report

by Abin Mathew Abraham

FILE	7_FINAL_REPORT_CONTENT_-_GOOGLE_DOCS.PDF (1.64M)		
TIME SUBMITTED	20-JUN-2017 06:07PM	WORD COUNT	9313
SUBMISSION ID	826347901	CHARACTER COUNT	45237

CHAPTER 1
INTRODUCTION

1.1 Introduction

Internet of Things (IoT) is about enabling connections between objects or things. It is about connecting anything, anytime, anywhere using some service over any network. IoT refers to the use of sensors and devices which can communicate over wired or wireless networks, and the value creation opportunities it presents in terms of new applications and services. The interconnection of smart devices, sensors, and actuators exposing data services presents opportunities for novel applications and new ways of thinking about links between virtual and physical world.

The primary goal of connecting all things is to create better context awareness and this will enable applications, machines and humans to take intelligent decisions and respond to their dynamic context. Providing all these services requires interactions between smart devices, network elements, cloud services and end users. Most of the architectures till date achieve this using statically defined services. These architectures address the issues of scaling to an extent. But the protocols these architectures use for device management are not common agreed upon standard for device management. Moreover they do not handle all types of devices. So a new architecture needs to be developed which uses protocols which can be effective even in constrained environment and should be able to manage all types of devices which come up in future. The protocols used thus should be widely accepted by the majority of the manufacturers. Moreover, IoT devices and their context are extremely dynamic. The unpredictability of the real world must be taken into account. Homogeneity cannot be expected in systems. So there is a need for an intelligent system which is Autonomous and Adaptive. In this context, Autonomous means that the system should figure out how to achieve a set of given goals and Adaptive means that the system should be able to take decisions based on the environment.

1.2 Area of Computer Science

1.2.1 Network Protocols

As IoT systems involve a lot of communication between the devices, study on the existing protocols is necessary. As the system is heterogeneous, same protocol stack cannot be expected on all devices. The project involves a lot of device management and translations between the protocols. LwM2M, CoAP etc are some of the protocols used.

1.2.2 Artificial Intelligence Planning

To make the system autonomous, machine intelligence is necessary. When provided with a task by the user, the system has to figure out how to achieve the goal. This is where Artificial Intelligence Planning comes into picture. From the task provided, an application on the management system need to provide the sequence of steps, in this context called plan, which should be executed to complete the task.

1.2.3 Software Development

This project includes setting up of various frameworks and environments and configuring them for software development. Moreover it includes building and packaging of the software, deploying them on proper hardwares etc. Therefore this project can be vaguely put under Software Development, as it provides an insight into some of the phases of the Software Development Lifecycle.

1.3 Objective of the project

The main objective of this project is to develop a prototype of an IoT Management system which is autonomous and adaptive so that proper systems can be developed on this prototype in future. The salient features of this system are efficient communication with the constrained devices, use of a widely accepted device management protocol and an autonomous and adaptive application to monitor and control the system and to take appropriate decisions relevant to the situation.

Surplus bandwidth for communication cannot be guaranteed on all devices. The system was developed keeping this bandwidth constraint in mind. Moreover the devices are not expected to have very good processing capability and power supply. Therefore the protocol used for the communication between the system and client which handles the devices should consume less bandwidth compared to the well known application protocols like HTTP. Coming to the device management protocol, there weren't any device management protocol for the whole IoT context until LwM2M was introduced and will be explained in detail in the coming sections. OMA DM protocol was used only for the Cellular Networks and it failed to be a standard as all the vendors modified it for their purpose. In future it is desirable to have same protocol for all the device management as it eases the communication in heterogenous environment.

Capturing all the conditions for the real world scenario is a hectic task. This is the reason why automating IoT Systems is ineffective. The aim is to create an application that can figure out the solutions for the problems and take decisions based on its current context. This approach works well in the dynamic real world scenarios.

1.4 Organisation of the Report

The rest of the report is organized as follows: Chapter 2 gives an overview of the background theory required to understand the current scenario, methodology and implementation details. Chapter 3 describes the methodology used, explains the implementation details and the use cases selected for implementation. Chapter 4 analyzes the results. Chapter 5 concludes the report with a summary and directions for future work.

CHAPTER 2
BACKGROUND THEORY

This chapter explains the background of the project. It explains the important protocols used, techniques used to make the system autonomous and adaptive and the application and framework used in the making of the system.

2.1 CoAP

The Constrained Application Protocol (CoAP) is a web transfer protocol specialized for use with nodes and network which have lots of constraints in terms of bandwidth in the Internet of Things scenario. It is designed for machine-to machine (M2M) application. Moreover it is an efficient RESTful protocol and thus can be easily translated to and from HTTP. Some of the CoAP features are the following:-

1. Embedded web transfer protocol (coap://)
2. Asynchronous transaction model
3. UDP binding with reliability and multicast support
4. GET, POST, PUT, DELETE methods
5. URI support
6. Small, simple 4 byte header
7. DTLS based PSK, RPK and Certificate security
8. Subset of MIME types and HTTP response codes
9. Built-in discovery
10. Optional observation and block transfer

2.2 LwM2M

The Lightweight Machine to Machine (LwM2M) is a protocol proposed by the Open Mobile Alliance for M2M and IoT device management. It is able to transfer service/application data. Prior to the introduction of LwM2M, OMA relied on OMA DM protocol for device management. But this was applicable only to cellular networks. And it was fragmented by handset vendors using proprietary mechanism. So there arose a need for a common protocol for device management which is independent of the type/category of device or the network used like Cellular, 6LoWPAN, WiFi and ZigBee IP or any IP based constrained networks. Thus LwM2M was developed and it provides an ideal M2M solution. Many object models were developed for mapping the device functionality/services/resources which can be used with LwM2M protocol. This Object Model is extensible. Moreover the object models proposed by IPSO alliance can be

adopted. This is a simple and efficient protocol. The interfaces and payload formats are not complex. LwM2M technical specification defines 9 normative objects, namely, LwM2M Security, LwM2M Server, LwM2M Access Control List, Device, Connectivity Monitoring, Firmware Update, Location, Connectivity Statistics and Software Management. There is provision for creating custom objects too. The defined interfaces are Bootstrap interface, Registration interface, Management interface and Reporting interface. Object access are defined through functionalities like Read, Write and Execute which can be translated to GET, PUT, POST request respectively.

2.3 AI Planning

Planning is an explicit deliberation process that chooses and organizes actions by anticipating their outcomes. It is aimed at achieving some pre stated objectives. The basic idea behind Artificial Intelligence Planning is to convert problems to a search problem. A search problem consists of a state space which is the abstraction of the world, a successor function which will be the actions and which will define the cost of those actions, a start state and a goal test. For describing a search problem, Planning Domain Definition Language, PDDL is used. It allows users to express all the actions in one action schema. Each state is represented as a conjunction of fluents which are the ground functionless atoms. In first order logic, fluents can be called predicates. The action schema lifts the level of reasoning from propositional logic to restricted subset of first order logic. A schema consists of an action name, a list of all the variables used in the schema, a precondition section and effect section. A precondition states the condition under which an action is enabled. The effect describes the way an action updates the predicates/fluents. PDDL inputs are interpreted by planner softwares. There are many open source planner softwares like FF planner, Metric FF planner, Optic planner etc. Planners basically use some search algorithms like Breadth First Search, A*, Enhanced Hill Climbing etc and their combinations and see if the goal state is reached on taking up the actions described from the start state. The sequence of actions which the planner took to reach the goal state is the plan to solve the problem. AI planning is very important in making systems autonomous. The proposed system can be made even more realistic if a knowledge based agent is used along with AI planning.

2.4 Leshan

Leshan is an open source Eclipse IoT project since 2014. It provides libraries that help people develop their own Lightweight M2M server and client. It has modular Java libraries and is using Californium CoAP implementation and Scandium DTLS implementation.

LwM2M data is sent over CoAP in Leshan implementation. It has IPSO objects support along with the provision to create custom objects.

2.5 openHAB 2.0

Open Home Automation Bus or openHAB is an open source automation software. It runs on Java Virtual Machine. The main aim of openHAB is to provide an integration platform for devices and technologies and make them into one single solution that allows overarching automation rules. It also provides uniform user interface. Some of the features of openHAB are:-

1. Absolutely vendor-neutral.
2. Hardware/protocol agnostic.
3. Runs on any device that is capable of running JVM.
4. Has powerful rule engine to fulfill all the automation needs.
5. Provides variety of web-based UIs.
6. Fully open source.
7. Easily extensible to integrate with new systems and devices.
8. APIs for being integrated in other systems.

CHAPTER 3
METHODOLOGY

This chapter explains in detail the methodology used, the implementation details and a few use cases which have been implemented. By the end of this chapter, a clear picture of the solution to the aforementioned problem can be obtained.

3.1 Overview of Solution

For developing a prototype of an autonomous, adaptive IoT Management System, the first step is to create a system which can handle multiple devices. For the effective management of a system which has multiple devices, the best solution is to have a server. As the devices are assumed to be in a constrained environment, CoAP protocol is used for the communication between the server and the clients which manage the devices. Moreover, device management will be an important functionality in future. So the server and the clients will be communicating using LwM2M over CoAP. The server will be responsible for the bootstrapping and registration of the clients. The server should expose APIs so that the clients can be handled by user. So the best approach is to have a REST architecture.

Now as the second step, an application need to be created which can take decisions based on the situation. This application can run either on the server or on a different node. Because of the flexibility and ease of development of the modular approach, the application will be on a different node, independent of the server. As the application is independent of the server, it can even be extended as a service in future. This application will require the current state of the devices. So it will constantly pull the states through the REST API provided by the server. Here the HTTP requests are translated to CoAP requests with LwM2M payloads and sent to the client by the server. The clients either read, write or execute the requests on the devices and send the response back to the server as LwM2M response payload over CoAP. Server translates this response to HTTP response and sends it back to the application. Using this response, the application can infer the current state of the system.

The third step is to make the application plan so that it can take decisions. Planning can either be implemented internally in the application or can be done using external planners. The prototype implementation of the system will be using external planner. The application has to generate the PDDL files dynamically based on the inputs from the server and then call the planner. The planner will generate a plan and return it back to the application and the application will execute those plans using the APIs of the server as before. Figure 3.1

shows the above explanation diagrammatically.

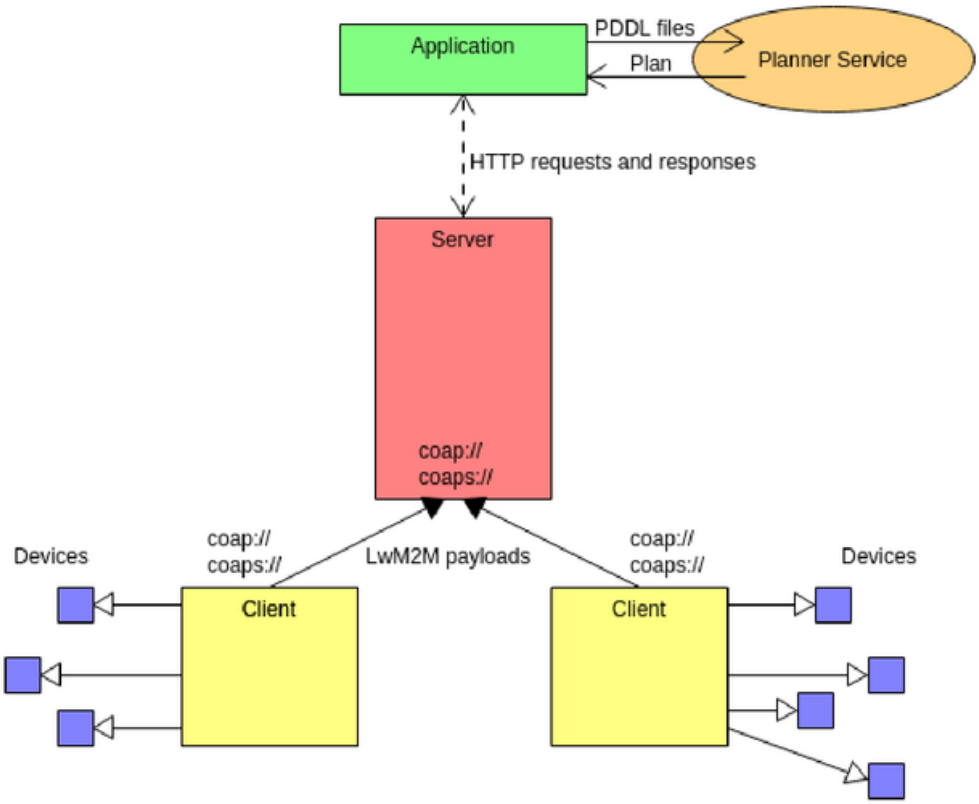

Figure 3.1 : Proposed system architecture

The system should allow simple mechanisms for incorporation of policies in order for effective management. Such incorporation should be handled transparently by the system and should not be implemented manually. Every action taken by the system should not violate the policies defined by the administrators of the system.

The real world scenarios are mostly based on time. For the prototype, Linear Temporal Logic integration is also included to capture the notions like always, eventually etc and make it more suitable for real world applications.

3.2 Implementation Details

Leshan is an open source Eclipse IoT project. It provides libraries that help people develop their own LwM2M server and client. In this open source project, LwM2M payload is transferred on top of CoAP protocol. This is ideal for the scenario. So Leshan Server is used and Leshan Clients specific for the devices are developed. Now for the application

part, openHAB 2.0 is used. Custom bindings are created specific to the requirement. The binding fetches the clients' details and displays it on openHAB's UI. Users can control the clients from this UI. openHAB 2.0 provides features like rules and scripts for automating the devices. These features are used for pulling the states of the devices. Using the states fetched from the client, the PDDL problem file is generated using an external Python Script.

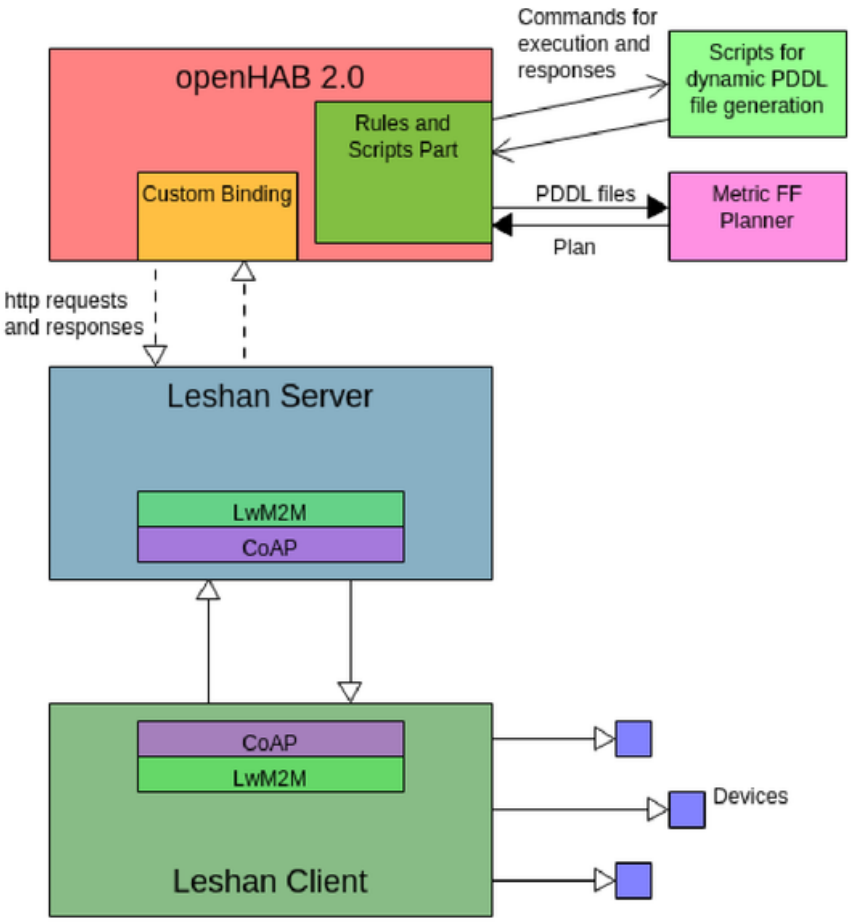

Figure 3.2 : Implemented Architecture

The PDDL file is sent to Metric FF planner and it returns a plan. This plan will be executed by the rules/scripts part of openHAB 2.0. openHAB 2.0 will send HTTP requests using the custom binding which was developed for the Leshan Server that will in turn translate the HTTP requests into CoAP requests and sends to the client and the client will handle the devices accordingly. Figure 3.2 shows the implementation details diagrammatically.

3.3 Use Cases

The whole project has been developed around mainly 2 use cases which are explained in detail below:-

3.3.1 Building use case

In this use case, there are fire sensors and sprinklers in every room in a building. There will also be critical devices like server in some room. Now if there is fire in some room, the fire sensor in that room will detect it. Ideally, when fire is sensed the sprinkler should be turned on. But if there is a critical device like server in that room, the sprinkler should not be turned on all of a sudden. The server must be shut down and then only the sprinkler should be turned on. These decisions come as a sequence of steps from the planner and openHAB 2.0 will execute it in the same sequence to achieve the goal.

Each room's fire sensor and sprinkler actuation will be handled by a Leshan client. Critical devices' actuation will also be handled by a Leshan client. All these clients connect to a Leshan Server. A custom binding was developed for openHAB 2.0 to communicate with this Leshan Server. The Leshan Server was modelled as a 'Thing' and all the Leshan clients' functionalities like fetch state and its response etc were modelled as 'Channels'. Cron jobs are written to constantly fetch the state of the clients. This is done in the scripts part of openHAB 2.0. The rules part continuously monitors the fire sensor channel of the clients to see if it has been activated. If it is activated, the rules part calls another script which will take all the state information, generate a PDDL problem file based on the current state and the goal state which is to have the sprinkler of the particular room turned on. After PDDL problem file generation, the Metric FF planner is called with the PDDL domain file which is the model of the use case and the dynamically generated PDDL problem file as arguments. The planner will generate a plan for the system. The script parses the plan and is executed in the same sequence by passing commands to the appropriate channels. The custom binding developed will translate these commands and corresponding API calls are done to the Leshan Server. Leshan Server translates these HTTP requests to CoAP requests with LwM2M payloads and sends it to respective Leshan Clients who handles the actuations like turning off/on the server/sprinkler etc.

3.3.2 Camera use case

The aim with this use case is to figure out a way to incorporate policies in system's decision making. In this use case, there are a number of cameras covering some regions. Out of the total regions, few of them are critical, that is, those regions should always be monitored by

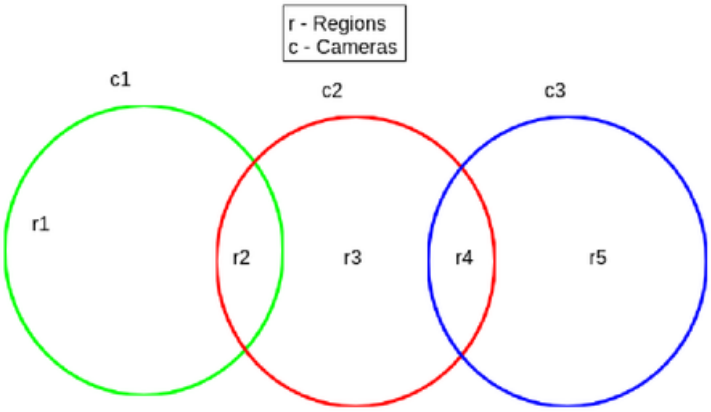

Figure 3.3 : Three camera use case

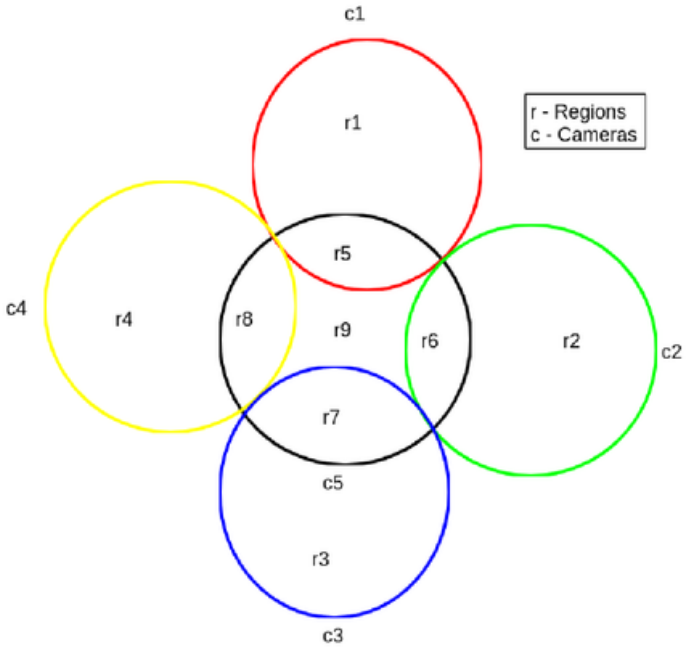

Figure 3.4 : Five camera use case

at least one camera. More than one camera can cover a region. The goal here is to upgrade all the cameras or some of them. To upgrade a camera it must be turned off which can result in some regions being not monitored. This is the problem that needs to be tackled. The system must figure out the order in which the cameras must be upgraded such that no critical regions are left unmonitored. This use case was analyzed using three camera situation, five camera situation and thirteen camera situation. Figures 3.3, 3.4 and 3.5 show

the problem diagrams. Three camera use case was used to approach the problem in a simple way. Five camera use case was used to see how the simple solution deduced from the above case can be adapted in complex cases. Thirteen camera use case was used to see how well the Metric FF planner performs when the problem scales. It has 61 regions out of which 24 are critical.

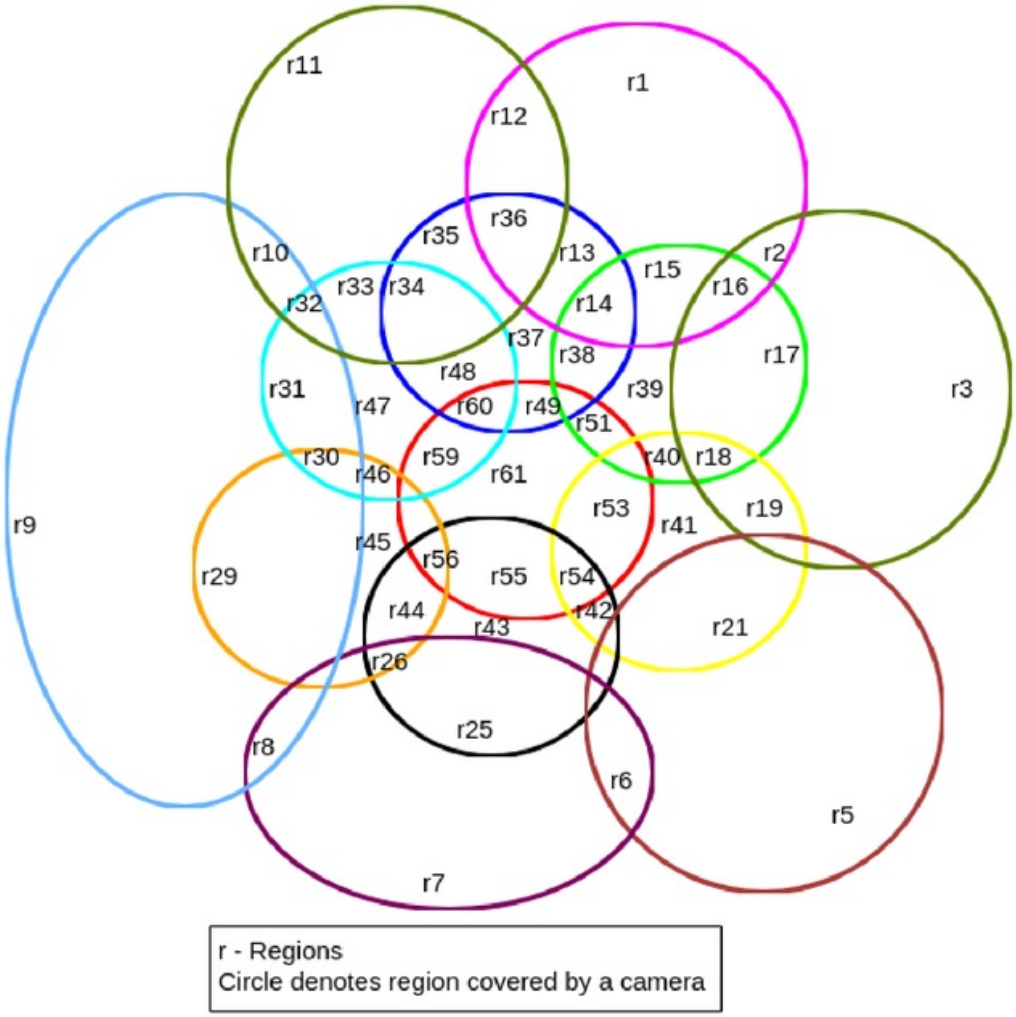

Figure 3.5 : Thirteen camera use case

Like in the previous use case, this problem too can be modeled such that the actions in the domain file have preconditions which ensure the above mentioned condition. But the aim is to incorporate policies in an elegant way to the system. Actions are something which the system must execute in the end to reach the goal. For every policy change, modifying the actions' preconditions in the domain file is a tedious task.

Ideally, policies are to be written in a separate node or simply a file and domain file needs to be dynamically generated before the planning step which will include the policy. To achieve this a dummy action is created. This action will add or remove a predicate representing safe/unsafe state. Whatever policies are there, it will come inside the effect part of this action as conditions. If the conditions are satisfied, predicate representing safe state is set. If any of the conditions are violated, the predicate will be removed. This predicate will be there in the precondition of other actions as a necessary condition. So other actions will be taken only if the system is in safe state.

The domain file can be modelled in such a way that in the plan, after every action which has a mapping to real action, there will be this dummy action in the sequence. If one of the actions chosen by the planner violates the policy, the dummy action which will be picked up next, will remove the predicate representing the safe state thus blocking the planner from proceeding to the goal state. Therefore the planner will generate a sequence of steps to the goal which will satisfy the policy on every action. Even if the policies are changed, it will be incorporated elegantly inside the dummy action. openHAB 2.0 scripts can be modified easily to remove the dummy actions in the plan given by the planner.

Linear Temporal Logic is used to capture notions like always, eventually, next time, until and release. There may be many plans to solve the problem. Among these plans, the required plan will satisfy the linear temporal logic formula in every step. Most of the mentioned operators of linear temporal logic can be viewed as an extension to the policies mentioned above. But things get complicated when they occur in combinations. Simple conjunction of the formulas would not work in this case. So the linear temporal logic formula is converted to a Büchi automaton [3] and then it is implemented in the dummy action mentioned in the previous paragraph.

CHAPTER 4
RESULT ANALYSIS

In this chapter the results will be analyzed using the two use cases explained in the previous chapter.

4.1 Building use case

First a LwM2M over CoAP server was created. Eclipse Leshan Server code was forked and modified to test communication with openHAB 2.0 which is the application. Figure 4.1 shows the screenshot of the console of this server. From the logs it can be observed that the

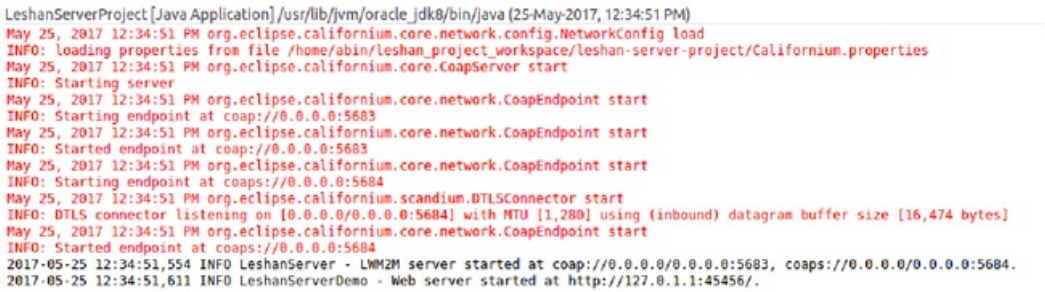

Figure 4.1- Leshan Server Logs

server is listening for CoAP communications on port numbers 5683 (coap://) and 5684 (coaps://) which are the well-know port numbers for CoAP. The web GUI for this server is set at port number 45456. By default it was 8080 but here it will be used for openHAB 2.0. Now 3 LwM2M clients are connected to this server. One of these clients will handle the building server and the other two will handle each rooms' fire sensor and sprinkler actuation. Figure 4.2 show the screenshot of the console of client that controls the building server.

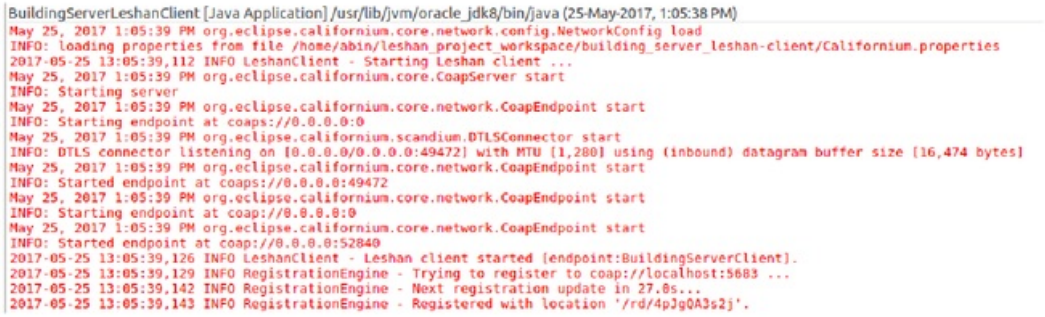

Figure 4.2 - Building Server Client Logs

It can be observed that the clients are connecting to port number 5683 which is the port on which the server is listening. The server returns an id to the client and it is visible on the figure in the end. All clients' console logs will be similar to this. Figure 4.3 shows the Web GUI of the server. The endpoint name and the unique registration id can be observed.

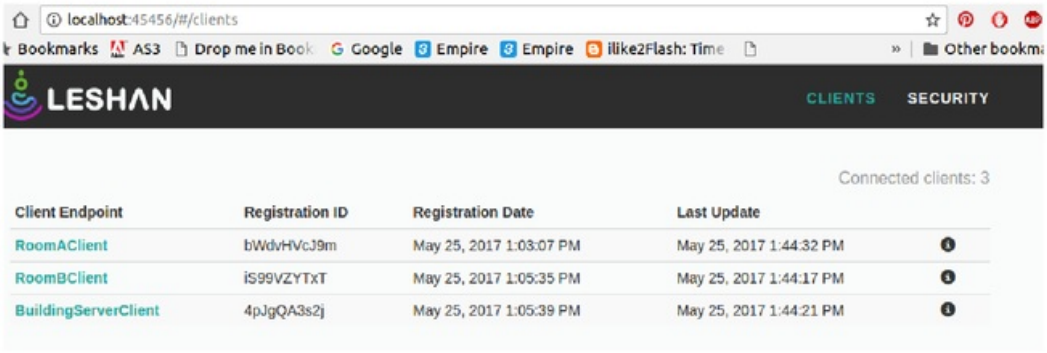

Figure 4.3 - Leshan Server GUI

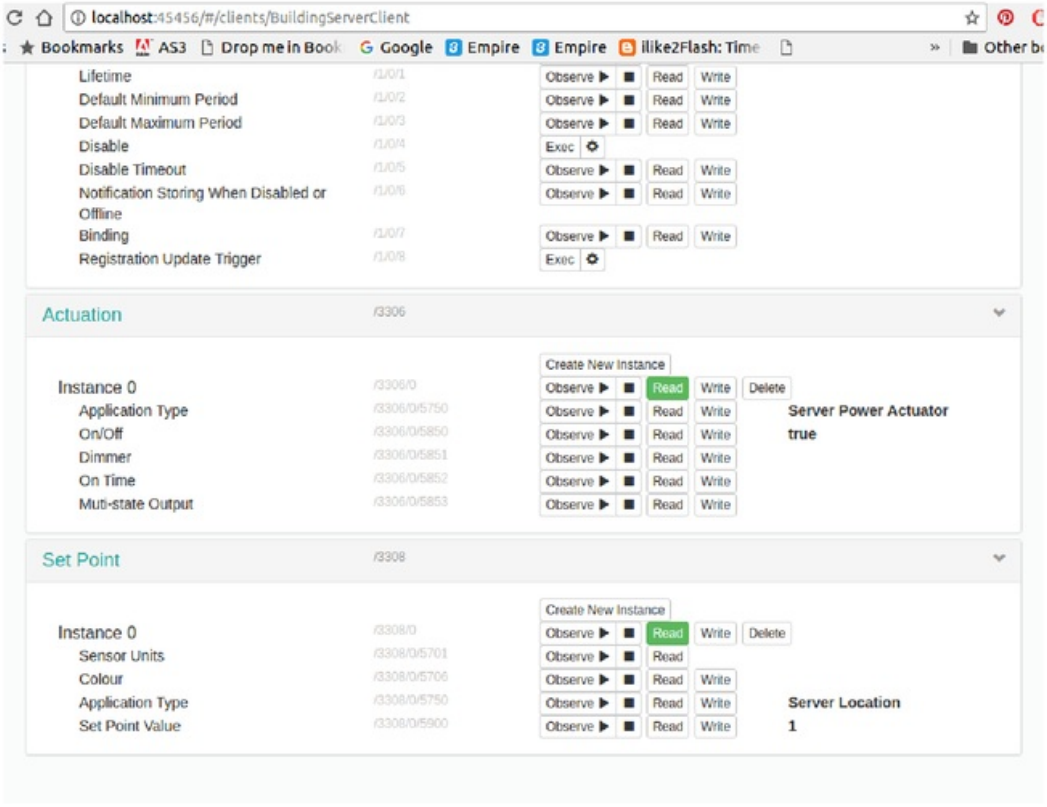

Figure 4.4 : Building Server Client Controls

Building Server Client has been modeled to have an Actuation Object which has been defined by IPSO Alliance through which user can turn on/off the server. It also needs to keep the information regarding the room where it is placed. A Set Point Object which is also defined by IPSO Alliance has been modelled for this purpose. Figure 4.4 shows the information which this client holds. Actuation Object's ID is 3306 and Set Point Object's ID is 3308. Only one instance is there for both these objects. The instances are represented by 0. Both the objects have a resource called Application Type and the resource ID is 5750 which is same in both the cases. This resource is used to describe what the particular object does. Actuation Object has a resource called On/Off and its ID is 5850. This resource shows the current state of the client's actuation, that is, whether the building server is on or off. User can also set the the value for resource and turn the building server on or off. Set point object's Set Point Value resource is used for storing the information regarding the room in which the building server is placed. Here value 0 means the building server is in Room A and 1 means in Room B.

Room clients' cases are also similar. They too have an Actuation Object using which user can turn on or off the sprinkler in the room similar to the turning on and off of the building server. The rooms also have a fire sensor. Generic Sensor Object which has been defined by IPSO Alliance is used for handling fire sensor values. The object ID for Generic Sensor Object is 3300. It has a resource called Sensor Value which has an ID 5700. This resource will provide the state of the fire sensor. In this if it 0 then the situation is normal and if it is 1 then there is fire. Figure 4.5 shows a room client's controls in server GUI.

So user can identify the states of the devices by fetching the required resources from the required instance of the objects and can change the state by writing to these resources. Leshan server has exposed the REST APIs to do this. The URL looks like <server_ip_address>:<web_gui_port_number>/api/clients/<client_endpoint>/<object_id>/<instance_id>/<resource_id>[?format=<JSON or TLV>]. To fetch a resource value, a HTTP GET request is used and for providing a value to a resource, a HTTP PUT request is used.

In Figure 4.6, the GET request on the Request Header source can be observed. The host is the server IP address and the web GUI port number which was mentioned earlier. From the URL in the request header (/api/clients/RoomAClient/3306/0/5750), it can be understood that the request is to fetch Room A Client's (/RoomAClient/) Actuation Object's (/3306/0/) Application Type (/5750/). JSON format is used throughout this project. In Figure 4.7, the response for the above mentioned GET request can be seen. The response is a JSON string which contains requested resource's ID and its value, in this case 'Room A Sprinkler

control Actuator'.

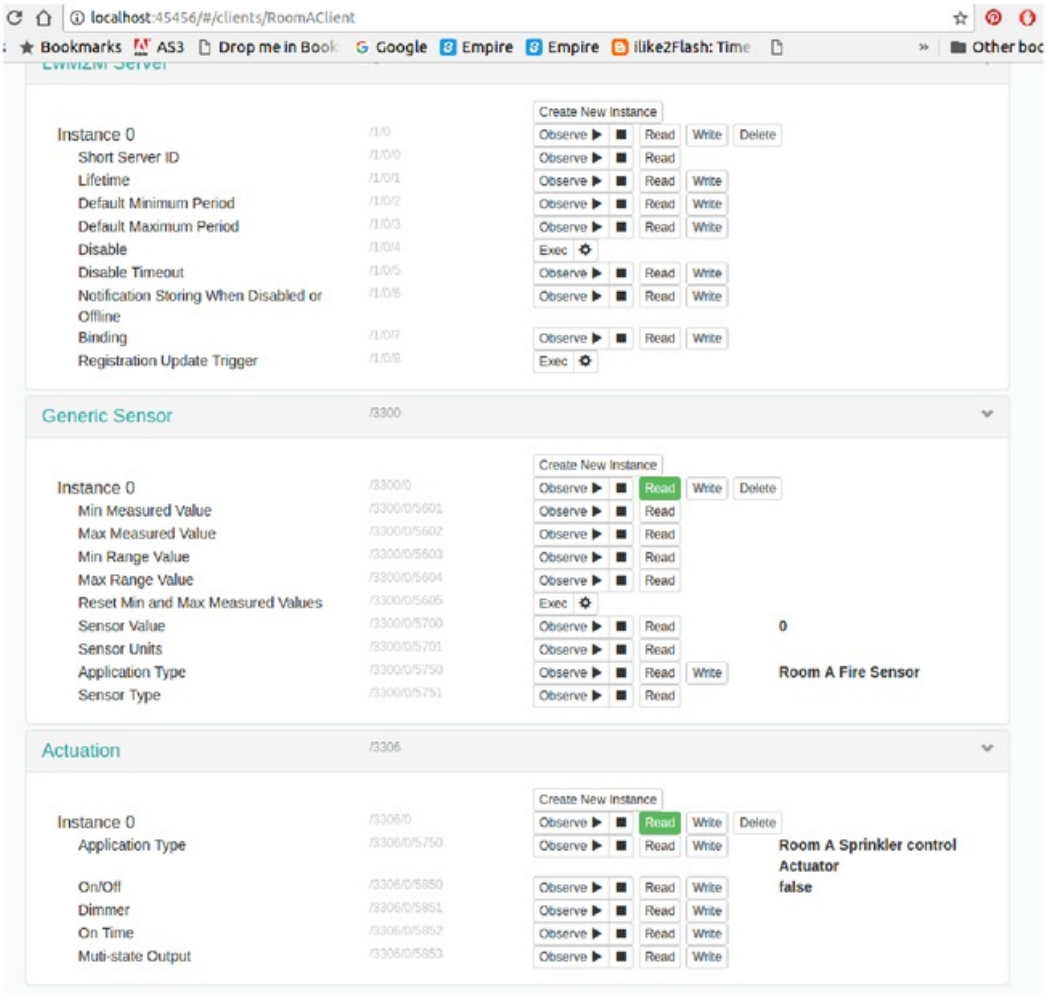

Figure 4.5 : Room A Client's Controls in Server GUI

Similarly Figure 4.8 shows the PUT request to the On/Off resource of Room A Client's Actuation Object which handles the sprinkler actuation. If this resource is made true the client turns on the sprinkler and vice versa. Like the GET request explanation, this request header also has the URL and the difference is that here it is PUT instead of GET. The On/Off resource's id 5850 can be seen towards the end of URL. As it is PUT request, a request payload which contains the resource's id and the value to be written can also be observed. In this case it is to turn on the sprinkler and hence the value in the request payload is 'true'. If the PUT request was successful, a response will be received in JSON format with status as 'CHANGED'. Figure 4.9 shows the response to the above mentioned PUT request.

Figure 4.6 : GET request on Application Type resource of Actuation Object

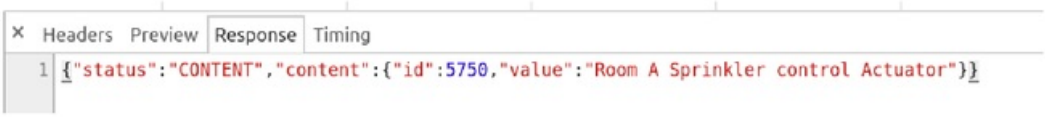

Figure 4.7 : GET request's response

Figure 4.8 : PUT request to On/Off resource with value 'true'

The application openHAB 2.0 uses these REST API's to talk to the Leshan Server as explained in the Methodology chapter. The custom binding was designed to make these GET and PUT requests to do 'Read' and 'Write' actions (POST request for 'Execute'; not used in this use case) on resources and this is how the states of the devices are fetched and the deduced plan is executed.

Figure 4.9 : PUT request's response on success

Till now it has been explained how the Leshan Server REST APIs are used to handle resources of a particular Leshan Client and how it is used from openHAB 2.0. But information about which all clients are there and its details like what all objects are implemented in each client must be known in the application part. Leshan Server keeps all

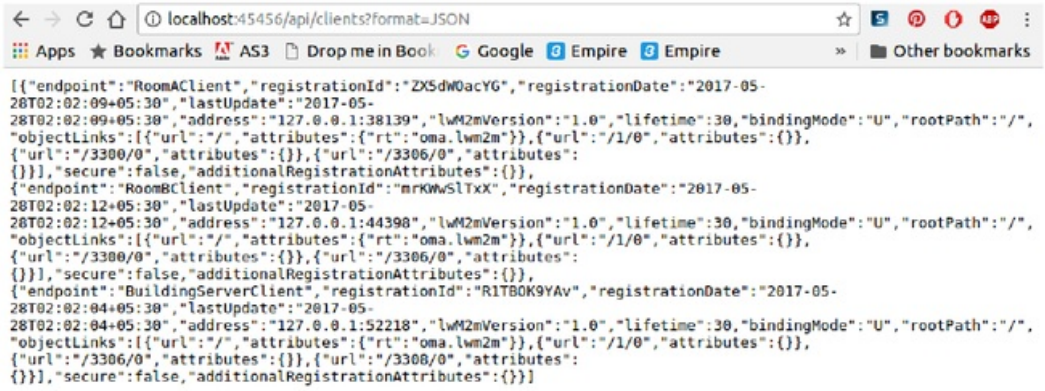

Figure 4.10 : REST API call for registry details in the Building use case

Figure 4.11 : Parsed response for registry API call

the details of its client in a registry. It exposes this registry details through a REST API. The URL looks similar to the previous one mentioned in one of the paragraphs above:

<server_ip_address>:<web_gui_port_number>/api/clients[?format=<JSON or TLV>]. Figure 4.10 shows the client details when fetched through a browser using the registry API. Figure 4.11 will provide a better understanding. Endpoint and Registration ID are the unique identifiers for each client. Endpoint is decided by the client and the Registration ID by the server. Lifetime is in seconds and the server deregisters the client if a registration update isn't received by the server from the client in that lifetime. Address is obviously the IP address of the client. Binding Mode 'U' means it is using UDP to send CoAP messages.

Figure 4.12 : Room A Client's object links

'Object Links' field provides the details on which all objects have been implemented in that client. Its URL can be used to form the API request URL for fetching its states or for modifying them. Figure 4.12 shows three objects for Room A Client. The first one in this field just gives the root. The second one with URL '/1/0' show that this client has LwM2M server object which keeps the details of the LwM2M server on client side. Appending '/1/0' to the base API url and adding the resource id towards the end will give us the means of accessing that resource. Similarly '/3300/0' and '/3306/0' shows that there is one Generic Sensor object implementation and one Actuation Object, in this case Fire Sensor and Sprinkler Actuation respectively.

openHAB is a Maven project. openHAB team has provided a script to help developers start with binding development. On running that script the basic files like pom.xml, binding.xml, thing_types.xml, HandlerFactory.java, Handler.java, Constants.java etc are generated. The basic file and folder structure generated is the required skelton and the custom binding was built on top of it. A configuration part was created where the IP address and port number of

the Leshan server is provided so that the binding can do the REST API calls to it. The Thing created using this binding initially has a switch called 'Refresh' which is used to fetch the whole client details from the Leshan Server and a text field to show the number of clients.

When the 'Refresh' button is clicked, the binding sends the API call to fetch the registry details of the Leshan Server. This procedure is done in a separate thread. The response is a JSON string. When parsed, the number of elements in the array gives the number of clients. Figures 4.10 and 4.11 gives an insight into this. A hash table of client handlers is maintained in the custom binding with their registration ids as the key. On parsing the response to suitable objects it first looks if there is any client in the table which is not there in the response. If there exist a client like this, it means the client has been deregistered on the Leshan server. So it needs to be removed from the hash table in the binding. All its corresponding channels and other memory allocations are disposed. After this procedure, the binding looks if any client in the parsed response is missing from the table. If there exist a client like this, it means that there is a new registration on the Leshan Server. Then it creates the corresponding channels by looking up the object links from the response. For this purpose a separate handler was created for each object like Generic Sensor object, Actuation object etc. These handlers are the ones which create the corresponding channels to fetch and modify the resources and show the response. Each client can have multiple object handlers as each client may be handling multiple objects. All these handlers are put into one list and then placed in the hash table with the client registration id as the key.

A channel unique id is obtained when a channel is created from openHAB. This is also maintained in another hash table to fetch the corresponding handlers by giving the channel unique id. On creating the channels the corresponding items like switches and text fields come up in the UI. Now when any of this items are used, for example if a switch is turned on or if a text field in edited, the binding will be notified from the openHAB core. In the binding code, the control is received on 'handleCommand' method in the Handler class which was created for the developer by the script along with the respective channel unique id and the command like turn on or turn off if the item is a switch. From there another thread is created to handle the command initiated by the user from the UI. Inside this thread the corresponding object handler from the channel hash table is fetched and the command will be passed to the object handler. Inside the object handler the binding will understand if it is a resource read request or modify request and the corresponding REST API calls are initiated. The response is shown in the corresponding item linked to the response channel by the same object handler. This maven based custom binding is exported and is used as an add on in openHAB. Figure 4.13 shows how this binding description is shown in

openHAB's UI.

Using this binding a 'Thing' is created which will be used to handle the Leshan Server. Figure 4.14 shows how the thing can be configured. Notice the IP address and port number. Figure 4.15 shows how the thing is shown in openHAB's UI after it is created.

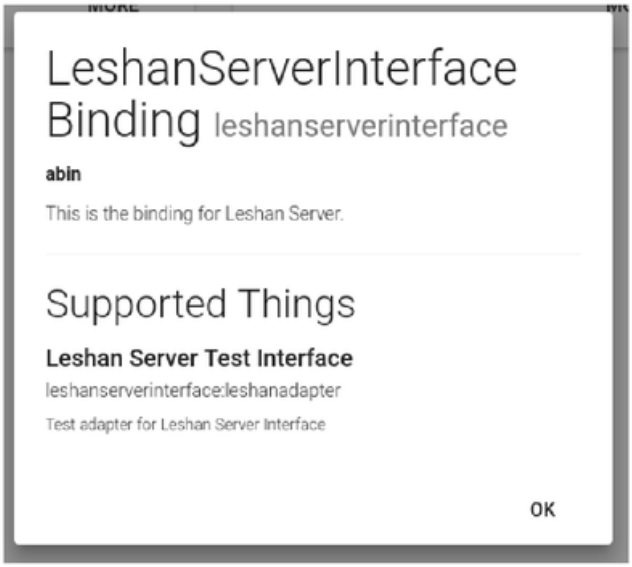

Figure 4.13 : Custom binding's description in openHAB UI

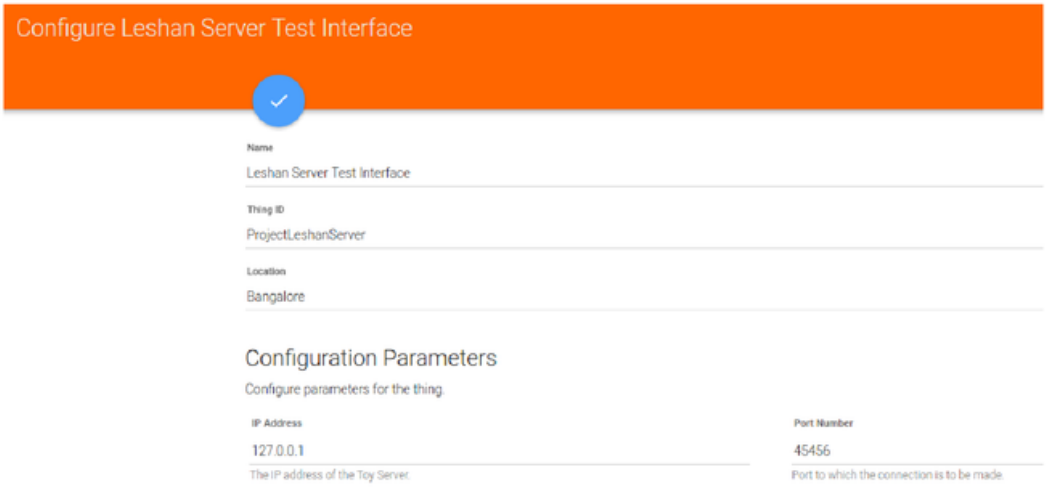

Figure 4.14 : Thing creation procedure using custom binding

It only has two channels initially, the refresh button and its response. On clicking the refresh button many dynamically generated channel create their respective items from the

response from the Leshan server and this is shown up in the UI. Figure 4.15 shows the new state of UI after the request to fetch the whole client details from the Leshan server was send. This UI has a different flavour from the previous UI but the rest are the same.

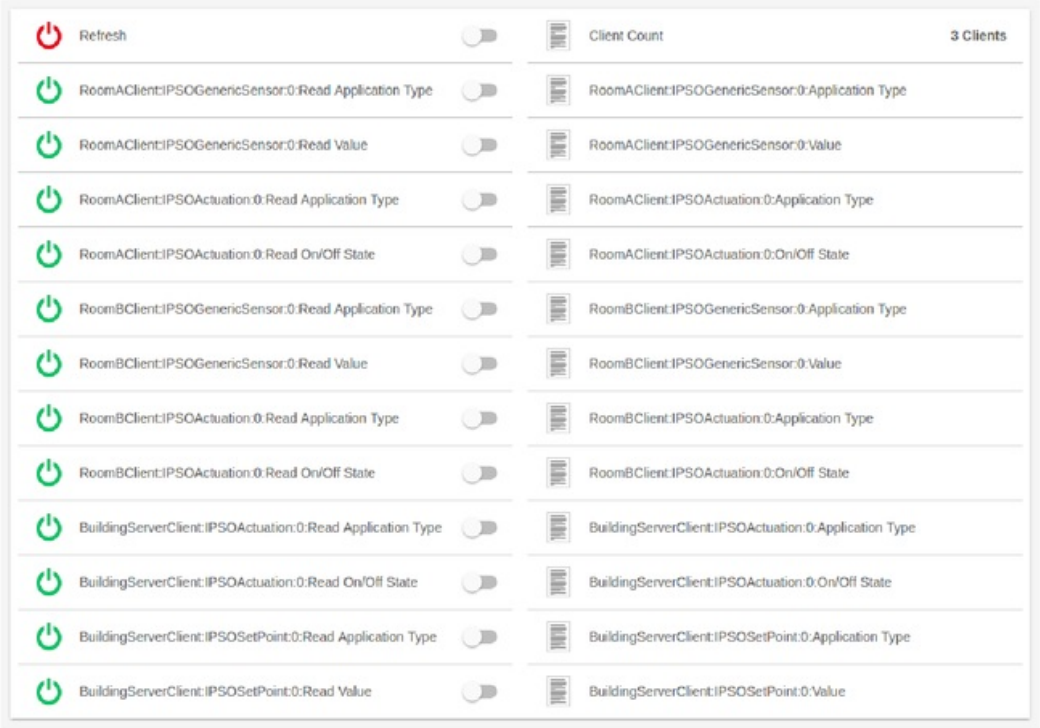

Figure 4.15 : Thing after clicking Refresh button

Clicking on the button will send the corresponding read request to the Leshan server and the response will be displayed in the respective items. Figure 4.16 shows how a typical response will be displayed on the UI.

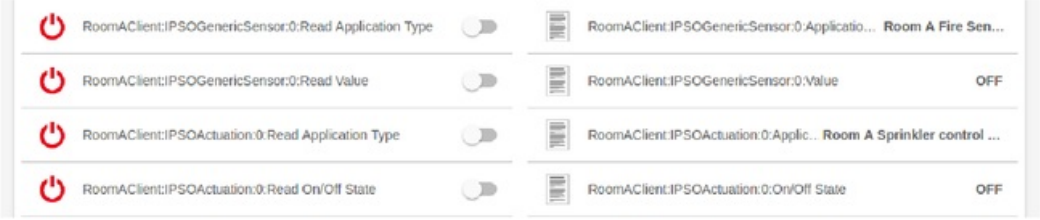

Figure 4.16 : Response for the button clicks

The system needs to fetch the device states automatically. In this example it needs to know the state of fire sensors, sprinkler actuators of the two room clients, building server location and its actuation state frequently. For this openHAB's rules and scripts part is used. A cron job was created which will fetch the required states in every ten seconds. Figure 4.17 shows

the code used to achieve this. The '0/10' in the 'when' part of the rule triggers it every 10 seconds. It can also be used to trigger events on particular dates but that does not suit the requirement. The six statements in the 'then' part issues command 'ON' to the six switches

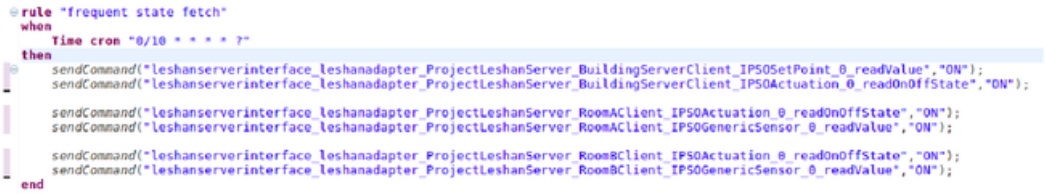

Figure 4.17 : Cron job code

which were used to fetch the device states manually. The moment one of the room client's fire sensor state is on, the corresponding script to handle this situation is called. Figure 4.18

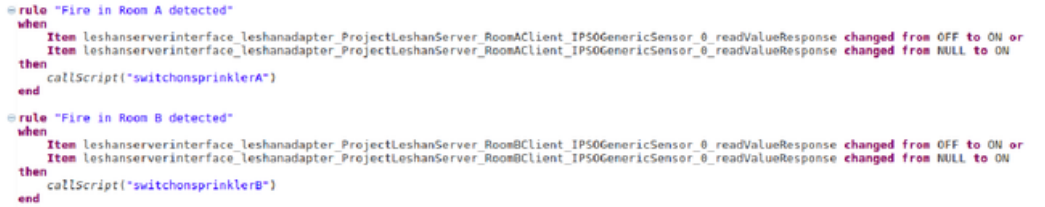

Figure 4.18 : Rule to trigger sprinkler when fire sensor is ON

shows how it is done. Inside the script which is called, a shell script is executed which will first generate the problem file from the current states of the client resources. Then it will call a planner service and provide the domain and the generated problem PDDL file. The planner service would return a plan and it will be captured by the openHAB as a string. In this case, as shown in figure 4.19, the variable 'response' holds the plan as string. Now this

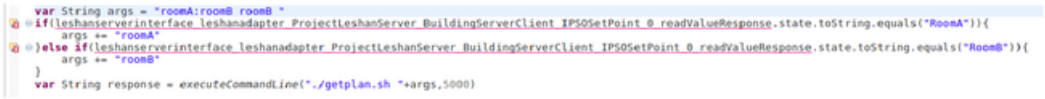

Figure 4.19 : Shell script execution for PDDL generation and planning

string is parsed as commands and respective parameters. These commands are mapped to commands on particular items in openHAB which will in turn send HTTP requests to Leshan Server as mentioned above. In this particular use case, only two actions have been modelled, one is to switch on the sprinkler and one is to switch off the server. Both these actions have room identifiers as its' parameters. Figure 4.20 and 4.21 shows the plan generated for different scenarios where in one fire is there in Room A and in later one in Room B. In both cases the building server was located in Room B. In first case the system

decided to turn on the sprinkler in Room A. In second case directly turning on the sprinkler would have affected the building server located in Room B. So the system figured out that

```
ff: found legal plan as follows
step    0: SWITCH_ON_SPRINKLER ROOMA
```

Figure 4.20 : Plan generated when building server is in Room B and Fire is in Room A

```
ff: found legal plan as follows
step    0: SWITCH_OFF_SERVER ROOMB
        1: SWITCH_ON_SPRINKLER ROOMB
```

Figure 4.21 : Plan generated when both building server and Fire is in Room B

the first step was to turn the building server off and in the next step the sprinkler was turned on. This show that the system is autonomous that is it figured out what to do in each scenarios. When the building server is moved to Room B, the system adapts to it and takes decisions based it.

```
ff: found legal plan as follows
step    0: SWITCH_OFF_SERVER ROOMA
        1: SWITCH_ON_SPRINKLER ROOMA
```

Figure 4.22 : Plan generated when both building server and Fire is in Room A

```
ff: found legal plan as follows
step    0: SWITCH_ON_SPRINKLER ROOMB
```

Figure 4.23 : Plan generated when building server is in Room A and Fire is in Room B

Figure 4.22 and 4.23 above shows the plan generated in this situation. This shows that the

```
while(i < len){
    cmd = commands.get(i)
    param = params.get(i)
    if(cmd.equals("switch_off_server")){
        item = "leshanserverinterface_leshanadapter_ProjectLeshanServer_BuildingServerClient_IPSOActuation_0_readOnOffStateResponse"
        sendCommand(item,"OFF")
    }else if(cmd.equals("switch_on_sprinkler") && param.equals("roomA")){
        item = "leshanserverinterface_leshanadapter_ProjectLeshanServer_RoomAClient_IPSOActuation_0_readOnOffStateResponse"
        sendCommand(item,"ON")
    }else if(cmd.equals("switch_on_sprinkler") && param.equals("roomB")){
        item = "leshanserverinterface_leshanadapter_ProjectLeshanServer_RoomBClient_IPSOActuation_0_readOnOffStateResponse"
        sendCommand(item,"ON")
    }
    i++
}
```

Figure 4.24 : Plan execution from openHAB scripts

system is adaptive. These plans are read as a string and parsed appropriately in openHAB

scripts section as mentioned above. Figure 4.24 shows how the mapping and execution is done. As the system execute the plans on its own, autonomy and adaptivity is brought into execution level. Figure 4.25 show the states after fire is detected in Room A.

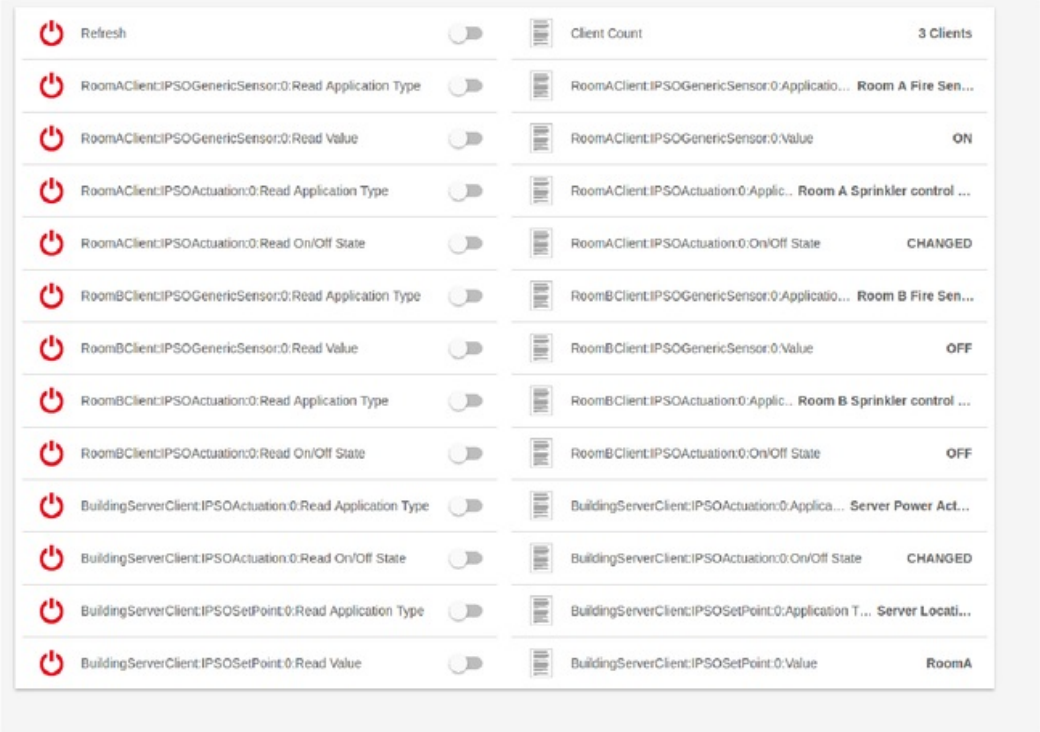

Figure 4.25 : States after fire sensor is triggered in room A

Notice the state of sprinkler actuation in Room A and Building server actuation in Figure 4.25. Both shows ‘CHANGED’. It implies that sprinkler was changed from off to on and server from on to off. After few seconds it will be changed to its actual states which is on

Figure 4.26 : Logs after the plan execution

and off respectively. Figure 4.26 shows the logs of these events. This gives the order in which the commands have been executed. Here, the building server is turned off first and

then the sprinkler is turned on.

4.2 Camera use case

Few informations are required before solving this use case where it is required to incorporate policies and linear temporal logic. Regions that can be covered by each camera needs to be known. It is also essential to know whether a camera is on or off. If it is on, the regions that can be covered by that camera will be monitored and vice versa. It is also required to know if a region is critical or not. And finally, the upgradation status of the camera should also be know.

The actions in domain file which can be mapped to real world actions are turn on, turn off and upgrade. All of these actions act on cameras. As explained in the previous chapter, a dummy action was created. It is called ‘safety check’ action. The domain file is created in such a way that after every action which can be mapped to the real world action, ‘safety check’ action will be taken. For this purpose, a ‘check’ predicate was used. Whenever this predicate is present, no other actions can be taken by the planner other than the ‘safety check’ action because in the precondition of those actions, the planner looks for the non existence of ‘check’ predicate and in ‘safety check’ it checks for the existence of the predicate. After every action which can be mapped to real world, the ‘check’ predicate is added in their effect. This is how planner is blocked from proceeding with other actions other than ‘safety check’. Now when ‘safety check’ is done, the ‘check’ predicate is removed in its effect. So the planner can only take up other actions except ‘safety check’ because ‘safety check’ requires existence of ‘check’ predicate. In the ‘safety check’ action, a ‘safe’ predicate is added, if the policy mentioned is satisfied and removed if it violated. If the policy is violated, the planner can not take any actions which map to the real world as all those actions require the existence of ‘safe’ predicate to be taken up. And the planner can not take ‘safety check’ again as it was taken before which removes the ‘check’ predicate whose existence is required in the precondition of the action and can only be added by the effect of other actions. So planner is left with no other option but to go back to the last state where ‘safe’ predicate existed.

This approach in effect ensures that all the actions that can be mapped to the real world when taken will not violate any policies and the system will be safe in all intermediate states before the goal state. Moreover, as all the policies are incorporated in the ‘safety check’ action, any change in the policies doesn’t break the domain file as these policies can be smoothly added to the action. In the previous use case all the conditions were added as precondition and effect in all the actions. It is really hard to incorporate policy changes in that approach as the developer needs to rewrite all the preconditions and effects.

In this use case the policy is that no critical section should be left unmonitored. Figure 4.27 shows that this is written in 'safety check' using PDDL. Note the 'effect' section. The first

```
(:action SAFETY-CHECK
  :parameters ()
      :precondition (check)
      :effect
              (and    (not(check))
                      (when
                              (forall (?r - region) (imply
                                       (critical ?r)
                                       (exists (?c) (iscovering ?c ?r))))
                              (safe)
                      )
                      (when
                              (exists (?r - region)
                                      (and (critical ?r)
                                       (forall (?c - cam) (not (iscovering ?c ?r)))))
                              (not (safe))
                      )
              )
)
```

Figure 4.27 : SAFETY-CHECK action for the camera use case

'when' statement introduces the 'safe' predicate and the second removes the 'safe' predicate. The first 'when' statement checks if the policy is satisfied. The condition can be understood as 'if for all region, if the region is a critical region (critical ?r), then if there exist a camera which covers that region (iscovering ?c ?r), then the state is safe'. The second 'when' statement can be interpreted in a similar way as 'if there is a region which is critical (critical ?r), for all cameras, none are covering that region (not (iscovering ?c ?r)), then the state is unsafe (not safe)'. The 'check' predicate whose existence is checked in the precondition is removed in the effect section. So after every 'safety check' action, it will be known if the system is in safe or unsafe state, that is, if the policies have been satisfied or violated.

Figure 4.28 shows the plan generated for the three camera use case shown in figure 3.3. 'r2' and 'r4' are the critical regions in this case. Here the second camera is initially off and first and third cameras are on. The second camera is turned on in the fourth step after upgradation. After this turning off both first and third cameras doesn't violate the policy as the second camera will be covering the critical regions. Figure 4.29 shows the plan generated for the five camera use case shown in figure 3.4. 'r5', 'r6', 'r7' and 'r8' are the critical regions in this case. Here all the cameras are on initially. Both the plans never violate the policy of no unmonitored critical region. Once the system gets this plan, it will strip off all the 'safety check' action and can execute the other actions in the same order.

```
ff: found legal plan as follows
step    0: SAFETY-CHECK
        1: UPGRADE C2
        2: SAFETY-CHECK
        3: TURN-ON C2
        4: SAFETY-CHECK
        5: TURN-OFF C1
        6: SAFETY-CHECK
        7: UPGRADE C1
        8: SAFETY-CHECK
        9: TURN-ON C1
       10: SAFETY-CHECK
       11: TURN-OFF C3
       12: SAFETY-CHECK
       13: UPGRADE C3
       14: SAFETY-CHECK
       15: TURN-ON C3
       16: SAFETY-CHECK
```

Figure 4.28 : Plan generated for three camera use case

```
ff: found legal plan as follows
step    0: SAFETY-CHECK
        1: TURN-OFF C1
        2: SAFETY-CHECK
        3: UPGRADE C1
        4: SAFETY-CHECK
        5: TURN-ON C1
        6: SAFETY-CHECK
        7: TURN-OFF C2
        8: SAFETY-CHECK
        9: UPGRADE C2
       10: SAFETY-CHECK
       11: TURN-ON C2
       12: SAFETY-CHECK
       13: TURN-OFF C3
       14: SAFETY-CHECK
       15: UPGRADE C3
       16: SAFETY-CHECK
       17: TURN-ON C3
       18: SAFETY-CHECK
       19: TURN-OFF C4
       20: SAFETY-CHECK
       21: UPGRADE C4
       22: SAFETY-CHECK
       23: TURN-ON C4
       24: SAFETY-CHECK
       25: TURN-OFF C5
       26: SAFETY-CHECK
       27: UPGRADE C5
       28: SAFETY-CHECK
       29: TURN-ON C5
       30: SAFETY-CHECK
```

Figure 4.29 : Plan generated for five camera use case

The ‘safety check’ was further optimized by removing the first ‘when’ clause which asserts that the system is safe. In this case, the system is in safe state unless an action is taken. So it is unnecessary to assert the system is safe. Figure 4.30 shows the new ‘safety check’ action.

```
(:action SAFETY-CHECK
  :parameters ()
      :precondition (check)
      :effect
              (and    (not(check))
                      (when
                              (exists (?r - region)
                                      (and (critical ?r)
                                       (forall (?c - cam) (not (iscovering ?c ?r)))))
                              (not (safe))
                      )
              )
)
```

Figure 4.30 : Optimized SAFETY-CHECK action

A policy injection script was developed to incorporate policies into the ‘safety check’ action. Tags are placed in the domain meta file. The policy must be specified in PDDL in the policy file. Figure 4.31 shows the policy file format which is used in this project. The

```
{
        "policy_condition":"(forall (?r - region) (imply (critical ?r) (exists (?c) (iscovering ?c ?r))))",
        "policy_effect":"(safe)"
}
```

Figure 4.31 : Policy file content used in the project

script written in Python will fetch the policy, negate it, and places in the place of the tag and domain file is generated. Figure 4.32 shows the ‘safety check’ action in the meta domain file with tags.

```
(:action SAFETY-CHECK
  :parameters ()
      :precondition (check)
      :effect
              (and    (not(check))
                      (when <policyTag> <policyEffectTag>)
                      (when <NegPolicyTag> <NegPolicyEffect>)
              )
)
```

Figure 4.32 : SAFETY-CHECK action in meta domain file

```
(:action SAFETY-CHECK
  :parameters ()
      :precondition (check)
      :effect
              (and    (not(check))
                      (when (forall (?r - region) (imply (critical ?r) (exists (?c) (iscovering ?c ?r)))) (safe))
                      (when (not (forall (?r - region) (imply (critical ?r) (exists (?c) (iscovering ?c ?r))))) (not (safe)))
              )
)
```

Figure 4.33 : SAFETY-CHECK action in the generated domain file

System administrators can edit the policy file. The domain file will be generated automatically when planning occurs. Figure 4.33 shows the 'safety check' action in the generated domain file. Writing policies in PDDL is not an elegant method and it is discussed in the Conclusion and Future Scope of Work chapter.

For including linear temporal logic in planning decisions, a fluent is used to keep the information of the state. In the camera use case, each camera can be in different states. So a fluent was required to keep the information of each camera. A number fluent was used in this case were the number signifies the state. The requirement here is that the camera cannot be turned on right after upgradation. It has to wait for at least three steps. The linear temporal logic formula used to analyze this use case is 'wait U (upgrade & XF(XXX turn_on))' where U, &, X and F are until, and, next step and eventually operators

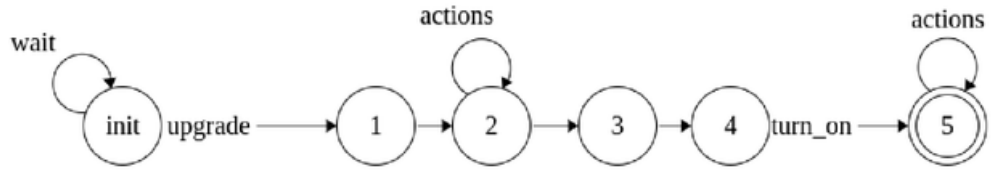

Figure 4.34 : Generated automaton for the LTL formula used in the use case

respectively. The Automaton generated is shown in Figure 4.34. This automaton is implemented inside the 'safety check' action. If the transitions doesn't follow the pattern shown in figure 4.34 the 'safe' predicate is removed blocking the planner from proceeding

```
;; IMPLEMENTATION OF BUCHI AUTOMATON
(when
    (exists (?c - cam) (and (>= (current_state ?c) 1) (<= (current_state ?c) 3) (ison ?c)))
            (not (safe)
    )
)

(forall (?c - cam)
        (when (and (= (current_state ?c) 0) (upgraded ?c) (not (ison ?c)))
              (increase (current_state ?c) 1)
        )
)

(forall (?c - cam)
        (when (and (>= (current_state ?c) 1) (<= (current_state ?c) 3))
              (increase (current_state ?c) 1)
        )
)
;;BUCHI AUTOMATON END
```

Figure 4.35 : Implementation of LTL formula in SAFETY-CHECK action

with the current plan sequence. Figure 4.35 shows how the automaton was implemented in the 'safety check' action. The 'when' clause shown in the figure checks if a camera has

been turned on before completing three steps and if yes, it asserts that the system is unsafe and blocks the planner from proceeding further. The first 'forall' clause makes the state of a camera which has got upgrade command from 0 ('init' in figure 4.34) to 1. The second 'forall' clause changes the states from 1 to 4 (including). 4 is the state from which 'turn on' action can be taken. If 'turn on' action is taken before reaching state 4, the 'when' clause will make it unsafe. But the plan generated for this problem had meaningless sequence as

```
78: UPGRADE C13
79: SAFETY-CHECK
80: TURN-OFF C6
81: SAFETY-CHECK
82: TURN-ON C6
83: SAFETY-CHECK
84: TURN-OFF C6
85: SAFETY-CHECK
86: TURN-ON C13
87: SAFETY-CHECK
88: TURN-ON C6
89: SAFETY-CHECK
```

Figure 4.36 : Meaningless plan generated by the planner after LTL test condition

shown in figure 4.36. Here camera 6 is toggled off and on so that camera 13 can finish 3 steps after its upgrade action before turning on. So another dummy action called 'pause' was added. This action can be used to notify the system to wait for some time. This action doesn't have any effects other than forcing the 'safety check'action. Figure 4.37 shows the

```
(:action PAUSE
      :parameters ()
      :precondition (and (not(check)) (safe))
      :effect
              (and (check))
)
```

Figure 4.37 : PAUSE action for notifying the system to wait

'pause' action. After this action was introduced the planner generated meaningful sequences. Figure 4.38 shows the plan generated after LTL formula was incorporated. Three 'pause' actions came between 'upgrade camera 13' step and 'turn on camera 13' step and two between 'upgrade camera 12' step and 'turn on camera 12' step. It can also be seen that there are at least 3 steps between a pair of upgrade and turn on actions for a camera. Between the pairs of camera 1 and 3 there are 4 steps but it doesn't break any conditions.

0: TURN-OFF C1	22: TURN-OFF C5	44: TURN-ON C7	66: UPGRADE C12
1: SAFETY-CHECK	23: SAFETY-CHECK	45: SAFETY-CHECK	67: SAFETY-CHECK
2: UPGRADE C1	24: UPGRADE C5	46: TURN-OFF C9	68: TURN-ON C11
3: SAFETY-CHECK	25: SAFETY-CHECK	47: SAFETY-CHECK	69: SAFETY-CHECK
4: TURN-OFF C2	26: TURN-ON C4	48: UPGRADE C9	70: PAUSE
5: SAFETY-CHECK	27: SAFETY-CHECK	49: SAFETY-CHECK	71: SAFETY-CHECK
6: UPGRADE C2	28: TURN-OFF C6	50: TURN-ON C8	72: PAUSE
7: SAFETY-CHECK	29: SAFETY-CHECK	51: SAFETY-CHECK	73: SAFETY-CHECK
8: TURN-OFF C3	30: UPGRADE C6	52: TURN-OFF C10	74: TURN-ON C12
9: SAFETY-CHECK	31: SAFETY-CHECK	53: SAFETY-CHECK	75: SAFETY-CHECK
10: UPGRADE C3	32: TURN-ON C5	54: UPGRADE C10	76: TURN-OFF C13
11: SAFETY-CHECK	33: SAFETY-CHECK	55: SAFETY-CHECK	77: SAFETY-CHECK
12: TURN-ON C1	34: TURN-OFF C7	56: TURN-ON C9	78: UPGRADE C13
13: SAFETY-CHECK	35: SAFETY-CHECK	57: SAFETY-CHECK	79: SAFETY-CHECK
14: TURN-ON C2	36: UPGRADE C7	58: TURN-OFF C11	80: PAUSE
15: SAFETY-CHECK	37: SAFETY-CHECK	59: SAFETY-CHECK	81: SAFETY-CHECK
16: TURN-OFF C4	38: TURN-ON C6	60: UPGRADE C11	82: PAUSE
17: SAFETY-CHECK	39: SAFETY-CHECK	61: SAFETY-CHECK	83: SAFETY-CHECK
18: UPGRADE C4	40: TURN-OFF C8	62: TURN-ON C10	84: PAUSE
19: SAFETY-CHECK	41: SAFETY-CHECK	63: SAFETY-CHECK	85: SAFETY-CHECK
20: TURN-ON C3	42: UPGRADE C8	64: TURN-OFF C12	86: TURN-ON C13
21: SAFETY-CHECK	43: SAFETY-CHECK	65: SAFETY-CHECK	87: SAFETY-CHECK

Figure 4.38 : Plan generated after including the LTL formula

CHAPTER 5
SUMMARY AND FUTURE SCOPE OF WORK

5.1 Summary

Using the building use case, a prototype of an IoT Management System which handle clients in constrained environment was developed and it is autonomous and adaptive. A Leshan Server was created to handle the devices. The communication between the server and the clients are through CoAP. CoAP performs better than HTTP in low bandwidth scenarios. Device management is done through LwM2M which is a standard industry protocol and this is going to be adopted in large scale in near future.

The management application is built around openHAB 2.0. It pulls the states from the Leshan server using HTTP protocol through a custom binding created for the server, as both the application and the server can afford better bandwidth than the clients. From these fetched states PDDL files are generated if a goal needs to be achieved and planning is done using a planning service to which these files are passed as arguments. The plan received back will be executed from openHAB 2.0 through the custom binding created for the Leshan server and the server will in turn get it done on the devices through LwM2M call over CoAP.

Using the camera use cases, this system was made more manageable by incorporating policies. A dummy action called ‘safety check’ was modelled and the policies are converted to suitable conditions and is added into this action which is chosen by the planner after every action which can be mapped to real action.

By adding capability to include linear temporal logic formulas to an extent, the system is able to seamlessly handle the dynamic nature of the real world by considering the actions’ causality. LTL formulas are converted to Büchi Automaton [3] and it is implemented inside the ‘safety check’ action.

5.2 Future Work

IoT Systems must handle large number of connections because it is expected that the number of connected devices will be really large. Technologies like Leshan and openHAB 2.0 claim that they can handle this provided there is sufficient hardware. But it has not been tested. So this needs to be verified. Along with that it is essential to verify if the planners are capable to plan big sequence of actions within tolerable time limits. If not then it is

important to research on how to convert the problem to sub-problems and figure out the sub-goals and achieve the ultimate goal.

The performance of the system in more dynamic situations needs to verified. It needs to be tested with more complex use cases.

In real world scenarios it is not assured that all the actions are performed successfully. While executing the plan, some action may fail or sometimes the response may fail. These issues need to be addressed. If an action has failed, replanning is required. This is significant as the current world state may be different after executing a few actions. More research need to be done on replanning to handle this situation. A fault injection mechanism to test this approach need to be developed.

Even though a method to include Linear Temporal Logic formulas in planning has been developed, it doesn't fully capture the notion of time. Moreover a way to include all LTL operators in a general way in PDDL action needs to be developed. Conversion to Büchi Automaton [3] is now done using a software and is implemented manually in the dummy action. This needs to be automated as changing the LTL formula requires remodelling the domain file.

The policies presently need to be written in PDDL syntax in a file. It would be better if it could be written in a normal policy syntax. Easy Approach to Requirements Syntax (EARS) is a good method for this [10]. A translator which can take in EARS text and convert it to Logic formulas for PDDL is required. This needs work and maybe it can stretch into the area of Natural Language Processing.

5.3 Future Scope

As the device management protocol used is LwM2M which will be widely adopted in future, this system will be able to handle many future devices. This system can handle constrained devices and thus can be adopted for long term projects. As the system can make decisions on its own, maintenance from the developers will be minimal. When the system is enhanced to address the challenges in the previous section, it will be able to handle unexpected scenarios. All the mentioned point will in turn make the project cost effective.

REFERENCES

[1] Semantic Interoperability, Release 2.0, AIOTI WG03 – IoT Standardisation, 2015

[2] Swarup Mohalik, Mahesh Babu Jayaraman, Badrinath Ramamurthy and AnetaVulgarakis, “SOA-PE : A Service-oriented Architecture for Planning and Execution in Cyber-physical Systems”, in IEEE-IC -SSS 2015

[3] Fabio Patrizi, Nir Lipoveztky, Giuseppe De Giacomo and Hector Geffner, “Computing Infinite Plans for LTL Goals Using a Classical Planner”

[4] B.Nebel, “The FF Planning System: Fast Plan Generation Through Heuristic Search”, in Journal of Artificial Intelligence Research, Volume 14, 2001, Pages 253 - 302

[5] Dan Klein and Pieter Abbeel. University of Berkeley. Artificial Intelligence [Online].Available:https://www.edx.org/course/artificial-intelligence-uc-berkeleyx-cs188-1x

[6] Dr. Gerhard Wickler and Prof. Austin Tate. University of Edinburgh. Artificial Intelligence Planning [Online]. Available: https://www.youtube.com/playlist?list=PLwJ2VKmefmxpUJEGB1ff6yUZ5Zd7Gegn2

[7] Leshan. [Online]. Available: https://github.com/eclipse/leshan

[8] OpenHAB documents. [Online]. Available: http://docs.openhab.org/

[9] Eclipse Smarthome.[Online].Available: http://www.eclipse.org/smarthome/

[10] John Terzakis. Intel Corporation. EARS: The Easy Approach to Requirements Syntax.[Online].Available:https://www.iaria.org/conferences2013/filesICCGI13/ICCGI_2013_Tutorial_Terzakis.pdf

PROJECT DETAILS

Student Details			
Student Name	**Abin Mathew Abraham**		
Register Number	150948015	Section / Roll No	13
Email Address	abinmathewabraham@gmail.com	Phone No (M)	9447514133
Project Details			
Project Title	**Machine Intelligence based IoT Management System**		
Project Duration	11 months	Date of reportin g	22/06/2017
Organization Details			
Organization Name	**Ericsson R&D**		
Full postal address with pin code	11th floor, Citrine Building (WTC-4), Bagmane World Technology Center, Mahadevapura, ORR, Bangalore, India. 560048.		
Website address	www.ericsson.com		
Supervisor Details			
Supervisor Name	**Mr. Swarup Kumar Mohalik**		
Designation	Principal Engineer, Research		
Full contact address with pin code	11th floor, Citrine Building (WTC-4), Bagmane World Technology Center, Mahadevapura, ORR, Bangalore, India. 560048.		
Email address	swarup.kumar.mohalik@ericsson.com	Phone No (M)	+919945698671
Internal Guide Details			
Faculty Name	**Sucharitha Shetty**		
Full contact address with pin code	Dept of Computer Science & Engg, Manipal Institute of Technology, Manipal – 576 104 (Karnataka State), INDIA		
Email address	sucha.shetty@manipal.edu		

report

ORIGINALITY REPORT

%1	%1	%0	%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	archive.org Internet Source	%1
2	Miguel A. Wister, J. A. Hernandez-Nolasco, Pablo Pancardo, Francisco D. Acosta, Antonio Jara. "Emergency Population Warning about Floods by Social Media", 2016 10th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2016 Publication	<%1
3	www.buenastareas.com Internet Source	<%1
4	www1.spms.ntu.edu.sg Internet Source	<%1

EXCLUDE QUOTES	ON	EXCLUDE MATCHES	< 10 WORDS
EXCLUDE BIBLIOGRAPHY	ON		