



저작자표시 2.0 대한민국

이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.
- 이차적 저작물을 작성할 수 있습니다.
- 이 저작물을 영리 목적으로 이용할 수 있습니다.

다음과 같은 조건을 따라야 합니다:



저작자표시. 귀하는 원저작자를 표시하여야 합니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

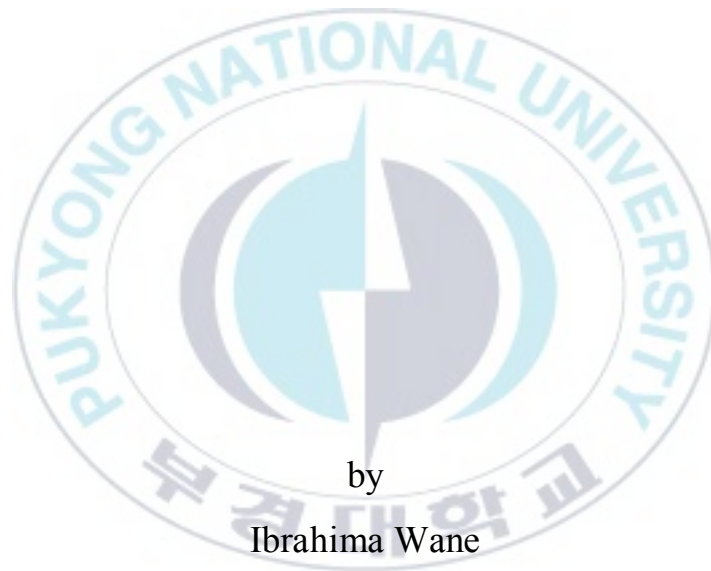
저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 [이용허락규약\(Legal Code\)](#)을 이해하기 쉽게 요약한 것입니다.

[Disclaimer](#) 

Thesis for the Degree of Master of Engineering

A Study on Implementations of IoT System Frameworks



by

Ibrahima Wane

Department of Information and Communications Engineering

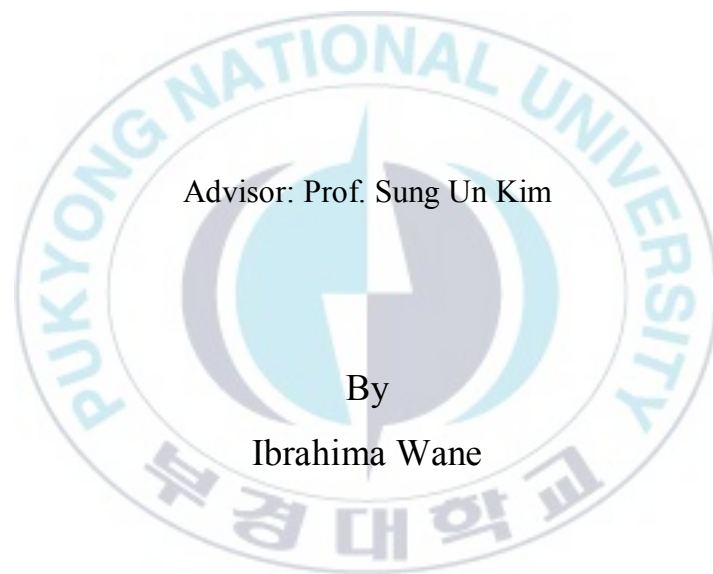
The Graduate School

Pukyong National University

August 2019

A Study on Implementations of IoT System Frameworks

IoT 시스템 프레임워크 구현에 대한 연구



Advisor: Prof. Sung Un Kim

By

Ibrahima Wane

A thesis submitted in partial fulfillment of requirements
for the degree of

Master of Engineering

in Department of Information and Communications Engineering,
The Graduate School,
Pukyong National University

August 2019

A Study on Implementations of IoT System Frameworks

A dissertation

By

Ibrahima Wane

Approved by:

(Chairman) Prof. Kyu Chil Park

(Member) Prof. Jee-Youl Ryu

(Member) Prof. Sung Un Kim

August 2019

CONTENTS

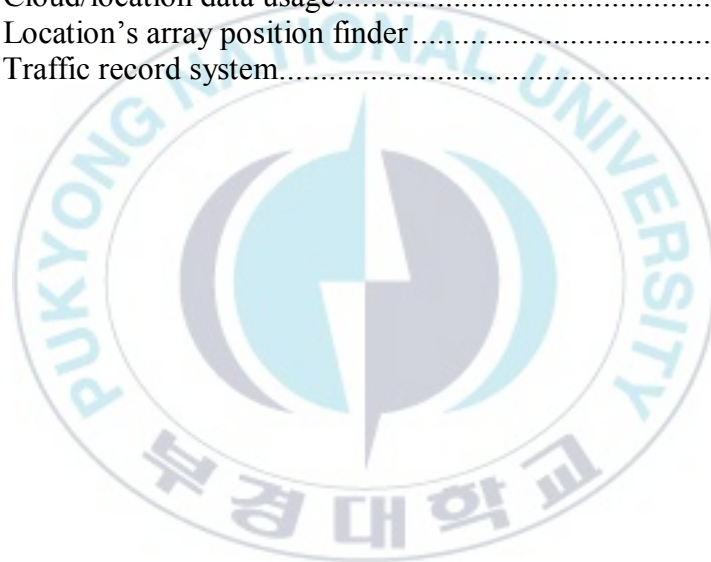
I.	Introduction	1
II.	IoT Framework for Monitoring and Controlling Daily Appliances	4
1.	System Configuration	4
2.	Communication Interfaces	6
2.1.	CoAP	7
2.2.	Controller-Sensor Communication Protocols	12
2.3.	Controller-Appliance Interface	19
3.	Pin-based IoT Message	20
4.	Mobile Application Design	23
4.1.	Add/Remove Controllers	25
4.2.	Add/Remove Appliances	28
4.3.	Control/Monitor Appliances	33
4.4.	Appliances Information	38
4.5.	Notifications Functionality	39
4.6.	Android Smartphone CoAP Implementation	42
5.	Controller Design	44
5.1.	CoAP Server	45
5.2.	Controller-Appliance/Sensor Interfaces	48
5.3.	Reservation	48
6.	Cloud for Multiple Users Interface	50
7.	Service Scenarios	52
7.1.	Appliances-Controlling Procedures	52
7.2.	Requesting Sensor Data Procedure	54
7.3.	Notification from the Central Computer	55
8.	Security Implementation	56

8.1. Key Exchange.....	57
8.2. Tag Generation.....	58
9. Performance Comparison.....	59
9.1. CoAP vs. HTTP.....	59
9.2. Advantages of Public Key Exchange	60
9.3. LEA vs. AES	61
III. IoT Framework for Road Accidents Mitigation.....	62
1. Overall System Structure	63
2. Road Speed Limiters.....	65
3. Location Techniques.....	68
4. Vehicle Cellular RSL Prototype.....	70
4.1. Raspberry Pi Software Implementation.....	71
4.2. Android App Programming.....	73
5. Vehicle Speed Record.....	81
Conclusion	83
References.....	84

LIST OF FIGURES

Figure 2-1. IoT system structure	5
Figure 2-2. Abstract layers of CoAP protocol	7
Figure 2-3. CoAP packet format	8
Figure 2-4. Reliable message transmission	10
Figure 2-5. Unreliable message transmission	10
Figure 2-6. Asynchronous serial message	14
Figure 2-7. SPI protocol	15
Figure 2-8. I ² C message.....	17
Figure 2-9. Multi-parameter IoT message format.....	20
Figure 2-10. Simple control service message formats	21
Figure 2-11. Sensor monitor service message formats.....	21
Figure 2-12. Android app user interfaces	24
Figure 2-13. Activities structure	25
Figure 2-14. Controller add user interface.....	26
Figure 2-15. Adding/ removing elements flowchart	27
Figure 2-16. Appliance adding UI	29
Figure 2-17. Appliance removing UI	30
Figure 2-18. Android picker UI	31
Figure 2-19. Image picking process	32
Figure 2-20. Appliances home.....	33
Figure 2-21. Appliance control	34
Figure 2-22. Time picker	37
Figure 2-23. Appliances information	39
Figure 2-24. Notification process.....	40
Figure 2-25. Notifications list UI.....	41
Figure 2-26. Californium data transmission	42
Figure 2-27. Californium data reception	43
Figure 2-28. Node.js UDP server implementation.....	45
Figure 2-29. Node-coap CoAP server implementation	46
Figure 2-30. Reservation pseudocode	49
Figure 2-31. Cloud as interface for multi-users	50
Figure 2-32. Appliance controlling process.....	53
Figure 2-33. Appliance controlling process (multi-interface)	54
Figure 2-34. Sensor data request procedure	55
Figure 2-35. Notification-sending procedure.....	56
Figure 2-36. Key generation procedure.....	57
Figure 2-37. Authentication tag generation	59

Figure 2-38. HTTP and CoAP packet content	60
Figure 2-39. LEA and AES performance graph	61
Figure 3-1. System structure	63
Figure 3-2. Multi-speed limits cell	64
Figure 3-3. Multi-speed limits management	65
Figure 3-4. Direct fuel control type of RSL structure	67
Figure 3-5. Trilateration scenario	69
Figure 3-6. System prototype	70
Figure 3-7. FCU prototype code	72
Figure 3-8. ECM prototype code	73
Figure 3-9. Driver-vehicle information registration UI	74
Figure 3-10. Firestore project's data structure	75
Figure 3-11. Acquisition of data from Firestore	76
Figure 3-12. Cloud/location data usage	78
Figure 3-13. Location's array position finder	79
Figure 3-14. Traffic record system	82



LIST OF TABLES

Table 1. CoAP fields	9
Table 2. Response codes.....	12
Table 3. IoT message format fields.....	22



A Study on Implementations of IoT System Frameworks

Ibrahima Wane

Department of Information and Communications Engineering, The Graduate School,
Pukyong National University

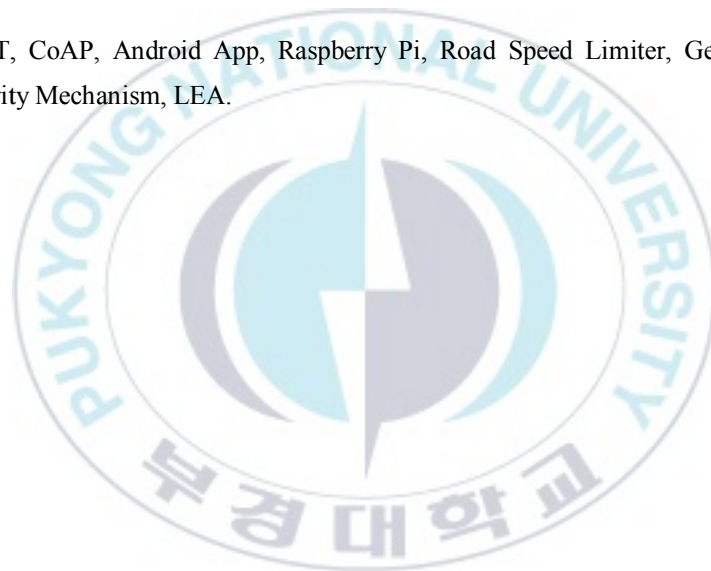
Abstract

This research can be divided into two main parts. The purpose of the first part of this research is to build a complete and easy-to-use framework for setting up controllable Internet of Things (IoT) Systems such as Home IoT Systems and Smart Farm Systems. We created a dynamic Android mobile application adaptable to the user's preferences and to various systems of appliances-managing computers/microcontrollers, which we will refer generally as controllers. In addition, we developed a controller side application for handling requests sent from the mobile application as well as for making requests in certain circumstances. We used multiple sensors and devices to represent real-life appliances. It is aimed that nonprofessional users, with some references, can easily set up their own IoT system using this framework. In terms of application layer protocols, instead of using the traditional HTTP as Internet application protocol, we used the Constrained Application Protocol (CoAP), which is a web transfer protocol especially designed for lossy networks and for memory-limited and low power devices. Based on the structure of the controllers, we built our own message formats for better parsing of the requested services. On the top of that, we append security tags to all messages transmitted between the user and the system using the (Lightweight Encryption Algorithm) LEA security mechanism. Finally, we made use of the Firebase's Cloud *Firestore* database to update certain information to all users in an IoT system.

The second part of the research has the aim of using the technologies proposed previously such as Cloud, Android mobile application, CoAP, to another type of system that we structured. The system imposes an area-based threshold speed to vehicles, to mitigate accidents due to excessive or unsuitable speed. Although traffic lights, road signs, police and other security measures in place play different roles to ensure that each vehicle is running at the appropriate speed, the driver is free to speed up as much as he wants and, he might, in result, cause a great damage or a loss of humane life. This project provides a practical solution to this issue by limiting the driver's freedom. On the one hand, the vehicle keeps its potentiality of running at high speed. On the other hand, it follows

automatically the area's rules and regulations. As cellular networks are almost omnipresent in inhabited places and where Global Positioning System (GPS) is seldom used, we primarily use this technology by setting in each cell a general threshold speed, and even within the cell, various speed limit values might be set. Each base station communicates with the vehicles within its cell and issues to each the appropriate speed. To specify a speed limit in a particular area within a particular cell, the GPS data of the vehicle might be optionally utilized. For controlling the vehicle's speed based on an input value, an Electronic Control Unit (ECU) is used. To know the current speed of the vehicle for the ECU to take action, speed sensor connected to the vehicle's gearbox is used. To ensure that the sensor is not sensing incorrectly, we use the positions of the vehicle and a timer to check the speed. Finally, we implemented a prototype to illustrate how the system might work in real-life.

Keywords: IoT, CoAP, Android App, Raspberry Pi, Road Speed Limiter, Geolocation, Serial Protocol, Security Mechanism, LEA.



I. Introduction

Internet of Things (IoT), in short, refers to ‘things’ connected via the Internet. A thing is meant by any object provided with a unique identifier and, as a result, can communicate with other identifiable objects through the Internet [1]. IoT devices include sensors, home appliances, actuators, vehicles, etc. There are many application areas for IoT systems including smart home and cities, transportation systems, industrial control systems, healthcare, etc.

Home IoT exemplifies one of IoT application areas. It allows to remotely manage and monitor home appliances, such as activating an air conditioner and verifying the indoor temperature [2]. This capability of managing and having information about one’s IoT system through a smart device, anywhere at any time, is not only convenient and comfortable but also contributes to energy efficiency and to local security. To elaborate on the last statement, consider if a homeowner forgets to lock his door when leaving home, he can be notified that the door is not locked – home security. Also, it is possible to set the thresholds of how long a particular appliance should be in a particular state – energy efficiency. Applying IoT to agriculture, can help a farmer get detailed information about his plants and the farming environment and, as a result, he can manage them easily. By the help of sensors, information related to plant growth, soil moisture and air temperature can be collected. In result, installed air conditioners, sprinklers, lightbulbs and heaters can be remotely actuated. Furthermore, when IoT is applied to road traffic it can be a mean of mitigating road traffic accidents and, even, reducing physical surveillance over the traffic.

Most of the IoT implementation projects focus on implementing a single IoT application area, such as Smart Bathroom or Smart Farm. Usually, the mobile application is hardcoded and therefore works specifically for a particular system. Also, the controller is programmed to interact only with a specific type of sensors and appliances. However, the communality between most of these IoT systems is that, there is a user with a handheld device who, through the Internet, controls an appliance or gets information from a sensor. Most of these systems also comprise of a single-board computer, such as Raspberry Pi, or a microcontroller, such as Arduino, to interface between the appliance and the service request message sent remotely by the user. These interface devices interact with the appliances and sensors through their pins. This forms a pattern, from which we inferred that we can put forth an IoT system framework for implementing various IoT application areas – which is the main objective of the first part. On the top of that, we exploited the LEA security mechanism in order to ensure that the data is genuinely sent by the user and it has not been counterfeited [3]. Furthermore, we make use of the cloud functionalities in order to make the system real-time and accessible to multiple users [4].

As for the road traffic control system, the second part, the objective is to control speed related road traffic accidents using IoT concepts. The idea is that the existing speed governing techniques known as Road Speed Limiters (RSL) will be used in combination with cellular networks, in order to continuously monitor the traffic and provide to each running vehicle the appropriate speed for each zone. It is assumed that any running vehicle will be obliged by the law enforcement to implement the system and its data will be kept on track. Optionally, the GPS technology will play a role to perfect the system in various aspects. The reasons of using cellular networks in preference of GPS are numerous. One being that the

GPS implementation is expensive and it only works outdoors, it quickly consumes battery power, and doesn't return the location as quickly as users want[5].

Our objective is to put forth a system that controls the traffic speed in a centralized fashion, using available technologies in both developed and underdeveloped countries. There are already techniques attempting to control a vehicle's speed but mostly give the freedom to the driver to implement those and set whatever speed he wishes; this is one of the issues. Because, the safety of a whole population should not be put on hands of a particular driver. Rather, it should be the work of the authority to monitor and govern road traffic speeds. In other words, like various mandatory vehicle system implementations such as *eCall* system, this type of systems also should be made mandatory to every vehicle. We, therefore, propose a driver-independent model of controlling vehicles' speeds based on area-determined speed limit.

The contribution of this research is twofold. First, it has contributed to making IoT systems easier to implement by providing a customizable and easy-to-use application and a message format for various services. Second, using the concept of IoT, it has proposed a new approach for mitigating road traffic accidents and a prototype that can be used to simulate real-life traffic controlling system.

II. IoT Framework for Monitoring and Controlling Daily Appliances

1. System Configuration

Basically, we will be structuring and implementing a system where a user, through a mobile phone, controls and monitors multiple appliances that are connected to a central computer. IoT systems with this structure are plenty. Obviously, a common framework for various IoT systems is more useful than multiple implementations that do basically the same work. However, as there are different IoT devices that perform the same tasks, we have to select some specific components as bases of the project.

For the basic setting of the IoT system, we use a single-board computer, Raspberry Pi, as the central computer and Android-based smartphone as the user interface device. In the Raspberry Pi side, a server environment called *Node.js* is used to run JavaScript files. The Android mobile development environment is called Android Studio. In both, extra packages are added for playing different roles which will be explained subsequently. For the web transfer protocol, instead of HTTP, CoAP protocol is used, because it is, relatively, efficient and more appropriate for low power and limited memory devices and for lossy networks. Extra Raspberry Pi controllers can be connected to the central one through Wi-Fi and Arduino microcontrollers to one of the single-board computers via Bluetooth. Figure 2-1 depicts an IoT system scenario with a single user and different controllers and appliances.

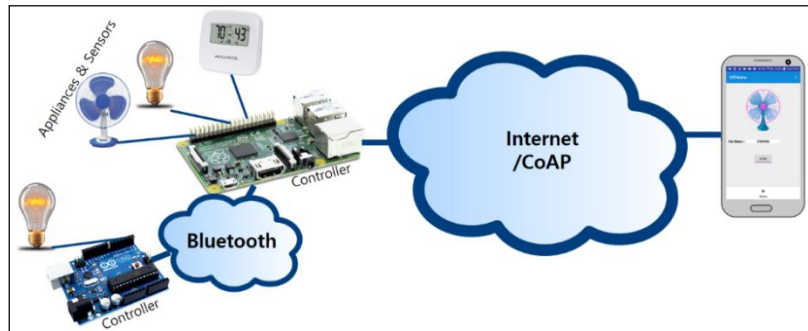


Figure 2-1. IoT system structure

We assume that a real-life appliance performs certain functions, following the rules of the controller. For example, an ordinary fan has an in-built speed control feature; nevertheless, we assume here that the controller will manage it using the inbuilt Pulse Width Modulation (PWM) technique.

We can classify the appliances into two categories: a controllable appliance or a data providing 'appliance' known as sensor. A controllable appliance is primarily not required to send any data but only receives commands such as turning on and off from a controller. In addition, a controllable appliance can be a two-state or multi-parameter appliance. Certain appliances, such as lightbulbs, can be only in two states: activated or deactivated. A two-state appliance uses only one controllable digital pin from a single board computer or microcontroller, where the pin will be powered or unpowered depending on the desired state to which the user wants to set the appliance. On the other hand, others, along with those two states, might have extra parameters such as speed, orientation, intensity and so on. We call them multi-parameter appliances. A multi-parameter appliance can use more than one pin. Although one analog pin can be used for a different level for each parameter, two different parameters will need to use two separate pins. For

example, an air conditioner typically has at least three parameters: on/off, speed and orientation. Speed and orientation have to use different analog pins, as they are not performing the same operation.

There exist many different types of sensors, e.g. light sensors, humidity sensors, temperature sensors, sound sensors, motion sensor, etc. Each sensor has a distinct way of sensing the environment around it and provides some data or measurement to its respective controller. However, each one may use a different protocol to communicate with the controller and the data itself may be structured differently. For now, the only supported protocols in this system are the Universal Asynchronous Receiver/Transmitter (UART), The Synchronous Peripheral Interface (SPI), Inter-Integrated Circuits (I²C), and 1-Wire. Whenever the controller retrieves some data from the sensor, the controller ‘knows’ which protocol to use, as it has been already programmed. Note, if an appliance contains a sensor and is controllable, each part must be considered as a ‘different’ appliance.

The smartphone sends a request to the Raspberry Pi controller through the Internet using the CoAP protocol. The Raspberry Pi, in turn, manages the requested service by directly interacting with the wired appliances or forwards it to the Arduino microcontroller via Bluetooth. The Arduino microcontroller, then, turns on/off the wired lightbulb.

2. Communication Interfaces

Here, we discuss some of the communication protocols that we will use in order to transfer data between the user’s smartphone and the single-board computer, and

also between the controller and various sensors. Also, we will detail about the techniques that the controller uses to control different appliances.

2.1. CoAP

Besides using lossy networks, IoT and machine-to-machine (M2M) devices have relatively small memory and low power supply, in contrast to the traditional Internet devices[6]. These constrained devices, known as nodes, require a simple protocol which enables them to communicate through the constrained networks and, in some cases, through the Internet with other constrained or general Internet devices. Their key requirements are low overhead, simple and multicast supporting protocol. In order to meet all those conditions, the CoAP has been put forward as a candidate.

Defined by RFC 7257 and standardized by the Constrained RESTful Environments Working group (CoRE) from the Internet Engineering Task Force (IETF), the CoAP is a specialized web transfer protocol for using constrained nodes and constrained networks (i.e. low power consumption and asynchronous communication) in IoT. The protocol is designed for M2M applications such as smart energy and building automation fields. Figure 2-2 shows the CoAP protocol abstract layer.



Figure 2-2. Abstract layers of CoAP protocol

This protocol is applied to the request and response-oriented services using the User Datagram Protocol (UDP) communication. Designed specifically for point-to-point communication, it can also support multipoint communication.

2.1.1. Message Types

CoAP has four message types: Confirmable (CON), Non-confirmable (NON), Acknowledgement (ACK) and Reset (RST). Similar to HTTP, in CoAP, a Request is to be sent by the client to the server requesting an action to be performed. Then the sender sends a Response – they are the message code. The CoAP message format, as illustrated in Figure 2-3, shows these fields as well as other basic header fields followed by optional variable-length fields.

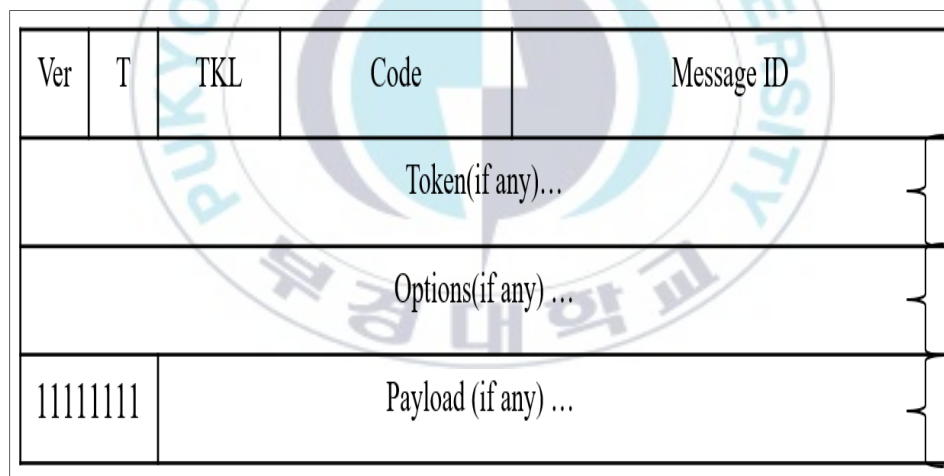


Figure 2-3. CoAP packet format

The CoAP header is short and simple (4-byte length), and its format is the same for Request and Response. To illustrate more, Table 1 shows what each field in the above packet represent and it explains the function of each.

Table 1. CoAP fields

Fixed Fields	Representation	Function
<i>Version(Ver)</i>	The version of the CoAP protocol (2 bits).	Informs about the CoAP version the message is using.
<i>Type(T)</i>	The message type (2 bits).	Tells if the message is reliable (CON) or unreliable (NON). Or, if it is acknowledged (ACK) or not processed (RST).
<i>Token Length(TKL)</i>	The length of an eventual token (4 bits).	Tells how long the token is (0 to 8 bytes) and is used when parsing it.
<i>Code</i>	The request or the response code carried by the message (8 bits).	Tells if a message is a request or a response, and the requested action or the result of a req.
<i>Message ID</i>	The ID of the current message (16 bits).	For avoiding duplicates and for optional reliability.

The fixed-length CoAP header may be followed by a Token which has a variable length, between 0 to 8 bytes (9 to 15 bytes are reserved), and is used to correlate requests and responses. Multiple Options may follow the Token and, in turn, followed by the Payload-Marker and finally the Payload.

CoAP messages are sent asynchronously transported over UDP. There are four types of messages defined in CoAP: CON, NON, ACK and RST. Depending on the method codes and response codes included in these messages, they carry requests and responses. Requests are transported in either the CON or NON message types. Responses are carried in these as well, and then may be piggybacked in the ACK message types in some cases. Figure 2-4 illustrates the process of a reliable message transmission.

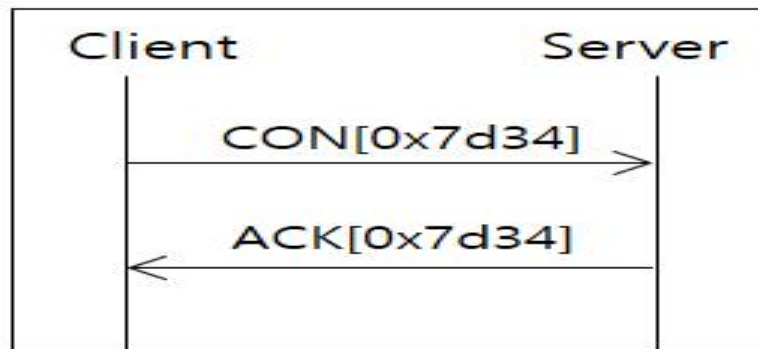


Figure 2-4. Reliable message transmission

Reliability is meant by the capability of retransmitting when the transmission is not successful. A message is referred as reliable when it is of CON type. Using a default timeout, confirmable messages are retransmitted and an exponential back-off is used between retransmissions until the recipient sends a message of ACK type with a same message ID. In case where the recipient is not able to process a Confirmable message, it responds with a message of RST type instead of an ACK. The unreliable message transmission process is illustrated in Figure 2-5.

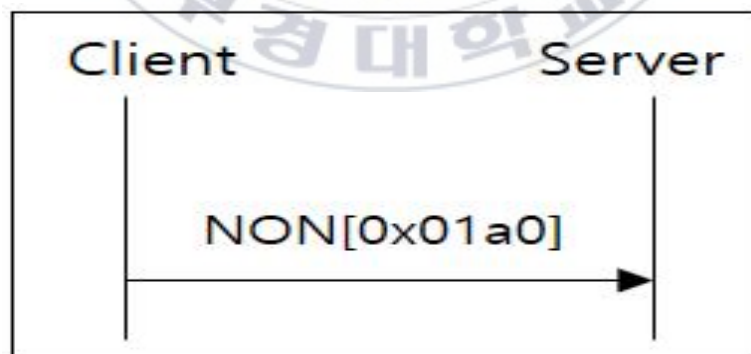


Figure 2-5. Unreliable message transmission

When the recipient does not require a reliable transmission, a NON message type can be sent. Although, no Acknowledgement is required in this case, the message ID (MID) is used to detect duplicates. As with Confirmable messages, when a recipient is unable to process a Non-Confirmable message, it sends a RST message.

2.1.2. Request Codes

Request codes, as in HTTP protocol, are of four types: GET, POST, PUT and DELETE. They represent the services requested by a client to the server to perform some specific actions. Let us discuss the function of each request as defined by RFC 7252.

The GET method, represented by '0', retrieves a representation for the information that currently corresponds to the resource identified by the request URI. The GET method is safe and idempotent. The POST method, represented by '1', requests that the representation enclosed in the request be processed. The actual function performed by the POST method is determined by the origin server and dependent on the target resource. It usually results in a new resource being created or the target resource being updated. POST is neither safe nor idempotent. The PUT method, represented by '2', requests that the resource identified by the request URI be updated or created with the enclosed representation. The representation format is specified by the media type and content coding given in the Content-Format Option, if provided. PUT is not safe but is idempotent. The DELETE, represented by '3', method requests that the resource identified by the request URI be deleted. DELETE is not safe but is idempotent.

2.1.3. Response Codes

CoAP Response Code is similar to the HTTP Status Code. It indicates the result of the attempt to understand and perform the requested action. As listed below in Table 2, CoAP response codes can be mapped to those of HTTP and are self-describing.

Table 2. Response codes

Success 2.xx	Client Error 4.xx	Server Error 5.xx
2.01 Created	4.00 Bad Request	5.00 Internal Server Error
2.02 Deleted	4.01 Unauthorized	5.01 Not Implemented
2.03 Valid	4.02 Bad Option	5.02 Bad Gateway
2.04 Changed	4.03 Forbidden	5.03 Service Unavailable
2.05 Content	4.04 Not Found	5.04 Gateway Timeout
	4.05 Method Not Allowed	5.05 Proxying Not Supported
	4.06 Not Acceptable	
	4.12 Precondition Failed	
	4.13 Request Entity Too Large	
	4.15 Unsupported Content-Format	

2.2. Controller-Sensor Communication Protocols

There is a variety of sensors using different techniques in order to transmit the data they sense to the controller (or MCU). Those techniques include Analog-to-

Digital Conversion (ADC), UART, SPI, I²C, 1Wire, etc. In this project we used some of these, and will therefore explain the way they basically work.

2.2.1. Analog-to-Digital Technique

The analog-to-digital conversion consists of sampling an analog signal (voltage) and converting it to a digital one. There is a multitude of techniques used to achieve this feature. The most common one takes the input analog voltage and uses it to charge a capacitor. Then, the time it takes to discharge the capacitor into a resistor, will be expressed as the value of the output signal. Smaller is the input, faster the capacitor will discharge and, thus, a small value will be the output. The Arduino board has a 10-bit ADC; that means, it provides a 2^{10} range of integer values as output from analog signals of 0 to 5 Volts (or 3 Volts). *Keyes Analog LDR* light sensor is an example among many sensors that use this technique.

2.2.2. Asynchronous Serial (UART)

Data can be sent through a communication link in a parallel way or serially. In parallel communication, several bits of data are sent at once through a link with several parallel channels. In contrast, the serial communication entails sending sequentially one bit of data at a time over a communication channel. There are two types of serial communication: asynchronous serial and synchronous serial.

In asynchronous serial, transmitting and receiving devices are not required to have any external wire for clock synchronization. Therefore, only one wire is used in order to transmit data. As no clock signal is used and for other reasons, some extra mechanisms are needed for a robust and error-free data transfer. These mechanisms consist of transmitting a message that has synchronization bits (start

and stop bits), parity bits and data bits and setting a fixed baud rate for both transmitter and receiver. The synchronization bits indicate the start and the end of a data transmission session. The parity bit is used to detect errors and data bits are the actual payload. The baud rate is the speed at which the data is sent; in other words, the baud rate represents the number of bits to be sent in a defined period of time. It must be fixed for both devices for a meaningful communication. A typical rate is 9600 bit per second.

Typically, to implement asynchronous serial communications, inside microcontrollers, a block of circuitry called UART is used. Between two communicating devices, the data line is always kept high when there is no ongoing data transfer. A bit is expressed by setting the data line high or low for a determined period of time. An asynchronous data transfer example is shown in Figure 2-6.

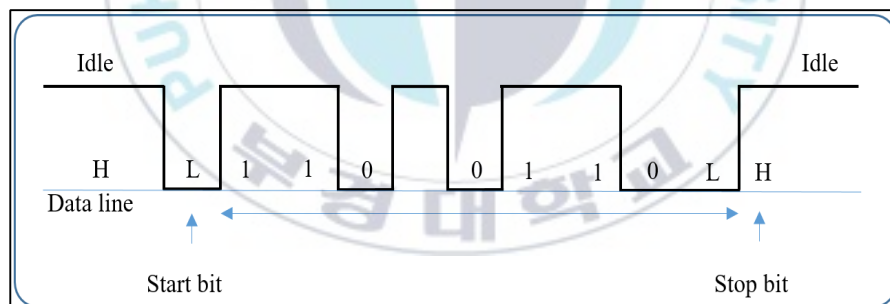


Figure 2-6. Asynchronous serial message

The line is high (H) when it is idle. Then, the transmitter sets the data line low (L) to notify the receiver that data will be transmitted. The receiver starts sampling to determine how long the line is low (should be at least the half of a bit duration) and if it realizes it is a start bit then it gets synchronized for receiving

data. The data is of 8-bit length, the ninth is the optional parity bit followed by a stop bit which puts the line high and then idle again. The RCWL-1603 Ultrasonic Module Distance Measuring Transducer Sensor is an example of a sensor that uses the UART serial communication protocol.

2.2.3. Synchronous Peripheral Interface

The SPI is a type of synchronous serial communication protocol used to transmit data between microcontrollers and small peripherals. SPI devices communicate in a master-slave fashion. The master manages which device should be using the medium and also provides a clock signal to the slave for it to receive and/or transmit data. The communication between the master and the slave is full duplex. Figure 2-7 shows the SPI master-slave structure.



Figure 2-7. SPI protocol

Four wires are used in the process. The Slave Select (SS) line is used by the master to select which device it communicates with. It sets the line low during the data exchange period – which is normally held high to disconnect slaves which are not addressed. If the line is set low, the slave is now activated and is ready for receiving and transmitting data. The master uses the Signal Clock (SCK) wire to generate a clock signal for synchronizing the data communication. The slave uses

the clock signal to sample the data that it receives through the Master Out Slave In (MOSI) line – the data is usually a command. E.g. a “read” command to a sensor. It also uses the clock signal generated by the master, to transmit data back synchronously, through the Master In Slave Out (MISO) line – this data can be a 8-bit sensor data. As SPI is not standardized, there are various ways a bit gets sampled and shifted out; all depends on the clock signal. If a clock signal is idle when being low, this mode is called “Non-Inverted”. On the other hand, if the clock signal is idle when being high, this mode is called “Inverted”. Each main mode is also comprised of two modes. The Non-Inverted mode 0 samples a data bit on the rising edge of the clock pulse and shifts out to another bit on the falling edge. The mode 1 samples a bit on the falling edge of a clock pulse and shifts out on the rising edge. The mode 2 and mode 3 are Inverted modes; the clock signal starts by being high. The mode 2 samples on the falling edge and shifts out on the rising edge of the clock pulse. Finally, the mode 3 samples on the rising edge and shifts out on the falling edge. Also, other parameters can vary such as the issue of transferring the Most Significant bit (MSB) first or the Least Significant bit (LSB). ArduCAM-M-2MP Camera Shield exemplifies a sensor that uses the SPI protocol.

2.2.4. Inter-Integrated Circuits

The I²C is also a synchronous serial communication protocol. This protocol, basically, combines the best features of SPI and UART together. Just like SPI, a master I²C device can communicate with multiple slave devices. Instead of four wires though, it uses only two wires like UART. The two wires are Serial Clock (SLC) and Serial Data (SDA). The SLC is managed by the master and generates a synchronizing clock signal used when data is transmitted. The data is transmitted through the SDA line. Differently from SPI, to communicate with a specific slave

among many, the I²C protocol does not require a Select Line but uses addresses. Whenever data is to be transferred, the address of the slave is transmitted first with two extra bits that we describe subsequently. Then the slaves compare the address with their own addresses. The ones that did not match it, ignore it, while the one that finds itself to be the addressee, prepares for data transmission with the master. Among the two extra bits mentioned, one is the Read/Write bit. This bit indicates which operation is required by the master; whether it wants to read (or receive) data from the slave, or it wants to write data. In the first case, synchronized to the clock signal, the data is immediately transmitted by the slave. Otherwise, the slave waits for an incoming data that will be sent by the slave. However, for addressing purposes and some other factors, the I²C messages are broken into frames as depicted in Figure 2-8.

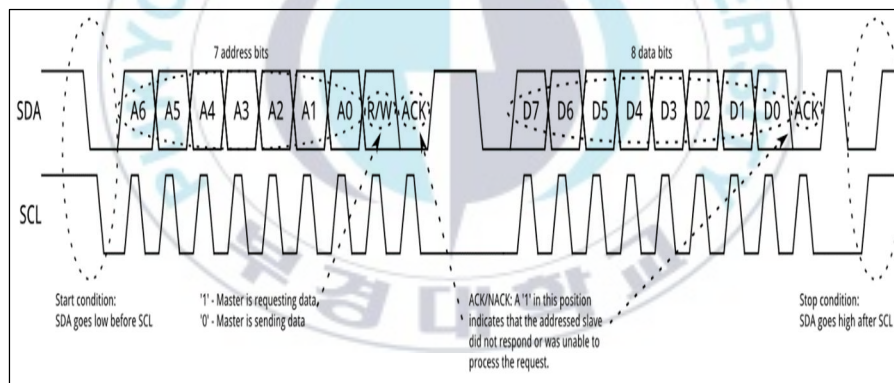


Figure 2-8. I²C message

The address frame containing a seven-bit slave's address, a R/W bit and a NACK/ACK bit. The remaining is one or multiple data frames composed of eight data bits and a NACK/ACK bit. Besides, there are 'start' and 'stop' conditions.

When starting a communication, the master sets the SDA line low and then sets the SCL line low some period of time afterwards – this is the ‘start’ condition. If a slave detects this condition, it understands that the master is about to send a frame. The 7-bit address is, afterwards, sent to indicate to the slave that the following order is addressed to it. The order is the R/W bit; telling the slave which operation the master is requesting – 1 (high) for ‘Read’ and 0 (low) for ‘Write’. The following apply to both, the address and the data frames. The NACK/ACK bit indicates whether the receiving device has properly received and understood the previous 8 bits sent. Whenever 8 bits are sent, the receiving device is given the control over the SDA line to pull it down, for it to show acknowledgement – if the line is high (1), this means the receiving device did not properly process the request or did not respond for some eventual reasons. Multiple data frames can be transmitted. When stopping the communication, the master sets the SDA line low and then after a determined period, it sets the SCL low – this represents the ‘stop’ condition. The data is sent with the most significant bit first (MSB).

Besides, there are other protocol parameters to consider when the I²C bus is not as simple as a master communicating with 7-bit address holding slaves. For example, a slave’s address can be of 10 bits. In this case, the pattern ‘11110’ is transmitted as first bits of the address frame (not part of the address). As no 7-bit address starts with that pattern, the slave knows that the address length is going to be 10 bits. Therefore, two address frames are transmitted before the data frame.

When there is more than one master in the bus, a master can indicate that it is going to exchange several messages in one turn, in order for others not to interfere. This is done by repeating the ‘start’ condition and putting a ‘stop’ condition only when the master is done. Finally, when a slave is not ready, doing some task, and the master wants to transmit a message, it can put the master on hold; this is called

“Clock stretching”. Although, the master is the one that controls the SCL line, a slave can set the SCL line low in order to apply clock stretching. An example of a sensor that uses the I²C interface is the *MINI Si7021* Temperature and Humidity Sensor.

2.3. Controller-Appliance Interface

There are different ways that a controller controls an appliance. The digital controlling technique is simple. It consists of turning the pin, to which the appliance is wired, high or low. As for the analog controlling, multiple values in certain range are periodically applied to the pin. The technique used for the analog controlling is called Pulse Width Modulation (PWM).

Typically, a controller sets the pin to whether 5v or 0v, but never directly to voltages in between. It is where the PWM technique comes into play. It consists of making a square wave signal by switching the voltage between high and low at a determinable rate. Through this, a simulation of a voltage ranging from the higher to the lower voltage can be generated. The output value depends on the time spent on “full on”, 5 Volts, and off, 0 Volt. The time spent being “on” is what is referred as pulse width. Greater is the pulse width, the simulated output voltage will be closer to “full on” and brighter is the LED. Lesser the pulse width, the voltage is nearer to 0 Volt.

This technique is used in the Raspberry Pi and Arduino boards. Depending on which one and which library used, the range of values mapped with those voltages might differ. The Arduino maps the voltage range of 0 ~ 5 Volts to 0 ~ 255 (8 bits); which means each 8-bit value has a corresponding simulated voltage value

between 0 and 5. The same value mapping is applied in the *pigpio* library. The *node-wiring-pi* library maps the voltage to 0 ~ 1023 (10 bits) instead.

3. Pin-based IoT Message

In the CoAP message payload field, we built message formats composed of fixed and optional fields. There are essentially three message formats with fields presents in the one in Figure 2-9 – therefore we will give a detailed description about this one; which includes all the fields.

CoAP Header	Controller Type	TC Add	Days	Hours	Minutes	Pin	Type	Action	Num. of Param.	Type of Param.	Param. Pins	Param. Levels	Requested Levels
≥4 B	2 bits	4 B	7 bits	5 bits	6 bits	1B	2 bits	2 bits	4 bits	N/A	N/A	N/A	N/A

Figure 2-9. Multi-parameter IoT message format

The light grey fields are required in any case of appliance controlling or monitoring, whereas the dark grey fields are optional and represent the parameters relative to certain controllers or appliances. This message is sent by the smartphone to the central computer (server) in order to provide it with the necessary information about the appliance/sensor and the requested service. Figure 2-10 shows the message format for controlling a single digital pin.

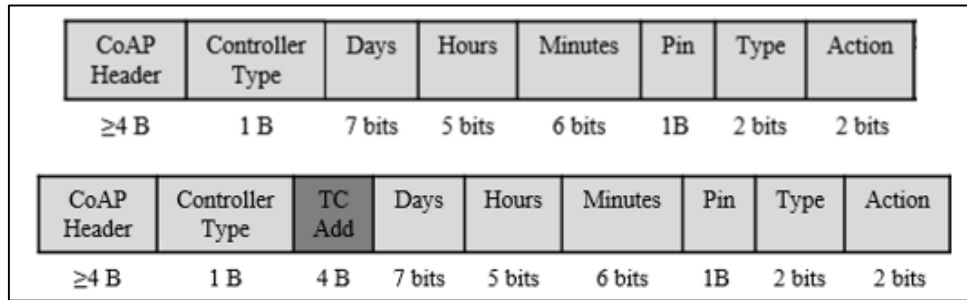


Figure 2-10. Simple control service message formats

The first message's destination is the central computer itself. The second is to be forwarded by the central computer to another controller through Wi-Fi or Bluetooth. The controller to which the message is forwarded parses the message and uses it.



Figure 2-11. Sensor monitor service message formats

Figure 2-11 shows the message formats of a service that requests data from a sensor wired to a central computer or to another controller communicating, through Wi-Fi or Bluetooth, with the central computer. The first is dedicated to the central computer and the second is to be forwarded. Table 3 explains each field and gives details about the usage of each. This table explains all the message formats and the reasons behind the field size and provide some examples.

Table 3. IoT message format fields

Field	Representation	Usage
Controller Type (CT)	The type of connection between the central computer (CC) and the appliance: wired to CC pin, using Wi-Fi interface or using Bluetooth interface. (2 bits to hold 3 values)	0: wired. 1: Wi-Fi 2 : Bluetooth
Target Controller Address	The address of the controller wired to the appliance. (4 bytes to hold the IPv4 Addresses)	CT = 0: this field is not added. CT = 1: The IP add. of the controller. CT = 2: The Bluetooth add. of the controller.
Days	The weekly days in which the requested service is to be performed. (7 bits – each for one day) 1 st bit is Sunday and the last is Saturday.	0: no appliance control on this day. 1: appliance control on this day.
Hours	The exact hour at which the service will be performed in each weekly day. (5 bits – for 24 hours)	The value of the control hour, from 0 to 23.
Minutes	The exact minute at which the service will be performed. (6 bits – for 60 minutes values)	The value of the control minute, from 0 to 59.
Pin	The pin number to which the appliance is wired to the controller. (1 byte – single-board computers have less than 256 pins but pins can be extended)	This pin number is used by the controller to control the appliance (On/off) wired to it, or to read data from the sensor wired to that pin.
Type	The type of service to be performed: control a single digital pin, request sensed data or control multiple pins. (2 bits to hold 3 values)	0: digital control of a single pin. 1: control of multiple pins. 2: read data from a sensor
Action/ Protocol	If the service is to control a digital pin, this field specifies if it is on or off. If the service is to read some sensor data, this field indicates the serial protocol used between the controller and the sensor. (2 bits for 4 values)	Control: 0: OFF 1: ON Protocol: 0: UART. 1: 1-Wire. 2: SPI. 3: I ² C
Num. of param.	This optional field is useful when multiple pins are controlled simultaneously. It tells how many pins (parameters) are going to be controlled. (4 bits – num. of param. fixed to 16)	The central computer uses this value to know the length of each of the following fields.
Type of param.	The nature of each of parameter: digital, PWM or SERVO. (Each type length is 2 bits).	0: PWM analog control 1: servo analog control 2: digital control
Param. Pins	The pin numbers of the parameters. (Each pin length is 1 Byte)	This tells the controller which pins to control specifically.
Param. Levels	The existing levels of each parameter. E.g. fast, normal and slow are 3 parameters. (4 bits each – fixed to 16).	The controller divides the highest PWM or servo fixed value to this value to convert the requested level value.
Requested Levels	The level requested to control currently (fast, normal or slow). (4 bits each – matches the previous field).	The controller uses the converted value of each to control a PWM or Servo appliance.

Once the central computer receives the message, it verifies if the control/monitor service is addressed to it or to another controller, by reading the *Controller Type* field. If it indicates that the service is to be performed by the central computer, it reads the following fields; otherwise it forwards the message to the targeted controller using the address provided in the *TC Add* field. The controller reads the *Days*, *Hours* and *Minutes* fields to see when to control the appliance. When it is time for controlling, the controller uses the *Pin* field to know which pin the appliance is wired to. It, then, uses the *Type* field to know what type of operation to do with the pin: to digitally control it or to read data from it. The *Type* field, also, indicates that there are extra pins to be controlled along with the given pin. The controller turns the given pin high or low based on the *Action* field value. If the operation is to read from the given pin, the *Action* field indicates the serial protocol to employ in order to communicate with the sensor wired to that pin. When the requested service is a simultaneous controlling of multiple pins, the controller reads the *Num. of Param.* field to know the number of pins to be controlled – this helps to parse the following fields. Afterwards, it reads the *Type of Param.* field to know by which operation (digital, PWM or Servo) it must control each pin. Reading the *Param. Pins* field, the controller knows the pins to be controlled. It converts the *Requested Levels* field's value to PWM or Servo value using the *Param. Levels* field and then apply the resulting value to the pin.

4. Mobile Application Design

We make a mobile application based on Android operating system. However, the concept behind it, which we will elaborate subsequently, is applicable to any other type of operating systems. The application has the aim of facilitating for the user to control and monitor any appliance connected to his IoT system. The

application is created using Android Studio, the official Android IDE that has Java as its official development language. In addition, we import a CoAP library in the Android project called *Californium*. This library is a sophisticated Java implementation of CoAP and has been developed by the Eclipse Foundation. We use mostly the client features it provides and in some cases the server functionalities.

The idea is to create an application where appliances and controllers' user interfaces can be dynamically added and deleted – so that they can be adjusted with any IoT system – and to put therein a variety of services. Figure 2-12 shows the main user interfaces of the smartphone application.

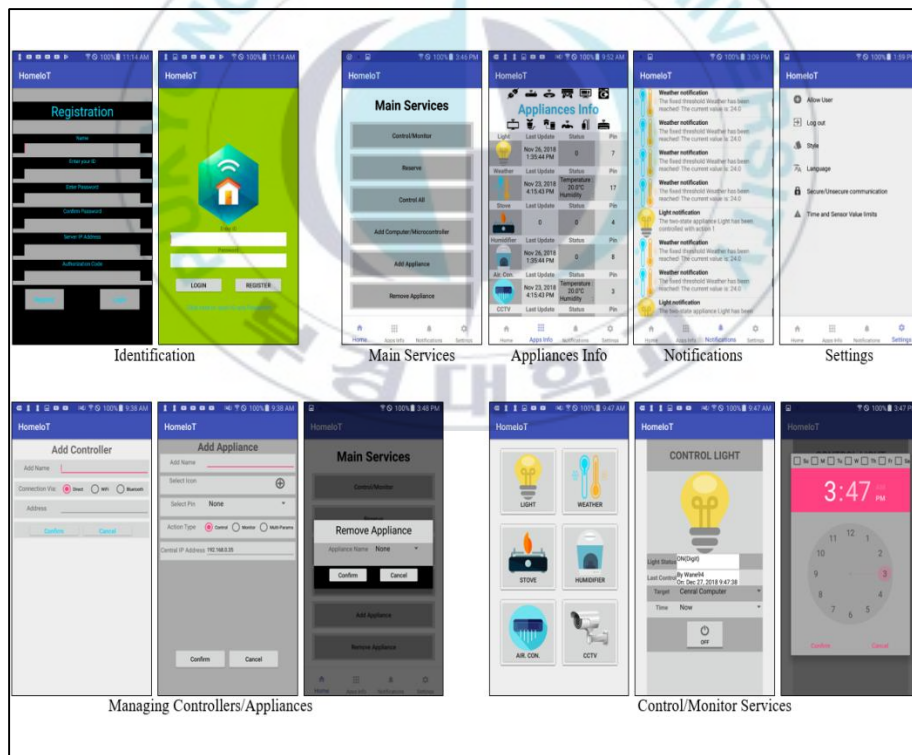


Figure 2-12. Android app user interfaces

Let us describe briefly the structure of the application activities. Opening the application, a UI listing the main existing services shows up. There are several buttons that direct to those services such as, adding controller, adding appliance, removing appliance, controlling/monitoring appliance, appliance information and notifications. These services can be provided in a form of an activity, Alert Dialog or Fragment.

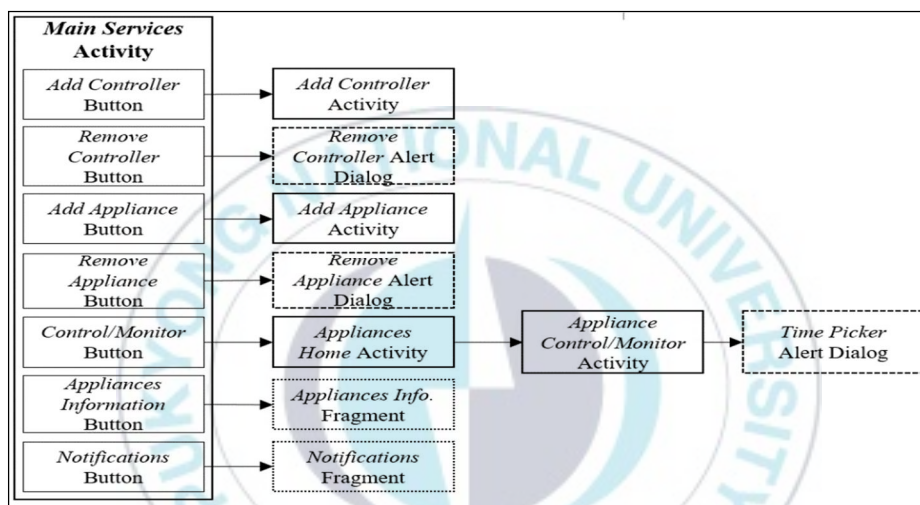


Figure 2-13. Activities structure

Figure 2-13 is a chart describing how the services are organized. We will detail subsequently the purpose of each operation. Clicking to a button in the main services activity redirects to an activity, fragment or alert dialog.

4.1. Add/Remove Controllers

As previously mentioned, a controller manages the controlling and monitoring of the appliances and sensors. The controller can be the main (or a supplementary) single-board computer, wired with the appliances and sensors via its general-

purpose input/output (GPIO) pins. In addition, it can be a microcontroller connected to the main computer through Wi-Fi/Bluetooth, wired with the appliances and sensors. It is important to add controllers' information in the smartphone application, especially when the system runs multiple controllers.

The controller's name, type and address are required to communicate with the smartphone. The name will be used to identify the controller in the smartphone. The type parameter is aimed at telling whether the central computer should act as a controller or as an intermediary device forwarding the requested service to the target controller. The central computer uses the address to forward the incoming service request to other controllers unless it executes the request directly. Figure 2-14 shows the user interface of the controller adding process.

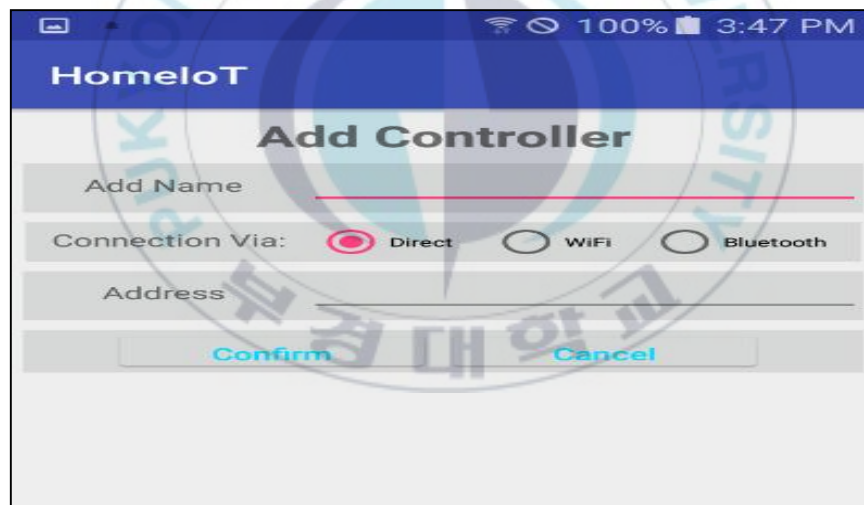


Figure 2-14. Controller add user interface

After supplying and confirming the necessary information by the user, it stores the information for later use. A long click on the main button can remove an instance of the controller.

Let us elaborate on the main idea behind the android activity and how the process works. Firstly, we create a document for storing the following variables: an array of names, array of connection types and an array of addresses. Whenever the information of a controller is provided for controller creation, each value is added in its corresponding array and the document is updated. The connection type is determined by the values returned from the radio buttons, meaning that the value '0' is assigned to the 'direct' button, '1' to 'Wi-Fi' and '2' to the 'Bluetooth' button. In the control activity, where the controllers' data is used, we retrieve the information of a controller by passing its position as a parameter to the arrays; it will be used to transmit the message to the appropriate destination and to add certain important information to it. When removing a controller, we simply remove its elements from the three arrays and then update the document with the new arrays. Figure 2-15 describes the adding process (a) and the removing process (b) of an element.

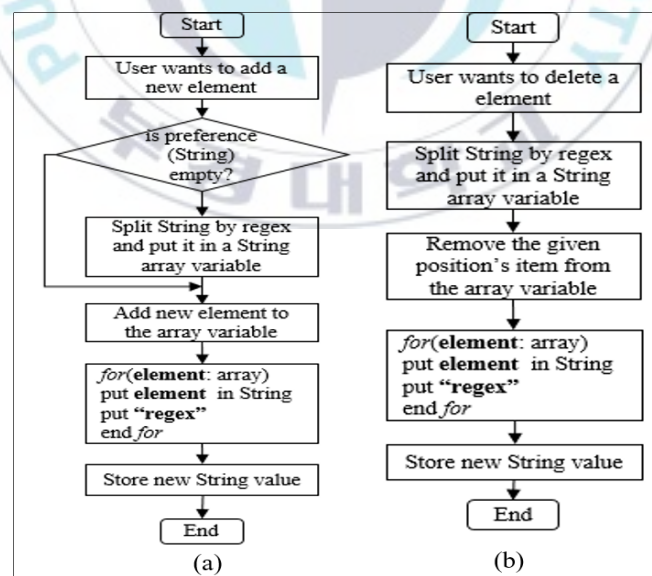


Figure 2-15. Adding/ removing elements flowchart

In a programmatic manner, we use the *SharedPreferences* Java interface to create and store the document, providing its name and mode. Then, using the provided *get* methods, we can add to this document as much as preferences we wish. But, as there is no explicit '*getArray*' method (for creating and storing an array), we have to manipulate the provided *get* methods with arrays to do so. We use the *getString()* method by which we will store all the items of the array in a single *String* separated by a regex. Then, if we need to restore the array back, we use the method '*split*' of the *String* class - which returns an array of string variables. Moreover, we use the *SharedPreferences.Editor* interface to edit preferences from the created document. Using the *putString()* method, we can edit the target preference by providing its key (name) and the new string value (preference). This reasoning is repeated throughout many parts of the Android application.

Also, as the information is stored in a cloud database for appliances' instant status update, we synchronize each local adding of a controller with a parallel creation of a controller document in the cloud database. However, the information added to the document is different; an array of pins and an array of the latest state of each pin are added.

4.2. Add/Remove Appliances

The user provides the necessary information to create an appliance virtual representation. The information consists of two major parts: one is the information associated with the particular appliance to be stored for user interface use, and the other is the information associated with that particular appliance for guiding the central computer to interact physically with the target appliance. The name of the appliance, an image, the type of appliance and the appliance's parameters are

required to create the appliance's user interface if the appliance is of a multi-parameter type. Figure 2-16 shows the UI for creating an appliance interface.

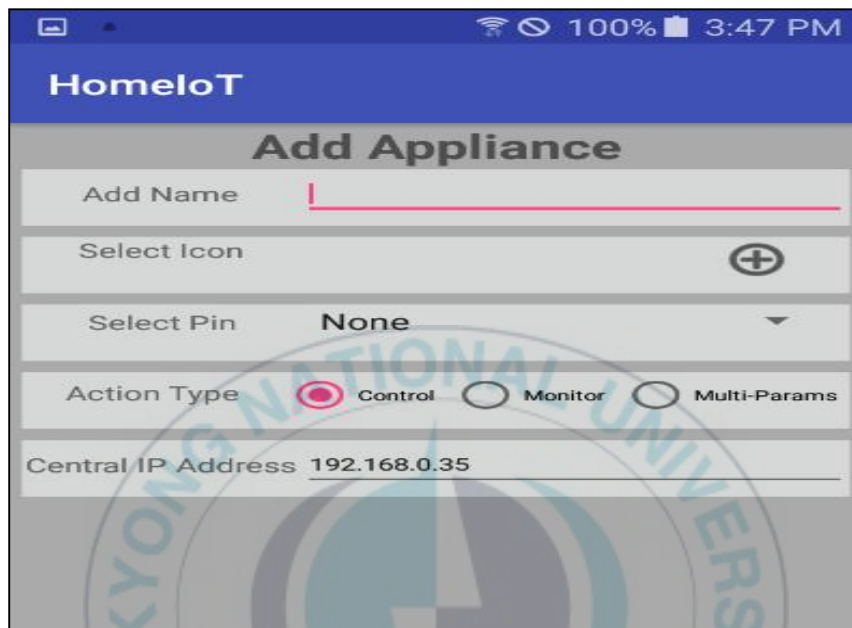


Figure 2-16. Appliance adding UI

This information will be stored and used to customize a button proper to the appliance. In the smartphone, the appliance's parameters information is used to display and customize the parameters' *SeekBar* views in the control page. Some of this information is also utilized to inform the central computer how to handle the requested service.

The removing functionality allows the user to remove a particular appliance virtual representation when he does not need it anymore. The user interface is shown in Figure 2-17. When the spinner is clicked, it displays a list of the existing appliances names, and, if selected and confirmed, all the stored data related to this particular appliance are deleted.

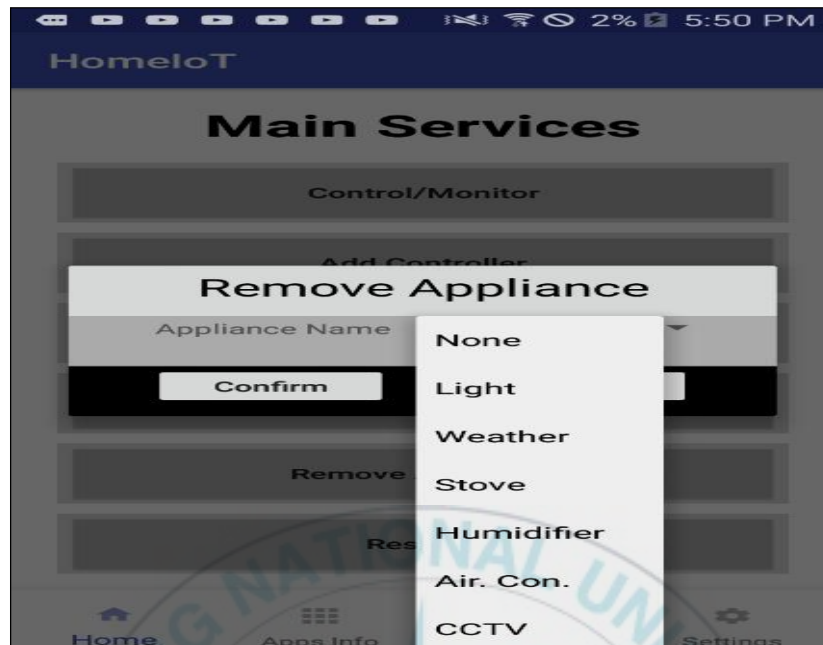


Figure 2-17. Appliance removing UI

Using the same reasoning as before, we add and remove appliance representation in an array fashion. Besides, an image is quite helpful for user to distinguish appliances. We use the Storage Access Framework (SAF) provided by Android 4.4 (API level 19) for this operation. As stated in the Android documentation, the SAF makes it simple for users to browse and open documents, images and other files across all preferred document storage providers [7]. There are many parts to be considered in SAF, such as document provider, client app and picker. We want our application to request an image from the existing document providers, not to play a role of a provider. Therefore, we are interested in the two last parts, in this particular operation, i.e. the client app and the picker. On Android 4.4 and onward, different from the previous version, Android 4.3, there is a possibility to use an intent which displays a system-controlled picker UI. This picker allows the user to browse all files that other apps (document providers) have

made available and pick from them. Figure 2-18 shows this picker having Google drive, cloud and USB as document providers.

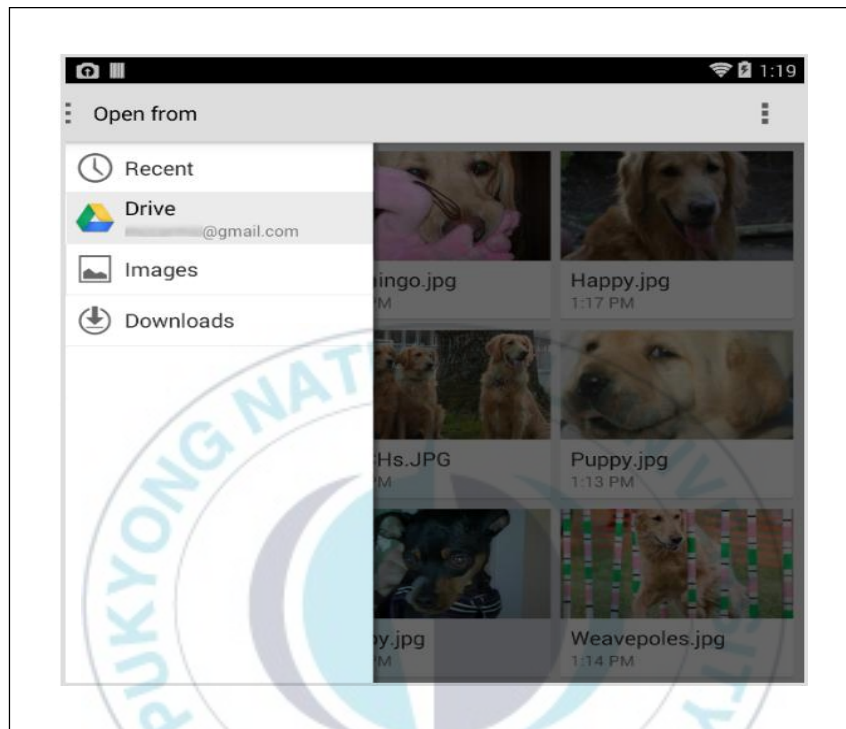


Figure 2-18. Android picker UI

Once a file is picked, its URI, not the file itself, is returned and can be used to open the corresponding image or save it for later use. Figure 2-19 describes the process exploited in our application as a flowchart.

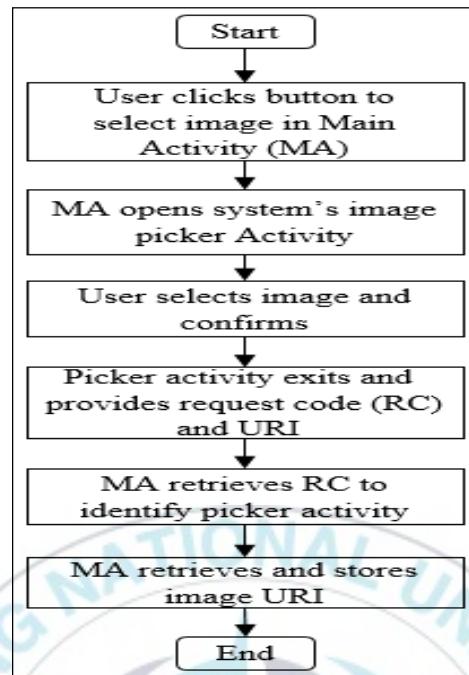


Figure 2-19. Image picking process

Once the button for selecting an icon is clicked, the intent `ACTION_OPEN_DOCUMENT`, the one to which we referred previously, is fired to open the system's picker user interface. But, as we are going to select readable image file types only, we, actually, filter the browsed files using methods that the intent provides before it gets fired. To filter, we set the category to be openable and specify only required image files. Then the intent is fired using the `startActivityForResult` method in order to identify the picker activity after the latter is closed. The activity is identified by a given request code, returned after exiting the picker activity. Right after the activity stops, the main activity calls the `onActivityResult`, a method called when a launched activity exits, giving the request code it was started with, the result code it returned, and any additional data from it [8]. In this method, we retrieve the mentioned

additional data, which is the URI of the selected picture, and put it in an array of image URIs that gets stored with the same reasoning as mentioned before.

4.3. Control/Monitor Appliances

After the appliance is virtually created, we can control the corresponding physical appliance, by sending a request to the central computer, the server, telling it to perform certain services based on the information in the given message. First, there is a user interface which lists all the created appliances as Figure 2-20 shows. Clicking on the appliance button from the list, the user is redirected to the appliance controlling/monitoring user interface of the appliance.



Figure 2-20. Appliances home

Let us describe what are the functions of the components in Figure 2-21, in order to gain an insight into what information the message must contain. It is an example of user interface for requesting a multi-parameter control service.



Figure 2-21. Appliance control

Actually, Figure 2-21 represents the appliance control/monitor page. There is a text view to show the current status or the result of the last request and another one to show when and which user has interacted with the appliance. In addition, there are two spinners for selecting, i.e. the target controller and the time at which the desired service should be performed. If the appliance has multiple parameters, the customized seek bars can be set at the desired positions configuring the levels of the corresponding parameter. After the user clicks the trigger button, the central

computer receives a request message which takes all the parameters at play into consideration.

The ‘appliances home’ user interface is a grid view, which contains buttons that redirect to the ‘appliance control/monitor’ user interfaces in a correlated fashion. User customizes the appliance name and image of each button. First, we should elaborate on how a grid view works in order to add items dynamically, and then detail how to customize the button.

There are different ways of implementing a grid view. We implement it with a cursor loader to make sure that all the data are shown at once. Just like a list view, we need to provide a grid view with an adapter to put content into its items. The adapter in question is, here, a class that extends the Java *ArrayAdapter* class. Briefly, the *ArrayAdapter* takes an array or multiple array variables and passes to each item of the list view a value of the same position from the provided array. As stated in the documentation, by default, the array adapter creates a view by calling *Object.toString()* on each data object in the provided collection, and places the result in a text view [9]. However, a user may also customize what type of view is used for the data object in the collection, by overriding the *getView(int, View, ViewGroup)* method and inflating a view resource. Since our content is a button consisting of image and text, we must customize the type of view. Therefore, we create an XML layout file in which we properly add and set a button view to its root layout. Afterwards, in the *getView()*, we inflate and return the created XML layout file (view) in order for the button to be displayed instead of the default text view.

As now the button view is inflated, we can customize it by employing each item of the ‘appliance name’ array passed to the array adapter as the buttons’ texts. In addition, we convert each item of the ‘appliance image’ array (URI) to a

drawable variable type and use the drawables as the buttons' images. The conversion is done by using the *openInputStream(Uri)* method from the *ContentProvider* class to open a stream on the content mapped with the provided content URI. The variable returned by this method is an *InputStream* object. We then convert it to a *bitmap* using the *decodeStream()* method from *BitmapFactory* class, and in turn, convert it to a *drawable* by the help of the *BitmapDrawable* class. The conversion is necessary here, because the method that we use requires *drawable* parameters. This method can set the *drawable* to appear at the start of, above, to the end of, and below the button text [10]. The information provided during the appliance creation is what we used to customize the buttons.

Actually, some of the 'appliance control/monitor' user interface child views are enumerated previously. Let us describe their background operations. The image Uri is retrieved from the stored 'appliance image' data. Then the method *setImageUri(Uri)* of the *ImageView* object associated with the image view appearing in the user interface, is called while passing it to the retrieved Uri as a parameter.

As for the seek bars they are actually added to a list view in order for the user to add dynamically as much seek bars as he wishes – so let us keep in mind that we will customize the seek bars in the custom array adapter class. As we already discussed about grid view implementation, the list view works in a similar way. Nonetheless, the seek bar implementation was not mentioned. As we need to get each seek bar's selected position, we create an array list having the items equal to '0' each. The array list size is determined by the number of parameters added when creating the appliance. After, we set the seek bar object to listen for any change and if any, it should add the selected position value to the array list. As the array list is an instance variable, we access and use it in the control activity class.

There are two spinners displayed; one for selecting the destination controller and the other for choosing the controlling time. Both of them call the *setAdapter()* method to implement their corresponding array adapter objects. We instantiate the array adapter and pass to it, as parameter, the data it should adapt to each spinner's *TextView*. The stored 'controller name' array is passed as parameter in the case of the destination controller's spinner. A click on a spinner item appends the associated type and address of the destination controller to the IoT message. As for the 'time spinner', an array list containing two string variables, "now" and "later", is provided. A selection of the "now" *TextView* sets the time value as negative – which means the controlling must be instant. Concerning the "later" *TextView*, a selection opens up an *AlertDialog* window in which the weekdays and a clock are shown as depicted in Figure 2-22.

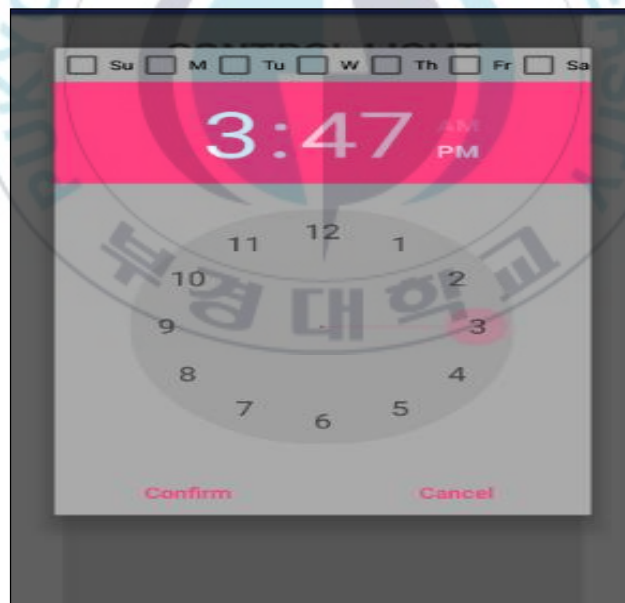


Figure 2-22. Time picker

The weekdays are represented by *CheckBox* views. Checking a box adds an integer '1' to the weekdays' array which was initially filled with seven '0's. While unchecking it, changes the value back to '0'. Each day is identified by its position in the array. For example, Sunday's position is '0' and Monday's is '1' and so on. A value '0' at position '2' means there is no controlling in Tuesday. On the bottom of the weekdays' boxes, we inserted a time picker view. This time picker can be used to indicate at which moment, exactly, the appliance should be controlled. We set its associated Java object to listen for time change, and instruct instance variables (hour and minute variables) to take the values returned by the overridden *OnTimeChanged()* method. Then after clicking the confirmation button below the clock, the time variables will be inserted somewhere in the message that will be sent for appliance controlling service. The delay is performed by the central computer.

4.4. Appliances Information

Being able to get certain information about each added appliance, all at once, can be very helpful in managing the IoT system properly. Notice that it can take time to verify the relative information one by one in a system with various appliances. Therefore, we created an interface that provides the user with information related to the status of each appliance, i.e. the last update time and the corresponding pin number of the appliance at the controller side. The interface represents the status of the controllable appliances as '0' for deactivated or as '1' for activated, respectively. If the appliance is a sensor type, it displays the latest data sensed after a request, as shown in Figure 2-23.

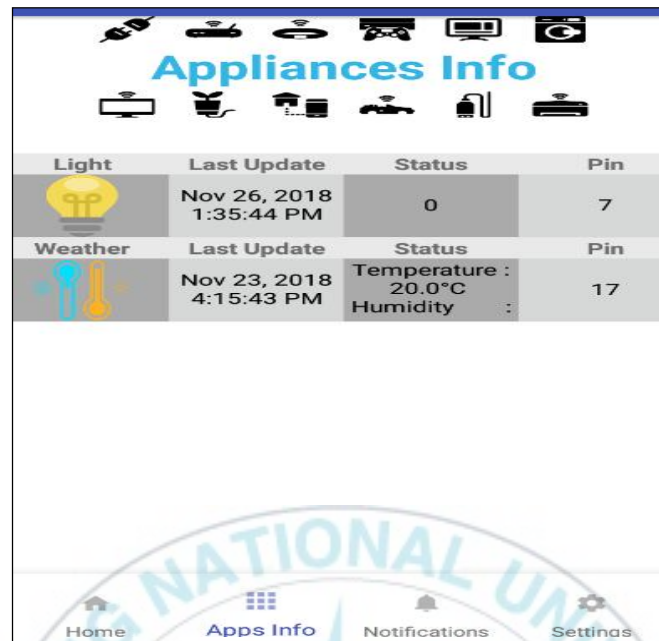


Figure 2-23. Appliances information

This user interface uses list view to add existing appliances and their information dynamically. The appliance name, image and pin are stored in the smartphone, whereas the latest update and the appliance state or sensor data are stored in the cloud database.

4.5. Notifications Functionality

The system has notification services for the user in certain defined cases. When executing notification services, the phone vibrates, its light flashes and the user interface displays a bar that contains the information about the service. The system performs a notification after a service execution, if the request was of a delayed type. In addition, the system informs the user when a sensor senses a value higher than the threshold value and when an appliance is activated more than the fixed

time interval set by the user – if the feature is enabled. After each notification, the notification information is grabbed and saved in the smartphone for the information to be used in the notifications logging list. This helps the user manage the notification policy.

Notice that a notification service can be initiated either by the central computer or by the phone itself. The former happens when delayed-controlling and sensor tracking services are handled in the central computer environment, whereas the latter case is when the appliance tracking service is based on the smartphone system. Either way, when requesting a notification service, each one provides a message following a simple format: type, pin and activity (reason for notification). By the help of this format, the message can be parsed into a list of notifications. Figure 2-24 shows the flow of the notification process.

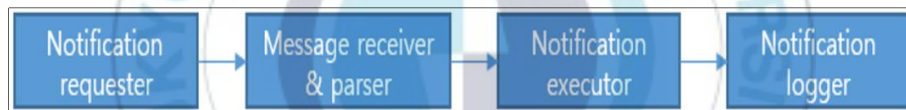


Figure 2-24. Notification process

The key element needed here is the application working in background even if the user interface is closed. We first start by creating a class for implementing the server features provided by the Californium package. This class *extends* the Java service class. In the method *onStartCommand()*, we start the server by calling a *start()* method from a *Californium CoapServer* object that we already create. In other words, stating: “when the service is started, start the server”. The service is started as soon as the application is run. There are some parameters to set for the server implementation such as specifying the URL and handling the request by responding and retrieving the payload. The payload is parsed and the type, pin and

data values are retrieved. The *notifier()* method that we created takes these variables as parameters in order to use them for notifying properly. The type is used to set the notification content that will be shown in the notification bar, such as “The fixed threshold has been reached! ...”. The pin is used to identify the appliance; using a search method which finds the position of the pin in the stored pins, the position value is used to retrieve the corresponding image and name. Then they are used for notification bar icon and title respectively. These are the only components required to notify. But as we already stated that the notifications must be logged, we must save them. Firstly, an *Integer* variable that tracks the number of existing notifications is created. Initially equal to ‘0’, this number is locally saved. Anytime the *notifier()* method is called, it increments this value by 1. This index value helps to dynamically create a *preference* for each notification. The preference names will be as following: ‘notification0’, ‘notification1’, ‘notification2’...; then in each one the notification data is stored.



Figure 2-25. Notifications list UI

Basically, in the *Californium* package, the CoAP message is created by writing the data to be sent (header, options and payload bits) bit by bit in a byte variable. Once the byte variable is complete (all the 8 bits added), the whole byte is then put in a length-varying byte array called *byteArrayOutputStream* (to hold it until the whole data is ready). Then the byte variable is emptied and the 8 following bits are written to it to repeat the same process again and again until the whole data is added to the *byteArrayOutputStream*. As the data is going to be send using a datagram packet, there is a need to convert it into a supported variable type. A datagram socket, in Java, takes a simple byte array as payload, therefore the data in the *byteArrayOutputStream* is retrieved to be written in a byte array. And finally, the byte array is put in the payload section of a datagram packet which will be sent by the datagram socket bound to an address and port set by the user. Figure 2-27 explains the message reception process in the *Californium* package.

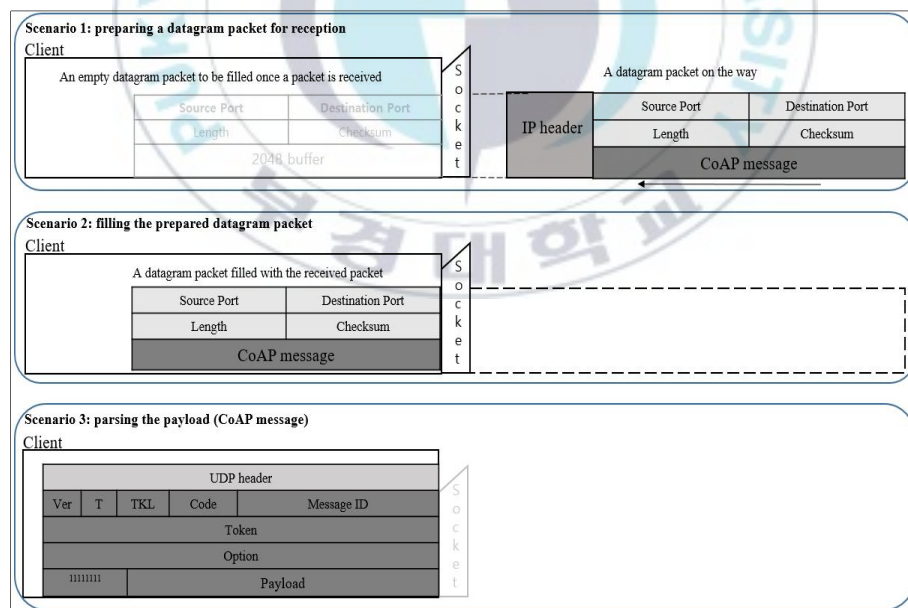


Figure 2-27. Californium data reception

A call of a request method starts, at least, two threads: *Sender thread* and *Receiver thread*. Once run, the *Receiver thread* instantiates a datagram object of 2048-byte buffer size that will be used later on to store the data to be received (the CoAP message). Afterwards, a socket bound to a local address and local port is ordered to receive the datagram packet corresponding to its address and port by filling the previously prepared datagram object's buffer with the recently received data. Now, as there is a payload in the datagram packet representing a CoAP message, it is read and parsed. Afterwards, the received CoAP payload can be retrieved. Note, the *Receiver thread* is run automatically if the server operation is launched.

5. Controller Design

The central computer, a Raspberry Pi single-board computer, runs *Raspbian* as official operating system[11]. We chose JavaScript as the programming language to perform the various operations required in the IoT system, such as the networking and the interaction with the appliances. We execute the codes using the *Node.js*, a JavaScript run-time environment built on Chrome's V8 JavaScript Engine. *Node* is an asynchronous event driven JavaScript runtime, designed to build scalable network applications and is well suited for the foundation of a web library or framework [12]. A supplementary Raspberry Pi controller, connected through Wi-Fi with the central computer, must run the same code as the central one.

The Arduino microcontroller has its own IDE that uses the Arduino Programming Language to upload some code to the Arduino board. The Arduino Software (IDE) runs on Windows, Macintosh OSX, and Linux operating systems [13]. Each controller has I/O pins through which it

interacts with the appliances or with interfacing modules, e.g. a Bluetooth chip.

5.1. CoAP Server

There exist various open-source CoAP libraries available for backend. We use one of them, called *node-coap*, among various programming languages' libraries. The *node-coap* is a *Node.js* library that provides various APIs for implementing the defined CoAP functionalities [14]. When we make call of these APIs and the functions they provide, the Raspberry Pi can listen for CoAP requests and respond after executing the services.

The CoAP message operates over UDP. Therefore, the library uses the *Node.js* datagram module and builds CoAP on it by serializing the payload part. Before going into details about the library, let us describe briefly how we can write a code using the datagram module to implement a CoAP server functionality. Figure 2-28 shows a code using the *Node.js* datagram module for creating a server.

```
1 var dgram = require('dgram');
2 var server = dgram.createSocket('udp4');
3 server.on('message', function(msg, rinfo) {
4   console.log('Message Received : ' + msg);
5 });
6 server.bind(8080);
```

Figure 2-28. Node.js UDP server implementation

We begin by importing the datagram module using the *require* keyword. The module provides an implementation of UDP datagram sockets through a call of its *dgram.createSocket('udp4')* API – let us call and store it in a variable named 'server'. Then from the 'server', we call the event listener that emits a callback function after detecting an incoming message event. The first parameter of the callback function represents the message as a *Buffer* object and the second one is an object that contains the address information of the sender. Inside of this function, we can retrieve the message and other optional information from the request by writing all the work that we want to be performed after message reception. Then we use the *server.bind(port)* function to precise on which port the socket should listen and optionally which address.

The *node-coap* uses the datagram module to make a server, the same way as we just did – which means the *node-coap* server can be boiled down to a datagram socket handling requests. The only difference is that *node-coap* has added more code to perform all the CoAP requirements; hence, it provided APIs that are layers on top of the datagram module in order to simplify the code.

```
1  const coap    = require('coap')
2  const server  = coap.createServer()
3
4  server.on('request', function(req, res) {
5    res.end('hello there!')
6    console.log('Message received: ' + req.payload.toString())
7  })
8
9  server.listen(3000, '192.168.0.10')
10 console.log('server started')
```

Figure 2-29. Node-coap CoAP server implementation

In Figure 2-29, a JavaScript code is shown as an example of implementation of the *node-coap* server. The *node-coap* CoAP server implementation can be, basically, mapped to the datagram code shown previously. The *createServer()* function ultimately calls the *createSocket()* function of the *dgram* library. Also, the event listener *server.on()* of the *dgram* is the base of the *server.on()* of the *node-coap*. The message variable of this event listener is parsed as a CoAP message by the *node-coap*. The *server.bind()* is a low-level function of the *node-coap server.listen()* function.

The *coap.createServer()* API, creates a CoAP server. Suppose that we call the API and store it in a variable named 'server'. Then we call an event listener which fires a function after receiving a 'request' event. The callback function's parameters include *request* and *response*; from the latter we can end the communication and optionally by responding with some text. As for the *request* variable, the payload such as the sender's address and other information can be retrieved from it. Again, it is in the curly brackets of this callback function that every service execution conditional to a request is written. For example, the appliance controlling and monitoring operations are written inside it; just to tell the controller: "interact with the appliance when you receive a request". Finally, a function called *server.listen()*, mappable to the *server.bind()* of the datagram socket, is used to specify on which port the server should be listening and optionally on which address. If the address is not provided, the server will accept connections directed to any IPv4 or IPv6 address by passing *null* as the address to the underlining socket [*server.bind(null)*].

In our turn, we parse the *request* object by following our defined IoT message format that we created and use it accordingly. Additionally, from the *request* object we can retrieve the path in order to direct the request to the proper service.

5.2. Controller-Appliance/Sensor Interfaces

The controller, a Raspberry Pi or Arduino, controls and requests data from the appliance through its pins. The *Raspbian* OS comes, by default, with a Python library called GPIO Zero. This python library provides various interfaces to interact with the GPIO pins for multiple purposes, such as input devices, output devices, SPI devices and internal devices.

Let us now dive into the actual libraries that we are directly using as interfaces. We, actually, use the *node-wiring-pi* library; which is a *Node.js* binding to the C library, *wiringPi*. This library has the main features provided by the GPIO Zero and more. Pins can be controlled in a digital way or also through analog means. Through some of the pins, the controller communicates with certain devices using protocols such SPI, I²C or UART. However, some sensors with other protocols should use their corresponding libraries.

The user can download the Arduino IDE from the official Arduino website. In this case, there is no external libraries needed for digital and analog controlling or for serial communication. Nevertheless, a sensor such as *DHT* temperature and humidity sensor can use its own communication interface, which requires using the corresponding library.

5.3. Reservation

A reservation is a delayed service requested by the user. Here, the idea is to take a request message, compute the waiting period for the service to be performed and instantly call a timeout function that presents the instruction of ‘when and which service to perform’. However, we also want to handle other reservations without overwriting the existing one. Therefore, we create object

variables in which the timeout and the request message are stored. If there is any incoming reservation request, the values of this objects are verified, and the related information is stored if available; otherwise, it jumps to the next until it finds another object available. After a service is performed, the object is emptied to hold reservation information. Figure 2-30 represents the pseudocode for implementing the reservation process.

```

1  container0 = {Timeout: -1, Request: 0}
2  container1 = {Timeout: -1, Request: 0}
3  ...
4  container10 = {Timeout: -1, Request: 0}
5
6  Retrieve days values from request object
7  days = [sun, mon, tue, wed, thu, fri, sat]
8  for(dayIndex = 0; dayIndex < 7; dayIndex++)
9      if(days[dayIndex] == 1)
10         compute timeout
11         switch(-1)
12             case container0.Timeout:
13                 container0.Timeout = timeout
14                 container0.Request = request
15                 perform the service of container0.Request after container0.Timeout
16                 container0.Timeout = -1
17                 container0.Request = 0
18         break
19     end case
20     case container1.Timeout:

```

Figure 2-30. Reservation pseudocode

Firstly, the pseudocode defines ten objects for containing each request's information. Note that the initial timeout values are '-1'; which means it is available for containing data. The code also retrieves and gathers the days' values from the received request object, in one array variable. A *for* loop is used to iterate the week days' values (0 to 6), where each day's value is compared with '1'; which asks: "is the service to be performed on this day?". If the condition is true, it computes the timeout to wait. Then using a switch statement, an available container is fetched and used to store the information. Then the instruction of launching the request service after the defined delay is given using the

setTimeout() function. Finally, after the service execution, the object is reinitialized.

6. Cloud for Multiple Users Interface

In order for a user to know about the latest status of his appliance, we can track each message from the central computer. However, consider a scenario, where there are multiple users in a system. A response message makes only the user who requested the service aware of the change, but not the others in the system. Therefore, we need to find a way to deliver instantly the user-appliance interaction information to each user presented in the system.

We use a cloud-hosted database from Firebase called *Cloud Firestore*, in order to update publicly the latest information about each appliance in the system. As depicted in Figure 2-31, although each smartphone communicates with the central computer, they indirectly communicate using cloud as interface.

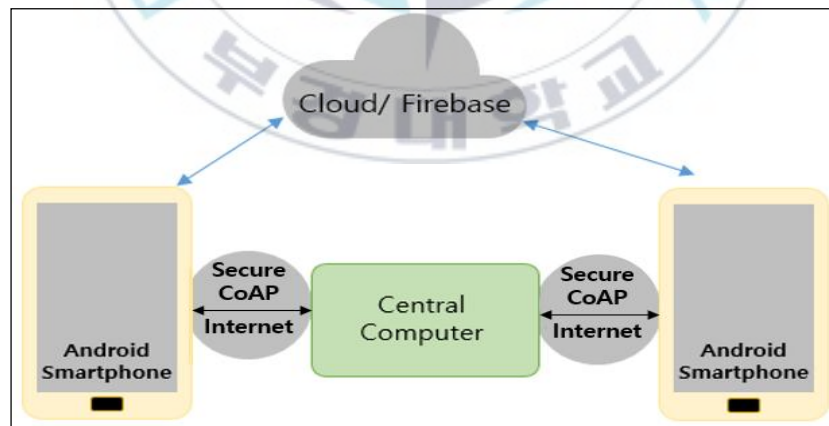


Figure 2-31. Cloud as interface for multi-users

If a user successfully controls or monitors the request, the request result will be stored in the cloud database created for this project. Let us discuss briefly about Cloud *Firestore* – the way it operates and how we implement it in our Android application.

As defined in the Firebase documentation, Cloud *Firestore* is a flexible and scalable NoSQL database for mobile, web, and server development from Firebase and Google Cloud Platform [15]. Most importantly, it keeps the data in sync across client apps through real-time listeners. The data storage model of Cloud *Firestore* is built upon data, document and collection. The data is stored in a document that contains fields mapped to values. In turn, the collection stores the document. Additionally, various types of data are supported.

After creating a Firebase project that we correlate with our Android Studio project, we add the required dependencies in order to import the Firebase library in our project [16]. Now that we have the APIs provided by Firebase accessible, we can create collections and documents to add some data. In order to synchronize the number of existing controllers with the documents created, whenever a controller is created, a corresponding document containing an array of 0s as data is created – ‘0’ represent the pin’s current state and its position in the array represents the pin number. To do that, we call a method that takes the path of the document as a parameter, called *document(“path”)*, to create and get the *DocumentReference* object in which we will add a *Map* type of data containing the array. For creating a specific document for each newly created controller, we keep track of the number of created controllers by incrementing a created integer by 1. After, we append the current controller number to the document name. Here the initial data (0s) are added but not the actual state of the pin or the data sensed by that pin from a sensor.

In the ‘control’, ‘monitor’ and ‘appliances information’ activities, the actual data gets retrieved and modified after successful requests. We start by creating an object of the document reference associated with the selected controller. From this object, the data is retrieved, modified and updated after calling some event listeners. To update the data instantly, we write this part of the code in the Java *Activity* class’s *onStart()* method. For simplification sake, we stated that we add one array variable representing pins but, in reality, there are two other arrays that we add in parallel with the values representing the users that interacted with the pins (appliances) and the time of interaction.

7. Service Scenarios

In this section, different service scenarios are shown. The first scenario, basically, represents two ways of an activation and deactivation of an appliance exemplified by a LED and fan. The second is an illustration of the user requesting sensed temperature and humidity data which is an example of sensor data monitoring. The final scenario shows the central computer starting the communication and sending a notification to the Android smartphone.

7.1. Appliances-Controlling Procedures

Figure 2-32 pictures the Android smartphone requesting from the central computer to turn on a lightbulb wired to a specific pin. The central computer, which acts as a server in this case, checks the URL and parses the payload in order to perform the requested action. The central computer performs the operation first by putting the given pin high, and after then, sends back a response message along

with a piggybacked acknowledgement message to the android client for confirmation.

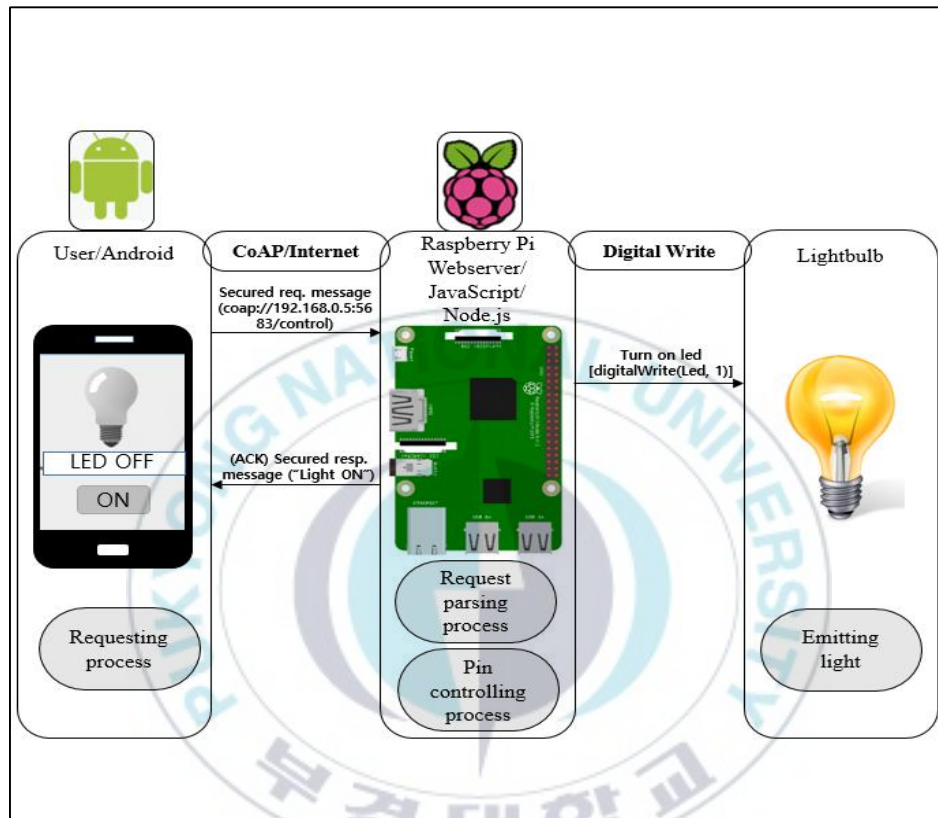


Figure 2-32. Appliance controlling process

The scenario depicted in Figure 2-33, involves communication through the Internet using the CoAP web transfer protocol, communication via Wi-Fi between two Raspberry Pi controllers and communication via Bluetooth between the second Raspberry Pi and an Arduino controller. Then, a controllable fan is wired to the Arduino controller.

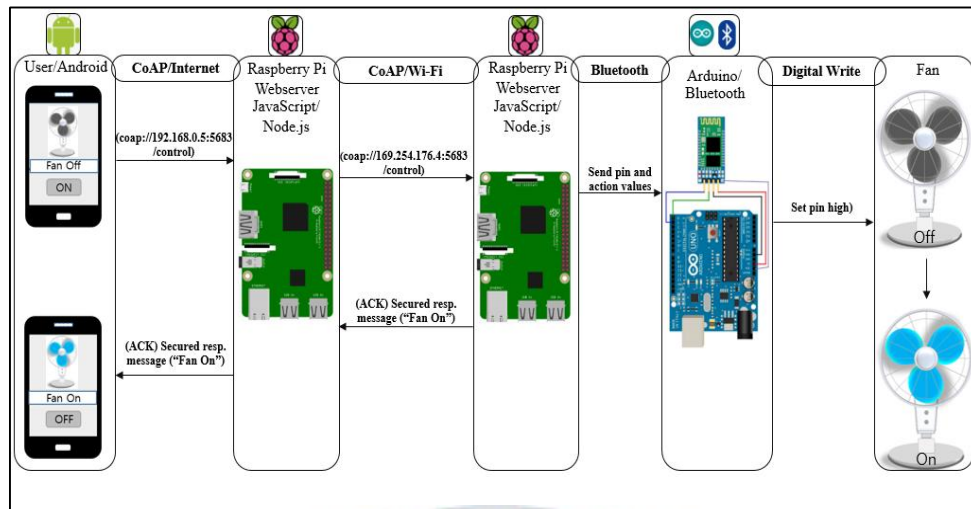


Figure 2-33. Appliance controlling process (multi-interface)

When the ‘On’ button is clicked, the smartphone sends a request to the central computer to control a given pin. The central computer checks if the message is destined to it or has to be forwarded. As the message is to be forwarded, it reads the address and forwards to it the CoAP message through Wi-Fi. After receiving, the controller checks the destination and then forwards it to the Arduino controller through Bluetooth. The Arduino controller then performs the operation. After transmitting the message to the Arduino controller, the controller that has direct link with it sends an acknowledgement to the central computer. The latter, in turn, sends it to the smartphone.

7.2. Requesting Sensor Data Procedure

Figure 2-34 shows the user, through his Android smartphone, sends a request to the central computer requesting the temperature and humidity values. It consists of a sensor from which data will be read after a request.

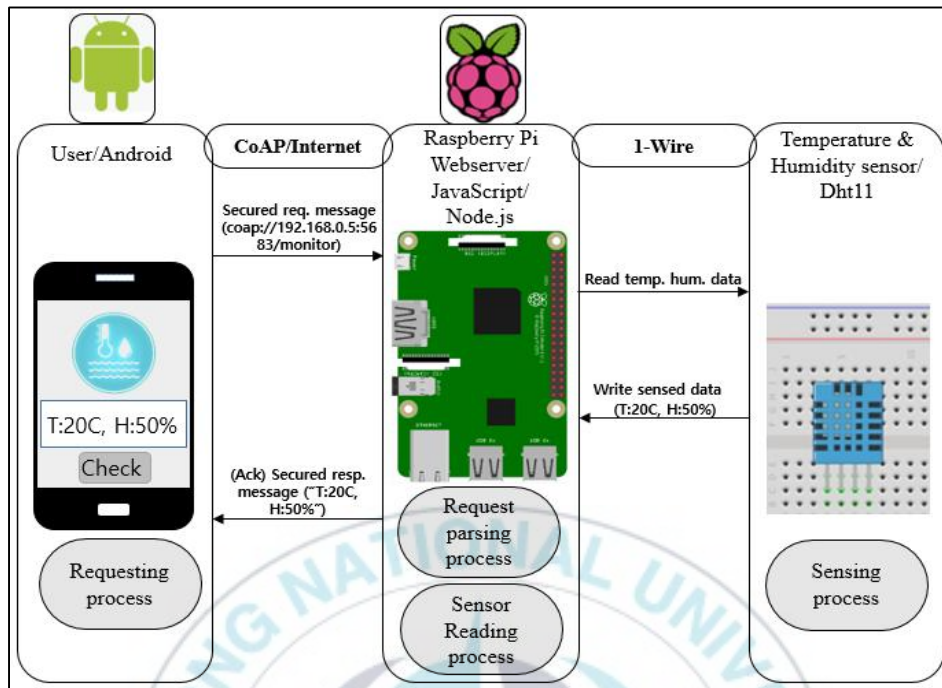


Figure 2-34. Sensor data request procedure

The central computer reads, via 1-Wire serial communication, the data sensed by the ‘temperature and humidity’ sensor in order to send it along with the acknowledgement message. In case, the sensor data is not ready, the server sends an empty message for acknowledgement and then sends the data, once ready.

7.3. Notification from the Central Computer

Figure 2-35 depicts the central computer initiating a communication with the smartphone. This scenario is, typically, for notifying the user that the current temperature is higher than the threshold.

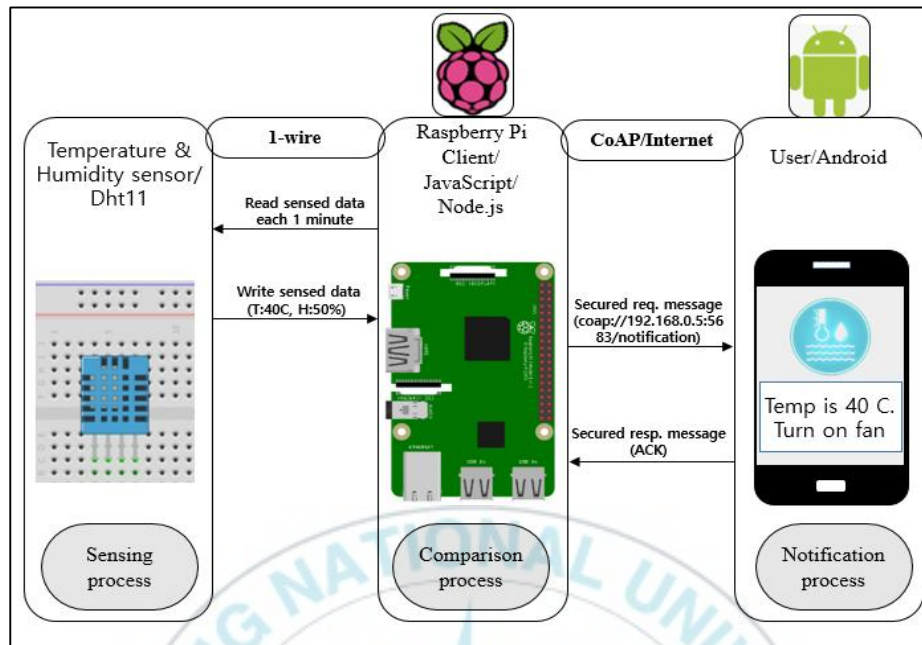


Figure 2-35. Notification-sending procedure

After comparing the sensed temperature with the threshold temperature, a request message is sent by the central computer to the smartphone to inform that “high temperature” is sensed. Then the smartphone sends an empty message response back to terminate the communication. In this scenario, the central computer is the client and the smartphone is the server.

8. Security Implementation

The messages that are transferred between the smartphone and the central computer of the system have to be authenticated before taking any action. This is to ensure that the message comes from an identified user and can be taken into account to interact with the system. Therefore, the messages are transmitted with an authentication tag appended to them.

To generate a tag from a particular message, we need, at least, three components: a secret key, an encryption mechanism and the plaintext. Let us elaborate on techniques used to generate secret keys securely between the two parties and the encryption algorithm used to generate the tag.

8.1. Key Exchange

One technique of key generation, consists of using a third-party server which authenticates and issues keys to the communicating parties. Here, as shown in Figure 2-36, instead of using a third party server, we systematically generate secret keys using the Diffie-Hellman Key exchange method [17]. We implemented it in Java programming language for the mobile phone and in JavaScript for the Raspberry Pi central computer.

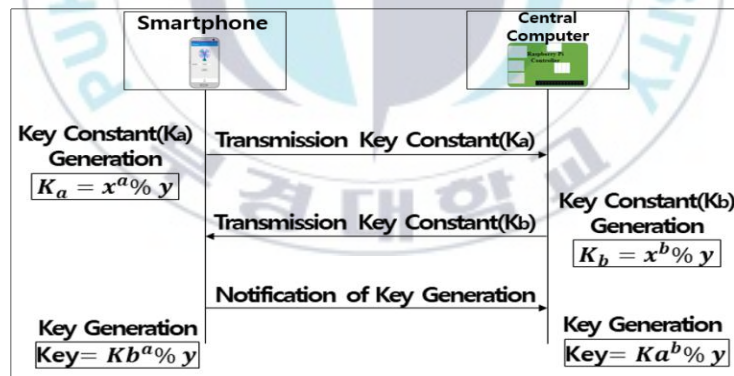


Figure 2-36. Key generation procedure

This method helps us generate secret keys by relying on complex computational difficulties. The process, in a simplified way, goes like the following. The communicating parties decide which values they use; these values have to be

related by one being the modulus and the other one being its base (the primitive root modulo of this modulus value). One party can decide and send this combination to the other publicly, as the values are not meant to be confidential. Each one computes the commonly shared values with a selected secret key using a defined formula. Afterwards, each one transmits the computed value to the other party publicly. Now, using a known formula, each one computes the received value with its secret key. And, voila, the result is a pair of secret keys of at both sides. This method, as it relies on computational difficulties, big numbers must be used and keys should be constantly regenerated.

Let us explain the process relatively to our system. The private constant (a or b) are decided by each party individually and the shared constants (x and y) are selected by the smartphone. The private constant is 1-byte value that each party knows secretly, and the shared constants are the values shared by both sides and each is 1 byte. Once the key constants exchange is complete, the smartphone sends a command of 'Notification of Key Generation' to the central computer to generate the key.

8.2. Tag Generation

The encryption is done by applying the LEA to the Cipher Block Chaining mode (CBC) of WPA2 [18]. Referring to the documentation, we created a Java LEA implementation for the tag generation and verification at mobile phone side. For tag generation and verification at the Raspberry Pi side, as we are employing the *Node.js*, we created a JavaScript implementation of the LEA. Figure 2-37 shows the integrity authentication tag generation method for the IoT system to execute operations in units of 4 bytes.

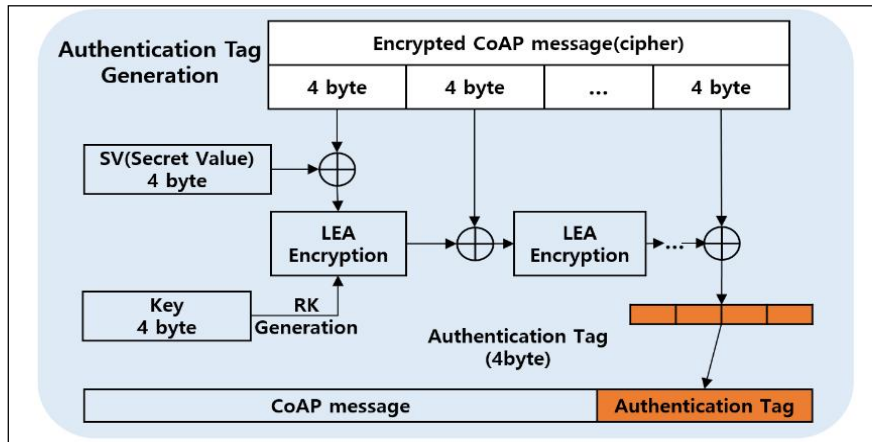


Figure 2-37. Authentication tag generation

We first encrypt the CoAP message by the LEA mechanism – not depicted. We, then, use the resulting cipher text as input for tag generation method. A 4-byte secret value (SV) and the 4-byte key generated previously are also used as inputs. In this process, we make use of the LEA algorithm which employs the XOR operation essentially and produces a result of 4-byte value. We use the whole 4-byte value as tag and append it to the end of the CoAP message.

9. Performance Comparison

9.1. CoAP vs. HTTP

As we know, we employed the CoAP protocol for web services instead of the HTTP protocol. CoAP allows low overhead and multicast; which are some of the key features required by constrained devices. Figure 2-38 depicts CoAP and HTTP packet captures.

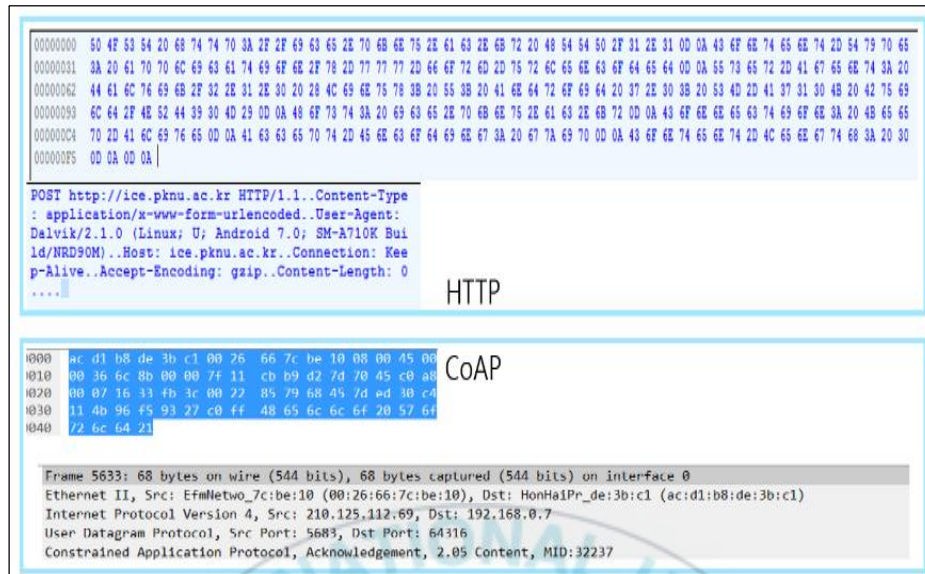


Figure 2-38. HTTP and CoAP packet content

Note that CoAP is more efficient than HTTP in the context of IoT because it has a shorter packet size. While CoAP employs UDP, HTTP uses primarily TCP which uses complex congestion controls. Although, reliability is an issue in UDP, CoAP defines a retransmission mechanism to compensate for it.

9.2. Advantages of Public Key Exchange

We proposed the message to be transmitted with an authentication tag to ensure that it has not been altered along the way. This requires an encryption algorithm and a key exchange method, as previously discussed. For the key exchange method, it is more convenient and cost-efficient to use a public-key cryptography such as Diffie-Hellman Key exchange, instead of symmetric-key cryptography. Because, no third party is required and keys can be publicly exchanged.

9.3. LEA vs. AES

We used the WPA2 with the LEA encryption for encrypting messages and generating tags. We compared this combination with the WPA2-AES encryption combination. Figure 2-39 is a graph illustrating how the LEA algorithm takes less time to perform encryption and tag generation compared to the AES algorithm.

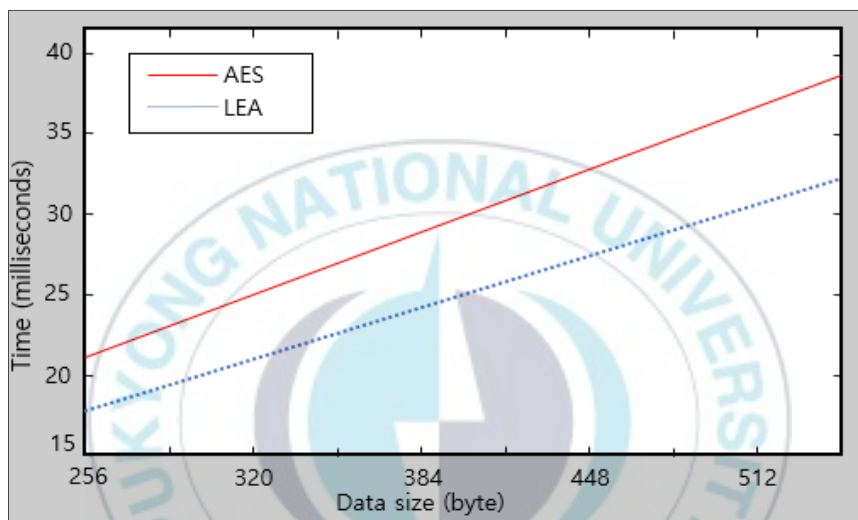


Figure 2-39. LEA and AES performance graph

We took samples by augmenting the data size by 64-byte units, from 256 bytes to 512 bytes. The data size being 256 bytes, the LEA takes 17 milliseconds to accomplish the task. For the same task and same data size, the AES requires 21 milliseconds. When the data size is 512 bytes, the LEA takes only 32 milliseconds in contrast to the AES that takes 37 milliseconds. The difference might not be significant in this case, but as the data size grows significantly the difference in time becomes huge as the graph shows. This shows that the LEA mechanism is more performant than the AES.

III. IoT Framework for Road Accidents Mitigation

Statistics show that road traffic crashes are the reason why more than 1.25 million people die every year [19]. These numbers do not include those who are injured and paralyzed in result of road traffic accidents. It has also been reported that the road traffic accidents are the 10th leading cause of deaths in the world and is expected to become the 7th by the year 2030 [20]. According to the World Health Organization (WHO), Africa is the continent that has more road disasters compared to others, followed by Asia. These alarming numbers suggest that extra measures need to be taken for road traffic safety in all possible areas.

Furthermore, road accidents are caused by various factors such as human, meteorological or technical cause. However, human causes arrive far ahead of meteorological or technical causes. Most of the time, it is a combination of causes. A combination that involves both purely human causes (the driver and other users), meteorological causes, technical etc. The human factor appears in more than 90% of the road accidents. Accidents that have purely meteorological or purely technical causes are therefore extremely rare. Speed is one of the major human causes.

According to the WHO, speed is not only playing a major role in road accidents occurrences, but also determines the severity of the resulting injuries. In developed countries, the rate of road accidents involving speed is 30% and 50% in underdeveloped countries [21].

Thus, speed impacts the frequency of road accidents as well as the severity of their damage. We, therefrom, concluded that controlling road traffic speed will

decrease the death rates in underdeveloped countries significantly and in developed countries as well. We propose an IoT system for controlling vehicle speeds in a centralized fashion.

1. Overall System Structure

Figure 3-1 illustrates the overall system structure by depicting the main idea and main components. It shows a vehicle in a cell area covered by a base station (BS) and a neighboring cell. Each BS is connected to a mobile telecommunications switching office (MTSO).

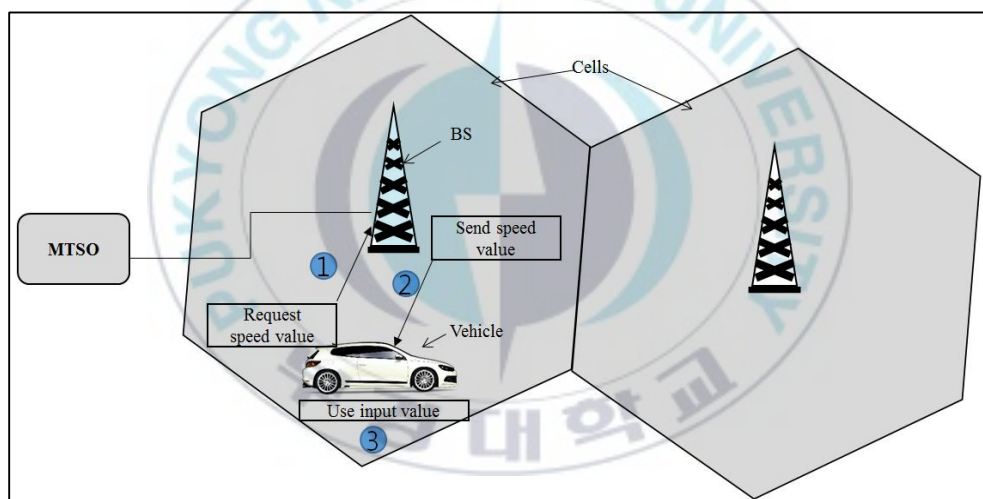


Figure 3-1. System structure

As soon as a vehicle equipped with the system enters a cell, it will monitor for the strongest signal. As neighboring cells use different control channels, it will select the channel of the current cell. After the connection establishment, the vehicle sends a message to the BS, requesting for the speed limit values suitable for this area. The BS forwards the request to the MTSO, where the speed limit

data are stored. The latter replies, sending back the proper speed limits. The vehicle retrieves the speed limit values from the message and then uses them until it enters a different cell. However, as shown in Figure 3-2, a cell can have various speeds limits.

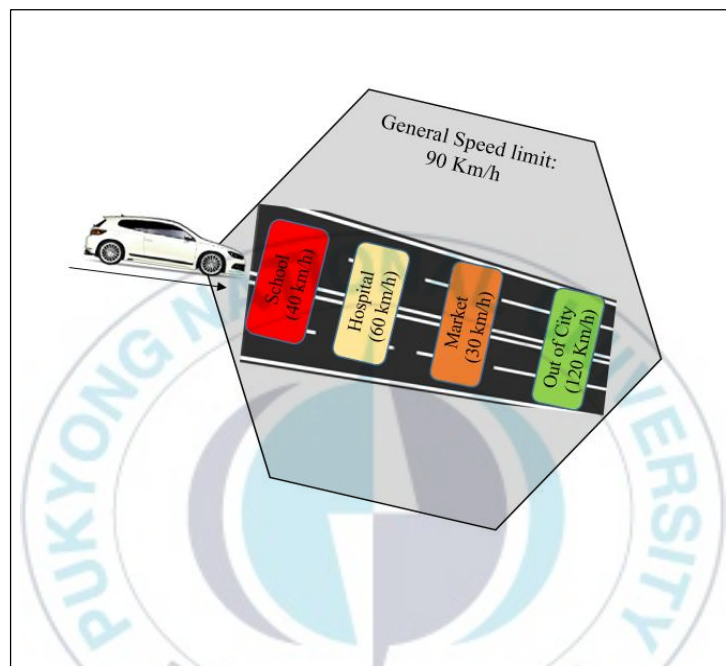


Figure 3-2. Multi-speed limits cell

This is where the exact position of the vehicle comes into play. Consider a hypothetical cell having multiple different speed-limited zones. The vehicle, therefore, needs all the cell's speed limit values in one response message or requests multiple times. Figure 3-3 shows one way to manage this in a flowchart fashion.

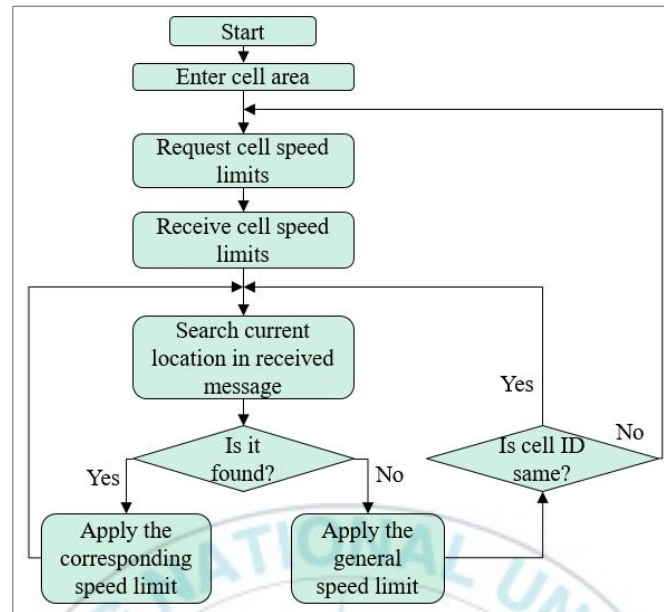


Figure 3-3. Multi-speed limits management

After entering a cell, the vehicle sends a request for all the cell's speed limit values and receives a response containing the speed limits and their corresponding locations. Tracking its location, the vehicle computes whether it matches with one of the specifically limited locations. If it matches, it applies the corresponding speed limit. Otherwise, the vehicle runs at the general speed limit and keeps monitoring the cell ID. If the cell ID changes the process repeats with the new cell.

2. Road Speed Limiters

There are various technologies which attempt to limit the speed of vehicles using a multitude of methods and providing different functionalities [22]; they are called Road Speed Limiters (RSL). They differ in functionality and in technique. One RSL type has the functionality of only monitoring the vehicle's speed and once it reaches a determined threshold, the driver is alerted. Another type limits the

speed but gives the driver the option of disabling the functionality at any time with simple pedal or manual manipulation – these methods assume that the driver is a law-abiding citizen and able to monitor the speed limits. However, there is a RSL type which governs the speed and do not provide handy functionality for modifying the system once set up – unless stopping the car and inputting a higher speed threshold or removing the RSL. The latter is the one that suits the most our project. This RSL is implemented in various ways taking into consideration, the type of vehicle and the speed sensing method.

Traditionally, a vehicle moves by burning fuel which, in turn, releases heat energy – called combustion. In the combustion process, for efficiency, a quantity of fuel is mixed with a controlled amount of air by the carburetor. Therefore, greater is the amount of perfect fuel-air mixture burned, faster the vehicle moves forward. This means, controlling the quantity of fuel, air or both will result in controlling the vehicle's speed.

The RSLs that target the fuel control are: The direct fuel control type and the cable type. They control the amount of fuel pumped into the engine but at different location of the vehicle.

The cable type, controls the speed by controlling the accelerator pedal's cable going to the throttle that controls the fuel flow. This system comprises a motor, an Electronic Control Unit (ECU), a cable and a speed sensor. The cable is connected to the throttle's cable at one end and to the motor at the other end. The motor is connected to the ECU which controls it. The ECU compares constantly the set speed with the current speed provided by the sensor. If the sensed speed exceeds the threshold, it triggers the motor which pulls out the cable. This reduces the throttle pressure and therefore the speed of the vehicle. The Direct Fuel Control

type of RSL is depicted in Figure 3-4. It comprises of a speed sensor, an ECU and a Fuel Control Unit (FCU).

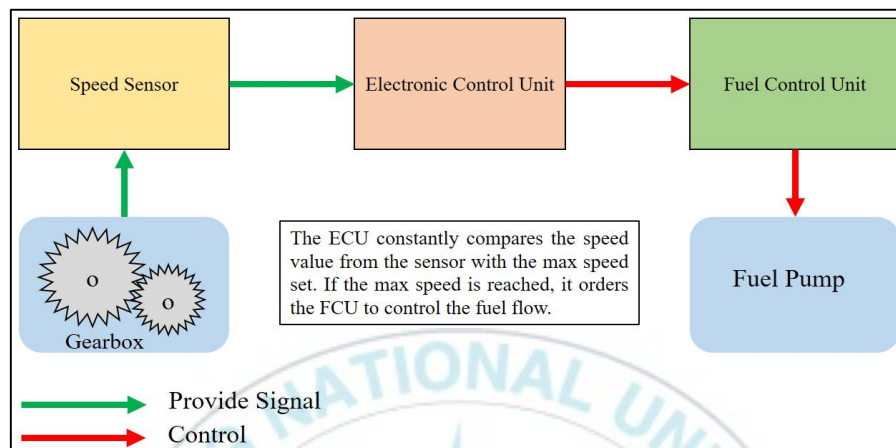


Figure 3-4. Direct fuel control type of RSL structure

The vehicle's gearbox provides the current speed to the speed sensor – although, other means can be used to sense speed. This sensor forwards the data to the ECU which, in turn, compares constantly the current speed with the maximum speed – typically the maximum speed is inputted to the ECU by a separate device when setting the system. If the current speed is found to exceed the defined threshold, the ECU sends a signal to the FCU ordering it to reduce or stop the fuel flow. The FCU has an opening through which the fuel enters coming from the fuel pump and another opening at the other side to let the fuel flow to the carburetor. As the fuel has to pass through the FCU, the ECU orders the FCU to close (or reduce) the openings to block the flow when required.

There is also a type of RSL which is solely dedicated to vehicles equipped with electronic engines and electronic accelerator pedals [23]. This type of electronic vehicles has, typically, Engine Control Modules (ECM). In this case, the only extra

components required are the ECU and a speed sensor. The ECU monitors the speed given to it by the speed sensor and compares it with the set maximum speed. If the speed signal is beyond the preset speed, the ECU reports it to the ECM to set the speed. This limits the speed of the engine at the desired speed.

3. Location Techniques

Finding a location of a cellular device using merely cellular means is an implemented and developing technology. For instance, in various part of the world, cell phone localization is used for emergency purposes. E911 in North America, E112 and eCall in the European Union are systems used to locate a caller automatically; localizing a cellular phone is one of the services they provide. The eCall service is used to rapidly locate vehicles involved in collisions and has been made mandatory to vehicles in the European Union. There are two methods of wireless localization used in these systems. One uses the cellular phone's built-in GPS functionality to get information about the position of the cellular device; the other one uses radiolocation in the cellular network. Let us discuss the basic idea behind radiolocation.

Radiolocation is, basically, the process of using radio waves to locate objects. Cellular networks use a set of radio towers to locate an object through radiolocation. There are several techniques adopted and are usually based on the received signal strength (RSS), angle of arrival (AOA) and time of arrival (TOA) [24]. These techniques might be combined with other complementary techniques as well. One of the most used techniques is known as trilateration. In trilateration, three cell towers determine a position of a mobile by computing the RSS or the TOA. Figure 3-5 (a) illustrates the idea behind this technique. Figure 3-5 (b) highlights the signal strength or the time of arrival.

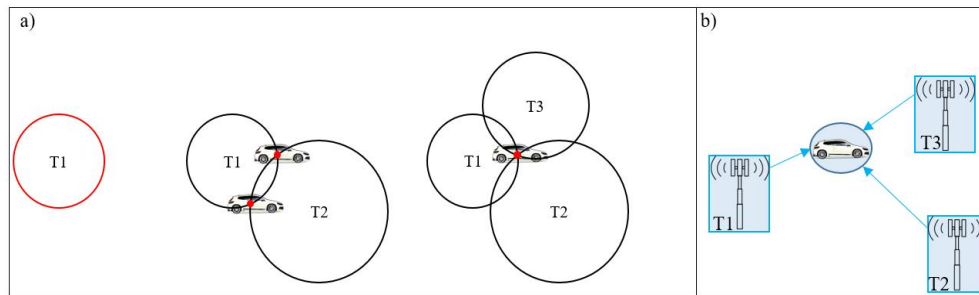


Figure 3-5. Trilateration scenario

In the case of one cell tower used to determine the position of the vehicle using the RSS or the TOA, the possibility is an entire circle with the tower (T1) as the center (a). However, using two towers (T1 and T2), the possible position is only one of the two intersections of the circles. A third tower (T3) helps to decide which point is the actual position, as it will be the intersection of all the three circles. In (b), the length of the arrows between a tower T and the vehicle can be considered as the strength of the signal or the time of arrival.

In real life, however, due to short-term fading, shadowing and reflections caused by topographical limitations, along with the trilateration extra algorithms are needed to estimate the position of an object. In addition to this issue, a moving object is far more difficult to locate than a stationary one. Besides, the GPS technology also use trilateration.

Especially for vehicle location, a system that locates vehicles automatically is known as Automatic Vehicle Location (AVL) or Automatic Vehicle Monitoring (AVM). These systems use radiolocation and, specially, trilateration as location techniques as stated in[25].

4. Vehicle Cellular RSL Prototype

As there is always an ECU involved in the RSL system, we will replace the it by a Raspberry Pi single-board computer. This requires only knowing the specifications of the ECU and the functionalities it provides and then implement them as a software. Ideally, the single-board computer should be provided with a simple cellular interface and a simple GPS module or a module supporting both functionalities. Then, in a single software, it manages the request/response messages, geolocation and the controlling of the Motor, FCU or ECM. This type of cellular/GPS shields are available but due to cost and specific and complex requirements, we decided to replace it with an Android smartphone instead.

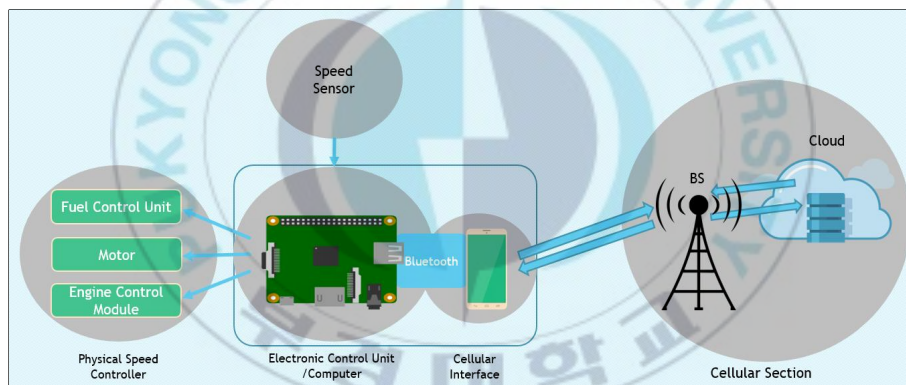


Figure 3-6. System prototype

As we use an Android smartphone as a cellular interface, we will program it to take care of making requests, handling responses and using them with geolocation data, while the Arduino microcontroller takes care of the controlling of the physical speed controllers. This makes it easier to implement cell based location functionalities as well as GPS location functionalities. Also, there are more cellular/GPS APIs provided in Android programming. Note, the smartphone is not

supposed to be a user interface device and should not be separated from the microcontroller, as it will be connected with the latter through Bluetooth.

The Raspberry Pi will be provided with a HC-05 Bluetooth module to communicate properly with the smartphone. They communicate through UART serial protocol by wiring the HC-05's *Rx* pin to the Raspberry Pi's *Tx* and *Tx* to *Rx*. For the datacenter, the data of each zone, cell, speed limits and other details are stored in a cloud database. We will use the Firebase's *Firestore* cloud database platform for the implementation.

For the speed sensor, we use the technique that combines an encoder wheel with a LED and a phototransistor in order to compute the speed at which the dummy vehicle is running. As for the physical speed controllers, we will try to emulate the FCU and the ECM. The FCU, as stated previously, regulates the fuel flow to control speed. In order to represent the FCU, we will use a linear servo. The extending operation of the linear servo's rod blocks the hose, while its retraction opens the hose. The ECM will be represented by an Arduino Uno microcontroller – the emphasis is on the communication protocol not the functionality of the ECM. The Raspberry Pi single-board computer sends to it the signal and, in turn, it “regulates the speed”.

4.1. Raspberry Pi Software Implementation

In the case of FCU prototype, the Raspberry Pi single-board computer receives the pseudo-speed limit value from a UART serial port and then uses it to control the linear servo. Here, we use a *Node.js* library called *node-wiring-pi* to use the UART functionalities. The code is shown in Figure 3-7.

```

1 var wpi = require('node-wiring-pi');
2 wpi.setup('wpi');
3
4 var controlPin = 9
5
6 var uartPort = '/dev/ttyS0' //The cellular module is connected to this serial port
7 var uart = wpi.serialOpen(uartPort, 9600) //we set up the RX serial port and baud rate
8
9 while(wpi.serialDataAvail(uart)){ //when serial data is available the following code will be run
10     var speedLimit = wpi.serialGetchar(uart) // get the speed limit value
11
12     wpi.pinMode(controlPin, wpi.OUTPUT)
13
14     /*Let us assume that the speed limits range from 30 to 130
15     The extension interval : 0 to 100*/
16     var rodPosition = speedLimit-30
17     wpi.softServoWrite(controlPin, rodPosition);
18 }

```

Figure 3-7. FCU prototype code

Firstly, we have to define which serial port we are using – ‘/dev/ttyS0’ is one of the standard Raspberry Pi’s serial ports. Then, we pass it as a parameter to the *serialOpen()* function along with the baud rate. This returns a file descriptor from which we can retrieve the transmitted data by calling the *serialGetchar()* method. Do not get distracted by the pseudo-formula used to calculate the rod position in relation to the speed limit – it is only to show that they should be correlated. Then, the resulting value is passed as a parameter to the *softServoWrite()* function along with the value of the pin to which the servo device is connected. Thus, the linear servo’s rod is controlled resulting in speed control.

In the case of ECM prototype, after receiving the pseudo-speed limit value from a UART serial port, the Raspberry Pi forwards it to the Arduino Uno board via the I²C serial protocol. The task of the Arduino is out of the scope. The emphasis here is on the communication protocol employed to transfer data

between the Raspberry Pi device and the Arduino microcontroller – the I²C protocol. Figure 3-8 shows the code for implementing this operation.

```
1 var wpi = require('node-wiring-pi')
2
3 var uartPort = '/dev/ttyS0' //The cellular module is connected to this serial port
4 var uart = wpi.serialOpen(uartPort, 9600) //we set up the RX serial port and baud rate
5
6 while(wpi.serialDataAvail(uart)){ //when serial data is available the following code will be run
7
8   var speedLimit = wpi.serialGetchar(uart) // get the speed limit value
9
10  var I2CAddress = 0x8 //The I2C port of the ECM
11
12  var i2c = wpi.wiringPiI2CSetup(I2CAddress) //we specify the I2C slave to send data to
13
14  wpi.wiringPiI2CWrite(i2c, parseInt(speedLimit)) //send data to the ECM
15
16 }
```

Figure 3-8. ECM prototype code

We firstly define the I²C address of the Arduino board and pass it as a parameter to the *wiringPiI2CSetup()* function to set up a communication with it. The returned value of this function is a file descriptor to which we can write data by calling the *wiringPiI2CWrite()* function. The data is the speed limit; it is sent to the Arduino board via I²C protocol.

4.2. Android App Programming

The Android application's primary role is to interface between the Raspberry Pi and the cellular network. It will make requests for speed limit values and handle the responses. After retrieving the list of speed limit values of the current cell, it will constantly check the vehicle's current location and find its corresponding

speed limit from the list and forwards it to the Raspberry Pi in case the speed limit is different from the previous one. It transfers data to the Raspberry Pi via Bluetooth communication and expects a confirmation from it. After the Raspberry Pi device confirming that it has controlled the speed to the Android app, the later sends a confirmation back to the cloud database. This process repeats until the App sends a 'stop' signal to the server on behalf of the driver. Furthermore, the App collects the vehicle's location data using cellular network and GPS.

Another role of the App's role is also to subscribe and add the user's information to the database. The information for subscription will be the identity number of the user and the vehicle's ID – an algorithm checking the validity of the provided information should be implemented at the backend. The user interface for registering a new user is shown in Figure 3-9.

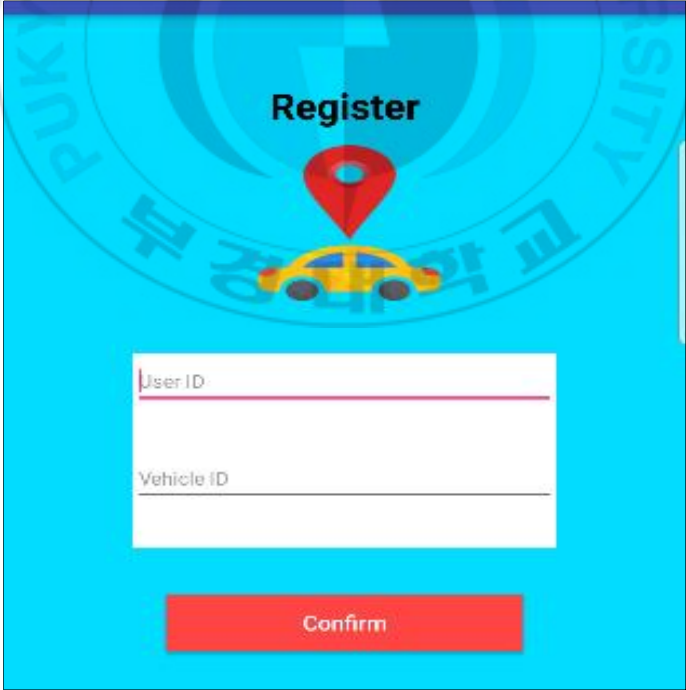
The image shows a mobile application interface for user registration. The background is a solid blue color. At the top center, the word "Register" is written in a bold, black, sans-serif font. Below the text is a red location pin icon, and directly beneath that is a yellow car icon. In the center of the screen, there is a white rectangular box containing two input fields. The first field is labeled "User ID" in a small, grey font, and the second field is labeled "Vehicle ID" in a similar font. Below these input fields, at the bottom center of the white box, is a red rectangular button with the word "Confirm" written in white, bold, sans-serif font. A large, faint, circular watermark is visible in the background, featuring a compass rose and the text "PUK TONGKONG INTERNATIONAL UNIVERSITY" around the perimeter.

Figure 3-9. Driver-vehicle information registration UI

Let us, firstly, illustrate how the user's information is retrieved and sent to the database. Then, we describe how it makes requests and makes use of the responses. After, we provide details about the location implementation. Finally, we describe how the Bluetooth functionality is implemented and how it forwards the speed limit values.

The user inputs his ID and vehicle ID in the edit text areas and confirm by clicking on the button labeled “confirm”. The information is retrieved and sent to the server which uses an algorithm to check the validity and compatibility of the provided information in order to store it.

As mentioned previously, we use the *Firestore* cloud database platform. We need to create a Firebase project and link it to the Android project. Firebase provides Java APIs to write and read from the project database's documents. In short, in *Firestore* database, fields (Strings, arrays...) are stored in documents; and documents are put in collections as shown in Figure 3-10.



Figure 3-10. Firestore project's data structure

When retrieving document's data, we need to provide its path to the *document()* API as parameter. We can fix the collection name but the document name has to be dynamic. Actually, the document name is based on cell IDs. Therefore, when requesting data, we need firstly to retrieve the vehicle's current cell ID and provide it as the path to the document. Now from the document we can get the fields that we need.

Related to the database, in each document, we added three fields: an array of locations (positions) values, a separate array of speed limits and a general speed limit. A vehicle needs to use its location data and figure out if its location (or approximate) is in the location array. When a corresponding value is found, it uses the position of the value in the array to retrieve the speed limit value from the speed limit array. If no correspondence is found, it falls back to the general speed limit of the cell.

In the Android app side, let us describe how we can make use of the APIs. Figure 3-11 is a snippet showing how to get an object of a *Firestore* document and the way to retrieve the fields stored therein.

```
cellDocument = FirebaseFirestore.getInstance().document("documentPath: "Position-Speed/Cell"+cellID);

cellDocument.get().addOnSuccessListener(new OnSuccessListener<DocumentSnapshot>() {
    @Override
    public void onSuccess(DocumentSnapshot documentSnapshot) {
        if (documentSnapshot.exists()) {

            positions = (ArrayList) documentSnapshot.get("Positions");

            speeds = (ArrayList) documentSnapshot.get("SpeedLimits");

            gen_speed = (long) documentSnapshot.get("GeneralSpeed");

        }
    }
});
```

Figure 3-11. Acquisition of data from Firestore

First, we instantiate the *Firestore* class, which is an entry point for all *Firestore* operations, and call its *document()* method. This method takes a path of an existing document and returns a *DocumentReference* object. Notice that the path is composed of the name of the collection followed by the name of the document. The document name is a concatenation of strings; a string “Cell” and a cell Id appended to it – we will show how to get the cell Id subsequently. Now that we have a *DocumentReference* object, we can call its *get()* method which returns an object from which we, again, call the *addOnSuccessListener()* method. This adds a listener that is called if the data request task completes successfully. Then we need to override the *onSuccess()* method, to state what to do when the task succeeds. From this method, we retrieve each field from the *DocumentSnapshot* object it provides by calling the *get()* method and passing to it the key of each field. As the field values come as *Object* types, we need to cast them into the desired variable types.

Let us briefly, discuss about the way we get the cell ID on the smartphone. First, one needs to get an object of the *TelephonyManager* class in order to use telephony services on the device. After, we need to request permission to make use of the services. Then we can call the *getCellLocation()* method which returns an object of the *CellLocation* abstract class. This abstract class is implemented by the *GsmCellLocation* and the *CdmaCellLocation* classes. Therefore, to make use of the returned object, we need to cast it to either one of them – we need to do for both but separately. Basically, we state if the *GsmCellLocation* object is equal to *null* – which means the phone is not using GSM currently – then try retrieving the location data from a *CdmaCellLocation* object. Both of them provide a method that returns an integer representing the ID of the current cell the phone is connected to.

Now that we retrieved the data from the cloud database and that of the current location, we need to create helper algorithms for making use of them. However, let us understand the bigger picture and then go into the helper method in detail. Figure 3-12 is a snippet of this bigger picture.

```

LocationListener locationListener = new LocationListener() {
    @Override
    public void onLocationChanged(final android.location.Location location) {
        final double current_latitude = location.getLatitude();
        final double current_longitude = location.getLongitude();
        cellID = getIpBaseStation();
        cellIDocument = FirebaseFirestore.getInstance().document( documentPath: "Position-Speed/Cell" + cellID);
        cellIDocument.get().addOnSuccessListener(new OnSuccessListener<DocumentSnapshot>() {
            @Override
            public void onSuccess(DocumentSnapshot documentSnapshot) {
                if (documentSnapshot.exists()) {
                    positions = (ArrayList) documentSnapshot.get("Positions");
                    speeds = (ArrayList) documentSnapshot.get("SpeedLimits");
                    gen_speed = (long) documentSnapshot.get("GeneralSpeed");
                    int pos_in_array = findPosition(current_latitude, current_longitude, positions.toArray());
                    long currentSpeedLimit = gen_speed;
                    if (pos_in_array > -1) {
                        currentSpeedLimit = Long.valueOf(speeds.get(pos_in_array).toString());
                    }
                    bluetoothConnector.send(currentSpeedLimit);
                }
            }
        });
    }
}

```

Figure 3-12. Cloud/location data usage

We can say that the content of Figure 3-12 is the core of the App. The *onLocationChanged()* method is repeatedly called when the device moves few miles and/or after a lapse of time. This is why we need to do the main tasks inside of it. Firstly, we retrieve the location coordinates and then request data from *Firestore*. Afterwards, we use a helper method to compare those variables to see

whether the cell's general speed limit should be used or one the specifically defined speed limits; the helper method is shown in Figure 3-13.

```

public int findPosition(double currLat, double currLong, Object[] cellLocations) {
    int p = -1;
    if (cellLocations.length > 0) {
        for (p = 0; p < cellLocations.length; p++) {
            double latitude_start = Double.valueOf(cellLocations[p].toString().split( regex: "/" )[0].split( regex: ";" )[0]);
            double longitude_start = Double.valueOf(cellLocations[p].toString().split( regex: "/" )[0].split( regex: ";" )[1]);
            double latitude_end = Double.valueOf(cellLocations[p].toString().split( regex: "/" )[1].split( regex: ";" )[0]);
            double longitude_end = Double.valueOf(cellLocations[p].toString().split( regex: "/" )[1].split( regex: ";" )[1]);

            if (((latitude_start <= currLat + 0.001 && latitude_start >= currLat - 0.001)
                && (longitude_start <= currLong + 0.001 && longitude_start >= currLong - 0.001))
                || ((latitude_end <= currLat + 0.001 && latitude_end >= currLat - 0.001)
                && (longitude_end <= currLong + 0.001 && longitude_end >= currLong - 0.001))) {

                return p;
            } else if (p == cellLocations.length - 1) {
                return -1;
            }
        }
    }
    return p;
}

```

Figure 3-13. Location's array position finder

Initially, we make the current speed limit variable to be the general one, and then check if there is a specific one corresponding to the current location. If there is, the current speed limit variable is assigned a new variable from the speed limits array provided by the cloud database. Otherwise, it will be kept as it was. In either case, the current speed limit variable is sent by Bluetooth to the Raspberry Pi, the ECU. Now, let us dive into the implementation of the geolocation task and then Bluetooth.

The software implementation of both location techniques – Cellular and GPS – is the same in Android except that we have to specify the one we want to utilize as a parameter to a certain method. Firstly, before using the location data, one has to ask the permission of the user – this should be specified in the *Manifest.xml* file as well as in the Java program. The *LocationManager* class provides a method that request updates of the device's location called *requestLocationUpdates()*. This method takes as first parameter the location technique: *GPS_PROVIDER* or *NETWORK_PROVIDER*. These self-explanatory variables are provided by the location manager class. The second and the third parameters are integers specifying, successively, the minimum period that should be awaited and the minimum distance that should be traveled to request new location updates. The last parameter is a *location listener* object which listens whenever location is updated. In fact, we have to instantiate this location listener separately and override its main methods in order to make use of the location data. One of the methods that we must override is called *onLocationChanged()*. It provides the location object which we can utilize to get many details about the device location, but for the sake of our project, we will only use the values of the latitude and longitude. We can update a global variable with the new location data if we want it to be used by other methods.

In order to turn on the Bluetooth functionality, first of all, the App must also obtain a permission from the user. This is done by requesting permission programmatically, which opens a window dialog where the user is asked to accept or decline the request. Here, we will not make our App to search for or pair with Bluetooth devices. Rather, it should be done through the system's provided Bluetooth user interface. After everything being properly set up, the App can start sending data – speed limit values in this case – and receive acknowledgements from the Raspberry Pi. Let us see how this is done programmatically.

The Bluetooth address must be manually added and then stored until overridden. However, let us assume that we have it stored beforehand. The entry point to communicating with surrounding Bluetooth devices is the *BluetoothAdapter* class. This class provides a method to get a representation of a Bluetooth device by providing it its address as parameter. If the representation of the device is now acquired, we can make use of its methods. The one that we definitely need to use is the *createRfcomSocketToServiceRecord()*. This method, if passed a UUID value as parameter, returns a Bluetooth socket which can connect us to the Bluetooth device. Before connecting though, it is preferable to call the *BluetoothAdapter*'s *cancelDiscovery()* method, otherwise the connection will be sluggish. To connect, we only need to call the *connect()* method from the socket object and, voila, the devices are connected via Bluetooth and share data. To read data, we should read from the input stream of the socket. And, obviously, we should write to the output stream when we want to write data. To the *InputStream*'s *read()* method, we pass an array of bytes to be used as a buffer when data is received. We pass to the *write()* method of the *OutputStream* class the variable to send, which is the speed limit.

5. Vehicle Speed Record

In order to keep each vehicle's speed record for analytical purposes, the vehicle uploads instantly its speed data to remote databases. These remote databases store the data persistently and any authorized agents can access the data through a webpage or a smartphone application. Figure 3-14 illustrates the system.

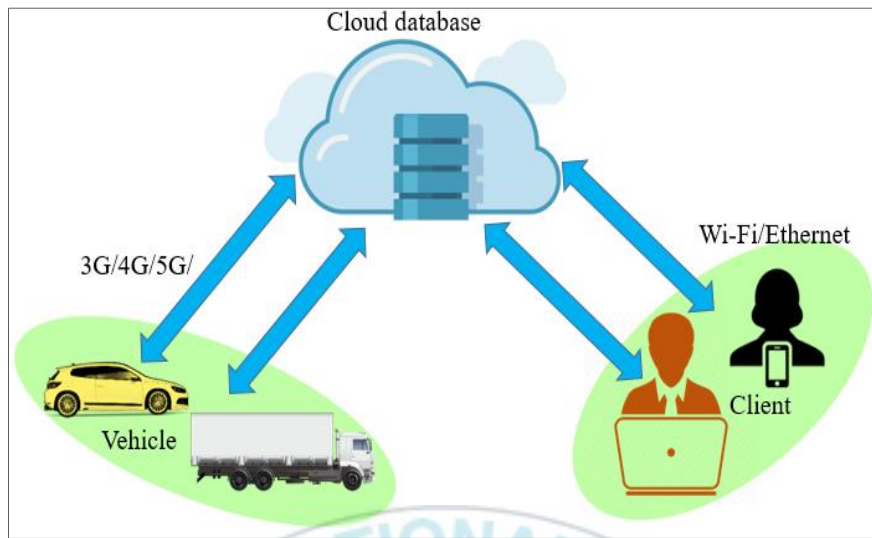


Figure 3-14. Traffic record system

In this context, the vehicles, as they are equipped with single board computers having cellular interface devices, upload their information through cellular networks to the remote servers automatically. The information will consist of the traveled speed and distance of each determined area.

This might help in finding patterns in road accident causes and, in turn, speed limits can be adjusted and even weather-related traffic accidents can be mitigated. For this purpose, machine learning algorithms can be exploited. Furthermore, the data can be used for penalty points by the law enforcement.

Conclusion

In this paper, firstly, we have shown how we built a framework for controlling and monitoring IoT systems that are based on the interaction of a user with a handheld device requesting services from a computer that interacts directly with a particular set of appliances. Additional features have been added, such as maximum value notification and other types of notifications. We have described how we designed and implemented the Android smartphone application and the functionalities it provides and how the computers control the appliances. We also described how CoAP is implemented in client-side and server-side. Furthermore, we have elaborated on the message formats that are used in order to indicate specifically when, which and how to handle a particular service. Additionally, we added algorithms to generate verification tags for messages sent through the Internet between the members of a system. We also implemented the *Firestore* Cloud for real-time data update in order for all users to get the latest information about each existing appliance in their particular IoT system.

Secondly, we proposed a system for mitigating road traffic accidents by controlling vehicle speed limits based on areas. To make it simpler, we divided areas based on network cell coverage; each cell with a general speed limit and eventual specific speed limits. We primarily proposed the cellular geolocation technique to be used in preference over GPS. However, the option of using GPS is explained and implemented. We explained in details how various existing road speed limiters work and we implemented a small scale prototype in order to show how the overall system might work. Finally, we proposed a system for the traffic data to be stored in order for it to be used by concerned agents to have more control over the traffic.

References

- [1] Ibrahima Wane, Minjeong Shin, Sungun Kim, Suk Jin Lee " Hardware and Software Framework for Controlling and Monitoring IoT Appliances", ICUFN 2019, [To be published]
- [2] Ibrahima Wane, Minjeong Shin, Sungun Kim, "Development of IoT System for Home Appliances", Proceedings, International Conferences NGCIT, vol. 152, pp.129-133, 2018
- [3] Ibrahima Wane, Minjeong Shin, Sungun Kim, Suk Jin Lee, "Development of Security Mechanisms for Industrial IoTs", Proceedings, JKCCS, pp. 99, 2019
- [4] Minjeong Shin, Jihyeon Woo, Ibrahima Wane, Sungun Kim, Heung-Sik Yu, "Implementation of Security Mechanisms in IIoT Systems", Lecture Note on Electrical Engineering, vol. 502, pp. 183-187, 2019
- [5] Android Developers, "Location Strategies", <https://developer.android.com/guide/topics/location/strategies>, [Accessed: Apr. 15, 2019]
- [6] IETF, "The Constrained Application Protocol (CoAP)", RFC 7252, 2014
- [7] Android Developers, "Documentation, Open Files Using Storage Access Framework", <https://developer.android.com/guide/topics/providers/document-provider>, [Accessed: Dec. 20, 2018]
- [8] Android Developers, "Documentation, Activity", <https://developer.android.com/reference/android/app/Activity>, [Accessed: Nov. 1, 2018]
- [9] Android Developers, "Documentation, Array Adapter", <https://developer.android.com/reference/android/widget/ArrayAdapter>, [Accessed: Jun. 6, 2018]

- [10] Android Developers, “Documentation, Text View”,
<https://developer.android.com/reference/android/widget/TextView>, [Accessed:
Dec. 24, 2018]
- [11] Raspberry Pi, “Downloads”, <https://www.raspberrypi.org/downloads>,
[Accessed: Oct. 1, 2018]
- [12] NodeJs, “About Node.js”, <https://nodejs.org/en/about>, [Accessed: Oct. 5,
2018]
- [13] Arduino, “Software”, <https://www.arduino.cc/en/Main/Software>, [Accessed:
Oct. 2, 2018]
- [14] Matteo Collina, et al. “Node-coap”, <https://github.com/mcollina/node-coap>,
[Accessed: Mar. 18, 2018]
- [15] Firebase, “Cloud Firestore”, <https://firebase.google.com/docs/firestore>,
[Accessed: Dec. 21, 2018]
- [16] Firebase, “Add Firebase to Your Android Project”,
<https://firebase.google.com/docs/android/setup>, [Accessed: Dec. 19, 2018]
- [17] IETF, “Diffie-Hellman Key Agreement Method”, RFC 2631, 1999
- [18] J. Park, “128-bit Block Cipher LEA”, TTA Journal, vol. 157, pp. 64-69, 2015
- [19] World Health Organisation, “Road Traffic Injuries Fact Sheet”,
<https://www.who.int/news-room/fact-sheets/detail/road-traffic-injuries>,
[Accessed: Mar. 20 2019]
- [20] Lydia Jackson, Richard Cracknell, “Road Accident Casualties in Britain and
the World”, House of Commons Library Briefing Paper, 2018
- [21] World Health Organisation, “Road Safety - Speed”, <https://www.who.int>,
[Accessed: Mar. 20, 2019]
- [22] Ahmed Farouk AbdelGawad, Talal Saleh Mandourah, “Proposed Simple
Electro-Mechanical Automotive Speed Control System”, American Journal of
Aerospace Engineering, vol. 2, pp. 1-10, 2015

- [23] Autograde, “Electronic Speed Governor”,
<http://www.autograde.in/electronic-speed-limiter-safedrive.php>, [Accessed:
Mar. 21, 2019]
- [24] Lei Wang, Maciej Zawodniok, “RSSI-based Localization in Cellular
Networks”, 6th IEEE Workshop on User Mobility and Vehicular Networks,
2012
- [25] Stephen Riter, Jan McCoy “Automatic Vehicle Location – an Overview”,
IEEE Transactions on Vehicular Technology, vol. 26, pp. 7-11, 1977



IoT 시스템 프레임워크 구현에 대한 연구

Ibrahima Wane (이브라힘)

부경대학교 대학원 정보통신공학과

요약

본 논문은 다양한 응용의 IoT (Internet of Things) 시스템 구현에 요구되는 프레임 개발에 대한 연구를 다룬다. 먼저 홈 IoT 시스템 프레임워크 구축을 위해 Android 모바일 어플리케이션을 구현한 후 요구하는 서비스에 대한 요청을 처리하기 위해 백-엔드 애플리케이션도 개발한다. 그리고 홈에서 사용되는 정보가전기기들을 임베디드 시스템에 여러 종류의 센서 및 인터페이스로 연결하고 효율적인 통신 프로토콜을 적용하여 통신을 한다. 또한 웹 서비스의 경우 HTTP (Hypertext Transfer Protocol) 프로토콜을 사용하는 대신 CoAP (Constrained Application Protocol) 프로토콜을 사용한다. CoAP 메시지의 정보 필드는 각각의 정보가전기기의 특성에 따라 분류하여 독창적인 메시지포맷을 정의 및 구현한다. 그리고 CoAP 메시지의 전송과정에서 LEA (Lightweight Encryption Algorithm) 알고리즘을 응용하여 빠르고 간단한 보안메커니즘을 개발 및 적용한다 또한 Firebase Cloud Firestore 데이터베이스를 활용하여 해당 홈 IoT 시스템 사용자들에게 각종 사용정보를 축적하여 활용하도록 구성한다.

연구의 두 번째 부분은, 다른 유형의 IoT 시스템 프레임워크 구축으로 도로 위에서 달리는 차량이 자동적으로 해당지역 기반 제한속도 지켜 과속으로 인한 사고를 줄이는데 기여한다. 즉 셀룰러 네트워크 기술을 활용해 통신회사가 운영하는 셀 정보를 사용하여 각 차량은 해당지역의 제한속도를 파악해 이것을 자동적으로 지키는 IoT 제어 시스템이다. 먼저 GPS (Global Positioning System) 및 셀룰러 위치기반기술을 사용하여 각 차량의 위치를 파악 후 특정 셀 내의 해당영역에서 제한속도를 파악하고 차량속도를 제어하기 위한 ECU (Electronic Control Unit)를 구현한다. 그리고 차량의 기어박스에 연결된 속도센서가 차량속도 값을 전달 후 해당지역의 제한속도에 맞게 제어하는 시스템을 구현한다. 마지막으로, 제안된 두 IoT 시스템들이 모두 실제 상황에서 동작될 수 있도록 프로토타입 구현하여 성능을 확인한다.

Acknowledgments

I am grateful to the Almighty Creator who has given me strength, health and wisdom to complete my Masters and endure every hardship to become a better human being.

I would like to express my utmost gratitude to my supervisor, Sung Un Kim, who sacrificed a lot of his time and effort for the successfulness of this thesis and the accomplishment of my master. I also thank all the professors of the department of Information and Communications Engineering who have taught me and gave me courage and passion to learn more and more.

I express my gratitude to my parents, Aichetou Sow and Chouaibou Wane, who gave me life, love and good upbringing to pursue my dreams and education in spite of the hardship of the environment we live in. I would like to thank each of my family member, from Hindou to Oumar, and relatives, and friends for their support and encouragements.

To all my lab mates, Minjeong Shin, Jihyeon Woo and Sang Hee Lee, I would like to thank you for the great time that we spent together. I thank especially all my Mauritanian friends, namely, Abou Tall, Abdoulaye Sall, Elimane Ba, Samba Ba, Sidi Yahya, Mohamed Sow, Ahmed Ba, Abdoulaye Ball and Hammoud Yahya, for helping me throughout this entire process of studying and living in Korea.

Finally, I thank the NIIED for making this process as easy as possible by providing me with a scholarship and lot of other things that lead to the successfulness of this master.