University of Pisa
MSc in Computer Engineering
Advanced Network Architectures and Wireless Systems

IoT project

Daniela Comola
Sara Lotano
Eugenia Petrangeli

Academic Year 2019/2020

Contents

1 Introduction

The aim of this project is to periodically monitor the temperature provided by sensors located in a 6LoWPAN/RPL WSN and provide these data to a client outside the network that communicates only with a Proxy.

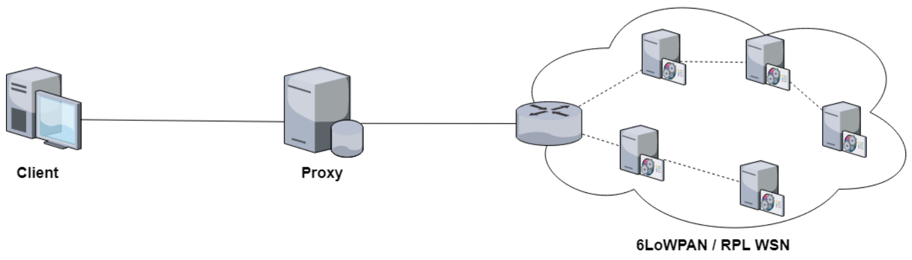


Figure 1: Network Architecture

1.1 Requirements

- 6LoWPAN multi-hop network (3-4 hops, 20-30 sensor nodes)
- Trickle algorithm
- Server CoAP with observable resource on Contiki motes
- JSON for data encoding (SenML)
- Proxy (implemented in Californium) with the following tasks:
  - observing CoAP Server resources
  - handling a cache
  - handling GET requests from CoAP Client

2 Design

The application consists of 3 basic components:

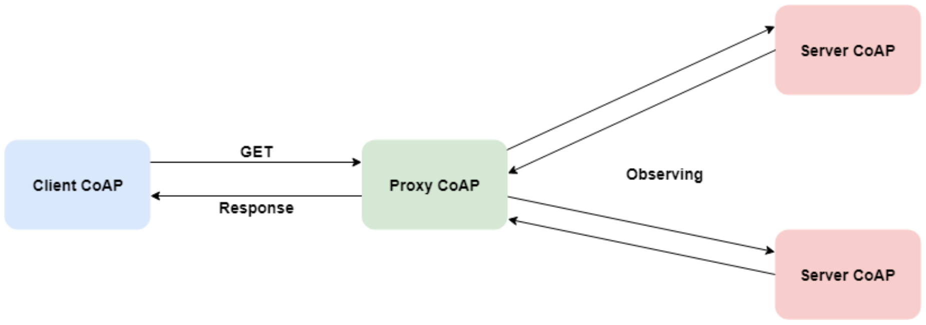


Figure 2: Application Architecture

1. **Client CoAP**: makes GET requests to obtain a specific resource.
2. **Proxy CoAP**: observes resources, stores data in the cache and provides resources to Client CoAP.
3. **Server CoAP**: simulates temperature sensing and exposes the value as an observable resource.

3 Implementation

3.1 6LoWPAN/RPL WSN topology

The deployed network is a multi-hop (3-4 hops) network composed by a RPL Border Router and 20 nodes that acts as CoAP Servers (each of them will run on a Z1 mote). A possible network topology is as follows, where the node with ID 1 is the RPL Border Router:

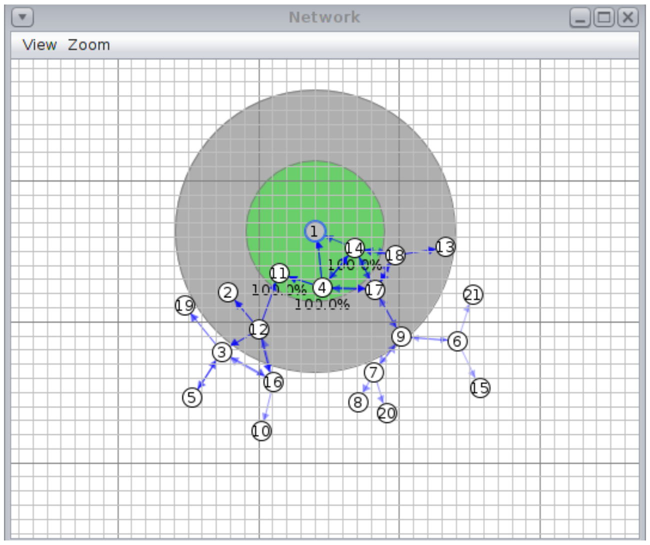


Figure 3: Cooja topology example

The network is deployed in the Cooja simulation environment.

3.2 Server CoAP

Each node in the 6LowPAN is a CoAP server that exposes the detected temperature as resource. Each server periodically sends the state of the resource to the registered clients. The temperature can assume values from 15 to 40 Celsius degrees and will be periodically increased (each 30 seconds) by a random value between 1 and 3 Celsius degrees. When upper limit is reached the value is restarted from 15.

The server allows only **JSON** format for the exchange of messages. For all other formats a NON_ACCEPTABLE status is set in the response and the client will receive an error.

The response message has the following fields:

```
{
    "n": "sensor_c00000002",
    "v": 21
}
```

The format is compliant to the specifications proposed in the **RFC 8428 – SenML** (Sensor Measurement Lists). More in details, the **n** field contains the unique sensor name and the **v** field contains the value of the measurement. Each CoAP Server will be implemented using **Erbium** (CoAP implementation in Contiki).

3.3 Proxy CoAP

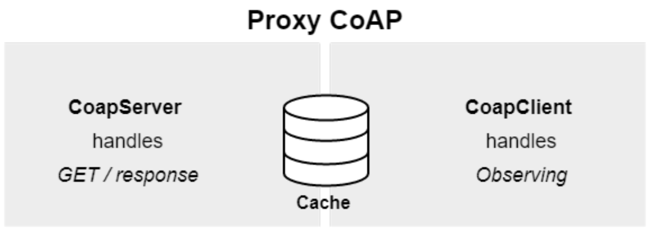


Figure 4: Proxy CoAP structure

The proxy performs observing on the resources exposed by the servers in the 6LoWPAN network (as a CoapClient), stores the latest value received from each node in a cache and satisfies the GET requests of a Client on behalf of the addressed node in the network, by means of the stored values. The proxy will be implemented in Java using **Californium** (CoAP library for Java).

3.4 Client CoAP

The only functionality of the client is to make GET requests to the Proxy. One of the most important assumptions is that the client already knows the URI of all the resources, and it writes the URI on the console. The client will be implemented in Java using Californium library.

4 Trickle parameters

The Trickle parameters that need to be configured are the following:[1]

```
    #undef RPL_CONF_DIO_REDUNDANCY
    #define RPL_CONF_DIO_REDUNDANCY 1

    #undef RPL_CONF_DIO_INTERVAL_MIN
    #define RPL_CONF_DIO_INTERVAL_MIN 12

    #undef RPL_CONF_DIO_INTERVAL_DOUBLINGS
    #define RPL_CONF_DIO_INTERVAL_DOUBLINGS 4
```

We choose to use a low value for the RPL_CONF_DIO_REDUNDANCY parameter, which represents the redundancy threshold K, because, low K values reduce the energy consumption of each node.
A possible drawback can be the formation of routes with low quality, but this is mitigated by the choice of a high value for I_{min} that allows to get optimal routes that do not need additional time to be refined.
On the other hand, high I_{min} values can lead to very high DODAG formation times. In order to verify the effects of this choice, 30 simulations with different seeds have been carried out in order to obtain an average value for the DODAG formation time.

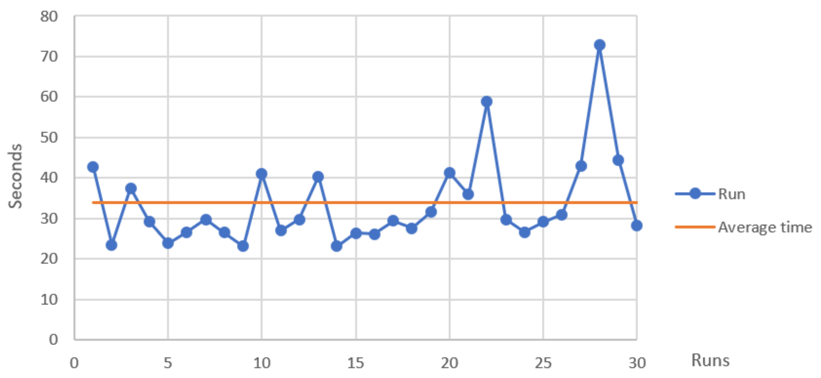


Figure 5: DODAG formation time with different seeds

The DODAG formation time takes 1 min and 30 sec in the worst case. The mean value is 33.547 sec and it can be considered compliant with the requirements of the application

[1]For the configuration of Trickle parameters, we considered the Trickle setting guidelines suggested in: E. Mingozzi C. Vallati. "Trickle-F: Fair broadcast suppression to improve energyefficient route formation with the RPL routing protocol".

5 Running the application

5.1 Prerequisites

To develop the 6LoWPAN network we used the **Iot Workshop VM**, which offers a complete **Contiki** development environment and includes **Cooja** simulation environment.

5.2 Running Cooja simulation

To simulate the 6LoWPAN network is necessary to:

- Put the *coojaCode* folder inside the VM *contiki* folder.
- Open the file *platform/z1/contiki-conf.h* and modify the following parameters:

```
    #ifndef NETSTACK_CONF_RDC
    #define NETSTACK_CONF_RDC          nullrdc_driver
    #endif

    #ifndef UIP_CONF_MAX_ROUTES
    #define UIP_CONF_MAX_ROUTES            25
    #endif
```

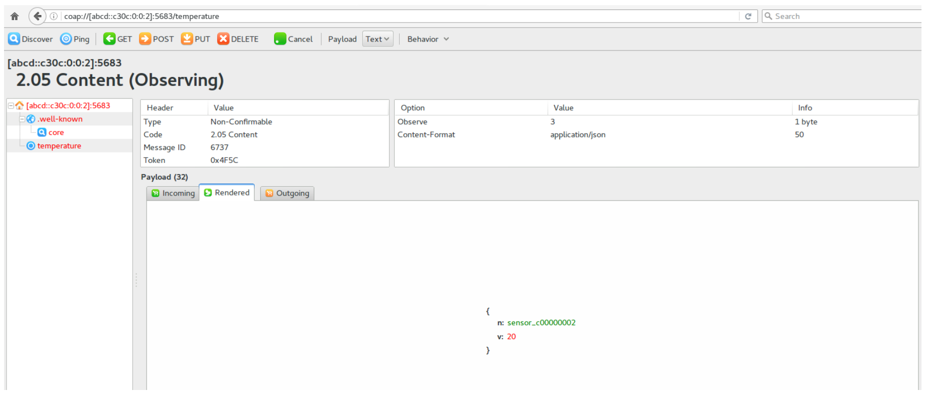
  Alternatively, you can replace the above file with the one located in *coojaCode/Platform_z1/contiki-conf.h* folder that already contains the modified parameters.

- Run **Cooja** and load the simulation located in *coojaCode/20nodes.csc.*
- From the *coojaCode/Border_router* folder, run the following command:

```
    $ make connect-router-cooja
```

Once all routes are found, the border router will print the message "DETECTED 20 ROUTES" inside the Mote output window.
After getting this message, you can start the Java application.
In order to check the list of detected routes, you can connect via web browser specifying the IPv6 address of the Border Router.
In our case is: *http://[abcd::c30c:0:1].* The IPv6 addresses of the CoAP Servers contained in the list are the same addresses written in the file *javaCode/src/main/java/code/Config.java* that will be used by the proxy to make observe requests.

In order to double-check the correct exposure of resources, you can use Copper by specifying the IP address of a specific CoAP Server and perform an Observe request to the available resource.



[abcd::c30c:0:0:2]:5683

2.05 Content (Observing)

Header	Value
Type	Non-Confirmable
Code	2.05 Content
Message ID	6737
Token	0x4F5C

Option	Value	Info
Observe	3	1 byte
Content-Format	application/json	50

Payload (32)

```
{
    n: sensor_c00000002
    v: 20
}
```

Figure 6: Resource exposed by CoAP Server with IPv6 abcd::c30c:0:0:2

5.3 Running ProxyCoAP and ClientCoAP

It is possible to run the Java application going into **javaCode** folder and executing the following commands:

```
$ mvn package

$ java -jar target/IoTProject-0.0.1-SNAPSHOT-jar-with-dependencies.jar
```

To make GET requests, you can use the same terminal from which the application was launched, by specifying the URI of the resource.
The terminal will show the following messages:

```
STARTING COAP CLIENT
Insert resource URI:
```

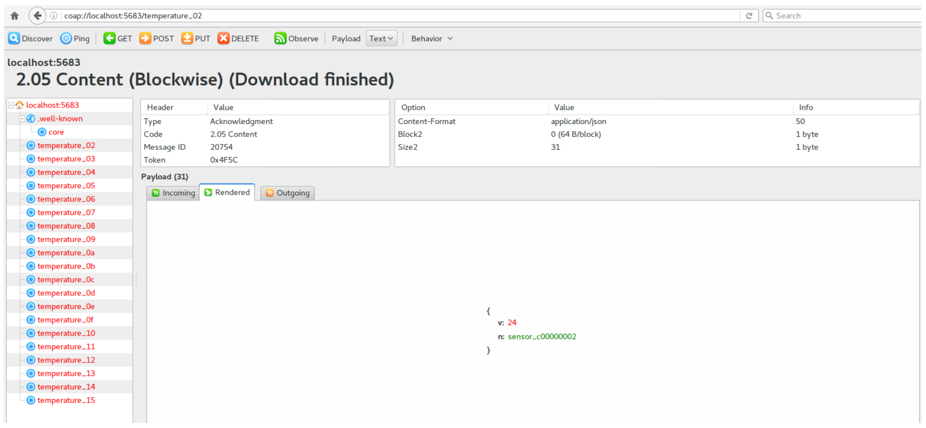
From now on, it is possible to make GET request. An example is:

```
coap://localhost/temperature_02
```

In order to verify which resources are managed by the proxy you can use Copper specifying:

```
coap://localhost
```

After making a Discovery, the list of resources will be shown and you can get their value by making a GET request.



localhost:5683

2.05 Content (Blockwise) (Download finished)

localhost:5683
- .well-known
  - core
- temperature_02
- temperature_03
- temperature_04
- temperature_05
- temperature_06
- temperature_07
- temperature_08
- temperature_09
- temperature_0a
- temperature_0b
- temperature_0c
- temperature_0d
- temperature_0e
- temperature_0f
- temperature_10
- temperature_11
- temperature_12
- temperature_13
- temperature_14
- temperature_15

Header	Value
Type	Acknowledgment
Code	2.05 Content
Message ID	20754
Token	0x4F5C

Option	Value	Info
Content-Format	application/json	50
Block2	0 (64 B/block)	1 byte
Size2	31	1 byte

Payload (31)

```
{
    v: 24
    n: sensor_c00000002
}
```

Figure 7: Resources handled by the Proxy