Internet of Things - Laboratory exercise 02

Department of Information Technology, Uppsala University

Overview

This laboratory exercise consists of two mandatory parts.

Administration

Student 1

Name: Anderson Leong Ke Sheng
UpUnet-S ID: anle5400

Student 2

Name: Qsai Alyounes
UpUnet-S ID: qsal1429

Student 3

Name:
UpUnet-S ID:

Agreement

I/we have independently worked on the following assignment solution. In the case of two or more people, we have both taken part in creating the solution, according to the assignment specification.

Sign 1: Anderson

Sign 2: Qsai

Sign 3:

Introduction to CoAP

The Internet today extensively uses Hypertext Transfer Protocol (HTTP) as an application layer protocol. However, HTTP is not suitable for IoT devices which are often battery operated with very limited computational power and available memory. CoAP or Constrained application protocol is an application layer protocol for Internet of Things (IoT) applications. CoAP in spirit is similar to HTTP. It performs similar function to HTTP but is specially designed for IoT applications. CoAP has small message size, low header overhead and a very low parsing complexity which helps in constrained environment of IoT devices. In addition, it implements security features designed for IoT applications. CoAP is a RESTful application layer protocol which works on top of UDP at the transport layer.

In the Internet, the resources (webpages) are identified by the URL (Uniform Resource Locator). In IoT applications, we need to be able to identify/address sensors as well. CoAP uses URI (Uniform resource identifier) as a way to identify these resources. A resource could be a temperature sensor, smart bulbs or other sensors or switches to control the sensors, e.g. to access a smart-bulb in your house. The address could be something like this:

```
coap://node.something.net/light
```

CoAP supports the following methods:

- `GET` - It is used to retrieve information from a resource specified by the URI. For example, retrieving a temperature value from the smart thermostat in your home. Accessing resources by using this method does not change the state of the resource.

- `POST` - This method requests the resource specified by the URI to process the data carried with the message. The actual function performed by the POST method is determined by the CoAP server and dependent on the target resource.

- `PUT` - This method requests that the resource identified by the request URI be updated or created with the enclosed representation.

- `DELETE` - This method requests to delete the resource specified by the request URI.

The PUT and POST methods are often misunderstood in that POST should be used to create a resource, and PUT to modify one. The URI in a POST request identifies the resource that will handle the enclosed entity. In contrast, the URI in a PUT request identifies the entity enclosed with the request. Therefore, it is logical to use POST method in applications where the request is to process the data carried with the request. For example, changing the color and intensity of smart bulb in your home. However, in this laboratory exercise, you will be using only GET and POST methods.
According to the CoAP standard (`RFC7252`), a request with an unrecognized or unsupported method must generate a *Method Not Allowed* piggybacked response.

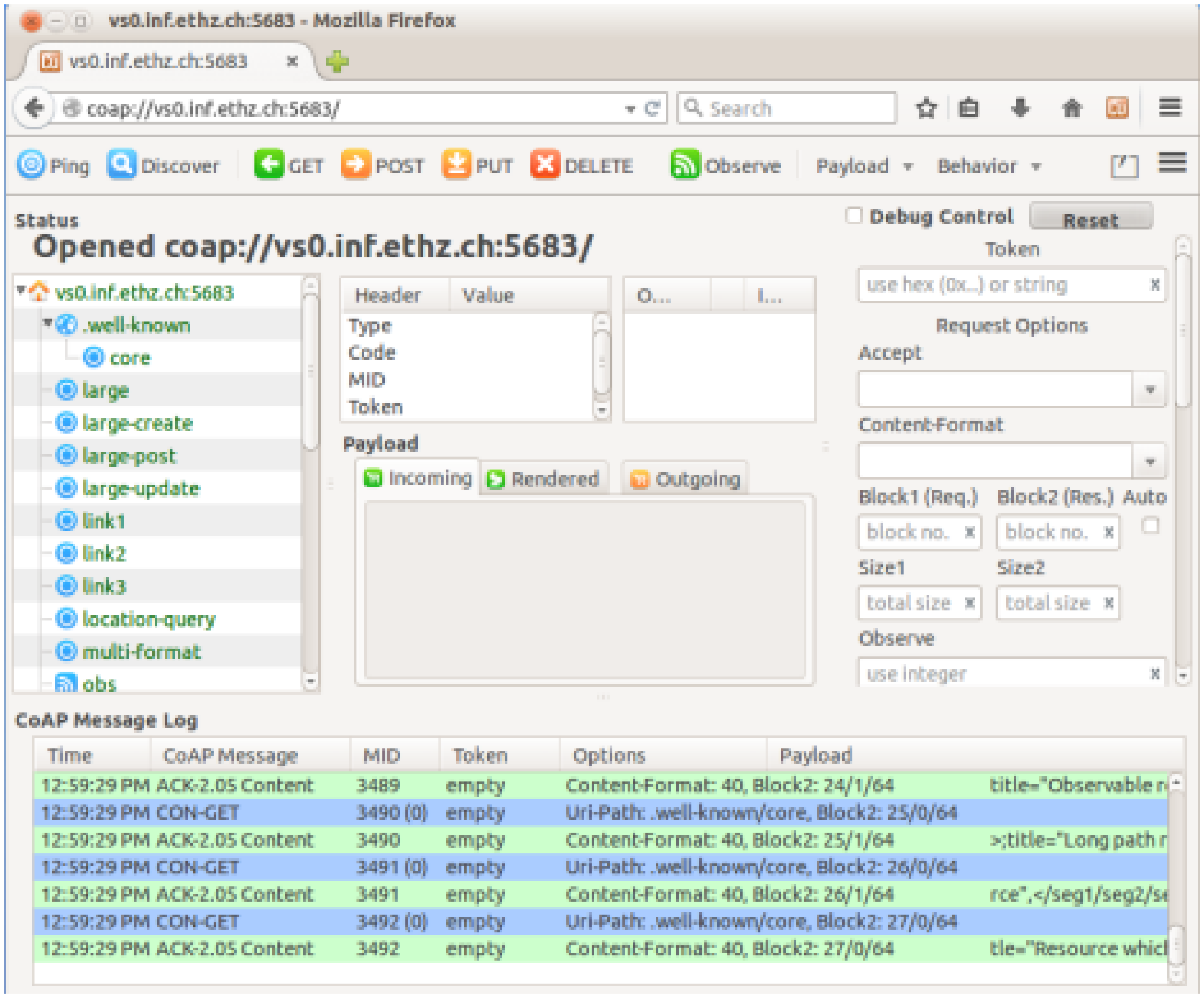
In CoAP, the client interested in the state of a resource initiates a request to the server. Then the server returns a response with a representation of the current state of the resource. However, this model does not work well when a client is interested in the current state of the resource over a period of time. Therefore, `RFC7641` introduces an extension to CoAP such that client can "observe" a resource on the server. When observing, the client receives a representation of the resource and requests this representation to be updated by the server as long as the client is interested in the resource.

Part 1

Introduction

In this part of the lab, you will use the web browser (Firefox) to interact with CoAP-enabled IoT devices. A plugin called *Copper* is already installed on the web browser (Firefox) of the virtual machine *(notice that the Copper installation might not work anymore if you updated Firefox to a newer version). Copper* is a user agent capable of sending and receiving CoAP messages from/to the CoAP-enabled IoT devices.

The following screenshot shows the Copper plugin while accessing a CoAP test server located at `coap://vs0.inf.ethz.ch`:
(T*his is used as an example, you will be interacting with a Cooja simulation in the exercise*)



Status
Opened coap://vs0.inf.ethz.ch:5683/

CoAP Message Log

Time	CoAP Message	MID	Token	Options	Payload
12:59:29 PM	ACK-2.05 Content	3489	empty	Content-Format: 40, Block2: 24/1/64	title="Observable r
12:59:29 PM	CON-GET	3490 (0)	empty	Uri-Path: .well-known/core, Block2: 25/0/64	
12:59:29 PM	ACK-2.05 Content	3490	empty	Content-Format: 40, Block2: 25/1/64	>;title="Long path r
12:59:29 PM	CON-GET	3491 (0)	empty	Uri-Path: .well-known/core, Block2: 26/0/64	
12:59:29 PM	ACK-2.05 Content	3491	empty	Content-Format: 40, Block2: 26/1/64	rce",</seg1/seg2/se
12:59:29 PM	CON-GET	3492 (0)	empty	Uri-Path: .well-known/core, Block2: 27/0/64	
12:59:29 PM	ACK-2.05 Content	3492	empty	Content-Format: 40, Block2: 27/0/64	tle="Resource whicl

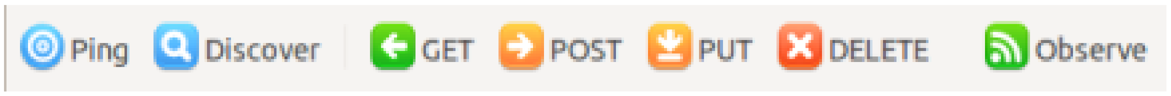
Similar to the first lab, you will use the Cooja simulator with six nodes running the Contiki operating system. In each of these nodes (except node with id 1), we have an instance of CoAP server running. The node 1 acts as a border router which acts as a bridge between the environment of the simulator where the CoAP server is running and the Linux operating system running in the virtual machine.

Copper

The version of *Copper* you use in this lab, implements several CoAP features, such as interacting through `GET`, `POST`, `PUT`, and `DELETE` methods, resource discovery, blockwise transfers and observing resources. In Copper, each of the above features have a button associated with it as is apparent in the above screenshot. Copper is extremely useful in developing large scale IoT systems. It can help significantly in debugging.

In order to access a CoAP resource, one can point directly to its address on the address bar of the web browser. For example, the resource `large` on the server `vs0.inf.ethz.ch:5683` can be accessed via the URL `coap://vs0.inf.ethz.ch:5683/large`.

Copper toolbar

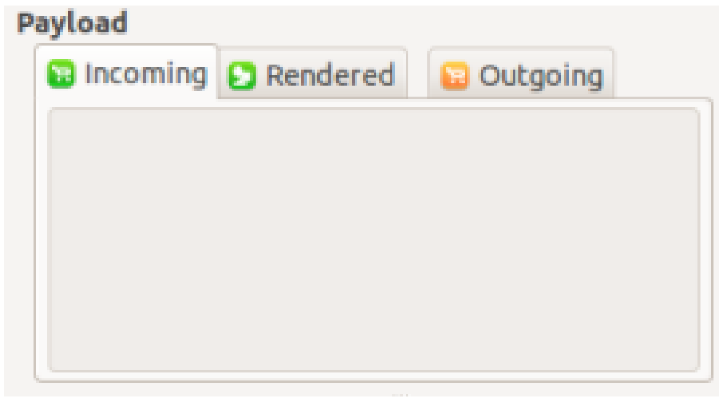


- The *Ping* button is used to send an empty confirmable message (reset message) in which the recipient of the message sends an acknowledgment back as a response. Therefore, *Ping* messages are used to check the liveness of an endpoint.
- The *Discover* button is used to discover which resources are available on a particular server. Available resources are displayed in resource pane.
- The buttons GET, POST, PUT and DELETE correspond to the `GET`, `POST`, `PUT` and `DELETE` methods in CoAP.
- The Observer button is used to "observe" resources, i.e., to retrieve a representation of a resource and keep this representation updated by the server over a period of time. This eliminates periodic use of `GET` requests.

Resource pane

The leftmost pane of Copper is the resource pane. All the resources of the server are shown here. The resources that are observable are represented with the icon 📶. Resources can be selected by clicking on them.

Payload tabs



There are two tabs that are important in this lab.

- *Incoming* tab shows the payload of the incoming CoAP messages. For example, output from the server related to GET method and the output when observing a resource is displayed here.
- *Outgoing* tab allows you to set payload of outgoing CoAP messages, e.g., when the POST method is used.

Getting started

1. file "*er-rest-iotlabs.tar.gz*" is already stored in the user directory ('/home/user'). Open a terminal and unzip it in the virtual machine using the below command, in the same directory:
   ```
   # tar xvzf er-rest-iotlabs.tar.gz
   ```
2. Change the directory to `er-rest-iotlabs` folder:
   ```
   # cd er-rest-iotlabs
   ```
3. Start the Cooja simulation by using the following command:
   ```
   # make TARGET=cooja iot-labs.csc
   ```
   - This command starts up Cooja with a pre-saved simulation.
   - In the *Network* subwindow of Cooja, you can see all six nodes.
   - Now, start the simulation by clicking the Start button in the *Simulation Control* subwindow.
   - *Notes on possible issues:* "Makefile:35: ../contiki-github/Makefile.include: No such file or directory. make: *** No rule to make target `../contiki github/Makefile.include'. Stop."
     - Change the path to Contiki: Open the Makefile and update line 4 from a relative to absolute path "CONTIKI=/home/user/contiki-github"
4. Let the simulation run for 120 simulated seconds in order to let RPL (you have learned about RPL in the previous lab) to construct the forwarding tree and set up routes among nodes. Here, you can see radio communication in the *Network* subwindow. (do not stop the simulation)
5. Now, we have to link the simulated IoT nodes in Cooja to the Internet (in case of the lab, our Linux machine). This would allow us to access these CoAP servers through the web browser (*Cooper*).

   - Open a new terminal as a tab in the existing terminal window by clicking File → Open Tab. This should open a terminal in a new tab in the current directory (`er rest-iotlabs`).

   - Use the following command to make a link between the border router and the network stack of the virtual machine.
     ```
     # make connect-router-cooja
     ```
     If you are asked, use "user" (no quotations) as the password.

   - After that, an output similar to the following should be displayed:
     ```
     Got configuration message of type P
     Setting prefix aaaa::
     Server IPv6 addresses:
      aaaa::212:7401:1:101
      fe80::212:7401:1:101
     ```

```
 (* In order to get a similar output, the simulator
should  be running)
```

7. Now open, the Firefox web browser by clicking the icon on the desktop.

8. The CoAP server on each node runs on port `5683` by default, as specified in the CoAP protocol. This port is used for communication between CoAP clients and servers. You can use the following URL format to access CoAP servers:
   `coap://[aaaa::212:740*:*:*0*]:5683/`
   Note that `*` represents the node ID. For example, the URL to access node 2 is `coap://[aaaa::212:7402:2:202]:5683/`.

   Enter the above URL in the web browser and access these nodes through the Copper plugin. Subsequently, answer the questions in the next section based on your understanding.
   Note that you can only interact with the simulation while it is running: For example, ensure that the simulation is running, visit node 3 in Firefox and click on "Discover" to find its resources.

Task

1. Write the names of the resources that you can discover on each node. (Hint: Use the discover feature of CoAP and list only the common resource names for all nodes)

All nodes have a core in a hidden folder ".well-known". All nodes except node 1 are able to discover the resources such as actuators that control the leds, and sensors for button and temperature. Pinging and discovering on node 1 does not yield results as seen in the same picture in the appendix as it is the router.

2. What are the types of CoAP access methods (GET, POST, PUT and DELETE) that are supported by the resources you discovered in the previous step?

POST/PUT mode=on|off. This is likely to set the actuators to the leds.
GET method can be run on button and temp resources since these resources are observing the respective data on the device.

3. Choose any node other than node 1. What error message do you get if you try to accesses an actuator (e.g., LEDs) on the node using the GET method?

We receive a message saying the method is not allowed.
"ACK-4.05 Method Not Allowed"

4. On the same node as in question 3, use the GET method to read the status of the *button* resource and write the status below (hint: see the payload in *incoming* tab)

GET button: EVENT 0
GET temp: TEMP 61

5. Now, switch back to Cooja and right click on the node you've selected in question 3 from the *Network* subwindow. Select "Click button on Sky *" to trigger a button click event on that node (* represents the node ID). Then use the GET method again as in question 4 to read the status of the button and write the response below.

EVENT 1

6. As you have seen in the Question 4 and 5, the GET method can be used to check the status of the button on the IoT nodes. Is there another feature available in CoAP which can allow you to get the status of button when an event happens ? e.g. to be notified when the button is clicked ?

Observe. This returns the new status in the payload whenever the resource being observed changes status.

Part 2

Description

In this task of the lab, you are going to assume that you are a contractor who is tasked to implement a simple monitoring software for a power plant. The plant has recently been upgraded with a number of CoAP-enabled sensor nodes which are equipped with temperature sensors and alarms. You are going to write a few simple applications to trigger alarms if the temperature exceeds safe operating conditions (> 50 °C).

You will be using the same simulation setup (1 border router and 5 sensor nodes) as in the previous task. If you have closed the windows, reopen them again for this part. In the simulation setup, the sensor nodes update their temperature reading every ten (simulated) second. To simulate a node triggering an alarm, you will turn on the red LED on the node.

Background

You will be programming in the *Java* programming language using the *Eclipse* programming environment. By itself, Java does not support the CoAP protocol. Therefore, you will be using an external library called *Californium*, which implements CoAP. But don't worry, everything is preconfigured for you.

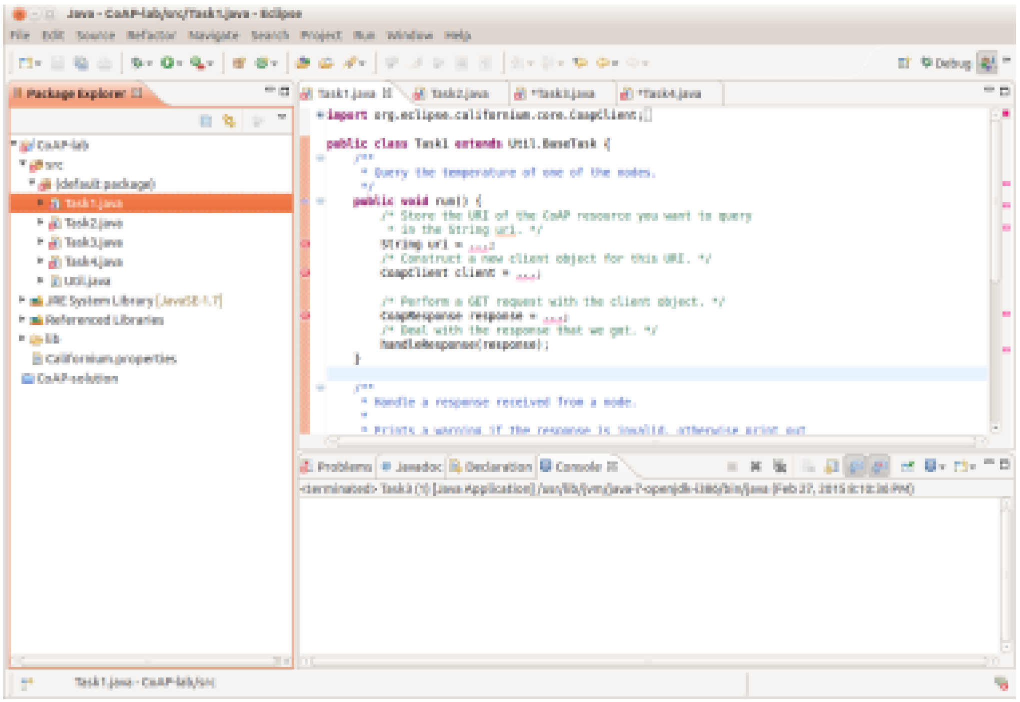
To make sure that you will be able to complete this task even if you have only little prior programming experience, we provide you with some code that you only need to complete.

The first step is to extract the provided code. The file "*coap-lab.tar.gz*" is stored in the user directory ('/home/user'). Open a terminal and unzip it in the virtual machine using the below command and store it in the same directory:

```
# tar xvzf coap-lab.tar.gz
```

```
(*tar xvzf command will extract the contents of a compressed archive file
named coap-lab.tar.gz in the current directory)
```

Start the Eclipse programming environment by clicking the icon on the desktop. Once Eclipse has started, switch to the Java perspective. In the menu bar, click *Window → Open perspective → Java*.



```
+import org.eclipse.californium.core.CoapClient;

public class Task1 extends Util.BaseTask {
    /**
     * Query the temperature of one of the nodes.
     */
    public void run() {
        /* Store the URI of the CoAP resource you want to query
         * in the String uri. */
        String uri = ...;
        /* Construct a new client object for this URI. */
        CoapClient client = ...;

        /* Perform a GET request with the client object. */
        CoapResponse response = ...;
        /* Deal with the response that we get. */
        handleResponse(response);
    }

    /**
     * Handle a response received from a node.
     *
     * Prints a warning if the response is invalid, otherwise print out
```

```
<terminated> Task3 (1) [Java Application] /usr/lib/jvm/java-7-openjdk-i386/bin/java (Feb 27, 2015 6:10:36 PM)
```

Now, import the project from the download. Click *File* → *Import*. In the dialog, select *General* → *Existing Projects into Workspace*. Click *Select root directory* and click the *Browse* button. Then, select the directory into which you have extracted the downloaded archive.

Once you have opened the project, you will see five classes in the **`src`** folder of the Package Explorer (left pane of the Eclipse window): Task1, ..., Task4, and Util. You will also notice the red markers on Task1, ..., Task4. They are there because the classes are missing some code that you need to fill in.

Task 1: Query the temperature on one node

Open the class Task1 by double clicking on it. Change the source code in the class so that it retrieves the temperature of any one of the nodes, e.g., node 5. Replace all occurrences of **"..."** in the source code with appropriate commands and method calls. Below you find a list of methods that you will probably need to complete the task.

Once you have completed the code, click *Run* → *Run* in the menu bar. You see your program's output in the *Console* pane in the bottom of the window.

Useful methods and constructors:

`new CoapClient(uri)`	creates a new **`CoapClient`** object.
`client.get()`	perform a GET request on a client.
`Util.isInvalid(response)`	returns true if a response is invalid.
`Util.parseTemperature(response)`	extracts the temperature from a valid response.

Task 2: Change the LED

For the second task, you need to turn on the red LED on the node if the temperature exceeds 50, and you need to turn it off if the temperature is below 50.

Open the class Task2 by double clicking it. You can copy the **`run()`** method from your Task1, but you need to modify the **`handleResponse()`** method.

In particular, you should also print the node's ID, and set the LED according to the temperature. To set the red LED, you need to create a new **`CoapClient`** object with an appropriate URI. In part 1 of the lab, you have learned that, for example, the red LED of node 2 is represented by the URI **`coap://[...]/actuators/leds?color=r.`** You can turn on the LED by sending a POST request with the message “**`mode=on`**” to the URI, and turn it off by sending a POST request with the message “**`mode=off`**”.

Make sure your code works correctly by checking in Cooja whether the LED is lit according to the reported temperature.

Useful to know: In Java you can concatenate strings and integers with the **`+`** operator.
For example, the following statements result in the String **`addr`** having the value **`"foo23"`**:

```
int i = 23;
String addr = "foo" + i;
```

Additional useful methods:

`Util.getNodeId(response)`	Get the ID (2-6) of the node that sent the response.
`Util.getBaseURI(response)`	Returns the base URI for a response. For example, if the response came from node **`aaaa::212:7401:1:101`**, then this method will return the String **`coap://[aaaa::212:7401:1:101]`**

Task 3: Query multiple nodes

Next, you will query all of the nodes in the network that is running a CoAP server, i.e., nodes 2 to 6.

Open the class Task3. If you look at the **`run()`** method, you will notice that the structure is different. There is an outer loop that is repeated while **`running`** is true. And there is an inner loop that counts up the integer variable **`i`** from 2 up to (including) 6. Afterwards, the program sleeps for 5 seconds. Your program will keep running until you enter **`"quit"`** in the *Console*.

Add suitable code to the body of the inner for-loop to query the *i*th node and print the result. You can use the **`handleResponse()`** function from Task2 by calling it directly, i.e., **`Task2.handleResponse(response);`**.

Task 4: Using observations to monitor nodes

In the previous task, you sent one GET request to each node every 5 seconds. In a resource constrained network, such active polling creates a considerable overhead and rapidly depletes the nodes' batteries.

As you know from part 1 of this lab, observations in CoAP allow a node to inform another device periodically about events. In this task, you will use observations to monitor the temperature on the nodes.

First, you need to modify the `run()` method of Task4. Just as in Task3, you have to create a **`CoapClient`** for each node; unlike in Task3, however, Task4 calls **`client.observe(this)`**. As a result, the method **`onLoad()`** of Task4 is called whenever a node reports a new temperature measurement. Notice also that **`client.observe(this)`** returns an object of type **`CoapObserveRelation`** that is stored in the list **`relations`**.
The **`CoapObserveRelation`** objects are very important, because they are used to cancel the

active observations. The nodes you are dealing with cannot handle more than three observers at the same time. And if a client (such as your program) forgets to cancel its reservation before it exits, the node may reject future observation attempts with the error message **"TooManyObservers."** Therefore, you should cancel all observations in the **shutdown()** method. There is a loop there that iterates over all **CoapObserveRelation** objects. Send a cancellation for each such object.
It is important that you only stop your program by entering **"quit"** in the console. Otherwise, **shutdown()** will not be called.

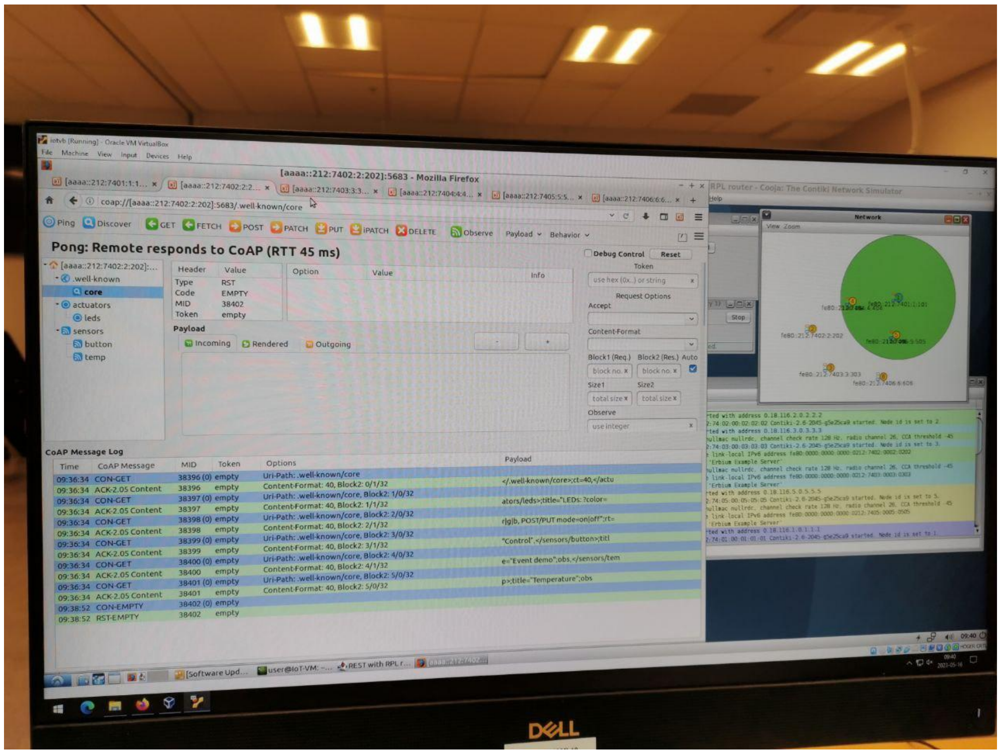
Finally, you need to implement **onLoad()**, the method that is called when a node reports a measurement. In this method, you should check whether the result is valid. If so, print the value the node reported along with the node ID. Note that unlike in the previous tasks, you do not need to turn on or off the red LED. (There appears to be a bug that prevents successful POST requests from being sent within **onLoad()**).

Show a demonstration of the complete code to a TA.

Additional useful method:

CoapObserveRelation.proactiveCancel()	Cancels an active observation.

Appendix



[aaaa::212:7402:2:202]:5683 - Mozilla Firefox

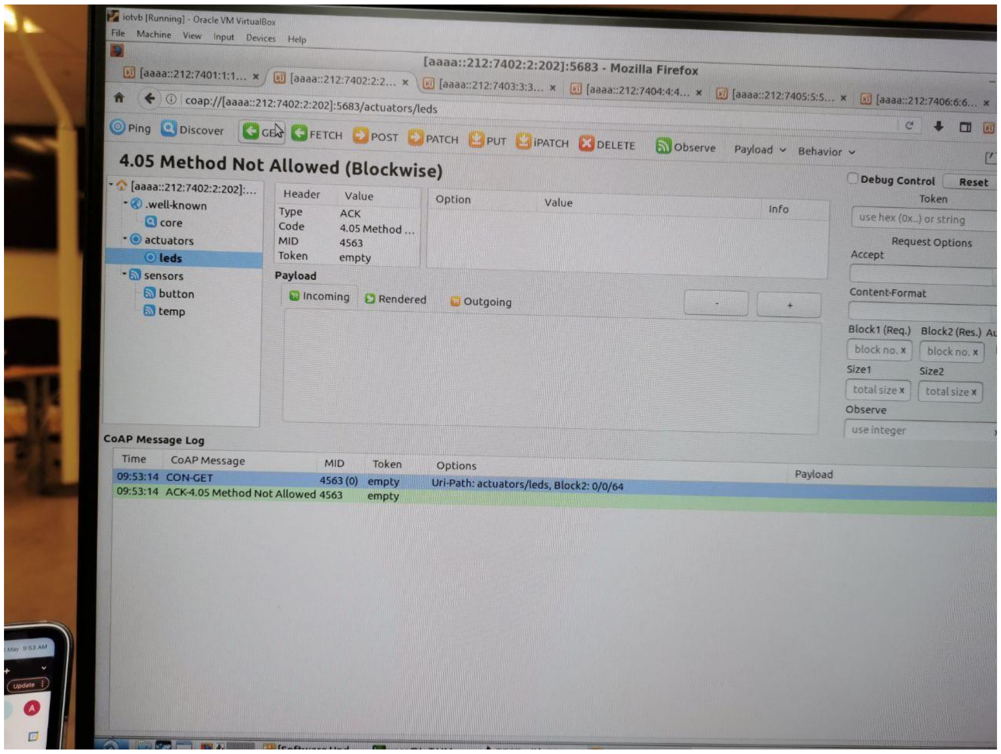
coap://[aaaa::212:7402:2:202]:5683/.well-known/core

Pong: Remote responds to CoAP (RTT 45 ms)

Header	Value
Type	RST
Code	EMPTY
MID	38402
Token	empty

CoAP Message Log

```
Time      CoAP Message       MID        Token  Options                                   Payload
09:36:34  CON-GET            38396 (0)  empty  Uri-Path: .well-known/core
09:36:34  ACK-2.05 Content   38396      empty  Content-Format: 40, Block2: 0/1/32        </.well-known/core>;ct=40,</actu
09:36:34  CON-GET            38397 (0)  empty  Uri-Path: .well-known/core, Block2: 1/0/32
09:36:34  ACK-2.05 Content   38397      empty  Content-Format: 40, Block2: 1/1/32        ators/leds>;title="LEDs: ?color=
09:36:34  CON-GET            38398 (0)  empty  Uri-Path: .well-known/core, Block2: 2/0/32
09:36:34  ACK-2.05 Content   38398      empty  Content-Format: 40, Block2: 2/1/32        r|g|b, POST/PUT mode=on|off";rt=
09:36:34  CON-GET            38399 (0)  empty  Uri-Path: .well-known/core, Block2: 3/0/32
09:36:34  ACK-2.05 Content   38399      empty  Content-Format: 40, Block2: 3/1/32        "Control",</sensors/button>;titl
09:36:34  CON-GET            38400 (0)  empty  Uri-Path: .well-known/core, Block2: 4/0/32
09:36:34  ACK-2.05 Content   38400      empty  Content-Format: 40, Block2: 4/1/32        e="Event demo";obs,</sensors/tem
09:36:34  CON-GET            38401 (0)  empty  Uri-Path: .well-known/core, Block2: 5/0/32
09:36:34  ACK-2.05 Content   38401      empty  Content-Format: 40, Block2: 5/0/32        p>;title="Temperature";obs
09:38:52  CON-EMPTY          38402 (0)  empty
09:38:52  RST-EMPTY          38402      empty
```



[aaaa::212:7402:2:202]:5683 - Mozilla Firefox

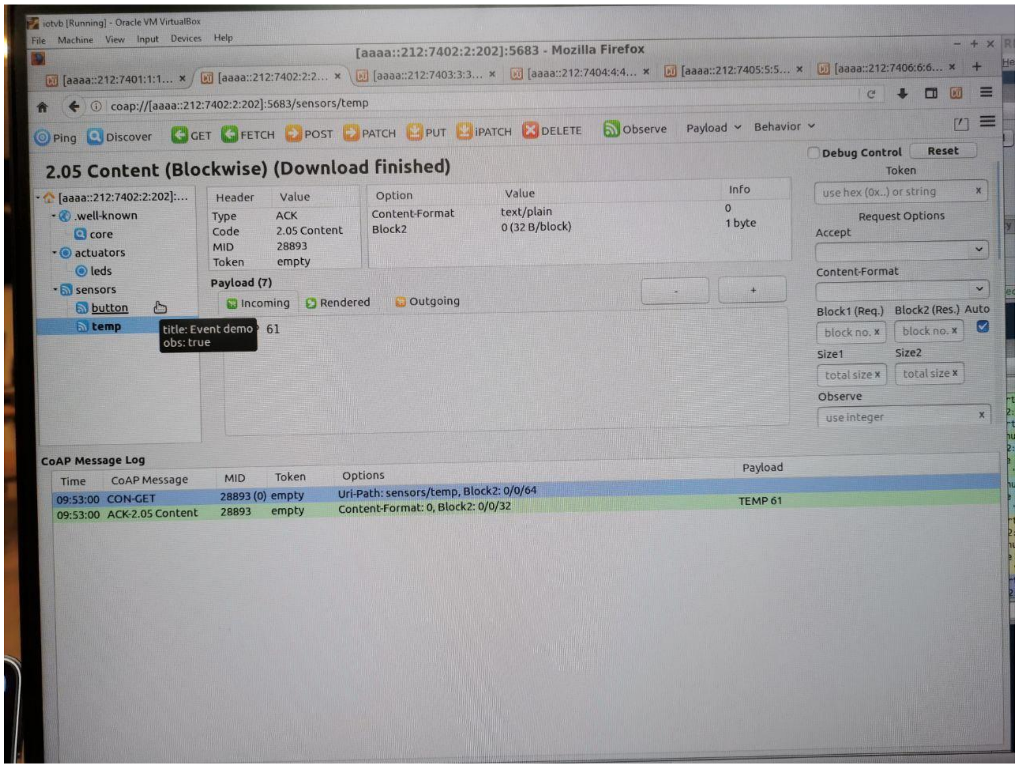
coap://[aaaa::212:7402:2:202]:5683/actuators/leds

4.05 Method Not Allowed (Blockwise)

Header	Value
Type	ACK
Code	4.05 Method ...
MID	4563
Token	empty

CoAP Message Log

```
Time      CoAP Message                   MID       Token  Options                                Payload
09:53:14  CON-GET                        4563 (0)  empty  Uri-Path: actuators/leds, Block2: 0/0/64
09:53:14  ACK-4.05 Method Not Allowed    4563      empty
```


iotvb [Running] - Oracle VM VirtualBox

[aaaa::212:7402:2:202]:5683 - Mozilla Firefox

coap://[aaaa::212:7402:2:202]:5683/sensors/temp

2.05 Content (Blockwise) (Download finished)

```
[aaaa::212:7402:2:202]:...
  .well-known
    core
  actuators
    leds
  sensors
    button
    temp
```

Header	Value
Type	ACK
Code	2.05 Content
MID	28893
Token	empty

Option	Value	Info
Content-Format	text/plain	0
Block2	0 (32 B/block)	1 byte

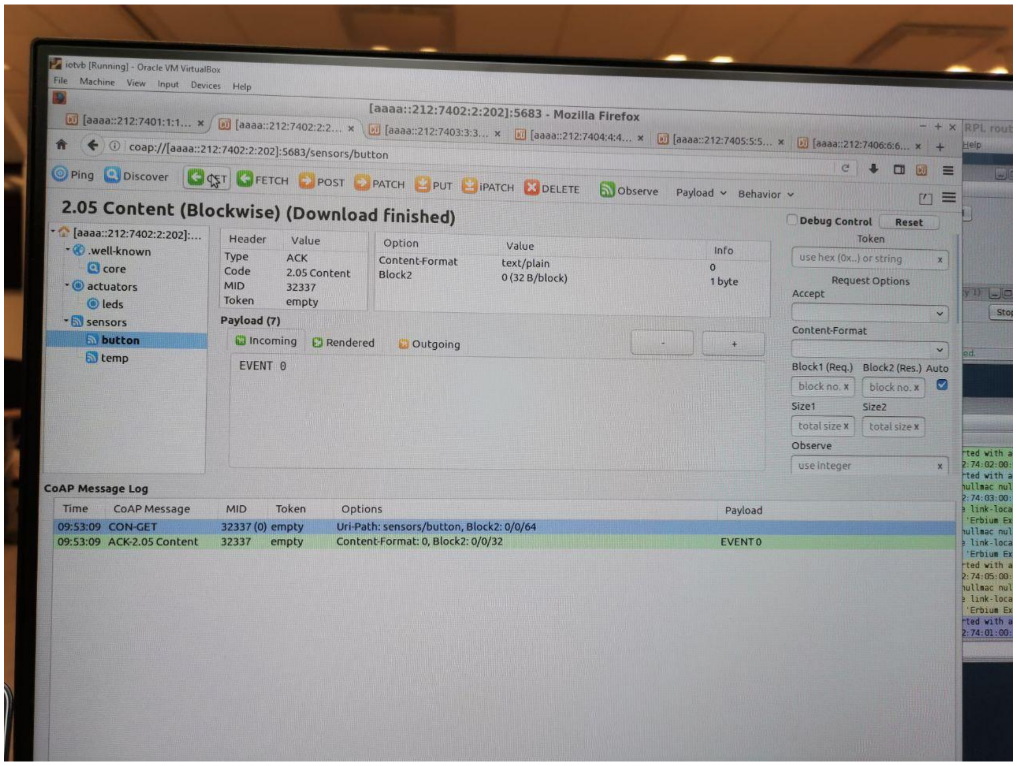
Payload (7)

title: Event demo
obs: true

61

CoAP Message Log

Time	CoAP Message	MID	Token	Options	Payload
09:53:00	CON-GET	28893 (0)	empty	Uri-Path: sensors/temp, Block2: 0/0/64	
09:53:00	ACK-2.05 Content	28893	empty	Content-Format: 0, Block2: 0/0/32	TEMP 61


iotvb [Running] - Oracle VM VirtualBox

[aaaa::212:7402:2:202]:5683 - Mozilla Firefox

coap://[aaaa::212:7402:2:202]:5683/sensors/button

2.05 Content (Blockwise) (Download finished)

```
[aaaa::212:7402:2:202]:...
  .well-known
    core
  actuators
    leds
  sensors
    button
    temp
```

Header	Value
Type	ACK
Code	2.05 Content
MID	32337
Token	empty

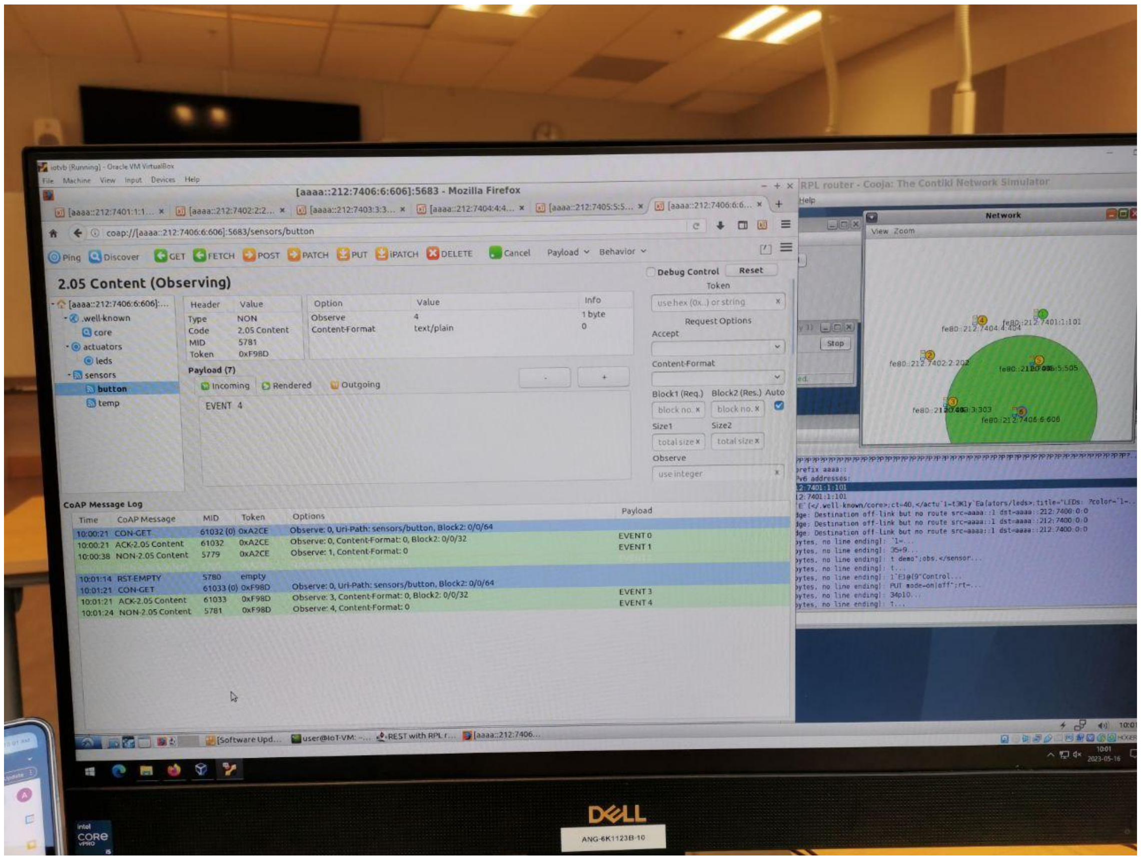
Option	Value	Info
Content-Format	text/plain	0
Block2	0 (32 B/block)	1 byte

Payload (7)

```
EVENT 0
```

CoAP Message Log

Time	CoAP Message	MID	Token	Options	Payload
09:53:09	CON-GET	32337 (0)	empty	Uri-Path: sensors/button, Block2: 0/0/64	
09:53:09	ACK-2.05 Content	32337	empty	Content-Format: 0, Block2: 0/0/32	EVENT 0


[aaaa::212:7406:6:606]:5683 - Mozilla Firefox
coap://[aaaa::212:7406:6:606]:5683/sensors/button
Ping
Discover
GET
FETCH
POST
PATCH
PUT
iPATCH
DELETE
Cancel
Payload
Behavior
Debug Control
Reset
[aaaa::212:7406:6:606]:...
.well-known
core
actuators
leds
sensors
button
temp
Network
RPL router - Cooja: The Contiki Network Simulator

2.05 Content (Observing)

Header	Value
Type	NON
Code	2.05 Content
MID	5781
Token	0xF98D

Option	Value	Info
Observe	4	1 byte
Content-Format	text/plain	0

Payload (7)

```
EVENT 4
```

CoAP Message Log

```
Time      CoAP Message        MID        Token   Options                                             Payload
10:00:21  CON-GET             61032 (0)  0xA2CE  Observe: 0, Uri-Path: sensors/button, Block2: 0/0/64
10:00:21  ACK-2.05 Content    61032      0xA2CE  Observe: 0, Content-Format: 0, Block2: 0/0/32        EVENT 0
10:00:38  NON-2.05 Content    5779       0xA2CE  Observe: 1, Content-Format: 0                        EVENT 1

10:01:14  RST-EMPTY           5780       empty
10:01:21  CON-GET             61033 (0)  0xF98D  Observe: 0, Uri-Path: sensors/button, Block2: 0/0/64
10:01:21  ACK-2.05 Content    61033      0xF98D  Observe: 3, Content-Format: 0, Block2: 0/0/32        EVENT 3
10:01:24  NON-2.05 Content    5781       0xF98D  Observe: 4, Content-Format: 0                        EVENT 4
```