A close up of a logo

Description automatically generated

**Mobile Computing WS-19/20– IoT Smart Home**

**Fire Alarm And Prevention**

**Name : Devan Premchandrakumar Nair**

**Matriculation Number : 1274369**

**Guidance : Mr. Gregor Frick**

**Mr. Besfort Shala**

**Prof. Armin Lehmann**

**Table of Contents**

[Case Description 3](#_Toc29433989)

[Network Architecture 4](#_Toc29433990)

[**Sensor-Actuator Emulator** 4](#_Toc29433991)

[**IoT Gateway** 4](#_Toc29433992)

[Protocol Descriptions 5](#_Toc29433993)

[**CoAP** 5](#_Toc29433994)

[**HTTP 1.1** 7](#_Toc29433995)

[Implementation Strategy 8](#_Toc29433996)

[**IoT Gateway Architecture** 8](#_Toc29433997)

[**Resource Discovery** 8](#_Toc29433998)

[**State Generation** 10](#_Toc29433999)

[**Sensor Actuator Emulator Environment** 13](#_Toc29434000)

[Project Setup Instructions 17](#_Toc29434001)

[**Ubuntu** 17](#_Toc29434002)

[**Windows** 17](#_Toc29434003)

[Implementation Analysis 18](#_Toc29434004)

[CoAP URI Summary 18](#_Toc29434005)

[HTTP URI Summary 19](#_Toc29434006)

[Appendix A 19](#_Toc29434007)

[Acknowledgements 19](#_Toc29434008)

[Reference 19](#_Toc29434009)

# **Case Description**

For the given case of smart fire alarm – prevention and detection, we consider a scenario of a home with multiple rooms. In our case, for the simulations we have chosen three rooms. The rooms are labelled as ‘Room 1’,’Room 2’and ‘Room 3’. Each of these rooms are equipped with the same nature of sensors and actuators. The sensors in these rooms are designed to emulate, measurement of a set of environment parameters namely – temperature (in deg Celsius),flash value (IR-spectrum intensity sensors) , smoke sensor (a finite state machine with 2 states – ‘minimal’ and ‘intense’). The actuators in the room are used to emulate a fire alarm(notification about each room), a smart-extinguisher actuator (which activates extinguisher for a specific volume rate and specific time). The activation of extinguisher actuator is expected to reduce the effects fire has on sensor parameters. So the sensors should start reading reduced values once the actuator starts working.

This use case proposes to emulate a scenario where there is fire in one of the rooms ,the gateway detects a fire through its measurement value retrieved from the sensors. The gateway then triggers the appropriate alarm related to the room which has the fire, thereby enabling client to know which room the fire has taken place and will also trigger an actuator that will extinguish the fire. The gateway will prompt the user for confirmation for actuation in the room once it recognizes that sensor measurement values for environment parameters exceeds the threshold values. This is to simulate a case of user-confirmation to make sure if there was an actual fire and it is not because of malfunctioning(hypothetical) of one of the sensors. After the user has prompted affirmation to extinguish the fire one can observe the temperature reducing in the respective room and the smoke reducing thereby emulating actual extinguishing of a fire before it progresses to a catastrophic end. It also enables a set of guidance actuators which will guide possibly trapped occupants of the room to emergency exits or some other mechanism which achieves a similar goal. It will trigger an alarm indicating the location of the room with the fire. The user may also deny permission for fire extinguisher which will prevent extinguisher from operation ,if the client-user confirmed in person that there was no fire and that excess measurement reading of environment parameters were a result of faulty sensors – a highly likely scenario since these sensors are battery-operated autonomous Internet connected devices . The IoT-Gateway has a web console which can be operated in an Internet browser to view the sensor measurements and actuator states in real-time and to respond to gateway alerts and prompts.

This document is expected to serve as a manual for operation and provides architectural description of the project. The description for the functionality and attributes are expressed in a programming language abstract fashion.

# **Network Architecture**

HTTP

Web Browser

IoT Gateway

Sensor-Actuator Emulator

CoAP

*{xml-payload}*

*{json-payload}*

localhost:8080

localhost:62235

localhost:5685,..,5699

Figure 1: Network Diagram

The project has two major components – Emulator for sensor-actuator environment and IoT gateway.

## **Sensor-Actuator Emulator**

The emulator has all the sensors and actuators related to the use-case residing in localhost. The different sensors and actuators have distinct L4 addresses (the port number range [5685-to-5699] and description for those are given in Appendix. A). The emulator nodes all use *coap* over UDP scheme. Each sensor and actuator runs a CoAP resource on a CoAP server(Datagram Server) on a single port performing its respective function(reading temperature, luminosity etc. ). The payload that is sent or received by the hosts in the emulator is of JSON content format.

## **IoT Gateway**

The IoT gateway has two interfaces . As shown in the figure the interface communicating via CoAP over UDP, resides on localhost:62235. It communicates to the sensors and actuators residing in the emulator environment. It hosts multiple CoAP client services and subscription listeners on a single endpoint at localhost:62235.It communicates with the emulator environment with JSON payload content format. The gateway also has a web-console which it hosts at the address localhost:8080. The web-console resides under the URL- *http://localhost:8080/console*. The gateway communicates with the browser over this interface via HTTP/1.1 ReSTful services (the descriptions of these HTTP service endpoints are provided [in](#_HTTP_URI_summary) HTTP URI summary section) . The services produce and consume payload of XML content format.

# **Protocol Descriptions**

## **CoAP**

A screenshot of a cell phone

Description automatically generated

Figure 2:CoAP over UDP stack

The sensors and actuators in the emulation environment communicate to the gateway via CoAP over UDP on their respective L4 port addresses. The CoAP interface of the gateway and the sensors and actuators in the emulator environment together constitute a Constrained ReSTful Environment (*abv .* CoRE) [1] [2]. The CoAP interface implements a resource discovery mechanism when the project starts up . The gateway sends a GET request to well known URI of each CoAP sensor or actuator - */.well-known/core* [1]. The underlying CoAP server operating in the elements of the emulator will respond with a list of available resources on the server along with the attributes available for them namely – content type: JSON, resource type: e.g. temperature , CoAP observability ,interface type: sensor, actuator etc. The resources and their attributes are returned via *Web-Link* [1] format as per RFC 6690 and RFC 7252 . The subsequent data are requested by the gateway via confirmable GET,POST methods. The nodes in the emulator environment respond to the methods with the appropriate *ACK* message containing the payload in JSON content format. Some of the nodes in the emulator environment operate via CoAP Observe mechanism [3]. The gateway initializes a Observe REGISTER to these nodes, after that these nodes send payload in their *ACK* responses whenever these nodes detect a change in representation of their resource.

Message sequence charts for different operations of gateway are shown :

CON : GET /temp

MID : a Token : x

2.05 ACK *{json-payload}*

MID : a Token : x

CON : GET /temp

MID : a + 1 Token : y

2.05 ACK *{json-payload}*

MID : a + 1 Token : x

GW

Sensor

CON : POST /foam *{json-payload}*

MID : a Token : x

2.04 ACK *{json-response-message}*

MID : a Token : x

CON : POST /foam *{json-payload}*

MID : a + 1 Token : y

2.04 ACK *{json-response-message}*

MID : a + 1 Token : x

GW

Actuator

Figure 4:CoAP GET for resource at /temp and POST for resource at /foam

OBSERVE seq : 0 GET /smoke REGISTER

2.05 ACK OBSERVE seq:1 *{json-payload}*

CON 2.05 OBSERVE seq : 2 *{json-payload}*

Token : x MID : b

Token : x MID : a

Token :x MID : a

ACK *Empty Message*

MID : b

CON 2.05 OBSERVE seq : 3 *{json-payload}*

Token : x MID : c

ACK *Empty Message*

MID : c

GW

Sensor

\*value change

\* value change

Figure 3:Coap Observe Mechanism for resource at /smoke

Figure 4:CoAP Observe Mechanism for resource at /smoke

## **HTTP 1.1**

## 

A screenshot of a cell phone

Description automatically generated

Figure 5:HTTP over TCP Stack

The Web browser communicates with the gateway via HTTP ReST messages. The application utilizes HTTP GET and POST methods in this case. All of the HTTP resources of the gateway sends and receives payload in XML content format.

An example message sequence chart of its operation is given below

HTTP : GET /data/globalStates

200 *{xml-payload}*

HTTP:POST data/isAct

200 *{No payload}*

Browser

GW

Figure 4:HTTP GET and POST method description

# **Implementation Strategy**

## **IoT Gateway Architecture**

The gateway architecture has two main functionalities with regards to its interactions with nodes in the emulator environment – Resource Discovery and State Generation. With regards to its interactions with HTTP client(*Web browser*), it parses the generated state and sends it to browser for every HTTP request it receives at its HTTP server socket(*localhost:8080*) as per the description of the designated URI.

### **Resource Discovery**

2.05 ACK *{Weblink payload }* /foamExt;if=Actuator …

2.05 ACK *{Weblink payload }* /smoke; if=Sensor ;obs…

GET /.well-known/core?if=\*

GET /.well-known/core?if=\*

2.05 ACK *{Weblink payload }* /temp;if=Sensor …

GET /.well-known/core?if=\*

Actuator

Obs.S

Sensor

GW

Figure 5: Resource Discovery Procedure Note: Obs.S : CoAP Observer enabled Sensor

The gateway translates CoAP protocol to HTTP/1.1 and vice-versa and also makes decisions based on the use-case. When the application starts, the gateway does not know the URI’s corresponding to resources where data of various sensors and actuators in the emulator environment resides. The gateway only knows the scheme(*coap*) and authority(*localhost:{port number}*) of the various nodes in the emulator environment. So it performs a resource discovery mechanism in this CoRE [1]. As shown in the simplified figure above, the gateway sends a GET request to */.well-known/core* to all the nodes in the emulator environment. Each node responds to this GET request with a WebLink list of all the available resources residing on that CoAP Server and the associated attributes to that resource. The gateway then uses the associated attributes to recognize the different sensors, actuators, finite-state-machines(sensors with CoAP Observe enabled [3] ). The gateway then constructs the complete URI under which each of the resources reside*(scheme + authority + path) .*The gateway then interacts with each of these nodes in the respective fashion acknowledging the attributes corresponding to these resources. Resource discovery is only performed once during startup of this application.

### **State Generation**

Compute States

Browser

GW

Sensor

Actuator

Obs.Sen

GET /temp

GET /smoke Observe REGISTER

2.05 *{json-payload}*

2.05 *{json-payload}*

GET /data/globalStates

200 *{xml-payload}*

POST *{xml-permission}*

Prompt User Permission

POST /foamExt *{json-actuation command}*

Repeat Process

CON *{json-payload}*

2.05 ACK

\*Value change

Figure 6:Simplified Gateway Operation Note: Obs.Sen : CoAP Observer enabled Sensor

The gateway autonomously generates a state vector corresponding to the different sensor measurement values and actuator states corresponding to each room at regular time interval(~*500ms* in application).The computed state vector has the fields – temperature, flash value, smoke state, actuator state. All these fields combined with the name of the room where this sensor resides is used to compute the *state* of the room. Based on comparison of these measured values with the threshold values pre-configured are used to determine whether the room has any fire in it. Based on the values of smoke sensors ,it also has warning for excessive smoke scenarios for each room. The above simplistic message sequence chart characterizes the messages exchanged in this process. Note however that it is simplified version and does not show all the sensors and actuators in the emulator environment , but for a simplified case of one sensor, one actuator and one finite-state-machine(Observable smoke sensor) which are part of one room(Note ,a room has another luminosity/flash sensor and an alarm/guidance assistance actuator). All the sensor measurements , actuator states and smoke warning and fire presence attributes together for one room constitute a *local state*. The compilation of local state for the other rooms together constitute a *global state*. The global state and the local states are available as representations of ReSTful resources as per summary provided under [HTTP URI Summary](#_HTTP_URI_summary) section.

A screenshot of a cell phone

Description automatically generated

Figure 7:UML diagram with dependencies and public methods of Gateway protocol translation and decision-making functionalities

A screenshot of a cell phone

Description automatically generated

Figure 8:UML class diagram with public methods of HTTP ReST endpoint implementation of gateway.

## **Sensor Actuator Emulator Environment**

**T**

**L**

**S**

**E**

**A**

**Room 1**

**T**

**L**

**S**

**E**

**A**

**Room 2**

**T**

**L**

**S**

**E**

**A**

**Room 3**

Temperature Sensor

IR Sensor

Smoke Sensor

Fire extinguisher actuator

Fire Alarm Actuator

Figure 9:Emulator Environment use-case scenario

The sensor-actuator emulator contains the emulated versions of a temperature sensor, smoke sensor , IR sensor , fire extinguisher actuator and a fire alarm actuator. Each of these sensors reads value corresponding to the parameter it is measuring from an environment file. An environment file exists for each of the rooms and has values of those physical parameters to measured represented by comma-separated values with each line denoting a particular instance of time. So as time moves forward ,the sensor will read next line in the environment file. The first value in the file represents the temperature value, the next value represents IR-intensity or luminosity indicator values, the final value represents the smoke state (which can take values only 0,1 or 2). Similarly an actuator writes values to the end of this file as per the actuation command given to it by the gateway and thereby we model a reduction of physical quantities.

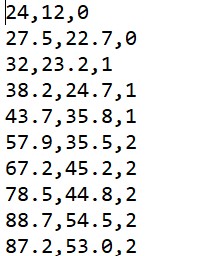


Figure 10:Environment file snippet

The smoke sensor is modelled as a finite state machine i.e. the gateway can register itself for a CoAP Observe [3] operation to this sensor and the sensor starts reading the physical quantity from the file but it sends the value to the gateway only when the value changes from one value to the next. So in the case of above environment file, the smoke values are sent to the gateway only when it changes from 0 to 1and from 1 to 2. The smoke sensor reads a line of the file after a fixed duration which corresponds to periodic measurements. The reason for modelling a smoke sensor this way is because an IoT enabled smoke sensor under real circumstances is operating likely on battery or some other auxiliary source of power. Responding to periodic requests from the gateway would mean spending energy over transmitting the same representation of the resource. Even practically a smoke sensor has very few set of operating states corresponding to either detection of smoke or absence of smoke, so polling the smoke sensor periodically is not as energy efficient as one might expect an IoT application to behave as. Each sensor is modelled as a resource residing on a CoAP server. Whenever this resource is invoked via any of the CoAP methods, the respective handler for it is called. In case of temperature or IR sensor the gateway periodically sends a GET request, so when the sensor receives a request it invokes its respective method handler to either read value from the environment file (in case of sensor) or write into an environment file(in case of an actuator receiving a POST message). For a smoke sensor the resource periodically reads the environment file and whenever it records a change in the state it notifies the subscribed client with the new representation for the resource.

The respective UML class relations and dependencies for the emulator environment is shown

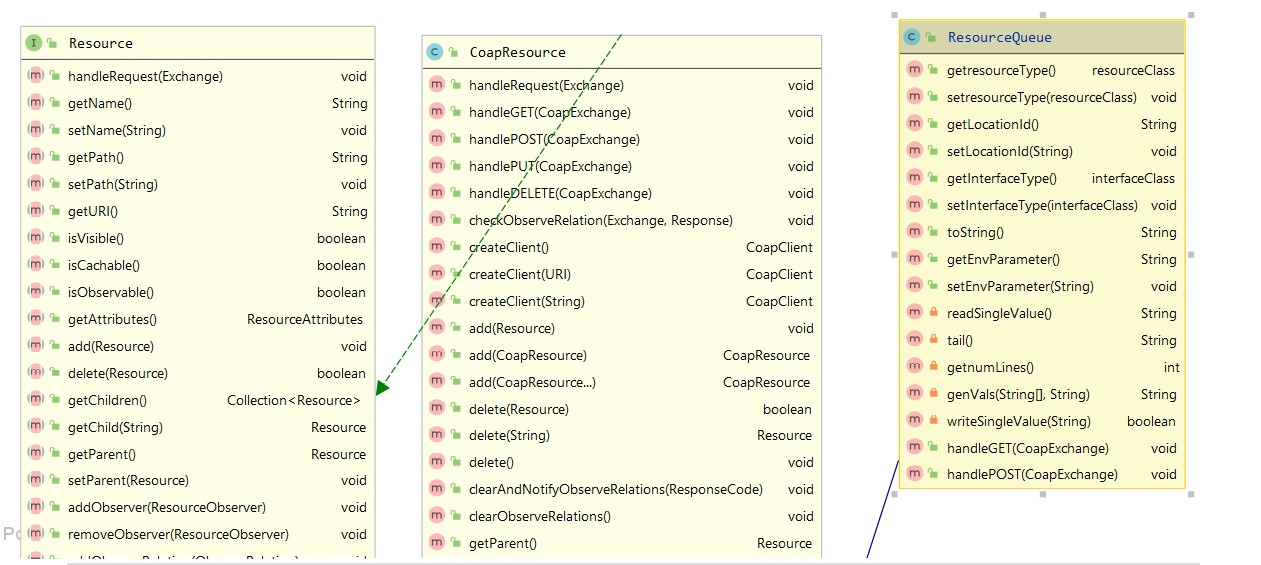
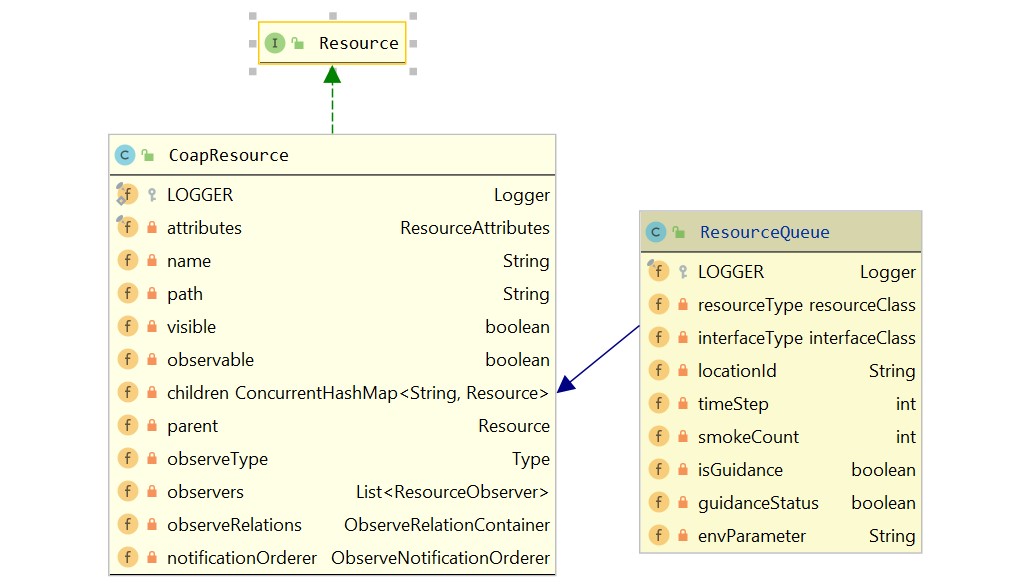


Figure 11:UML class diagram for extended CoAP resource used in sensor and actuators

A screenshot of a cell phone

Description automatically generated

Figure 12:UML class diagram for CoAP Observable resource used in smoke sensor

# **Project Setup Instructions**

## **Ubuntu**

1. Move the jar *iotservice-1.0.0.jar* to the folder where you wish to run the application.
2. The application will create a *Californium.properties* file (properties file containing CoAP stack configuration values) and *EnvironmentData* folder, so make sure the required permissions are given to the application to enable creation of necessary setup files and folders.
3. Open bash shell.
4. Make sure that none of the ports used by the project are used by some other application running on the machine. Execute *lsof -i -P* and check with the list of ports used by this smart home application mentioned in [Appendix A](#_Appendix_A). If there are any application running on the ports used by this project , close that application by executing *kill -9 {pid-of-the application}*.
5. Navigate to the folder where you have moved the application jar file.
6. Execute *java -jar iotservice-1.0.0.jar*.
7. After you see the application has successfully started running ,open a new browser tab and navigate to the web dashboard of the smart-home use-case at [*http://localhost:8080/console*](http://localhost:8080/console).
8. Respond to the requested prompts to observe the use case operation.

## **Windows**

1. Move the jar *iotservice-1.0.0* to the folder where you wish to run the application.
2. Open a command shell prompt and navigate to the folder where you have kept the jar file.
3. Execute *java -jar iotservice-1.0.0.jar*.
4. After you see the application has successfully started running ,open a new browser tab and navigate to the web dashboard of the smart-home use-case at [*http://localhost:8080/console*](http://localhost:8080/console).
5. Respond to the requested prompts to observe the use case operation.

# **Implementation Analysis**

# **CoAP URI Summary**

|  |  |  |
| --- | --- | --- |
| CoAP URL | Description | Resource attributes  (if Type;IsObservable?) |
| coap://localhost:5685/temp | Temperature Sensor | Sensor; No |
| coap://localhost:5686/irSense | IR Sensor | Sensor; No |
| coap://localhost:5687/smoke | Smoke Sensor | Sensor; Yes |
| coap://localhost:5688/foamExt | Fire Extinguisher actuator | Actuator; No |
| coap://localhost:5689/alarm | Alarm Actuator | GuidanceActuator; No |
| coap://localhost:5690/temp | Temperature Sensor | Sensor; No |
| coap://localhost:5691/irSense | IR Sensor | Sensor; No |
| coap://localhost:5692/smoke | Smoke Sensor | Sensor; Yes |
| coap://localhost:5693/foamExt | Fire Extinguisher actuator | Actuator; No |
| coap://localhost:5694/alarm | Alarm Actuator | GuidanceActuator; No |
| coap://localhost:5695/temp | Temperature Sensor | Sensor; No |
| coap://localhost:5696/irSense | IR Sensor | Sensor; No |
| coap://localhost:5697/smoke | Smoke Sensor | Sensor; Yes |
| coap://localhost:5698/foamExt | Fire Extinguisher actuator | Actuator; No |
| coap://localhost:5699/alarm | Alarm Actuator | GuidanceActuator; No |
| coap://localhost:62235 | IoT Gateway endpoint | - |

# **HTTP URI Summary**

|  |  |  |
| --- | --- | --- |
| URL | Description | Method |
| http://localhost:8080/console | Displays the main console dashboard of the application | GET |
| http://localhost:8080/data/globalStates | Returns the state of all sensors and actuators and rooms used in the application | GET |
| http://localhost:8080/data/Room%201 | Returns all the sensor measurement and actuator state values for Room 1 | GET |
| http://localhost:8080/data/Room%202 | Returns all the sensor measurement and actuator state values for Room 2 | GET |
| http://localhost:8080/data/Room%203 | Returns all the sensor measurement and actuator state values for Room 3 | GET |
| http://localhost:8080/data/locn | Returns list of names of rooms in the emulator environment. | GET |
| http://localhost:8080/data/isAct | Sends the user prompt response to enable or disable actuation as  <isAct><actEnable*>{user response}*</actEnable></isAct> | POST |
| http://localhost:8080/about | Displays the name and matriculation details | GET |

# **Appendix A**

# **Acknowledgements**

# **References**

|  |  |
| --- | --- |
| [1] | Z. Shelby, "Constrained RESTful Environments (CoRE) Link Format". |
| [2] | Z. Shelby, K. Hartke and C. Bormann, "The Constrained Application Protocol (CoAP)". |
| [3] | K. Hartke, "Observing Resources in the Constrained Application Protocol (CoAP)". |
| [4] | R. Fielding, J. Mogul, H. Frystyk, L. Masinter, P. Leach and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1". |