INSTITUT POLYTECHNIQUE DE PARIS

Final Project using FIT/IOT Lab Wireless

Tianchi Yu

January 8, 2021

1. OVERVIEW

Recent years, with the development of IoT technique and the industrial automation, using sensors to capture the environment and connecting things with network are common and popular ways to make the industrial processes safe and efficient, to make our life convenient. Abundant IoT applications are implemented in various kinds of scenarios, here in this project, I propose one IoT application working on the safety of sensitive cargo during transportation.

1.A. DEFINITION AND INTRODUCTION OF THE ASSUMED IOT APPLICATION

Food and medical supplies often require temperature and humidity control during storage and transportation. In order to facilitate control, logistics companies use sealed containers with environmental controls. These containers are equipped with sensors that will send status reports to the central network to monitor these containers to control humidity and temperature. If there is a problem with the environment inside the container, if a leak in the sealed container is found, the sensor will immediately send an alarm to the central network to take appropriate measures. The use of the Internet of Things can reduce the deterioration of sensitive goods and prevent pollution.

It's a very important application according to the scope of using, and also a very prospective application for different levels of security. For example, because insulin drugs have higher requirements for the delivery environment than ordinary drugs, UNOPENED insulin is best stored inside the fridge [2° to 8°Celcius (36° to 46°Fahrenheit)]. Heat makes insulin break down and will not work well to lower your blood sugar. Also, if insulin is frozen(lower than 0°Celcius), do not use even after thawing. Because freezing temperature will break down the insulin and then it will not work well to lower your blood sugar. Then the entire process needs to be accurately controlled at 2-8°C, and real-time temperature monitoring is critical. Insulin is also very sensitive to sunlight, indoor lights.[1]

Another example is the COVID-19 VACCINE by Pfizer. The vaccine transport, storage and continuous temperature monitoring have extreme constraints. For example, there are three options for storage: ultra-low-temperature freezers, which are commercially available and can extend shelf life for up to six months; the Pfizer thermal shippers, in which doses will arrive, that can be used as temporary storage units by refilling with dry ice every five days for up to 30 days of storage; refrigeration units that are commonly available in hospitals. The vaccine can be stored for five days at refrigerated 2-8°C conditions.[2]

To be more precise and describable, we set the scenario as a medical drug delivery vehicle in motion, where the drug need to be maintained at a strict interval of temperature and several other conditions, which we will talk about later in the architecture. The IoT application is named *Medicine-Guardian*.

2. ARCHITECTURE OF THE SYSTEM

2.A. THE FUNCTIONS IMPLEMENTED

Here I provide several automation functions of *Medicine-Guardian*, which implement three different scenarios for several medicines on the delivery vehicle:

1) Collecting light information and calling the alarm when the light is on. Once we get the signal of alarm for light, switch the states of lights(turn off the light).

Corresponding scenario: Supposing that there is one drug that need one strict light constraint of darkness, it cannot suffer from lightness.

2) Collecting temperature information information. If the temperature is >8°C or <2°C, turn on the heater(temperature controller) until the temperature arrives 5. However, if there is the temperature is extremely low or high (<0°C or > 30°C), or no heater could be used, send alarm to the monitor. To make this scenario more interesting, we will add more details for the heater at the part of implementation.

Corresponding scenario: Supposing that there is one drug that the best storage temperature is between 2 and 8. If the temperature is lower than 0 or higher than 30, the drug will be useless.

3) Collecting accelerator information, and sending alarm if accelerator sensor has a great value or it makes a great change in short time period.

Corresponding scenario: Supposing that there is one drug that cannot accept the shaking, we need to avoid the possible violent variation of velocity.

2.B. SERVERS AND CLIENTS INSTALLED

In this project, we set two kinds of clients: on the front-end and on the sensor node. To realise the communication, we also need one server on the sensor node. In the experiment, we reserve some M3 nodes on the Grenoble site, and set them up with flashing two firmwares nodes. Also, to compare the difference between HTTP and COAP, we can access nodes over HTTP over IPv6 from the SSH frontend by using a Contiki tunslip6 bridge and launch HTTP server or launch requests on a CoAP Contiki Erbium server implementation.

3. DEMO OF MEDICINE-GUARDIAN

To give a full view of the demo, we offer the code of the CoAP server/client and the HTTP server/client and other files needed.

3..1. ADD EXPERIMENTS

We build a test bed with 3 sensor nodes with Architecture m3 (at86rf231) of Grenoble site, named "comasic4";

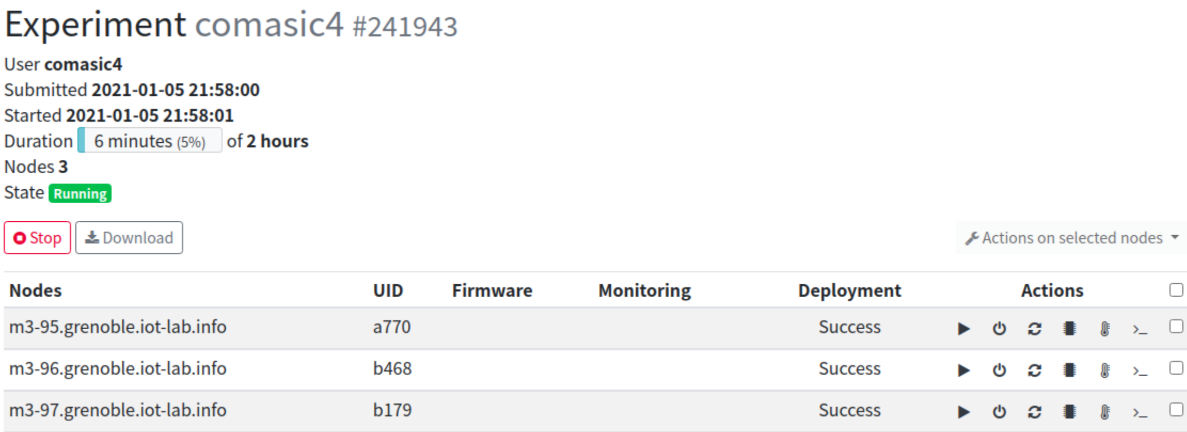

Figure 3.1: Test Bed

Basic processes:

Compile firmwares on SSH front end;

```
comasic4@grenoble:~/iot-lab/parts/contiki/examples/iotlab/04-er-rest-example$ iotlab-node --update ~/iot-lab/parts/contiki/examples/iotlab/04-er-rest-example/er-example-server.iotlab-m3 -e grenoble,m3,95
{
    "0": [
        "m3-96.grenoble.iot-lab.info",
        "m3-97.grenoble.iot-lab.info"
    ]
}
```

Figure 3.2: Example: flash CoAP firmare

Choose an available IPv6 prefix for the site you are experimenting on. For example in Grenoble testbed : we choose 2001:660:5307:3110::/64

```
comasic4@grenoble:~/iot-lab/parts/contiki/examples/ipv6/rpl-border-router$ sudo tunslip6.py -v2 -L -a m3-95 -p 20000 2001:660:5307:3110::1/64
Switch from 'root' to 'comasic4'
Calling tunslip:
        'tunslip6 [u'-v2', '-a', 'm3-95', '-L', '-p', '20000', '2001:660:5307:3110::1/64']'
slip connected to ``172.16.10.95:20000''

22:20:09 opened tun device ``/dev/tun1''
0000.000 ifconfig tun1 inet `hostname` mtu 1500 up
0000.008 ifconfig tun1 add 2001:660:5307:3110::1/64
0000.012 ifconfig tun1 add fe80::660:5307:3110:1/64
0000.017 ifconfig tun1

tun1: flags=4305<UP,POINTOPOINT,RUNNING,NOARP,MULTICAST>  mtu 1500
        inet 192.168.1.5  netmask 255.255.255.255  destination 192.168.1.5
        inet6 fe80::ea73:7ab8:6198:5765  prefixlen 64  scopeid 0x20<link>
        inet6 2001:660:5307:3110::1  prefixlen 64  scopeid 0x0<global>
        inet6 fe80::660:5307:3110:1  prefixlen 64  scopeid 0x20<link>
        unspec 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00  txqueuelen 500  (UNSPEC)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

0256.793
0256.793
0257.793 Platform starting in 1...
0257.793 GO!
0257.795 [in clock_init() DEBUG] Starting systick timer at 100Hz
0257.796
0257.797 *** Address:2001:660:5307:3110::1 => 2001:0660:5307:3110
0257.840  Starting 'Border router process' 'Web server'Got configuration message of type P
0257.840 Setting prefix 2001:660:5307:3110::
0258.795 Server IPv6 addresses:
0258.797  2001:660:5307:3110::a770
0258.797  fe80::a770
```

Figure 3.3: Get BR address

Choose one node in the nodes list to implement the Border Router (BR) node - here we choose node m3-95

Start tunslip6 on the SSH frontend
Deploy the Border Router (BR) node on selected node using the CLI tool iotlab-node.

Then we need to choose one node and wait for deploying: build HTTP server - simple IPv6 node **or** IoT-LAB version of CoAP Erbium server;
FOR HTTP
Deploy one HTTP server node (here node 96).
FOR COAP

```
comasic4@grenoble:~$ iotlab-node --update ~/iot-lab/parts/contiki/examples/ipv6/rpl-border-router/border-router.iotlab-m3 -l grenoble,m3,95
{
    "0": [
        "m3-95.grenoble.iot-lab.info"
    ]
}
```

Figure 3.4: Deploy Border Router

Choose one CoAP server node(here node 96).

For the server/client on sensor node, we need deploy the server and client separately on two nodes.

Note that, every time we change the code or add resoureces, we need to compile for iotlab-m3 target and update the node.

For now, we have a basic environment for the experiment. Then we will implement the three functions(mainly realized by CoAP).

3.A. Implementation of CoAP Clients

3.A.1. CoAP clients on the Front-end

For this experiment, we only need two nodes actually, one for the border router and another for the CoAP server. To make the whole test consistent, we still choose three nodes, while only two of them are used. We want to implement the first function by this method, which means we want to collect the light information by the front-end and test the alarm when the light is on and activate the command to turn the light off.

First Scenario

At first, we need to deploy the Border Router node and CoAP server node. We choose m3-95 and m3-96 separately. Then use "aiocoap-client coap://[2001:660:5307:3110::b468]:5683/XXX/XXX" command by the client(front-end), we can get the information from server.

To realise the first function implemented, we need to do the following steps:

1) Collecting light information:

```
extern int light_info;
extern int pressure_info;
extern int temper_info;
extern int useless;
extern int heater_info;
extern int acc_x, acc_y, acc_z;
~
~
~
~
```

Figure 3.5: Some extern variables

First, make sure that we have the "light" resource, and extern it in the CoAP server code, and activate the light sensor. We add one global variable in the extern_variable.h to record the value of light by the server.

Because the real state of light is always off(the value of light_info is always 0), we need to simulate the situation when the light becomes on. So I changed the code of res_light.c to control the value of light. By using a random integer, we set the light on when the integer is a multiple of 5. So we can collect the light information and get the state of light.

```
uint16_t light = light_sensor.value(0) / LIGHT_SENSOR_VALUE_SCALE;

/* Add test model:
 * Build a virtual light signal to simulate the light is on and off
 * Here we use a random number named flag, when the number is multiple
 * of 5, we set the light on.
 * */
printf("The real light value: %d\n",light);

printf("--------Test for the model---------\n");
int flag = rand() % 1000;
// printf("flag is %d\n",flag);
if(flag%5!=0 && light_info == 0){
        light = 0;
        printf("LIGHT VALUE : %d\n",light);
        printf("The light is off, GOOD STATE!\n");
} else {
        light = 1;
        printf("LIGHT VALUE : %d\n",light);
        printf("The light is on, we need to turn it off!\n");
}
light_info = light;
```

Figure 3.6: Light sensor

As the code above in the res_light, we simulate the signal of light. Because in the real case, the light could be always turned off, we couldn't see the difference. The signal of light stands for the state of light. Notice that, once the light is turned on, it has to be turned off by calling the res_switch resource. It means if we don't use another resource to change the state, the value of light_info will always be 1;

2) Testing the alarm:

When the light is on, we need to test the alarm and we have a clear information that there is an alarm for light! The command is "aiocoap-client coap://[2001:660:5307:3110::b468]:5683 /my_res/new_alarm"

3) Turning off the light

Once we get the information of alarm, the workers will know the light of the package is turned on and unacceptable. Then the next step is to turn the light off manually. Here, we add one resource res_switch, it could turn off the light when we ask it by the command.

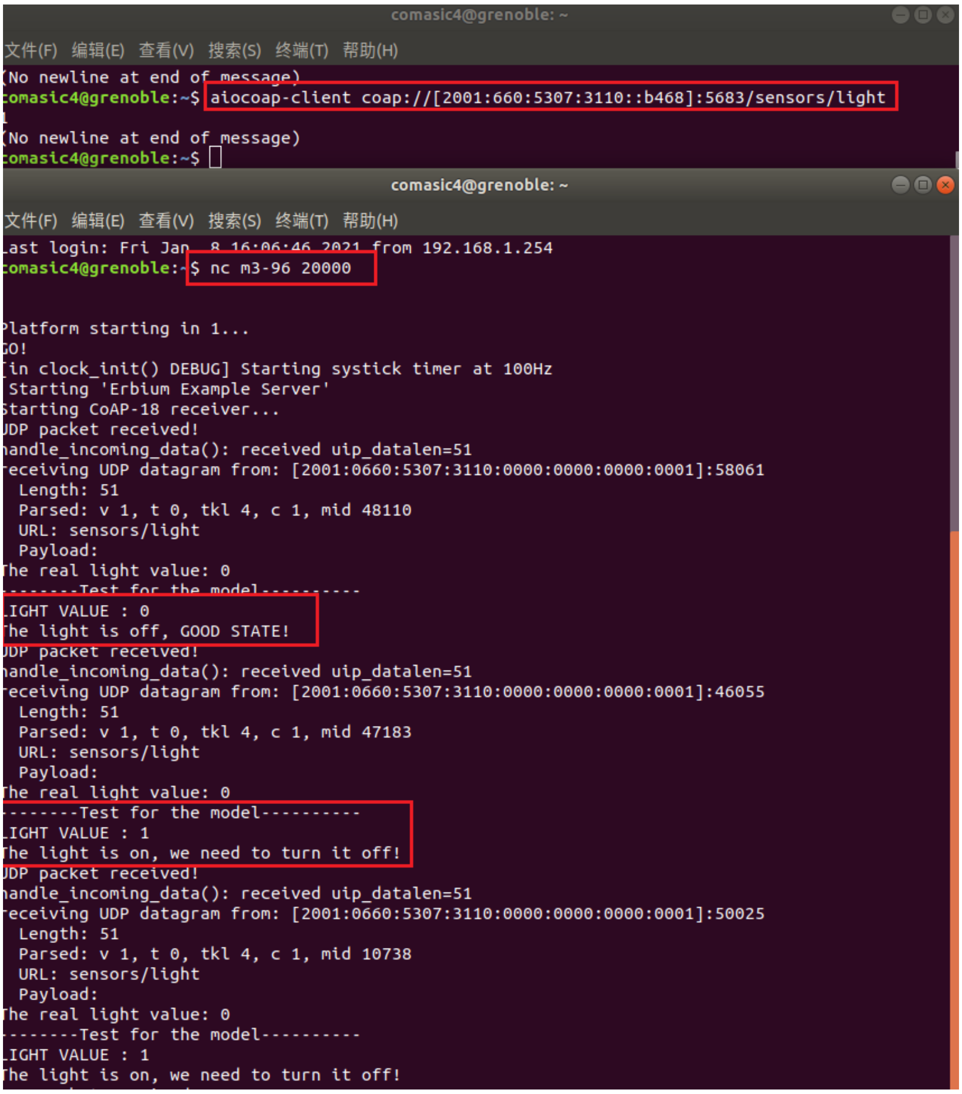

Figure 3.7: Collecting light information

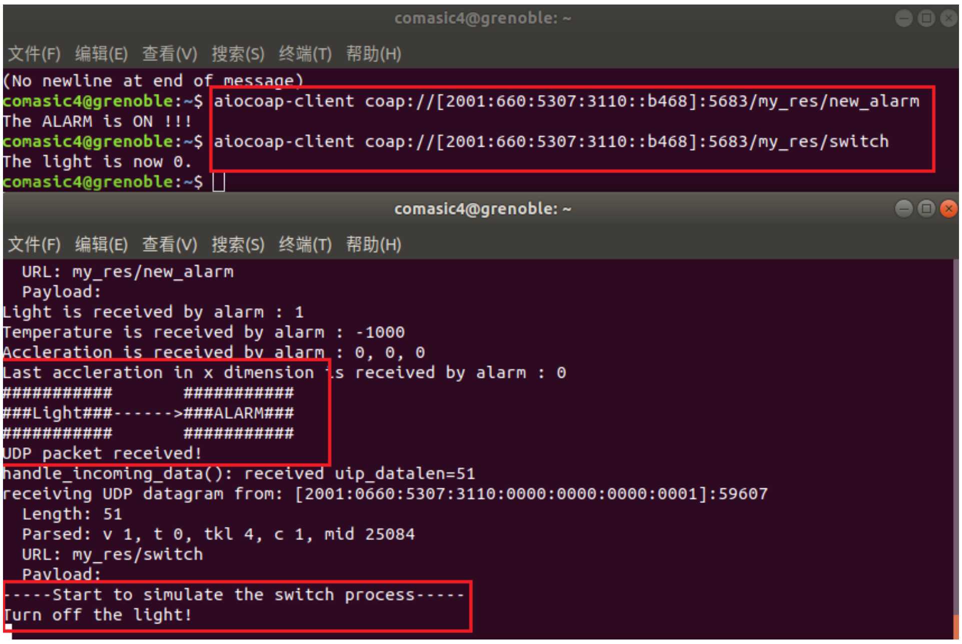

Figure 3.8: Testing the alarm and turning off the light

3.A.2. CoAP clients on the sensor node

To make it clearly, we always set the m3-96 as the CoAP server and deploy the client on the node of m3-97.

Figure 3.9: Deploy client and server CoAP

First Scenario

To explain: We also implement this scenario by the CoAP clients method. We only display the result for this scenario here.

Second Scenario

The heater in this experiment is not only used for heat the drug. Actually, it stands for temperature. To simplify the situation, the heater can adjust the temperature directly to 5°C if the temperature is between 0 and 2 or between 8 and 20. And the heater can only be used for once, which makes sense. Because if the temperature keep going wrong even with

```
The real light value: 0
--------Test for the model----------
LIGHT VALUE : 0
The light is off, GOOD STATE!
UDP packet received!
handle_incoming_data(): received uip_datalen=37
receiving UDP datagram from: [2001:0660:5307:3110:0000:0000:0000:b179]:5683
  Length: 37
  Parsed: v 1, t 0, tkl 0, c 1, mid 52260
  URL: my_res/new_alarm
  Payload: Alarm - Toggle!
Light is received by alarm : 0
Temperature is received by alarm : -1000
Accleration is received by alarm : 0, 0, 0
Last accleration in x dimension is received by alarm : 0
UDP packet received!
handle_incoming_data(): received uip_datalen=35
receiving UDP datagram from: [2001:0660:5307:3110:0000:0000:0000:b179]:5683
  Length: 35
  Parsed: v 1, t 0, tkl 0, c 1, mid 52261
  URL: my_res/switch
  Payload: Switch - Toggle!
-----Start to simulate the switch process-----
No need to switch !
```

(a) pic1.good state of light

```
UDP packet received!
handle_incoming_data(): received uip_datalen=34
receiving UDP datagram from: [2001:0660:5307:3110:0000:0000:0000:b179]:5683
  Length: 34
  Parsed: v 1, t 0, tkl 0, c 1, mid 52262
  URL: sensors/light
  Payload: Light - Toggle!
The real light value: 0
--------Test for the model----------
LIGHT VALUE : 1
The light is on, we need to turn it off!
UDP packet received!
handle_incoming_data(): received uip_datalen=37
receiving UDP datagram from: [2001:0660:5307:3110:0000:0000:0000:b179]:5683
  Length: 37
  Parsed: v 1, t 0, tkl 0, c 1, mid 52263
  URL: my_res/new_alarm
  Payload: Alarm - Toggle!
Light is received by alarm : 1
Temperature is received by alarm : -1000
Accleration is received by alarm : 0, 0, 0
Last accleration in x dimension is received by alarm : 0
###########       ###########
###Light###------>###ALARM###
###########       ###########
UDP packet received!
handle_incoming_data(): received uip_datalen=35
receiving UDP datagram from: [2001:0660:5307:3110:0000:0000:0000:b179]:5683
  Length: 35
  Parsed: v 1, t 0, tkl 0, c 1, mid 52264
  URL: my_res/switch
  Payload: Switch - Toggle!
-----Start to simulate the switch process-----
Turn off the light!
```

(b) pic2.bad state of light

Figure 3.10: Light function implementation

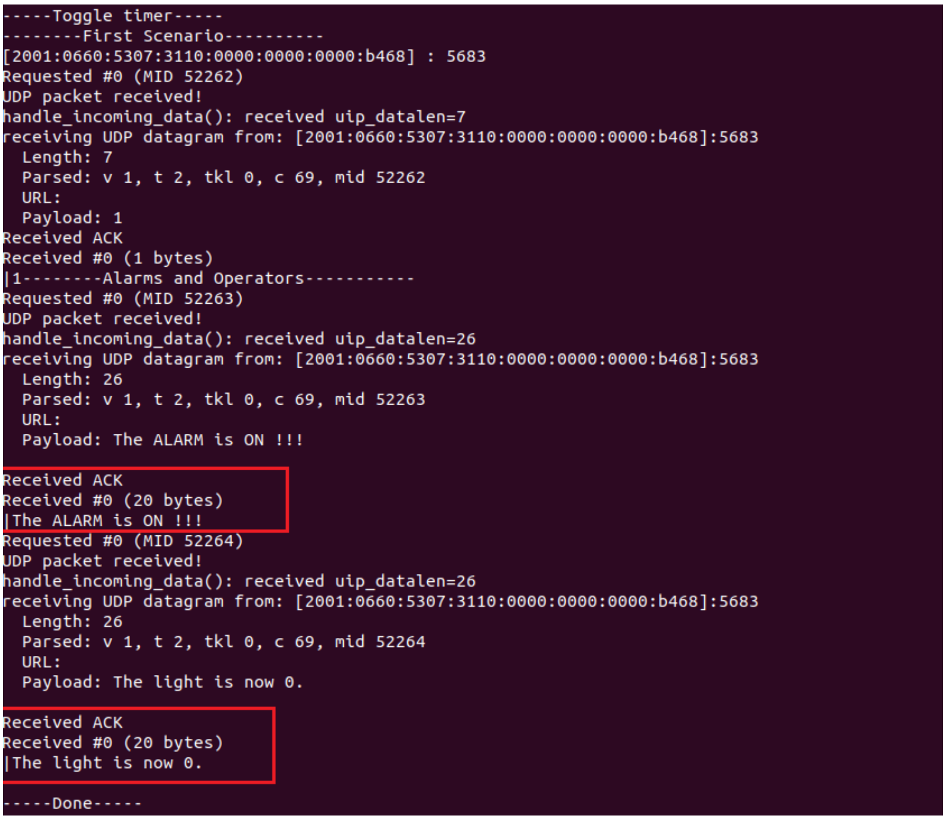

Figure 3.11: Received by client

the heater during the transportation, there is no way to add one more heater to control the temperature.

If there is the temperature is extremely low or high (<0°C or > 30°C), send alarm to the monitor, and the drug will be useless.

1) Collecting temperature information information:

Firstly, we need to add the temper_info as the global variable to record the temperature information. Note that, set the initial value of temper_info as -1000 is to sign the initial state, which will not cause the alarm on.

Notice that in grenoble platform, the sensors don't have the capacity of collect temperature. Here we offer two options, one is using another available sensor who can capture one feature(for example, pressure) as the value of temperature, another option is creating self-resource and create the feature of temperature by ourselves. The first method will take the similar process as the first scenario, then we decide to use the second method.

We create the "new_temperature" resource, which will create the value of temperature according to some logic defined.

1. The initial temperature falls down into the 2-8.

2. Set a possible noise by the random integer and some hash methods, to decide the variation of the temperature.

3. For the following collection, the temperature will inherit the previous value + some noise.

2) Using Heater:

The client node will decide whether using heater every time it collects temperature information. When the temperature is out of normal interval, the heater will work and adjust the temperature. However when the temperature keeps going worse or it becomes extremely high or low, the heater will be useless, also the drug.
As we have discussed, the heater resource is created as following:

3) Testing the Alarm:

The client node will test the alarm every time it collects temperature information. If the heater could control the situation, it keeps calm. Instead, the alarm will be on.

```
static void
res_get_handler(void *request, void *response, uint8_t *buffer, uint16_t preferred_size, int32
_t *offset)
{
  printf("-----Start to simulate the temperature sensors-----\n");

  const char *len = NULL;
  int temperature_old = temper_info;
  int intern_temperature = temper_info;
  if(temper_info == -1000){
          intern_temperature = rand() % 6 + 2;
  }else{
          intern_temperature = temperature_old;
  }

  int flag = rand() % 50;
  if(flag % 3 ==0){
          intern_temperature += 5;
  }else if(flag % 10==0){
          intern_temperature += 30;
  }else if(flag % 5==0){
          intern_temperature -= 4;
  }else{
  }

  temper_info = intern_temperature;

  printf("Temperature is captured: %d\n",temper_info);

  if(useless ==1 ){
          printf("No matter what is the temperature, the drug is already useless !\n");
  }else{
          if(intern_temperature<0){
                  printf("The drug should be useless ! (The alarm should be on)\n");
                  useless = 1;
          }else if (intern_temperature <2){
                  printf("The temperature is too low ! Heater should be turned on !\n");
          }else if (intern_temperature >= 2 && intern_temperature <= 8){
                  printf("The temperature is normal !");
          }else if (intern_temperature >30){
                  printf("The drug should be use less ! (The alarm should be on)\n");
                  useless = 1;
          }else{
                  printf("The temperature is too high ! Heater should be turned off\n");
          }
  }
```

Figure 3.12: Self-defined temperature sensor resource

```
static void
res_get_handler(void *request, void *response, uint8_t *buffer, uint16_t preferred_size, int32
_t *offset)
{
  printf("-----Start to simulate the heater process-----\n");
  const char *len = NULL;
  if((temper_info >=0 && temper_info <2) || (temper_info <= 20 && temper_info > 8)){
          if(heater_info == 0){
                  heater_info = 1;
                  printf("$$$$$$$Heater is turned on now.$$$$$$$\n");
                  temper_info = 5 ;
                  printf("Temperature is justed by heater to: %d\n",temper_info);
          }else{
                  printf("Heater is already turned on ! It cannot change the temperature now.\
n");
          }
  }else{
          printf("Heater is not useful for this situation !\n Watch the alarm !\n");
  }
```

Figure 3.13: Heater Resource

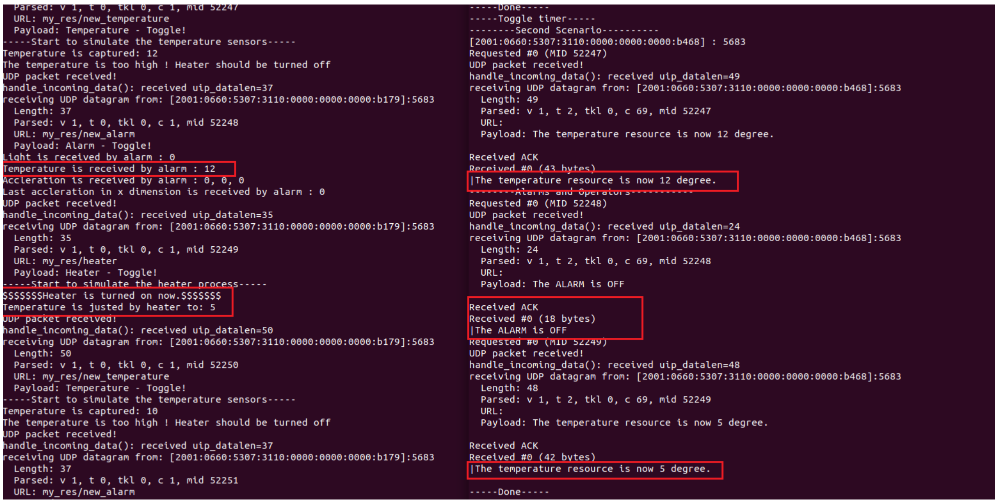

Figure 3.14: Heater works for second scenario

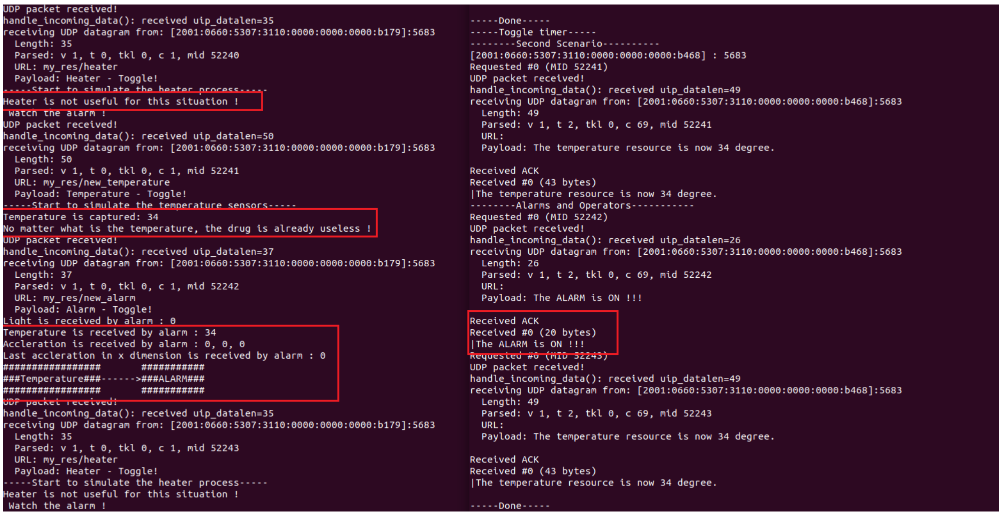

Figure 3.15: Extreme temperature in the second scenario

Third Scenario

1) Collecting acceleration information:

By the accelerometer sensor, we can collect the acceleration in three dimension. To simplify, we choose the value of x dimension as the constraint.

Also we need add several global variables in extern_var.h.

2) Testing the Alarm:

When the value of acc_x is larger than 90 or smaller than 90, or the difference of acceleration between two continuous collection is larger than 8, the alarm will be on to remind the driver be calmer.

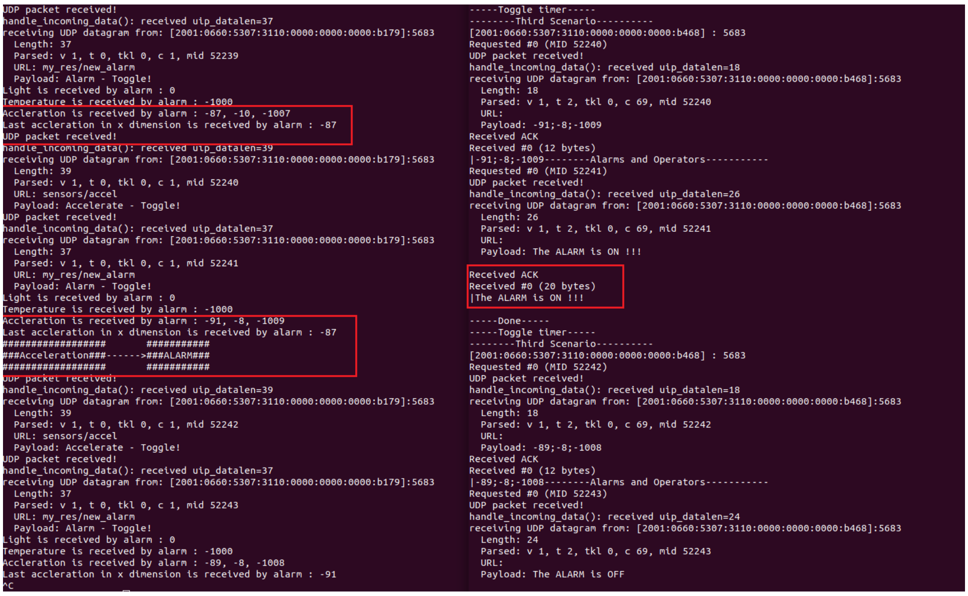

Figure 3.16: Implementation of the third scenario

3.A.3. Adding CoAP resources

In the above description, we have already give some displays of new resource. To conclude, we have added the following resources:

1. res_heater: Temperature Controller in the second scenario

2. res_switch: Light switch in the first scenario

3. res_new_alarm: For all three scenario

4. res_new_temperature: Simulating the temperature sensor

Note that, after add the resources, we also need to activate them and add commands to calling them. Here we explain one of the most important resource ALARM.

In the alarm resource, all extern global variables are called and calculated in order to define the signal of alarm. In all these three scenarios, no matter which one constraint is not satisfied, the alarm should be the ON state.

```
static void
res_get_handler(void *request, void *response, uint8_t *buffer, uint16_t preferred_size, int32
_t *offset)
{
  printf("Light is received by alarm : %d\n",light_info);
  printf("Temperature is received by alarm : %d\n",temper_info);
  printf("Accleration is received by alarm : %d, %d, %d\n",acc_x, acc_y, acc_z);
  const char *len = NULL;
  int alarm_info = 0;

  if(acc_x_old ==0){
          acc_x_old = acc_x;
  }
  printf("Last accleration in x dimension is received by alarm : %d\n",acc_x_old);

  if(light_info ==1){
          printf("###########       ###########\n");
          printf("###Light###------>###ALARM###\n");
          printf("###########       ###########\n");
          alarm_info = 1;
  }
  if(temper_info >30 || (temper_info < 0 || temper_info == -1000)){
          printf("#################       ###########\n");
          printf("###Temperature###------>###ALARM###\n");
          printf("#################       ###########\n");
          alarm_info = 1;
  }

  int dif = acc_x - acc_x_old;

  if(acc_x > 90 || acc_x < -90 || dif >8 || dif < -8){
          printf("##################       ###########\n");
          printf("###Acceleration###------>###ALARM###\n");
          printf("##################       ###########\n");
          alarm_info = 1;
  }

  if(alarm_info == 0){
          REST.set_header_content_type(response, REST.type.TEXT_PLAIN); /* text/plain is the d
efault, hence this option could be omitted. */
          snprintf((char*)buffer, REST_MAX_CHUNK_SIZE,"The ALARM is OFF \n");
  }else{
          REST.set_header_content_type(response, REST.type.TEXT_PLAIN); /* text/plain is the d
efault, hence this option could be omitted. */
          snprintf((char*)buffer, REST_MAX_CHUNK_SIZE,"The ALARM is ON !!!\n");

  }
```

Figure 3.17: Example: New Alarm Resource

3.B. Simple Implementation of HTTP Clients

To implement a HTTP client, we need to add one more thread that start the client process in the code of http_sever.c. We need to set the IP address of the server to collect the message.

Then because it is difficult to add resources, we only add the light sensor into the code and try to build the communication between the http client and http server and collect the light information. Here is the implementation.

4. Performance evaluation: Comparison of CoAP and HTTP

The performance of CoAP is very good and fluent, it is easy to implement basic functions using less data. Here we would like to discuss more about the comparison of CoAP and

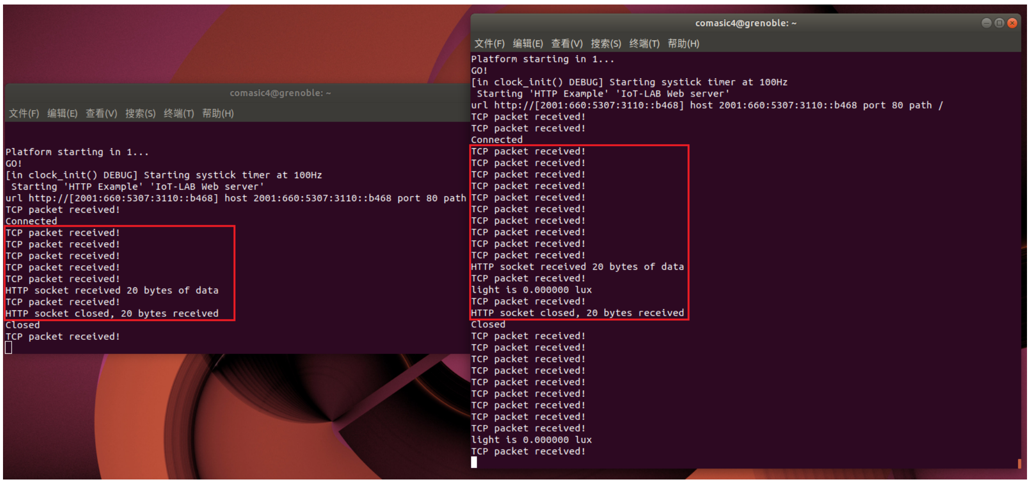

Figure 3.18: Example: HTTP Server/Client nodes

HTTP.

At first, to compare CoAP and HTTP, we need understand the code.

1) The difference of the protocol level is that CoAP uses UDP and HTTP uses TCP. We can easily get this difference by the code or the output of the server.

```
#if UIP_IPV6_MULTICAST
  process:
#endif

  while(1) {
    switch(*uip_next_hdr){
#if UIP_TCP
    case UIP_PROTO_TCP:
      /* TCP, for both IPv4 and IPv6 */
      printf("TCP packet received!\n");
      goto tcp_input;
#endif /* UIP_TCP */
#if UIP_UDP
    case UIP_PROTO_UDP:
      /* UDP, for both IPv4 and IPv6 */
      printf("UDP packet received!\n");
      goto udp_input;
#endif /* UIP_UDP */
    case UIP_PROTO_ICMP6:
      /* ICMPv6 */
      goto icmp6_input;
    case UIP_PROTO_HBHO:
      PRINTF("Processing hbh header\n");
      /* Hop by hop option header */
```

Figure 4.1: UDP & TCP

2) Both architectures support the combination of server and client on the same node. We have verify this by the above implementations.

3) Actually HTTP has the Synchronous Communication. We can tell this by watching the

output of server node and client node. TCP protocol follows the way of synchronous communication.

4) HTTP is more complex and has more overhead. To verify this, we need to balance the packet transmission. For HTTP, there are some other parts of other transmission which need to be removed, but the CoAP should send only light sensor information to keep same as HTTP(in this case).

Note that, we should deploy the CoAP server/client to both nodes this time.

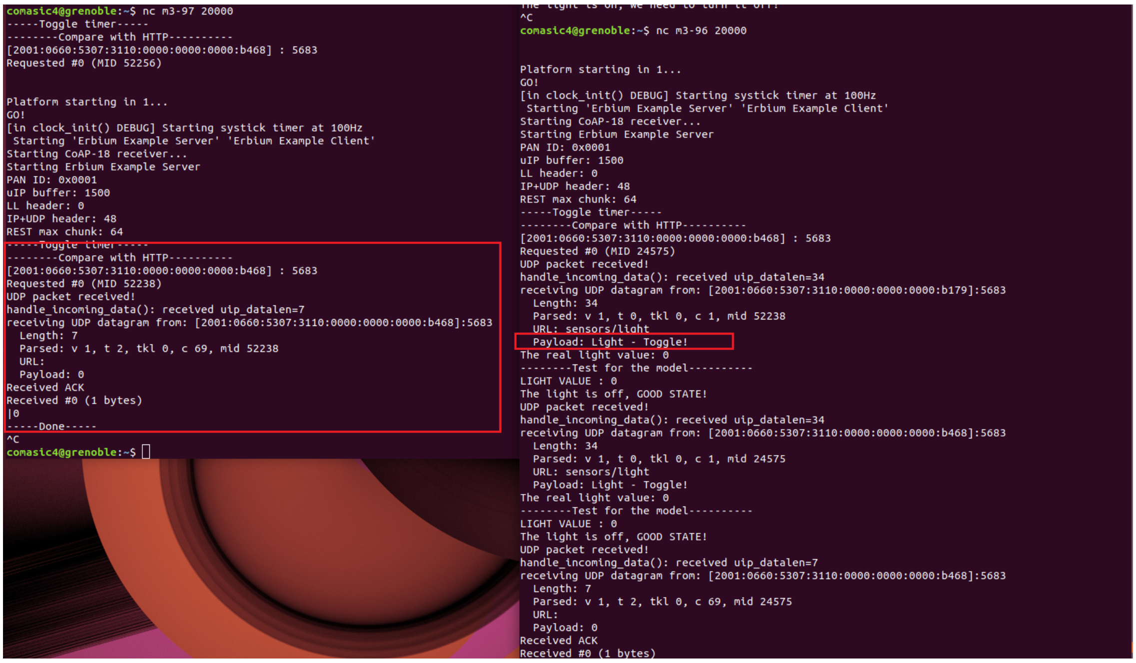

Figure 4.2: Example: Implementation with only collecting light information

5. Conclusion

For now, the main three functions have been realised by our architecture. Although there are still some details need to be discussed and practiced, I believe this kind of IoT application *Medicine-Guardian* will make big differences in the future !

6. Bibliography

[1] https://consumermedsafety.org/tools-and-resources/insulin-safety-center/storage-of-insulin

[2] https://www.pfizer.com/news/hot-topics/covid_19_vaccine_u_s_distribution_fact_sheet