UNIVERSITY OF PARMA
INTERNET OF THINGS 2015-2016

Hands on CoAP: Exercises

Ing. Luca Davoli (davoli@ce.unipr.it)

06 May 2016

1 Introduction

Welcome to the “Hands on CoAP” tutorial! You will learn how to develop RESTful IoT applications based on CoAP protocol. We will connect various devices and services. For this, we will use different CoAP libraries and frameworks, each with its own focus and benefits. You will need the Californium (*Cf*) CoAP framework. We recommend to use the Eclipse IDE for its good Java support. This tutorial assumes that you have a basic understanding of the Java language.

2 Getting Started

Before we dive into the first CoAP exercise, make sure you have a working development environment. Import into Eclipse the project named *IoTCourseProject*:

File > Import > General > Existing Projects into Workspace > Browse > Select the project

3 *Cf* Tutorial Server

First, you will implement your own CoAP server with *Cf* library. Then, you can test it with Copper (*Cu*) which is a plugin for Firefox browser which acts as CoAP client.

3.1 Create Main Server Class

The central class that implements the server will also contain the main. Your server class must extend the `CoapServer` class from the Californium framework.
In Eclipse, right-click on `src` folder and choose:

- *New > Class*
  - Package: `it.unipr.tlc.iot2016.cf.server`
  - Name: `TutorialServer`
  - Superclass: `org.eclipse.californium.core.CoapServer`
  - Tick `public static void main(String[] args)`
  - Finish

In the main method, you should run the server. To do so, do the following:

- Create an instance of the `TutorialServer` class
- Start the server

```
public static void main(String[] args) {
      TutorialServer tutorialServer = new TutorialServer();
      tutorialServer.start();
}
```

Now execute your main class by right-clicking and choosing *Run* > *Run As* > *Java Application*. You should see status logs in the console.

3.2 Add Resources

You can already query your CoAP server with the Copper plugin, by typing in Firefox:

coap://127.0.0.1:5683/

in particular the address *127.0.0.1* corresponds to the IP loopback address of your machine (localhost) and *5683* is the default port for CoAP (that can also be omitted). When you browse your CoAP server with Copper (Cu), it will already reply to pings and provide the */.well-known/core* resource for discovery.
However, there are no resources on your server yet. Let's start adding the obligatory "HelloWorld!"

Create a new resource class in a resources sub-package under `src` folder:

- *New* > *Class*
    - Package: `it.unipr.tlc.iot2016.cf.server.resources`
    - Name: `HelloWorldResource`
    - Superclass: `org.eclipse.californium.core.CoapResource`
    - Finish

Once completed the creation of the CoAP resource, implement its constructor, that it has to accept a string parameter. In order to correctly build the CoAP resource, in its constructor the first instruction must be the call to the `super` constructor, that owns all properties to instantiate a CoAP resource.
The constructor parameter, passed to the `super` constructor, represents the name of the resource, with which you can refer during interactions through CoAP clients (e.g., Copper).

```
public class HelloWorldResource extends CoapResource {
      //Constructor
      public HelloWorldResource(String name) {
            super(name);
      }
}
```

Californium manages CoAP requests based on the type of incoming instance. Since that our purpose is to build a `HelloWorld` resource that replies to simple GET request, what we have to implement is a component that is able to handle GET demands.

- Declare a GET handler, modifying the behavior of the following method:

```
@Override
public void handleGET(CoapExchange exchange)
```

- Let the handler respond with “Hello World!”, using a feature of the provided exchange:

```
exchange.respond("Hello World!”);
```

Since you have defined only the payload into the GET handler, this means that your resource reply to a request creating a CoAP message with:

  - Code: *2.05*
  - Payload: *Hello World!*
  - Payload MIME-type: *text/plain*

```
public class HelloWorldResource extends CoapResource {
      public HelloWorldResource(String name) {
            super(name);
      }

      @Override
      public void handleGET(CoapExchange exchange) {
            exchange.respond(ResponseCode.CONTENT,
                        "Hello World!",
                        MediaTypeRegistry.TEXT_PLAIN);
      }
}
```

Now your `HelloWorld` resource is ready, but your Californium server does not know the existence of this new resource. You have to add your new `HelloWorld` resource to the Californium server, to be able to query it.
In your CoAP server implementation, add the following instructions, before starting it:

```
HelloWorldResource hello = new HelloworldResource("hello-world");
tutorialServer.add(hello);
```

Finally, stop every instances of your Californium server (to avoid that your newly deployed behavior doesn’t reflect server work); then run your Californium server and browse it with Copper. You should be able to receive a “Hello World!” by making a GET call on the *hello-world* resource (also note the Content-Format in the response).

Now it’s time to improve your Californium server! Go ahead and add more resources that also handle POST, PUT, or DELETE requests, in an identical fashion of the `HelloWorld` resource.

3.3 Observing Resources

Now we will create an observable resource that informs all observers whenever its state changes.

- Create a new Californium CoAP resource called `ObservableResource` , in a similar manner to that described for the `HelloWorld` resource.
- Implement the GET handler, so that it responds with the currently stored data.

To make the resource observable, do the following steps:

- In the `TutorialServer` class, use the constructor in order to create a new instance of the `ObsResource` class:

```
ObsResource obsRes = new ObsResource("observable-resource");
```

- Then make it observable by typing:

```
obsRes.setObservable(true);
obsRes.getAttributes().setObservable();
```

- Don't forget to add the resource to the server:

```
tutorialServer.add(obsRes);
```

Add a private variable to the class `ObsResource`, which will store the current value of the resource, and initialize it to zero.

```
private int mValue = 0;
```

As already done for the *hello-world* resource, implement the `handleGET` method in order to reply to GET calls:

```
public void handleGET(CoapExchange exchange) {
      exchange.respond(ResponseCode.CONTENT,
                       mValue+"",
                       MediaTypeRegistry.TEXT_PLAIN);
}
```

Now the observable resource can be accessed via Copper with GET calls and can be observed as well. However, if you try to observe the resource you will note that the observed value is always 0! This is because your variable do not change with time! In order to let the variable change over time you can simply copy-paste this code in the constructor of the class:

```
Timer timer = new Timer();
timer.schedule(new UpdateTask(this), 0, 1000);
```

You have also to define the `UpdateTask` class: copy this code into the `ObsResource` class:

```
private class UpdateTask extends TimerTask {
      private CoapResource mCoapRes;
```

```
		public UpdateTask(CoapResource coapRes) {
			mCoapRes = coapRes;
		}
		@Override
		public void run() {
			mValue = new Random().nextInt(20);
			mCoapRes.changed();
		}
	}
```

Now the value of your resource change periodically with a period of 1 second and takes a random value between 0 and 20. You should see this behavior by observing the resource with Copper plugin.

4 Cf Tutorial Client

Until now, you have used the Copper plugin in order to make query to your CoAP server. Now is time to implement your own Java implementation of a CoAP client through *Cf* framework.

4.1 Synchronous CoAP client

We are going to create a simple CoAP client which make a GET call to a predefined resource.
In order to do so, as already done for the CoAP server, you have to create a new class with a `main` method. You can denote this class as `SimpleCoapClient` and you will use it in order to run your CoAP client implementation.
In Eclipse, right-click on `src` folder and choose:

- *New > Class*
    - Package: `it.unipr.tlc.iot2016.cf.client`
    - Name: `SimpleCoapClient`
    - Tick `public static void main(String[] args)`
    - Finish

In you `main` method create a new object of the class `CoapClient` and use its constructor in order to set the address of the resource you want to get (if the server resides on your machine its address corresponds to the IP loopback address):

```
	CoapClient client = new CoapClient("coap://127.0.0.1:5683/<resname>");
```

Then you have to declare a `Request` object. `Request` represents a generic CoAP request (could be GET, POST, PUT and DELETE). In order to create a GET request you can use its constructor:

```
	Request request = new Request(Code.GET);
```

Now you can actually send the GET request and print the CoAP server response by writing:

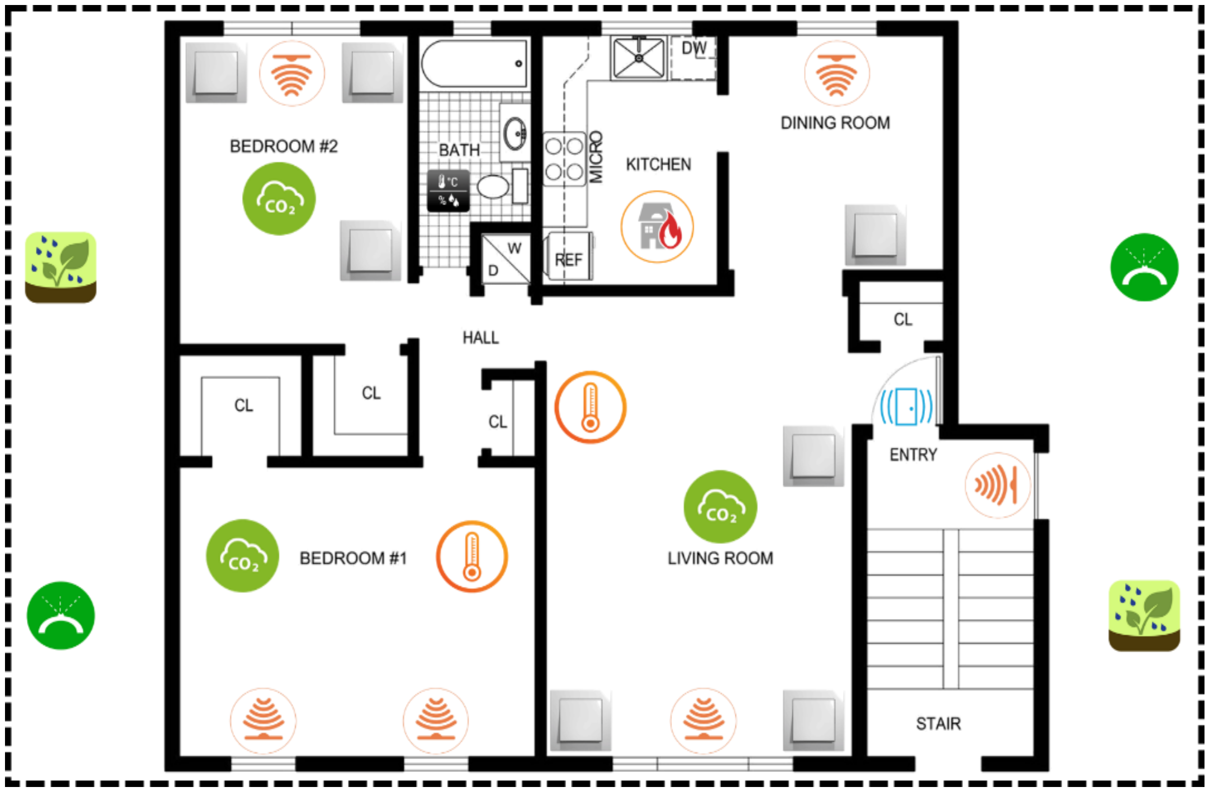
```
//Synchronously send the GET message (blocking call)
CoapResponse coapResp = client.advanced(request);

//The "CoapResponse" message contains the response.
System.out.println(Utils.prettyPrint(coapResp));
```

Try to query your CoAP server with your CoAP client. Note that this is a **synchronous** implementation of a CoAP client. This means that when the `advanced(request)` method is invoked, the program wait for the response (blocking call) before proceeding with the other instructions. The Californium libraries provide also an **asynchronous** implementation of the CoAP client as well as the observing functionalities.

5 Experimental Scenario

After all the previous steps, now you can try to implement an experimental scenario: you have to deploy a Smart House scenario, in which you need to monitor a residential house.



Each domestic environment has to be managed by a single CoAP server, that maintains several CoAP sensors, as shown in the figure (e.g., temperature, humidity, proximity, gas, intrusion, garden rain). By this, you have to implement different CoAP servers and different CoAP clients able to interact with these servers.