DIPARTIMENTO DI INGEGNERIA INFORMATICA, MODELLISTICA, ELETTRONICA E SISTEMISTICA

Master Degree in Telecommunication Engineering: Smart Sensing, Computing and Networking

Smart Distributed Health Monitoring System

Relatore:

Prof.
De Rango F.

Candidato:

Vacca Andrea
Mat. 252416

Mohamed Mahmoud
Mat. 252048

Thiam Papa
Mat. 252113

ANNO ACCADEMICO 23/24

Contents

Chapter 1

Introduction

This project has as its application context what might be a nursing home for the elderly where there is a need to constantly monitor the various vital parameters of the residents over time. This translates into constant monitoring of these variables on a continuous basis.

The data obtained are then transferred, analyzed, and processed in order to contextualize the data so as to understand whether the data obtained indicate a dangerous situation, or is it just a normal condition of the patient due to the physical context.

A key detail of this project is to increase the frequency with which the devices capture data so that we have as much data as possible in case of the first symptoms of a possible illness.

In addition, the data once analyzed will be saved within a database so that we can have the account over time of the subject's parameters in question. An additional feature of the project allows the patient's trusted staff to be alerted in case the system detects a potentially problematic situation.

The project is based on the fact that a pre-programmed ESP32 card allows it to pick up data sent by a smartwatch worn by the patient. The device can provide various information about the patient's health status, starting with steps, kilocalories consumed, heartbeats and blood pressure. After that, the data passes to an ESP8266 card that analyzes and processes the data and passes through Amazon AWS to be saved in a noSQL database.

Chapter 2

Project Description

The operation of the project is based on two fundamental conditions that depend on the position of the subject in question with respect to the master device.

This means that if the subject is located inside the same building as the master device (and therefore presumably the two devices are connected to the same network), then the communication protocol between the two devices will be based on WebSocket technology since the devices belong to the same network.

Instead, when the subject is not located in the same building as the master device, and presumably the two devices belong to two different networks, another type of communication based on mqtt brokers is established. In each of these two contexts, different system architectures can be denoted and, as previously mentioned, two different communication modes.

The slave device has the task of connecting via Bluetooth Low Energy to the data source, receiving the data present in the uuid characteristics and services and processing them in order to sift through the data and collect the fundamental ones in a JSON file.

It must then send this data depending on the position of the subject.

The master device has the task of initiating the reception of the message, receiving or requesting data from the slave depending on the communication protocol and finally accumulating a certain amount of data and processing it in

such a way as to understand whether or not there is a danger to the patient's health.

Finally, the system must archive the recorded data and send notifications capable of warning the patient, or a second subject, in the event of a health danger.

In the following pages we will see how the two protocols work in detail and when (or based on what) the type of communication protocol is changed, how the data is received and processed, how it is archived and the notification.

2.1 Architecture of the system

The basic architecture of the system remains the same depending on the two contexts of use.

It is based on the fact that the master device (ESP8266) is fixed in a specific place, probably the subject's home, while the slave device (ESP32) is a mobile device that must be strictly close to the data source which is the smartwatch.

In order for the slave device to be a mobile device it is possible to power it via a powerbank. This is the physical architecture of the system, while the software architecture is based on one (or two) mqtt brokers located somewhere in the network that are a passageway for the raw data and an arrival point for the processed data.

Chapter 3

Connection and data management via BLE

In this section we analyze how the slave device connects and receives data from the data source.

To connect the two devices, a specific library and the MAC address of the data source were needed, obtained through a secondary sketch.

The choice to adopt this module is motivated by the fact that it natively implements BLE, also implemented on the smartwatch, and this offers several advantages in addition to being able to connect the two devices.

This communication protocol allows a low connection latency, that is, it allows a connection between two devices within milliseconds, unlike the classic Bluetooth which takes a few seconds.

The main feature of this protocol is the fact that it allows excellent energy savings thanks to its "duty cycle" which limits the activity of the device to only the moments of data transmission and reception.

This aspect is fundamental in the project as the devices that incorporate this technology are subject to limited available energy, therefore energy saving is essential to extend the uptime of the system.

The project architecture for this single part includes Piconet with a single slave.

Thanks to a preliminary study of the smartwatch it was possible to dis-

cover the fundamental properties of the device such as the MAC address of the device and all its service uuid and the uuid characteristics that identify the data that interest us. In particular, it was discovered that the data we are interested in resides within two features in particular, and therefore a feature can contain more than one data.

In the code inside the module it was possible to connect to the device via a particular Bluetooth library that allows saving storage space inside the device as the classic libraries used for this type of modules occupy excessive internal memory.

Using the MAC address it is possible to connect to the device and navigate all the data within the services. Once the ones of interest were isolated, we had to analyze and process them in order to discard the useless bytes and isolate only those containing important information.

3.1 Data analysis and processing

The data analysis and processing part is based on the updating of the data contained within the two services. In particular, we process the data within the two services separately.

Within service1 there is data relating to steps and kilocalories consumed, while in service2 there is data relating to heartbeats and blood pressure.

In the code, every time a piece of data is updated within the services of our interest, the data present is reanalyzed and reworked each time.

In particular, the data is in hexadecimal format, from which it was possible to recognize the data of interest by transforming each byte present within the service into decimal numbers.

Therefore, within each service, the data is saved within a byte array and only the bytes of interest are isolated, discarding those that do not carry information. This data is then saved within some variables to be inserted into a json file in order to send this data at a later time.

First of all, thanks to tests carried out on the device, all the Bluetooth

devices in the vicinity were scanned and, by comparing the MAC addresses, it was possible to find the device of interest.

Subsequently, through secondary codes uploaded to the module, we proceeded to analyze all the various features and services. From a human perspective, at least initially, it was not possible to identify the features of interest at first glance since the data was in hexadecimal format.

To overcome this problem, the values of the variables of interest visible on the device display were noted and transformed into hexadecimal values, in order to compare these values with the data present in the features.

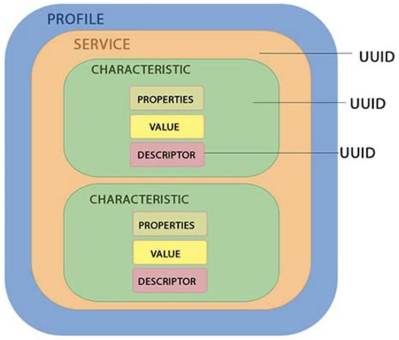
Once a match is found, to be sure, another comparison is made with different data to ensure that the found feature coincides with the variable of interest.

The found feature is then saved inside one of the pre-established variables which, subsequently, is manipulated with the aim of isolating the bytes of interest and converting them back to decimal.

The final result is that the data of interest is saved inside some variables. These variables are used to create a Json file through a specific library.

In particular, it was decided to add this step because the json format allows for excellent readability of the data by compilers and users, it also combines a key and a value which allows for excellent portability and is widely supported in various applications.

Within the code there are functions to manage data updating, which every time a data update notification is present the bytes within the feature are processed as described above.



Chapter 4

Indoor Context Methodology

In this part, we analyze the websocket connection that allows communication between the two boards.

This communication was made mandatory by the fact that the entire project cannot be loaded on a single board because it does not have the right memory requirements.

The communication between these two boards is based on websocket because this protocol has several advantages over other protocols that can be used for this type of applications.

This protocol provides a stable and bidirectional connection that allows the two devices to communicate simultaneously, and therefore also guarantees low latency, which is a great feature for applications of this type that require a rapid exchange of messages, it also reduces the volume of traffic compared to other protocols such as HTTP and is very simple to implement.

From a superficial point of view, we can say that the architecture provides that the esp32 module is the slave that collects data from the device, while the esp8266 module is the master that requests data from the slave at time intervals via polling messages.

The websocket is used to manage this communication and in the image below we can see the format of the data.

Communication via WebSocket is based on the fact that the master requests data from the slave device at variable intervals and the latter, upon

receiving this polling message, responds with the data of interest. Variable data request intervals have been implemented to improve efficiency and data reliability, and these intervals can decrease if a potential dangerous situation is detected.

Each time a dangerous situation is detected, the data request interval is halved each time until it reaches a minimum threshold. In this way, data is collected in a shorter time interval in order to "map" the possible dangerous situation more reliably.

This method is also useful for limiting energy consumption, since when there is no need for timely data collection, the system requests data within a few seconds (this threshold was set for testing purposes, but in a real case it could easily be increased to a few dozen seconds).

The system set up in this way could detect a problem within a few dozen seconds.

A problem in this part was found in the dynamic management of IP addresses by the two devices, since messages sent via websocket need the recipient's IP address. In fact, when the two devices are connected to the same network, it is not certain that the IP addresses always remain the same, so a way was designed so that the two devices, when they connect to the same network, share their IP addresses before establishing a connection. This was possible thanks to the use of a protocol called ESP-NOW where we will see later how it was used.

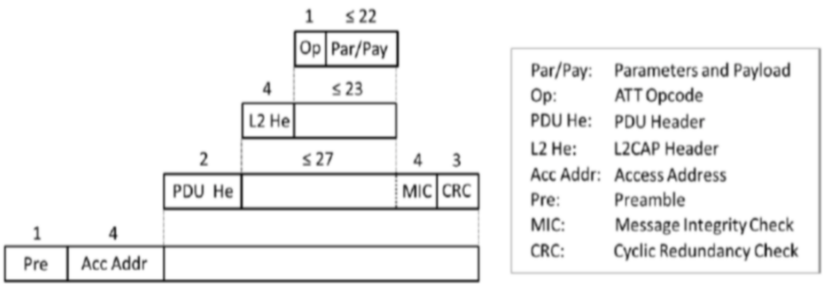


Figura 5 - BLE: Formato trama [19]

4.1 Dynamic IP address management

To connect the two devices to share data, you need to first create a websocket connection using a specific port and the recipient's IP address.

This, as mentioned before, could create some problems because the IP address management is dynamic and this means that you cannot establish a static IP address, unless you act differently but it would lead to a long and uncomfortable work.

To solve this problem, a mechanism has been set up for which the two devices share their IP addresses in order to establish this connection.

Address sharing cannot happen after connecting to Wi-Fi because obviously you would not know the recipient's address in advance, so it is shared after the Wi-Fi connection but using a protocol called ESP-NOW.

ESP-NOW is a wireless communication protocol designed to enable low-latency and low-power communication between devices. This protocol allows direct communication between devices coexisting with the Wi-Fi connection.

This protocol was chosen for its features such as low latency, thanks to the fact that there is no need to manage TCP/IP connection, but it also guarantees low energy consumption and allows for a unicast or broadcast connection, as well as supporting 128-bit encryption.

Once the IP address has been transmitted, this IP address is used to establish the connection, and this happens every time the device connects to the primary network, thus ensuring reliable address management.

4.2 Energy saving mechanism

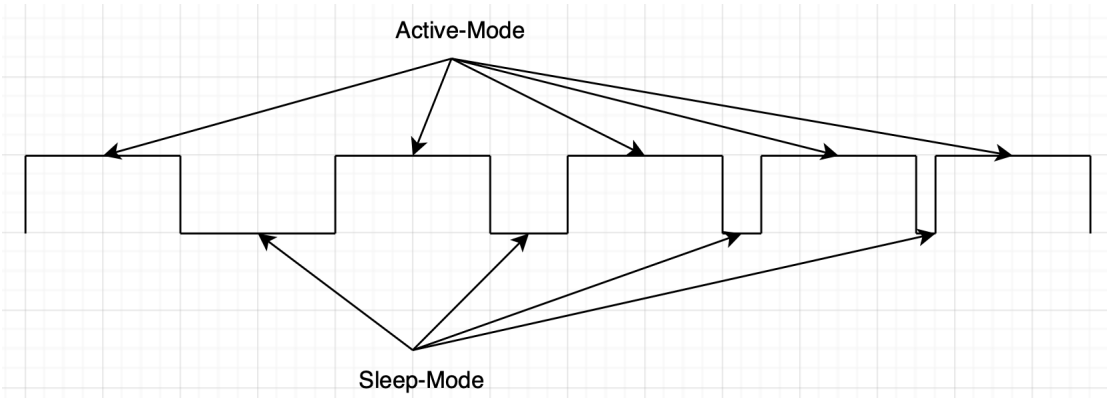
Inside the indoor context this energy saving mechanism has been implemented in order to save energy and extend the life of the system powered by a limited energy source. This system allows to reduce energy consumption when the project does not need to send data using Wi-Fi but keeps the microprocessor active, in this way it can continue to process the data.

It was chosen to use an energy saving mode called Modem-Sleep because it better fits the needs of the system. In fact the Modem-Sleep mode of the ESP8266 is one of the most used energy saving techniques to reduce energy consumption when the Wi-Fi module is not in continuous use.

The Wi-Fi module of the ESP8266 is deactivated or "put to sleep" when it is not actively needed, such as during periods in which it is not transmitting or receiving data, in our case it would be the period between sending one piece of data and another spaced by a variable period of time.

Unlike Deep-Sleep mode, where both Wi-Fi and CPU are turned off, in Modem-Sleep the CPU continues to run. This allows the device to react immediately to events without having to wait for the time needed for it to fully wake up from Deep-Sleep mode.

This feature is essential for the system as it constantly receives updates from other devices so it is essential that its response is timely. This mode is perfect for IoT devices that require intermittent Wi-Fi connections but need to perform calculations or control sensors in between tasks like in our case.



4.3 Connecting Device

As already mentioned before, the implementation of this protocol is very easy and the connection of the two devices is even simpler. It's all manageable thanks to a wifi connection and a specific library that allows you to initialize the websocket.

The connection is established in the code setup via two instructions which involve the creation of a websocket server on the device port, which in this case is port 80, and the second instruction which involves specifying the IP address of the device to connect to.

After that, a function was created that manages the different situations that can happen, but basically there are two. The first is to print a message when the connection with the master has occurred to acknowledge the connection, The second manages the messages containing the data arriving from the master.

Piconet con
singolo slave

4.4 Managing Data

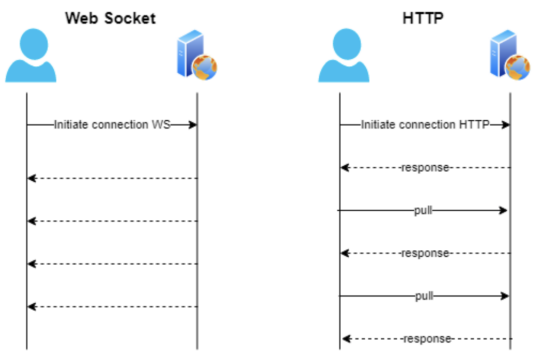
Once data has been received via websockets, the next step is to manage it and derive the relevant information and process it. In terms of data management, firstly, the data coming from the master is deserialized in order to extract the values from the json file and save the data in local variables.

The first two values we want to analyze are the steps taken and the heartbeats. These two parameters are fundamental for relating the heart rhythm to the patient's physical activity. This relationship arises from the fact that measuring only heartbeats to monitor heart health could make us fall into error since an abnormal heartbeat, which can be a danger signal at rest, does not constitute a danger if it occurs in conditions of a good level. of physical activity.

Therefore, the information on the steps taken by the patient must be processed with the aim of understanding whether the subject is carrying out a certain physical activity.

Naturally, analyzing a single value at a time does not lead to truthful and significant results, so to achieve a good level of data reliability we chose to collect ten measurements, one every 5 seconds.

These measurements from the data are saved each time within some array and the data is processed only when the array is complete. As mentioned before, the analysis of the patient's physical activity is calculated based on the steps taken.



WebSocket vs HTTP communication

This is possible because all the steps taken from the beginning of the measurement to the end are calculated, so the difference between the first and tenth step measurement is made. The time taken to make these 10 measurements is also measured. Thanks to these parameters, calculated starting from the raw data, it was possible to obtain the number of steps taken per second during the measurement.

Precisely this variable will be used to establish whether the patient is doing physical activity, as according to some readings, it can be classified as physical activity when one takes more than two steps per second.

The objective of measuring heartbeats is to be able to recognize potential situations of Tachycardia, Arrhythmia and Brachycardia. Precisely from the heartbeat measurements we obtain the average of these ten measurements and, if the average is greater than 100, it means that on average they have more than 100 beats per minute in a time of approximately 50 seconds, and if the steps per second are less than 2, this means that the subject is not performing a physical activity, a json object is created within the "alarm" key which identifies a possible tachycardia.

The same applies to the possible situation of brachycardia, that is, when the number of steps per second does not exceed 2 and when the average heartbeat is less than 50.

While with regards to cardiac arrhythmia a different approach was chosen. This approach is always based on some well-known mathematical formulas. Since cardiac arrhythmia is characterized by changes in the number of heart beats, the standard deviation was used to identify this pathology.

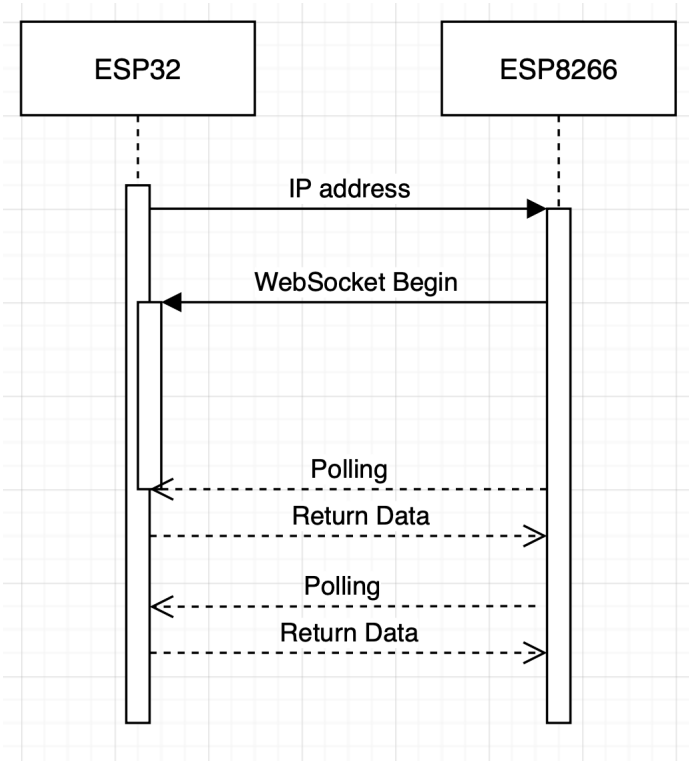
Standard deviation is appropriate for this application because it quantifies the dispersion of the data around the mean.

Then the standard deviation is calculated and the steps per second are also compared, even if not strictly correlated with the cardiac arrhythmia. This is to distinguish the case in which part of the measurements are carried out at rest and part are carried out under stress, this could lead to wrong conclusions as there could be a large difference between the beats at rest and

those under stress which would induce the deviation standards to grow. In this way we can distinguish this case from the normal one. Also in this case, if the two conditions are met, a json object is created that reports a danger message. Finally, associated with the creation of the alert json object, the change of a flag variable that identifies a potential danger is set.

This is useful because when this flag variable becomes true a mechanism is activated whereby the time interval in which the master requests data from the slave is halved, consequently producing the doubling of the report frequency. Obviously this happens every time a set of measurements presents a dangerous situation, thus increasing the number of data available and the reliability of the data. However, the increase in the slave's report frequency is limited by a preset threshold.

In fact, the time interval in which the master requests data from the slave is initially set at 5 seconds and can be halved a maximum of 3 times, i.e. a time interval between one piece of data and another of 625 ms. This means that, in a non-alarm situation, the ten measurements take place in approximately 50 seconds, up to a maximum of 6.25 seconds in maximum alarm conditions.



Chapter 5

Outdoor Context Methodology

In this part we analyze how the system acts in the case in which the slave device connects to a network other than the primary one, which to ensure good reliability has been chosen as the secondary network the router of your cell phone.

When the system connects to this secondary network it is understood that the subject is outside the perimeter of the primary network and therefore it would be useless to send the data via websocket so another communication protocol is used so that the data arrives from the slave to the master. This is possible through the use of a mqtt broker placed between the two devices that acts as an intermediary.

The first thing to do is obviously monitor the status of the network thus making it possible to recognize the belonging to the first or second network. When the system recognizes that it belongs to the secondary network then it begins to initialize a connection via mqtt broker.

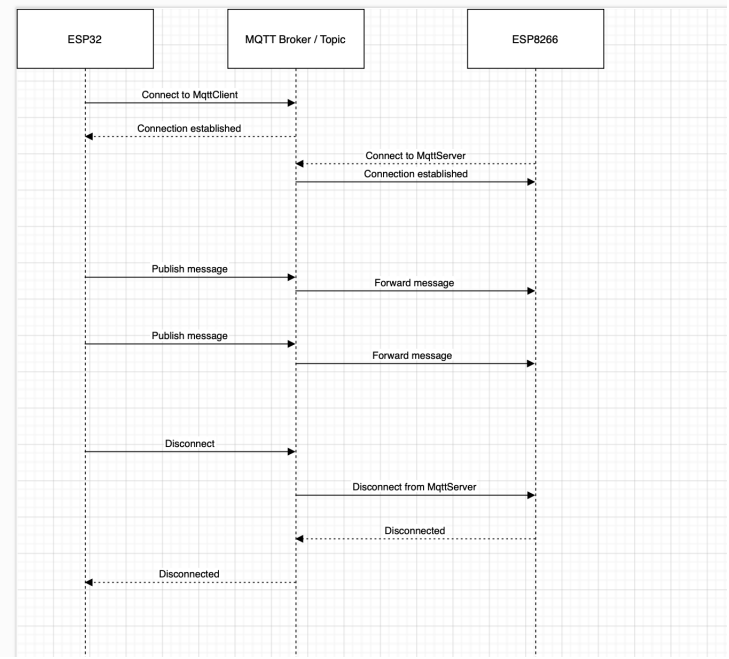
5.1 Managing Data

Once the connection with the mqtt broker has been created, it is possible to send the data in json format, which is the same that the system uses when communicating via websocket, to a mqtt broker on a specific topic created specifically for the project. The json data is sent to the mqtt broker at regular intervals. Here ends the task of the slave device, now it is up to the master device to be able to find the data, extract it and manage it.

The master device must be able to find the data, but it cannot know if the data was sent via websocket or via mqtt, unless using a very complex methodology, monitoring the continuous reception of the IP address.

To overcome this problem, the master device monitors the status of both the websocket connection and the connection via mqtt, so it does not need to recognize the communication methodology.

At this point the device only needs to connect to the same broker and the same topic, not as a publisher but as a subscriber. Using a callback function, it is possible to constantly monitor the topic, so that it is possible to obtain the data sent as a message on the specific topic. Once the data has been received, it is deserialized and managed in the same way as data received via websocket.



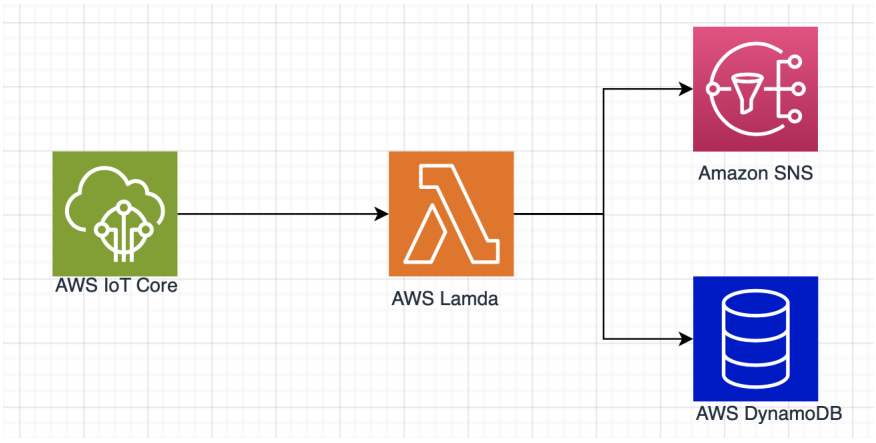
Chapter 6

Data migration and management in the Cloud

Once the data has arrived at the master device and processed as mentioned before, it is sent to an Amazon AWS MQTT broker. This involves a sharing mechanism similar to the one used in the case of an outdoor context, adding some permissions to ensure an excellent level of security.

In this project the Cloud part has several important tasks such as receiving the data, processing it, checking that the values are within some security parameters and reporting via notifications if some problems have been found.

finally the code must save all the data sets in JSON within the tables of a noSQL database completely managed by the provider, in addition to managing the resources present in the cloud.



6.1 Receiving Data

To send messages, as mentioned before, certificates are used and must be implemented in the code to introduce a higher level of security. There are three of these certificates and they are used to configure a secure SSL/TLS connection, they are all downloadable from the AWS server when AWS IoT Core is installed.

In particular, a root certificate has been configured, which is essential to be able to establish a secure connection between the device and the AWS server. The device certificate has also been implemented, which allows you to certify the device, identify it uniquely and guarantees that only certified devices can communicate with the endpoint.

And finally, the private key is a certificate that allows you to sign data coming from a certified device. The data coming from the device and arriving at the AWS IoT Core service allow you to trigger an AWS Lambda function that manages the data and manages the messages as we will see later.

6.2 Saving Data

Saving data in AWS DynamoDB is a very simple operation, first you need to create a table on the Database. Inside the Lambda code it must be instated and then, the messages that trigger the function are broken down into data saved inside local variables and, if this variable contains numeric values then they are saved in the database.

Otherwise, and therefore the data is not present under its own label, then a null value is indicated in the database.

Since AWS provides multiple storage services, this service was chosen due to the various advantages it offers, such as the fact that it is automatically scalable and completely self-managed by the service provider.

DynamoDB is designed to offer extremely low latencies (in the order of milliseconds) for each request, regardless of the size of the database. This makes it ideal for applications that require rapid response times like this ap-

plication. it also offers excellent durability since each database is replicated multiple times in different areas ensuring data availability and durability. Finally, in this case we use key-value pair data format, but this type of database allows you to use different data formats.

6.3 Notification Service

The notification service is managed by AWS SNS which allows you to notify the user using different protocols such as SMS, email, HTTP, etc. etc. In the case of the project, the user is reached by an email notification if the alarm key of the JSON file has an alarm message associated with it. But he will also be notified if the JSON file contains non-optimal blood pressure values with the aim of warning the user that it is time to do further tests.

In the project, it was possible to choose between different available protocols but email was chosen over the SMS service because SMS introduces a delay between the reception of the alarm and therefore the sending of the message and its arrival of several seconds.

Instead, using the email notification, it was realized that the delay is enormously smaller, in the order of milliseconds because to the human eye it seems like an instant notification.

Chapter 7

Technologies Used

In this chapter, we will analyze all the protocols and technologies that have been used within the project to transmit data from one device to another.

In the following project, several cutting-edge technologies have been integrated to ensure maximum efficiency, scalability and security. Each technology has been selected to respond to specific functional and performance needs, helping to build a robust and versatile system.

The following technologies have been used to ensure a good level of security, in addition to ensuring scalability and adaptability in different situations and contexts.

Overall, the combination of these technologies has allowed us to develop a solution that not only fully meets the requirements of the project, but is also ready for future expansions and integrations.

7.1 Bluetooth Low Energy

Bluetooth Low Energy (BLE) is a wireless communication technology developed to enable low-energy devices to communicate with each other.

It is part of the Bluetooth 4.0 specification (and later versions) and was designed for applications that require low-speed data transmission with minimal power consumption.

It is particularly well suited to our project because, as we have already

seen, very low speeds are available and minimal power consumption is required since the system duration on a single charge must be extended as much as possible.

In the project, communication is based on the sharing of packets contained in features and services, and features and services are fundamental elements for the structure and operation of communication between devices.

Each feature and service is identified by a universally unique identifier, known as a UUID

A service is a logical set of features that together provide a specific functionality. For example, a "Heart Rate Measurement" service may include features for heart rate, battery level, etc. While services are used to group related characteristics and organize how data is structured and accessed in a BLE device.

7.2 ESP-NOW

ESP-NOW is a wireless communication protocol used primarily in ESP8266 and ESP32 microcontrollers.

This protocol allows data transmission between ESP devices without the need for a traditional Wi-Fi connection, making it ideal for low-power, short-range, and low-latency applications.

ESP-NOW is based on a peer-to-peer (P2P) communication system, where ESP devices can communicate directly with each other without going through a Wi-Fi access point or router. Communication occurs via data packets at the MAC (Media Access Control) layer, bypassing the TCP/IP stack used in standard Wi-Fi connections. This leads to several advantages, such as reduced latency and energy efficiency.

Since this protocol works without going through a Wi-Fi access point, this means that it does not introduce any disturbance to Wi-Fi connectivity, making these two protocols suitable for coexistence. Thanks to this feature, it was possible to share the IP address of the slave device with the master device,

thus allowing for a stable and completely automatically managed connection without having to manually enter the IP address of the slave.

This protocol allowed us to solve the problem of IP address dynamics.

Since the data is transmitted directly at the MAC level, latency is extremely low, making ESP-NOW suitable for applications where response time is critical, such as remote control or real-time data collection.

The protocol is designed to consume very little energy, which makes it ideal for battery-powered devices, such as the slave device that is assumed to be powered by a power bank. Furthermore, the range of ESP-NOW is similar to that of Wi-Fi, and can vary depending on the environment, signal strength and antenna used. In open spaces, it can reach several tens of meters.

This fits perfectly in our context as we need the device to have this feature so that it can share the IP address even at the edges of the connectivity range of our Wi-Fi network.

Finally, in addition to being extremely easy to implement, ESP-NOW supports data encryption via a pre-shared key (PSK). This ensures that the data transmitted is secure and not accessible to unauthorized devices.

7.3 WebSocket

WebSockets are a powerful solution for building interactive and reactive applications that require real-time communication between clients and servers. Offering a persistent, bidirectional connection with low latency, they are particularly suited for scenarios where traditional HTTP request-response models cannot provide the required performance or efficiency.

As in our case, under maximum stress conditions (which in our case is sending messages every 625 ms, but in reality this threshold could be lowered) introducing traditional models such as HTTP may not produce the desired results since introducing a delay between requests and responses would ruin efficiency.

It is a technology that enables bidirectional, real-time communication

between a client (such as the device) and a server (such as the master device) over a single TCP connection. This protocol is particularly useful for applications that require instant updates or continuous interactions.

Being a bidirectional communication technology allowed us to introduce a way of communicating between the slave device and the master that is roughly similar to a polling technique.

Where the master queries, with a specific message, the slave, and if the message received is the same as the specific message in question, then the slave sends the data in that format.

Once established, the WebSocket connection significantly reduces the overhead compared to traditional HTTP requests. There is no need to send full HTTP headers for each message, which makes the communication lighter and faster.

Since the connection remains open, messages can be sent immediately without the delay associated with establishing new connections, resulting in extremely low latency.

7.4 MQTT

MQTT is a lightweight messaging protocol designed for resource-constrained devices and low-bandwidth networks such as the devices that were used in this project. It was designed to be extremely lightweight, with minimal protocol overhead, making it ideal for resource-constrained devices.

MQTT uses a "publish/subscribe" communication model. In this model, devices (clients) publish messages to specific "topics", and other devices "subscribe" to those topics to receive the messages. This allows for a decoupling between message producers and consumers, improving the scalability and flexibility of the system. Communication between clients occurs through a central intermediary called a "broker". The broker receives messages published to a given topic and distributes them to all clients subscribed to that topic.

Chapter 8

Results

In this chapter we will analyze the results obtained from the simple test of the project, then analyzing how the project behaves in the context of normal use and how it behaves when it is necessary to change scenario with a context transition between indoor and outdoor.

We will also analyze the performance analyses based on some typical parameters of communication networks such as latency, bandwidth, error rate, reliability, etc.

8.1 Test

Tests were carried out on the project, noting that, beyond a small settling time to allow the devices to connect to their respective networks, initialize and connect the devices for the connection, the system starts operating in a few seconds.

We realized that, assuming that the smartwatch has already recorded data before the system connects to it, when the connection is made the first data received from the devices are not truthful as they all indicate zero values. This situation does not present problems for the reliability of the data and the project itself since, even assuming a minimum data acquisition time, this type of corrupted data would not be taken into consideration.

Operation in a simple context, both indoors and outdoors, does not

present any problems and the project works as expected.

The test carried out in the transition of the contexts produced excellent results, denoting the fact that the system reliably recognizes when the system is not within range of the main network and therefore tries to connect promptly to the reserve network. Although, when you come back into range of the primary network the system waits for the user to disconnect the secondary network so that the system can connect to the primary network.

8.2 Performance

8.2.1 Latency

In this analysis, the latency of messages was measured using the WebSocket protocol.

Latency is the total time between sending a request and receiving the response. It is a crucial measure to evaluate the efficiency and performance of a network.

The latency time has three crucial components which are, the transmission time, the propagation time and the processing time, but in the analysis performed, the RTT is measured, that is, the Round-Trip Time which takes into account all these times. However, the transmission times and the processing times can be considered insignificant and therefore we can consider the propagation time as the only significant time and equate it to the RTT.

The propagation latency depends on the signal strength, and therefore influences any lost data packets, and on the distance between the point of origin and the point of destination of the data.

Within the analysis, simple mechanisms were used to measure this quantity such as ping and trace routes.

During the analysis, the distance between the two communicating devices was varied in order to obtain different measurements and to enclose this analysis in a range that includes the different cases that can be found in a real

application. From the data analysis, a latency between the range of 20 and about 230 ms was measured.

8.2.2 Bandwidth

To calculate the bandwidth between the smartwatch and the slave device, the maximum amount of data sent to the slave device in a unit of time was measured. The calculated estimate was just over 80 bps.

8.2.3 Error rate

The error rate in telecommunication networks is a measure of the quality of communication, indicating the frequency with which errors occur during data transmission. This rate is crucial to evaluate the reliability and efficiency of a network.

In this case the measure involves the number of incorrect messages compared to the total number of messages transmitted. The only incorrect messages found were those described above, typical of the initialization of the connection.

As already said before, these data do not influence the reliability of the system, but even if we want to consider them incorrect messages, considering that these messages appear only at the beginning of the connection and in a limited number, we can safely say that the error rate is approximately zero.

8.2.4 Signal quality

To measure the quality of the Wi-Fi connection between the two devices, the RSSI index was used.

Calculating the signal quality through the RSSI (Received Signal Strength Indicator) is a common practice in wireless communications to evaluate the signal strength between two devices, such as an ESP32 and an ESP8266. RSSI is a value that indicates the power of the signal received by a wireless device, and is often used to estimate the quality of the connection.

Using secondary sketches created adhoc to measure this index, the index was also measured at different distances between the two devices in order to map the index even in non-optimal conditions.

The results obtained show a good signal quality with an RSSi index between -30 dBm and -60 dBm.

8.2.5 Energy Consuption

From an energy point of view we can say that the energy saving mechanism allows to extend the life of the system exponentially

In normal conditions the average consumption of a device that uses Wi-Fi networks to transmit data would be between 100 and 240 mA, depending on the length of the data to be sent.

If we assume that our mobile device is connected to a 5000mAh power bank the system has just under 3 days of autonomy.

If instead we implement this mechanism we can save a huge amount of energy that would otherwise be wasted.

On average the device takes about 50ms to send a piece of data and then falls into sleep mode for 5 seconds where the device consumes about 15mA.

Instead during transmission it consumes about 160 mA, but for the calculation that was carried out it was chosen to put ourselves in the worst possible hypothesis and therefore consider a maximum consumption of the device of 240 ms.

So by adding the energy contributions we discover that in a cycle of active mode and sleep mode lasting 5.05 seconds the device consumes 87 mA of energy and therefore about 17 mA every second. If we divide the capacity of the power source and the average consumption every second of the device we have as a result the number of hours of autonomy that our device has with a single charge adopting this energy saving protocol.

The result of the calculations is about 290 hours which corresponds, rounding down, to 12 days.

8.2.6 Security

In terms of security, the system has an average level, which can certainly be improved, given that it is an experimental project.

An additional level of security could be added to communication via websocket, since a basic protocol is used, but replacing it with a WSS would provide the same service but more secure thanks to the implementation of SSL/TLS certificates.

WebSocket Secure is the secure version of the WebSocket protocol, which uses TLS (Transport Layer Security) to encrypt data in transit. WSS is equivalent to HTTPS for WebSocket connections and offers protection against interception and data alteration. During the initial handshake, the client and server establish a secure connection using TLS, and all data transmitted via this connection is encrypted.

As for communication with AWS services, the level of security is quite high, given the implementation of various certificates in the device and policies applied to the services.

Performance Analisys			
Parameters	Min	Max	Units
Latency	20	230	ms
Bandwidth	80		bps
Error rate	$\simeq 0\%$		
Signal quality (RSSI)	-60	-30	dBm
Energy Consuption	100	240	mA
System lifetime	3	4	days
Manageability	High		
Security	Medium		
Implementation Costs	Low		

Chapter 9

Discussion

9.1 Analysis of the results

From the analysis of the results it is clear that, from the point of view of network performance, the results are judged quite good, considering a good range of signal quality, a low error rate.

The system also has a low bandwidth that however does not affect the system performance since the measured bandwidth is a good enough level for the application in question. A higher bandwidth would not bring advantages to the system, but on the contrary would only bring potential problems.

Good results also regarding latency that, on average, does not present problems for the application and that, in the worst case scenario, does not present levels that would introduce problems within the project.

Chapter 10

Conclusions

10.1 Future prospects

As future prospects, we hypothesize the introduction of acknowledgment messages when data is received via WebSocket, adding a Timeout that when it expires and the acknowledgment is not detected, the system checks the connection. This improvement would contribute to increasing the reliability of the system.

Furthermore, it could be possible to introduce structural changes to increase the memory that is heavily tested by the multiple libraries installed for the operation of the system.

This increase in memory would allow us to add a mechanism for saving the data on the local file, thus increasing the reliability of the data, thus allowing the system to function even if the AWS servers should have problems. It would be enough to have the main network working or to have your own personal hotspot available.

Code Documentation

Authors: Vacca A., Thiam P., Mahmoud M.,

September 17, 2024

10.2 Source Code of ESP32

Below is the source code of the Slave device:

```
#include <WiFi.h>
#include <WebSocketsServer.h>
#include <ArduinoJson.h>
#include <NimBLEDevice.h>
#include <esp_now.h>
#include <PubSubClient.h>  // Libreria MQTT

// Indirizzo MAC del dispositivo ESP8266
uint8_t esp8266Address[] = {0xB4, 0xE6, 0x2D, 0x39, 0x77, 0xB0};

// Variabili e costanti per la gestione dei dati
#define TARGET_DEVICE_MAC "78:02:b7:48:6e:87"  // Indirizzo MAC del dispositivo target
#define SERVICE_UUID1 "0000fee7-0000-1000-8000-00805f9b34fb"  // UUID del primo servizio
#define CHARACTERISTIC_UUID1 "0000fea1-0000-1000-8000-00805f9b34fb"  // UUID della prima caratteristica
#define SERVICE_UUID2 "000055ff-0000-1000-8000-00805f9b34fb"  // UUID del secondo servizio
#define CHARACTERISTIC_UUID2 "000033f2-0000-1000-8000-00805f9b34fb"  // UUID della seconda caratteristica
#define MAX_LENGTH 10

uint8_t byteArray1[MAX_LENGTH];
uint8_t byteArray2[MAX_LENGTH];
size_t byteArrayLength1 = 0;
size_t byteArrayLength2 = 0;

uint8_t selectedBytes[2];
int PassiEffettuati = 0;
int KcalConsumate = 0;

int lastPassiEffettuati = 0;
```

```
29 int lastKcalConsumate = 0;
30 int lastBattitoCardiaco = 0;
31 int lastPressioneSistolica = 0;
32 int lastPressioneDiastolica = 0;
33 int BattitoC = 0;
34 int PressioneS = 0;
35 int PressioneD = 0;
36
37 String output;
38
39 NimBLEClient* pClient = nullptr;
40 bool isConnected = false;
41
42 // Configurazione WebSocket e MQTT
43 WebSocketsServer webSocket = WebSocketsServer(80);
44 WiFiClient wifiClient;
45 PubSubClient mqttClient(wifiClient);  // Client MQTT
46
47 // Callback per ESP-NOW
48 void onDataSent(const uint8_t *mac_addr, esp_now_send_status_t
       status) {
49   Serial.print("Messaggio inviato: ");
50   Serial.println(status == ESP_NOW_SEND_SUCCESS ? "Successo" :
       "Fallito");
51 }
52
53 void onDataRecv(const esp_now_recv_info_t *info, const uint8_t
       *incomingData, int len) {
54   Serial.print("Indirizzo IP ricevuto da ESP8266: ");
55   Serial.write(incomingData, len);
56   Serial.println();
57 }
58
59 // Callback per il client BLE
60 class MyClientCallback : public NimBLEClientCallbacks {
61   void onConnect(NimBLEClient* pclient) override {
62     Serial.println("Connected to the device");
```

```
63      isConnected = true;
64    }
65
66    void onDisconnect(NimBLEClient* pclient) override {
67      Serial.println("Disconnected␣from␣the␣device");
68      isConnected = false;
69    }
70  };
71
72  // Callback per la caratteristica 1
73  void notifyCallback1(NimBLERemoteCharacteristic*
      pRemoteCharacteristic, uint8_t* pData, size_t length, bool
      isNotify) {
74    String hexValue = "";
75    byteArrayLength1 = 0;  // Reset della lunghezza dell'array di
        byte
76    for (int i = 0; i < length; i++) {
77      char c = pData[i];
78      if (c < 16) hexValue += '0';
79      hexValue += String(c, HEX);
80
81      // Salva il valore della caratteristica nell'array di byte
82      if (byteArrayLength1 < MAX_LENGTH) {
83        byteArray1[byteArrayLength1++] = c;
84      }
85
86      // Salva il secondo e il terzo byte nell'array
          selectedBytes
87      if (i == 1) {
88        selectedBytes[1] = c;
89      } else if (i == 2) {
90        selectedBytes[0] = c;
91      }
92
93      if (i == 7) {
94        KcalConsumate = c;
95      }
```

```
96    }
97
98    // Converti i byte selezionati in valore decimale
99    PassiEffettuati = (selectedBytes[0] << 8) | selectedBytes[1];
100
101   if (PassiEffettuati != 0) {
102     lastPassiEffettuati = PassiEffettuati;
103   }
104
105   if (KcalConsumate != 0) {
106     lastKcalConsumate = KcalConsumate;
107   }
108 }
109
110 // Callback per la caratteristica 2
111 void notifyCallback2(NimBLERemoteCharacteristic*
      pRemoteCharacteristic, uint8_t* pData, size_t length, bool
      isNotify) {
112   byteArrayLength2 = 0;  // Reset della lunghezza dell'array di
        byte
113   String hexValue = "";
114   int BattitoCardiaco = 0;
115   int PressioneSistolica = 0;
116   int PressioneDiastolica = 0;
117
118   for (int i = 0; i < length; i++) {
119     char c = pData[i];
120     if (c < 16) hexValue += '0';
121     hexValue += String(c, HEX);
122
123     // Salva il valore della caratteristica nell'array di byte
124     if (byteArrayLength2 < MAX_LENGTH) {
125       byteArray2[byteArrayLength2++] = c;
126     }
127
128     // Converti i byte rimanenti in valori decimali se i primi
        due byte sono 0xE5 e 0x11
```

```
129     if (i >= 2 && byteArray2[0] == 0xE5 && byteArray2[1] == 0
            x11) {
130       BattitoCardiaco = (BattitoCardiaco << 8) | c;
131     }
132
133     if (byteArray2[0] == 0xC7 && byteArray2[1] == 0x00) {
134       if (i == 3) {
135         PressioneSistolica = c;
136       } else if (i == 4) {
137         PressioneDiastolica = c;
138       }
139     }
140   }
141
142   if (BattitoCardiaco != 0) {
143     lastBattitoCardiaco = BattitoCardiaco;
144   }
145
146   if (PressioneSistolica != 0) {
147     lastPressioneSistolica = PressioneSistolica;
148   }
149
150   if (PressioneDiastolica != 0) {
151     lastPressioneDiastolica = PressioneDiastolica;
152   }
153 }
154
155 void setup() {
156   Serial.begin(115200);
157
158   WiFi.mode(WIFI_STA);
159   WiFi.setSleep(false);
160
161   reconnectWiFi();
162 }
163
164 void loop() {
```

```
165   if (WiFi.status() != WL_CONNECTED) {
166     reconnectWiFi();
167   }
168
169   if (mqttClient.connected()) {
170     mqttClient.loop();
171   } else {
172     Serial.println("MQTT disconnesso, tentando la riconnessione
          ...");
173     while (!mqttClient.connect("clientId-7H6uqniSer")) {
174       delay(1000);
175     }
176   }
177
178   mqttClient.loop();
179
180   if (!isConnected) {
181     if (pClient->connect(NimBLEAddress(TARGET_DEVICE_MAC))) {
182       Serial.println("Connesso al dispositivo target");
183
184       NimBLERemoteService* pRemoteService1 = pClient->
            getService(SERVICE_UUID1);
185       if (pRemoteService1) {
186         NimBLERemoteCharacteristic* pRemoteCharacteristic1 =
              pRemoteService1->getCharacteristic(
              CHARACTERISTIC_UUID1);
187         if (pRemoteCharacteristic1 && pRemoteCharacteristic1->
              canNotify()) {
188           pRemoteCharacteristic1->subscribe(true,
                notifyCallback1);
189         } else {
190           Serial.println("Caratteristica 1 non trovata");
191         }
192       } else {
193         Serial.println("Servizio 1 non trovato");
194       }
195
```

```
196       NimBLERemoteService* pRemoteService2 = pClient->
              getService(SERVICE_UUID2);
197       if (pRemoteService2) {
198         NimBLERemoteCharacteristic* pRemoteCharacteristic2 =
                pRemoteService2->getCharacteristic(
                CHARACTERISTIC_UUID2);
199         if (pRemoteCharacteristic2 && pRemoteCharacteristic2->
                canNotify()) {
200           pRemoteCharacteristic2->subscribe(true,
                  notifyCallback2);
201         } else {
202           Serial.println("Caratteristica 2 non trovata");
203         }
204       } else {
205         Serial.println("Servizio 2 non trovato");
206       }
207     } else {
208       Serial.println("Connessione al dispositivo target fallita
              ");
209     }
210   }
211
212   StaticJsonDocument<200> doc;
213   doc["PassiEffettuati"] = PassiEffettuati != 0 ?
          PassiEffettuati : lastPassiEffettuati;
214   doc["KcalConsumate"] = KcalConsumate != 0 ? KcalConsumate :
          lastKcalConsumate;
215   doc["BattitoCardiaco"] = BattitoC != 0 ? BattitoC :
          lastBattitoCardiaco;
216   doc["PressioneSistolica"] = PressioneS != 0 ? PressioneS :
          lastPressioneSistolica;
217   doc["PressioneDiastolica"] = PressioneD != 0 ? PressioneD :
          lastPressioneDiastolica;
218
219   serializeJson(doc, output);
220   Serial.println(output);
221
```

```
222   if (WiFi.SSID() == ssid1) {
223     //webSocket.broadcastTXT(output);
224   } else if (WiFi.SSID() == ssid2) {
225     mqttClient.publish(mqttTopic, output.c_str());
226     delay(4000);
227   }
228 }
229
230 void reconnectWiFi() {
231   if (WiFi.status() != WL_CONNECTED) {
232     Serial.println("Connessione Wi-Fi persa. Tentativo di
          riconnessione...");
233
234     WiFi.begin(ssid1, password1);
235     int retryCount = 0;
236     while (WiFi.status() != WL_CONNECTED && retryCount < 10) {
237       delay(1000);
238       Serial.println("Connessione WiFi...");
239       retryCount++;
240     }
241
242     if (WiFi.status() == WL_CONNECTED) {
243       Serial.print("Connesso a Wi-Fi (prima rete) con IP: ");
244       Serial.println(WiFi.localIP());
245       setupESPNow();
246     } else {
247       Serial.println("Connessione alla prima rete fallita,
            tentando la seconda rete...");
248       WiFi.begin(ssid2, password2);
249       retryCount = 0;
250       while (WiFi.status() != WL_CONNECTED && retryCount < 10)
            {
251         delay(1000);
252         Serial.println("Connessione WiFi alla seconda rete...")
              ;
253         retryCount++;
254       }
```

```
255
256       if (WiFi.status() == WL_CONNECTED) {
257         Serial.println("Connesso␣alla␣seconda␣rete!");
258         Serial.print("ESP32␣IP␣address:␣");
259         Serial.println(WiFi.localIP());
260         setupMQTT();
261       } else {
262         Serial.println("Impossibile␣connettersi␣a␣nessuna␣rete␣
              WiFi");
263       }
264     }
265   }
266 }
267
268 void setupESPNow() {
269   if (esp_now_init() != ESP_OK) {
270     Serial.println("Errore␣nell'inizializzazione␣di␣ESP-NOW");
271     return;
272   }
273
274   esp_now_register_send_cb(onDataSent);
275   esp_now_register_recv_cb(onDataRecv);
276
277   esp_now_peer_info_t peerInfo;
278   memcpy(peerInfo.peer_addr, esp8266Address, 6);
279   peerInfo.channel = 0;
280   peerInfo.encrypt = false;
281
282   if (esp_now_add_peer(&peerInfo) != ESP_OK) {
283     Serial.println("Errore␣nell'aggiunta␣del␣peer");
284     return;
285   }
286
287   String ipStr = WiFi.localIP().toString();
288   esp_err_t result = esp_now_send(esp8266Address, (uint8_t *)
        ipStr.c_str(), ipStr.length());
289   if (result == ESP_OK) {
```

```
290     Serial.println("Indirizzo␣IP␣inviato␣con␣successo");
291   } else {
292     Serial.print("Errore␣nell'invio␣dell'indirizzo␣IP,␣codice:␣
          ");
293     Serial.println(result);
294   }
295
296   webSocket.begin();
297   webSocket.onEvent(webSocketEvent);
298 }
299
300 void setupMQTT() {
301   mqttClient.setServer(mqttServer, mqttPort);
302   if (!mqttClient.connect("clientId-7H6uqniSer")) {
303     Serial.println("Connessione␣al␣broker␣MQTT␣fallita");
304   } else {
305     Serial.println("Connesso␣al␣broker␣MQTT");
306   }
307 }
308
309 void webSocketEvent(uint8_t num, WStype_t type, uint8_t *
     payload, size_t length) {
310   switch (type) {
311     case WStype_TEXT:
312       Serial.printf("[%u]␣Messaggio␣ricevuto:␣%s\n", num,
            payload);
313       if (strcmp((char *)payload, "messaggio_specifico") == 0)
            {
314         webSocket.sendTXT(num, output);
315       }
316       break;
317     default:
318       break;
319   }
320 }
```

10.3 Source Code of ESP8266

Below is the source code of the Master device:

```
1
2  #include <ESP8266WiFi.h>        // Libreria per gestire la
       connessione WiFi
3  #include <WebSocketsClient.h>   // Libreria per gestire la
       connessione WebSocket
4  #include <ArduinoJson.h>        // Libreria per gestire i file
       JSON
5  #include <WiFiClientSecureBearSSL.h>
6  #include <PubSubClient.h>
7  #include <ESP8266WiFi.h>
8  extern "C" {
9  #include <espnow.h>
10 }
11
12 uint8_t esp32Address[] = { 0xD0, 0xEF, 0x76, 0x44, 0xCC, 0x3C
       };
13 String esp32IpAddress = "";
14
15 const char* ssid = "AIRNET_C1FDB0";
16 const char* password = "30800106";
17 // Sostituisci con i tuoi dati di rete
18
19
20 const char* mqttServer = "mqtt-dashboard.com";
21 const int mqttPort = 1883;
22 const char* mqttTopic = "testtopic/andrea";
23
24 // Crea un'istanza di WiFiClient e PubSubClient
25 WiFiClient espClient1;
26 PubSubClient mqttclient(espClient1);
27
28 static const char* awsEndpoint = "a1v69d0m0sa5ro-ats.iot.us-
       east-1.amazonaws.com";
29
```

```
30 static const char* rootCA = R"EOF(
31 -----BEGIN␣CERTIFICATE-----
32 MIIDQTCCAimgAwIBAgITBmyfz5m/
      jAo54vB4ikPmljZbyjANBgkqhkiG9w0BAQsF
33 ADA5MQswCQYDVQQGEwJVUzEPMA0GA1UEChMGQW1hem9uMRkwFwYDVQQDExBBbWF6
34 b24gUm9vdCBDQSAxMB4XDTE1MDUyNjAwMDAwMFoXDTM4MDExNzAwMDAwMFowOTEL
35 MAkGA1UEBhMCVVMxDzANBgNVBAoTBkFtYXpvbjEZMBcGA1UEAxMQQW1hem9uIFJv
36 b3QgQ0EgMTCCASIwDQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEBALJ4gHHKeNXj
37 ca9HgFB0fW7Y14h29Jlo91ghYPl0hAEvrAIthtOgQ3pOsqTQNroBvo3bSMgHFzZM
38 9O6II8c+6zf1tRn4SWiw3te5djgdYZ6k/oI2peVKVuRF4fn9tBb6dNqcmzU5L/
      qw
39 IFAGbHrQgLKm+a/sRxmPUDgH3KKHOVj4utWp+
      UhnMJbulHheb4mjUcAwhmahRWa6
40 VOujw5H5SNz/0egwLX0tdHA114gk957EWW67c4cX8jJGKLhD+
      rcdqsq08p8kDi1L
41 93FcXmn/6pUCyziKrlA4b9v7LWIbxcceVOF34GfID5yHI9Y/QCB/IIDEgEw+
      OyQm
42 jgSubJrIqg0CAwEAAaNCMEAwDwYDVR0TAQH/BAUwAwEB/
      zAOBgNVHQ8BAf8EBAMC
43 AYYwHQYDVR0OBBYEFIQYzIU07LwMlJQuCFmcx7IQTgoIMA0GCSqGSIb3DQEBCwUA
44 A4IBAQCY8jdaQZChGsV2USggNiMOruYou6r4lK5IpDB/G/
      wkjUu0yKGX9rbxenDI
45 U5PMCCjjmCXPI6T53iHTfIUJrU6adTrCC2qJeHZERxhlbI1Bjjt/
      msv0tadQ1wUs
46 N+gDS63pYaACbvXy8MWy7Vu33PqUXHeeE6V/
      Uq2V8viTO96LXFvKWlJbYK8U90vv
47 o/
      ufQJVtMVT8QtPHRh8jrdkPSHCa2XV4cdFyQzR1bldZwgJcJmApzyMZFo6IQ6XU
48 5MsI+yMRQ+
      hDKXJioaldXgjUkK642M4UwtBV8ob2xJNDd2ZhwLnoQdeXeGADbkpy
```

```
rqXRfboQnoZsG4q5WTP468SQvvG5
-----END␣CERTIFICATE-----
)EOF";

// Device Certificate
static const char* certificate = R"KEY(
-----BEGIN␣CERTIFICATE-----
MIIDWjCCAkKgAwIBAgIVALwoi/REHmdhhozVPZjBRb+
    q1T8HMA0GCSqGSIb3DQEB
CwUAME0xSzBJBgNVBAsMQkFtYXpvbiBXZWIgU2VydmljZXMgTz1BbWF6b24uY29t

IEluYy4gTD1TZWF0dGxlIFNUPVdhc2hpbmd0b24gQz1VUzAeFw0yNDA3MDUxNzI0

MjJaFw00OTEyMzEyMzU5NTlaMB4xHDAaBgNVBAMME0FXUyBJb1QgQ2VydGlmaWNh

dGUwggEiMA0GCSqGSIb3DQEBAQUAA4IBDwAwggEKAoIBAQDFDJbW3C3BwBav1+
    uU
7ulnK64Imm85HBPw+6hQduf6C+
    dfyCSoKbls2xTgr7ffMavSgJ87NYnuoj9pfZNd
9WCkqcjs+
    hEbJzrcR5wxA4QV7hdwvGwQXITH2b1Ocl2A1EXpKStFXuntTsqRxuZE
DOK5YBoful+7NU4mtsNCSVL7+Uq9XkZCvZYNGvA/5
    rmM4SnXI0X4eCcPnXWnMlP7
13vQIN6R8L1bmuapMS5QbgwodPWEWhUb+lUf5K7h6/VuDidnML79+
    dXSCb0bcvjw
75pqHdEskbEKem9w55lrudXq9tCQiZIY7b2PEUqB6cWYJETuz/B23DF1VL+
    tXYtv
HBwPAgMBAAGjYDBeMB8GA1UdIwQYMBaAFBiXLRC1QM81Cmo4OsAlmJwpjKe3MB0G

A1UdDgQWBBSKk8GR7guzq2fCzH/
    nwUO1GwVKLTAMBgNVHRMBAf8EAjAAMA4GA1Ud
DwEB/wQEAwIHgDANBgkqhkiG9w0BAQsFAAOCAQEAWEoMGuo48a/
    pAQerkCuBClT2
sud9laUy4vKa938iI/COdpJI8Obj5+1f/7+FxQmAinhfX/ibA6uPmeXsA2dQnsR
    +
iJd4+/qH6AziOlB4/6
    VVfLz9YGlscQKG2ViwBT6QAKTVRV71IQGTc8VKttJ0Jz2U
```

```
oxTS+Pw5jlDbYYi1DBFkCeJLYnNH1+Y0/sDfQ832hOcODCR+twjxektA2kX9+k7H
Ah3X2dGM9bXd4b+QR7h7C2gXpIh63IUdCucnbY9HoWyPjP3gsAiKNj1U3gYRPuPB
tXiUdJ6J2fM3crfKoWa7hhao8vZjPZDfj+9Yyf13Bh1vXadXBKuYkkBort3fQw==
-----END CERTIFICATE-----
)KEY";

// Device Private key
static const char* privateKey = R"KEY(
-----BEGIN RSA PRIVATE KEY-----
MIIEowIBAAKCAQEAxQyW1twtwcAWr9frlO7pZyuuCJpvORwT8PuoUHbn+gvnX8gk
qCm5bNsU4K+33zGr0oCfOzWJ7qI/aX2TXfVgpKnI7PoRGyc63EecMQOEFe4XcLxs
EFyEx9m9TnJdgNRF6SkrRV7p7U7KkcbmRA9CuWAaH7pfuzVOJrbDQklS+/lKvV5G
Qr2WDRrwP+a5jOEp1yNF+HgnD511pzJT+9d70CDekfC9W5rmqTEuUG4MKHT1hFoV
G/pVH+Su4ev1bg4nZzC+/fnV0gm9G3L480+aah3RLJGxCnpvcOeZa7nV6vbQkImS
GO29jxFKgenFmCRE7s/wdtwxdVS/rV2LbxwcDwIDAQABAoIBAGwe1EVg2tqNA+XI
nENENF2f4gZmdtDnTynTbC96jx7smTA9KZ/8BPpt267NvB6Dqrv+R2C9p8mAzTqc
5
NeRZE31u4IMVIaJqApmYJWkUD2YPtRlDwaLPXBLUUxSBGxEDXg6VrvhsIm9yfjj

IkYF+Z6t1KfEI/Mc9JzZe5pLkXUgBh6M3lkZJco4FnprWddz+o4sCvBLDawe83oF
luUAJAeXX2TiSCzddjiPfhw56oRW2dvsZhJ4ZcpBQNFejoelOtq16bLV8g9wDfNo

q0E0AjofG763su3/CMsGEgEg2EPdH5c284ffQnlm1vX2hqFBO/iZ26CllUbpmxgL
G7I6X8ECgYEA4k8Th6SDdiXW9Ru/RgqPC6UUZwunCY/X35guWKrqjChAhTUyveuO
```

```
92 3IoyMc2KTGF6A+JqfcyZUmID2vh+dfogSp1HsRIPVPXg5UF9bhUUkMSM1dHSvD
       +5
93 8zKWsrUNcN5IbC1YpgSt09u55bxS+2
       iwJnaKy0zRXbWNwG6bLtzreMMCgYEA3ubI
94 D7uBCbma4p0127Ypqui2s3CPv6XXBDJ9OfHvqZtEBch71x8ig4APDmy8LSNKO
       +91
95 0CgJMEjZLJ/oLG3sItceKMwyOm3FbFumKR2Vau8yK/M9Q6/
       p11oseRYa3zxBP0fd
96 S/6zrhEh2yQlw5SaAxAQJkkSlmEwDzyqL/CE0sUCgYB/
       H8hqs6EdJxhey11grOG5
97 utuBHuyP1HuBIG04ZZonbR0BNubwBHhVrlOP4lPDoCX4NKF5VQrzWhesU1ZGU2D8

98 SOlSsEBoi03vuiAJTFInGCG8oobsNCfyUwKQFGAefN7V1YsshwhWL+
       F3CZjbnO0G
99 TiSdGuOo7ilYZUlKw9KKHQKBgBi8kc01KM/
       UejzU2aTFZYBjDQuC3WEOXXtIwx7w
100 G4G+CmF960hnWyQuzPzz0jpMJUvbej6cgtCJ9Rf/svtjQ4ZmSyGJ77UOQ4+
       P6DO9
101 5
       bwVSYMZHl5polDU4AScEGVfwXntVsC9RmF140T3kP1Qe3sFiFVHXLm1lWjLebOS

102 RIplAoGBANDDqqhNLdnYycaTanN1S3b82bPNUZX5roSb1bEIF4NM6Qo/
       ZT9K8Dlb
103 VBSNh4M6c8wgt5Sm9swZpTXbXQrLUkdrdwqk8GCfE96qyxqHnqptAhSZAmPUvgSf

104 4YFIhqbUt5kNP7jG1Kj23LzhB9OEDnChuAB0FSNjmAG+6c3tA9VG
105 -----END␣RSA␣PRIVATE␣KEY-----
106 )KEY";
107
108 BearSSL::WiFiClientSecure espClient;
109 PubSubClient client(awsEndpoint, 8883, espClient);
110 WebSocketsClient webSocket;
111
112 // Gestiamo il tempo di report di ESP8266
113 unsigned long lastTime = 0;
114 unsigned long timerDelay = 5000;               // invia un
       messaggio ogni 5 secondi
```

```
115 const unsigned long minTimerDelay = 625;       // valore minimo
    del timerDelay
116 const unsigned long defaultTimerDelay = 5000;  // valore
    predefinito del timerDelay
117
118 int battiti_cardiaci[10];     // array per memorizzare le
    misurazioni del battito cardiaco
119 int passi[10];                // array per memorizzare le
    misurazioni dei passi
120 int count = 0;                // contatore per le misurazioni
121 unsigned long startTime = 0;  // tempo di inizio della raccolta
    delle misurazioni
122 bool allarmeAttivo = false;   // stato dell'allarme
123
124
125
126 void webSocketEvent(WStype_t type, uint8_t* payload, size_t
    length) {
127   switch (type) {
128     case WStype_CONNECTED:
129       Serial.printf("[WSc] Connesso al server: %s\n", payload);
130       break;
131     case WStype_TEXT:
132       Serial.printf("[WSc] Messaggio ricevuto: %s\n", payload);
133
134       // Analizza il payload JSON
135       DynamicJsonDocument doc(2048);  // aumentato la
          dimensione del buffer
136       DeserializationError error = deserializeJson(doc, payload
          );
137
138       if (error) {
139         Serial.print(F("deserializeJson() failed: "));
140         Serial.println(error.c_str());
141         return;
142       }
143
```

```
144       // Verifica se l'indice    nei limiti
145       if (count < 10) {
146         battiti_cardiaci[count] = doc["BattitoCardiaco"];  //
              memorizza la misurazione del battito cardiaco
147         passi[count] = doc["PassiEffettuati"];             //
              memorizza la misurazione dei passi
148         Serial.printf("Battiti␣cardiaci:␣%d,␣Passi:␣%d\n",
              battiti_cardiaci[count], passi[count]);
149
150         // Se    la prima misurazione, inizia a misurare il
              tempo
151         if (count == 0) {
152           startTime = millis();
153         }
154         count++;
155       }
156
157       // Se abbiamo raccolto 10 misurazioni, calcola il tempo
           impiegato e i passi al secondo
158       if (count == 10) {
159         unsigned long endTime = millis();
160         unsigned long timeTaken = endTime - startTime +
              timerDelay;  // tempo totale in millisecondi
161         float timeTakenInSeconds = timeTaken / 1000.0;
                            // converte il tempo in secondi
162
163         doc["Allarme"] = "Nessun␣Allarme";
164
165         // Calcola il totale dei passi
166         int totalPassi = passi[9] - passi[0];
167
168         float passiPerSecondo = totalPassi / timeTakenInSeconds
              ;  // calcola i passi al secondo
169         Serial.printf("Tempo␣impiegato␣per␣raccogliere␣10␣
              misurazioni:␣%lu␣ms\n", timeTaken);
170         Serial.printf("Totale␣passi:␣%d,␣Passi␣al␣secondo:␣%.2f
              \n", totalPassi, passiPerSecondo);
```

```
171
172         // Calcola la media dei battiti cardiaci
173         int sommaBattiti = 0;
174         for (int i = 0; i < 10; i++) {
175           sommaBattiti += battiti_cardiaci[i];
176         }
177         float mediaBattiti = sommaBattiti / 10.0;
178         Serial.printf("Media battiti cardiaci: %.2f\n",
                mediaBattiti);
179
180         bool nuovoAllarme = false;
181
182         if (mediaBattiti > 100 && passiPerSecondo < 2) {
183           // Crea un oggetto JSON con una stringa
184           doc["Allarme"] = "Probabile Tachicardia";
185           nuovoAllarme = true;
186         }
187
188         if (mediaBattiti < 50 && totalPassi != 0) {
189           // Crea un oggetto JSON con una stringa
190           doc["Allarme"] = "Probabile Brachicardia";
191           nuovoAllarme = true;
192         }
193
194         // Calcola la deviazione standard dei battiti cardiaci
195         float sommaDifferenzeQuadrato = 0;
196         for (int i = 0; i < 10; i++) {
197           sommaDifferenzeQuadrato += pow(battiti_cardiaci[i] -
                mediaBattiti, 2);
198         }
199         float deviazioneStandardBattiti = sqrt(
               sommaDifferenzeQuadrato / 10);
200         Serial.printf("Deviazione standard battiti cardiaci:
               %.2f\n", deviazioneStandardBattiti);
201
202         if (deviazioneStandardBattiti > 20 && passiPerSecondo <
              2) {
```

```
203           doc["Allarme"] = "Probabile Aritmia";
204           nuovoAllarme = true;
205         }
206
207         if (nuovoAllarme) {
208           // Dimezza il timerDelay se non ha gi  raggiunto il
                valore minimo
209           if (timerDelay > minTimerDelay) {
210             timerDelay = max(timerDelay / 2, minTimerDelay);
211           }
212           allarmeAttivo = true;
213         } else if (allarmeAttivo) {
214           // Ripristina il timerDelay al valore predefinito se
                non ci sono pi  allarmi
215           timerDelay = defaultTimerDelay;
216           allarmeAttivo = false;
217         }
218
219         // Serializza l'oggetto JSON e stampalo a video
220         String output;
221         if (serializeJson(doc, output) == 0) {
222           Serial.println("Failed to serialize JSON");
223         } else {
224           Serial.println(output);
225           if (client.connected()) {
226             client.publish("iot/health", output.c_str());
227           }
228         }
229         count = 0;  // resetta il contatore per una nuova
              raccolta
230       }
231       break;
232   }
233 }
234
235
236
```

```
237 void onDataSent(uint8_t* mac_addr, uint8_t sendStatus) {
238   Serial.print("Messaggio inviato: ");
239   Serial.println(sendStatus == 0 ? "Successo" : "Fallito");
240 }
241
242 void onDataRecv(uint8_t* mac_addr, uint8_t* incomingData,
      uint8_t len) {
243   // Converti i dati ricevuti in una stringa e salva in
        esp32IpAddress
244   esp32IpAddress = String((char*)incomingData).substring(0, len
        );
245
246   Serial.print("Indirizzo IP ricevuto da ESP32: ");
247   Serial.println(esp32IpAddress);
248
249   webSocket.begin(esp32IpAddress, 80);
250   webSocket.onEvent(webSocketEvent);
251 }
252
253
254
255 void setup_wifi() {
256   delay(10);
257   WiFi.begin(ssid, password);
258   while (WiFi.status() != WL_CONNECTED) {
259     delay(500);
260     Serial.print(".");
261   }
262   Serial.println("WiFi connected");
263   Serial.print("IP address: ");
264   Serial.println(WiFi.localIP());
265 }
266
267 void reconnect() {
268   while (!client.connected()) {
269     Serial.print("Attempting MQTT connection...");
270     if (client.connect("Esp32")) {
```

```
271       Serial.println("connected");
272     } else {
273       Serial.print("failed, rc=");
274       Serial.print(client.state());
275       Serial.println(" try again in 5 seconds");
276       delay(5000);
277     }
278   }
279 }
280
281 void reconnectmqtt() {
282   // Loop fino a quando non siamo connessi
283   while (!mqttclient.connected()) {
284     Serial.print("Attempting MQTT connection...");
285
286     // Crea un client ID casuale
287     String mqttclientId = "ESP8266Client-";
288     mqttclientId += String(random(0xffff), HEX);
289
290     // Prova a connetterti
291     if (mqttclient.connect(mqttclientId.c_str())) {
292       Serial.println("connected");
293
294       // Iscrivi ai topic desiderati
295       mqttclient.subscribe(mqttTopic);  // Sostituisci con il
              tuo topic
296     } else {
297       Serial.print("failed, rc=");
298       Serial.print(mqttclient.state());
299       Serial.println(" try again in 5 seconds");
300
301       // Attendi 5 secondi prima di ritentare
302       delay(5000);
303     }
304   }
305 }
306
```

```
307 void callback(char* topic, byte* payload, unsigned int length)
    {
308   Serial.print("Message␣arrived␣on␣topic:␣");
309   Serial.print(topic);
310   Serial.print(".␣Message:␣");
311
312   // Stampa il messaggio ricevuto
313   for (int i = 0; i < length; i++) {
314     Serial.print((char)payload[i]);
315   }
316   Serial.println();
317
318   DynamicJsonDocument doc(2048);  // aumentato la dimensione
        del buffer
319   DeserializationError error = deserializeJson(doc, payload);
320
321   if (error) {
322     Serial.print(F("deserializeJson()␣failed:␣"));
323     Serial.println(error.c_str());
324     return;
325   }
326
327   // Verifica se l'indice    nei limiti
328   if (count < 10) {
329     battiti_cardiaci[count] = doc["BattitoCardiaco"];  //
          memorizza la misurazione del battito cardiaco
330     passi[count] = doc["PassiEffettuati"];             //
          memorizza la misurazione dei passi
331     Serial.printf("Battiti␣cardiaci:␣%d,␣Passi:␣%d\n",
          battiti_cardiaci[count], passi[count]);
332
333     // Se    la prima misurazione, inizia a misurare il tempo
334     if (count == 0) {
335       startTime = millis();
336     }
337     count++;
338   }
```

```
339
340   // Se abbiamo raccolto 10 misurazioni, calcola il tempo
          impiegato e i passi al secondo
341   if (count == 10) {
342     unsigned long endTime = millis();
343     unsigned long timeTaken = endTime - startTime + timerDelay;
            // tempo totale in millisecondi
344     float timeTakenInSeconds = timeTaken / 1000.0;
                        // converte il tempo in secondi
345
346     doc["Allarme"] = "Nessun Allarme";
347
348     // Calcola il totale dei passi
349     int totalPassi = passi[9] - passi[0];
350
351     float passiPerSecondo = totalPassi / timeTakenInSeconds;
          // calcola i passi al secondo
352     Serial.printf("Tempo impiegato per raccogliere 10
          misurazioni: %lu ms\n", timeTaken);
353     Serial.printf("Totale passi: %d, Passi al secondo: %.2f\n",
            totalPassi, passiPerSecondo);
354
355     // Calcola la media dei battiti cardiaci
356     int sommaBattiti = 0;
357     for (int i = 0; i < 10; i++) {
358       sommaBattiti += battiti_cardiaci[i];
359     }
360     float mediaBattiti = sommaBattiti / 10.0;
361     Serial.printf("Media battiti cardiaci: %.2f\n",
            mediaBattiti);
362
363     bool nuovoAllarme = false;
364
365     if (mediaBattiti > 100 && passiPerSecondo < 2) {
366       // Crea un oggetto JSON con una stringa
367       doc["Allarme"] = "Probabile Tachicardia";
368       nuovoAllarme = true;
```

```
369     }
370
371     if (mediaBattiti < 50 && totalPassi != 0) {
372       // Crea un oggetto JSON con una stringa
373       doc["Allarme"] = "Probabile Brachicardia";
374       nuovoAllarme = true;
375     }
376
377     // Calcola la deviazione standard dei battiti cardiaci
378     float sommaDifferenzeQuadrato = 0;
379     for (int i = 0; i < 10; i++) {
380       sommaDifferenzeQuadrato += pow(battiti_cardiaci[i] -
            mediaBattiti, 2);
381     }
382     float deviazioneStandardBattiti = sqrt(
           sommaDifferenzeQuadrato / 10);
383     Serial.printf("Deviazione standard battiti cardiaci: %.2f\n
           ", deviazioneStandardBattiti);
384
385     if (deviazioneStandardBattiti > 20 && passiPerSecondo < 2)
           {
386       doc["Allarme"] = "Probabile Aritmia";
387       nuovoAllarme = true;
388     }
389
390     if (nuovoAllarme) {
391       // Dimezza il timerDelay se non ha gi  raggiunto il
            valore minimo
392       if (timerDelay > minTimerDelay) {
393         timerDelay = max(timerDelay / 2, minTimerDelay);
394       }
395       allarmeAttivo = true;
396     } else if (allarmeAttivo) {
397       // Ripristina il timerDelay al valore predefinito se non
            ci sono pi  allarmi
398       timerDelay = defaultTimerDelay;
399       allarmeAttivo = false;
```

```
400     }
401
402     // Serializza l'oggetto JSON e stampalo a video
403     String output;
404     if (serializeJson(doc, output) == 0) {
405       Serial.println("Failed to serialize JSON");
406     } else {
407       Serial.println(output);
408       if (client.connected()) {
409         client.publish("iot/health", output.c_str());
410       }
411     }
412     count = 0;  // resetta il contatore per una nuova raccolta
413   }
414 }
415
416
417
418 void setup() {
419   Serial.begin(115200);
420   WiFi.mode(WIFI_STA);
421
422   if (esp_now_init() != 0) {
423     Serial.println("Errore nell'inizializzazione di ESP-NOW");
424     return;
425   }
426
427   esp_now_set_self_role(ESP_NOW_ROLE_COMBO);
428   esp_now_register_send_cb(onDataSent);
429   esp_now_register_recv_cb(onDataRecv);
430   esp_now_add_peer(esp32Address, ESP_NOW_ROLE_COMBO, 1, NULL,
        0);
431
432
433   WiFi.begin(ssid, password);
434   while (WiFi.status() != WL_CONNECTED) {
435     delay(1000);
```

```
436     Serial.println("Connessione␣WiFi...");
437   }
438   Serial.println("Connesso!");
439
440   setup_wifi();
441
442   Serial.print("Resolving␣hostname:␣");
443   Serial.println(awsEndpoint);
444
445   IPAddress ip;
446   if (WiFi.hostByName(awsEndpoint, ip)) {
447     Serial.print("MQTT␣broker␣IP␣address:␣");
448     Serial.println(ip);
449   } else {
450     Serial.println("Failed␣to␣resolve␣hostname");
451   }
452
453   // Set up the client (same as before)
454   espClient.setInsecure();
455   espClient.setClientRSACert(new BearSSL::X509List(certificate)
        , new BearSSL::PrivateKey(privateKey));
456
457
458   mqttclient.setServer(mqttServer, mqttPort);
459   mqttclient.setCallback(callback);
460
461   WiFi.setSleepMode(WIFI_MODEM_SLEEP);
462 }
463
464 void loop() {
465
466   if (!mqttclient.connected()) {
467     reconnectmqtt();
468   }
469
470   if ((millis() - lastTime) > timerDelay) {
471     // invia un messaggio
```

```
472     webSocket.sendTXT("messaggio_specifico");
473     lastTime = millis();
474       Serial.println("Going to sleep (Modem-Sleep)...");
475     // Gestisci la connessione e i messaggi MQTT
476   }
477
478   if (!client.connected()) {
479     reconnect();
480   }
481   client.loop();
482   mqttclient.loop();
483
484
485   webSocket.loop();
486
487
488
489
490   // Nel loop gestiamo il tempo in cui deve essere mandato il
        messaggio di trigger ad ESP8266
491 }
```

10.4 Source Code of AWS Lambda

Below is the source code of the AWS Function:

```
1
2 import json
3 import boto3
4 from statistics import mean
5 from decimal import Decimal
6 from datetime import datetime
7
8 iot_data = boto3.client('iot', region_name='us-east-1')
9 dynamodb = boto3.resource('dynamodb', region_name='us-east-1')
10 cloudwatch = boto3.client('cloudwatch', region_name='us-east-1'
     )
```

```
11
12 table_name = 'Mean_Output'
13 table = dynamodb.Table(table_name)
14
15 sns = boto3.client('sns', region_name='us-east-1')
16
17 topic_arn = 'arn:aws:sns:us-east-1:339713093591:mytopic'
18
19 def lambda_handler(event, context):
20     for record in event['records']:
21
22         passi_effettuati = record['PassiEffettuati']  if '
              PassiEffettuati' in record else None
23         kcal_consumate = record['KcalConsumate'] if '
              KcalConsumate' in record else None
24         battito_cardiaco = record['BattitoCardiaco'] if '
              BattitoCardiaco' in record else None
25         pressione_sistolica = record['PressioneSistolica'] if '
              PressioneSistolica' in record else None
26         pressione_diastolica = record['PressioneDiastolica'] if
               'PressioneDiastolica' in record else None
27         allarme = record['Allarme'] if 'Allarme' in record else
               None
28
29         now = datetime.now()
30         timestamp_str = now.strftime("%Y-%m-%d %H:%M:%S")
31
32         table.put_item(
33             Item={
34                 'id': timestamp_str,
35                 'passi_effettuati': Decimal(passi_effettuati)
                      if isinstance(passi_effettuati, float) else
                      passi_effettuati,
36                 'kcal_consumate': Decimal(kcal_consumate) if
                      isinstance(kcal_consumate, float) else
                      kcal_consumate,
37                 'battito_cardiaco': Decimal(battito_cardiaco)
```

```
                    if isinstance(battito_cardiaco, float) else
                    battito_cardiaco,
38              'pressione_sistolica': Decimal(
                    pressione_sistolica) if isinstance(
                    pressione_sistolica, float) else
                    pressione_sistolica,
39              'pressione_diastolica': Decimal(
                    pressione_diastolica) if isinstance(
                    pressione_diastolica, float) else
                    pressione_diastolica,
40              'allarme': allarme
41
42          }
43
44      )
45
46      if pressione_sistolica is not None and
            pressione_diastolica is not None:
47          if 140 <= pressione_sistolica <= 159 or 90 <=
                pressione_diastolica <= 99:
48              sns.publish(
49                  TopicArn=topic_arn,
50                  Message='Attenzione! Ipertensione di Grado
                        1'
51              )
52
53          elif 160 <= pressione_sistolica <= 179 or 100 <=
                pressione_diastolica <= 109:
54              sns.publish(
55                  TopicArn=topic_arn,
56                  Message='Attenzione! Ipertensione di Grado
                        2'
57              )
58
59          elif pressione_sistolica >= 180 or
                pressione_diastolica >= 110:
60              sns.publish(
```

```
61                      TopicArn=topic_arn,
62                      Message='Attenzione! Ipertensione di Grado
                            3'
63                  )
64
65              elif pressione_sistolica >= 140 and
                    pressione_diastolica < 90:
66                  sns.publish(
67                      TopicArn=topic_arn,
68                      Message='Attenzione! Ipertensione sistolica
                            isolata'
69                  )
70
71              elif pressione_sistolica < 140 and
                    pressione_diastolica >= 90:
72                  sns.publish(
73                      TopicArn=topic_arn,
74                      Message='Attenzione! Ipertensione
                            diastolica isolata'
75                  )
76
77          if allarme is not None:
78              if allarme == 'Probabile Tachicardia':
79                  sns.publish(
80                      TopicArn=topic_arn,
81                      Message='Attenzione! Pericolo tachicardia'
82                  )
83
84              elif allarme == 'Probabile Brachicardia':
85                  sns.publish(
86                      TopicArn=topic_arn,
87                      Message='Attenzione! Pericolo Brachicardia'
88                  )
89
90              elif allarme == 'Probabile Aritmia':
91                  sns.publish(
92                      TopicArn=topic_arn,
```

```
93                      Message='Attenzione! Pericolo Aritmia'
94                  )
95
96      return {
97          'passi_effettuati': passi_effettuati,
98          'kcal_consumate': kcal_consumate,
99          'battito_cardiaco': battito_cardiaco,
100         'pressione_sistolica': pressione_sistolica,
101         'pressione_diastolica': pressione_diastolica,
102         'allarme': allarme
103     }
```