

# **CaseStudy1**

## **Showing sonar distances on a radar**

Antonio Natali

Alma Mater Studiorum – University of Bologna  
viale Risorgimento 2, 40136 Bologna, Italy  
[antonio.natali@unibo.it](mailto:antonio.natali@unibo.it)

## Table of Contents

CaseStudy1 Showing sonar distances on a radar .....	1
<i>Antonio Natali</i>	
1 Starting .....	2
1.1 Interacting with the radar .....	3
1.2 Using the radar: a model-based approach .....	4
1.2.1 Reusing conventional objects .....	4
1.2.2 The radar as information handler .....	5
1.2.3 A radar client .....	5
1.2.4 Sending messages and emitting events .....	6
2 The problem to solve .....	7
3 Requirements analysis .....	7
3.1 The virtual environment .....	7
3.2 The mbot robot .....	8
3.2.1 IOT reference architecture .....	8
3.3 The robot (on RaspberryPi) .....	9
3.3.1 The rover as a command interpreter .....	9
3.3.2 The connection to the Unity virtual environment .....	10
3.3.3 Activating the motors for a prefixed time .....	10
3.3.4 Activating the motors for a long time: reactive actions .....	11
3.3.5 The emitter .....	11
3.3.6 The real sonar .....	12
4 Problem analysis .....	13
4.1 The logical architecture of the robot-radar system .....	13
5 Beyond QActors .....	15
5.1 Writing the application in Node.js .....	15
5.2 A <i>QActor</i> system as an adapter .....	16
6 (Qa)Models as system integrators .....	18
6.1 An activation model for the system components .....	18
6.2 Automatic activation of the robot-radar system .....	20
7 Publish-Subscribe interactions .....	22
7.1 Extending the robot as a publisher of events .....	22
7.2 Extending the radar as a subscriber of events .....	23
8 A project architecture .....	24
9 Refactoring .....	25
9.1 The robot executor .....	25
9.1.1 A message-based command executor .....	26
9.1.2 Composed robot moves .....	26
9.1.3 Activating the Unity system .....	27
9.1.4 Executing turn moves .....	27
9.1.5 Emitting <i>polar</i> events .....	28
9.2 The robot application agent .....	28
10 A front-end server (in Node.js) .....	30
10.1 The front-end server implementation .....	30

## 1 Starting

Open an empty directory (e.g. `C:/iss2018Lab`) and

1. Clone the `iss2018Lab` repository by executing the following command:

```
git clone https://github.com/anatali/iss2018Lab.git
```

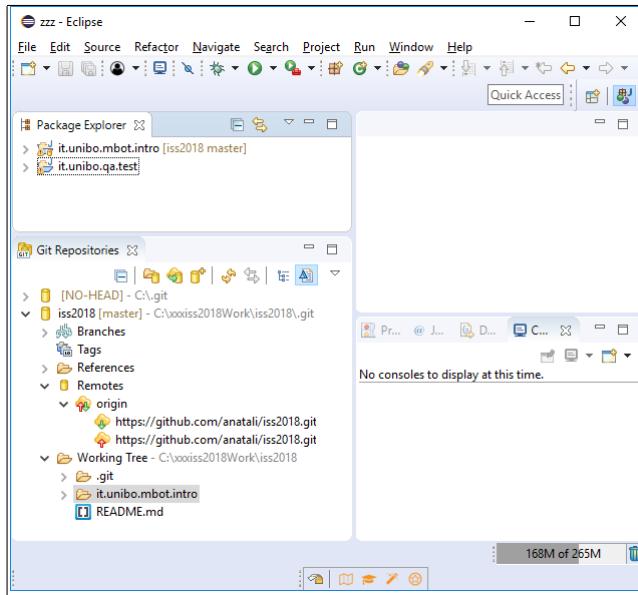
To update the repository, the command is `git pull`.

2. Now open the Eclipse working space into your working directory (e.g. `C:/iss2018Work`) and do:

Window -> ShowView -> Other -> Git -> Git Repositories

Add an existing local Git repository (`C:/iss2018Lab`)

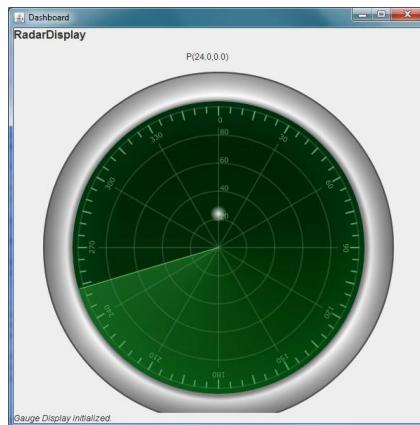
3. Import (by copying) into the current working space<sup>1</sup> the project `it.unibo.mbot.intro`. The result is:



4. In the project `it.unibo.mbot.intro`, the file `runnable/it.unibo.ctxRadarBase.MainCtxRadarBase-1.0.zip` includes the implementation of a software system able to display distance values on an output device that simulates the screen of a radar. To execute the application, `unzip` the file (into some other directory) and execute:

```
java -jar it.unibo.qactor.radar-1.0.jar
```

The virtual display shown by the radar system is:



<sup>1</sup> To avoid any conflict in project updating.

---

## 1.1 Interacting with the radar

In order to use the radar system, we must send messages to it, by using a TCP client connection on the port 8033. The messages must be *Strings* with the following structure:

```
1 msg(polarMsg,dispatch,SENDER,radarguibase,POLAR,MSGNUM)
```

where

- **SENDER** is the name (in lowercase) of the sender ;
- **POLAR** is a value of the form **p(D,ANGLE)**, with **0<=D<=80, 0<=ANGLE<=180**;
- **MSGNUM** is a natural number

Let us implement a TCP client by using the **net** module of Node.js, that provides an asynchronous network wrapper. We start with the code that establishes a connection with the radar:

```
1 var net = require('net');
2 var host = "localhost";
3 var port = 8033;
4
5 console.log('connecting to ' + host + ":" + port);
6 var conn = net.connect({ port: port, host: host });
7 conn.setEncoding('utf8');
8
9 // when receive data back, print to console
10 conn.on('data',function(data) {
11     console.log(data);
12 });
13 // when server closed
14 conn.on('close',function() {
15     console.log('connection is closed');
16 });
17 conn.on('end',function() {
18     console.log('connection is ended');
19 })
```

**Listing 1.1.** TcpClientToRadar.js: set up a connection

Now, let us define some utility functions to send messages:

```
1 var sendMsg = function( msg ){
2     try{
3         console.log("SENDING " + msg );
4         conn.write(msg+"\n"); //asynchronous!!!
5     }catch(e){
6         console.log("ERROR " + e );
7     }
8 }
9
10 function sendMsgAfterTime( msg, delay ){
11     setTimeout( function(){ sendMsg( msg ); }, delay );
12 }
```

**Listing 1.2.** TcpClientToRadar.js: utility

The **send** function writes the given data on the connection in asynchronous way; thus, it immediately returns control to the caller. The **sendMsgAfterTime** function allows us to delay the call after a given delay.

Finally, we send some data to the radar:

```
1 var msgNum=1;
2
3 sendMsgAfterTime("msg(polarMsg,dispatch,jsSource,radarguibase, p(50,30)," + msgNum++ +"")", 1000);
4 sendMsgAfterTime("msg(polarMsg,dispatch,jsSource,radarguibase, p(50,90)," + msgNum++ +"")", 2000);
5 sendMsgAfterTime("msg(polarMsg,dispatch,jsSource,radarguibase, p(50,150)," + msgNum++ +"")", 3000);
6
7 //setTimeout(function(){ conn.end(); }, 4000);
```

**Listing 1.3.** TcpClientToRadar.js: send data to radar

The radar shows the points, while the output of our client is:

```

1 connecting to localhost:8033
2 SENDING msg(polarMsg,dispatch,jsSource,radarguibase, p(50,30),1)
3 SENDING msg(polarMsg,dispatch,jsSource,radarguibase, p(50,90),2)
4 SENDING msg(polarMsg,dispatch,jsSource,radarguibase, p(50,150),3)
5 connection is ended
6 connection is closed

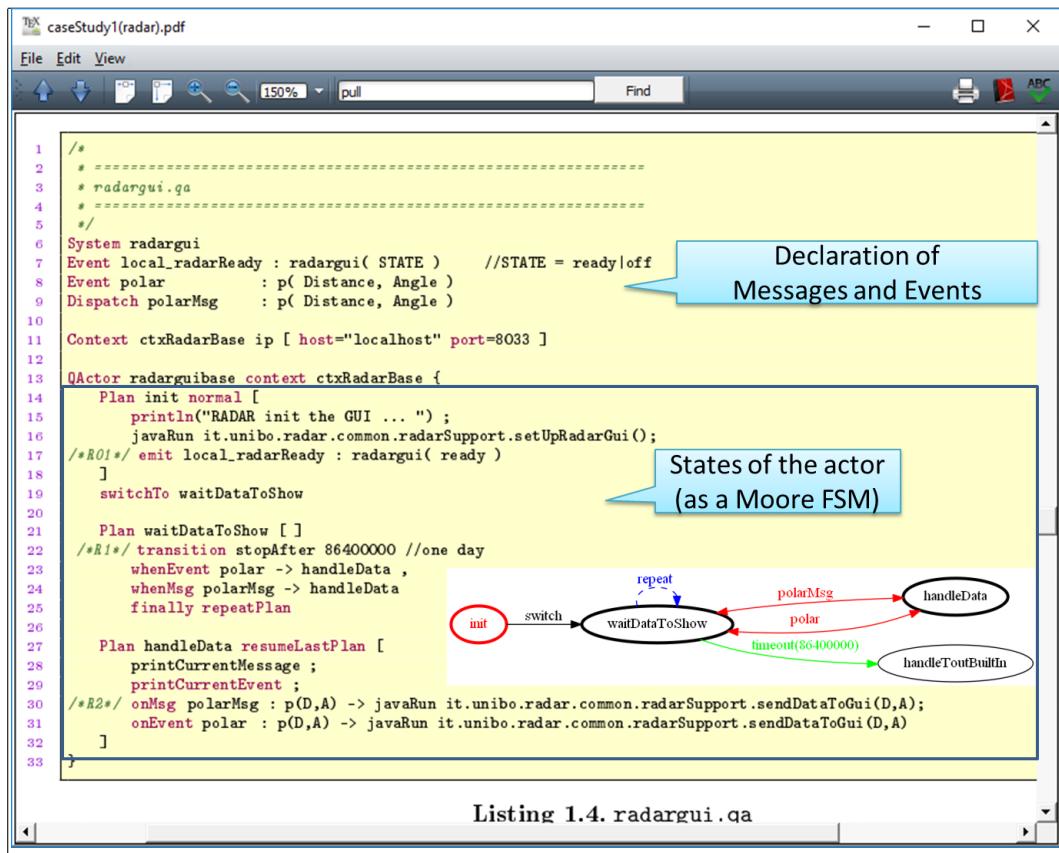
```

Instead of using Node, we could write a client for the radar in Java. This task is left to the reader.

## 1.2 Using the radar: a model-based approach

In the previous version of the radar client, we did not have any knowledge on the internal structure of the radar system. We exploited only the knowledge on the low-level structure of messages handled by the radar.

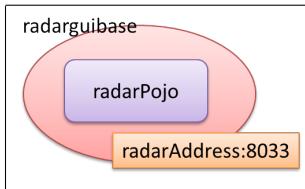
But, fortunately, there exist a high level description of the radar system, expressed in the high-level, custom modelling language *QActor*. This description is a (executable) model defined as follows:



The model describes the structure, the interaction and the behaviour of the radar.

### 1.2.1 Reusing conventional objects .

Internally, the `radargui` (re)uses a POJO (`radarPojo`) that implements the radar GUI.



The `radarPojo` is implemented as a Java class named `it.unibo.radar.common.radarSupport`. Note that the name of the class starts with a lower-case letter for a constraint imposed by the current implementation of the `QActor` software factory.

Moreover, each operation provided by the Java class must have as its first argument a variable of type `QActor`. For example:

```

1 public static void setUpRadarGui( QActor qa ) {
2     try {
3         radarControl = new RadarControl( qa.getOutputEnvView() );
4     } catch (Exception e) {
5         e.printStackTrace();
6     }
7 }
```

### 1.2.2 The radar as information handler .

The radar is able to handle messages and events *explicitly declared* at the very beginning of the model<sup>2</sup>:

```

1 Dispatch polarMsg : p( Distance, Angle )
2 Event    polar   : p( Distance, Angle )
```

Since the message `polarMsg` is declared as a `Dispatch`, the interaction is of type 'fire-and-forget'.

### 1.2.3 A radar client .

Thus, another way to introduce a client of the radar system is to define the client by using the same modelling language used for the radar. Let us introduce an example of such a model<sup>3</sup>:

<sup>2</sup> The event `local_radarReady` is a local event used internally.

<sup>3</sup> The code is in the file `it.unibo.mbot.intro/src/radarUsage.qa`.

```

/*
 * =====
 * radarUsage.qa
 * =====
 */
System radarUsage
Dispatch polarMsg : p( Distance, Angle )
Event polar : p( Distance, Angle )

Context ctxRadarUsage ip [ host="localhost" port=8022 ]
Context ctxRadarBase ip [ host="localhost" port=8033 ] -standalone

QActor radartest context ctxRadarUsage {
Rules{
    p(80,0). p(80,30). p(30,50). p(80,60). p(60,70).
    p(80,90). p(80,160). p(10,130). p(80,150). p(80,180).
}
Plan init normal [
    println("radartest STARTS ")
]
switchTo dotest

Plan dotest [
    delay 1000 ;
    [ !? p(X,Y) ] println( sending(p(X,Y)) );
    [ ?? p(X,Y) ] sendto radarguibase in ctxRadarBase -m polarMsg : p( X, Y ) else endPlan "radartest ENDS";
    delay 2000 ;
    [ !? p(X,Y) ] println( emitting(p(X,Y)) );
    [ ?? p(X,Y) ] emit polar : p(X,Y) else endPlan "testDone"
]
finally repeatPlan
}

```

Listing 1.4. radarUsage.qa

The model states that:

- Our `radarUsage` system is a distributed system composed of two computational nodes (`Contexts`).
- The node named `ctxRadarBase` is external to the systems (flag `-standalone`): it is the node that executes the given radar system. The context `ctxRadarUsage` represents the node in which we will run our radar client.
- Our radar client is modelled as a `QActor` (`radartest`): it works as a finite state machine that (in the state `doteст`) sends messages and emits events. We will expand this point in Subsection 1.2.4.
- The messages and events involved in our system are *the same* defined in the radar model:

```

1 Dispatch polarMsg : p( Distance, Angle )
2 Event polar : p( Distance, Angle )

```

- The data sent by our client are defined in the actor's knowledge base as a sequence of facts (in Prolog syntax) and are 'consumed' in the state `doteст`, by using `guards`.

From the model above, the `QActor` software factory generates an executable version written in Java. The main program is in the file:

```
1 it.unibo.mbot.intro/src-gen/it/unibo/ctxRadarUsage/MainCtxRadarUsage.java
```

If we run this file, the radar will show 10 points.

#### 1.2.4 Sending messages and emitting events .

The concept of message in the `QActor` world implies that we must know the name of the message destination, that must be another `QActor`. This fact is reflected in the sentence:

```
1 sendto radarguibase in ctxRadarBase -m polarMsg : p( X, Y )
```

Note that the knowledge of the name of the receiving radar actor (`radarguibase`) is not required for events:

```
1 emit polar : p(X,Y)
```

---

## 2 The problem to solve

The problem now is the following:

with reference to a mbot physical robot working in virtual environment, build an application that sends to the radar the data sensed by the virtual and the real sonars. More specifically:

- the data of the *virtual sonar sonar1* must be displayed on the direction of angle=30;
- the data of the *virtual sonar sonar2* must be displayed on the direction of angle=120;
- the data of the *virtual sonar* on the virtual robot must be displayed on the direction of angle=90 at the fixed distance of 40;
- the data of the *real sonar* on the physical robot must be displayed on the direction of angle=0;

## 3 Requirements analysis

We ask the customer for the following basic information:

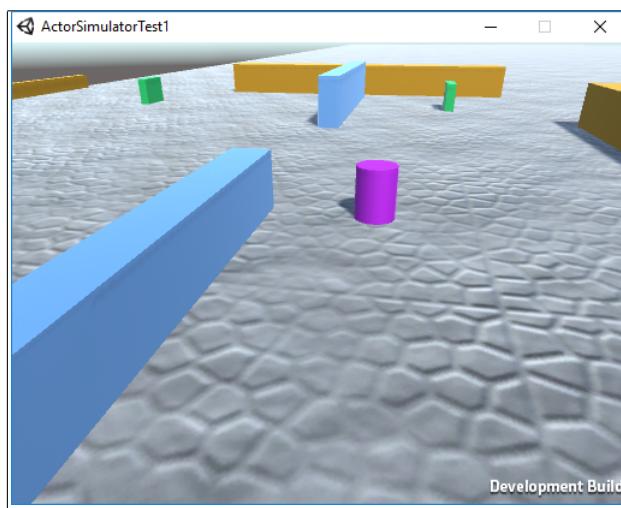
1. what is a mbot?
2. what is the virtual environment?
3. in which way we can obtain data from a virtual or from a real sonar?

### 3.1 The virtual environment

As regards the virtual environment, the answer is in the project [it.unibo.issMaterial](#), available by cloning the `iss2018` GIT repository:

```
git clone https://github.com/anatali/iss2018.git
```

The virtual environment is an application written in Unity, included in the file: [it.unibo.issMaterial/issdocs/Lab/virtualRobot.zip](#). Let us *unzip* this file, and run [VirtualRobotE80.exe](#). We obtain a scene showing an environment made of a set of walls and fixed obstacles, a mobile obstacle (the cylinder) and a sonar (the small boxes in green, named `sonar1` and `sonar2`):



The original Unity environment has been modified to interact with *QActor* systems. Details about this point are given in [IntroductionQa2017.pdf](#) (section 12)<sup>4</sup>.

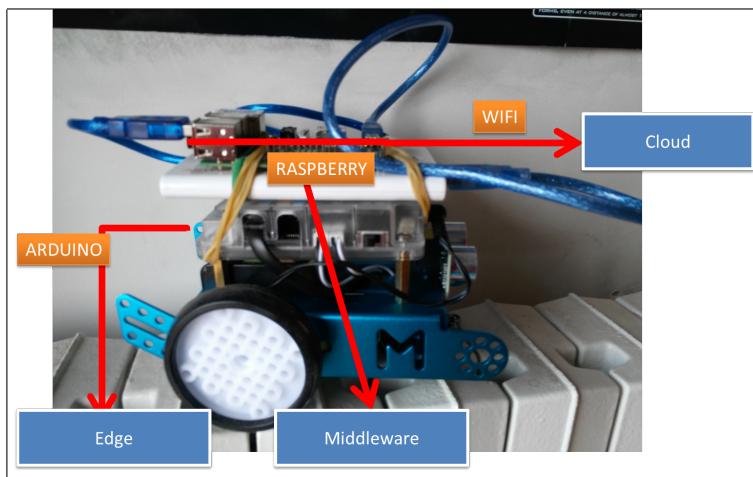
<sup>4</sup> The file is available in [it.unibo.issMaterial/issdocs/Material](#).

The point to highlight here is that, when the virtual robot (`rover`) is intercepted by a sonar, the modified Unity system emits the `QActor` event `sonar : sonar(SONARNAME, TARGET, DISTANCE)` where `SONARNAME` is `sonar1` or `sonar2`. Moreover, the virtual `rover` is equipped (in its front) with a sonar, that emits the event `sonarDetect : sonarDetect(TARGET)` when detects an obstacle.

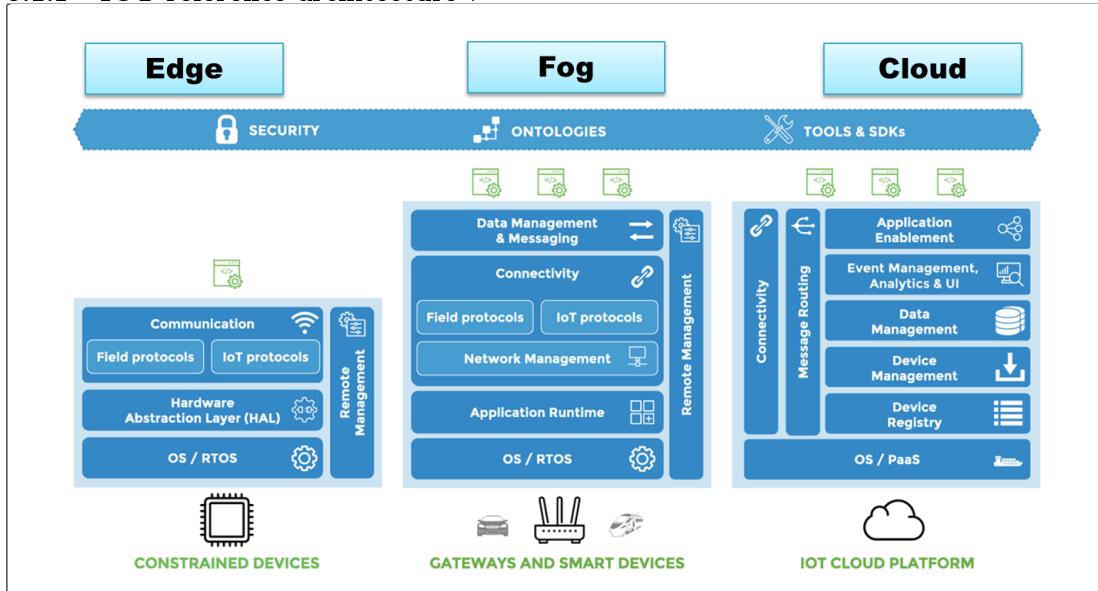
As regards the other questions, our goal is to build a model for the `mbot` and a model for the real sonar. The goal of these models is to clarify how we can exchange information with the corresponding entities of our `application domain`. The internal structure and the internal behaviour of the entities have relatively less importance at this stage.

### 3.2 The mbot robot

The `mbot` architecture is an example of a IoT architecture in which the `edge` part is implemented on Arduino, the `middleware` part is implemented on a RaspberryPi and the `cloud` part is implemented on a conventional PC.



#### 3.2.1 IoT reference architecture .



---

Arduino handles physical devices such as motors and sensors, while RaspberryPi provides support for interaction with a remote node. More precisely, as regards the interactions:

- the robot communicates with external world via a WIFI local network. Let us denote as `mbotAddress` the IP of the robot in such a network;
- the robot accepts move commands sent via a browser connected to the address `mbotAddress:8080`;
- through the same web interface, the robot accepts a command to connect itself to a modified virtual Unity environment working on a remote PC in the local network;
- the robot emits `QActor` events (`sonar` and `sonarDetect`). This means that we can build a new `QActor` system including the robot and a `QActor` element able to perceive and handle these events.

### 3.3 The robot (on RaspberryPi)

The subsystem that implements the robot is formally described as follows:

```

1  /*
2   * =====
3   * * mbotControl.qa
4   * =====
5   */
6 System mbotControl
7 Event usercmd : usercmd(CMD)
8 Event sonar : sonar(SONAR, TARGET, DISTANCE) //From (virtual) sonar
9 Event sonarDetect : sonarDetect(X)           //From (virtual robot) sonar
10 Event realSonar : sonar(DISTANCE)          //From real sonar on real robot
11 Event polar : p( Distance, Angle )
12 Event unityAddr : unityAddr( ADDR )         //From user interface
13
14 Context ctxMbotControl ip [ host="localhost" port=8029 ] -g cyan -httpserver

```

**Listing 1.4.** mbotControl.qa: starting the model

This starting part of the model says that the robot provides a TCP server on port 8029 and a Web interface (flag `-httpserver`). This (sub)system does not known any other component; it can be executed in a '`standalone`' way.

#### 3.3.1 The rover as a command interpreter .

The next part of the model defines the actors working in the `ctxMbotControl` context. The first one is the actor that extends a basic `mbot` with the possibility to receive remote commands from the human user:

```

1 QActor rover context ctxMbotControl {
2     Plan init normal [
3         println("rover START")
4     ]
5     switchTo waitUserCmd
6
7     Plan waitUserCmd[ ]
8     transition stopAfter 600000
9     whenEvent usercmd -> execMove
10    finally repeatPlan

```

**Listing 1.5.** mbotControl.qa: waiting for user commands

The behavior of the rover actor can be expressed as a set of states in which we send commands to Arduino according to the input command given by the human user (perceived as the event `usercmd`):

```

1 Plan execMove resumeLastPlan[
2     printCurrentEvent;
3     onEvent usercmd : usercmd( robotgui(w(X)) ) -> switchTo moveForward;
4     onEvent usercmd : usercmd( robotgui(s(X)) ) -> switchTo moveBackward;
5     onEvent usercmd : usercmd( robotgui(a(X)) ) -> switchTo turnLeft;
6     onEvent usercmd : usercmd( robotgui(d(X)) ) -> switchTo turnRight ;
7     onEvent usercmd : usercmd( robotgui(h(X)) ) -> switchTo stopTheRobot ;
8     onEvent usercmd : usercmd( robotgui(f(X)) ) -> javaRun it.unibo.rover.mbotConnArduino.mbotLinefollow() ;
9     onEvent usercmd : usercmd( robotgui(unityAddr(X)) ) -> switchTo connectToUnity; //X=localhost at the moment

```

```

10     onEvent usercmd : usercmd( robotgui(x(X)) ) -> switchTo terminataAppl
11 ]

```

**Listing 1.6.** mbotControl.qa: command interpreter

The built-in operation `javaRun` activates user-defined Java code (see Section 6 ).

### 3.3.2 The connection to the Unity virtual environment .

The built-in operation `createUnityObject` creates a virtual robot object game in the Unity virtual environment. Thus, the `connectToUnity` state can be defined as follows:

```

1 Plan connectToUnity resumeLastPlan[
2     onEvent usercmd : usercmd(robotgui(unityAddr(ADDR))) -> println(ADDR); //ADDR=localhost in this version
3     [ !? unityOn ] println( "UNITY already connected" )
4 ]
5 switchTo [ not !? unityOn ] doconnectToUnity
6
7 Plan doconnectToUnity resumeLastPlan[
8     println("ACTIVATING UNITY. Wait a moment ... " );
9     javaRun it.unibo.commToRadar.polarToRadar.customExecute("C:/Didattica2018Run/unityStart.bat");
10    delay 10000; //wait until Unity activated
11    connectUnity "localhost";
12    createUnityObject "rover" ofclass "Prefabs/CustomActor";
13    backwards 70 time ( 800 );
14    right 70 time ( 1000 ); //position
15    addRule unityOn
16 ]

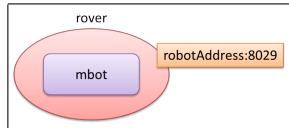
```

**Listing 1.7.** mbotControl.qa: connect to Unity

The fact `unityOn` is used as a `guard` to avoid the activation of Unity if it is already done.

### 3.3.3 Activating the motors for a prefixed time .

Internally, the `rover` reuses a POJO (`mbot`) that provides a set of operations to move the physical robot and to acquire data from its sensors.



The basic `mbot` is implemented by the Java class `it.unibo.rover.mbotConnArduino`. Note that the name of the class starts with a lower-case letter for a constraint imposed by the current implementation of the `QActor` software factory. Moreover, each operation provided by the Java class must have as its first argument a variable of type `QActor`. For example:

```

1 public static void mbotForward(QActor actor) {
2     try { if( conn != null ) conn.sendCmd("w"); } catch (Exception e) {e.printStackTrace();}
3 }

```

Thus, the states in which we execute simple moves of the robot can be defined as follows:

```

1 Plan turnLeft resumeLastPlan [
2     javaRun it.unibo.rover.mbotConnArduino.mbotLeft();
3     [ !? unityOn ] left 40 time(750) else delay 900;
4     javaRun it.unibo.rover.mbotConnArduino.mbotStop()
5 ]
6 Plan turnRight resumeLastPlan [
7     javaRun it.unibo.rover.mbotConnArduino.mbotRight();
8     [ !? unityOn ] right 40 time(750) else delay 900;
9     javaRun it.unibo.rover.mbotConnArduino.mbotStop()
10]
11 Plan stopTheRobot resumeLastPlan [

```

```

12     stop 40 time ( 10 );
13     javaRun it.unibo.rover.mbotConnArduino.mbotStop()
14 ]

```

**Listing 1.8.** mbotControl.qa: simple moves

The `rover` runs on the RaspberryPi<sup>5</sup> and gives to the `mbot` the capability to interact with the external world.

### 3.3.4 Activating the motors for a long time: reactive actions .

The user command to move the robot forward or backward cannot be handled in such a simple way. In fact we have to move 'forever' (or for a long time) unless some event is raised from the external world.

In the next states we exploit the `QActor` feature of *reactive actions* (see `IntroductionQa2017.pdf` - section 14) to perform (long lived) actions while being able to 'react' to input data such as the sonar events or other user commands:

```

1 Plan moveForward resumeLastPlan[
2     javaRun it.unibo.rover.mbotConnArduino.mbotForward()
3 ]
4 reactive onward 40 time( 15000 )
5     whenEnd      -> endOfMove
6     whenTout 30000    -> handleTout
7     whenEvent sonarDetect -> handleRobotSonarDetect
8     or whenEvent sonar -> handleSonar
9     or whenEvent usercmd -> execMove
10
11 Plan moveBackward resumeLastPlan[
12     javaRun it.unibo.rover.mbotConnArduino.mbotBackward()
13 ]
14 reactive backwards 40 time ( 15000 )
15     whenEnd      -> endOfMove
16     whenTout 30000    -> handleTout
17 //   whenEvent sonarDetect -> handleObstacle //no sensor on robot back
18     whenEvent sonar      -> handleSonar
19     or whenEvent usercmd -> execMove

```

**Listing 1.9.** mbotControl.qa: reactive actions

### 3.3.5 The emitter .

Another component that must run on the RaspberryPi is an actor able to make sonar events available to the external world. Here is an example that maps sonar events into events that can be understood by the radar:

```

1 QActor sonardetector context ctxMbotControl{
2     Plan init normal [ ]
3     switchTo waitForEvents
4
5     Plan waitForEvents[ ]
6     transition stopAfter 600000
7         whenEvent sonar      -> sendToRadar,
8         whenEvent sonarDetect -> sendToRadar,
9         whenEvent realSonar -> sendToRadar
10    finally repeatPlan
11
12    Plan sendToRadar resumeLastPlan [
13        printCurrentEvent;
14        onEvent realSonar : sonar( DISTANCE )           -> emit polar : p(DISTANCE, 0) ;
15        onEvent sonar : sonar(sonar1, TARGET, DISTANCE ) -> emit polar : p(DISTANCE,30) ;
16        onEvent sonar : sonar(sonar2, TARGET, DISTANCE ) -> emit polar : p(DISTANCE,120) ;
17        onEvent sonarDetect : sonarDetect(TARGET)          -> switchTo showObstacle
18    ]
19
20    Plan showObstacle resumeLastPlan[

```

<sup>5</sup> The `rover` could run also on a conventional PC with Arduino connected to the PC.

```
21     println( "found obstacle" );
22     emit polar : p(30,90)
23 //      sendto radarguibase in ctxRadarBase -m polarMsg : p( 30, 90 )
24   ]
25 }
```

**Listing 1.10.** mbotControl.qa: the emitter

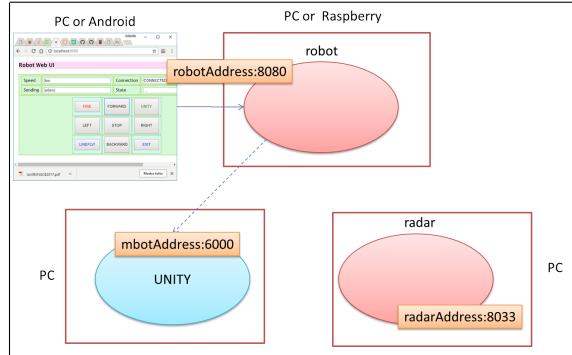
Note that the choice to emit events (and not to send messages) avoid the need to known the name of any destination actor.

### 3.3.6 The real sonar .

The `sonardetector` actor handles events emitted both from the virtual world (`sonar`, `sonarDetect`) and from the real world (`realSonar`). The reader should imagine the code to be put on Arduino to 'create' events at the `sonardetector` level.

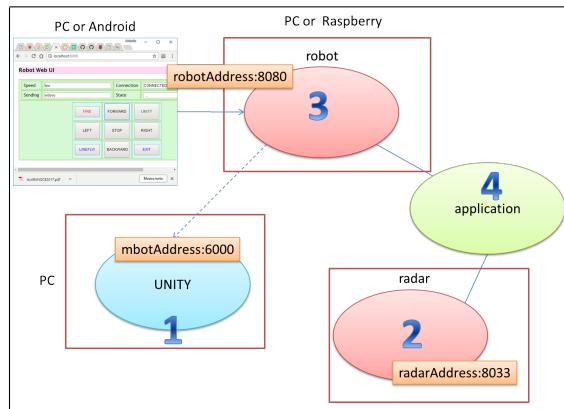
## 4 Problem analysis

From the requirement analysis we have a better understanding of **WHAT** we have to do and a concrete (operational) view of the *domain components* already available. Each of these components looks as a **micro-service** that can be executed independently from the others.



Now, we have to *understand* the problem posed by the requirements, the problematic issues involved by the problem and the constraints imposed by the problem or by the context. This is the main goal of the problem analysis phase, whose outcome is the logical architecture involved by the problem itself.

The following picture shows, in a informal way, a possible architecture of the system that arises from our current 'technology assumptions':



From the logical point of view, our application is a 'router' of events from the robot to the radar.

### 4.1 The logical architecture of the robot-radar system

The robot is a source of information expressed as *QActor* events of the form `polar:p(DISTANCE,ANGLE)`. This kind of information is available by any other software component belonging to a *QActor* system. Since also the radar is implemented as a *QActor* application, we can state that the logical architecture of our application can be expressed by the following *QActor* model:

```

1 System robotRadarAppl
2 Event polar      : p( Distance, Angle )
3
4 Context ctxRobotRadarAppl ip [ host="localhost" port=8095 ] -g white
5 /*
6 * The following two contexts must be already active
7 */
8 Context ctxMbotControl   ip [ host="localhost" port=8029 ]

```

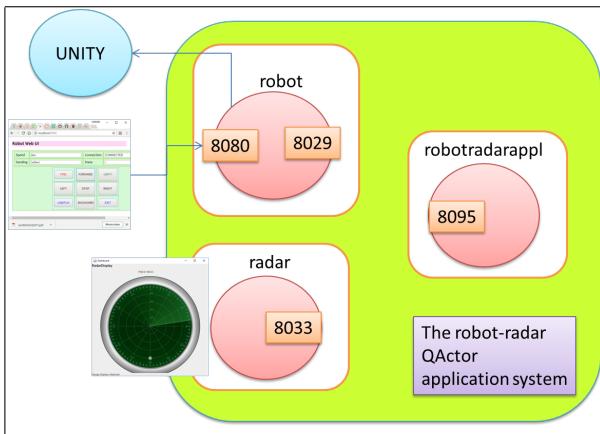
```

9 | Context ctxRadarBase      ip [ host="localhost" port=8033 ]
10|
11| QActor robotradarappl context ctxRobotRadarAppl{
12|   Plan init normal [
13|     println("robotradarappl STARTS " );
14|     delay 600000
15|   ]
16| }

```

**Listing 1.11.** robotRadarAppl.qa

Our application is composed of actors working in 3 different contexts: two of these contexts are already available and can be viewed as tested and usable micro-services. The third one context is the 'glue' that connects together the available components to form our application.



In this case, the 'application' has nothing to do, since the *QActor* infrastructure provides the transmission of the **polar** (non-local) events from the robot to the other contexts composing the application and the radar is already able to handle this kind of events.

Since this system specification is executable, we have built in a very short time a working prototype of our application.

---

## 5 Beyond QActors

The question now is: *are we obliged to implement our application by using QActors?* Of course, not.

Let us note that, if we want to build the application by using some other conventional programming language, we have to face the following main *problematic issues*:

1. The components that represent the robot and the radar are provided as executable (micro)services, without giving us the possibility to access/modify their source code.
2. Our application must be able to send information to the radar. Thus it could work as already discussed in Subsection 1.1.
3. The robot is a source of information expressed (at low-level) as event-Strings of the form:

```
1 msg(polar,event,SENDER,none,p(DISTANCE,ANGLE),MSGNUM)
```

The problem is how to deliver this information to our non-QActor code, without changing the source code of the robot.

In the **project phase** that follows our analysis, we could decide to build our application by using `Node.js`. In particular, we could build a TCP server that *i*) connects itself to the radar as a TCP client as done in Subsection 1.1, *ii*) waits for an event-String and *iii*) sends proper data to the radar.

### 5.1 Writing the application in Node.js

The connection to the radar of our application server written in `Node.js` can be defined as follows (the function `emitEventForRadar` is an utility function to send data to the radar):

```
1 var net = require('net');
2 var utils = require('../utils'); //for handling uncaughtException
3
4 /*
5 * -----
6 * CONNECTION TO RADAR
7 * -----
8 */
9 var radarhost = "localhost";
10 var radarport = 8033;
11 var msgNum = 1;
12
13 console.log('connecting to RADAR at ' + radarhost + ":" + radarport);
14 var conn = net.connect({ port: radarport, host: radarhost });
15 conn.setEncoding('utf8');
16
17 var emitEventForRadar = function( data ){
18     try{
19         var msg = "msg(polar,event,jsSource,none," + data + "," + msgNum++ +")";
20         console.log("SENDING " + msg );
21         conn.write(msg+"\n"); //Asynchronous!!!
22     }catch(e){
23         console.log("ERROR " + e );
24     }
25 }
```

**Listing 1.12. nodeCode/PolarToRadar.js: connection to the radar**

The server part, waiting for low-level event-Strings, can be defined as follows:

```
1 /*
2 * -----
3 * SERVER
4 * -----
5 */
6 var host = "localhost";
7 var port = 8057;
8
9 var server = net.createServer(function(socket) {
```

```

10     socket.on('data', function( data ) { //data has the form p(DISTANCE,ANGLE)
11         var dataNoWhiteSpaces = ("."+data).trim();
12         emitEventForRadar( dataNoWhiteSpaces );
13     });
14
15     socket.on('end', socket.end);
16 });
17
18 //STARTING
19 emitEventForRadar("p(70, 0)"); //just to show
20 console.log("PolarToRadar starts at " + port);
21 server.listen(port, host );

```

**Listing 1.13.** nodeCode/PolarToRadar.js: connection to the radar

Now we could (see Section 4) :

1. Activate the Unity virtual environment.
2. Activate the radar.
3. Activate the robot.
4. Activate the Node service with the command:

```
1 node PolarToRadar.js
```

However, this set of activations **does not build a system**. In fact, the problem 2 above is still unsolved: the events generated by the robot are not propagated to our **PolarToRadar** Node application.

## 5.2 A *QActor* system as an adapter

To create a system able to perceive the events emitted by the robot without changing the source code of the robot, we can introduce a *QActor* system that can work as an 'adapter' from the *QActor* world of the robot to the Node world of **robotToRadarAdapter.js**:

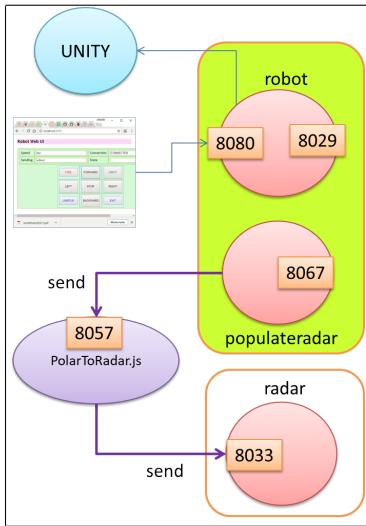
```

1 System robotToRadarAdapter
2 Event polar : p( Distance, Angle )
3
4 Context ctxRobotToRadarAdapter ip [ host="localhost" port=8067 ]
5 Context ctxMbotControl ip [ host="localhost" port=8029 ]
6
7 QActor populateradar context ctxRobotToRadarAdapter{
8     Plan init normal [ ]
9     switchTo waitForEvents
10
11    Plan waitForEvents[ printCurrentEvent ]
12    transition stopAfter 600000
13        whenEvent polar : p( D, A ) do
14            javaRun it.unibo.commToRadar.polarToRadar.sendPolarToNodeServer( D, A )
15        finally repeatPlan
16    }

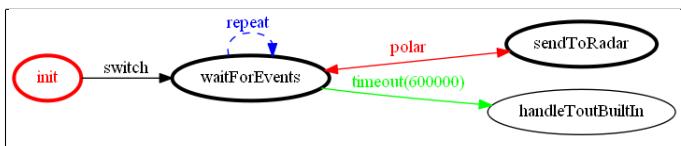
```

**Listing 1.14.** robotToRadarAdapter.qa

The application is now introduced as an 'extension' of the robot. In fact we define a distributed system composed by the robot (as a standalone *QActor* entity) and an actor able to populate the radar with the events emitted by the robot.



The behaviour of the actor **populateradar** is defined as a finite state machine that 'reacts' to events (**polar**) emitted by the robot:



The operation **sendPolarToNodeServer** is defined by the application designer in the Java utility class **it.unibo.commToRadar.polarToRadar**; it performs the sending of event data to the Node server by exploiting the **QActor** built-in operation **sendTcpMsg**:

```

1  public static void sendPolarToNodeServer(QActor qa, String D, String A) throws Exception {
2      String msgContent = "p(" + D + "," + A + ")";
3      qa.println("sendPolarToNodeServer: " + msgContent);
4      qa.sendTcpMsg( "localhost", "8057", msgContent );
5  }

```

**Listing 1.15.** `sendPolarToNodeServer` in `it.unibo.commToRadar.polarToRadar.java`

In summary, we have defined a **QActor** 'extension' of the robot that works as the producer of data for the Node (micro)service that is able to populate the radar.

---

## 6 (Qa)Models as system integrators

The definition of executable models can be useful not only to define the logical architecture of a software system, but can be also used a a way to integrate - to form a system - a set of software components written in different languages.

In Subsection 5.2 we have seen how it is possible to launch user defined actions written in `Java` to send messages to a TCP server written in `Node.js`. Here is a list of the **built-in actions** that can allow us to exploit the *QActor* modelling language to interact with software parts written in different languages and to perform other useful actions (for the syntax of `PHead`, see [IntroductionQa2017.pdf](#) - section 2):

— EXECUTE CODE —	
<code>javaRun QualifiedName(VarOrStringArgs)</code>	execute a Java operation; the arguments must be Strings.
<code>javaOp String</code>	execute the Java operation denoted in the given String.
<code>nodeOp String -o?</code>	execute the Node program denoted by the given String. The optional <code>-o</code> shows the output of the Node program.
<code>actorOp PHead</code> DEPRECATED: consider the use of <code>javaOp</code> or - better - <code>javaRun</code> .	execute the given Java method written by the application designer (see <a href="#">IntroductionQa2017.pdf</a> - section 2.10).
— REST —	
<code>sendRestGet urlString</code>	perform a REST GET operation.
<code>sendRestPut urlString -m msgString</code>	perform a REST PUT operation.
— UNITY —	
<code>connectUnity addrString</code>	connect to the modified Unity virtual environment. At the moment, the <code>addrString</code> is ignored, since the "localhost" address is used.
<code>createUnityObject "rover" ofclass "Prefabs/CustomActor"</code>	create an avatar for a <i>QActor</i> named <code>rover</code> (at the moment this name is mandatory).
— MQTT —	
<code>pubSubServer addrString</code>	specifier the name of a MQTT server to be used in the application.
<code>connectAsPublisher topicString</code>	connect as publisher to the given topic of the specifier MQTT server.
<code>connectAsSubscriber topicString</code>	connect as subscriber to the given topic of the specifier MQTT server.
<code>publishMsg topicString for actorNameString -m msgRef:PHead</code>	publish a <i>QActor</i> message to for the specified destination actor.
<code>publishEvent topicString -e msgRef : PHead</code>	publish a <i>QActor</i> event.
— EXTERNAL —	
<code>sendto dest in extctx -m msgRef : PHead</code>	forward a dispatch to an external qactor working in the already activated extctx.
<code>sendToExternalServer serverNameString -m msgString</code>	send an answer message to an external server that has sent a <i>QActor</i> message to our application.

The set of these 'useful' actions is open-ended, i.e. we could add other built-in actions to our modelling language, if it will be the case.

### 6.1 An activation model for the system components

The `javaRun` built-in action is very useful to extend our models with user-defined actions defined in user-defined `Java` classes. With `javaRun` we do not have to modify the source code of an actor (as happens for `actorOp`).

As an example, let us define here another way to activate the components of our application by introducing an `activation model`:

```

1 System robotToRadarActivator
2
3 Context ctxRobotToRadarActivator ip [ host="localhost" port=8055 ]
4
5 QActor componentactivator context ctxRobotToRadarActivator {
6     Plan init normal [
7         //The radar takes some time to start and to end its testing phase;
8         println("ACTIVATING RADAR") ;
9         javaRun it.unibo.commToRadar.polarToRadar.customExecute("C:/Didattica2018Run/radarStart.bat") ;
10
11        //The robot activates Unity, if required;
12        delay 1000;
13        println("ACTIVATING ROBOT") ;
14        javaRun it.unibo.commToRadar.polarToRadar.customExecute("C:/Didattica2018Run/robotStart.bat") ;
15
16        //PolarToRadar requires the radar;
17        delay 10000;
18        println("ACTIVATING NODE APPLICATION") ;
19        nodeOp "./nodeCode/PolarToRadar.js -o" //WARNING: the actor is engaged since PolarToRadar.js waits
20    ]
21 }

```

**Listing 1.16.** robotToRadarActivator.qa

The operation `customExecute` is defined by the application designer (in the Java utility class `it.unibo.commToRadar.polarToRadar`) in order to execute a given command:

```

1 public static void customExecute(QActor qa, String cmd) {
2     try {
3         Runtime.getRuntime().exec(cmd);
4     } catch (IOException e) { e.printStackTrace(); }
5 }

```

**Listing 1.17.** customExecute in `it.unibo.commToRadar.polarToRadar.java`

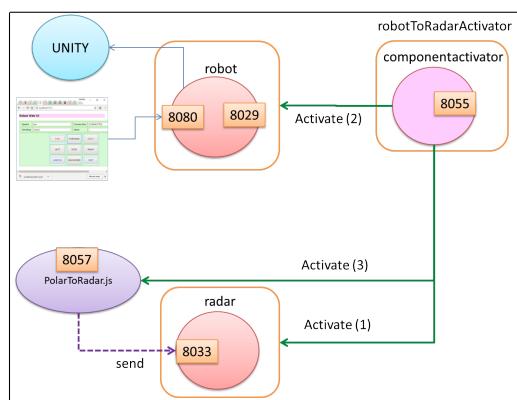
In this case, the commands to execute are batch files that put in execution our application components by storing their output in a proper `log file`. For example, the batch file `radarStart.bat` is defined as follows:

```

1 echo off
2 title radarStart
3 echo START RADAR
4 cd C:\Didattica2018Run\it.unibo.ctxRadarBase.MainCtxRadarBase-1.0
5 java -jar it.unibo.qactor.radar-1.0.jar > logRadar.txt

```

The following picture shows that the result of running the activator model is the start-up of three independent (micro) services:



Of course, these services **do not form yet any system**: we have simply automated the manual activation activities of Subsection 5.1. Of course, the same effect can be obtained by using other tools, like `batch` files. However, the advantage to have an activation model is that we can specify the logic of applications whose goal is to activate other (distributed) applications.

## 6.2 Automatic activation of the robot-radar system

To build and run a robot-radar system (see Section 4 ), now we:

1. Run the activator model of basic components (see Subsection 6.1) <sup>6</sup>.
2. Run the application of Subsection 4.1.

A batch file that does this work could be:

```

1 echo off
2 title robotRadarApplStart
3 echo START THE RADAR
4 cd .\it.unibo.ctxRadarBase.MainCtxRadarBase-1.0
5 START /B java -jar it.unibo.qactor.radar-1.0.jar > ../logRadar.txt
6 cd ..
7 echo please press CR
8 PAUSE
9
10 echo START THE ROBOT
11 cd .\it.unibo.ctxMbotControl.MainCtxMbotControl-1.0
12 START /B java -jar it.unibo.mbot.intro-1.0.jar > ../logRobot.txt
13 cd ..
14 PAUSE
15
16 echo START THE ROBOT-RADAR APPLICATION
17 cd .\it.unibo.ctxRobotRadarAppl.MainCtxRobotRadarAppl-1.0
18 START /B java -jar it.unibo.robotRadarAppl-1.0.jar > ../logRadarRobotAppl.txt
19 cd ..
20 PAUSE

```

However, these tasks can be automatized by the following system-activation model:

```

1 System robotRadarApplActivator
2
3 Context ctxRobtRadarApplActivator ip [ host="localhost" port=8078 ]
4
5 QActor robotradarapplactivator context ctxRobtRadarApplActivator {
6     Plan init normal [
7         //The radar takes some time to start and to end its testing phase;
8         println("ACTIVATING RADAR") ;
9         // javaRun it.unibo.commToRadar.polarToRadar.customExecute("C:/Didattica2018Run/radarStart.bat") ;
10        javaOp "customExecute(\"C:/Didattica2018Run/radarStart.bat\")" ;
11
12        //The robot activates Unity, if required;
13        delay 1000;
14        println("ACTIVATING ROBOT") ;
15        // javaRun it.unibo.commToRadar.polarToRadar.customExecute("C:/Didattica2018Run/robotStart.bat") ;
16        javaOp "customExecute(\"C:/Didattica2018Run/robotStart.bat\")" ;
17
18        delay 10000;
19        println("ACTIVATING THE SYSTEM") ;
20        // javaRun it.unibo.commToRadar.polarToRadar.customExecute("C:/Didattica2018Run/robotRadarStart.bat")
21        javaOp "customExecute(\"C:/Didattica2018Run/robotRadarStart.bat\")"
22    ]
23 }

```

**Listing 1.18. robotRadarApplActivator.qa**

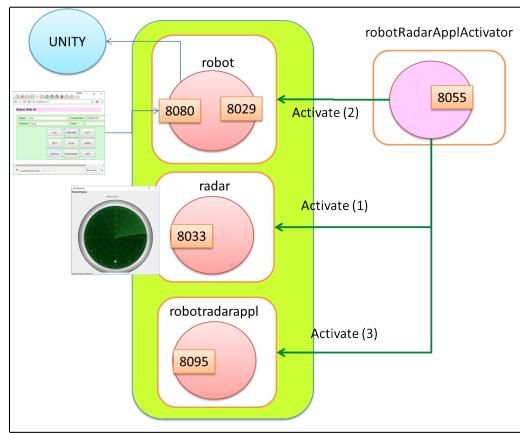
The batch file `robotRadarStart.bat` puts in execution the `robotRadarAppl` of Subsection 4.1 ; it is defined as follows:

```

1 echo off
2 title robotRadarStart
3 echo START ROBOT-RADAR APPLICATION
4 cd C:\Didattica2018Run\it.unibo.ctxRobotRadarAppl.MainCtxRobotRadarAppl-1.0
5 java -jar it.unibo.mbot.intro-1.0.jar > logRadarRobotAppl.txt

```

<sup>6</sup> Note that the activation of the `PolartoRadar` server is useless if we do not use the `robotToRadarAdpater` extension.



## 7 Publish-Subscribe interactions

Let us suppose that:

- The robot had been designed and implemented as a **publisher** of information to the external world (e.g. by using the **MQTT** protocol).
- The radar had been designed and implemented as a **subscriber** of information published in the world (e.g. by using the **MQTT** protocol).

Under these hypotheses, our logical 'router' application of Section 4 could be quite easy to build, since the given components would have been more appropriate to our needs.

Thus, another way to face the analysis of the problem is to ask whether the given components are well suited for the problem to solve.

If we detect some relevant distance from *what we have* and *what we should have*, we say that we are in presence of an **abstraction gap** that we should overcome.

In our case, the point is how we can start from a robot conceived as a publisher of information and a radar conceived as one of the possible subscribers of information emitted by the robot. There are two main ways:

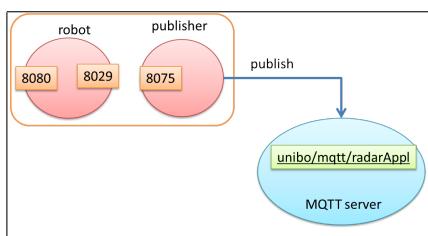
1. Introduce proper extensions like done in Subsection 5.2.
2. Modify the behaviour of the components by accessing their source code

In the rest of this section, we will follow the 'extension' approach.

### 7.1 Extending the robot as a publisher of events

```
1 System robotToPublisher
2 Event polar : p( Distance, Angle )
3
4 //pubSubServer "tcp://m2m.eclipse.org:1883"
5 pubSubServer "tcp://test.mosquitto.org:1883"
6
7 Context ctxRobotToPublisher ip [ host="localhost" port=8075 ] -g green
8 Context ctxMbotControl ip [ host="localhost" port=8029 ] -standalone
9
10 QActor roboteventpublisher context ctxRobotToPublisher{
11     Plan init normal [
12         connectAsPublisher "unibo/mqtt/radarAppl" ;
13         println("roboteventpublisher STARTED")
14     ]
15     switchTo waitForEvents
16
17     Plan waitForEvents[ printCurrentEvent ]
18     transition stopAfter 600000
19     whenEvent polar : p( Distance, Angle ) do //println( p( Distance, Angle ) )
20         publishEvent "unibo/mqtt/radarAppl" -e polar : p( Distance, Angle )
21     finally repeatPlan
22 }
```

Listing 1.19. robotToPublisher.qa

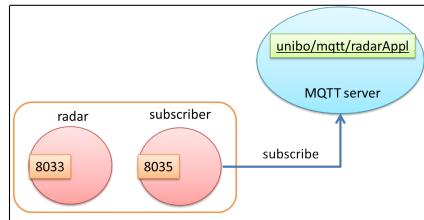


## 7.2 Extending the radar as a subscriber of events

```

1 System radarToSubscriber
2 Event polar : p( Distance, Angle )
3
4 //pubSubServer "tcp://m2m.eclipse.org:1883"
5 pubSubServer "tcp://test.mosquitto.org:1883"
6
7 Context ctxRadarToSubscriber ip [ host="localhost" port=8035 ]
8 Context ctxRadar ip [ host="localhost" port=8033 ] -standalone
9
10 QActor radareventssubscriber context ctxRadarToSubscriber{
11     Plan init normal [
12         connectAsSubscriber "unibo/mqtt/radarAppl" ;
13         println("radareventssubscriber START")
14     ]
15     switchTo waitForEvents
16
17     Plan waitForEvents[
18         printCurrentEvent;
19         delay 1000
20     ]
21     transition stopAfter 600000
22         whenEvent polar : p( D, A ) do println ( perceived( p( D, A ) ) )
23         finally repeatPlan
24 }
```

**Listing 1.20.** robotToPublisher.qa



To run the system we <sup>7</sup>:

- execute the activator of the basic components (see Subsection 6.1) [src-gen/it/unibo/ctxRobotToRadarActivator/MainCtxRobotToRadarActivator.java](#).
- execute [src-gen/it/unibo/ctxRadarToSubscriber/MainCtxRadarToSubscriber.java](#).
- execute [src-gen/it/unibo/ctxRobotToPublisher/MainCtxRobotToPublisher.java](#).

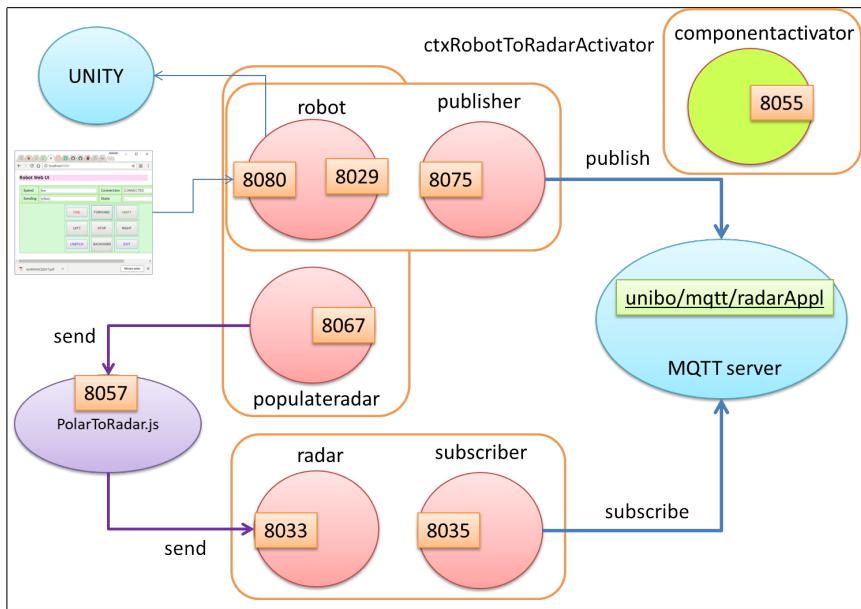
<sup>7</sup> Note that the **VPN** (if any) must be disabled.

## 8 A project architecture

With the components introduced in the previous sections, we can build a robot-radar system in different ways. For example:

1. Execute the activation model of the system components of Subsection 6.1.
2. In order to build a system, choose one of these two possibilities:
  - Execute the `ctxRadarToSubscriber` and the `ctxRobotToPublisher` of Section 7, in order to exchange sensor data by using the MQTT protocol.
  - Execute the `ctxRobotToRadarAdapter` of Subsection 5.2 in order to send sensor data by using the Node server.

The following picture show an overview of the architectures of the resulting systems :



Thus, our final, concrete project architecture will be different from the logical architecture of our first prototype introduced in Subsection 4.1. However all the constraints imposed by the logical architecture are preserved. Moreover, several components introduced in the logical architecture are reused, without any change to their source code.

---

## 9 Refactoring

In this part we intend to re-factor the model of the robot system introduced in Subsection 3.3 in order to enhance system modularity and to introduce a robot command interpreter based on messages rather than on events (see Subsection 3.3.1). More precisely, we will:

1. Define a robot executor, i.e. a component (microservice) that executes remote commands to move the robot. The commands should move both a real robot and a virtual robot avatar (rover) on the (modified) Unity system.
2. Allow a robot executor to work on a RaspberryPi and to interact with an Arduino microcontroller that manages the motors and the sensors.
3. Define a robot agent, i.e. a component (microservice) that exploits the robot executor to implement some application.
4. Build a robot-radar system as done in Subsection 4.1.
5. Define a front-end RESTful service, that allows us to interact with the robot executor via HTTP requests.

### 9.1 The robot executor

Our re-factoring of the robot executor of Subsection 3.3 starts with introducing a component that transform a user command event into a message sent to the robot:

redefining the command interpreter as a *message-based system* rather than an event-based system:

```
1 System mbotExecutor
2 Event usercmd : usercmd(CMD)           //from web gui
3 Event sonar   : sonar(SONAR, TARGET, DISTANCE) //From (virtual) sonar
4 Event sonarDetect : sonarDetect(X)        //From (virtual robot) sonar
5 Event realSonar : sonar(DISTANCE)        //From real sonar on real robot
6 Event polar   : p(Distance, Angle)
7 Event local_rover : roverState(STATE)    // STATE: ready | ...
8
9 Dispatch moveRover : cmd(CMD)           //from usercmdmanager
10
11 Context ctxMbotExecutor ip [ host="localhost" port=8029 ] -g green -httpserver //192.168.43.67
12 //Context ctxRadarBase ip [ host="localhost" port=8033 ] -standalone
13
14 /*
15 * -----
16 * The usercmdmanager handles commands sent by the user
17 * -----
18 */
19 QActor usercmdmanager context ctxMbotExecutor {
20     Plan init normal[ ]
21     transition stopAfter 3600000 //1h
22     whenEvent usercmd -> execMove
23     finally repeatPlan
24
25     Plan execMove resumeLastPlan[
26         onEvent usercmd : usercmd(robotgui(w(X))) -> forward rover -m moveRover : cmd(moveForward);
27         onEvent usercmd : usercmd(robotgui(s(X))) -> forward rover -m moveRover : cmd(moveBackward);
28         onEvent usercmd : usercmd(robotgui(a(X))) -> forward rover -m moveRover : cmd(turnLeft);
29         onEvent usercmd : usercmd(robotgui(d(X))) -> forward rover -m moveRover : cmd(turnRight);
30         onEvent usercmd : usercmd(robotgui(h(X))) -> forward rover -m moveRover : cmd(moveStop);
31         onEvent usercmd : usercmd(robotgui(f(X))) -> forward rover -m moveRover : cmd(followLine);
32         onEvent usercmd : usercmd(robotgui(x(X))) -> switchTo terminataAppl;
33         onEvent usercmd : usercmd(robotgui(unityAddr(X))) -> forward rover -m moveRover : cmd(connectToUnity)
34     ]
35     Plan terminataAppl [
36         forward rover -m moveRover : cmd(moveStop);
37         actorOp terminateSystem
38     ]
}
```

Listing 1.21. mbotExecutor.qa

### 9.1.1 A message-based command executor .

The next step our re-factoring consists in redefining the command interpreter as a codemessage-based system rather than an *event-based* system:

```

1 }
2 /*
3 * -----
4 * The rover is a an interpreter of commands
5 * -----
6 */
7
8 QActor rover context ctxMbotExecutor {
9     Rules{
10         //onRaspberry.
11         unityConfig("localhost", "unityStart.bat").
12         //unityConfig('192.168.43.229', "").
13         foundObstacle :- retract( realDistance(D) ), eval( lt, D, 20 ) .
14     }
15     Plan init normal [
16         [ !? onRaspberry] javaRun it.unibo.rover.mbotConnArduino.initRasp() ;
17         println("rover START")
18     ]
19     switchTo waitForCmd
20
21     Plan waitForCmd[ ]
22     transition stopAfter 3600000 //1h
23         whenMsg moveRover -> execMove
24     finally repeatPlan
25
26     Plan execMove resumeLastPlan[
27         printCurrentMessage;
28         onMsg moveRover : cmd(moveForward) -> javaRun it.unibo.utils.robotMixMoves.moveRobotAndAvatar("forward","40",-1");
29         onMsg moveRover : cmd(moveBackward) -> javaRun
30             it.unibo.utils.robotMixMoves.moveRobotAndAvatar("backward","40",-1");
31         onMsg moveRover : cmd(turnLeft) -> switchTo turnLeft;
32         onMsg moveRover : cmd(turnRight) -> switchTo turnRight ;
33         onMsg moveRover : cmd(moveStop) -> javaRun it.unibo.utils.robotMixMoves.moveRobotAndAvatar("stop","40",-1") ;
34         onMsg moveRover : cmd(connectToUnity) -> switchTo [ not !? unityOn ] connectToUnity ;
35         onMsg moveRover : cmd(followLine) -> javaRun it.unibo.rover.mbotConnArduino.mbotLinefollow()

```

Listing 1.22. mbotExecutor.qa

### 9.1.2 Composed robot moves .

The user-defined class `it.unibo.utils.robotMixMoves.java` defines a method that allows us to move both the real robot (if any) and the avatar. The last argument denotes the time of the move; if it is "`-1`", we intend a 'forever' move<sup>8</sup>.

```

1 package it.unibo.utils;
2 import it.unibo.qactors.akka.QActor;
3
4 public class robotMixMoves {
5
6     public static void moveRobotAndAvatar(QActor qa, String move, String speedStr, String timeStr) {
7         try {
8             int moveTime = Integer.parseInt(timeStr);
9             int moveSpeed = Integer.parseInt(speedStr);
10            switch( move ) {
11                case "forward" :{
12                    it.unibo.rover.mbotConnArduino.mbotForward(qa);
13                    qa.execUnity("rover","forward", moveTime, moveSpeed, 0);
14                    break;
15                }
16                case "backward" :{
17                    it.unibo.rover.mbotConnArduino.mbotBackward(qa);
18                    qa.execUnity("rover","backward", moveTime, moveSpeed, 0);
19                    break;

```

<sup>8</sup> In practice, we set a time of 36000 secs, i.e. 10 hours.

```

20         }
21     case "left" :{
22         it.unibo.rover.mbotConnArduino.mbotLeft(qa);
23         qa.execUnity("rover","left", moveTime, moveSpeed, 0);
24         break;
25     }
26     case "right" :{
27         it.unibo.rover.mbotConnArduino.mbotRight(qa);
28         qa.execUnity("rover","right", moveTime, moveSpeed, 0);
29         break;
30     }
31     case "stop" :{
32         it.unibo.rover.mbotConnArduino.mbotStop(qa);
33         qa.execUnity("rover","stop", moveTime, moveSpeed, 0);
34         break;
35     }
36   }
37 } catch (Exception e) { e.printStackTrace(); }
38 }
39 }
```

**Listing 1.23.** robotMixMoves.qa

Important to note that the `moveRobotAndAvatar` operation immediately return the control if the `timeStr` is `"-1"`, otherwise the operation keeps the actor blocked for the specified time (`movetime` milliseconds).

### 9.1.3 Activating the Unity system .

The task to activate the virtual environment in Unity is delegated to a user-defined class that executes a batch file.

```

1 Plan connectToUnity[
2   [ !? unityConfig(UNITYADDR, BATCH) ] javaRun it.unibo.utils.external.connectRoverToUnity( UNITYADDR, BATCH ) ;
3   addRule unityOn
4 ]
5 switchTo setAvatarInitialPosition
6 Plan setAvatarInitialPosition[
7   backwards 70 time ( 1000 ) ;
8   right    70 time ( 1000 ) ;
9   emit local_rover : roverState( ready )
10 ]
11 ]
```

**Listing 1.24.** mbotExecutor.qa

The batch file can take the following form:

```

1 echo off
2 title startUnity
3 :: This line is a comment
4 echo START UNITY
5 cd C:\Didattica2018Run\UnityRobot
6 VirtualRobotE80.exe
```

**Listing 1.25.** unityStart.bat

### 9.1.4 Executing turn moves .

The implementation of the moves that turn a robot of 90 degrees at left o right is done directly at model level:

```

1 Plan turnLeft [
2
```

```

3     javaRun it.unibo.rover.mbotConnArduino.mbotLeft();
4     [ !? unityOn ] left 40 time(750) else delay 900;
5     javaRun it.unibo.rover.mbotConnArduino.mbotStop()
6   ]
7   switchTo waitForCmd
8
9   Plan turnRight [
10    javaRun it.unibo.rover.mbotConnArduino.mbotRight();
11    [ !? unityOn ] right 40 time(750) else delay 900;
12    javaRun it.unibo.rover.mbotConnArduino.mbotStop()
13  ]

```

**Listing 1.26.** mbotExecutor.qa

### 9.1.5 Emitting polar events .

The last part of of executor system is an actor that converts sonar data into **polar** events:

```

1  */
2 QActor sonardetector context ctxMbotExecutor{
3   Plan init normal [
4     println("sonardetector STARTS ")
5   ]
6   switchTo waitForEvents
7
8   Plan waitForEvents[ ]
9   transition stopAfter 3600000 //1h
10  whenEvent sonar      -> sendToRadar,
11  whenEvent sonarDetect -> showObstacle,
12  whenEvent realSonar -> handleRealSonar
13  finally repeatPlan
14
15  Plan sendToRadar resumeLastPlan [
16    printCurrentEvent;
17    onEvent sonar : sonar(sonar1, TARGET, DISTANCE ) -> emit polar : p(DISTANCE,30) ;
18    onEvent sonar : sonar(sonar2, TARGET, DISTANCE ) -> emit polar : p(DISTANCE,120)
19  ]
20
21  Plan handleRealSonar resumeLastPlan[
22    printCurrentEvent;
23    onEvent realSonar : sonar( DISTANCE ) -> emit polar : p(DISTANCE, 0)
24  ]
25  Plan showObstacle resumeLastPlan[
26    println( "found obstacle" );
27    emit polar : p(30,90)
28  ]

```

**Listing 1.27.** mbotExecutor.qa: the sonardetect actor

## 9.2 The robot application agent

Now let us introduce a component that is able to move the robot while reacting to events:

```

1  /*
2  * =====
3  * mbotAgent.qa
4  * A robot that is able to move and to react to events
5  * =====
6  */
7 System mbotAgent
8 Event sonar      : sonar(SONAR, TARGET, DISTANCE) //From (virtual) sonar
9 Event sonarDetect : sonarDetect(X)                 //From (virtual robot) sonar
10 Event realSonar : sonar( DISTANCE )              //From real sonar on real robot
11
12 Dispatch moveRover : cmd( CMD )                  //from usercmdmanager
13
14 Context ctxMbotAgent ip [ host="localhost" port=8039 ] -g yellow //192.168.43.229

```

```

15 | Context ctxRadarBase ip [ host="localhost" port=8033 ] -standalone
16 | Context ctxMbotExecutor ip [ host="localhost" port=8029 ] -standalone //192.168.43.67
17 |
18 | QActor roveragent context ctxMbotAgent{
19 |
20 |     Plan init normal [
21 |         println("rovercontroller STARTS")
22 |     ]
23 |     switchTo goAhead
24 |
25 |     Plan goAhead[
26 |         println("rovercontroller going forward ")
27 |         sendto rover in ctxMbotExecutor -m moveRover : cmd(moveForward);
28 |     ]
29 |     transition stopAfter 600000
30 |         whenEvent sonarDetect -> handleRobotSonarDetect ,
31 |         whenEvent realSonar -> handleRobotRealSonar ,
32 |         whenEvent sonar      -> handleSonar
33 |
34 |         //Move back, far from the obstacle
35 |         Plan handleRobotSonarDetect [
36 |             sendto rover in ctxMbotExecutor -m moveRover : cmd(moveBackward);
37 |             delay 300;
38 |             sendto rover in ctxMbotExecutor -m moveRover : cmd(moveStop)
39 |         ]
40 |         switchTo goAhead //TODO
41 |
42 |         Plan handleRobotRealSonar [
43 |             onEvent realSonar : sonar( D ) -> addRule realDistance(D)
44 |         ]
45 |         switchTo [ !? foundObstacle ] avoidRealObstacle
46 |
47 |         Plan avoidRealObstacle [
48 |             sendto rover in ctxMbotExecutor -m moveRover : cmd(moveBackward);
49 |             delay 800;
50 |             sendto rover in ctxMbotExecutor -m moveRover : cmd(moveStop)
51 |         ]
52 |         switchTo goAhead //TODO
53 |
54 |         /*
55 |          * NOTE: The robot could moving either forward or backward
56 |          * Move avatar out of sonar range
57 |          *
58 |          */
59 |         Plan handleSonar [
60 |             printCurrentEvent ;
61 |             sendto rover in ctxMbotExecutor -m moveRover : cmd(moveStop) ;
62 |             println("handleSonar") ;
63 |             sendto rover in ctxMbotExecutor -m moveRover : cmd(moveForward);
64 |             delay 500;
65 |             sendto rover in ctxMbotExecutor -m moveRover : cmd(moveStop) ;
66 |             onEvent sonar : sonar(SONAR, TARGET, DISTANCE) -> println( rrrrrrrr(SONAR, TARGET, DISTANCE) )
67 |         ]
68 |         switchTo takeData
69 |
70 |         Plan takeData[
71 |             //printCurrentEvent ; //here null
72 |             println("rover turnLeft") ;
73 |             sendto rover in ctxMbotExecutor -m moveRover : cmd(turnLeft) ;
74 |             delay 1000;
75 |             println("rover turnRight") ;
76 |             sendto rover in ctxMbotExecutor -m moveRover : cmd(turnRight);
77 |             delay 1000
78 |         ]
79 |         switchTo goAhead

```

Listing 1.28. mbotAgent.qa

---

## 10 A front-end server (in Node.js)

The *QActor* software factory is able to provide a simple HTTP server on port 8080 for a Context associated to the flag `-httpserver`. This is useful during the first prototyping phase of a system, in order to introduce in a short time a web-based interface. However, during the project phase, software designers could decide to build a more advanced server, supporting features like user authentication, security, logging, etc.

Thus, in this section we face the following problem:

Design and build a front-end server for the robot-radar application to support: *i*) user authentication, *ii*) visualization of the set of sonar data detected in a given period of time, *iii*) ...

Since our goal is to reuse the application already developed, our front-end server will send messages to the robot executor of Subsection 9.1. To build the server, the `Node.js` platform and related frameworks, such as `Express`, `MongoDB`, `Angular` will be selected as the reference implementation technology.

To perform a `GET` we can use a browser), while for a `PUT/POST` we can use `codecurl` or `POSTMAN` or an `httpClient`.

Thus, to run the system we can:

- execute the robot : `src-gen/it/unibo/cctxMbotControl/MainCtxMbotAgent.java`.
- activate the `Node.js` server working on port
- perform some `PUT/GET` to port 8098.

### 10.1 The front-end server implementation

Let us define in `Node.js` and `Express` a front-end server that responds to HTTP-PUT requests with the following arguments:

- `/rover/w`: move the robot forward;
- `/rover/s`: move the robot backward;
- `/rover/a`: move the robot left;
- `/rover/d`: move the robot right;
- `/rover/h`: stop the robot ;

The server will respond to HTTP-GET requests by returning an HTML page.

For example:

```
1 curl -X PUT -d "" http://localhost:8098/rover/w
2 curl -X PUT -d "" http://localhost:8098/rover/h
3 curl http://localhost:8098
```

In this section we will follow the guidelines reported in [NodeExpressWeb.pdf](#), section 7.8.