future internet

Article

Open-Source MQTT-Based End-to-End IoT System for Smart City Scenarios

Cristian D'Ortona †, Daniele Tarchi † and Carla Raffaelli *,†

Department of Electrical, Electronic and Information Engineering "Guglielmo Marconi", University of Bologna, 40136 Bologna, Italy; cristian.dortona@studio.unibo.it (C.D.); daniele.tarchi@unibo.it (D.T.)
* Correspondence: carla.raffaelli@unibo.it
† These authors contributed equally to this work.

Abstract: Many innovative services are emerging based on the Internet of Things (IoT) technology, aiming at fostering better sustainability of our cities. New solutions integrating Information and Communications Technologies (ICTs) with sustainable transport media are encouraged by several public administrations in the so-called *Smart City* scenario, where heterogeneous users in city roads call for safer mobility. Among several possible applications, recently, there has been a lot of attention on the so-called Vulnerable Road Users (VRUs), such as pedestrians or bikers. They can be equipped with wearable sensors that are able to communicate their data through a chain of devices towards the cloud for agile and effective control of their mobility. This work describes a complete end-to-end IoT system implemented through the integration of different complementary technologies, whose main purpose is to monitor the information related to road users generated by wearable sensors. The system has been implemented using an ESP32 micro-controller connected to the sensors and communicating through a Bluetooth Low Energy (BLE) interface with an Android device, which is assumed to always be carried by any road user. Based on this, we use it as a gateway node, acting as a real-time asynchronous publisher of a Message Queue Telemetry Transport (MQTT) protocol chain. The MQTT broker is configured on a Raspberry PI device and collects sensor data to be sent to a web-based control panel that performs data monitoring and processing. All the architecture modules have been implemented through open-source technologies. The analysis of the BLE packet exchange has been carried out by resorting to the Wireshark packet analyzer. In addition, a feasibility analysis has been carried out by showing the capability of the proposed solution to show the values gathered through the sensors on a remote dashboard. The developed system is publicly available to allow the possible integration of other modules for additional Smart City services or extension to further ICT applications.

Keywords: Internet of Things; Smart Cities; MQTT; BLE; Android; Raspberry PI; open-source

Citation: D'Ortona, C.; Tarchi, D.; Raffaelli, C. Open-Source MQTT-Based End-to-End IoT System for Smart City Scenarios. *Future Internet* **2022**, *14*, 57. https://doi.org/10.3390/fi14020057

Academic Editor: Joel J. P. C. Rodrigues

Received: 26 January 2022
Accepted: 10 February 2022
Published: 15 February 2022

1. Introduction

Internet of Things (IoT) and related technologies have been undergoing exponential growth for a few years, rising from 15 billion connected devices in 2015 up to 30 billion in 2020, and their numbers are intended to grow over the next decade [1]. IoT defines a network of devices, such as sensors, actuators, gateways and cloud services, interconnected among them with the aim of offering a specific service. Among several verticals, urban mobility is receiving a lot of attention, driven by the introduction of different micro-mobility options as an alternative urban mobility solution in the COVID-19 era [2]. Such mobility options require careful attention towards parameter-tracking for safety reasons.

The number of industrialized cities, which are more and more aware of eco-sustainability issues, has been constantly growing over the past few years [3]; this trend led government authorities to sponsor the adoption of eco-friendly means of transportation, such as e-bikes, electric scooters, Segways and so on. However, as the number of non-conventional means

of transportation has been rising exponentially, there has been a similar rise in the number of road accidents, involving so-called vulnerable road users (VRUs) [4]; therefore, the implementation of wearable technologies is remarkably beneficial in terms of user safety by helping people connect to a much broader intelligent ecosystem often referred to as a Smart City.

The main issue in an ubiquitous computer environment, such as IoT, is how impractical it is to impose standards that everyone has to comply to [5]; hence, one of the biggest constraints is to deal with closed source systems, which act as black boxes where the underlying structure is unknown. This leads to IoT solutions that are very hard to interact with, especially due to the nature of IoT being a very complicated heterogeneous network platform. An outright example is highlighted in [6], a survey performed on 26 different IoT clouds that can be grouped into 10 different genres of applications, which showed, as expected, the lack of heterogeneity management. Moreover, deploying a large number of wearable devices all over the city would have a large impact on the costs of such a system; therefore, we deem that, cost-wise, a full-stack solution including all the elements for complete control of the system from the source to the customer premises would be highly beneficial as it allows cutting off all those costs linked to third-party software and hardware, which an IoT solution usually relies on.

Another reason for cost reduction is the presence of an open-source approach where, once the community is fully engaged, the rate of progress can rapidly accelerate, and the project can potentially progress at a rate that can overcome closed-source development [1]. Applied to our scenario, the open-source approach allows developing a core system that can be further extended by developers and users also giving the specific requirements of any city. The proposed open-source approach could be the basis for further refinement as well as additional plug-in software and devices.

An End-to-End (E2E) Open-Source Proof-of-Concept (PoC) IoT architecture is proposed here, aiming to properly address the previously introduced issues. It is based on the integration of different open-source technologies, whose main purpose is to monitor, through the use of sensors connected to a micro-controller, the information related to the road users. The system has been implemented by integrating an ESP32, i.e., a System on Chip (SoC), equipped with sensors used to acquire environmental information that are able to interact through a Bluetooth Low Energy (BLE) connection with an Android device, which acts as an intermediate point that gathers the information produced by the sensors in a real-time asynchronous way. Among different messaging protocols for data exchange in IoT systems, we envisage the use of the Message Queue Telemetry Transport (MQTT) [7] protocol between the Android device and the intended users that require monitoring the road user's behavior. Similarly, the use of an Android device as an intermediate point between the wearable device and the rest of the network can be considered a viable option since we can rely on its presence with any user in an urban environment. Experiments on MQTT application protocol to support data acquisition and transfer in the cloud has been performed in the past [8] using a Linux operating system in a fixed workstation. Here, a mobile scenario is considered where the Android device acts as a publisher with the role of sending data acquired by the external sensor and internal sensors. The subscriber is here implemented through a dashboard deployed in the customer premises. In order to deploy the MQTT communication flow, a broker has also been implemented by resorting to a Raspberry PI node. Proper wireless communication links have been considered for allowing the interconnection between the publisher(s), the broker and the subscriber, even if any broadband and narrow-band solution can be used. The main novelties of the paper can be summarized in:

- A perfectly functional PoC of an E2E system for message delivering in a Smart City scenario has been released;
- The PoC is composed of COTS technologies; thanks to this, their integration allows having a low-cost, deployable solution for different Smart City applications;

- The PoC is based on open-source technologies, allowing its expansion with other components and services through simple open interfaces by any interested user;
- The integration of an Android-based app into the architecture allows the system to be potentially used by a large number of people, leading to a widely used echo-system;
- The modularity of the architectural design, along-with the open-source approach of the different components, allows a rapid evolution of the system toward other potential application scenarios.

The paper is structured in such a way. In Section 2, we enforce the rationale behind our solution by analyzing the most important solutions proposed up to now in practical implementations and by conducting a literature survey. Then, in Section 3, the high-level description of the architecture is presented, focusing on the main functional aspects of each node. Following this, in Section 4, the description of the PoC is given by sticking around the technological characteristics of each element considered in the system. Finally, some feasibility results are provided, showing the effectiveness of the proposed solution, in Section 5, and a final discussion is conducted in Section 6.

2. Technological Background

In this section, we review some of the most impacting implementations and proposed solutions that aim to solve a similar problem to the one we are considering.

In 2020, the 5G Automotive Association, an industrial-based association aiming at bridging automotive and telecommunication industries, released a White Paper concerning the protection of VRUs [9]. The report identifies the following road-user types as vulnerable: pedestrians, cyclists (including eBikes), motorcyclists, road workers, wheelchair users, scooter, skateboard and Segway users. In particular, three main scenarios have been described where VRUs can gain from the presence of ICT. In VRU high-risk zones, drivers (or automated vehicles) are delivered warnings when they enter a high risk area, where there is a likely presence of many VRUs. Dedicated roadside infrastructure could play a vital role in disseminating warning messages to VRUs and vehicles as well. The second scenario instead focuses on interactive communications between VRUs and vehicles, where a negotiation between the VRU's device and a vehicle is performed. The third scenario is maybe considered the most frequent in the future and involves vehicles and smartphones. In this scenario, VRUs' devices and vehicles send out safety messages. It is clear from [9] how important the use of smartphone technologies and its integration in a 5G/IoT echo-system will be for both VRUs and vehicles. However, in the same document, some technical enablers have been identified for dealing with VRU protection in the three above-described scenarios. It is clear how the role of smartphones, due to their embedded sensors, jointly with proper vehicular communication systems, play a vital role in protecting VRUs. This allows not only understanding the users' statuses but also detecting their relative position and predicting future directions.

An architecture enabling the integration of VRUs, referred to as Cooperative Information Technology Services, is discussed in [10]. The cooperative solution is supposed to be implemented in vehicles; supposedly, VRUs can use the service through their smartphone or through dedicated devices. In particular, two use cases are identified: (i) the possibility of knowing if VRUs are near potentially dangerous situations and (ii) the possibility of estimating potential collisions with other vehicles exploiting the prediction of their trajectories. The communication technologies for VRUs are discussed in [11], where the authors focus mainly on the vehicle-to-pedestrian paradigms. A framework for vehicle-to-pedestrian systems is here proposed, mainly focusing on different pre-crash scenarios enabling the possibility of understanding how different VRU groups can act.

Interestingly, the exploitation of user devices, in particular smartphones, constitutes a clear trend when managing the safety of VRUs. An early example is in [12], where a collision prediction algorithm is proposed, which exploits the communication between pedestrian and vehicles. Such an approach allows enhancing the classical approaches based on visibility among users. In particular, the authors propose a system where the devices'

and the vehicles' positions are broadcast reciprocally so as to minimize potential accidents. In [13], instead, the authors propose the Pedestrian-Oriented Forewarning System (POFS) that aims to protect distracted pedestrians. POFS exploits four possible smartphone states: screen, voice, screen-voice and silent. Based on these states, a collision prediction algorithm is proposed that is able to send alert messages to pedestrian users. Another example is in the 5GCAR project [14], where the authors propose a 5G radio-based positioning system jointly with a road users trajectory estimation. In such a system, an alert is sent every time the driver has to react to a potential warning. Each pedestrian was supposed to bring a smartphone with a protect me app, which is able to alter any warning situation. In [15], the authors propose a VRU warning system, where users exchange among themselves warning messages about potential issues they may encounter when in motion. Such messages are exchanges through commercial smartphones. In [16], the authors consider the possibility of detecting VRUs outside the field of view through wireless communications by considering BLE and WiFi Direct. Apart from the smartphone, the integration of sensor nodes has been considered a promising approach. As an example, in [17], the authors propose the Cooperative Safety System for Vulnerable Road Users (CS4VRU) architecture, which aims to alert cyclists about potential cars approaching them through wearable devices in the helmet. In addition, the developed smartphone app allows the cyclists to share their position through a VANET.

In contrast to the previous approaches, we aim at: (i) developing an E2E solution able to convey the VRU data acquired through external sensors and/or smartphone-embedded sensors toward a centralized premise; (ii) exploiting multiple wireless technologies so as to optimize different links; (iii) exploiting a pub-sub paradigm through the MQTT protocol, hence enabling a logical decoupling between source and destination; (iv) using open-source technologies in order to build a core system where additional plug-in can be added for enabling scenario-specific solutions.

3. End-to-End Architecture

In this section, we describe the proposed end-to-end architecture by focusing on its main functionalities from a high-level point of view. We bear on a reference IoT stack architecture in order to leverage our solution by coarsely analyzing each layer of the stack. In Section 4, a more in-depth overview of the adopted software and hardware technologies will be illustrated, together with a more detailed description of their implementation.

The architecture deployed in this paper is depicted in Figure 1 and consists of functional nodes, each with a specific role to allow the set-up of an open-source end-to-end workflow. Complete handling of message transfer from the data source to the destination of the sensed data is provided within a functional infrastructure for managing and monitoring the expected road-user activities.

The designed platform is intended to be used for acquiring data from any of the devices in the area. As an example, focusing on Intelligent Transport System applications, one can think of an alert about other vehicles arriving in the opposite direction, as well as feedback from the sensors in case any of them are measuring an out-of-range value. It is worth noticing that the designed platform is general, which allows implementing different Smart City applications and scenarios. The designed architecture is thus able to implement a complete E2E message delivery through proper hardware, software and networking solutions. To this aim, the selection of BLE and WiFi as wireless technologies for connecting the devices allows fully exploiting the characteristics of the data to be handled by each link. While the BLE allows transporting small-sized data packets, making it suitable for connecting sensors and smartphones, WiFi is the perfect choice for connecting nodes in a relatively longer distances. To this aim, 4G/5G technologies could also be used.

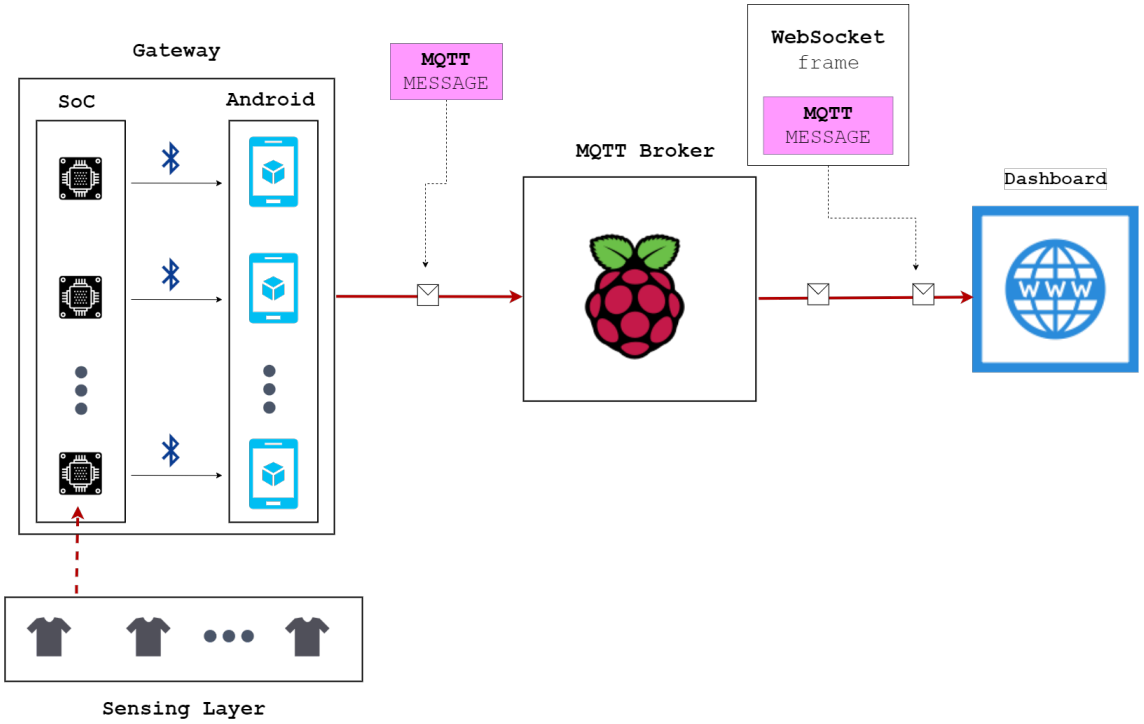

Figure 1. The end-to-end architecture of the proposed solution.

Due to the heterogeneity of IoT solutions and implementations, there is no standard approach in the definition of the system architecture and cloud platform [1]; hence, we resorted to open-source implementation, where each single function has been mapped on our reference IoT architecture.

The MQTT protocol is supposed to be used at the application layer for transmitting data.

The designed architecture is composed of four main functional layers:

- The **Sensing Layer** is responsible for implementing the cyber–physical interface, enabling the possibility of sensing physical data from the users. The sensed data can be human-related data (e.g., hearth monitor, position), vehicle-related data (e.g., battery charge, vehicle position) or environmental data (e.g., humidity, air temperature, pressure). Wearable sensors can be deployed in sportswear without needing to use expensive vehicles.
- The **Gateway** is responsible for collecting data from the sensing layer through any proprietary/custom protocol embedded in an IP packet to be delivered through the Internet. The gateway layer is implemented through two different nodes, one enabling the interface towards the sensing layer while the other, acting as a user's personal device, for transmitting data toward the final destination. In particular, the gateway acts as the MQTT publisher and is implemented on a mobile device that is presumably held by the users. This choice allows the approach to be available for any kind of road users, even in simple and cheap vehicles, such as bikes or simple pedestrians. The mobile device can be replaced, in principle, by other devices already installed in more complex vehicles, such as cars.
- The **MQTT Broker** acts as an intermediate point of the publisher/subscriber communication architecture. It is responsible of receiving any MQTT input from the gateways and notifying the subscriber about updated sensed data.
- The **Dashboard** acts as an MQTT subscriber, enabling the possibility of showing all the data collected by the gateway through internal sensors and at the Sensing layer.

The sensing layer allows the collection of data, such as health information of the user with the wearable device, and the surrounding environmental information by collecting the

sensors' data that have been implemented. Sensors and actuators, in fact, are the primary sources of information in an IoT system and provide the data that will then be processed by the device, e.g., a micro-controller, they are connected to.

According to this view, we have implemented a general-purpose embedded system for wearable devices through the use of the ultra-low power SoC Esp32 produced by the Espressif System [18]. Thus, we do not have to consider the type of wearable that will be used by the provider of the service; rather, we focused on the development of a general purpose underlying the infrastructure that can be later used regardless of the kind of wearable adopted. In fact, the advantage of having an open-source system is not only letting people contribute to the development of such technology but also allowing companies to adjust the infrastructure according to their needs. Such a modular design permits having a standardized infrastructure among companies that would use the proposed architecture, allowing the implementation of different wearable devices with different hardware and sensors, which are tailored around the needs of that particular company, making this system a fully heterogeneous IoT solution.

These information streams, gathered through the sensors, will then be sent to a smartphone—in this case, an Android device—through a BLE connection. The role of the Android device is to provide an edge gateway that allows us to offload the computational tasks directly to the mobile device rather than transmitting them to the cloud. This solution has been already considered a viable option when Android devices are considered as an edge processing node [19]. It is worth noticing that, nowadays, every user likely carries a smartphone, giving the proposed solution short-term practical applicability. Considering this, an Android app is developed that allows the user to gather the information acquired by the sensors on the wearable device and provide a general view on the stats of their activity. The app allows the user to connect to the embedded system mounted on the wearable device through BLE connections, which is a common communication technology for constrained devices with very limited battery life. The software developed for this application is fully available as open-source, resulting in the possibility of further development with additional features in order to meet the needs of the service provider [20]. Moreover, we deem that having control over the information of a certain service, the user is important in order to prevent and offer fast first-aid in the case of a road accident or any issue that might harm the VRU who is using that particular service deployed through the use of our architecture. To this aim, we developed a control panel that allows visualizing and monitoring those stats gathered through the sensors of every user by using a simple web-service that acts as a control dashboard. This web-service, which is still part of the PoC, is able to acquire the sensors' data from the smartphone device through the use of the communication protocol MQTT.

It has to be highlighted that the selection of MQTT as the application layer protocol allows decoupling the transmitter, i.e., the publisher, and the receiver, i.e., the subscriber. A pub/sub paradigm allows implementing an efficient way for delivering messages through an intermediate node, even in those situations where one of the two parts may be disconnected or temporarily unavailable, allowing, at the same time, to provide an efficient way for delivering messages. The MQTT connection is implemented thanks to the broker that is running on a Raspberry Pi 3B [21]. For what concerns the Broker software implementation, we use Mosquitto [22], an open-source broker developed by Oracle, while the Android device and the web-service act as MQTT clients. The developed Android app also integrates the MQTT connection through the use of the Paho library [23], developed by Oracle, and publishes the information acquired by the sensors on predefined topics. The control panel acts as a MQTT subscriber by subscribing to the topics the Android devices are publishing; this capability has been implemented through the use of the Paho JavaScript library. However, due to the HTTP definition, it is impossible to have a direct link for data exchange between the broker and the web-service; hence, we resorted to the use of web-sockets.

4. Proof-of-Concept

In this section, the implementation of each node of the architecture is discussed by analyzing the used hardware together with the developed software in order to provide a detailed explanation of the PoC.

4.1. Gateway Interface to Sensing Layer: The ESP32 Platform

As stated before, the first goal of the proposed IoT architecture is to integrate wearable devices, requiring the development of the software that manages the data, as well as the connections of each node with the rest of the architecture, and a feasibility study of the hardware, in particular the SoC and the sensors.

We predicat that the microcontroller with the sensors will be worn by the users, providing a fully functional smart-device. With these premises, one of the issues we aimed to solve was determining which microcontroller would best fit the requirements that a wearable solution, due to the constrained sizes of the device, generally requires, even though it is difficult to have a system where all these requirements are met. In order to process the stream of data acquired by the sensors, we have decided to use the ultra-low power SoC ESP32 [18], having all the most updated state-of-the-art characteristics of low-power chips, such as clock gating, power modes and dynamic power scaling. The ESP32 SoC is a single 2.4 GHz WiFi and Bluetooth combo chip designed by the Taiwan Semiconductor Manufacturing Company (TSMC) ultra-low-power 40 nm technology, which mounts a Tensilica Xtensa LX6 dual-core microprocessor together with 448 KB of ROM, 520 KB of SRAM and 4 MB of flash memory. Moreover, by having a look at the architecture, the Ultra Low Power co-processor stands out, which is an FSM (Finite State Machine) designed to perform measurements using the Analog/Digital Converter (ADC) or external Inter-Integrated Circuit (I2C) sensors, while the main processor is in deep-sleep mode; this is particularly important as it is used to wake up the chip from its sleeping mode. In the literature, there are many use cases [24,25] where this SoC is used; such applications range from IoT solutions to much more complex systems, such as voice encoding or music streaming, where a huge quantity of resources are involved.

The main reason for choosing the ESP32 SoC is that it jointly implements Bluetooth and WiFi connections with a very efficient power management scheme, which permits drawing a current as low as 10 µA when it is used in sleep-mode; for what concerns the transmission of BLE packets, the datasheet defines 130 mA as the typical power consumption considering measurements taken with a 3.3 V supply at 25 °C of ambient temperature. This microcontroller's advantages make it a feasible chose for IoT devices;indeed, there are many examples of plug-and-play devices that are currently available on the market to purchase. Among others, it is worth citing the LilyGo platform, which is an open-source hardware Smart Watch based on the ESP32-PICO-D4 [26].

In addition to the ESP32 platform, we considered an external sensor device. In particular, we resorted to the BME280 [27], an integrated environmental sensor developed by Bosch Sensortech, offering the possibility of sensing the relative humidity, the barometric pressure and the ambient temperature. This chip, with its 8-pin metal-lid $2.5 \times 2.5 \times 0.93$ mm^3 LGA package, has mainly been developed for IoT devices, particularly wearable devices, where size and low power consumption are key design parameters. Current-wise, it is indeed perfectly feasible for low current consumption, and it has a 0.1 µA current consumption when it is in sleep-mode and 3.6 µA when it is in active-mode. In the developed prototype, we use a pre-build module based on the BME280, where it is integrated together with a Low Drop Out (LDO) Voltage Regulator and a I2C Voltage Level Translator.

The connection between the BME280 module and the ESP32 is implemented through the I2C connection interface, which allows having synchronous communication between the two devices and a data-stream with a bit rate equal to 100 kb/s. Regarding the I2C connection, the ESP32 works as a master and the sensor as a slave, and its I2C 7-bit address is `b'0111911x`, where the first 6-bits are fixed, but it is possible to choose the value of the least significant bit.

4.1.1. Software Implementation

In order to program the ESP32, we used the C++ programming language through the Arduino framework, which is a relatively easy and fast prototyping tool that allows us to experiment on our models before production, although most embedded system products ready for production implement the real-time operating system (FreeRTOS) kernel [28], as it is designed to be small and simple.

In order to enable a communication between the ESP32 and the BME280, we have used the drivers developed by Adafruit [29], which allow connecting the sensor and easily reading the values obtained from the measurement of temperature, pressure and humidity from its registers. The output of the readings has to be sent through a BLE connection to the Android gateway, and another library [30] has been used in order to manage this connection. In the case of the SoC, the three main BLE protocol layers (i.e., Application, Host and Controller) are implemented in the same chip in order to save space and have a more miniaturized device.

Before analyzing the implemented structure, it is important to introduce the BLE Generic Access Profile (GAP) and the BLE Generic Attribute Profile (GATT), and their role in the relationship between the two devices [31]. The GAP defines which of the two mechanisms are used by a BLE device for communicating with other devices, i.e., broadcasting or connecting. The GAP defines how BLE-enabled devices can make themselves available and how two devices can communicate directly with each other. The GATT instead defines the role of a specific device, which can be a client or a server. The client typically sends a request to the GATT server and can read and/or write attributes in the server. The server stores the attributes. Once the client makes a request, the server must make the attributes available.

The ESP32 acts as the GAP Peripheral, meaning that it transmits advertising packets with the aim of establishing a connection with the device; in this case, the Android smartphone, acting as the GAP Central, is constantly listening to advertising packets sent on air by nearby peripherals in order to connect to the correct one. The Broadcaster role for the SoC has been discarded, making it work as a BLE Beacon, due to possible security issues, e.g., eavesdropping, that someone would face while transmitting data without establishing a secure connection first. Regarding the GATT role, it can be assigned to a device regardless of its GAP role, and it is even interchangeable; in our case, the SoC works as the GATT Server during the whole time since it sends the data packets without receiving any. The GATT Client role has instead been assigned to the smartphone. The implemented GATT Server, hierarchically organizing all the attributes defined by the Attribute Protocol, is represented in Figure 2.

The GATT server has been implemented with two services. The *BME280_Service* contains the attributes labeled *Characteristic*, following the GATT convention, assigned to different sensors, while the *Hearth_Service* is foreseen to be assigned to a heart-rate monitor, although it is not implemented in the deployed PoC, and used as a future plug-in extension. The BME280 service has four characteristics corresponding to the temperature, humidity, pressure and altitude readings, where the latter, although not directly read by the sensor, can be calculated through the other readings' information. The heart-rate service has only one characteristic, which is connected to the reading of the heart-rate of the service user. In order to work with every attribute, the Universally Unique Identifier (UUID) should be defined, allowing to universally identify the attribute. Even though the BLE Special Interest Group (SIG) provides a list of short 16-bit or 32-bit UUIDs that are standardized and might be used in applications, in the considered system, we have decided to resort to custom UUIDs due to the custom and specific implementation; in this case, the full 128-bit UUID value should be used since we are not using the standard base UUID [32]. At the software level, we have defined each attribute by specifying the type, and hence service, characteristic or descriptor, then we assigned a corresponding UUID for each one, as previously defined, and the properties. In this case, the attributes should be read-only since we do not want the Android gateway to be able to overwrite them, we only want it to read their values corresponding to the sensors readings.

Figure 2. GATT Server Hierarchy.

In addition, we implemented the *Notify* property, as defined by the BLE Core Specification. Since we want to send the readings from the sensors asynchronously and in real-time to the GATT Client to avoid the client cycling asking the server for new readings, since it is not a power-efficient solution, the Notify property allows the server to automatically notify the client whenever there is a new sensor reading, and we have a flow that is different from the usual request/response pairs. However, for each characteristic, it must be implemented as a descriptor, as can be seen in Figure 2. For the purposes of this implementation, we resorted to the Client Characteristic Configuration Descriptor (CCCD), a GATT-defined descriptor, which works similarly to a switch, enabling or disabling server-initiated updates [33]. This function allows the client to decide whether or not it wants to receive automatic asynchronous updates from the server by writing the attribute's value, a two-bit field; the client will simply use a Write Request Attribute Protocol (ATT) packet, which set the least significant bit to 1, while the server will reply with a Write Response ATT packet. When the process is successful, the server will be able to send automatic updates.

4.2. Gateway Interface to the Broker: The Android Platform

IoT systems consist of heterogeneous interconnected devices, leading to an ever-growing demand for ubiquitous connectivity. Regardless of the implemented vertical, the IoT infrastructure has to manage several sensors, which may significantly differ either for connectivity or acquired data type. With such requirements, the role of a gateway in such architectures is crucial, as it represents the bridge connecting the sensing layer, composed of the different sensors with different characteristics, to the network layer.

The architecture proposed in this paper considers the gateway as a combination of two devices: the SoC and the Android device. The latter represents the mobile part and has the role of interfacing all the sensors of the wearable device to the network after the data have been acquired by the ESP32. This gateway layout permits offloading the heavy workload, which gateways would generally have to process, between the two nodes: the processing of the data from the sensors has been assigned to the SoC, while the bridge with the network layer is handled by the Android device.

The open-source application for managing the system running on the smartphone has been properly designed by us. The goal of the Android app is two-fold: (i) the implementation of the Bluetooth connectivity, connecting the smartphone with the SoC on the wearable device, and (ii) the implementation of the MQTT connectivity, which, instead, allows the user to connect to the external network.

In this section, we will briefly describe the main aspects of the Android implementation, analyzing how the Bluetooth connectivity has been implemented and then introducing the Paho library [23] used for the implementation of the MQTT protocol. The app has been developed in Java through the use of the Android Studio IDE and the minimum SDK, which has been set to Marshmellow (API 23). Android provides a standard Bluetooth stack that supports both classic Bluetooth and BLE (For major clarity, BLE support was introduced with the API 18 and enabled Android smartphones to communicate with devices, such as proximity sensors, light-bulbs, wearables.) By recalling the BLE protocol stack, it is clear that the Android device acts as an ATT role of Central, which means it will remain listening for advertising packets transmitted by the ESP32 in order to establish a secure connection with it through the bounding procedure. Conversely, for what concerns the GATT role, the Android smartphone works as a client since it receives and reads the payload of the Characteristics of the SoC GATT server, as introduced in Section 4.1.

The Android app has been organized in two Android Activities in order to be used by any type of device, even low-end terminals. The only requirement is to have installed Android API 23. In Figure 3, the graphic user interface (GUI) of the main Android Activity is reported. The user can interact with the app through the toolbar, which includes all the functionalities the application provides, such as the BLE connection to the GATT server, scanning of BLE devices nearby, data transmission to the MQTT broker and a button that allows the user to send SoS requests. Moreover, the GUI shows which wearable device the user is connected to and whether or not there is an ongoing connection, while a scroll-view has been used in order to display the values acquired by the sensors.

In the considered scenario, we used the BME280, which is able to jointly sense pressure and humidity data. In addition to this, the embedded Smartphone sensors are used. Moreover, it is worth noting that virtually any sensor could be plugged in, resorting to any of the standard interfaces we have used in our system. As an example, in the case of a VRU, we can also think of a Bluetooth-interfaced sensor able to measure the battery status of the electric bike/scooter, as well as the user's health status through a smartwatch device.

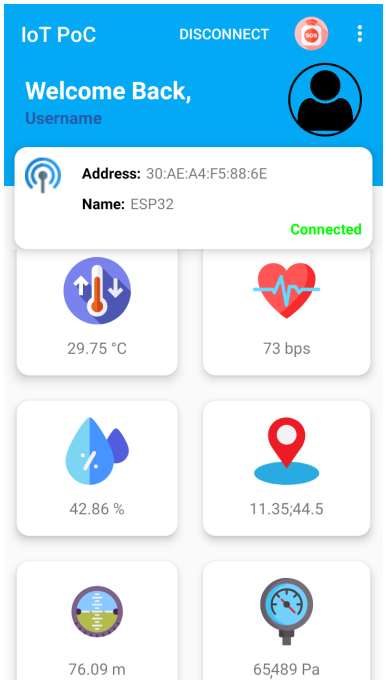

Figure 3. The Android App main Activity.

4.2.1. BLE Management

The first action to be performed in order to establish a Bluetooth connection between the two devices is a scan of the nearby BLE devices with the ATT role of peripherals.

This is achieved by interacting with the Android app toolbar, whose role is to display the list of the nearby BLE devices so that the user is able to pick the device they wish to connect to from the list. In order to initialize the scan, it is mandatory to declare in the `AndroidManifest.xml` file the correct permissions the Android needs in order to use the Bluetooth API. Moreover, since the API 23 Android requires the user to accept at run-time those permissions that are labeled as dangerous, the user has to allow the app to acquire the permissions regarding the location of the user since they are a mandatory requirement for the BLE API to work. Once the permissions have been granted, an object of the class `BluetoothAdapter()` is instantiated, which abstracts the Bluetooth Radio, integrated in the smartphone, and allows interacting with it through the software code.

The first operation allows checking if the device running the app supports BLE; in case it does not, the app will not allow the user to proceed further. This instance is used later to instantiate an object of the class `BluetoothLeScanner()`, which is used for all those operations regarding the scanning of BLE devices. Moreover, the Android allows implementing a white list of the scanned devices we might be interested in by discarding the others. Since we aim to connect to a small pool of devices the IoT wearables implementing the SoC programmed previously, we believe implementing a white list will allow the Android app to connect to only to the ESP32 SoCs on the wearable devices to prevent the Android app from connecting to other devices. Once the scan has been initialized, the results will be available in a callback function and the graphic interface is updated at run-time with all the nearby devices scanned previously being filtered.

In order to implement the Bluetooth communication, it is necessary to use multithreading techniques due to the asynchronous behavior of the communication; hence, we have a message queue in the Android with all the threads to be run. Once the user has chosen the correct wearable device to connect to, they are redirected back to the main activity where they will initialize the connection to the GATT server. The connection to the GATT server starts by instantiating an object of the class `BluetoothDevice()`, which is used to invoke the method `connectGatt(Context context, boolean autoConnect, BluetoothGattCallBack bluetoothGattCallBack)`. This allows the smartphone to automatically connect to the wearable devices, as long as they are bounded. In order to acquire the output of this method call, we have defined a callback function through the use of the Java anonymous inner class construct, which creates a class that implements a Java interface; hence, certain Abstract methods must be overridden as in Listing 1.

Listing 1. Gatt Callback Function.

```
BluetoothGattCallback gattCallBack = new BluetoothGattCallback() {
@Override
public void onConnectionStateChange(BluetoothGatt gatt, int status, int newState)
    {
super.onConnectionStateChange(gatt, status, newState);
...
}

@Override
public void onServicesDiscovered(BluetoothGatt gatt, int status) {
super.onServicesDiscovered(gatt, status);
...
}

@Override
public void onCharacteristicChanged(BluetoothGatt gatt,
    BluetoothGattCharacteristic characteristic) {
super.onCharacteristicChanged(gatt, characteristic);
...
}

@Override
public void onDescriptorWrite(BluetoothGatt gatt, BluetoothGattDescriptor
    descriptor, int status) {
```

```
super.onDescriptorWrite(gatt, descriptor, status);
...
}
}
```

The method **public void** onConnectionStateChange(BluetoothGatt gatt, **int** status, **int** newState) is used to check on the status of the Bluetooth connection between the two devices involved in the data exchange, indicating when a GATT client has connected/disconnected to/from a remote GATT server. In case the connection occurs, the method discoverServices() is invoked in order to acquire and sync with the list of remote services, characteristics and descriptors. This is an asynchronous operation that triggers the callback method **public void** onServicesDiscovered(Bluetooth gatt, **int** status) and, if the discovery was successful, the services can be retrieved by calling the function getServices().

As pointed out in Section 4.1.1, each BLE characteristic of the ESP32 implements the Notify property, allowing the client to be notified whenever a change in one of the BLE characteristics occurs. However, the client must explicitly express its desire to use such system by writing the CCCD of the characteristic it wishes to receive notifications from. In order to enable the property, we have to check, first, if the characteristic is present; in this case, the descriptor is retrieved, with the function getDescriptor(UUID uuid) and its GATT-defined 32-bit UUID equal to h'0x2902, which is the standard for characteristic configuration descriptors. Once the sync is completed, the method setValue(BluetoothGattDescriptor.ENABLE_NOTIFICATION_VALUE) must be called in order to modify the locally stored cached value of this descriptor. The new value assigned to the descriptor is BluetoothGattDescriptor.ENABLE_NOTIFICATION_VALUE, corresponding to a two-bit field equal to b'0x01, while, if we enable the indication property, the bit-field would have been equal to b'0x10.

The server should be aware of our intention to use that property; hence, the function writeDescriptor(BluetoothGattDescriptor descriptor) is called, and it writes the value of the locally stored cache onto the descriptor associated with the remote device. This function triggers the callback method **public void** onDescriptorWrite(BluetoothGatt gatt, BluetoothGattDescriptor descriptor, **int** status), which is part of the callback functions defined in Listing 1. Its role is to collect the result of a write operation concerning GATT descriptors; in our case, if the write request is successful, then, inside this callback function, the method setCharacteristicNotification(Descriptor descriptor, **boolean** enable) is updated, allowing the Android GATT client to asynchronously listen to GATT automatic server.

In case one of the characteristics, whose descriptor has been written in order to allow the notify property, changes, the server sends a notification that triggers the callback function onCharacteristichChanged(BluetoothGatt gatt, BluetoothGatCharacteristic characteristic), which contains the characteristic that has been updated and its new value.

In the app, the UI should be updated with the new sensor values whenever they are notified; this is accomplished by using a broadcast-receiver. Android apps can send or receive broadcast messages from the Android system similarly to the publish/subscribe design pattern we find in network protocols, such as MQTT. All the broadcasts sent will be routed to the activity that has been expressively declared through the use of specific intent filters. They can be used as a messaging system across apps and outside the normal user flow. In our case, the Bluetooth service we have implemented in the background allows sending custom broadcasts to the main Activity with the aim of updating the UI with the new sensor values.

4.2.2. Android MQTT Client

The developed Android app, as previously stated, is able to send sensor's updates received from the BLE GATT server, as well as other kinds of information regarding the user, to a web dashboard through the MQTT protocol. Such connection has been implemented on the Android side by integrating a further background service that enables the smartphone to work as an MQTT client. The events occurring in such connection are handled by the Paho library, which provides a quite easy and straightforward solution when it comes to MQTT integration in Android devices.

The connection with the broker and the publishing of the messages on the different topics occurs whenever the user interacts with the UI of the app and selects the option that starts the MQTT service in the background. Once the service is called, an instance of the class `MqttAsyncClient()` is created. It allows the client to initiate MQTT actions and then carry on working, while the MQTT actions are being completed in the background thread. Moreover, in order to check on the connection status, the interface `MqttCallBack()` is implemented, allowing the client to be notified when asynchronous events related to the client occur.

The values published by the app are those acquired through the BME280 sensor, i.e., temperature, humidity, pressure and altitude; in addition, an *SoS* message is published, which contains the GPS location of the user acquired through the smartphone GPS sensor. The MQTT messages are formatted with four possible topics, each one according to the MQTT specifications [7], having a PDU composed of three different fields: a fixed header, a variable header and a payload. The first four bits of the fixed header define the supported message types, and the remaining four bits of the first byte are used to define the different header flags, such as QoS (Quality of Service), DUP (duplicate) and RETAIN. In our case, the topics, whose payload contains one of the values read by the sensor encoded as a JSON object, have a QoS equal to zero in their 4-bit flag, where the QoS level defines how reliable the reception of the published message by the client subscribed to that specific topic is. In our case, a value equal to zero corresponds to the least QoS level; thus, there is no guarantee the message will be received by the subscribers subscribed to the topic; indeed, there is no acknowledgment packet sent, proving the publisher is aware of the reception of the message. The reason for selecting such QoS is due to its low resource-hungry requirements if compared to the other QoS levels; moreover, the loss of a packet would not be a problem as there is no essential information being sent.

The SoS request topic carries the position of the user at the time the request is sent in the payload, while the QoS level is equal to 1 in this case. A QoS level of 1, also referred to as *At Least Once*, is a quite reliable transmission method in the MQTT protocol, but, as a drawback, it is resource-expensive, and the message could be sent more than once. The SoS message is essential and must be received by the subscribers at all costs; thus, receiving it more than once would lead to no issue. Nonetheless, it is supposed to be infrequently published by the client compared to the other topics implemented; hence, spending more resources than usual would be acceptable. The higher reliability of this level of service is accomplished by a two-level handshake between the publisher and subscriber, allowing the message to be received at least once. Figure 4 depicts the general PDU exchange that occurs in the IoT system we implemented.

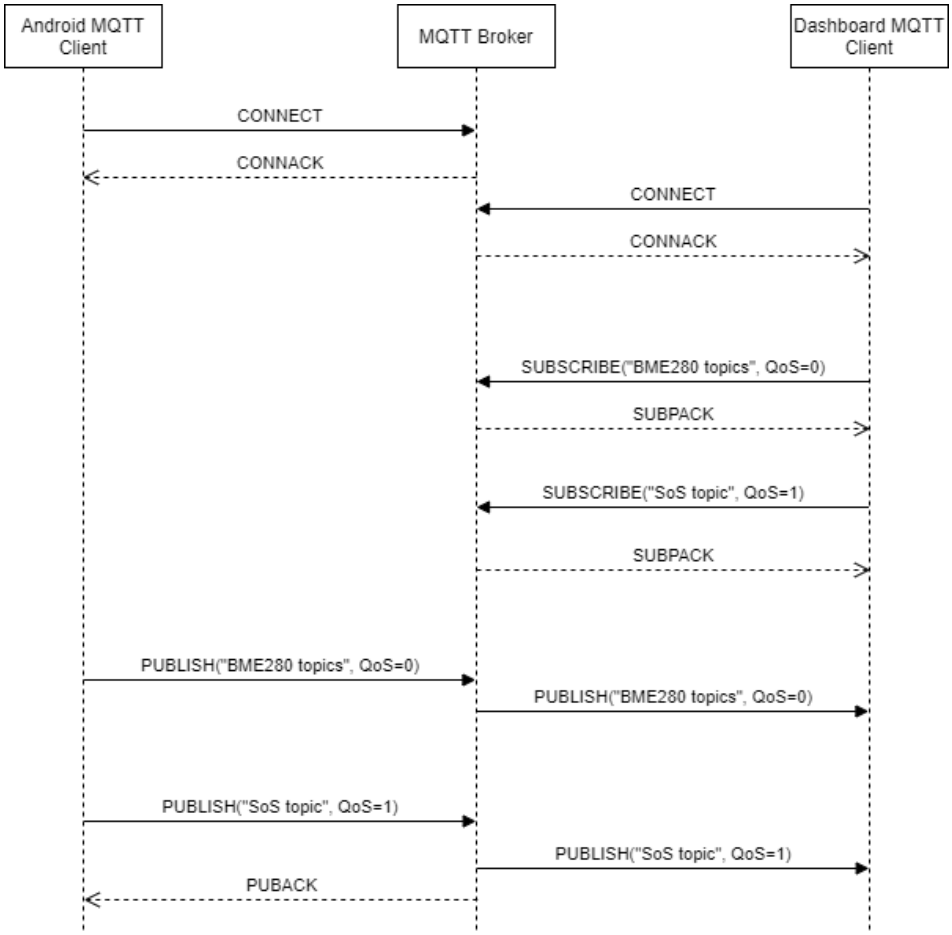

Figure 4. MQTT Packets Exchange.

4.3. Raspberry PI 3B+ Node

In the architecture we presented, the node in the middle, between the gateways and the dashboard, is a Raspberry Pi 3B+. It is a fairly low cost and portable computer developed by the Raspberry Pi Foundation. It is endowed with a 64-bit quad core processor running at 1.4 Ghz, dual-band 4.2 Ghz and 5 Ghz wireless LAN, Bluetooth 4.2/BLE, Ethernet connection and an extended 40-pin GPIO header, which allows interfacing external components [21]. It has been deployed with the Raspberry Pi OS, which allows updating it in order improve the stability, performance and security of the system.

The MQTT broker is actually installed on a Raspberry PI B3+ node. In a practical implementation, it is possible to think of several MQTT brokers that are supposed to be deployed in a Smart City scenario, e.g., on light poles or traffic lights, enabling the possibility of receiving MQTT messages from the vehicles in proximity through any wireless connection, e.g., IEEE 802.11x-based solutions. Despite the proposed solution being based on a pre-deployed MQTT approach, modern container/Virtual Machine-based approaches can be used. As an example, a Docker-based MQTT broker can be used for flexibly deploying MQTT brokers on devices.

4.3.1. Mosquitto

The logical role assigned to the Raspberry Pi node is the MQTT broker, which allows communication between different MQTT clients. Mosquitto is an open-source lightweight message broker that implements the MQTT protocol versions 5.0, 3.1.1 and 3.1 and is suitable for all devices from low-power single-board computers to full servers [22]. The rationale behind MQTT is its light weight; in fact, it is intended for devices with limited power capabilities, or when we have to deal with constrained and unreliable networks,

e.g., cellular networks, affected by a high packet loss rate [34]. These advantages make it a feasible solution for web-based data monitoring operations [35].

The MQTT clients are implemented on the Android smartphones running the developed app, acting as MQTT publishers by publishing the readings of the wearable device's sensors, as well as on a webservice dashboard, later illustrated in Section 4.4, acting as a MQTT subscriber.

The main role of the broker is to provide a way for routing the packets on a certain topic to the clients subscribed to that specific topic. However, the broker has several other responsibilities; one of those is authentication: in order to secure the connection of the IoT system, it is important not only to define the authentication credentials, such as a username or password, to be used by every client to sign into the broker but also establish an encrypted communication between the broker and the clients through TLS and SSL. In our case, we have created a custom configuration for Mosquitto, which let us define a password and username that will be used to log into the broker, but, more importantly, since we are dealing with a webservice client, we had to enable MQTT over websockets by defining the socket number the broker has to listen to as 9001.

4.4. Webservice Dashboard

As previously stated, one of the main features of the proposed architecture is the real-time monitoring of the statistics collected by the sensors mounted on the wearable devices. This enables whoever is using our E2E architecture, e.g., a certain company deploying their services, to supervise the activity that is being performed by the users of that certain service. This feature is particularly important in those applications targeting VRUs since they have a higher probability of being engaged in accidents, putting their life at risk. Hence, being able to have a real time and accurate pool of data helps provide a clearer overview of the statistics of one user's health, as well as the environmental ones. This knowledge is fundamental since it allows the provider to offer immediate first aid in case of an unwell user.

With such premises, we developed a webservice dashboard that enables receiving the sensors' data through the Android smartphone and the MQTT connection. Such dashboard has been implemented exploiting the Javascript functions inherited from the open-source Paho library, enabling a browser-based Javascript MQTT Client.

The developed web application relies on Websockets in order to allow bi-directional communication between the dashboard and the MQTT broker. It is worth noticing the HTTP, due to its architectural structure, which is not designed for real-time full duplex web applications [36]. Moreover, in IoT systems, especially those requiring real-time data exchange, latency is one of the main issues. In the case of HTTP, we would have to implement HTTP polling, where the client sends a request to the server every polling interval. Therefore, in order to use this method, we should be aware of the frequency used by the sensor for updating the data, which is unfeasible to acknowledge with an asynchronous architecture. By increasing the polling interval, we would also cause a high number of requests, which might be inconsistent, as the server might not have the value of the new sensors yet, leading to a non-optimized IoT solution. This latter problem might be solved with long polling, which efficiently handles the information push from servers to client by holding the client's request until there is a new sensor value available to send as a response rather than sending an empty response PDU. However, it has been demonstrated how the Websockets protocol is the overall best solution for full-duplex communications when packets have to travel long distances over congested networks [37]. MQTT over Websockets allows receiving the messages we have in a standard publish/subscribe paradigm in an environment, the web browser, where the defined paradigm is request/response, thus allowing the browser to leverage all MQTT features, such as displaying real-time information from the Android gateway. Since the web service only accepts Websockets, the broker must be able to handle them by encapsulating the MQTT packets in Websockets frames.

The dashboard has been developed with a rather simple graphic interface, and the first thing to do is define the IP address and port of the MQTT broker that the client has to connect to. In this specific case, the port number is 9001, which is the one generally used for Websockets; it is mandatory to enter the username and password that were previously set up during the configuration of the browser, as described in Section 4.3.1. Once the connection has been established, it is possible to receive the sensor's readings, which will then be displayed on screen, as shown in Figure 5.

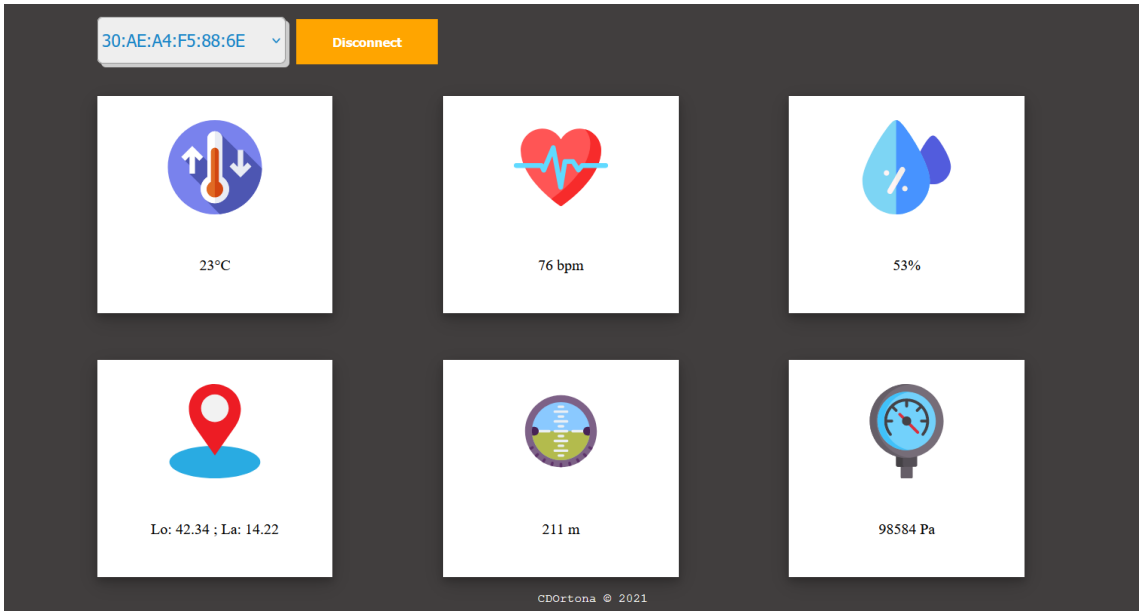

Figure 5. Dashboard: Sensors Display.

The Paho MQTT client running in background allows the user to subscribe to two different topics. The first one concerns the wearable device to be monitored, picked up through the scrolling menu where there is a list of all the wearable devices that have been deployed. This approach is feasible because each wearable device publishes the information gathered by its sensors on different topics through the Android gateway with the structure `BleAddress/valueType`, which allows jointly defining the BLE node and the sensed value. Thus, we can have multilevel topics, where the first level refers to the Bluetooth MAC address of the radio mounted on the ESP32 SoC, while the second layer refers to the type of sensor value being handed over, i.e., temperature, pressure, humidity, etc.

The second topic subscribed by the webservice is `emergency/sos`; it is system-defined and shared among all SoCs. In fact, as the first layer suggests, it does not depend on a single device. This topic allows receiving SoS calls from any wearable device at any point of time regardless of which device is being monitored. This is accomplished simply by integrating the payload with a JSON object. It is then parsed using Javascript, and from it, we extract the key-value pairs, which, respectively, contain the Bluetooth MAC address of the radio mounted on the SoC requesting the SoS and the position of the user, as acquired through the Android device.

5. Feasibility Evaluation

In order to evaluate the feasibility of the proposed system, a Proof of Concept has been setup. In Figure 6, a picture of the real setup of the PoC is presented. It is possible to notice the wearable prototype node that is implemented on the development dashboard where both the BME280 sensor and the ESP32 SoC are installed. An Android smartphone with the developed app is also part of the PoC where a proper BLE link is used for its connection with the wearable node. The Raspberry PI 3B+ node is used as the MQTT Broker; indeed, it receives MQTT messages from the Android device, acting as the publisher and

sends them to the connected MQTT subscriber, here represented as a Windows 10 PC. As previously mentioned, in the implemented PoC, the MQTT subscriber is here implemented as a Webservice, where a proper Dashboard is used for representing the sensed values.

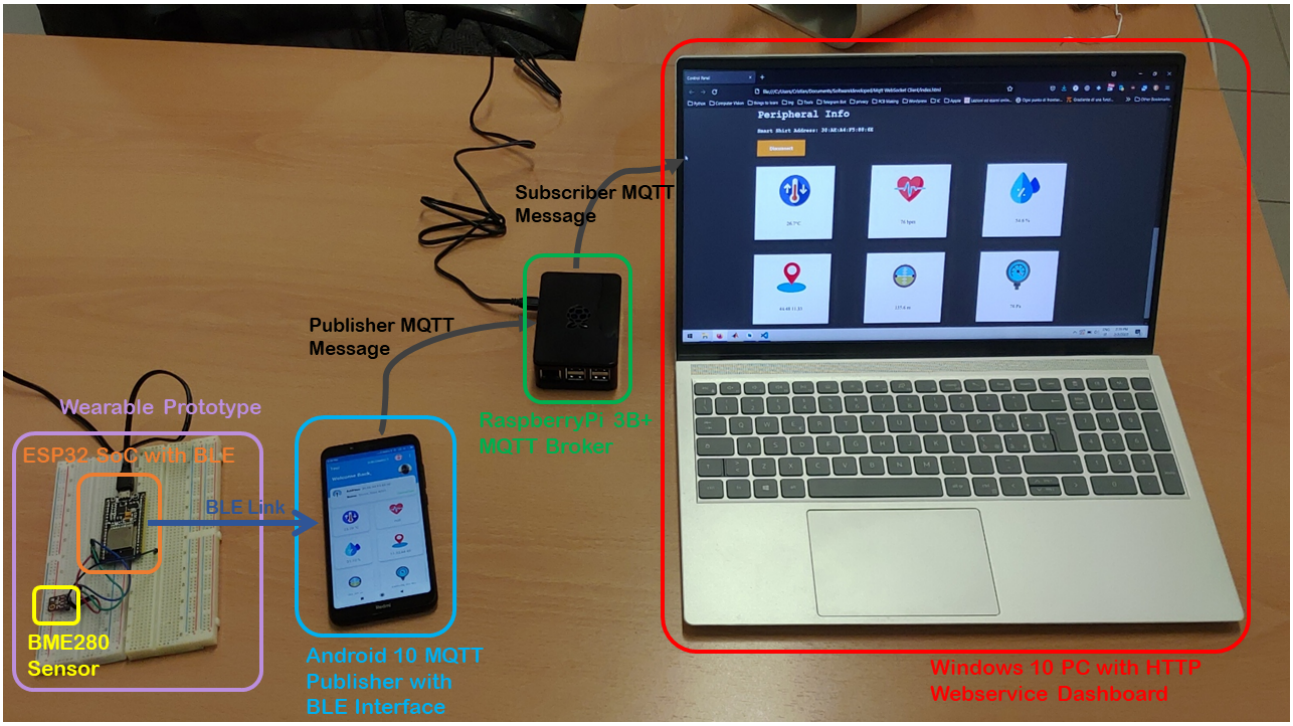

Figure 6. The real setup of the PoC.

In this section, the feasibility of the proposed solution will be analyzed by focusing on the exchange of BLE data packets occurring between the Android smartphone and the ESP32 SoC. While the E2E implementation has been proven by considering the visualization on the Dashboard, as represented in Figure 5, we focus now on the BLE packet exchange. The aim of the analysis carried out is to provide insight into the BLE PDU exchanged between the two nodes and prove that the information contained in such packets do map perfectly to those that have been implemented on the software side. It is worth noticing that the system we have implemented is aimed at testing a single E2E connection going from a wearable device, acting as source of data, to an MQTT subscriber node, acting as sink. To this aim, we have not performed tests over fully loaded scenarios. It is, however, clear from other papers, e.g., [38], that an Eclipse Mosquitto-based solution, despite not being designed as a scalable implementation, allows processing around 20,000 messages per second, which is far beyond those envisaged in the considered scenario.

Smartphones generally have implemented the two most important BLE protocol layers, i.e., the host and controller. The host represents the upper layer of the protocol stack and is generally more resource-hungry than the controller; hence, quite often, it is integrated into the main CPU. It can communicate with the controller through the Host–Controller Interface (HCI). Such configuration is known as Dual-IC-over-HCI since the two protocol layers are embedded into two different Integrated Circuits communicating with each other through a communication mean defined by the Bluetooth specification, implying that any host can exchange data through HCI with any controller, regardless of the manufacturer.

The interaction between these two layers is part of our analysis: we have used the Wireshark packet analyzer tool in order to sniff both the HCI events and the HCI commands, which are the two different PDUs sent by the HCI between those two endpoints. Whenever the user interacts with our app and performs actions that imply the use of the BLE chip, each request is first received and processed by the host, and, later on, the command to be executed is sent to the BLE radio controller, which will answer with an HCI event.

A common HCI command is sent. Whenever the user initializes a scan of the BLE devices nearby, the BLE radio controller will answer with a list of events, i.e., the advertising packets sent out by the nearby Bluetooth devices; it is worth mentioning that HCI events and commands are vendor-specific. In Figure 7, a scan response is shown, which was sent by the ESP32 and contains, in the payload, all the information that was previously discussed in the software development of the ESP32 node.

```
▾ Bluetooth HCI Event - LE Meta
    Event Code: LE Meta (0x3e)
    Parameter Total Length: 43
    Sub Event: LE Advertising Report (0x02)
    Num Reports: 1
    Event Type: Scan Response (0x04)
    Peer Address Type: Public Device Address (0x00)
    BD_ADDR: Espressi_f5:88:6e (30:ae:a4:f5:88:6e)
    Data Length: 31
  ▾ Advertising Data
    ▾ Flags
        Length: 2
        Type: Flags (0x01)
        000. .... = Reserved: 0x0
        ...0 .... = Simultaneous LE and BR/EDR to Same Device Capable (Host): false (0x0)
        .... 0... = Simultaneous LE and BR/EDR to Same Device Capable (Controller): false (0x0)
        .... .1.. = BR/EDR Not Supported: true (0x1)
        .... ..1. = LE General Discoverable Mode: true (0x1)
        .... ...0 = LE Limited Discoverable Mode: false (0x0)
    ▾ Device Name: ESP32
        Length: 6
        Type: Device Name (0x09)
        Device Name: ESP32
    ▾ Tx Power Level
        Length: 2
        Type: Tx Power Level (0x0a)
        Power Level (dBm): 3
    ▾ 128-bit Service Class UUIDs
        Length: 17
        Type: 128-bit Service Class UUIDs (0x07)
        Custom UUID: 4fafc201-1fb5-459e-8fcc-c5c9c331914b (Unknown)
    RSSI: -64 dBm
```

Figure 7. Advertising Scan Response.

Once the connection of the Android smartphone to the SoC has been correctly established, an exchange of ATT PDUs between the two endpoints takes place. Many attributes' protocol PDUs use a sequential request–response protocol [33], where once a client sends a request to the server, there are no more requests being sent from that client until a response PDU has been received; a request–response pair is a defined transaction. This pattern is not valid for notifications, which do not have a response PDU since they are asynchronous; hence, for commands that do not have a response PDU, there is no flow control, and commands can be sent any time without having to wait for a response.

Herein, we intend to limit our analysis to the ATT PDU, which we deem worth mentioning. The first transaction we analyzed starts with the smartphone sending a command to the ESP32 with opcode `ATT_FIND_INFORMATION_REQ`, which is used to obtain the mapping of attribute handles with their associated types, allowing the client to discover the list of attributes and their type of GATT server. If at least one attribute is returned, the `ATT_FIND_INFORMATION_RSP` PDU will be sent from the server to the client; if no attribute is returned, the `ATT_ERROR_RSP` PDU is returned with the respective error code. Another transaction that occurs is the one that starts with requests having opcode `ATT_READ_BY_TYPE_REQ`, which is used for obtaining the values of attributes, when the attribute type is known while the handle is not. The response to this PDU has opcode `ATT_READ_BY_TYPE_RSP` and contains the pair of handles and values of the attributes that have been read.

As stated in Section 4.1, each characteristic has the Notify property, and it is enabled by the Android device by writing to the CCCD of every characteristic of the GATT server that the GATT client would receive notifications from. Such an operation is performed

at the network layer by sending the ATT PDU with opcode `ATT_WRITE_REQ`, which is used to request the server write the value of an attribute; in this case, the CCCD is the attribute of the server we wish to write with value `b'0x01` since it allows enabling the notification property. The PDU with opcode `ATT_WRITE_RSP` will then acknowledge whether the attribute was correctly written or not. The whole initialization procedure aimed at establishing a means of communication between the two endpoints has been analyzed to lasts only 3.7 s, in the worst case scenario, when the distance of the two BLE devices is greater than 1 m. In Figure 8, the PDU corresponding to the delivered notification from the GATT server is shown, which has as its opcode `ATT_HANDLE_VALUE_NTF`, and it is not longer sequential—the transaction pattern defined before is no longer followed.

```
▾ Bluetooth Attribute Protocol
  ▾ Opcode: Handle Value Notification (0x1b)
      0... .... = Authentication Signature: False
      .0.. .... = Command: False
      ..01 1011 = Method: Handle Value Notification (0x1b)
  ▾ Handle: 0x002a (Unknown: Unknown)
      [Service UUID: 4fafc2011fb5459e8fccc5c9c331914b]
      [UUID: beb5483e36e14688b7f5ea07361b26a8]
    Value: 32312e3934
```

Figure 8. Handle Value Notification.

The system has been tested in different communication scenarios with the goal of estimating its performance in a realistic setting. To this aim, we considered connecting the Android device with the MQTT Broker through different wireless technologies, i.e., WiFi, EDGE, HSPA+ and 4G. In Table 1, the results obtained through a measurement campaign performed on slots of 80 s for 10 different times at different hours of the day are reported. We can see the communication technologies' great impact. It is, however, worth noticing that, even in the worst case obtained through the EDGE, we are able to implement some of the applications over the presented scenario. As an example, moving vehicles in traffic through a calm neighborhood were sufficiently alerted about other vehicles nearby.

Table 1. MQTT message latency through different communication technologies.

Technology	WiFi	EDGE	HSPA+	4G
Average Delay	58 ms	1158 ms	651 ms	467 ms
Maximum Delay	104 ms	1997 ms	1430 ms	589 ms
Minimum Delay	31 ms	746 ms	343 ms	302 ms

Moreover, we have to clarify that, following BME280 specifications, each sensor is supposed to generate one sensed value with a rate equal to 21 Hz, i.e., every 47 ms. Hence, the Android node receives one BLE packet every 47 ms. In addition to this, the Android-embedded sensors are set in order to receive data every 200 ms (i.e., by setting the delay to SENSOR_DELAY_NORMAL). In the worst case scenario, since we have five values through the BLE interface and one internal, we have around 110 MQTT packets per second. Considering that each MQTT packet is around 5 B for small data, the data rate is around 4.4 kb/s. We have to add the TCP/IP and Layer 2 overheads. However, we can state that the data are largely supported by any wireless technology.

6. Discussion

Novel solutions to efficiently and safely manage emerging services in smart cities have been considered. Citizens are much more likely to use alternative mobility solutions, especially as a consequence of the COVID-19 pandemic, characterized by different vulnerabilities that co-exist with more traditional ones. The need to manage and protect vulnerable road users and protect them from accidents is emerging and can be solved by advanced

IoT technologies that are suitable even for cheap vehicles. In this paper, we have proposed a system that introduces an E2E solution that is able to monitor the VRU data through wearable devices. Through the proper use of different communication technologies (i.e., BLE, MQTT), devices (i.e., ESP32, Android, Raspberry PI) and software technologies (i.e., Websocket), we have built a Proof-of-Concept solution to monitor the user data in real time. The possibility of benefiting from commonly used devices, such as Android smartphones, as well the open-source release of the whole system code, allows the fast deployment of the proposed solution in smart city scenarios. This solution, which has been demonstrated to be feasible, adopts a flexible open-source software that makes it available to support an additional plug-in for future services.

While main solutions for IoT are based on ad hoc data collection that lacks flexibility and scalability, the solution proposed in this paper shows that by using common off-the-shelf elements, it is possible to enable the design of a highly scalable and flexible IoT system, that, even though presented for road users, can be also applied, with the data collected by the same set of sensors, to other purposes, such as road maintenance or crowd management, and many other use cases. We believe that showing a complete workflow for this kind of application can foster the development of large-scale IoT systems, being of interest not only for researchers but also for enterprises that can rise and spread in this field.

The next steps involve the insertion of multiple users able to act as both the publisher and subscriber so as to enable a user-to-user communication paradigm. Moreover, the possibility of exploiting virtualization and containerization technologies will be exploited in order to create a more flexible environment. Finally, we will endeavor to solve privacy and security concerns. At this time, the solution includes online basic security mechanisms, as detailed in the paper. It is widely known that MQTT necessitates additional security mechanisms [39,40], and in the future, they will become part of the project under development.

Author Contributions: Conceptualization, D.T. and C.R.; methodology, D.T. and C.R.; software, C.D.; validation, C.D., D.T. and C.R.; investigation, C.D.; resources, C.D.; data curation, C.D.; writing—original draft preparation, C.D.; writing—review and editing, D.T. and C.R.; visualization, C.D.; supervision, D.T. and C.R. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The data presented in this study are openly available in Github at [20].

Acknowledgments: The authors would like to acknowledge Marco Moricoli, Andrea Castronovo and Alberto Iantorni for their development as a result of Bachelor Thesis work.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Rayes, A.; Salam, S. *Internet of Things from Hype to Reality—The Road to Digitization*, 2nd ed.; Springer: Cham, Switzerland, 2019. [CrossRef]
2. Belli, L.; Cilfone, A.; Davoli, L.; Ferrari, G.; Adorni, P.; Di Nocera, F.; Dall'Olio, A.; Pellegrini, C.; Mordacci, M.; Bertolotti, E. IoT-Enabled Smart Sustainable Cities: Challenges and Approaches. *Smart Cities* **2020**, *3*, 52. [CrossRef]
3. Srivastava, A.; Gupta, M.S.; Kaur, G. Green Smart Cities. In *Green and Smart Technologies for Smart Cities*, 1st ed.; Tomar, P., Kaur, G., Eds.; CRC Press: Boca Raton, FL, USA, 2019; pp. 1–18.
4. Fernandes, B.; Neves, J.; Analide, C. Road Safety and Vulnerable Road Users—Internet of People Insights. In Proceedings of the 6th International Conference on Smart Cities and Green ICT Systems, Porto, Portugal, 22–24 April 2017; pp. 311–316. [CrossRef]
5. Razzaque, M.A.; Milojevic-Jevric, M.; Palade, A.; Clarke, S. Middleware for Internet of Things: A Survey. *IEEE Internet Things J.* **2016**, *3*, 70–95. [CrossRef]
6. Ray, P.P. A survey of IoT cloud platforms. *Future Comput. Inform. J.* **2016**, *1*, 35–46. [CrossRef]
7. MQTT Specification. Available online: https://mqtt.org/mqtt-specification/ (accessed on 31 December 2021).

8. Borsatti, D.; Cerroni, W.; Tonini, F.; Raffaelli, C. From IoT to Cloud: Applications and Performance of the MQTT Protocol. In Proceedings of the 2020 International Conference on Transparent Optical Networks (ICTON), Bari, Italy, 19–23 July 2020; IEEE: Piscataway, NJ, USA, 2020. [CrossRef]
9. 5GAA Automotive Association. Vulnerable Road User Protection. White Paper, 5GAA. 2020. Available online: https://5gaa.org/news/vulnerable-road-user-protection/ (accessed on 31 December 2021).
10. Scholliers, J.; van Sambeek, M.; Moerman, K. Integration of vulnerable road users in cooperative ITS systems. *Eur. Transp. Res. Rev.* **2017**, *9*, 1–9. [CrossRef]
11. Sewalkar, P.; Seitz, J. Vehicle-to-Pedestrian Communication for Vulnerable Road Users: Survey, Design Considerations, and Challenges. *Sensors* **2019**, *19*, 358. [CrossRef] [PubMed]
12. Hussein, A.; García, F.; Armingol, J.M.; Olaverri-Monreal, C. P2V and V2P communication for Pedestrian warning on the basis of Autonomous Vehicles. In Proceedings of the 2016 IEEE 19th International Conference on Intelligent Transportation Systems (ITSC), Rio de Janeiro, Brazil, 1–4 November 2016; IEEE: Piscataway, NJ, USA, 2016; pp. 2034–2039. [CrossRef]
13. Liu, Z.; Pu, L.; Meng, Z.; Yang, X.; Zhu, K.; Zhang, L. POFS: A novel pedestrian-oriented forewarning system for vulnerable pedestrian safety. In Proceedings of the 2015 International Conference on Connected Vehicles and Expo (ICCVE), Shenzhen, China, 19–23 October 2015; IEEE: Piscataway, NJ, USA, 2015; pp. 100–105. [CrossRef]
14. Saur, S.; Mizmizi, M.; Otterbach, J.; Schlitter, T.; Fuchs, R.; Mandelli, S. 5GCAR Demonstration: Vulnerable Road User Protection through Positioning with Synchronized Antenna Signal Processing. In Proceedings of the 24th International ITG Workshop on Smart Antennas, Hamburg, Germany, 18–20 February 2020.
15. Napolitano, A.; Cecchetti, G.; Giannone, F.; Ruscelli, A.; Civerchia, F.; Kondepu, K.; Valcarenghi, L.; Castoldi, P. Implementation of a MEC-based Vulnerable Road User Warning System. In Proceedings of the 2019 AEIT International Conference of Electrical and Electronic Technologies for Automotive (AEIT AUTOMOTIVE), Turin, Italy, 2–4 July 2019; IEEE: Piscataway, NJ, USA, 2019. [CrossRef]
16. Waldemar, T.; Boehm, F.; Schlegel, T. Prototyping Approach of Networking Road Users for Cooperative Collision Avoidance using Smartphones. In Proceedings of the 2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS), Granada, Spain, 22–25 October 2019; IEEE: Piscataway, NJ, USA, 2019; pp. 374–379. [CrossRef]
17. Hernandez-Jayo, U.; Perez, J.; de-la Iglesia, I. Poster: Wearable warning system for improving cyclists safety in the scope of Cooperative systems. In Proceedings of the 2015 IEEE Vehicular Networking Conference (VNC), Kyoto, Japan, 16–18 December 2015; IEEE: Piscataway, NJ, USA, 2015; pp. 153–154. [CrossRef]
18. Espressif Systems. ESP32 Wi-Fi & Bluetooth MCU. Available online: https://www.espressif.com/en/products/socs/esp32 (accessed on 31 December 2021).
19. Tarchi, D.; Grandi, S.; Cerroni, W. Android-based Implementation of a Fog Computing and Networking Environment. In Proceedings of the 2019 IEEE Wireless Communications and Networking Conference (WCNC), Marrakech, Morocco, 15–18 April 2019; IEEE: Piscataway, NJ, USA, 2019; pp. 1–6. [CrossRef]
20. Internet of Vulnerable Road Users. Available online: https://github.com/DanieleTarchi/IoVRU (accessed on 31 December 2021).
21. Raspberry Pi (Trading) Ltd. Raspberry PI Documentation—Raspberry PI OS. Available online: https://www.raspberrypi.com/documentation/computers/os.html (accessed on 31 December 2021).
22. Eclipse Mosquitto. Available online: https://mosquitto.org/ (accessed on 31 December 2021).
23. Eclipse Paho. Available online: https://www.eclipse.org/paho/ (accessed on 31 December 2021).
24. Babiuch, M.; Foltýnek, P. Creating a Mobile Application with the ESP32 Azure IoT Development Board Using a Cloud Platform. In Proceedings of the 2021 22nd International Carpathian Control Conference (ICCC), Velké Karlovice, Czech Republic, 31 May–1 June 2021; IEEE: Piscataway, NJ, USA, 2021. [CrossRef]
25. Carducci, C.G.C.; Monti, A.; Schraven, M.H.; Schumacher, M.; Mueller, D. Enabling ESP32-based IoT Applications in Building Automation Systems. In Proceedings of the 2019 II Workshop on Metrology for Industry 4.0 and IoT (MetroInd4.0 IoT), Naples, Italy, 4–6 June 2019; IEEE: Piscataway, NJ, USA, 2019; pp. 306–311. [CrossRef]
26. LILYGO® T-WATCH-2020 V3. Available online: http://www.lilygo.cn/prod_view.aspx?TypeId=50053&Id=1380&FId=t3:50053:3 (accessed on 31 December 2020).
27. Bosch Sensortech. Humidity Sensor BME280. Available online: https://www.bosch-sensortec.com/products/environmental-sensors/humidity-sensors-bme280/ (accessed on 31 December 2021).
28. FreeRTOS—Market Leading RTOS (Real Time Operating System) for Embedded Systems with Internet of Things Extensions. Available online: https://www.freertos.org/ (accessed on 31 December 2021).
29. Adafruit BME280 Library. Available online: https://github.com/adafruit/Adafruit_BME280_Library (accessed on 31 December 2021).
30. ESP32 BLE for Arduino. Available online: https://github.com/nkolban/ESP32_BLE_Arduino (accessed on 31 December 2021).
31. Chang, K.H. Bluetooth: A viable solution for IoT? [Industry Perspectives]. *IEEE Wirel. Commun.* **2014**, *21*, 6–7. [CrossRef]
32. OSI Networking and System Aspects—Naming, Addressing and Registration. Information Technology—Procedures for the Operation of Object iDentifier Registration Authorities: Generation of Universally Unique Identifiers and Their Use in Object Identifiers. Rec. X.667; ITU-T. 2012. Available online: https://www.itu.int/ITU-T/studygroups/com17/oid/X.667-E.pdf (accessed on 25 January 2022).

33. Townsend, K.; Cufí, C.; Akiba; Davidson, R. *Getting Started with Bluetooth Low Energy*; O'Reilly Media, Inc.: Sebastopol, CA, USA, 2014.
34. Durkop, L.; Czybik, B.; Jasperneite, J. Performance evaluation of M2M protocols over cellular networks in a lab environment. In Proceedings of the 2015 18th International Conference on Intelligence in Next Generation Networks, Paris, France, 17–19 February 2015; IEEE: Piscataway, NJ, USA, 2015; pp. 70–75. [CrossRef]
35. Grgić, K.; Špeh, I.; Heđi, I. A web-based IoT solution for monitoring data using MQTT protocol. In Proceedings of the 2016 International Conference on Smart Systems and Technologies (SST), Osijek, Croatia, 12–14 October 2016; IEEE: Piscataway, NJ, USA, 2016; pp. 249–253. [CrossRef]
36. Lubbers, P.; Albers, B.; Salim, F. *Pro HTML5 Programming—Powerful APIs for Richer Internet Application Development*, 1st ed.; Apress: New York, NY, USA, 2010. [CrossRef]
37. Pimentel, V.; Nickerson, B.G. Communicating and Displaying Real-Time Data with WebSocket. *IEEE Internet Comput.* **2012**, *16*, 45–53. [CrossRef]
38. Mishra, B.; Mishra, B.; Kertesz, A. Stress-Testing MQTT Brokers: A Comparative Analysis of Performance Measurements. *Energies* **2021**, *14*, 5817. [CrossRef]
39. Butun, I.; Österberg, P.; Song, H. Security of the Internet of Things: Vulnerabilities, Attacks, and Countermeasures. *IEEE Commun. Surv. Tutor.* **2020**, *22*, 616–644. [CrossRef]
40. Singh, M.; Rajan, M.; Shivraj, V.; Balamuralidhar, P. Secure MQTT for Internet of Things (IoT). In Proceedings of the 2015 Fifth International Conference on Communication Systems and Network Technologies, Gwalior, India, 4–6 April 2015; pp. 746–751. [CrossRef]