

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 with the resource. In that way, the client will be notified when the resource change.

# 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:

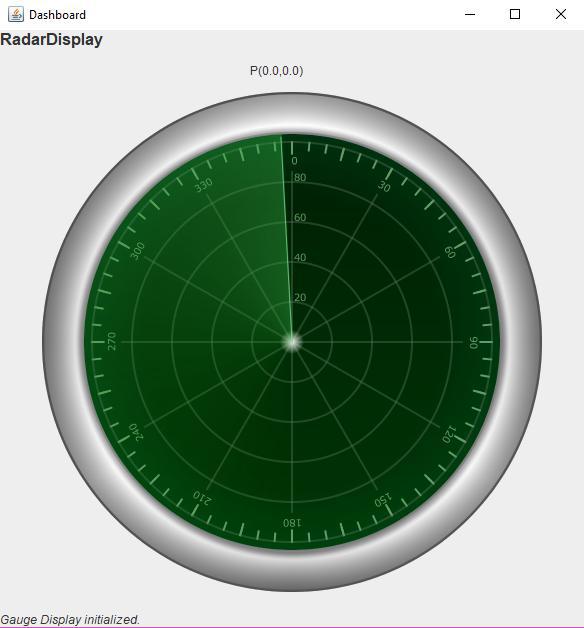


Figure : RadarDisplay GUI

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 : 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. Remove all unnecessary QActor interaction.
6. Allow async Get and Put requests.

# 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.

PolarSender ---polar---> RadarDisplay

## 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 : 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 its (the communication complexity) the complexity. What we’ll do from now will be remove (quello che faremo d’ora in avanti sarà rimuovere) 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 is carried out (venga svolta) anyway 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.

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 4: 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.

Now let's see (più formale) the server QActor model.

|  |  |
| --- | --- |
| 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 5: 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 having managed (dopo aver gestito) the request, it returns in the waiting state creating an infinite loop.

## Client

Il client rappresenta un client CoAP, il quale fa uso delle primitive di comunicazione GET e PUT per ricevere ed inviare le coordinate del radar. Esso non interagisce direttamente con il radar, ma lo fa indirettamente tramite il server CoAP. Vediamone il modello formale:

|  |  |
| --- | --- |
| 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 6: client actor in coapSystem.qa – coapRadar version 0

The client 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.

## Manual testing

To test if everything works fine, we can run the model and look at the output console what 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 7: 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 the current point position is correctly updated when a PUT request arrives and is send out when a GET request arrives (suona malissimo, da riformulare).

# 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).

## 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 (potrebbero essere state usate) 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.

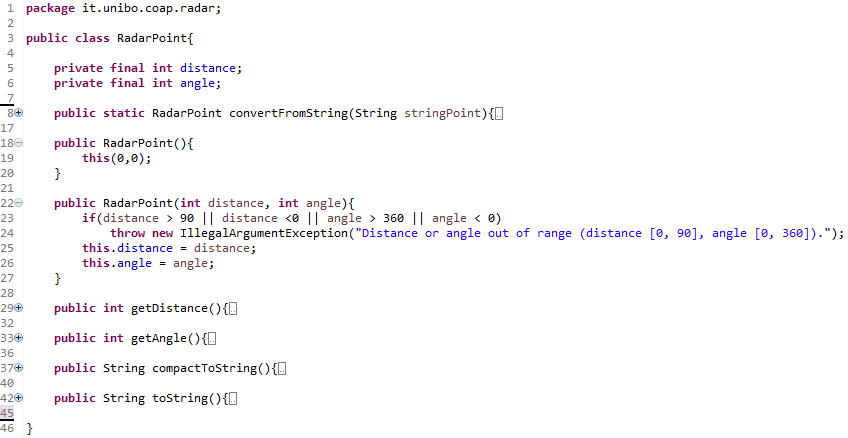


Figure 8: RadarPoint.java – coapServer version 1

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.

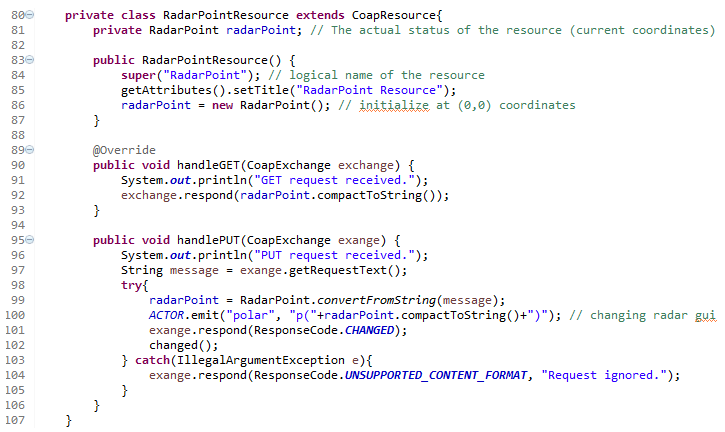


Figure 9: RadarPointResource class in coapRadarServer.java – coapServer version 1

## 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 (molto brutta).



Figure : 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 11: 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 12: 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 request 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 13: coapSimpleClient.qa – coapClient version 1

From the model is clear that the client sends a PUT request and immediately 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 consols 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 14: 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 15: 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.

## 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 : coapClient.qa - coapClient version 2

## Radar GUI controller

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

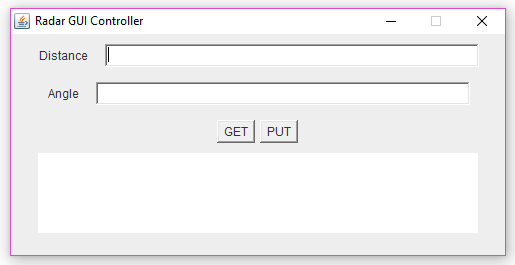


Figure 17: Radar GUI Controller – coapClient version 2

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:

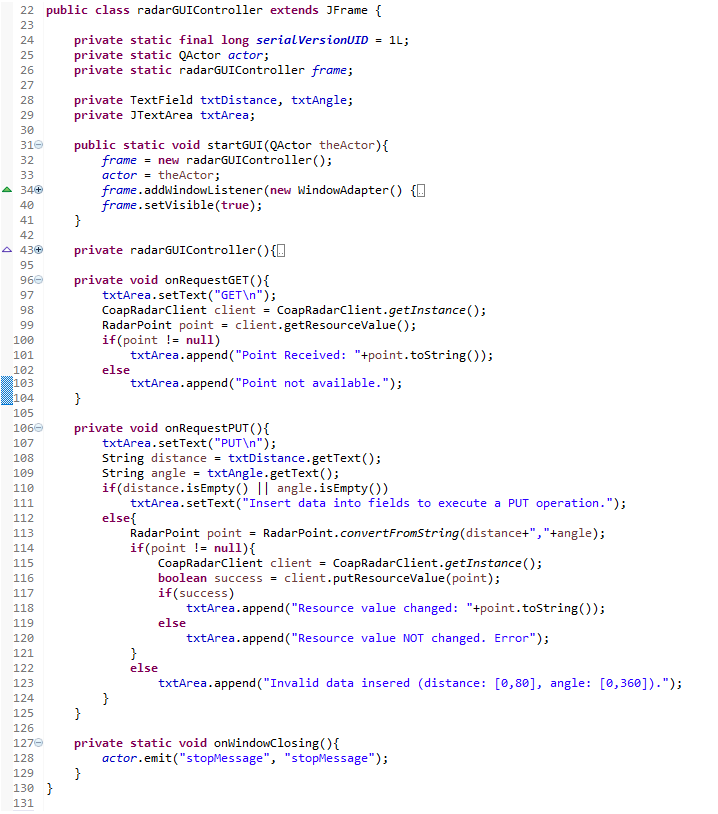


Figure : radarGuiController.java – coapClient version 2

## 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 image.

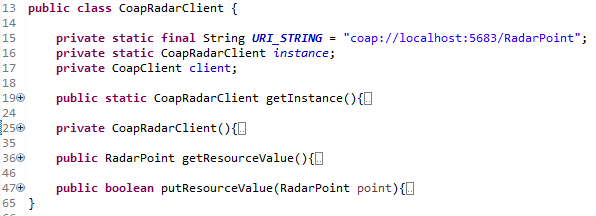


Figure : CoapRadarClient.java - coapClient version 2

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

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. 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.

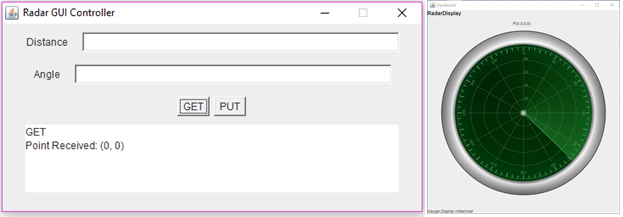


Figure 20: Radar GUI Controller test GET - coapClient version 2

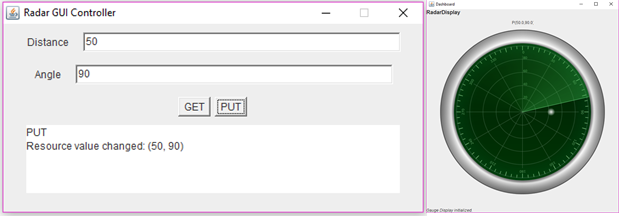


Figure 21: Radar GUI Controller test PUT - coapClient version 2

# Requirement e) Remove all unnecessary QActor infrastructure

# Problematiche legate all’uso di più request

* Far riferimento all’esempio precedente per ipotizzare una scena dove vengano inviate più request e si voglia ottenere una response associata ad una determinata request inviata. Nasce l’esigenza di un componente dedicato che faccia da intermediario tra il client ed il server.
* Mostrare una prima ipotetica soluzione per risolvere questo problema (int, object, etc).