

The CoAP protocol

Project Work on “Ingegneria dei Sistemi Software M”



a.a. 2018-2019

Kevin Leto

# Introduction and aims

The aim of this project work is to create a small IoT application to make possible the study of interactions between sensor and actuator components.

The goal is to test if CoAP (Constrained Application Protocol) is compatible with the QActor, or more in general, the Java Framework.

The increasing number of IoT devices in everyday life has driven us to go deeper in this protocol because it was developed specifically for low-energy devices that are connected to lossy-networks.

This IoT application is only a sample created in order to use CoAP protocol in a real case of study.

# CoAP e Californium

## The CoAP protocol

### The Constrained Application Protocol is a light-weight protocol that works on UDP (or DTLS, if a security context is needed). It has been created to connect low-energy devices on the same network or on different networks connected by the internet and it can be easily integrated with HTTP. For that reason, the CoAP protocol allows a REST (REpresentational State Transfert) communication based on a client-server architecture in the general case (or an observer-observable architecture in some special cases).

### Client-Server architecture

The general use case of CoAP is based on a client-server architecture. The server is the one that exposes resources on the network and makes them available via URI (Uniform Resource Identifier). The client is the one that has access to the resources via HTTP messages to the server.

The CoAP communication is stateless, a request-reply interaction with no permanent server-side state.

In general, the client-server communication can be reliable or not. In the first case, the delivery of the message is guaranteed (retransmission mechanism exists) and communication is based on TCP. In the second case, there are no mechanisms for not delivered messages and communication is based on UDP (like CoAP).

## Californium

CoAP is only a communication protocol. A program that wants use it needs to have a CoAP implementation provided by an external library. Our application will be written in Java, so we need a Java implementation of CoAP that is the Californium library. Californium is a framework that allows applications to control at 360 degrees CoAP interactions.

### CoAP characteristics in Californium

Every CoAP implementation provides to the user a series of features that allow the user to adapt the protocol to his needs.

In particular, Californium allows three types of communication:

* Synchronous: the client blocks its execution until the reply comes.
* Asynchronous: the client continues its execution and when the reply comes a callback is executed.
* Observable: the client register itself as an observer of the resource. In that way, the client will be notified when the resource’s state changes.

# A case of study: The Radar Controller

The IoT application that we want to create consists of a radar controller that can control a Radar Display (already available in QActor). The Radar Display (RD) shows a point (the current point position) in a circular plane centered in (0,0). The point is identified by a couple of coordinates (x,y).

That is the Radar Display GUI:

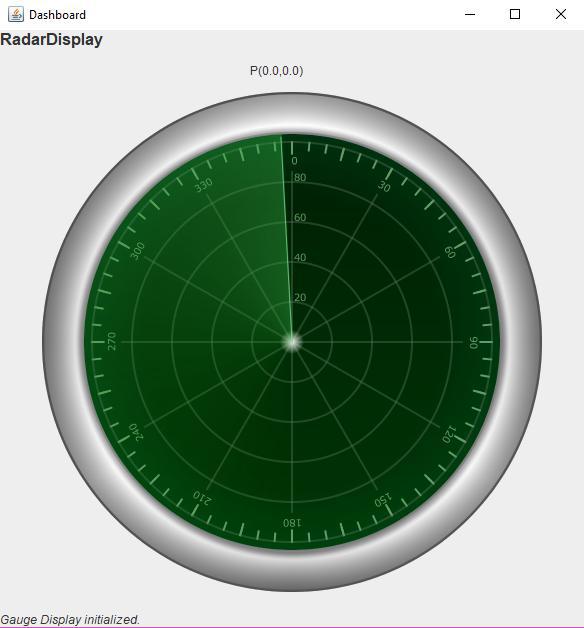


Figure 1: RadarDisplay GUI

The first one refers to the distance between the radar and the detected object (0 <= x <= 90) and the second one refers to the angle clockwise between the vertical axis and the object (0 <= y <= 360).

A RadarPoint will be exposed by the server as a CoAP resource and the client will be able to do two types of operation:

* GET request: ask for the current point position.
* PUT request: update the current coordinates.

The Radar Display is a component that can receive point information via QActor and show it in the GUI. It needs to receive a QActor event exactly like that:

|  |  |
| --- | --- |
| 1 | **Event** polar : p( Distance, Angle ) |

Figure 2: Polar event format

## Agile approach

In the developing of this project, we will follow an Agile approach. It requires to develop the application in an iterative and incremental way, developing one requirement at each iteration.

## Requirements

That is the list of requirements that we have to develop in this project work:

1. Create a basic architecture that allows to send coordinate to the Radar Display (with the QActor infrastructure).
2. Create a Client-server architecture (with QActor).
3. Create a client-server model that implements Get and Put requests with CoAP.
4. Add a Client GUI controller.
5. From sync to async communication.

# Requirement a): Basic architecture

We need to create a first formal model to describe the basic architecture.

Since we have to communicate with a QActor application (the Radar Display) and the QActor model is an executable model, is easy to create another QActor component that can interact with the RD. That is a good choice because it isn’t the final release (it’s the first prototype) and is very quick to implement in that way.

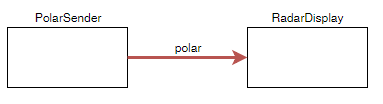


Figure 3: polarSender scheme system

## First formal model - PolarSender

The first formal model is represented by the seguent QActor code:

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | **System** polarsender  **Event** polar : p( Distance, Angle )  **Context** ctxPolarSender **ip** [ **host=**"localhost" **port=**8009 ]  **Context** ctxRadarBase **ip** [ **host=**"localhost" **port=**8033 ] **-standalone**  **QActor** sender\_actor **context** ctxPolarSender{  **Plan** init **normal**  [  **println**("sender\_actor starts.")  ]  **switchTo** sendPolarEvent    **Plan** sendPolarEvent  [  **emit** polar : p(90,20);  **delay** 1500;  **emit** polar : p(40,120);  **delay** 1500;  **emit** polar : p(0,0)  ]  } |

Figure 4: polarSender.qa

In this QActor model, the actor sender\_actor sends polar events to the radar component that shows the received information (distance, angle) in the GUI.

In that model, there is no trace of CoAP and the communication takes place using the QActor infrastructure that hides the communication complexity. From this point forward, we are going to remove the infrastructure offered by the QActor framework and replace it with the CoAP one using Californium.

To use the Radar Display already available, is necessary that the final communication has to take place only with QActor (the source code is not available). But the client-server interaction will be completely independent.

This first model is very limited and it isn't properly a client-server interaction, because the communication is only monodirectional (from the PolarSender to the RadarDispay) and the RadarDisplay cannot reply to the PolarSender events (the request, in that case, is called "fire & forget" where the sender doesn't expect a response).

# Requirement b) Client-Server architecture

Now, we can modify the previous model to create a real client-server interaction in which concepts of client and server really exist. Also in that model, the interaction between the two component will take place with the QActor infrastructure.

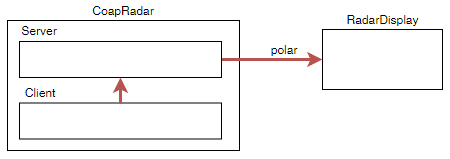


Figure 5: coapRadar-v0 scheme system

The formal model defines the ways in that client and server can communicate with each other. The first part of the model has that scope.

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11 | **System** systemV0  **Event** polar : p( Distance, Angle ) // between server and RadarDisplay  **Dispatch** getValue : getValue // from client to server  **Dispatch** putValue : value ( Distance, Angle ) // from client to server  **Dispatch** sendValue: value ( Discance, Angle ) // from server to client  **Context** ctxSystemV0 **ip** [ **host=**"localhost" **port=**8000 ]  **Context** ctxRadarBase **ip** [ **host=**"localhost" **port=**8033 ] **-standalone** |

Figure 6: event and dispatch descriptions in coapSystem.qa – coapRadar version 0

The event polar (a not reliable message) is used by the server to communicate with the Radar Display. It cannot be used by the client (the client should not know anything about it).

The dispatch getValue (a reliable message) is used by the client to contact the server and ask for a GET request. The dispatch sendValue is used by the server to reply at a client's GET request, to send back the current point position. Finally, the dispatch putValue is used by the client to send to the server the new coordinates of the point.

Now, we look at the client and the server separately to explain in a more detailed way how they work.

## Server

The server is the only one that can communicate directly with the radar (it is the only one that knows the polar event). All the external interactions have to pass through the server that, according to the request's type, will communicate the necessary information to the radar.

The server QActor model is depicted below.

|  |  |
| --- | --- |
| 12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61 | **QActor** server **context** ctxSystemV0 {  **Rules**  {  point(X,Y) :- distance(X), angle(Y).  }    **Plan** init **normal**  [  **println**("server: start.");  **emit** polar : p(0,0);  **addRule** distance(0);  **addRule** angle(0)  ]  **switchTo** waitingMessage    **Plan** waitingMessage  [  **println**("server: Waiting message...")  ]  **transition** **stopAfter** 360000  **whenMsg** putValue -> putReceived,  **whenMsg** getValue -> getReceived  **finally** **repeatPlan**    **Plan** putReceived **resumeLastPlan**  [  **println**("server: Put received.");  **println**("server: Distance = ");  **onMsg** putValue : value(D, A) -> **println**(D);  **println**("server: Angle = ");  **onMsg** putValue : value(D, A) -> **println**(A);  **removeRule** distance(\_);  **removeRule** angle(\_);  **onMsg** putValue : value(D, A) -> **addRule** distance(D);  **onMsg** putValue : value(D, A) -> **addRule** angle(A);  **onMsg** putValue : value(D, A) -> **emit** polar : p(D, A)  ]    **Plan** getReceived **resumeLastPlan**  [  **println**("server: Get received.");  [ !? point(X,Y) ] **forward** client **-m** sendValue : value(X,Y);  **println**("server: Point emitted:");  **println**("server: Distance = ");  [ !? point(X,Y) ] **println**(X);  **println**("server: Angle = ");  [ !? point(X,Y) ] **println**(Y)  ]  } |

Figure 7: server actor in coapSystem.qa – coapRadar version 0

Basically, the first thing that the server does is initialize the radar. After that, it goes in a waiting state until a GET or PUT request arriving. When a request arrives, if it's a GET, the server replies with the current point position. Otherwise, if it's a PUT, the server updates the stored point position and communicates the new point to the radar. After the request has been managed, it returns in the waiting state creating an infinite loop.

## Client

That is the client formal model. Its execution is very simple. It starts, executes a first GET request and prints out the received coordinates. Then it executes a PUT request and immediately a GET request to test if the coordinates sent are correctly received and stored. Then it terminates its execution.

|  |  |
| --- | --- |
| 62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112 | **QActor** client **context** ctxSystemV0{  **Plan** init **normal**  [  **println**("client: start.");  **delay** 500  ]  **switchTo** getInitialValue    **Plan** getInitialValue  [  **forward** server **-m** getValue : getValue;  **println**("client: Emitted GET.")  ]  **transition** **stopAfter** 360000  **whenMsg** sendValue -> receiveInitialValue    **Plan** receiveInitialValue  [  **println**("client: Response GET received.");  **println**("client: Distance = ");  **onMsg** sendValue : value(Distance, Angle) -> **println**(Distance);  **println**("client: Angle = ");  **onMsg** sendValue : value(Distance, Angle) -> **println**(Angle)  ]  **switchTo** sendingMessage    **Plan** sendingMessage  [  **forward** server **-m** putValue : value(10,10);  **println**("client: Message sent: value(10,10).");  **println**("client: Verifying update...");  **forward** server **-m** getValue : getValue  ]  **transition** **stopAfter** 360000  **whenMsg** sendValue -> receiveValue    **Plan** receiveValue  [  **println**("client: Response GET received.");  **println**("client: Distance = ");  **onMsg** sendValue : value(Distance, Angle) -> **println**(Distance);  **println**("client: Angle = ");  **onMsg** sendValue : value(Distance, Angle) -> **println**(Angle)  ]  **switchTo** clientStop    **Plan** clientStop  [  **println**("client: Client stopped.")  ]  } |

Figure 8: client actor in coapSystem.qa – coapRadar version 0

## Manual testing

To test if everything works fine, we need to start at first the RadarDisplay and then the coapRadar. So, running the model and looking at the output console, it will be like that:

|  |
| --- |
| server: start.  client: start.  server: Waiting message...  client: Emitted GET.  server: Get received.  server: Send value: Distance = 0, Angle = 0.  server: Waiting message...  client: Response GET received: Distance = 0, Angle = 0  client: Emitted PUT: value(45,90).  client: Verifying update...  server: Put received: Distance = 45, Angle = 90.  server: Waiting message...  server: Get received.  server: Send value: Distance = 45, Angle = 90.  client: Response GET received: Distance = 45, Angle = 90.  server: Waiting message...  client: Client stopped. |

Figure 9: output console of coapSystem.qa – coapRadar version 0

Looking at the output console we can follow all the client and server steps: starting, sending/receiving requests and stopping. Everything works fine because when a request arrives, if it’s a PUT the current point position is correctly updated and if it’s a GET the current coordinates are send out.

# Requirement c): Client-Server architecture with CoAP

Now is time to remove the QActor infrastructure (where possible) and replace it with the CoAP one.

We will replace the current system model (systemV0) with another one in which the client and the server are into two different sub-systems (coapServer and coapClient). The interactions between the sub-systems will be implemented with Californium (the CoAP library).

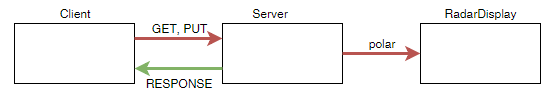


Figure 10: coapClient-v1 and coapServer-v1 scheme systems

## Californium nuggets

Before starting to explain the new system model, is useful to give a quick presentation of the Californium library, in order to give an explanation of the main Java Classes used in this project work.

The main Californium Java classes used in this project are:

* **CoapResource**: represents a CoAP resource identified by a URI, it can be accessed by clients with REST operations. Every resource determines which REST operations are allowed on it implementing the corresponding handle methods.
* **CoapServer**: represents a RESTFUL CoAP server which exposes its resources to clients (the listening default CoAP port is 5683).
* **CoapClient**: represent a CoAP client which can make REST requests to obtain resources information or manipulate them.
* **CoapResponse**: represents a CoAP response message containing all information concerning it.
* **CoapExange**: represents an exchange of a CoAP request and response and provides a user-friendly API to subclasses of CoapResource for responding to requests.
* **ResponseCode**: represent a response code of a CoAP response. It can tell us if the communication took place correctly, there was an error or something else happened.

Other classes might have been used but for now, these are enough.

## Our resource

First of all, the server needs at least a resource to expose.

In our case, the resource exposed is a point in the radar space. We need that the point can be represented in the Java language. It can be done creating a new Java class that encapsulates the radar point behavior: RadarPoint, shows below.

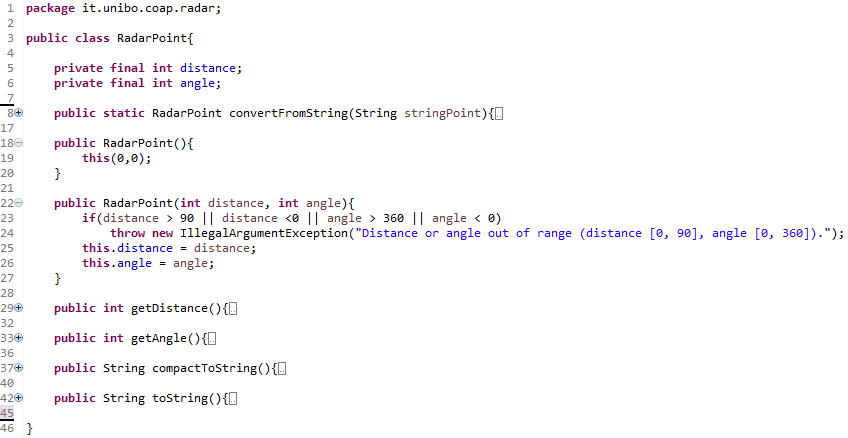


Figure 11: RadarPoint.java – coapServer version 1

The RadarPoint class contains accessor methods to its field (distance and angle) and two utility methods:

* convertFromString(): a static method that transforms a string in a RadarPoint instance (if it is well-formatted).
* compactToString(): an instance method that applies the reverse transformation, from RadarPoint to string.

This class is only a simple Java object, it is not a CoAP resource. The server needs a CoapResource to expose our resource. So, what we can do is to create another Java class, called RadarPointResource that extends CoapResource and encapsulates a RadarPoint object. In that way, the class contains an instance of RadarPoint and can manipulate it every time that receives a REST request.

Looking at the code below, we can see that only GET and PUT requests have behavior associated.

The handleGET() method reply to the client with a string representation of the current RadarPoint instance. The handlePUT() methods reeds the request's payload, convert it into a RadarPoint instance replacing the old one.

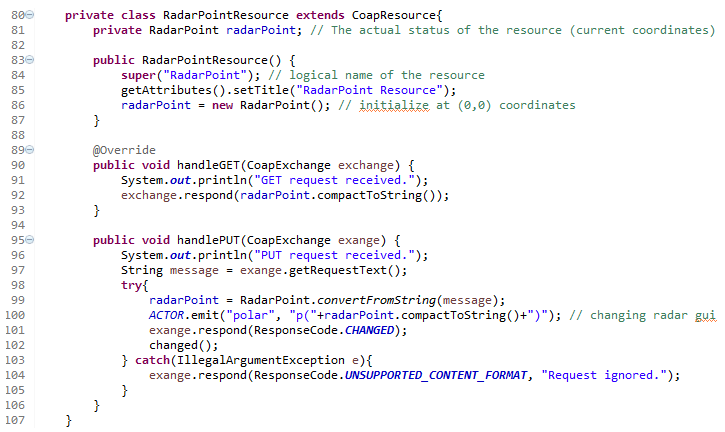


Figure 12: RadarPointResource class in coapRadarServer.java – coapServer version 1

## Server

Now, we are ready to create our CoAP server. A CoAP server, in Californium, must extend the CoapServer class (our implementation is below).

This code shows us that the real coap server is a dedicated Java thread, called workingThread. That because QActor is a single-threaded system. The QActor model must keep running but we need to manage every received request contemporaneously, so we need a dedicated thread.



Figure 13: coapRadarServer.java – coapServer version 1

Now is time to look at the QActor model. It's similar at the previous one but all the server behavior is given to the workingThread. So, the remaining model is simpler than before.

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | **System** coapServer  **Event** polar : p( Distance, Angle )  **Dispatch** stopMessage : stopMessage  **Context** ctxCoapServer **ip** [ **host=**"localhost" **port=**8044 ]  **Context** ctxRadarBase **ip** [ **host=**"localhost" **port=**8033 ] **-standalone**  **QActor** server\_actor **context** ctxCoapServer{    **Plan** init **normal**  [  **println**("radarCoapServer: start.");  **javaRun** it.unibo.radar.coap.server.coapRadarServer.startServer()  ]  **switchTo** running    **Plan** running [ ]  **transition** **stopAfter** 36000000  **whenMsg** stopMessage -> stopping  **Plan** stopping  [  **println**("radarCoapServer: stop.");  **javaRun** it.unibo.radar.coap.server.coapRadarServer.stopServer()  ]  } |

Figure 14: coapServer.qa – coapServer version 1

## Client

To keep an executable formal model (that is easy to start), we continue to use the QActor language to describe the client's model, but we no longer use its communication infrastructure.

Also in that case, a CoAP client must extend or use the CoapClient class of Californium.

Here there is the implementation code:



Figure 15: coapRadarClientSimple.java – coapClient version 1

The putResourceValue() method converts the parameters distance and angle in a RadarPoint object, then sends to the server a PUT request with a string representation of that point in the payload.

The getResourceValue() method sends to the server a GET request and waits for the reply containing the string representation of the coordinates. It transforms the string into a RadarPoint object and then emits an event called value\_event that allows the client actor to receive the point information.

The QActor model is very similar to the previous one.

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55 | **System** coapSimpleClient  **Event** value\_event : value(Distance, Angle)  **Dispatch** value : value(Distance, Angle)  **Dispatch** stopMessage : stopMessage  **Context** ctxCoapSimpleClient **ip** [ **host=**"localhost" **port=**8055 ]  **EventHandler** handlevalue **for** value\_event {  **forwardEvent** client\_simple\_actor **-m** value  };  **QActor** client\_simple\_actor **context** ctxCoapSimpleClient{  **Plan** init **normal**  [  **println**("coapClientSimple: start.");  **javaRun** it.unibo.radar.coap.client.coapRadarClientSimple.initClient()  ]  **switchTo** putValue    **Plan** putValue  [  **println**("coapClientSimple: Emitted PUT: value(45,90).");  **javaRun** it.unibo.radar.coap.client.coapRadarClientSimple.putResourceValue("45", "90")  ]  **switchTo** getValue    **Plan** getValue **resumeLastPlan**  [  **println**("coapClientSimple: Emitted GET.");  **javaRun** it.unibo.radar.coap.client.coapRadarClientSimple.getResourceValue()  ]  **transition** **stopAfter** 2000  **whenMsg** value -> printValue    **Plan** printValue  [  **println**("coapClientSimple: Response GET received.");  **println**("coapClientSimple: Distance = ");  **onMsg** value : value(Distance, Angle) -> **println**(Distance);  **println**("coapClientSimple: Angle = ");  **onMsg** value : value(Distance, Angle) -> **println**(Angle)  ]  **switchTo** stopping    **Plan** stopping  [  **println**("coapClientSimple: stop.")  ]    } |

Figure 16: coapSimpleClient.qa – coapClient version 1

From the model is clear that the client sends a PUT request and just after a GET request in order to test if everything works.

## Manual testing

As we have made before, we can check if the system works correctly starting it and looking into the output console.

coapServer and coapSimpleClient represent two different systems. They must be started separately. We need to start the components in that order:

* RadarDisplay.
* coapServer.
* coapSimpleClient.

Since the sub-systems are two (one client and one server), also the output consoles are two.

These consoles will be like those:

|  |
| --- |
| radarCoapServer: start.  radarCoapServer: started (port=5683).  radarCoapServer: PUT request received.  radarCoapServer: message: 45,90  radarCoapServer: GET request received. |

Figure 17: output console of coapServer – coapServer version 1

|  |
| --- |
| coapClientSimple: start.  coapClientSimple: Emitted PUT: value(45,90).  coapCLientSimple: Resource's value changed.  coapClientSimple: Emitted GET.  coapClientSimple: Response GET received: Distance = 45, Angle = 90  coapClientSimple: stop. |

Figure 18: output console of coapClientSimple – coapClient version 1

Everything works correctly.

# Requirement d): Create a Client GUI controller

The goal of this section is to create a Radar GUI controller that allows the user to control manually GET and PUT operations (client side). While it is necessary to update the client sub-system, the server one will remain unchanged. The Client GUI can be realized in Java.

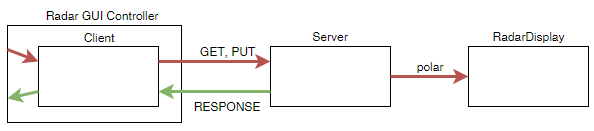


Figure 19: coapClient-v2 scheme system

## The simplified model

The formal model is very small because all the computation is done through the GUI controller.

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | **System** coapClient  **Event** stopMessage : stopMessage // from GUI to client\_actor  **Context** ctxCoapClient **ip** [ **host =** "localhost" **port =** 8055 ]  **QActor** client\_actor **context** ctxCoapClient{    **Plan** init **normal**  [  **println**("radarCoapClient: start.");  **javaRun** it.unibo.radar.gui.radarGUIController.startGUI()  ]  **switchTo** running    **Plan** running [ ]  **transition** **stopAfter** 3600000  **whenEvent** stopMessage -> stopping  **finally** **repeatPlan**    **Plan** stopping  [  **println**("radarCoapClient: stop.")  ]  } |

Figure 20: coapClient.qa - coapClient version 2

## Radar GUI controller

The graphical user interface, implemented in Java, is very simple. It will look like that:

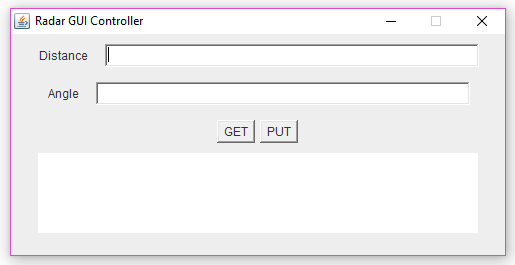


Figure 21: Radar GUI Controller – coapClient version 2

The user can click on the GET button to send a GET request to the server and then its reply will be shown in the bottom text field. Alternatively, the user can insert distance and angle data and click the PUT button to emit a PUT request and send the values to the server. In that way, the user can control manually all the operation that he wants.

The implementation code is here:

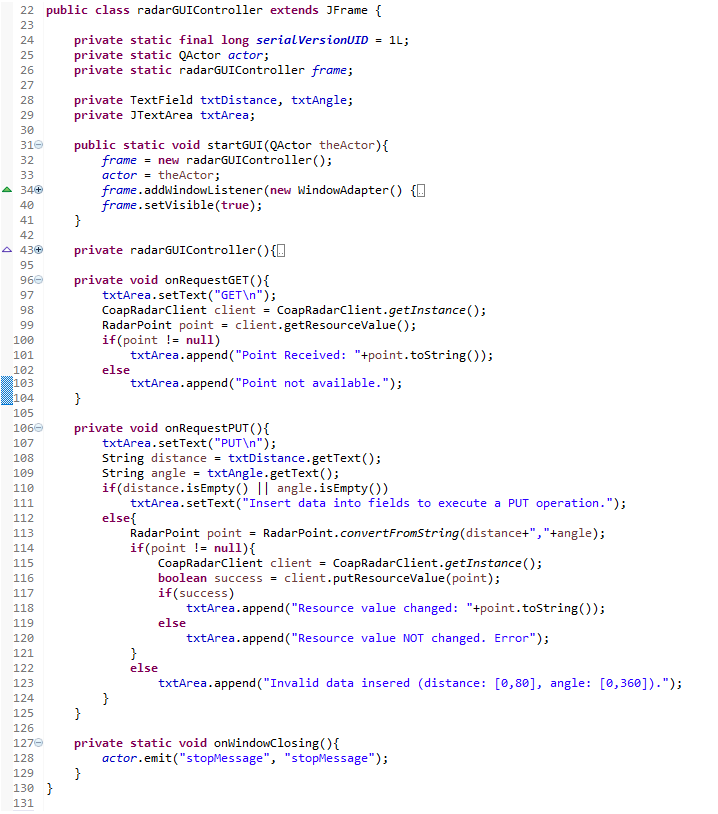


Figure 22: radarGuiController.java – coapClient version 2

## Client update

The CoapRadarClient class has been adapted to be invoked by the GUI. All the QActor references are removed as showed in the following code snippet.

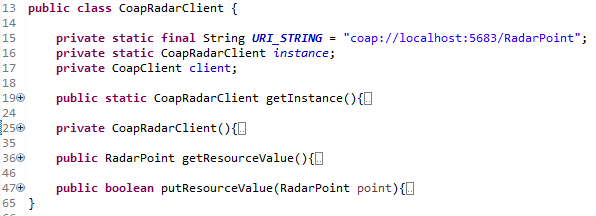


Figure 23: CoapRadarClient.java - coapClient version 2

The QActor model for the client is no longer necessary, it will be removed in the next step.

It's important to note that the CoapRadarClient adopts a synchronous communication model with the server, it means that the client blocks its execution until the response arrives.

## Manual testing

Now we are ready to test the new system (like we did before). We can try to execute a GET request and observe if the response value is the same as displayed into the Radar Display. We can also make a PUT request with some values (for instance 50,90) and observe if the Radar Display changes correctly.

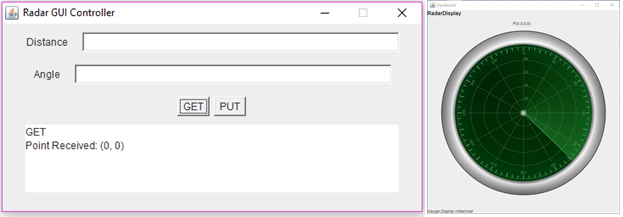


Figure 24: Radar GUI Controller test GET - coapClient version 2

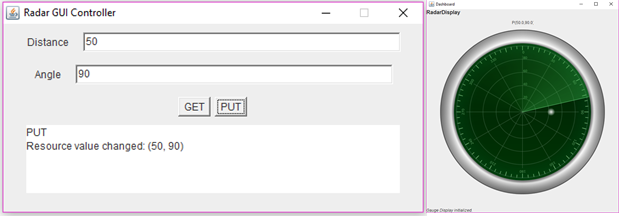


Figure 25: Radar GUI Controller test PUT - coapClient version 2

# Remove all unnecessary QActor infrastructure

In this chapter, we will remove all the unnecessary QActor framework. In particular, we have to remove the QActor infrastructure from the client-side.

## Client-side

We must remove the QActor model (coapClient.qa) and all auto-generated files. Then replace it with a Main class that will start the system. The main class added is very simple, it only starts the GUI. The implementation is here below.

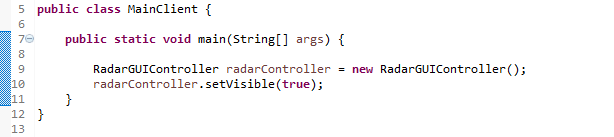


Figure 26: MainClient.java - coapClient version 3

The RadarGuiController class is updated removing all QActor references. The new class' signature is here.

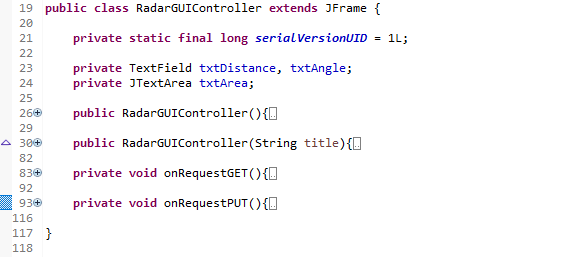


Figure 27: RadarGUIController.java - coapClient version 3

# Requirement e): From sync to async communication

The goal of this chapter is to transform the communication from synchronous to asynchronous.

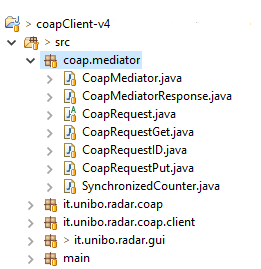
We can realize that introducing a middleware between the client and the server, called CoapMediator. In that way, the client sends the request to the mediator and continues its execution. The middleware receives the client request and forwards it to the server, then waits for the server reply and stores it. When the client wants to receive the response will ask for it to the mediator and if the reply is available it is passed back.

## 

Figure 28: coapClient-v4 scheme system

## The CoapMediator package

The CoAP mediator is an important module of this system. It requires multiple classes to model its behavior.



The class CoapRequest represent a general REST request from the client to the mediator. The classes CoapRequestGET and CoapRequestPUT represent specifications of that class to match the GET and PUT behavior.

The CoapRequestID represents a request identifier. It's used by the mediator to recognize the requests and store their responses.

The CoapMediatorResponse represents a response from the mediator. In the general case (when no error occurs), it contains the server response to the client request. That class gives to the caller information about the response status (arrived, not arrived yet, errors, etc.).

The mediator requires a SynchronizedCounter to generate the request IDs in a safe way in order to assure that all the generated IDs are different and consistent.

The last one is the CoapMediator class, it's the class used by the GUI to execute GET, PUT and RESPONSE requests. For each operation’s type, there is a method that manages the request.

### The CoapMediator class

This class is the most important class in the package because it encapsulates the mediator behavior.

The methods Get() and Put() are very similar. First of all, a CoapRequest object is created and stored, the counter value is increased and a new thread is started. The thread has the goal to contact the server and execute the client request. The interactions between the thread and the server are still synchronous. When the response arrives, it will be stored in the mediator.

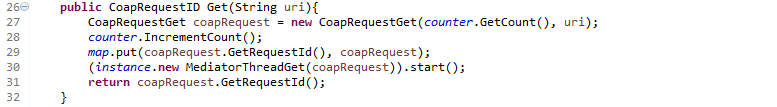


Figure 29: Get() method in CoapMediator.java - coapClient version 4

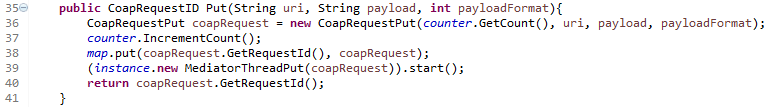


Figure 30: Put() method in CoapMediator.java - coapClient version 4

There are two additional methods in that class. The first one is GetResponse(), directly invoked by the GUI to retrieve the CoAP response. It creates a CoapMediatorResponse containing the server response if exists. When the response is not available yet or an error occurs, an error response code is returned. The second method is RegisterResponse() that allows the mediator’s threads to store the server response when it arrives.

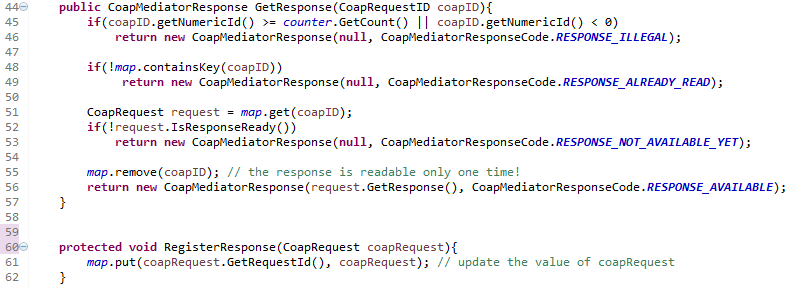


Figure 31: GetResponse() and RegisterResponse() methods in CoapManager.java - coapClient version 4

### The MediatorThead classes

Since the mediator can receive more requests simultaneously, a multi-threaded implementation gives better results. For each REST request (GET or PUT) a dedicated thread is created. It will manage a single request.

Another solution could be to create a thread poll, in that way each thread can manage more than one request (one at a time) decreasing the overhead caused by creation and destruction threads.

These classes work as the GUI buttons of the previous system.

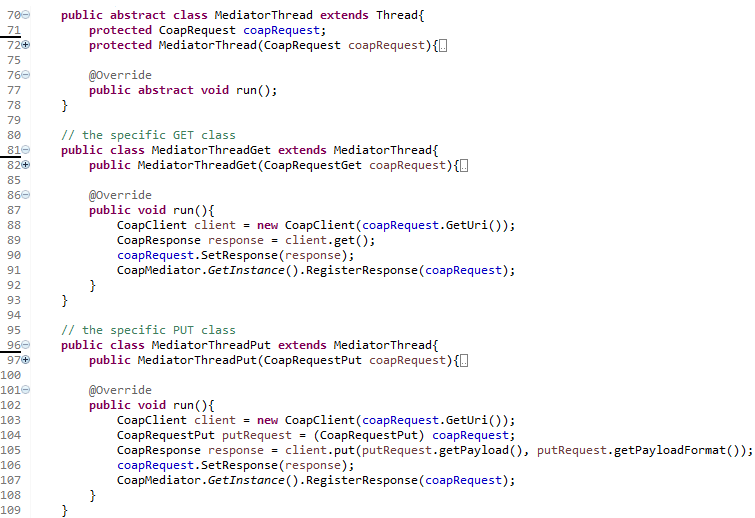


Figure 32: MediatorThread classes in CoapMediator.java - coapClient version 4

## Radar GUI Controller

The Radar GUI Controller has to be updated to allow the user to obtain a response later.

The new interface is that:

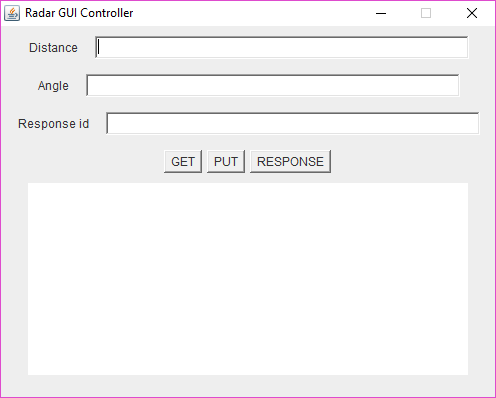


Figure 33: Radar GUI Controller- coapClient version 4

A new text field and button are added. The user can insert the request identifier in the field and pushing the button can receive the server reply (if it is already available).

Now, we have to change the methods attached to the buttons. So that, instead of contact directly the server, they will invoke the CoapMediator in the way showed below.

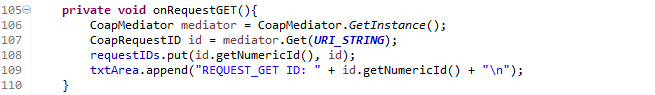


Figure 34: onRequestGET() in RadarGUIContoller.java - coapClient version 4

This method is invoked when the user clicks on the GET button. The method obtains a CoapMediator instance, delegates to the mediator the execution of a GET request and saves the corresponding request identifier (that will be used from the user to ask for the request's response).

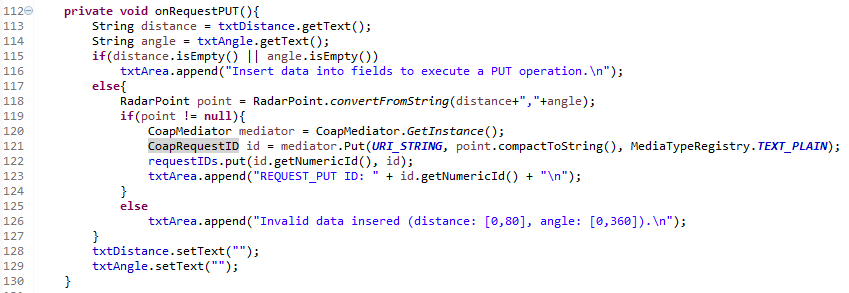


Figure 35: onRequestPUT() in RadarGUIController.java - coapClient version 4

The onRequestPUT method takes the distance and angle values from the GUI and invokes the mediator for a PUT request. Also in that case, it saves the request identifier.

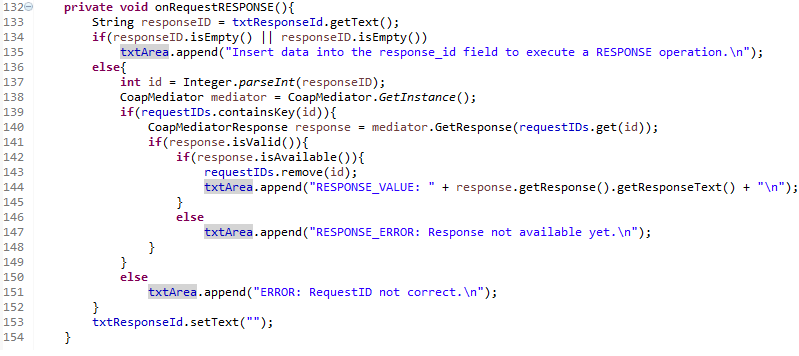


Figure 36: onRequestRESPONSE() in RadarGUIClient.java - coapClient version 4

The onRequestRESPONSE() method is invoked when the user inserts on the GUI a request ID and presses the RESPONSE button. If the ID is correct, the GUI asks for the response to the mediator. The latter replies with the received answer, otherwise with a message error.

# CoapMediator as an independent sub-system

In the last version of our system, the CoAP mediator can communicate with any CoAP server, this is not true with respect to CoAP clients.

We will allow any client to contact it and ask for async requests. This can simply be made splitting up the current system into two different sub-systems: the mediator and the client sub-systemS. To allow clients to contact it, we can use TCP connections that can be created in all high-level programming languages.

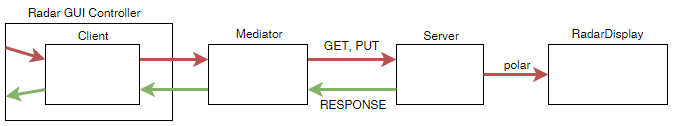


Figure 37: coapClient-v5 and coapMediator-v1 scheme systems

## Creating the new CoapMediator system

The first thing that we have to do is create another Java project. Then we have to copy the whole package *coap.mediator*. Then a main class must be created. In that class, the mediator has to expose a TCP server to receive client’s TCP connections.

The main method creates a TCP listener socket and waits for connections. For each connection, it detects which type the request is and starts a dedicated thread (like the CoAP server does).

The new system is completed and doesn't require anything else.

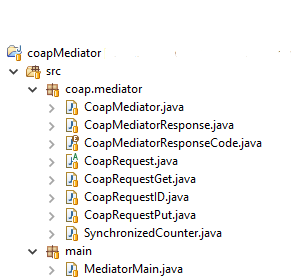


Figure 38: mediator sub-system - coapMediator version 1

## Updating the client

In that system version, a new CoapMediatorClient is created. It encapsulates all the client TCP communication code (GET, PUT, RESPONSE) to simplify the RadarGUIController.

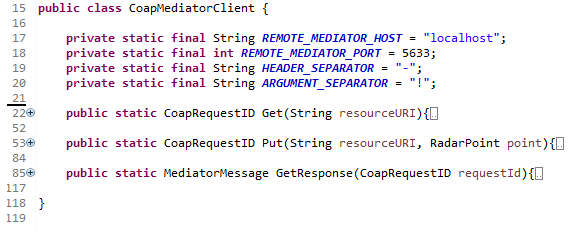


Figure 39: CoapMediatorClient.java - coapClient version 5

We can notice that the GetResponse() method returns a MediatorMessage object. It is a new class that represents a compact version of a CoapMediatorResponse, used to communicate information between the mediator client and the GUI. It contains the mediator response code and the response message.

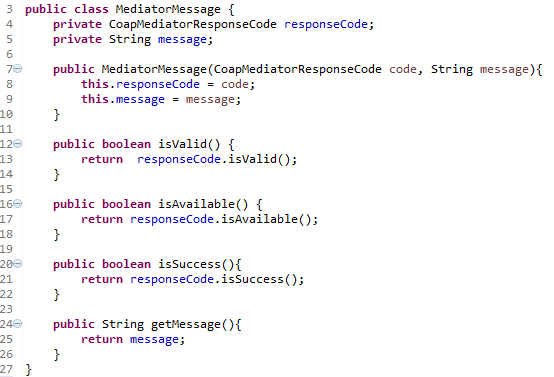


Figure 40: MediatorMessage.java - coapClient version 5

Finally, the GUI must be updated in order to use the CoapMediatorClient instead of creating TCP connections by itself.

# Conclusion

This project work is finally terminated. We have achieved the project's aim and proved that the CoAP protocol can be easily integrated with the QActor framework. Our final project can be used as a Java library to allow the framework to communicate by CoAP.

At the moment, only GET and PUT operations are available. But in the future, the library can be upgraded to admit other REST operation like DELETE, HEAD and POST.

In this project we have implemented only two of the three communication types allowed by Californium, the synchronous and the asynchronous one. The third type is the observable communication, in that communication the client register itself as an observer of the resource. In that way, it will be notified when the resource’s state changes. This feature can be implemented in a future version of that library.