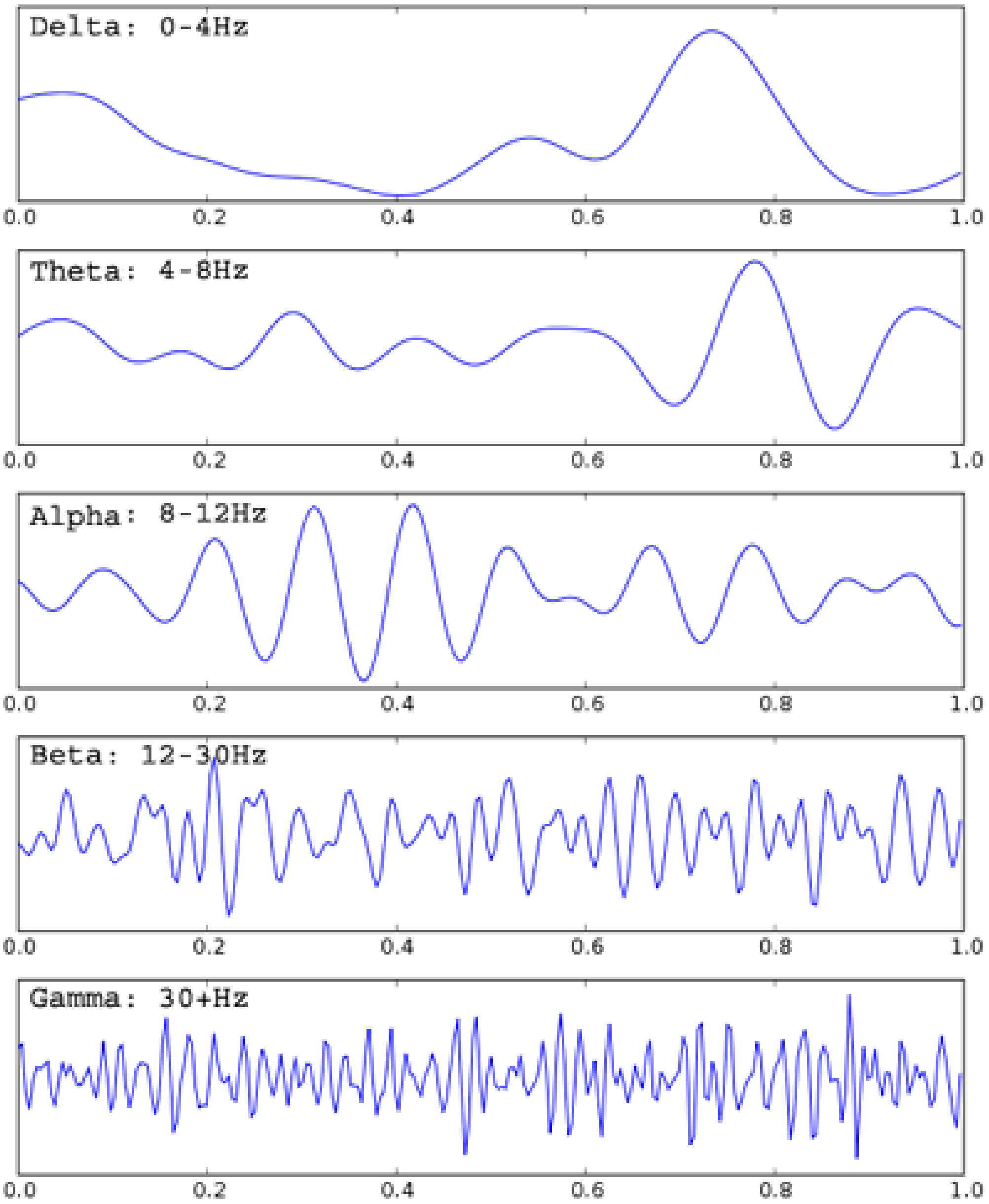


Figure 4. Brain rhythms. Reprinted from Wearable Sensing. [8.]

Beta waves are connected to light sleep, mental activation and drowsiness, and present a frequency starting from 13 Hz. Theta waves occur mostly in the frontal regions with a frequency of 4-7 Hz and various amplitudes. Lambda waves are similar to theta waves and are frequent in small children and sometimes in young adults in connection with complex visual stimuli. Delta waves present frequency smaller than 4 Hz, and occur during the waking state in young children and in the elderly, and during slow-wave sleep.

[1;7.] Table 2 shows the previously mentioned brain rhythms and their respective frequencies.

Table 2. Brain rhythms. Data collected from Prutchi (2005). [7.]

Name	Frequency
Delta	0.5 – 4 Hz
Theta	4 – 7.5 Hz
Alpha	7.5 – 13 Hz
Beta	13 – 38 Hz
Gamma	30 – 70 Hz

Spindles are non-rhythmic waves produced by the thalamus, and have a frequency oscillating between 7 and 14 Hz in periods of one to two seconds. In humans, spindles occur normally in stage 2 of Non-REM sleep. During stages 3 and 4 of Non-REM sleep, the occurrence of sleep spindles decreases, in connection with the onset of a slower rhythm with greater amplitude, named “delta”. Delta waves present a frequency of 1-4 Hz, and are divided into two groups, originating in the thalamus and in the cerebral cortex respectively. [4.]

2.3 Electroencephalography

The aim of electroencephalography (EEG) is to measure and record electrical brain activity. This activity is created by the exchange of signals between neurons, and is obtained by recording the changes over time in the difference of potential measured by couples of electrodes positioned on the scalp. EEG is a method of biopotential measurement, which has numerous applications in healthcare, such as the investigation of sleep disorders and epilepsy. [4.]

The main tools used in electroencephalography consist of electrodes accurately placed to perceive electrical activity, amplifiers and filters to obtain sufficiently accurate signals and handle noise and interferences, and devices to record and display the obtained information. Once the signal has been gathered through the electrodes, amplified, and handled through the use of the required filters, the data can be displayed in different ways depending on the type of instrumentation used and on the specific application [1]. Usually the data is displayed as waveforms. Routine electroencephalography recordings feature a paper speed of 30 mm/sec. A lower paper speed around 10mm/sec is used for encephalographic monitoring during sleep. [1]

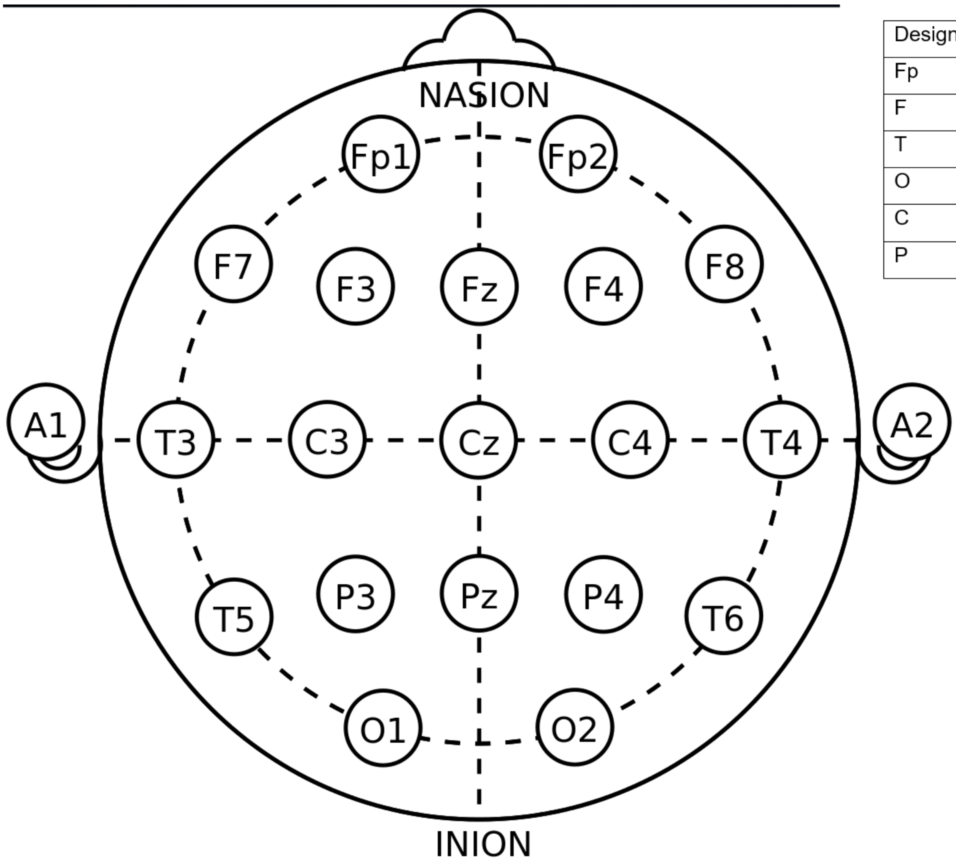
Each of the previously mentioned components will be analysed in further detail in the following sections. Additionally, the topic of electrical security for electroencephalography instrumentation will be outlined.

2.3.1 Electrodes

Electrodes are cup-shaped electrical conductors. The electrodes can be made of different metals, although most commonly silver electrodes coated with silver chloride (Ag/AgCl) are used [7]. Electrodes are individually connected to instrumentation with wires, and positioned on the scalp, in some cases with the use of a conductive gel or paste [1]. The electrodes employed in electroencephalography can be intracellular or surface electrodes, depending on the accuracy required. Surface electrodes are applied on the scalp, and measure the signals produced by groups of neurons. Intracellular or depth electrodes are used to detect potentials across individual cell membranes. In most cases the employment of surface electrodes is sufficient. [2;4.] For the scope of this paper, only surface electrodes will be considered.

In electroencephalography, electrodes are used in pairs, by recording the difference of the potentials measured by each electrode in the pair [2]. Electrodes can be disposable or reusable. The number of electrodes employed for measurements varies depending on the specific application. For example, in the case of infant electroencephalography, usually a smaller than average number of electrodes is needed. [4.]

In most applications, the location of each electrode is based on the international standard 10-20 system illustrated in figure 6. The figure displays the electrode positioning in the 10-20 system, in addition to the designation of each area of the scalp used in this system.



Designation	Position
Fp	Frontopolar
F	Frontal
T	Temporal
O	Occipital
C	Central
P	Parietal

Figure 5. Electrode positioning in 10-20 system. Reprinted from Wikipedia. [9.]

The 10-20 system uses physiological features as reference points to determine the placement areas. The individual electrodes are then placed at a distance between each other equal to either 10% or 20% of the total length of the skull either from the front to the back or from the left to the right side.

2.3.2 Amplifiers

Amplifiers are a part of electroencephalography instrumentation which amplifies the voltage measured by each electrode. The signals handled by devices for biopotential measurements have typically a small amplitude. This is particularly true in the case of electroencephalography, in which the analysed signals present an amplitude around 10µV to 100 µV for adults. For this reason, amplifiers connected to each pair of electrodes are a fundamental component of medical instrumentation aimed at recording physiological signals. In order for the signals to reach an amplitude suitable for data display and handling, a gain (ratio between input signal and output signal amplitude) between 100 and 100 000 is required. [7.]

The level of sensitivity in electroencephalography recordings varies depending on the instrumentation, and can be adjusted depending on the specific application. Therefore, the instrumentation requires a filter with adjustable amplitude range. Moreover, most electroencephalography devices feature the possibility of adjusting the sensitivity of all recording channels simultaneously. [4.]

2.3.3 Artifacts and Filters

All the signals obtained through encephalographic recordings, which are not produced by the physiological activity of the brain, are called artifacts. Artifacts can be physical or biological in origin, and their management must be implemented in electroencephalography instrumentation in order to obtain useful and reliable data. Artifact management is implemented on both hardware and software levels through the use of proper instrumentation and of artifact detection algorithms. [4;7.]

Biological artifacts are caused by physiological activity outside the brain, such as that of muscles and eye movement. Physical artifacts include those produced by movement of the patient, and electromagnetic and electric artifacts due to interference and noise from power supplies and the instrumentation itself. In particular, noise produced from amplifiers must be taken into account. [4.]

Filters are among the main tools for artifact management. Filters are needed in order to isolate relevant signals from unneeded ones. In some cases, unnecessary signals present a different frequency than relevant ones. For this reason, signal channels require adjustable low-pass and high-pass frequency filters. However, the frequency ranges of relevant and irrelevant signals can vary depending on the case, and may overlap. Moreover, it must be taken into account that filters typically cause a phase difference between instrumentation input and output. In addition to the low-pass and high-pass filters, which exclude artifact due to movement and unrelated physiological activity, a 60-Hz notch filter is usually available to exclude artifact from power lines [4.]

2.3.4 Safety

Medical instrumentation employed in electroencephalography recording is implemented as to guarantee electrical safety. Electrical safety includes the grounding and electrical insulation of the equipment. Insulation of the electroencephalography device is required for both input and output systems, particularly when the instrumentation is used in connection with additional medical equipment. [4.]

Official regulations exist concerning minimum safety requirements for medical instrumentation. Nevertheless, such regulations may vary depending on the country. This requires electroencephalography instrumentation to feature electrical safety systems conforming to the requirements of the region of employment. [4.]

3 Materials

3.1 OpenBCI 32-bit Board

The OpenBCI 32-bit board uses an Arduino-compatible PIC32MX250F128B microcontroller, and an ADS1299 low-noise, 8-channel, 24-bit analog-to-digital converter by Texas Instruments, optimized for electroencephalogram applications [9; 11]; however, it can be used to measure brain, heart, and muscle activity. In addition, the board includes a micro SD card slot and an accelerometer [10].

The board is powered by a 6V AA battery pack, and can communicate wirelessly with a computer via Bluetooth by using the included USB programmable dongle and RFduino Low Power Bluetooth radio modules. The dongle uses a serial connection on Mac devices and a COM connection for PC and Linux. The OpenBCI 32-bit board can either be used on its own, or together with the Daisy module expansion card, which increases the number of input channels from eight to sixteen. [10.]

Figures 6 A and B show the OpenBCI board and its components.

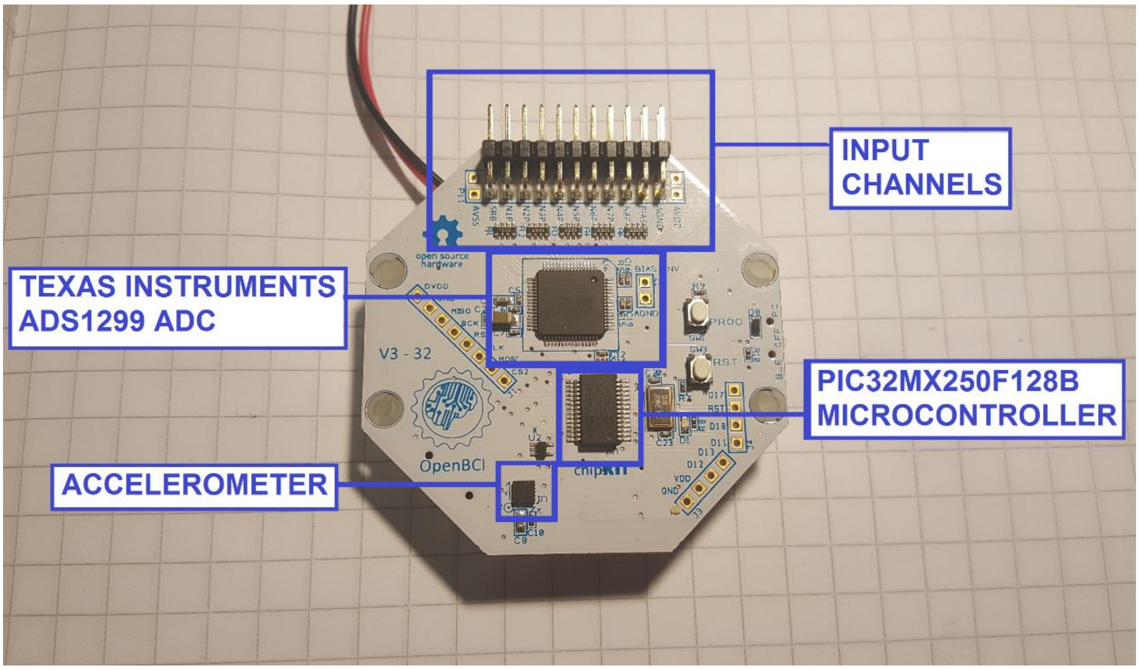

INPUT CHANNELS
TEXAS INSTRUMENTS ADS1299 ADC
PIC32MX250F128B MICROCONTROLLER
ACCELEROMETER

Figure 6a. OpenBCI board components (front).

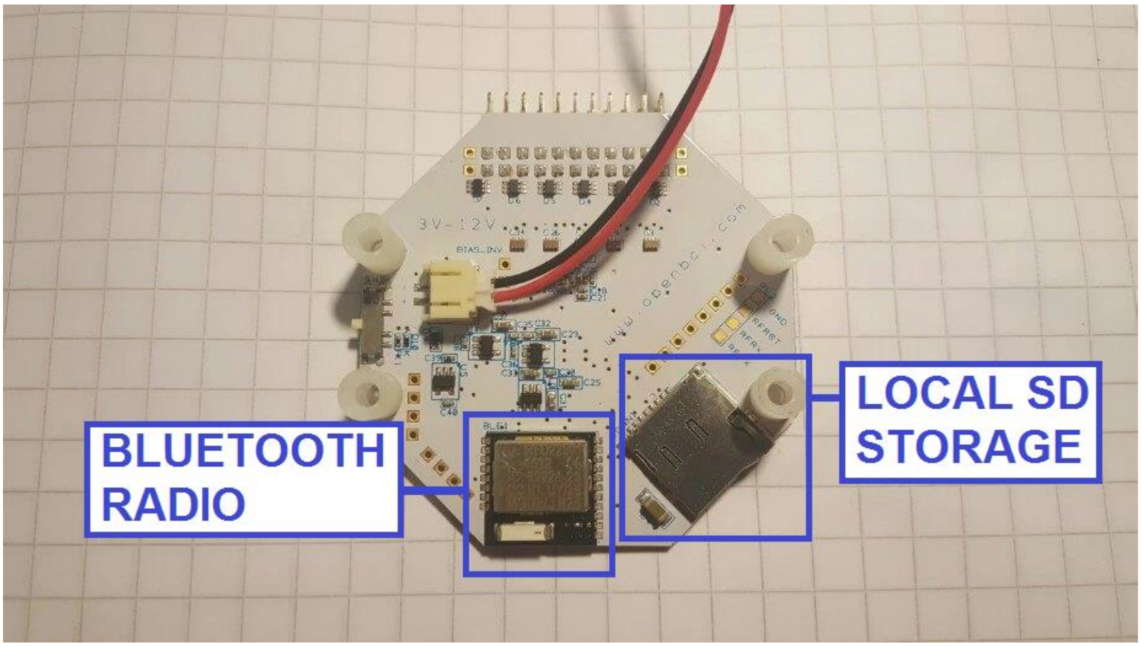


Figure 6b. OpenBCI board components (back).

Although OpenBCI offers an electrode starter kit, which includes 10 gold cup electrodes and a jar of conductive electrode paste, other types of electrodes can also be used, and converters to allow electrodes terminated in touch-proof design to be connected to the board are available [10].

3.2 OpenBCI Graphical User Interface

The OpenBCI graphical user interface (GUI) allows the user to start using the board to measure and display biopotential measurements with minimal required setup. The software can process data either in live mode from the board, or in playback mode from a file, and is compatible with both eight and sixteen channel systems. In live mode the data is fed to the computer wirelessly by using the USB dongle. As to the software side of the system, only the OpenBCI GUI and the FTDI drivers need to be installed on the computer.

The GUI includes several graphs to display the measurements in human readable format, in addition to various kinds of filters and utilities to adapt and optimize the system for the desired type of measurement. Moreover, the user can hide individual channels from the data stream user interface. However, this feature is available only when the bottom row of pins on the board is used to connect to the electrodes. Figure 6 shows a screen capture of the software streaming synthetic data in an 8-channel setting.

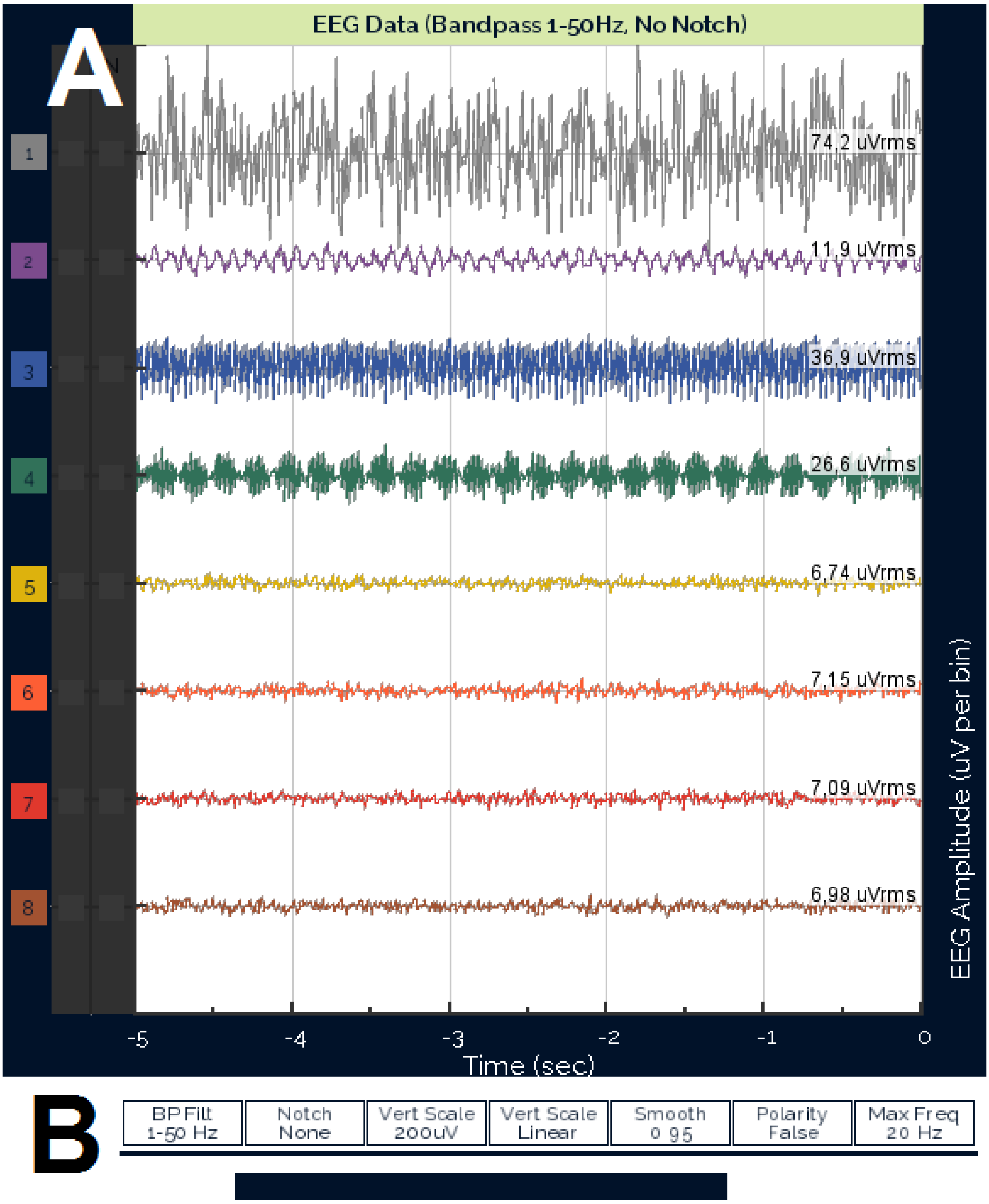
3.3 Other tools

The main software tools used during the project were the following:

- OpenBCI GUI
- OpenBCI 32-bit firmware
- OpenBCI_Python software library
- Spyder development environment
- Python 2.7.

As regards the hardware tools employed in the project, the OpenBCI 32-bit board was used together with one OpenBCI wireless USB dongle, 10 colour coded electrode cables, EEG electrodes 30mm x 24 mm, a 6V battery pack with four 1.5V, AA size alkaline batteries as power source for the board, and soldered pin connectors to adapt the available electrode cables to the pins on the board.

The OpenBCI GUI was used exclusively in the initial phase of the project. The GUI was employed to test and configure the connections between the electrodes and the board, between the board and the USB dongle, and between the USB dongle and the computer. Despite its usefulness during the preparation of the hardware, the GUI, of which some details are shown in figure 7, was found not to be needed to gather and handle EEG data from the board. Figure 7 displays the measured signals as diagrams (7A), and some of the available filters and settings (7B).



BP Filt	Notch	Vert Scale	Vert Scale	Smooth	Polarity	Max Freq
1-50 Hz	None	200uV	Linear	0 95	False	20 Hz

Figure 7. Details from the OpenBCI GUI.

During the following phases of the project, Spyder was used as the development environment. Spyder was chosen due to the presence of Python-specific testing and debugging features and of signal processing libraries and tools.

Python 2.7 was chosen as the programming language due to its focus on code readability, and the existence of a software library fitting with the scope of the project, OpenBCI_Python [12]. In addition to the OpenBCI library, the following other Python libraries, frameworks, and modules were used:

- json
- socket
- Queue
- Autobahn
- Twisted
- Tornado
- Handlebars.

The json module is included in the Python standard library, and was used to convert the EEG data samples to JSON strings to allow and ease transmission to the server. The socket module was initially used to implement the transmission of data to the server over TCP protocol. The Queue module was used to allow the gathering of samples from the board and the transmission of samples to the server to run simultaneously. Autobahn and Twisted were used to allow the transmission of the samples to the server with the WebSocket protocol. The WebSocket protocol was used to replace the socket module. This protocol runs over the TCP protocol, and allows the client script to stream the sample data to the server without requiring the opening of multiple connections. In the late part of the project, the previously mentioned libraries were replaced with Tornado. The reason behind the replacement is described in detail in chapter 4.2. In addition, the Handlebar library was used to manage front-end forms.

4 Results

4.1 Hardware Preparation

In order to prepare the recording system for use, the following process was used. First, the previously mentioned software (firmware and graphical user interface) was installed on a desktop computer. Second, the switch on the USB dongle was set to direct the signal to the RFduino GPIO6 pin, and the dongle was then connected to the computer. Third, the batteries were inserted to the battery pack, and the pack was then connected to the socket on the bottom side of the board. For the initial testing phase, two pin bars with four pins each were soldered together and plugged on the board. Fourth, the board was powered on by moving the power switch to the “PC” position. Finally, the OpenBCI GUI software was launched; the following settings were used for the configuration:

- Data source: LIVE
- Serial/COM port: COM3 (port for the USB dongle)
- Channel count: 8 channels
- Write to SD?: Do not write to SD.

The “Start System” button was then used to successfully establish a connection between the computer and the board.

During the initial testing phase, the connector bars were soldered in order to accommodate only six cables. The connections established between the electrodes and the board were based on the instructions provided by the OpenBCI website, with minimal changes to accommodate the previously mentioned differences in the pin setup.

Table 3 shows the electrodes used during this phase, the pin to which each of them was connected, and the function and position of each electrode.

Table 3. Test phase electrode connections

Cable	**Pin**	**Role**	**Position**
White	SRB2	Reference electrode	A1/left earlobe
Black	BIAS2	Ground and noise cancellation	A2/right earlobe
Purple	Bottom N2P	Input	Fp2/left side of forehead
Green	Bottom N4P	Reference electrode for channel 4	On muscle of right forearm
Blue	Top N4P	Input	Inner right wrist
Red	Bottom N7P	Input	O1/back of the head, left side

After connecting and positioning the electrodes, the live data stream was started from the OpenBCI GUI, and the unused channels 1, 3, 5, 6, and 8 were hidden from the graphs displayed by the software. Moreover, in the channel settings channel 4 was removed from BIAS and SRB2 in order for the ECG and EMG signals not to interfere with the EEG input on channels 2 and 7. In addition, the notch filter was switched to 50 Hz.

Figure 8 displays the soldered pins, and the connections between the electrodes and the board. The colours of the cables listed in table 3 are marked on the respective connectors in the top picture, and the connectors are linked to their respective pin in the bottom picture.

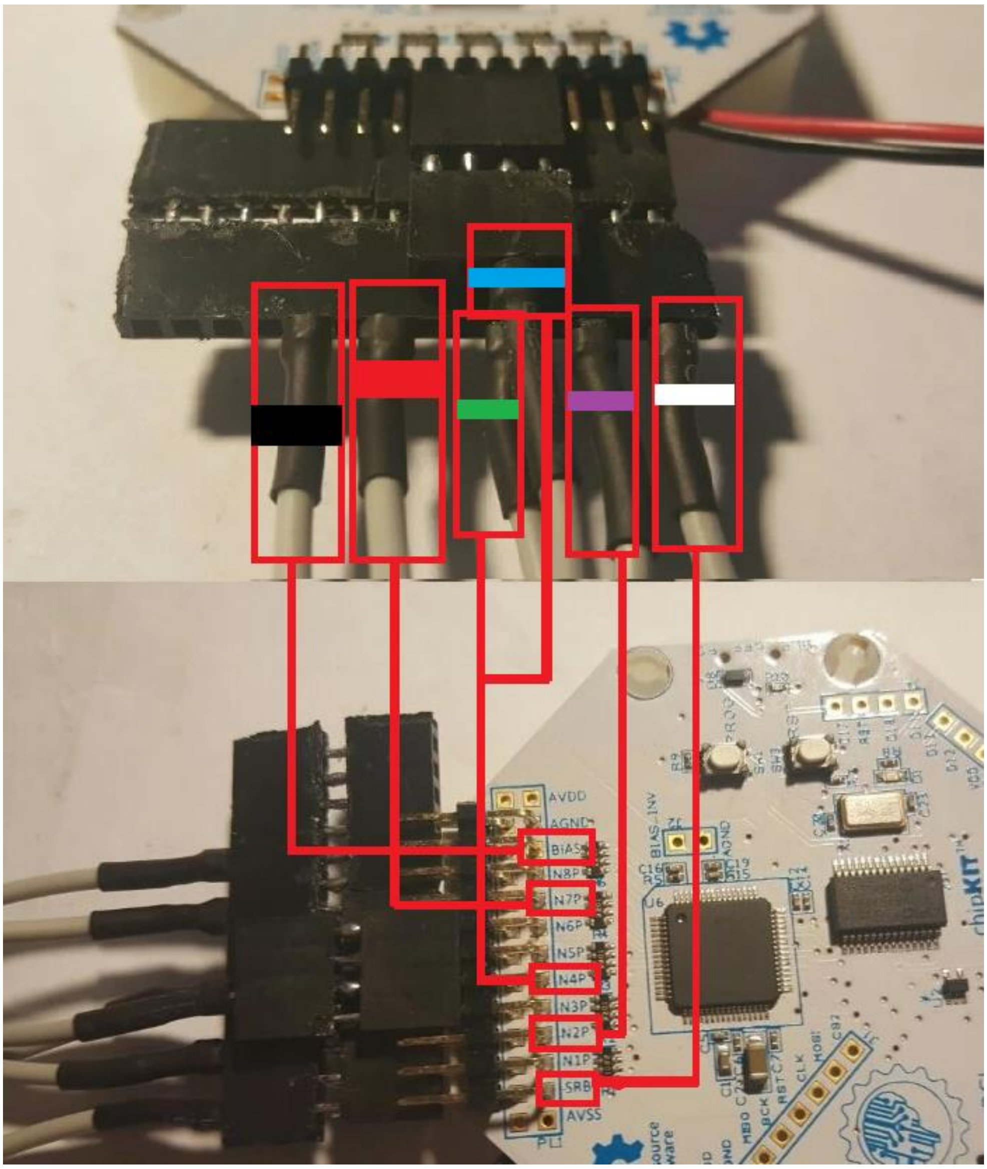


Figure 8. Electrode connections to board.

As the previously mentioned setup yielded satisfactory results, I proceeded to solder additional pin bars to accommodate a total of ten cables. As none of the signals of interest was recorded from ECG or EMG, the bars were applied exclusively to the pins on the bottom row of the board. The correct functioning of the soldered pins was then tested using the OpenBCI GUI. The signals obtained were sufficiently precise and without artifacts, with the exception of the issues caused by the presence of hair between the electrode and the scalp. The situation was improved by increasing the electrode stability using medical tape on each electrode and a fabric band around the head.

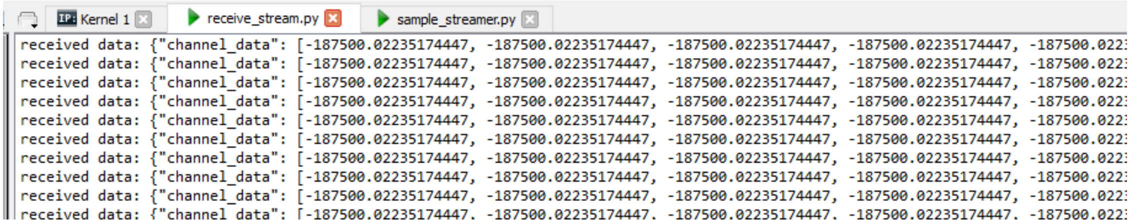
4.2 Data Transmission to the Server

The following phase of the project focused on fetching the raw EEG data recorded from the board and transmitting it to a server. For purpose of investigating the format of the data produced by the board, initially the scripts and plugins included in the OpenBCI_Python library were used. Specifically, the use of the “print” plugin allowed me to use the command line interface to establish a connection between the board and the computer, and to print a stream of EEG data samples. Each sample consisted of an array of the voltage from each of the eight channels on the board in microvolts, in addition to a progressive integer identification number in order to allow the detection of skipped packets. However, after checking the available plugins, due to the lack of the functionalities needed by the project, I decided to create my own scripts to handle the data and send it to the server.

When planning the implementation of the scripts, I decided to send the data to the server over the Transmission Control Protocol (TCP). The TCP was chosen rather than User Datagram Protocol (UDP) in order to ensure a reliable data stream. Moreover, I designed two separate scripts for the client and the server side respectively. From the OpenBCI_Python library I kept only the file containing the implementation of the OpenBCIBoard and the OpenBCISample classes, in order to ease the processes of connecting to the board and of managing the EEG data samples respectively.

As illustrated in Appendix 1, I imported the file in the client side script, configured the connection to the local host and the one between the USB dongle and the computer, and created an instance of the OpenBCIBoard class. After starting the stream from the board, I stored each sample as Python dictionary containing the sample identification number and the array of channels voltage values, and then converted the dictionary to JSON string. Finally, after successfully testing the process of fetching the data from the board by printing the resulting samples, I submitted each string to the server. In the server side script, I configured the TCP connection, and implemented the handling and printing of the data transmitted by the client.

Figure 9 shows the first data stream received and printed by the server.



```
received data: {"channel_data": [-187500.02235174447, -187500.02235174447, -187500.02235174447, -187500.02235174447, -187500.022
received data: {"channel_data": [-187500.02235174447, -187500.02235174447, -187500.02235174447, -187500.02235174447, -187500.022
received data: {"channel_data": [-187500.02235174447, -187500.02235174447, -187500.02235174447, -187500.02235174447, -187500.022
received data: {"channel_data": [-187500.02235174447, -187500.02235174447, -187500.02235174447, -187500.02235174447, -187500.022
received data: {"channel_data": [-187500.02235174447, -187500.02235174447, -187500.02235174447, -187500.02235174447, -187500.022
received data: {"channel_data": [-187500.02235174447, -187500.02235174447, -187500.02235174447, -187500.02235174447, -187500.022
received data: {"channel_data": [-187500.02235174447, -187500.02235174447, -187500.02235174447, -187500.02235174447, -187500.022
received data: {"channel_data": [-187500.02235174447, -187500.02235174447, -187500.02235174447, -187500.02235174447, -187500.022
received data: {"channel_data": [-187500.02235174447, -187500.02235174447, -187500.02235174447, -187500.02235174447, -187500.022
received data: {"channel_data": [-187500.02235174447, -187500.02235174447, -187500.02235174447, -187500.02235174447, -187500.022
```

Figure 9. First data stream to server.

In order to implement the TCP connection, both scripts used the socket Python module. During the first test, the data was received successfully from the server. However, the client script opened a new socket connection for each sample, creating unnecessary traffic. Therefore, the socket variable was changed to global, allowing the socket connection to be opened only once during the stream. On the server side, several values were tested for the buffer size, and 250 was found to fit the format of the samples. The server script was implemented as to listen for data from the client, and to close the connection if no data is received. The server script received the data successfully and in correct format. However, due to the use of TCP on its own, and the consequent lack of a defined end of transmission character (EOT), the stream was automatically interrupted shortly after starting the connection.

To better fit the transmission of the sample as a stream, I decided to use the WebSocket protocol to send data. To implement this, Autobahn [13] and Twisted [14] libraries were imported in both client and server scripts. After implementing the transmission with the new protocol, the connection to the server was established successfully. However, no data was received by the server. The issue was caused by the sample reading, which blocked the client script, preventing the data from being sent. The problem was fixed by moving the sample reading and the sample transmission in separate threads running simultaneously. The communication between threads was handled using the Queue module. After the change, the samples were received successfully by the server.

..

After an additional testing session for both the server and the client scripts, I decided to switch to the Tornado library [15] for the server script. The change was motivated by the fact that Tornado offered in a single library the features from Autobahn, Twisted, and

Http required in the project. Therefore, I proceeded to rewrite the server script in order to use only the Tornado library, while maintaining the same functionalities. As the main functionalities of both server and client scripts had been implemented, I proceeded to design and develop the database for the project.

4.3 Database and API

The following section will describe the design and implementation of the database, and of the API handling the stored data. To implement the database, I started by designing an ER model. The preliminary model, shown in figure 10, included only two entities, Sessions and Samples.

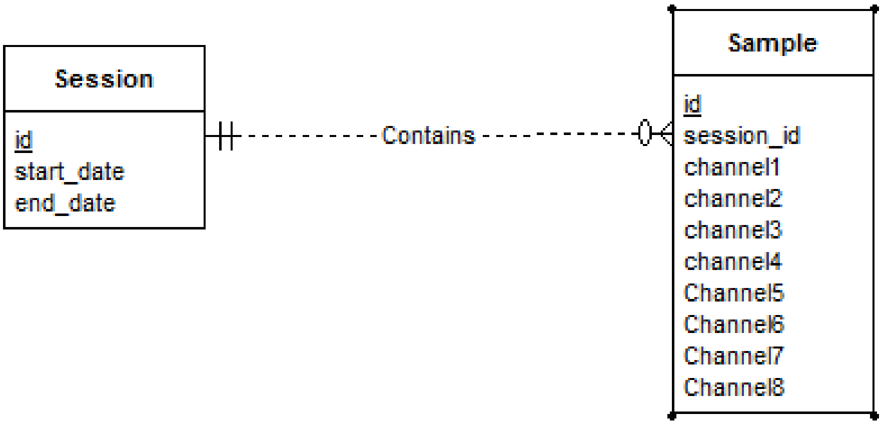


Figure 10. First ER model.

I then created a new file for the purpose of gathering the functionalities related to the database, and the definition of the Database class. In this file implemented a function to create the Sessions and Sample tables, and functions to store in the database instances of the session and sample objects. The creation a session record was implemented in relation to the opening of a WebSocket connection, while the creation of a sample record was triggered by the arrival of a new message to the server. Successively, the sample time column in the Samples table was replaced with the order number of the individual sample in reference to the Session containing it. The numbering of the samples and the detection of skipped packets was implemented based on the packet ids sent by the board with each sample.

After testing the storage of information to the database, handlers for the listing of sessions and samples records were created. These functions were implemented through functions to fetch all existing records in the sessions table, and all the records in the sample table with the given session id. The results were passed from the database to the handlers in the form of lists. The sets of result were then converted to dictionaries. The “get” function was then implemented to display the fetched data in the browser.

4.4 Data presentation and database extension

The following section will outline the process of implementation of the front-end side of the project, in addition to the design of an extended version of the database. To contain the static HTML and JavaScript files, a new sub-directory was created in the main server directory. The path to the static file folder was then defined in the “make_app” function of the “__main__” server-side file. The built-in Tornado class StaticFileHandler was then used to allow the application to be accessed from browser. Successively, the file “index.html” was created to contain the implementation of the front-end layout and form templates, and for the inclusion of external libraries for scripting and styling. Moreover, the file “scripts.js” was added to implement the front-end functionalities and the transmission of data from and to the server.

During this phase, major changes were applied to the database structure. The Session table was extended to store data about the sample rate and measurement resolution of each session. Moreover, in order to better fit the educational purpose of the project, three new entities were added: course, patient, and instructor. Relationships were then implemented between each course and the instructor responsible for it, between each session and the course in which the measurements were involved, and between each session and the patient from which the measurements were taken.

Figure 11 displays the extended ER model, which includes the added classes and attributes.

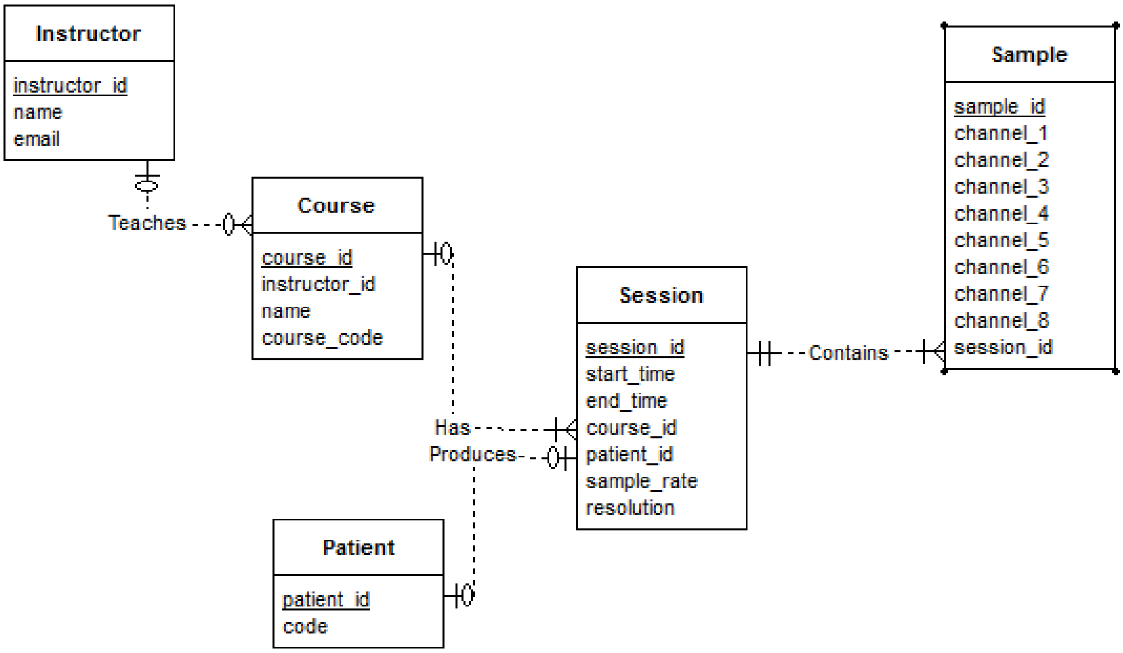
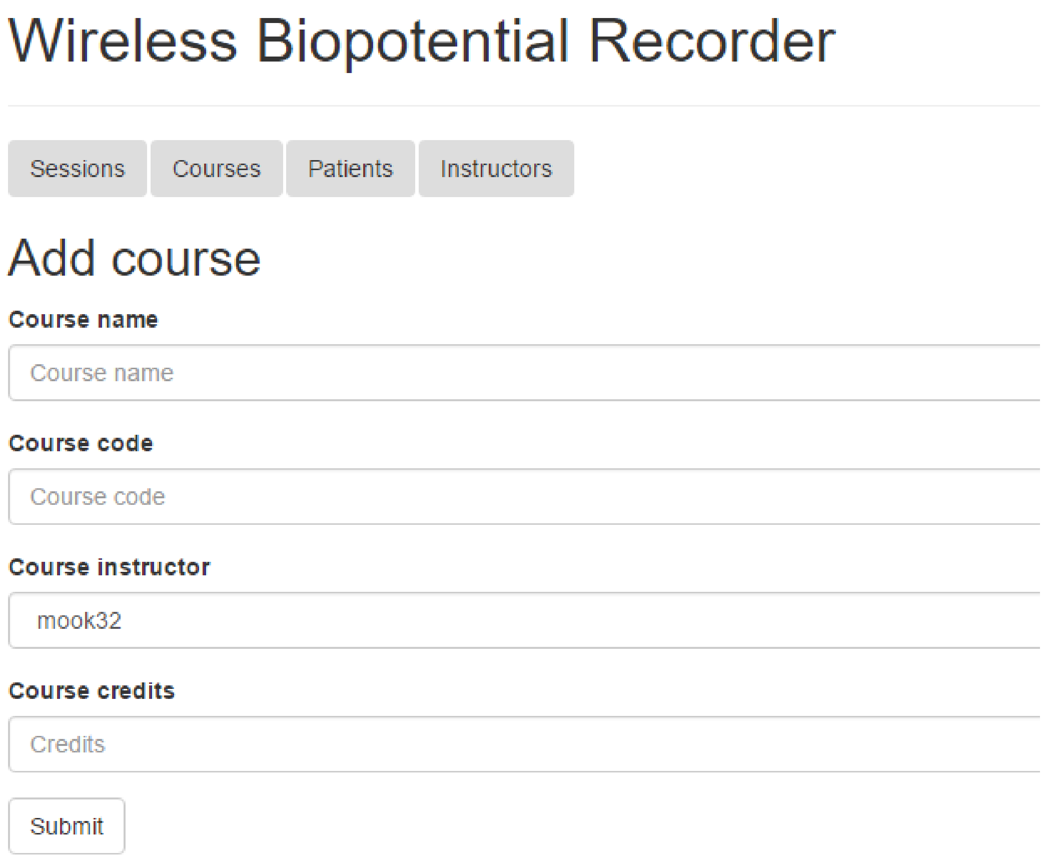


Figure 11. Extended ER model.

After applying the changes to the project database, the scripts for the interface were implemented. First, one function for each entity was implemented in the database handler class to fetch all the records stored in each table for the purpose of listing the records to the user. Second, for each entity functions were implemented in the script file to fetch the records from the server and display them as lists in tables. Third, a navigation bar was added to the layout to allow the user to move between the available entity records.

Forms were added to the application, in order to allow users to create and modify instances of courses, patients, instructors, and sessions. Handlebars [16] was used to implement forms based on templates defined in the HTML file. A script was then added to send the data gathered through the form to the server using the POST method.

On the back-end, each class was extended with a post function to receive the form data, parse it and submit it to the database. Finally, a function was added to the database class to create new database records based on the form data. Figure 12 displays one of the implemented forms, used to create a new course record. Moreover, the figure displays the previously mentioned navigation bar.



Wireless Biopotential Recorder

Add course

Course name

Course name

Course code

Course code

Course instructor

mook32

Course credits

Credits

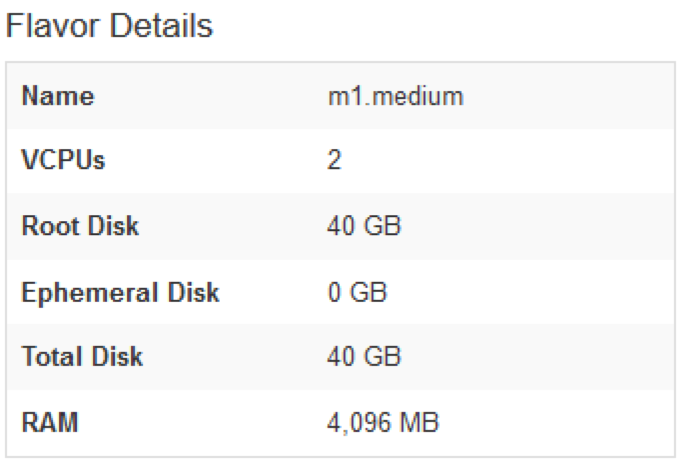
Figure 12. ”Add course” form.

After implementing the creation of a new instance for courses, instructors, and samples, the functionality of modifying existing records of each entity type were created. To achieve this, a new class was created for each entity type, in order to handle individual records. Then, each form template was modified to use data from existing records, if present, as input value. Finally, new scripts were added to create links to the modification form for each record, the form themselves, and the logic to serve record data to the form and submit form inputs to the server. After the newly created functionalities were tested, the display of sample data from an individual session as a line graph was developed.

Initially the graph creation was achieved using the Chart.js library. However, such library was found to scale badly when a high number of points is plotted. Moreover, the configuration of the graphs and of the canvas element containing them was complex and not flexible. Therefore, the Chart library was replaced with the Plotly library for JavaScript. The sample listing for an individual session was then implemented as to display an individual line chart for each of the board input channels. The Plotly graph creation function receives as arguments the page element in which the graph has to be displayed, the data to be plotted on the graph, and a variable containing the graph layout configuration data. The result displayed on the page is a line chart which allows the users to easily zoom on a section of the graph of their choice, and to freely navigate the length of the chart.

4.5 Application transfer to Metropolia cloud

After I implemented and tested all the required functionalities locally, the application was transferred to its final location the Metropolia cloud. First, I created a virtual machine (VM) on the cloud server to run and manage the application. The VM was configured to run Debian 8, and to have 40 GB of disk space, to allow sufficient storage for the sample database. Second, I configured the VM to allow the reception of the sample data over TCP. Third, I transferred the application files to the VM. Figure 13 displays the summary of the virtual machine technical specification.



Flavor Details

Name	m1.medium
VCPUs	2
Root Disk	40 GB
Ephemeral Disk	0 GB
Total Disk	40 GB
RAM	4,096 MB

Figure 13. VM details.

Before running the application in its new location, some changes were applied to the sample server and database scripts. The changes allowed the user to configure basic features of the application, such as the path where the database file is stored, from the command line without the need of editing the code. Moreover, logging functionalities were added to the server script, in order to clearly display the start and end of a sample reception session on the command line interface.

For added security, a new basic user named “samplesserver” was created on the VM. The application files were then transferred to /home/samplesserver, and the application was set up to be run by samplesserver user, in order to have the application run by a user with no root rights. Moreover, nginx [17] was used as reverse proxy server to handle communication between the application server and the internet. Figure 14 illustrates the structure of a reverse proxy server.

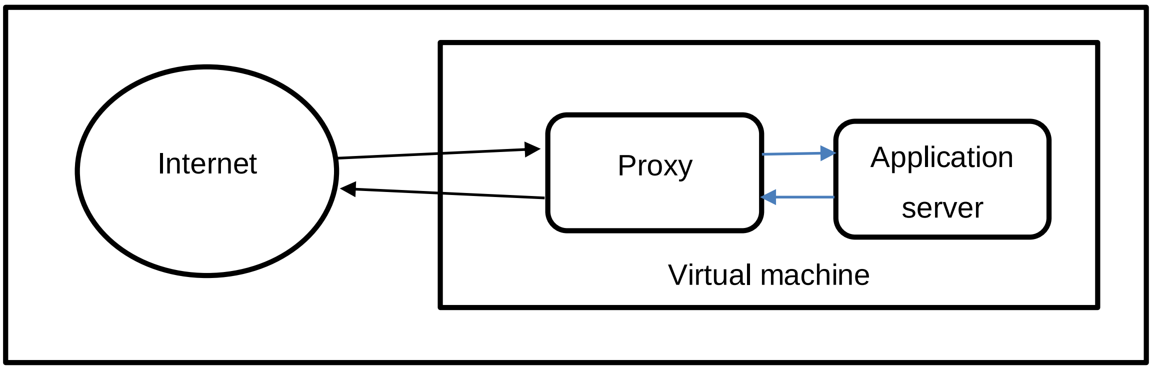


Figure 14. Reverse proxy diagram.

Finally, systemd was used as to manage the processes for both nginx and the application. This allows the application to be run automatically when the VM is started, and to be restarted in case of shutdown of the server, the virtual machine, or of the application itself.

The application was tested in order to ensure that the newly implemented functionalities and configuration worked correctly. Testing of the automated launch of the application, of the proxy server, and of the logging functionalities yielded positive results. However, due to the transfer of the application to the Metropolia private cloud, the application was accessible exclusively from the Metropolia network. Nevertheless, as the application was implemented for educational purposes as part of Metropolia courses, this limitation was considered not to be a major issue.

5 Discussion

The result of the project was an application to be used as part of Metropolia courses. The application records biopotential measurements using an OpenBCI board, transmits them wirelessly to a client script, and submits the recorded data to the Metropolia cloud. The samples are then stored in a database, and can be displayed and managed through a browser. In order to improve the use of the application as an educational tool, and to provide a more organized system to users, additional entities related to the sample sessions were added to the database and to the interface. These entities include courses, instructors, and patients. Using a browser while connected to the Metropolia network, the user can view information about the instances of each entity stored in the database. Moreover, the measurements from each session can be displayed graphically as line diagrams. Figure 15 illustrates the structure of the whole application.

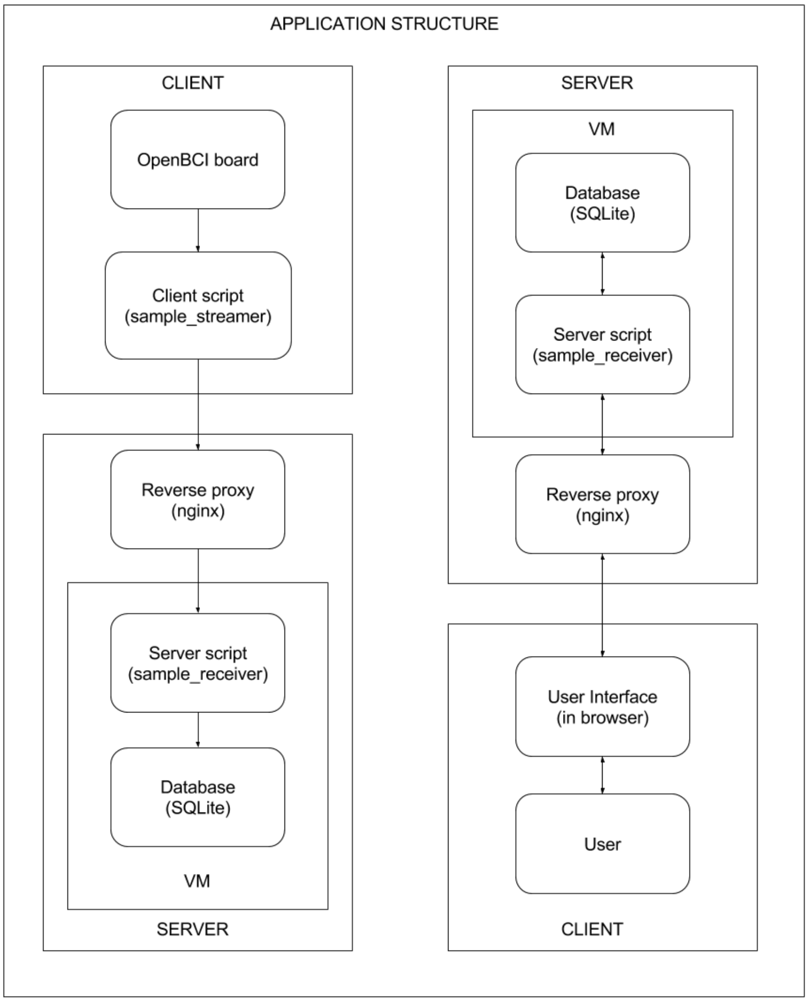
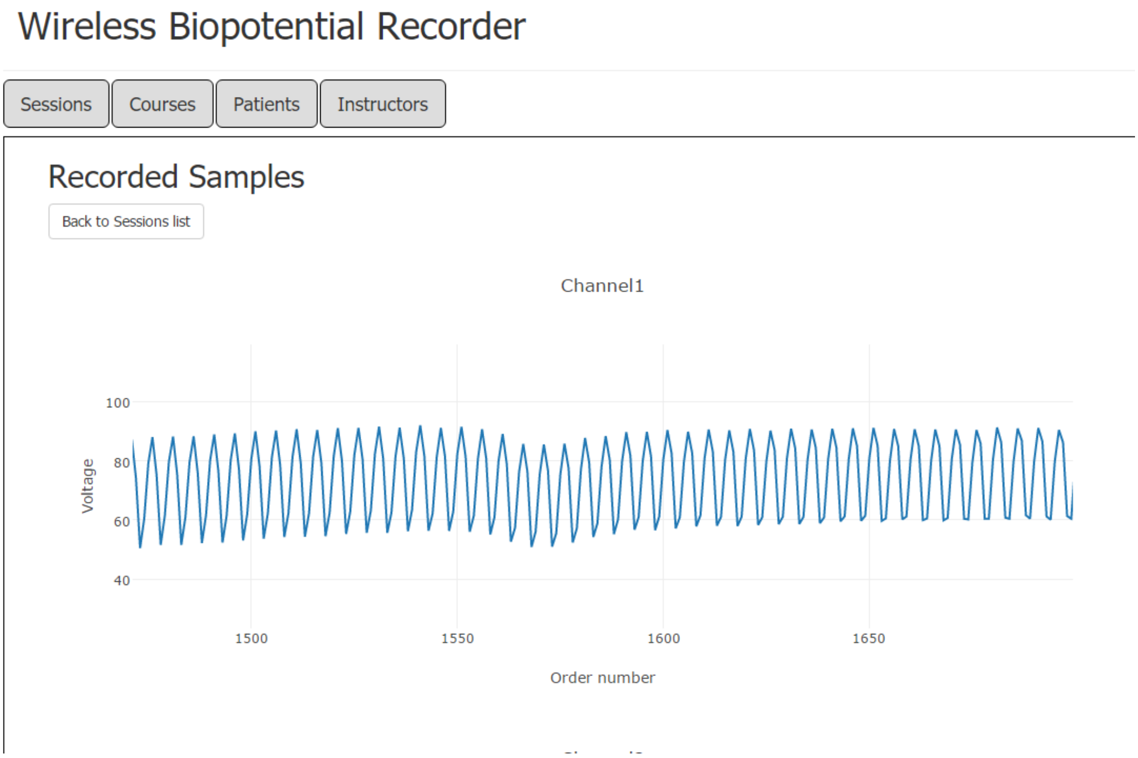


Figure 15. Application structure.

While viewing the sample graphs from an individual session, users can zoom in and out on any section of each diagram, in addition to being able to scroll through the graph using the mouse, and to download the graph in PNG format. In addition, the same data can be accessed in JSON format using the application API. Figure 14 shows an example of the samples measured on the first channel of an individual session displayed as graph.



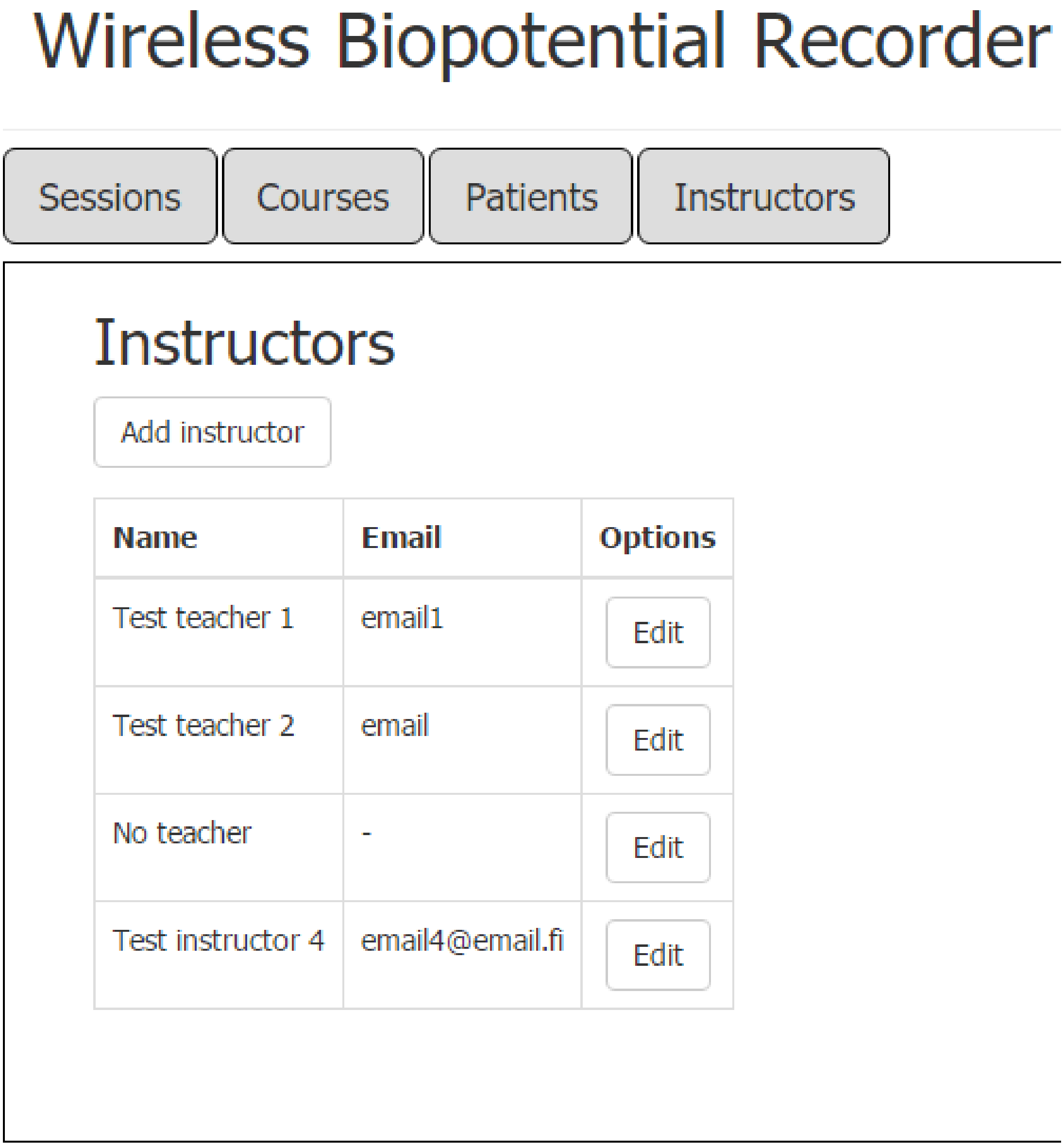
Wireless Biopotential Recorder

Recorded Samples

Figure 16. Samples displayed as graph.

However, due to time constraints, the application presents some limitations. As can be seen in Appendix 2, the client script does not include configuration features in the user interface. Therefore, to modify the measurement settings such as sample frequency, the user is required to modify the script code directly. Moreover, the user interface accessed with browser is extremely basic, and presents limited features. In addition, both the interface forms and the application database feature insufficient and generic data validation. Finally, both the client script and the browser user interface can be used only within the Metropolia network.

Figure 17 shows an example of the presentation of the list of instructors currently stored in the database.



Wireless Biopotential Recorder

Instructors

Name	Email	Options
Test teacher 1	email1	Edit
Test teacher 2	email	Edit
No teacher	-	Edit
Test instructor 4	email4@email.fi	Edit

Figure 17. Instructor list.

As a side note, due to time and skill limitations, and to library and environment compatibility issues, the application was implemented using Python 2, rather than Python 3. The final version of Python 2 was released in 2010, whereas Python 3 is under ongoing development. Therefore, it may be beneficial to consider a migration of the application to Python 3 in the future.

6 Conclusion

The goal of the project was to implement a wireless biopotential recorder for educational purposes. The developed application allows the transmission of EEG sample data to the Metropolia cloud. The samples were recorded using a 32bit OpenBCI board, and sent to the client script via Bluetooth. The data is then sent to the server, where it is stored in a database. Moreover, the data can be accessed and managed through a browser. In addition, an interface organizing sample sessions in relation to courses and patients was created.

The main purpose of the project was achieved successfully, allowing the application to be used as an educational tool in Metropolia courses. Due to the location of the application on the Metropolia cloud, the application itself is accessible only from within the Metropolia network. Moreover, despite the general purpose of the application being implemented, several improvements and additions are possible. For example, the functionalities for displaying the stored sample data could be expanded. Moreover, the forms in the user interface could be provided with data validation features, in order to ensure that data in the correct format is sent to the server. In addition, a more user-friendly and configurable client interface could be implemented. Finally, the functionality to display the samples on a graph could be improved.

The OpenBCI 32bit board allows the user to measure biopotentials from brain, heart and muscle activity in a relatively inexpensive and simple way. As several compatible pieces of software are available in open-source format, and as the board is delivered with a pre-installed firmware, using the basic features of the platform does not require a particularly wide understanding of hardware functioning. Nevertheless, the client script can be modified to allow the configuration of measurements settings such as filtering and sampling frequency, and to fit the use with different versions and modules of the OpenBCI hardware. Therefore, the OpenBCI platform, paired with the application developed in this project, can be a useful tool for understanding and elaborating EEG, ECG and EMG measurements for both inexperienced users and more experienced developers.

References

1. Tatum WO, Husain A, Bembadis S. Handbook of EEG interpretation. New York: Demos Medical Publishing; 2014.

2. Theis, FJ, Meyer-Bäse A. Biomedical Signal Analysis. Cambridge, US: MIT Press;2010.

3. Brodal P. The central nervous system: structure and function. Oxford University Press; 2004.

4. Niedermeyer, E. Electroencephalography: Basic Principles, Clinical Applications, and Related Fields. LWW, Philadelphia, PA, USA.

5. OpenStax College. Anatomy & Physiology [online]. Connexion Web site; Jun 19, 2013.
   URL: http://cnx.org/content/col11496/1.6/
   Accessed 14 September 2016.

6. Ruiz Villarreal, M. Sodium-potassium pump [online]. Wikimedia Commons.
   URL: https://commons.wikimedia.org/wiki/File%3AScheme_sodium-potassium_pump-en.svg
   Accessed 14 September 2016.

7. Prutchi D, Norris M. Design and Development of Medical Electronic Instrumentation. Hoboken, US: Wiley-Interscience;2005.

8. Wearable sensing. Brain rhythms [online]. Wearable Sensing.
   URL: http://www.wearablesensing.com/images/EEG.png
   Accessed 14 September 2016.

9. Wikipedia. 10-20 system [online]. Wikipedia.
   URL: https://en.wikipedia.org/wiki/10-20_system_(EEG)
   Accessed 14 September 2016.

10. Texas Instruments. ADS1299 Datasheet: Low-Noise, 8-Channel, 24-Bit Analog Front-End for Biopotential Measurements [online]. Texas Instruments Incorporated, Texas; August 2012.
    URL: http://www.ti.com/lit/ds/symlink/ads1299.pdf
    Accessed 17 February 2016.

11. OpenBCI. OPENBCI 32bit Board [online]. OpenBCI; 2015.
    URL: http://openbci.com/
    Accessed 17 February 2016.

12. OpenBCI. OPENBCI HARDWARE DOCUMENTATION [online]. OpenBCI; 18 October 2015.
    URL: http://docs.openbci.com/hardware/01-OpenBCI_Hardware#openbci-hardware-documentation-openbci-32bit-board-32bit-board-specs
    Accessed 7 February 2016.

13. OpenBCI. OpenBCI_Python software library [online]. OpenBCI;2016.
    URL: https://github.com/OpenBCI/OpenBCI_Python
    Accessed 7 August 2016.

14. Tavendo GmbH. Autobahn|Python [Online]. Tavendo GmbH; 2016.
    URL: http://autobahn.ws/python/
    Accessed 16 August 2016.

15. Twisted Matrix Labs. What is Twisted? [online]. Twisted Matrix Labs; 2016.
    URL: https://twistedmatrix.com/trac
    Accessed 16 August 2016.

16. The Tornado Authors. Tornado [online]. The Tornado Authors, 2016.
    URL: http://www.tornadoweb.org/en/stable/
    Accessed 16 August 2016.

17. Katz Y. Handlebars [online]. Github; 2016.
    URL: https://github.com/wycats/handlebars.js
    Accessed 16 August 2016.

18. Nginx. Nginx: about [online]. Nginx, Inc; 2016.
URL: https://nginx.org/en/
Accessed 29 August 2016.

Appendix 1: Initial implementation of data transmission over TCP

Client side – sample_streamer.py

```
import socket
import open_bci_v3 as bci
import json

sock = None



# configure board and server connection
def start_streaming():
# USB dongle settings
port = "COM3"
baud = 115200
# server target
tcp_ip = "127.0.0.1"
tcp_port = 1912
# open connection to server
global sock
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect((tcp_ip, tcp_port))
# connect to board
board = bci.OpenBCIBoard(port=port, baud=baud, filter_data=False)
board.start_streaming(process_sample)



# format sample as json string
def process_sample(sample):
dict_sample = {
"id": sample.id,
"channel_data": sample.channel_data
}
```

```
json_sample = json.dumps(dict_sample)
send_sample(json_sample)


# send sample data to server
def send_sample(json_sample):

sock.send(json_sample)


if __name__ == "__main__":
start_streaming()
```

Server side – receive_stream.py

```
import socket


tcp_ip = "127.0.0.1"
tcp_port = 1912
buffer_size = 250


sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.bind((tcp_ip, tcp_port))
sock.listen(1)

conn, addr = sock.accept()
print "Connection address:", addr
while 1:
    data = conn.recv(buffer_size)
    if not data: break
    print "received data:", data
    conn.send(data)
conn.close()
```

Appendix 2: Final implementation of client script

```
import open_bci_v3 as bci
import json
from Queue import Queue
from twisted.internet.defer import inlineCallbacks,
          CancelledError
import sys
import os

from twisted.python import log
from twisted.internet import reactor, threads
from autobahn.twisted.websocket import WebSocketClientProtocol, \
    WebSocketClientFactory


queue = Queue()

order_number = 0
last_packet_id = None


def read_samples():
     # USB dongle settings
    port = "COM3"
    baud = 115200
    # connect to board
    board = bci.OpenBCIBoard(
                    port=port,
                    baud=baud,
                    filter_data=False
                    )
    board.start_streaming(process_sample)


def process_sample(sample):
    """ format sample as json string and add it to queue """
    global last_packet_id
    global order_number
    if last_packet_id is not None:
        # it is assumed that no more than 255 packets are skipped
        difference = (sample.id - last_packet_id) % 256
        order_number = order_number + difference
    last_packet_id = sample.id
    dict_sample = {
        "channel_data": sample.channel_data,
        "order_number": order_number
    }
    json_sample = json.dumps(dict_sample)
    queue.put(json_sample)


def wait_for_sample():
    """ defer queue reading """
    d = threads.deferToThread(read_queue)
    #reactor.callLater(0, read_queue, d.callback)
    timeout = reactor.callLater(5, d.cancel)
    def cancel_timeout(result):
        if timeout.active():
            timeout.cancel()
        return result
    d.addBoth(cancel_timeout)
    return d


def read_queue():
    ### get sample values stored in queue and give results to
Deferred ###
    sample = queue.get()
    return sample
```

```
class MyClientProtocol(WebSocketClientProtocol):

    def onConnect(self, response):
        print("Server connected: {0}".format(response.peer))

    @inlineCallbacks
    def onOpen(self):
        print("WebSocket connection open.")
        while True:
            try:
                sample = yield wait_for_sample()
                self.sendMessage(sample)
            except CancelledError:
                self.sendClose()

    def onClose(self, wasClean, code, reason):
        print("WebSocket connection closed: {0}".format(reason))
        os._exit(0)

if __name__ == '__main__':

    log.startLogging(sys.stdout)

    factory = WebSocketClientFactory(
                    "ws://127.0.0.1:1912/api/samples"
                    )
    factory.protocol = MyClientProtocol
    reactor.connectTCP("127.0.0.1", 1912, factory)
    reactor.callInThread(read_samples)
    reactor.run()
```

Links to the complete code of the website

The code of this project will be available for 3 years, starting from September 2016 until September 2019. The code contains the version of the product as it was when this thesis was completed. However, the actual code used as educational tool at Metropolia UAS could be modified.

Public Github repository on Tatiana Tassi/Squonkbadger personal account:
URL: https://github.com/squonkbadger/thesis