Hardware and Software Framework for Controlling and Monitoring IoT Appliances

Ibrahima Wane
Information and Communications Engineering
Pukyong National University (PKNU)
Busan, South Korea
mamawane94@hotmail.fr

Minjeong Shin
Information and Communications Engineering
Pukyong National University (PKNU)
Busan, South Korea
minjung3373@pukyong.ac.kr

Sungun Kim
Information and Communications Engineering
Pukyong National University (PKNU)
Busan, South Korea
kimsu@pknu.ac.kr

Suk Jin Lee
TSYS School of Computer Science
Columbus State University
GA, USA
lee_suk@columbusstate.edu

***Abstract*— The purpose of this research is to build a complete and easy-to-use framework for setting controllable IoT systems, such as Home IoT systems. We created a dynamic 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 making requests in certain circumstances. We used multiple sensors and devices to represent real-life appliances. It is aimed that any nonprofessional with some references can easily acquire the necessary information for setting up his own IoT system. In terms of application layer protocols, we used Constrained Application Protocol (CoAP), which is a web transfer protocol especially designed for low power and memory limited devices and for lossy networks [1], rather than using the traditional HTTP as Internet application protocol. Based on CoAP, we built our own message formats for better parsing of the requested services.**

Keywords*— *CoAP, IoT, Home IoT, Raspberry Pi, Arduino, Node.js.

I. INTRODUCTION

Internet of Things (IoT), in a nutshell, 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. 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. [2][3]. Home IoT exemplifies one of IoT application areas, which enhances Home security, plays a big role in reducing energy wastage and gives the homeowner a better control over his home[4]. 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. Furthermore, installed air conditioners, sprinklers, lightbulbs and heaters can be remotely actuated [5].

Most of the related projects focus on implementing a single IoT application area, such as Bathroom or Farm IoT systems. Usually, the mobile application is hardcoded and therefore works specifically for a particular system such as [6] and [7]. Also, the controller is programmed to interact only with a specific type of sensors and appliances [5].

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 request message remotely sent 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.

In this paper, we mainly focus on building a complete and easy-to-use framework for controlling and monitoring IoT systems. Therefore, the contribution of this paper is twofold. First of all, we have built the service messages' format that is used for handling a particular request in the CoAP message payload – a single-board computer parses the request messages and apply them to different appliances and to sensors using different communication protocols. Secondly, we have developed real-life customizable applications of handheld devices and single-board computers to control/monitor remote IoT appliances.

The remaining of the article discusses about the system setting and the message format in Section 2. Afterwards, a detailed explanation of the mobile application implementation and structure is given in Section 3. Then, in Section 4, the implementation in the controller side is discussed in detail. Finally, we conclude this paper in Section 5.

II. System Configuration and Pin-Based IoT Message

A. System Configuration

For the basic setting of IoT system, we use a single-board computer, Raspberry Pi, as the central computer and Android-based smartphone as the user interface device. We also connect an extra Raspberry Pi controller to the central one through Wi-Fi and an Arduino microcontroller to one of the single-board computers via Bluetooth. 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 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. Certain appliances, such as lightbulbs, only use 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. 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 distinct interfaces for sensing the environment around it and must provide 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 have different types of structure. For now, the only supported protocols in this project are UART, SPI, 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 told. Note, if an appliance is classified into two categories simultaneously, i.e. contains a sensor and is controllable, each part must be considered as a different 'appliance'.

Figure 1 depicts an IoT system scenario with a single user and different controllers and appliances.

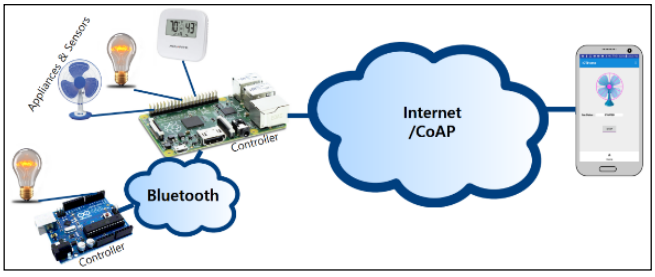


Fig. 1. IoT System Scenario

The smartphone sends a request to the Raspberry Pi controller through 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.

B. Pin-Based IoT Message

In the CoAP message payload field, we built a message format composed of fixed and optional fields. As shown in Figure 2, the light grey fields are required in any case of controlling or monitoring appliance, 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.

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

Fig. 2. Pin-based IoT Message Format

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.

III. Mobile Application Design

We use the 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 is aimed at 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 [8]. We use mostly the client features it provides and in some cases the server functionalities.

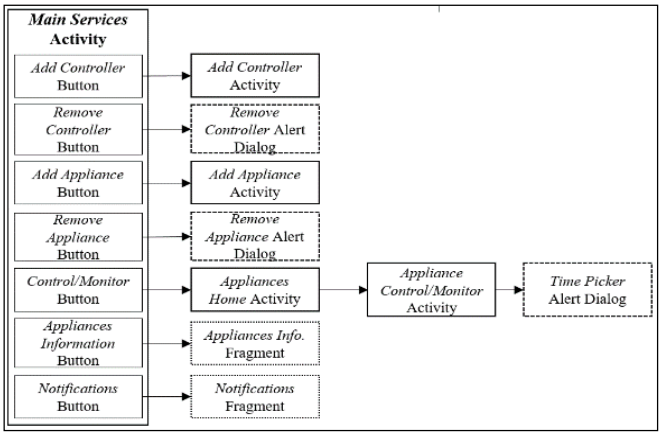


Fig. 3. Activities Structure

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. 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 3 shows how the services are organized. We will detail subsequently the purpose of each operation.

A. Add/Remove Controllers

A 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. Figure 4 shows the user interface of the controller adding process.

Fig. 4. Controller Add User Interface

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

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 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.

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. Figure 5 describes the adding process (a) and the removing process (b) of an element.

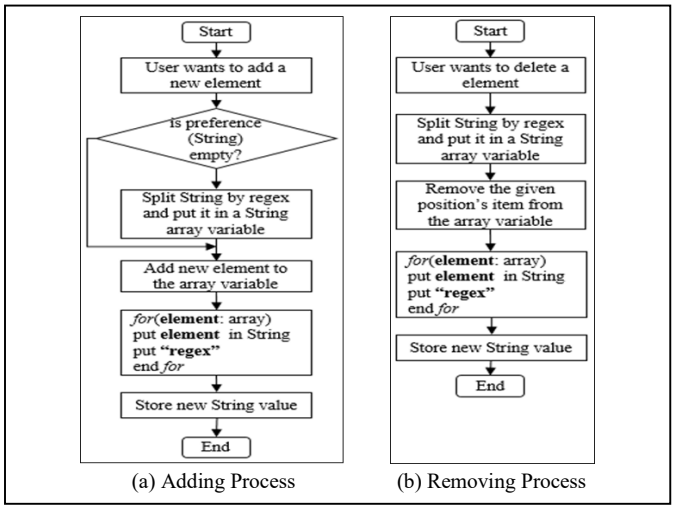


Fig. 5. Add/Remove Element

B. Add/Remove Appliances

The user provides the necessary information to create an appliance's user interface. 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 6 shows the UI for creating an appliance interface. 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.

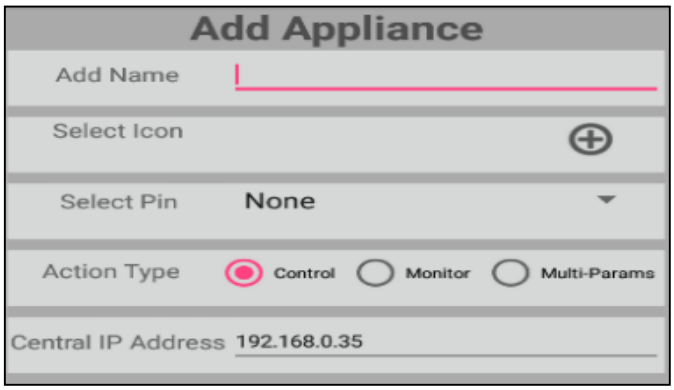
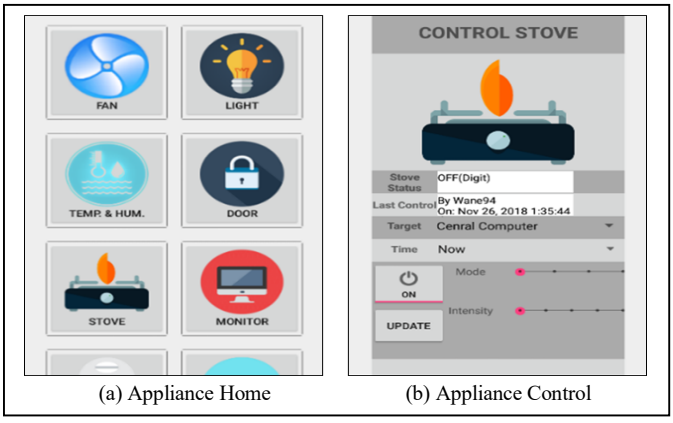


Fig. 6. Add Appliance User Interface

C. 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, and telling the computer to perform certain tasks based on the information in the given message.

First of all, let us describe what are the functions of the components in Figure 7(b), in order to gain an insight into what information the message must contain. Actually, it 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, as shown in Figure 7(b). After user clicks the trigger button, the central computer receives a request message which takes all the parameters at play into consideration.



(a) Appliance Home (b) Appliance Control

Fig. 7. Control/Monitor User Interfaces

The 'appliances home' user interface, depicted as Figure 7(a), 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.

Actually, some of the 'appliance control/monitor' user interface child views are enumerated previously. Let us describe their background work. The image Uri is retrieved from the stored 'appliance image' data. 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 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. A click on the first spinner item appends the associated type and address of the destination controller to the IoT message. As for the 'time spinner', two clickable *TextViews*, "now" and "later" are 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 a time picker window in which the weekdays and a clock are shown. Then after setting up the time and confirming, the time variables will be inserted somewhere in the message that will be sent for appliance controlling service.

D. 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 is 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 logged into a list of notifications. Figure 8 shows the flow of the notification process.

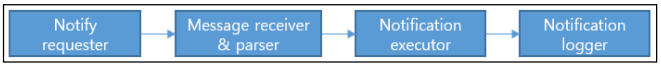


Fig. 8. Notification Process

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 shown in the user interface, such as "The fixed threshold has been exceeded!". 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 user interface purposes. 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. To log each notification, we use a list view in which the data is passed to a group of views contained in a list item, as Figure 9 indicates.

Fig. 9. Notifications List User Interface

IV. Controller Design

The central computer, a Raspberry Pi single-board computer, runs Raspbian as an official operating system [9]. 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 [10]. 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.

Each controller has I/O pins through which it interacts with the appliances or with interfacing modules, e.g. a Bluetooth chip.

A. CoAP Server

There exist various CoAP libraries available. We use one of them, called node-coap, among various programming languages' libraries. Node-coap is a JavaScript/node library that provides various APIs for implementing the defined CoAP functionalities. 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 service.

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 server functionality.

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. Figure 10 represents the actual code described previously.

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 UDP socket handling requests. The only difference is that node-coap has added too much code to perform all the CoAP requirements; hence, it provided APIs that are layers above the datagram module in order to simplify the code.

```
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);
```

Fig. 10. Node.js UDP datagram Implementation

The node-coap server implementation can be, basically, mapped to the code shown in Figure 10. The coap module provides a *coap.createServer()* API, which create 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 task's 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 UDP 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)]*.

B. 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. The following paragraph describes how to implement this process in a programmatic manner.

Figure 11 simply represents the JavaScript code to run reservation process. Firstly, the code 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 (as described in section 3.C) 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.

```
container0 = {Timeout: -1, Request: 0}
container1 = {Timeout: -1, Request: 0}
…
container10 = {Timeout: -1, Request: 0}

Retrieve days values from request object
days = [sun, mon, tue, wed, thu, fri, sat]
for(dayIndex = 0; dayIndex<7; dayIndex++)
    if(days[dayIndex] == 1)
        compute timeout
        switch(-1)
            case container0.Timeout:
                    container0.Timeout=timeout
                    container0.Request = request
                    perform the service of container0.Request after container0.Timeout
                    container0.Timeout = -1
                    container0.Request = 0
            break
            end case
            case container1.Timeout:
            ….
```

Fig. 11. Reservation Pseudocode

V. Conclusion

In this paper, we have shown how we built a framework for controlling and monitoring IoT systems requiring the interaction of the user with a device, and in turn, it requests a service from a computer that interacts directly with the appliance. Additional features have been added, such as maximum value notification and other types of notifications. We have described both parties; how we designed 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 control and monitor services' message format that is used in order to indicate specifically when, which and how to handle a particular request. Finally, we proposed an algorithm for performing multiple service reservations without the previous ones being overridden by the newer ones.

In our future work, we have a plan to add other features such as adding an instant video streaming window; for appliances such as surveillance cameras. Although, most of this application's features can be used in real life, sometimes there might be some difficulties for a layman to implement them, we therefore plan to improve the application for it to be more user-friendly.

References

[1] IETF, "The constrained application protocol (CoAP)", RFC 7252, 2014.

[2] Charith Perera, Chi Harold Liu, Srimal Jayawardena, Min Chen, "A Survey on Internet of Things From Industrial Market Perspective," IEEE Access, vol. 2, pp. 1660-1679, 2014.

[3] Li Da Xu, Wu He, Shancang Li, "Internet of things in industries: a survey," IEEE Transactions on Industrial Informatics," vol. 10, no. 4, pp. 2233-2243, 2014.

[4] Raj G Anvekar, Rajeshwari M Banakar, "IoT application development: home security system", IEEE International Conference on Technological Innovations in ICT For Agriculture and Rural Development, pp. 68-72, 2017.

[5] S. R. Prathibha, Anupama Hongal, M. P. Jyothi, "IoT based monitoring system in smart agriculture", International Conference on Recent Advances in Electronics and Communication Technology, pp. 81-84, 2017.

[6] Minwoo Ryu, Jaeseok Yun, Ting Miao, Il-Yeup Ahn, Sung-Chan Choi, Jaeho Kim, "Design and implementation of a connected farm for smart farming system", in Proc. 2015 IEEE Sensors., pp.1724-1727.

[7] Tanish Sehgal, Shubham More, "Home automation using IoT and mobile app", International Research Journal of Engineering and Technology, pp. 694-698, 2017.

[8] Eclipse Foundation, "Eclipse californium (Cf) CoAP Framework", https://projects.eclipse.org/projects/technology.californium, [Accessed: Nov. 1, 2018].

[9] Raspberry Pi, "Downloads" https://www.raspberrypi.org/downloads/, [Accessed: Oct. 1, 2018].

[10] NodeJs, "About Node.js" https://nodejs.org/en/about/, [Accessed: Oct. 5, 2018.