The CoAP protocol

Project Work on "Ingegneria dei Sistemi Software M"

A.A. 2018-2019

KEVIN LETO

1   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.

2   CoAP e Californium

2.1   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).

2.1.1   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).

2.2   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.

2.2.1   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 call-back 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.

3 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 centred in (0,0). The point is identified by a couple of coordinates (x, y).
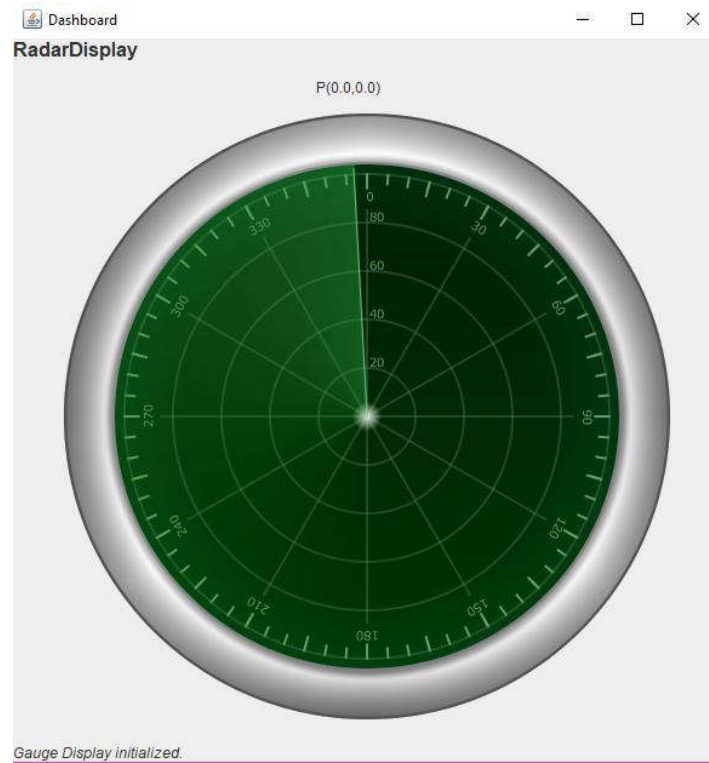
That is the Radar Display GUI:

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

3.1 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.

3.2 REQUIREMENTS

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

a) Create a basic architecture that allows to send coordinate to the Radar Display (with the QActor infrastructure).

b) Create a Client-server architecture (with QActor).

c) Create a client-server model that implements Get and Put requests with CoAP.

d) Create a Client GUI controller.

e) From sync to async communication.

4 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 system's scheme

4.1 FIRST FORMAL MODEL - POLARSENDER

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

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

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).

5 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.
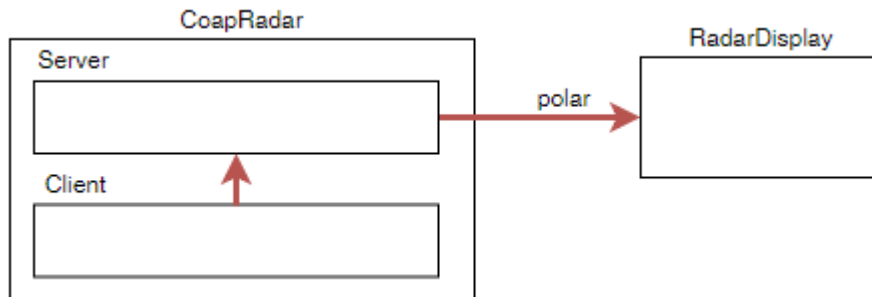

Figure 5: coapRadar-v0 system's scheme

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

Figure 6: event and dispatch descriptions in coapSystem.qa – coapRadar v0

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.

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

Figure 7: server actor in coapSystem.qa – coapRadar v0

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.

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

Figure 8: client actor in coapSystem.qa – coapRadar v0

5.3 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 v0

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.

6 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).
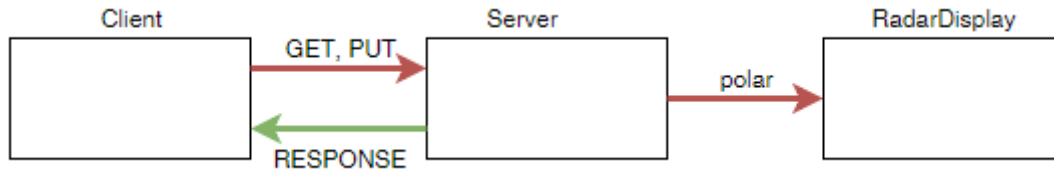

Figure 10: coapClient-v1 and coapServer-v1 systems' scheme

6.1 CALIFORNIUM: THE FUNDAMENTALS

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.

6.2 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 behaviour: RadarPoint, shows below.

```
 1  package it.unibo.coap.radar;
 2
 3  public class RadarPoint{
 4
 5      private final int distance;
 6      private final int angle;
 7
 8⊕     public static RadarPoint convertFromString(String stringPoint){⬚}
17
18⊖     public RadarPoint(){
19          this(0,0);
20      }
21
22⊖     public RadarPoint(int distance, int angle){
23          if(distance > 90 || distance <0 || angle > 360 || angle < 0)
24              throw new IllegalArgumentException("Distance or angle out of range (distance [0, 90], angle [0, 360]).");
25          this.distance = distance;
26          this.angle = angle;
27      }
28
29⊕     public int getDistance(){⬚}
32
33⊕     public int getAngle(){⬚}
36
37⊕     public String compactToString(){⬚}
40
42⊕     public String toString(){⬚}
45
46  }
```

Figure 11: RadarPoint.java – coapServer v1

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 behaviour 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.

```
80    private class RadarPointResource extends CoapResource{
81        private RadarPoint radarPoint; // The actual status of the resource (current coordinates)
82
83        public RadarPointResource() {
84            super("RadarPoint"); // logical name of the resource
85            getAttributes().setTitle("RadarPoint Resource");
86            radarPoint = new RadarPoint(); // initialize at (0,0) coordinates
87        }
88
89        @Override
90        public void handleGET(CoapExchange exchange) {
91            System.out.println("GET request received.");
92            exchange.respond(radarPoint.compactToString());
93        }
94
95        public void handlePUT(CoapExchange exange) {
96            System.out.println("PUT request received.");
97            String message = exange.getRequestText();
98            try{
99                radarPoint = RadarPoint.convertFromString(message);
100               ACTOR.emit("polar", "p("+radarPoint.compactToString()+")"); // changing radar gui
101               exange.respond(ResponseCode.CHANGED);
102               changed();
103           } catch(IllegalArgumentException e){
104               exange.respond(ResponseCode.UNSUPPORTED_CONTENT_FORMAT, "Request ignored.");
105           }
106       }
107   }
```

Figure 12: RadarPointResource class in coapRadarServer.java – coapServer v1

6.3 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.

```
38  public class coapRadarServer extends CoapServer{
39      private static QActor ACTOR;
40      private static final int COAP_PORT = NetworkConfig.getStandard().getInt(NetworkConfig.Keys.COAP_PORT);
41      private static Thread workingThread;
42
43      private coapRadarServer() throws SocketException { add(new RadarPointResource()); }
44
45⊖     public static void startServer(QActor actor) {
46          coapRadarServer.ACTOR = actor;
47
48⊖         workingThread = new Thread("Working thread") {
49⊖             public void run(){
50                  try {
51                      coapRadarServer server = new coapRadarServer(); // create coap server
52                      server.addEndpoints(); // add endpoints on all IP addresses
53                      server.start();
54                      System.out.println("CoapRadarServer started (port="+COAP_PORT+").");
55                  } catch (SocketException e) {
56                      System.err.println("Failed to initialize server: " + e.getMessage());
57                  }
58              }
59          };
60          workingThread.start();
61      }
62
63⊖     public static void stopServer(QActor actor){
64          if(workingThread != null){
65              workingThread.interrupt();
66              workingThread = null;
67          }
68      }
69
70⊕     private void addEndpoints() {⊡}
79
80⊕     private class RadarPointResource extends CoapResource{⊡}
108
109 }
```

Figure 13: coapRadarServer.java – coapServer v1

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

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

```
26          ]
27  }
28
```

Figure 14: coapServer.qa – coapServer v1

6.4 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:

```java
15  public class coapRadarClientSimple {
16
17      public static final String URI_STRING = "coap://localhost:5683/RadarPoint";
18      private static CoapClient client;
19
20      public static void initClient(QActor actor){
21          URI uri = null; // URI parameter of the request
22          try {
23              uri = new URI(URI_STRING);
24          } catch (URISyntaxException e) {
25              System.err.println("Invalid URI: " + e.getMessage());
26              System.exit(-1);
27          }
28          client = new CoapClient(uri);
29      }
30
31      public static void getResourceValue(QActor actor){
32          CoapResponse response = client.get();
33          if (response != null) {
34              RadarPoint point = RadarPoint.convertFromString(response.getResponseText());
35              actor.emit("value_event", "value("+point.getDistance()+","+point.getAngle()+")");
36          } else {
37              System.out.println("No response received.");
38          }
39
40      }
41
42      public static void putResourceValue(QActor actor, String distance, String angle){
43          RadarPoint point = RadarPoint.convertFromString(distance+","+angle);
44          if(point != null){
45              CoapResponse response = client.put(point.compactToString(), MediaTypeRegistry.TEXT_PLAIN);
46              if(response != null) {
47                  if(response.getCode() == ResponseCode.CHANGED)
48                      System.out.println("Resource's value changed.");
49                  else
50                      System.out.println("Resource's value NOT changed.");
51              }
52              else
53                  System.out.println("No response received.");
54          }
55          else
56              System.out.println("Invalid RadarPoint");
57      }
58  }
59
```

Figure 15: coapRadarClientSimple.java – coapClient v1

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

Figure 16: coapSimpleClient.qa – coapClient v1

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.

6.5 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 v1

```
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 v1

Everything works correctly.

7 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.
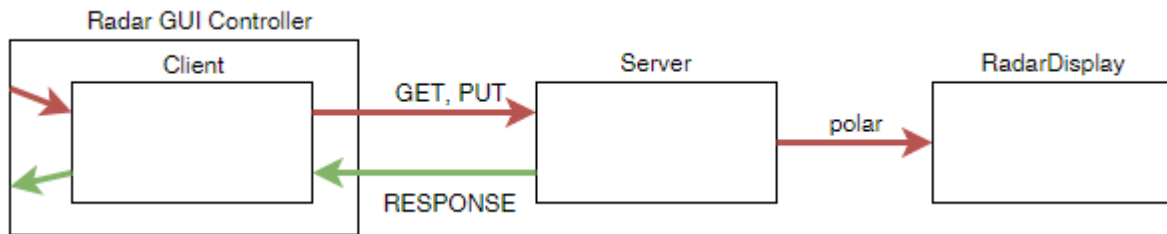

Figure 19: coapClient-v2 scheme system

7.1 THE SIMPLIFIED MODEL

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

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

Figure 20: coapClient.qa - coapClient v2

7.2 RADAR GUI CONTROLLER

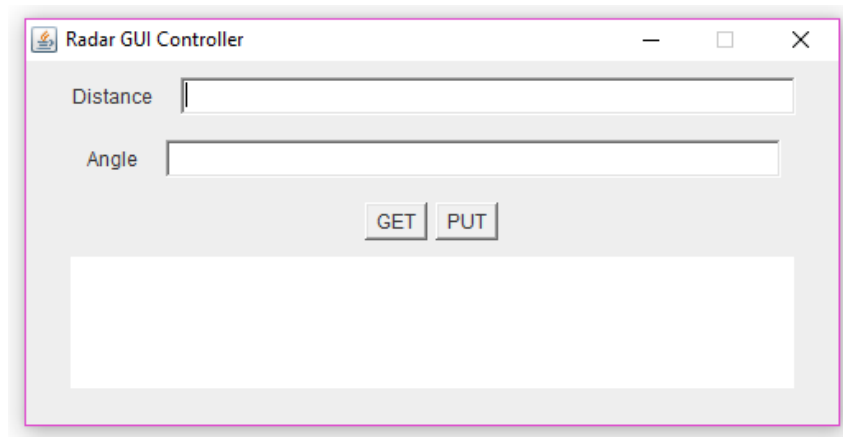The graphical user interface, implemented in Java, is very simple. It will look like that:

Figure 21: Radar GUI Controller – coapClient v2

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:

```java
22  public class radarGUIController extends JFrame {
23
24      private static final long serialVersionUID = 1L;
25      private static QActor actor;
26      private static radarGUIController frame;
27
28      private TextField txtDistance, txtAngle;
29      private JTextArea txtArea;
30
31      public static void startGUI(QActor theActor){
32          frame = new radarGUIController();
33          actor = theActor;
34          frame.addWindowListener(new WindowAdapter() {
40          frame.setVisible(true);
41      }
42
43      private radarGUIController(){
95
96      private void onRequestGET(){
97          txtArea.setText("GET\n");
98          CoapRadarClient client = CoapRadarClient.getInstance();
99          RadarPoint point = client.getResourceValue();
100         if(point != null)
101             txtArea.append("Point Received: "+point.toString());
102         else
103             txtArea.append("Point not available.");
104     }
105
106     private void onRequestPUT(){
107         txtArea.setText("PUT\n");
108         String distance = txtDistance.getText();
109         String angle = txtAngle.getText();
110         if(distance.isEmpty() || angle.isEmpty())
111             txtArea.setText("Insert data into fields to execute a PUT operation.");
112         else{
113             RadarPoint point = RadarPoint.convertFromString(distance+","+angle);
114             if(point != null){
115                 CoapRadarClient client = CoapRadarClient.getInstance();
116                 boolean success = client.putResourceValue(point);
117                 if(success)
118                     txtArea.append("Resource value changed: "+point.toString());
119                 else
120                     txtArea.append("Resource value NOT changed. Error");
121             }
122             else
123                 txtArea.append("Invalid data insered (distance: [0,80], angle: [0,360]).");
124         }
125     }
126
127     private static void onWindowClosing(){
128         actor.emit("stopMessage", "stopMessage");
129     }
130 }
131
```

Figure 22: radarGuiController.java – coapClient v2

7.3 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.

```
13  public class CoapRadarClient {
14
15      private static final String URI_STRING = "coap://localhost:5683/RadarPoint";
16      private static CoapRadarClient instance;
17      private CoapClient client;
18
19⊕     public static CoapRadarClient getInstance(){..
24
25⊕     private CoapRadarClient(){..
35
36⊕     public RadarPoint getResourceValue(){..
46
47⊕     public boolean putResourceValue(RadarPoint point){..
65  }
```

Figure 23: CoapRadarClient.java - coapClient v2

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.

7.4 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.
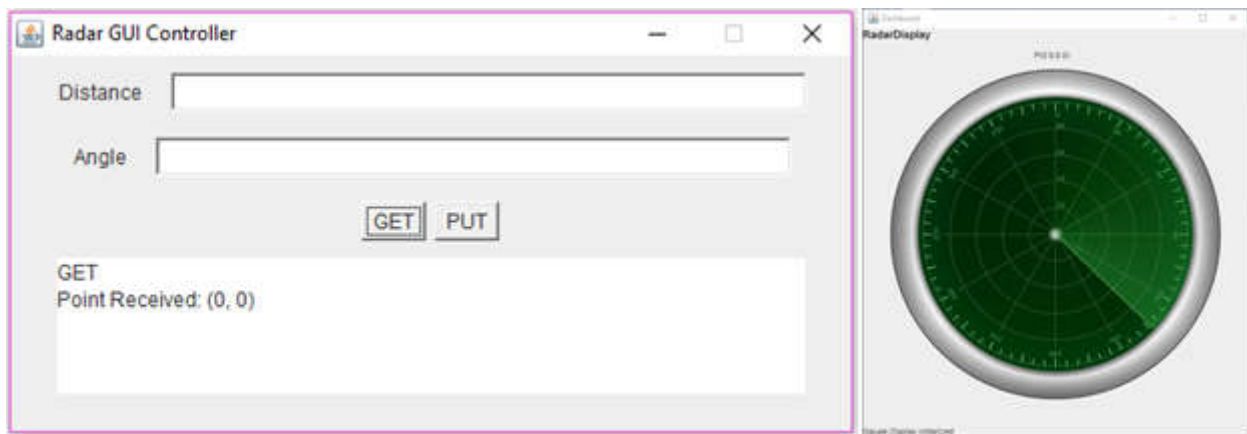

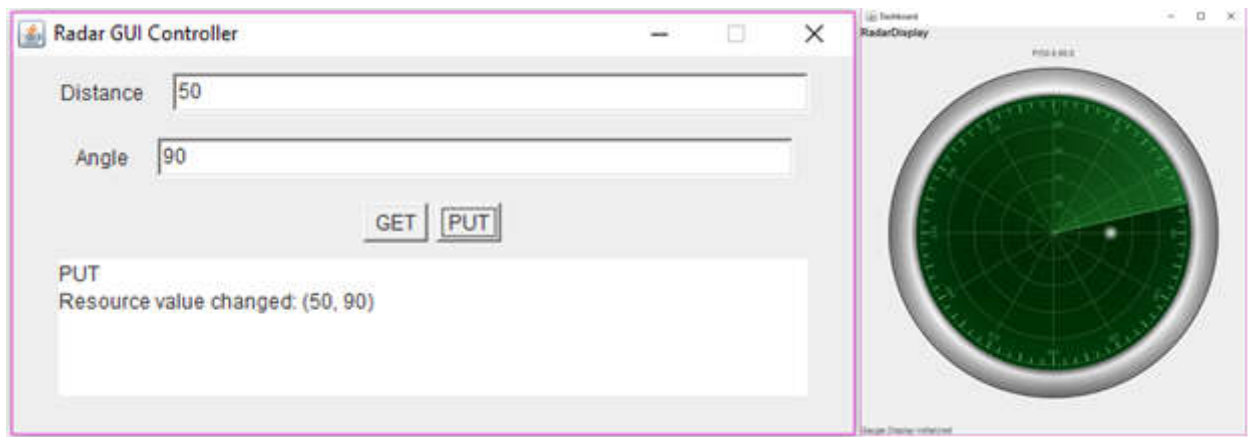*Figure 24: Radar GUI Controller test GET - coapClient v2*

Figure 25: Radar GUI Controller test PUT - coapClient v2

8 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.

8.1 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.

```
 5  public class MainClient {
 6
 7⊖      public static void main(String[] args) {
 8
 9          RadarGUIController radarController = new RadarGUIController();
10          radarController.setVisible(true);
11      }
12  }
13
```

Figure 26: MainClient.java - coapClient v3

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

```
19  public class RadarGUIController extends JFrame {
20
21      private static final long serialVersionUID = 1L;
22
23      private TextField txtDistance, txtAngle;
24      private JTextArea txtArea;
25
26⊕      public RadarGUIController(){⬚
29
30⊕      public RadarGUIController(String title){⬚
82
83⊕      private void onRequestGET(){⬚
92
93⊕      private void onRequestPUT(){⬚
116
117  }
118
```

Figure 27: RadarGUIController.java - coapClient version 3

9 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.
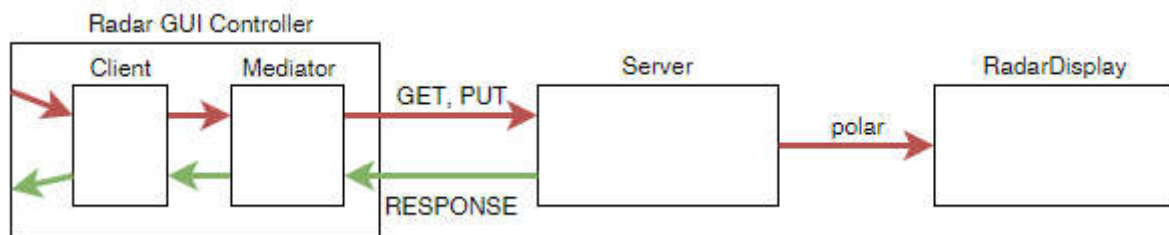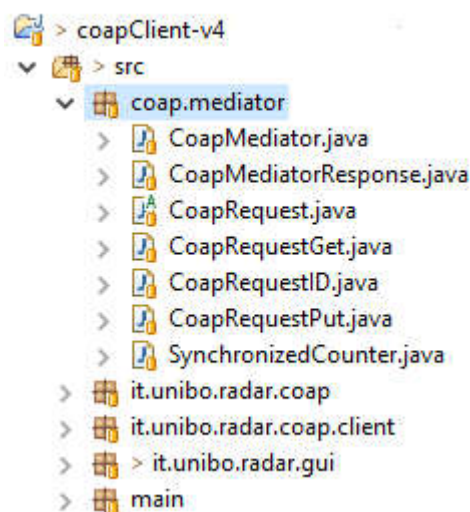

Figure 28: coapClient-v4 scheme system

9.1 THE COAPMEDIATOR PACKAGE

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

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.

9.1.1 The CoapMediator class

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

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.

```java
26  public CoapRequestID Get(String uri){
27      CoapRequestGet coapRequest = new CoapRequestGet(counter.GetCount(), uri);
28      counter.IncrementCount();
29      map.put(coapRequest.GetRequestId(), coapRequest);
30      (instance.new MediatorThreadGet(coapRequest)).start();
31      return coapRequest.GetRequestId();
32  }
```

Figure 29: Get() method in CoapMediator.java - coapClient v4

```java
35  public CoapRequestID Put(String uri, String payload, int payloadFormat){
36      CoapRequestPut coapRequest = new CoapRequestPut(counter.GetCount(), uri, payload, payloadFormat);
37      counter.IncrementCount();
38      map.put(coapRequest.GetRequestId(), coapRequest);
39      (instance.new MediatorThreadPut(coapRequest)).start();
40      return coapRequest.GetRequestId();
41  }
```

Figure 30: Put() method in CoapMediator.java - coapClient v4

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.

```java
44  public CoapMediatorResponse GetResponse(CoapRequestID coapID){
45      if(coapID.getNumericId() >= counter.GetCount() || coapID.getNumericId() < 0)
46          return new CoapMediatorResponse(null, CoapMediatorResponseCode.RESPONSE_ILLEGAL);
47
48      if(!map.containsKey(coapID))
49          return new CoapMediatorResponse(null, CoapMediatorResponseCode.RESPONSE_ALREADY_READ);
50
51      CoapRequest request = map.get(coapID);
52      if(!request.IsResponseReady())
53          return new CoapMediatorResponse(null, CoapMediatorResponseCode.RESPONSE_NOT_AVAILABLE_YET);
54
55      map.remove(coapID); // the response is readable only one time!
56      return new CoapMediatorResponse(request.GetResponse(), CoapMediatorResponseCode.RESPONSE_AVAILABLE);
57  }
58
59
60  protected void RegisterResponse(CoapRequest coapRequest){
61      map.put(coapRequest.GetRequestId(), coapRequest); // update the value of coapRequest
62  }
```

Figure 31: GetResponse() and RegisterResponse() methods in CoapManager.java - coapClient v4

9.1.2 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 did in the previous system.

```java
 7  public abstract class MediatorThread extends Thread {
 8      // In Message format: request_type HEADER_SEPARATOR argument1 ARGUMENT_SEPARATOR argument2 ...
 9      // Change those separators carfully! There are trouble if they are present in the message text.
10      public static final String HEADER_SEPARATOR = "-";
11      public static final String ARGUMENT_SEPARATOR = "!";
12
13      protected Socket clientSocket;
14      protected DataOutputStream out;
15      protected String uri;
16
17      public MediatorThread(Socket clientSocket, String uri) throws IOException{
18          this.clientSocket = clientSocket;
19          this.out = new DataOutputStream(clientSocket.getOutputStream());
20          this.uri = uri;
21      }
22
23      @Override
24      public abstract void start();
25  }
```

Figure 32: MediatorThread.java - coapClient v4

```java
 9  public class GetMediatorThread extends MediatorThread{
10
11      public GetMediatorThread(Socket clientSocket, String uri) throws IOException {
12          super(clientSocket, uri);
13      }
14
15      @Override
16      public void start() {
17          // out message format: "REQUEST_ID" HEADER_SEPARATOR id
18
19          CoapRequestID id = CoapMediator.Get(uri);
20          System.out.println("GetThread started.");
21          String message = "REQUEST_ID"+HEADER_SEPARATOR+id.getNumericId();
22          try{
23              out.writeUTF(message);
24              System.out.println("Message --> "+message);
25              clientSocket.close();
26          } catch(Exception e){
27              System.out.println("GetThread: out socket error.");
28          }
29      }
30  }
```

Figure 33GetMediatorThread.java - coapClient v4

```java
 9  public class PutMediatorThread extends MediatorThread{
10
11      private String payload;
12      private int payloadFormat;
13
14⊖     public PutMediatorThread(Socket clientSocket, String uri,String payload, int payloadFormat) throws IOException {
15          super(clientSocket, uri);
16          this.payload = payload;
17          this.payloadFormat = payloadFormat;
18      }
19
20⊖     @Override
21      public void start() {
22          // out message format: "REQUEST_ID" HEADER_SEPARATOR id
23
24          CoapRequestID id = CoapMediator.Put(uri, payload, payloadFormat);
25          System.out.println("PutThread started.");
26          String message = "REQUEST_ID"+HEADER_SEPARATOR+id.getNumericId();
27          try{
28              out.writeUTF(message);
29              System.out.println("Message --> "+message);
30              clientSocket.close();
31          } catch(Exception e){
32              System.out.println("GetThread: out socket error.");
33          }
34      }
35  }
```

Figure 34: PutMediatorThread.java - coapClient v4

```java
10  public class ResponseMediatorThread extends MediatorThread{
11
12      private int requestId;
13
14⊖     public ResponseMediatorThread(Socket clientSocket, int requestId, String uri) throws IOException {
15          super(clientSocket, uri);
16          this.requestId = requestId;
17      }
18
19⊖     @Override
20      public void start() {
21          // out message format:  "RESPONSE" HEADER_SEPARATOR responseCode ARGUMENT_SEPARATOR responseText/failureMessage
22
23          CoapMediatorResponse response = null;
24          CoapRequestID id = new CoapRequestID(requestId, uri);
25          System.out.println("ResponseThread started.");
26          try{
27              response = CoapMediator.GetResponse(id);
28              String message = "RESPONSE"+HEADER_SEPARATOR;
29
30              message += response.getResponseCode() + ARGUMENT_SEPARATOR;
31
32              if(response.isValid() && response.isSuccess())
33                  message += response.getResponse().getResponseText();
34              else
35                  message += response.getResponseCode().getDescription();
36
37              out.writeUTF(message);
38              System.out.println("Message --> "+message);
39              System.out.println("ResponseThread stopped.");
40              clientSocket.close();
41          } catch(Exception e){
42              System.out.println("ResponseThread: out socket error.");
43              e.printStackTrace();
44          }
45      }
46  }
```

Figure 35: ResponseMediatorThread.java - coapClient v4

9.2 RADAR GUI CONTROLLER

The Radar GUI Controller has to be updated to allow the user to obtain a response later.
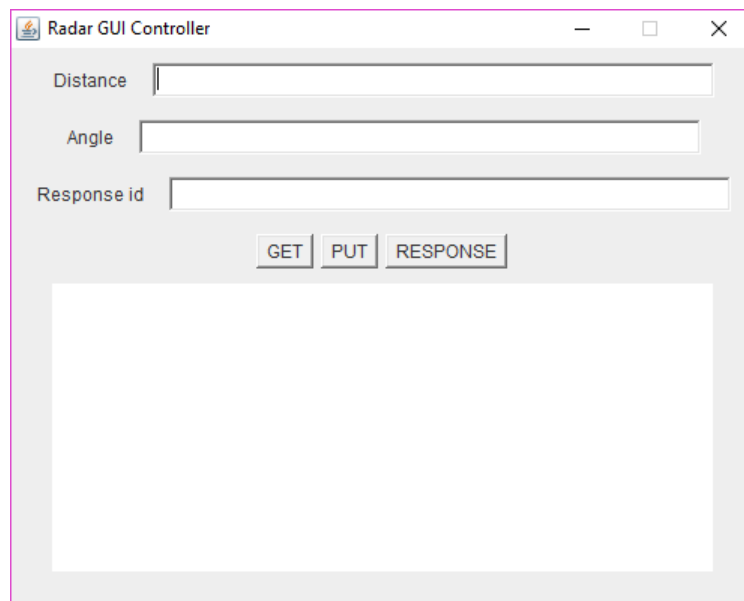
The new interface is that:

Figure 36: Radar GUI Controller- coapClient v4

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.

```
105⊖    private void onRequestGET(){
106         CoapMediator mediator = CoapMediator.GetInstance();
107         CoapRequestID id = mediator.Get(URI_STRING);
108         requestIDs.put(id.getNumericId(), id);
109         txtArea.append("REQUEST_GET ID: " + id.getNumericId() + "\n");
110     }
```

Figure 37: onRequestGET() in RadarGUIContoller.java - coapClient v4

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).

```
112⊖    private void onRequestPUT(){
113         String distance = txtDistance.getText();
114         String angle = txtAngle.getText();
115         if(distance.isEmpty() || angle.isEmpty())
116             txtArea.append("Insert data into fields to execute a PUT operation.\n");
117         else{
118             RadarPoint point = RadarPoint.convertFromString(distance+","+angle);
119             if(point != null){
120                 CoapMediator mediator = CoapMediator.GetInstance();
121                 CoapRequestID id = mediator.Put(URI_STRING, point.compactToString(), MediaTypeRegistry.TEXT_PLAIN);
122                 requestIDs.put(id.getNumericId(), id);
123                 txtArea.append("REQUEST_PUT ID: " + id.getNumericId() + "\n");
124             }
125             else
126                 txtArea.append("Invalid data insered (distance: [0,80], angle: [0,360]).\n");
127         }
128         txtDistance.setText("");
129         txtAngle.setText("");
130     }
```

Figure 38: onRequestPUT() in RadarGUIController.java - coapClient v4

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.

```
132⊖    private void onRequestRESPONSE(){
133         String responseID = txtResponseId.getText();
134         if(responseID.isEmpty() || responseID.isEmpty())
135             txtArea.append("Insert data into the response_id field to execute a RESPONSE operation.\n");
136         else{
137             int id = Integer.parseInt(responseID);
138             CoapMediator mediator = CoapMediator.GetInstance();
139             if(requestIDs.containsKey(id)){
140                 CoapMediatorResponse response = mediator.GetResponse(requestIDs.get(id));
141                 if(response.isValid()){
142                     if(response.isAvailable()){
143                         requestIDs.remove(id);
144                         txtArea.append("RESPONSE_VALUE: " + response.getResponse().getResponseText() + "\n");
145                     }
146                     else
147                         txtArea.append("RESPONSE_ERROR: Response not available yet.\n");
148                 }
149             }
150             else
151                 txtArea.append("ERROR: RequestID not correct.\n");
152         }
153         txtResponseId.setText("");
154     }
```

Figure 39: onRequestRESPONSE() in RadarGUIClient.java - coapClient v4

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.

10 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-system. To allow clients to contact it, we can use TCP connections that can be created in all high-level programming languages.
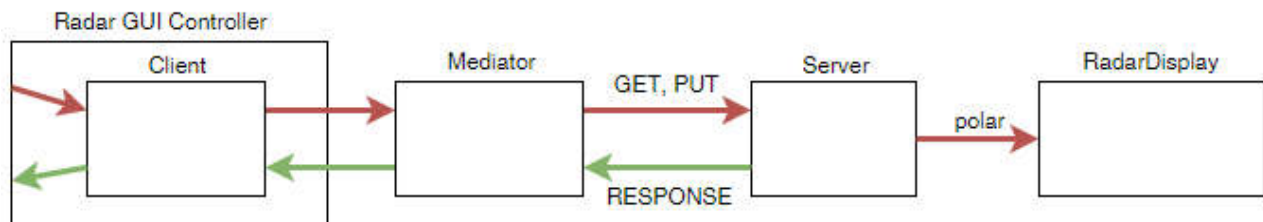

Figure 40: coapClient-v5 and coapMediator-v1 scheme systems

10.1 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).

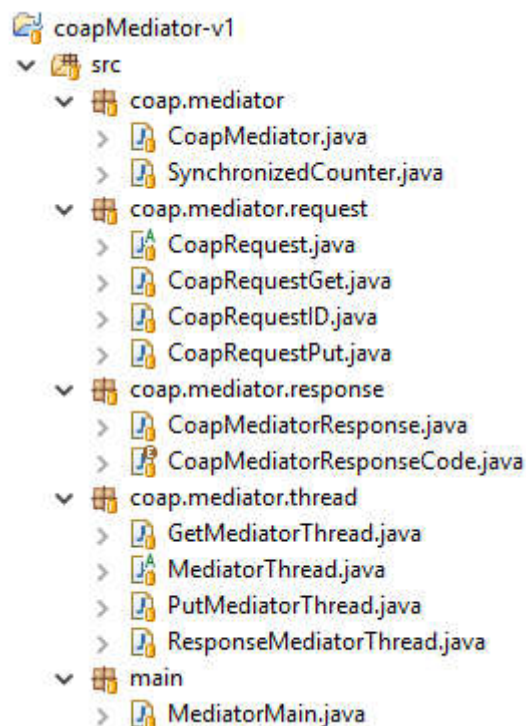The new system is completed and doesn't require anything else.

Figure 41: Mediator subsystem - coapMediator v1

10.2 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.

```java
15  public class CoapMediatorClient {
16
17      private static final String REMOTE_MEDIATOR_HOST = "localhost";
18      private static final int REMOTE_MEDIATOR_PORT = 5633;
19      private static final String HEADER_SEPARATOR = "-";
20      private static final String ARGUMENT_SEPARATOR = "!";
21
22⊕     public static CoapRequestID Get(String resourceURI){…
52
53⊕     public static CoapRequestID Put(String resourceURI, RadarPoint point){…
84
85⊕     public static MediatorMessage GetResponse(CoapRequestID requestId){…
117
118  }
119
```

Figure 42: CoapMediatorClient.java - coapClient v5

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.

```java
3   public class MediatorMessage {
4       private CoapMediatorResponseCode responseCode;
5       private String message;
6
7⊖      public MediatorMessage(CoapMediatorResponseCode code, String message){
8           this.responseCode = code;
9           this.message = message;
10      }
11
12⊖     public boolean isValid() {
13          return  responseCode.isValid();
14      }
15
16⊖     public boolean isAvailable() {
17          return responseCode.isAvailable();
18      }
19
20⊖     public boolean isSuccess(){
21          return responseCode.isSuccess();
22      }
23
24⊖     public String getMessage(){
25          return message;
26      }
27  }
```

Figure 43: MediatorMessage.java - coapClient v5

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

11 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.

A different use case requires the implementation of:

- a new CoapResource, that represents the resource that we have to manage.
- a new coapServer, it will wrap the resource's state and provide the REST API to accede at the resource.

Nothing else. 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.