ÉCOLE POLYTECHNIQUE DE LOUVAIN

UCLouvain

LINGI2146 - Mobile & embedded computing - 2021

Project

Authors

Merlin	Camberlin	0944-17-00
Arthur	Pisvin	3711-17-00
Zoé	Schoofs	3502-17-00

May 13, 2021

https://github.com/mcamberlin/LINGI2146-InternetOfThingsDesign

Professor:
Ramin Sadre

Teaching assistant:
Sébastien Strebelle

1 Communication protocol and message format

The communication protocol implemented is based on a query-response model similar to *CoAP*. When a device want to know information about another, it sends a query to it and waits for its reply.

Our connected ecosystem exchanges messages between motes and server via datagrams on top UDP/IPv6. The communication between motes and server is made through datagrams of the same format and the same length. The format of this message is illustrated in the FIGURE 1.

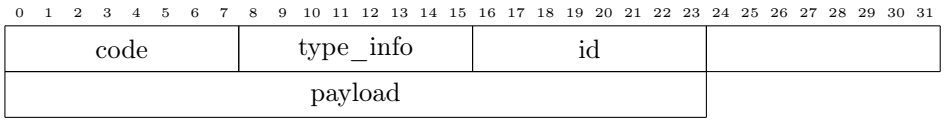

Figure 1: Message format

In the implementation, the message is described via the below C-structure named `datagram`.

```
typedef struct
{
    u_int8_t code ;
    u_int8_t type_info;
    u_int8_t id;
    char payload[4];
} datagram;

```

The message and the structure are 7 bytes long. The structure contains four attributes.

1. The first one is the `code`, which is a `u_int8_t` that is either 1 or 2, to indicate whether the message is query or a response. To ease the implementation, 8 bits are reserved even if only 2 of them are in practice used. This choice is appropriate since the message are only 7 bytes long. Moreover, letting unused bits allows further extensions of the protocol.
2. The second attribute is also a `u_int8_t` and is named `type_info`. This attribute characterizes the type of data communicated. It allows the server to correctly interpret the content of the payload and the type of information it contains. Is it a temperature data or is it a proximity measure ? This attribute makes the distinction. The mapping between the code and the corresponding content in the payload is indicated in the FIGURE 2. Currently, as four devices are implemented, its value is in the range between 1 and 4. However, our protocol allows future extension where more kind of different connected devices would join the ecosystem. With 8 bits reserved, the ecosystem is able to contain up to 2^8 different types of connected devices.

`type_info`	description
1	proximity data
2	lamp state
3	temperature data
4	door state

Figure 2: Mapping between `type_info` and the meaning of the payload

3. The third attribute is a `u_int8_t` and is named `id`. This attribute uniquely identifies a sensor of a certain type. This makes the distinction between several sensors of the same thing. For example, it allows to distinguish several temperature sensors located in different places. In the current ecosystem, as there is only one device by kind of IoT device. This id is always set to 1. But our protocol would allow future extension that could distinguish up to 2^8 different devices of the same IoT king.
4. The last attribute is a list of 4 characters and is named `payload`. As the name suggests, this attribute represents the payload of the datagram. It is where the content of replies are put. It is only 4 bytes long but it is enough since it will only contains data in the form of an integer (for measure, temperature or state).

2 Supported IoT devices and their capabilities

The ecosystem simulated and illustrated on FIGURE 3 contains 4 IoT devices. Their node numbers in the figure is in the range 2 to 5. The node with node 1 corresponds to the border router and is responsible for the communication between the ecosystem and the server. The latter is located outside the network.

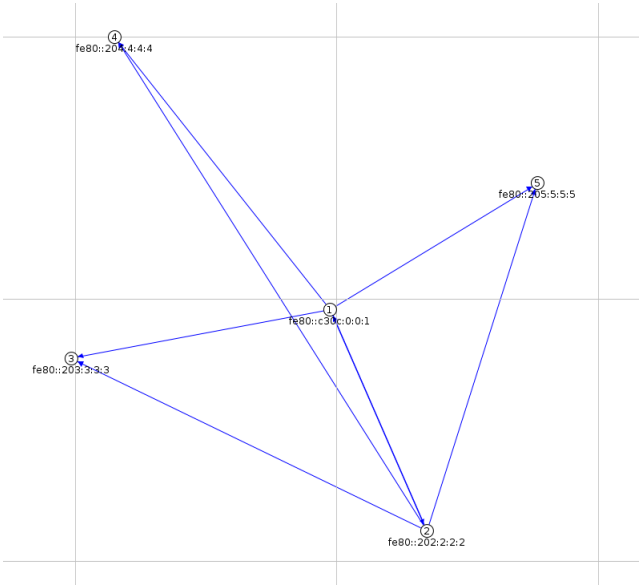

Figure 3: Network simulated

In the ecosystem, four IoT devices have been implemented.

- **Proximity sensor** (node:2) : its behavior is described in the file `proximity-sensor.c`. Basically, this sensor returns a measures in centimeter of the distance between it and an obstacle. To simulate it, this sensor returns a random measure between 17 and 24 centimeters. When it receives a query asking for the proximity, it places the proximity measure in the payload of the answered datagram.
- **Connected lamp** (node:3) : its behavior is described in the file `lamp.c`. Basically, this lamp maintains a single state that correspond to whether or not the lamp is switch on. When the lamp receives a query asking to change its state (On/Off), it will update it by the opposite of its actual state and returns the new state. The payload is then a int, either 0 (Off) or 1 (On).
- **Temperature sensor** (node:4) : its behavior is described in the file `temperature-sensor.c`. Basically, this sensor returns a temperature measure in Celsius degree. To simulate it, this sensor returns a random temperature between 17 and 24 [°C]. When it receives a query asking for the temperature, it places the temperature measure in the payload of the answered datagram.
- **Connected door lock** (node:5) : its behavior is described in the file `door.c`. Basically, this lamp maintains a single state that correspond to whether or not the door is opened. When the door lock receives a query asking to change its state (opened/closed), it will update it by the opposite of its actual state and returns the new state. The payload is then a int, either 0 (closed) or 1 (opened).

3 Control center program and its interfaces

The file named `server.c` is responsible for the management of all the communications between motes and server. The terminal is used as an interface between the user and the server. This section aims to explain how the user can interact with the network of motes.

During the execution of `server.c`, different global static variables are created and two threads are executed. The global static variables hold all the outputs of the different sensors or actuators.

The first one named *server* listens to the motes and permanently executes the function `server` responsible for the exchange of messages.

If a new message arrived, it decodes it (convert the data received into a `datagram` structure). The pair of attributes `type_info` and `ID` uniquely identifies which mote sent the `datagram`. Based on that identification,

the server extracts the payload from the datagram and puts it in the corresponding global variable. `Mutexes` are used to protect the critical section that represent the global variables since they are shared with the other thread named *input*.

If the server doesn't receive a datagram from the network, it checks if the user want to do something. He can either ask a distance to the proximity sensor, change the state of the lamp, ask a temperature to the temperature sensor, change the state of the door lock or stop the program.

The communication between the user and the program is managed by the thread named *input*. It acts like a command-line interface. The user types an integer between 1 and 5 in the terminal for different actions. All these actions are specified in the beginning of the terminal as we can see on FIGURE 4.

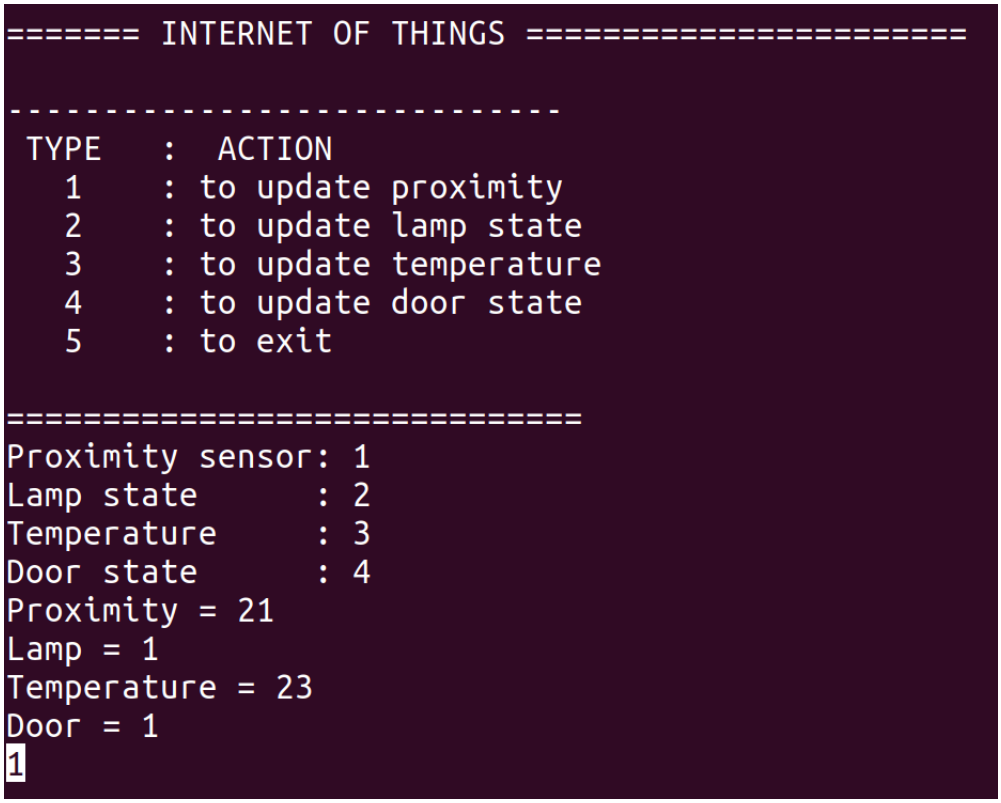

Figure 4: Command-line interface

4 How messages are exchanged between mootes and server

Messages in the ecosystem are exchanged between motes and server as datagrams on top `UDP/IPv6`. The format of this message is described in section 1.

To communicate with the motes, the server sends `UDP` messages (on user demand) on port 3000. The querying datagram is placed within the payload of the `UDP` message. This is done via the following command where %d refers to the encoded datagram and %s to the address of the mote to contact: `echo %d | nc -u %s 3000`.

As each mote is configured to listen on port 3000, the corresponding one receives the message, decodes it and reacts depending on the kind of IoT device it is. For example, if it is a temperature sensor, the mote reacts by answering the current measure of temperature.

To retrieve the answers from the motes, a socket is created. This socket is bound to the source address of the server but is unconnected to a specific destination address. This allows the server to receive answers from any moote sending it data on port 3023 (this is the one chosen).

5 How automation is handled in the ecosystem

For the automation, we made two simple examples. Based on the client's needs, the scenario would be easily adapted.

The two scenarios are the following:

- If the server receives a proximity measure bellow 20[cm] from a proximity sensor, then the server asks for a connected lamp to change its state. The threshold chosen (20 [cm]) is randomly fixed and can be easily modified to correspond to a the smart application wanted. For example this automation could be used to automatically turn on and turn off the light in a hall.
- If the server receives a temperature measure above 20[°C] from a temperature sensor, then the server asks for a connected door lock to unlock enabling a door to open automatically. For example, this automation could be used to automatically open a window when the temperature is above 20 [°C].