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

Table of Contents

CaseStudy1 Showing sonar distances on a radar . . . 1
Antonio Natali

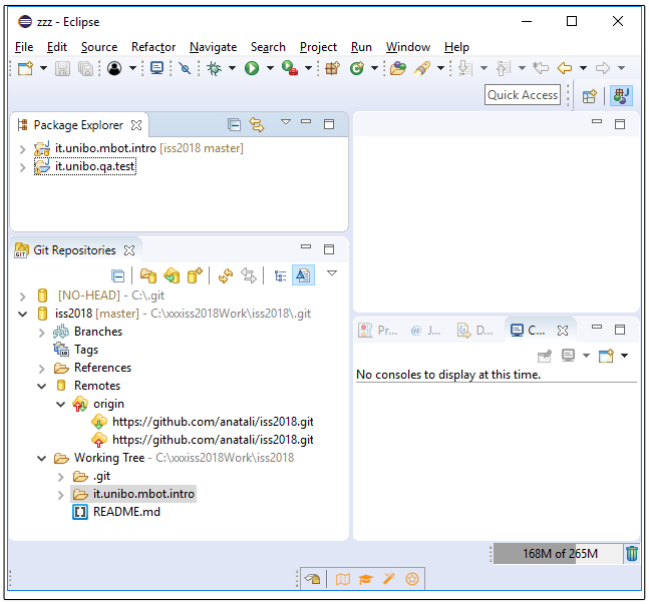
1 Starting . . . 2
1.1 Interacting with the radar . . . 3
1.2 Using the radar: a model-based approach . . . 4
1.2.1 Reusing conventional objects . . . 4
1.2.2 The radar as information handler . . . 5
1.2.3 A radar client . . . 5
1.2.4 Sending messages and emitting events . . . 6
2 The problem to solve . . . 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 *QActor* 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 A first refactoring . . . 25
9.1 The robot command executor . . . 25
9.1.1 Capturing user-commands . . . 26
9.1.2 A message-based command executor . . . 26
9.1.3 Virtual robots and real robots . . . 27
9.1.4 The command interpreter . . . 27
9.1.5 Activating the Unity system . . . 27
9.1.6 Executing turn moves . . . 28
9.1.7 Composed robot moves . . . 28
9.1.8 Emitting `polar` events . . . 29
9.2 A robot application agent . . . 29
9.2.1 An event logger . . . 30

9.3 From components to systems . . . 30
10 Another refactoring: Mqtt-based components . . . 32
10.1 The *QActor* extensions for Mqtt-interaction . . . 32
10.2 The mqtt-based robot executor . . . 33
10.2.1 Mapping events into massages . . . 34
10.2.2 The robot command executor . . . 34
10.2.3 The sonardetector . . . 35
10.3 The mqtt-based robot agent . . . 35
10.3.1 Proactive-reactive systems . . . 36
10.3.2 Non real-time systems . . . 36
10.3.3 Avoiding the loss of alarm events . . . 37
10.3.4 Event processing . . . 37
10.3.5 Alarm handling . . . 38
10.4 Testing the application agent . . . 38
11 Using a real robot . . . 40
11.1 Running the system . . . 41
12 A front-end server (in Node.js) . . . 42
12.1 The front-end server implementation . . . 42
12.1.1 Connect with the *QActor* application . . . 42
12.1.2 Handle the answer from the *QActor* node . . . 43
12.1.3 Create the server . . . 43
12.1.4 Emit an event for the QActor application . . . 44
12.1.5 Build an (HTML) response . . . 44
55

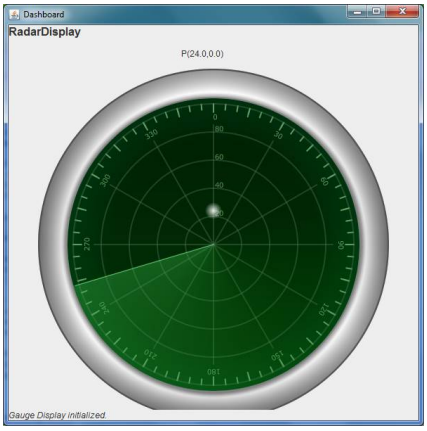
1 Starting

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[1] 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:



[1] 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:

```
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:

```
var net   = require('net');
var host  = "localhost";
var port  = 8033;

console.log('connecting to ' + host + ":" + port);
var conn = net.connect({ port: port, host: host });
conn.setEncoding('utf8');

// when receive data back, print to console
conn.on('data',function(data) {
    console.log(data);
});
// when server closed
conn.on('close',function() {
    console.log('connection is closed');
});
conn.on('end',function() {
    console.log('connection is ended');
});
```

Listing 1.1. TcpClientToRadar.js: set up a connection

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

```
var sendMsg = function( msg ){
    try{
        console.log("SENDING " + msg );
        conn.write(msg+"\n"); //Asynchronous!!!
    }catch(e){
        console.log("ERROR " + e );
    }
}

function sendMsgAfterTime( msg, delay ){
    setTimeout( function(){ sendMsg( msg ); }, delay);
}
```

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:

```
var msgNum=1;

sendMsgAfterTime("msg(polarMsg,dispatch,jsSource,radarguibase, p(50,30)," + msgNum++ +")", 1000);
sendMsgAfterTime("msg(polarMsg,dispatch,jsSource,radarguibase, p(50,90)," + msgNum++ +")", 2000);
sendMsgAfterTime("msg(polarMsg,dispatch,jsSource,radarguibase, p(50,150)," + msgNum++ +")", 3000);

//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:

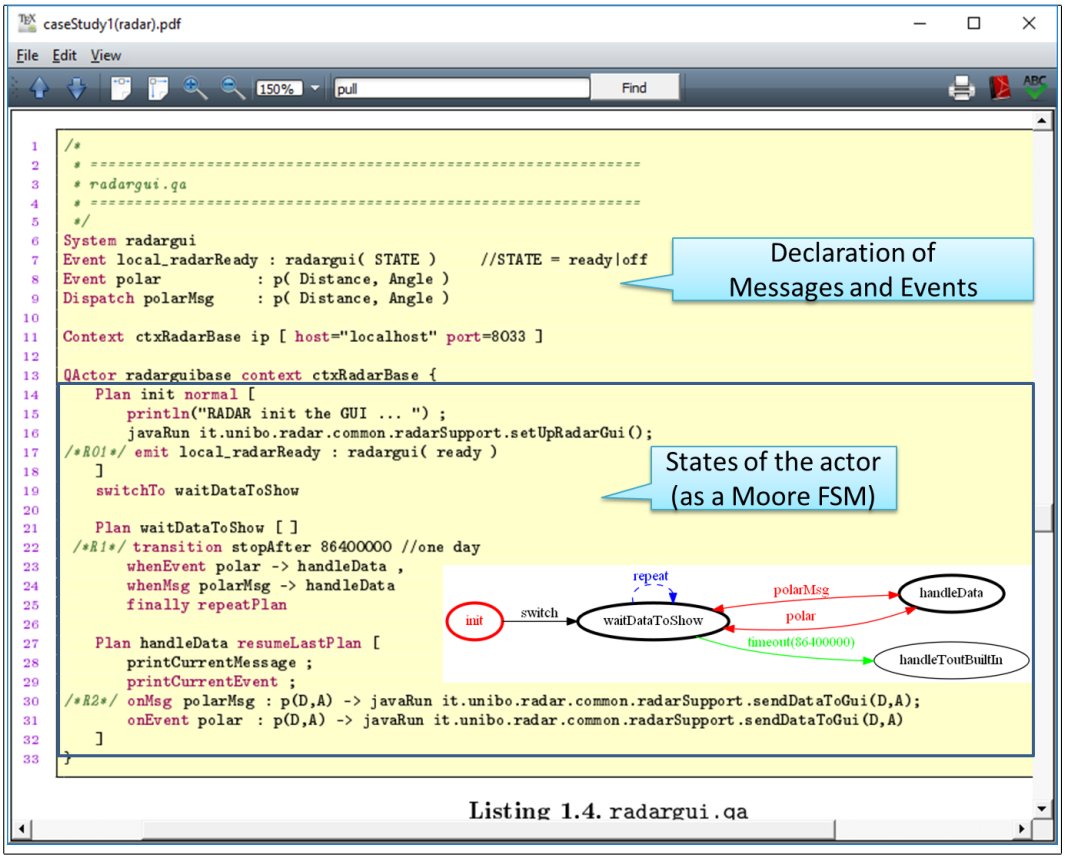
```
connecting to localhost:8033
SENDING msg(polarMsg,dispatch,jsSource,radarguibase, p(50,30),1)
SENDING msg(polarMsg,dispatch,jsSource,radarguibase, p(50,90),2)
SENDING msg(polarMsg,dispatch,jsSource,radarguibase, p(50,150),3)
connection is ended
connection is closed
```

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:



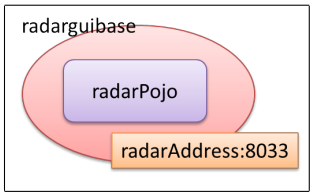

```
/*
 * ================================================================
 * radargui.qa
 * ================================================================
 */
System radargui
Event local_radarReady : radargui( STATE )     //STATE = ready|off
Event polar            : p( Distance, Angle )
Dispatch polarMsg      : p( Distance, Angle )

Context ctxRadarBase ip [ host="localhost" port=8033 ]

QActor radarguibase context ctxRadarBase {
    Plan init normal [
        println("RADAR init the GUI ... ") ;
        javaRun it.unibo.radar.common.radarSupport.setUpRadarGui();
/*R01*/ emit local_radarReady : radargui( ready )
    ]
    switchTo waitDataToShow

    Plan waitDataToShow [ ]
 /*R1*/ transition stopAfter 86400000 //one day
        whenEvent polar -> handleData ,
        whenMsg polarMsg -> handleData
        finally repeatPlan

    Plan handleData resumeLastPlan [
        printCurrentMessage ;
        printCurrentEvent ;
/*R2*/ onMsg polarMsg : p(D,A) -> javaRun it.unibo.radar.common.radarSupport.sendDataToGui(D,A);
        onEvent polar  : p(D,A) -> javaRun it.unibo.radar.common.radarSupport.sendDataToGui(D,A)
    ]
}
```

Listing 1.4. radargui.qa

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[2]:

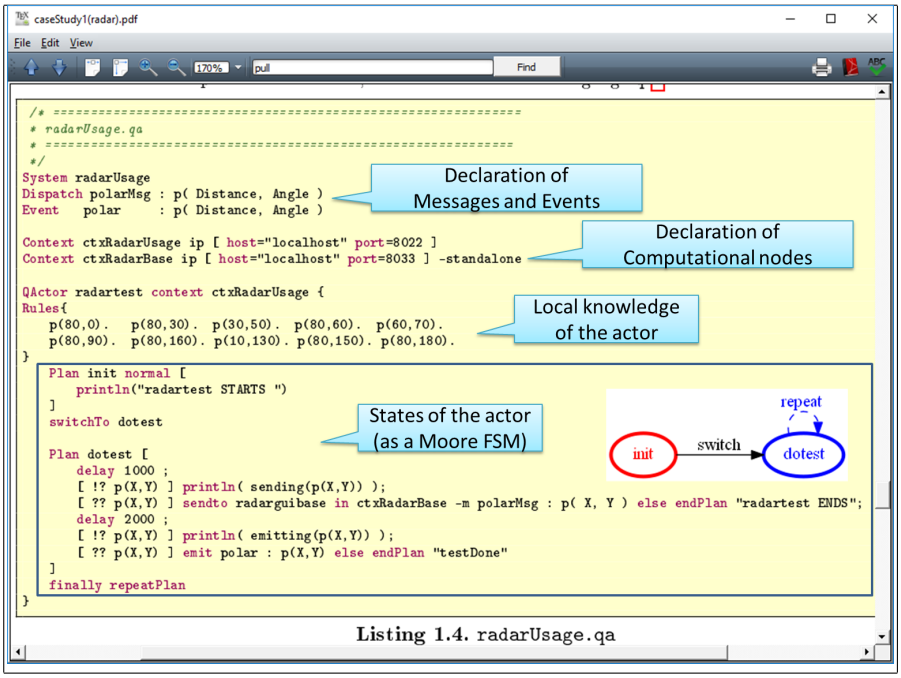
```
1 Dispatch polarMsg : p( Distance, Angle )
2 Event    polar    : p( Distance, Angle )
```

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[3]:

[2] The event local_radarReady is a local event used internally.
[3] The code is in the file it.unibo.mbot.intro/src/radarUsage.qa.



Listing 1.4. radarUsage.qa

The model states that:

1. Our `radarUsage` system is a distributed system composed of two computational nodes (`Contexts`).
2. 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.
3. Our radar client is modelled as a *QActor* (`radartest`): it works as a finite state machine that (in the state `dotest`) sends messages and emits events. We will expand this point in in Subsection 1.2.4.
4. The messages and events involved in our system are *the same* defined in the radar model:

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

5. 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 `dotest`, 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:

```
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 known the name of the message destination, that must be another *QActor*. This fact is reflected in the sentence:

```
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:

```
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:

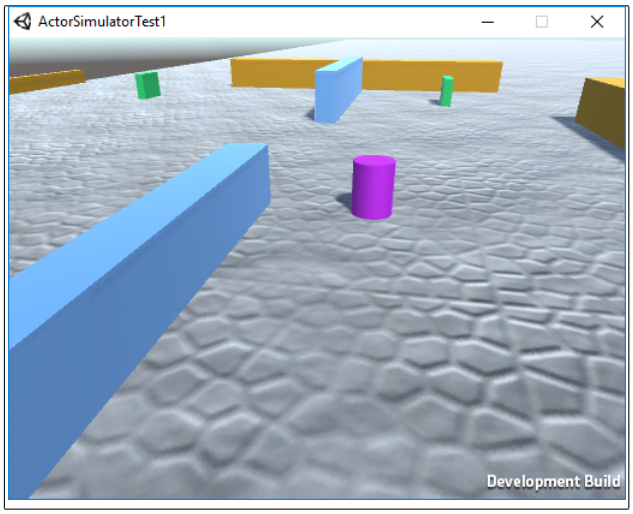
1. what is a `mbot`?
2. what is the virtual environment?
3. in which way we can obtain data from a virtual or from a real sonar?

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`)[4].

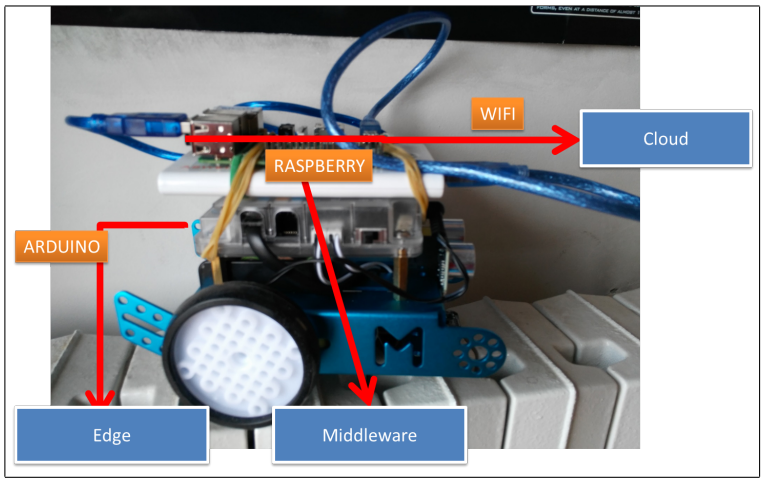
[4] The file is available in `it.unibo.issMaterial/issdocs/Material`.

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.

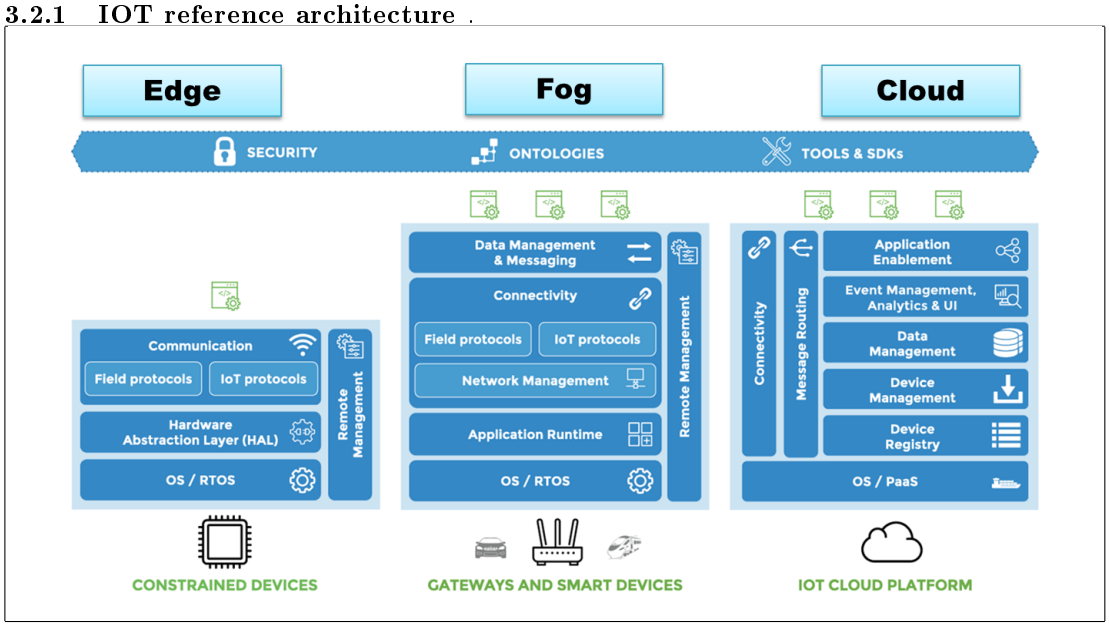
As regards the other questions, our goal is to build a model for the `mbot` and a model for the real sonar. The goal of these models is to clarify how we can exchange information with the corresponding entities of our `application domain`. The internal structure and the internal behaviour of the entities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

```
/*
 * ================================================================
 * mbotControl.qa
 * ================================================================
 */
System mbotControl
Event usercmd    : usercmd(CMD)
Event sonar      : sonar(SONAR, TARGET, DISTANCE) //From (virtual) sonar
Event sonarDetect : sonarDetect(X)             //From (virtual robot) sonar
Event realSonar  : sonar( DISTANCE )           //From real sonar on real robot
Event   polar    : p( Distance, Angle )
Event unityAddr  : unityAddr( ADDR )           //From user interface

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:

```
QActor rover context ctxMbotControl {
   Plan init normal [
          println("rover START")
       ]
       switchTo waitUserCmd

   Plan waitUserCmd[  ]
   transition stopAfter 600000
       whenEvent usercmd -> execMove
   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):

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

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

Listing 1.6. `mbotControl.qa`: command intepreter

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:

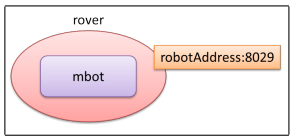
```
    Plan connectToUnity resumeLastPlan[
        onEvent usercmd : usercmd(robotgui(unityAddr(ADDR))) -> println(ADDR);//ADDR=localhost in this version
        [ !? unityOn ] println( "UNITY already connected" )
    ]
    switchTo [ not !? unityOn ] doconnectToUnity

    Plan doconnectToUnity resumeLastPlan[
        println("ACTIVATING UNITY. Wait a moment ... ") ;
        javaRun it.unibo.commToRadar.polarToRadar.customExecute("C:/Didattica2018Run/unityStart.bat") ;
        delay 10000;  //wait until Unity activated
        connectUnity "localhost" ;
        createUnityObject "rover" ofclass "Prefabs/CustomActor" ;
        backwards 70 time ( 800 ) ;
        right 70 time ( 1000 ) ; //position
        addRule unityOn
    ]
```

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:

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

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

```
    Plan turnLeft resumeLastPlan [
        javaRun it.unibo.rover.mbotConnArduino.mbotLeft();
        [ !? unityOn ] left 40 time(750) else delay 900;
        javaRun it.unibo.rover.mbotConnArduino.mbotStop()
    ]
    Plan turnRight resumeLastPlan [
        javaRun it.unibo.rover.mbotConnArduino.mbotRight();
        [ !? unityOn ] right 40 time(750) else delay 900;
        javaRun it.unibo.rover.mbotConnArduino.mbotStop()
    ]
    Plan stopTheRobot resumeLastPlan[
```

```
        stop 40 time ( 10 );
        javaRun it.unibo.rover.mbotConnArduino.mbotStop()
    ]
```

Listing 1.8. `mbotControl.qa: simple moves`

The `rover` runs on the `RaspberryPi`[5] 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 is 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:

```
    Plan moveForward resumeLastPlan[
        javaRun it.unibo.rover.mbotConnArduino.mbotForward()
    ]
    reactive onward 40 time( 15000 )
        whenEnd             -> endOfMove
        whenTout 30000      -> handleTout
        whenEvent sonarDetect -> handleRobotSonarDetect
        or whenEvent sonar  -> handleSonar
        or whenEvent usercmd -> execMove

    Plan moveBackward resumeLastPlan[
        javaRun it.unibo.rover.mbotConnArduino.mbotBackward()
    ]
    reactive backwards 40 time ( 15000 )
        whenEnd             -> endOfMove
        whenTout 30000      -> handleTout
//      whenEvent sonarDetect -> handleObstacle //no sensor on robot back
        whenEvent sonar     -> handleSonar
        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:

```
QActor sonardetector context ctxMbotControl{
    Plan init normal [ ]
    switchTo waitForEvents

    Plan waitForEvents[ ]
    transition stopAfter 600000
        whenEvent sonar     ->  sendToRadar,
        whenEvent sonarDetect -> sendToRadar,
        whenEvent realSonar ->  sendToRadar
    finally repeatPlan

    Plan sendToRadar resumeLastPlan [
        printCurrentEvent;
        onEvent realSonar : sonar( DISTANCE )            -> emit polar : p(DISTANCE, 0) ;
        onEvent sonar     : sonar(sonar1, TARGET, DISTANCE ) -> emit polar : p(DISTANCE,30) ;
        onEvent sonar     : sonar(sonar2, TARGET, DISTANCE ) -> emit polar : p(DISTANCE,120) ;
        onEvent sonarDetect : sonarDetect(TARGET)        -> switchTo showObstcle
    ]

    Plan showObstcle resumeLastPlan[
```

[5] 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

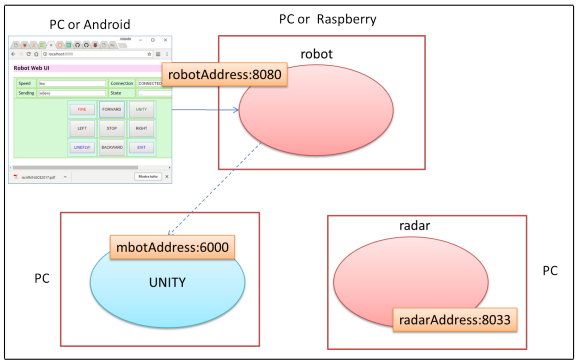
Note that the choice to emit events (and not to send messages) avoid the need to known the name of any destination actor.

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.

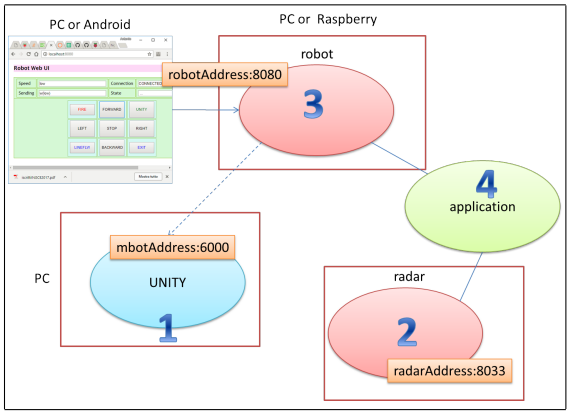
4 Problem analysis

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

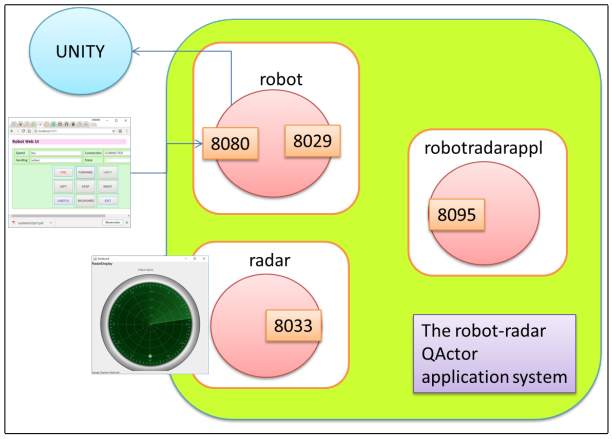
The robot is a source of information expressed as *QActor* events of the form polar:p(DISTANCE,ANGLE)). This kind of information is available by any other software component belonging to a *QActor* system. Since also the radar is implemented as a *QActor* application, we can state that the logical architecture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

```
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):

```
var net = require('net');
var utils = require('./utils'); //for handling uncaughtException

/*
 * ------------------------------------------------
 * CONNECTION TO RADAR
 * ------------------------------------------------
 */
var radarhost  = "localhost";
var radarport  = 8033;
var msgNum     = 1;

console.log('connecting to RADAR at ' + radarhost + ":" + radarport);
var conn = net.connect({ port: radarport, host: radarhost });
conn.setEncoding('utf8');

var emitEventForRadar = function( data ){
    try{
        var msg = "msg(polar,event,jsSource,none," + data + "," + msgNum++ +")";
        console.log("SENDING " + msg );
        conn.write(msg+"\n"); //Asynchronous!!!
    }catch(e){
        console.log("ERROR " + e );
    }
}
```

Listing 1.12. `nodeCode/PolarToRadar.js: connecetion to the radar`

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

```
/*
 * ------------------------------------------------
 * SERVER
 * ------------------------------------------------
 */
var host  = "localhost";
var port  = 8057;

var server = net.createServer(function(socket) {
```

```
    socket.on('data', function( data ) { //data has the form p(DISTANCE,ANGLE)
        var dataNoWhiteSpaces = (""+data).trim();
        emitEventForRadar( dataNoWhiteSpaces );
      });

    socket.on('end', socket.end);
});

//STARTING
emitEventForRadar("p(70, 0)"); //just to show
console.log("PolarToRadar starts at " + port);
server.listen(port, host );
```

Listing 1.13. `nodeCode/PolarToRadar.js: connecetion 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:

```
    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.

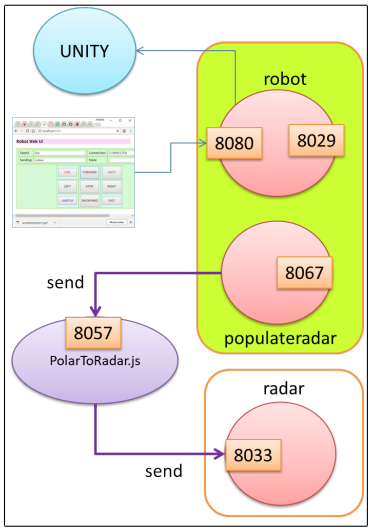
5.2 A *QActor* system as an adapter

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`:

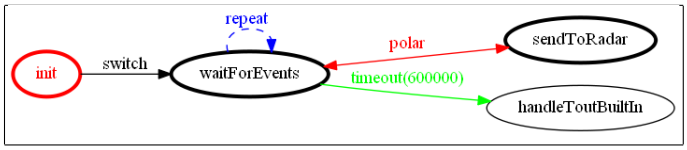
```
System robotToRadarAdapter
Event   polar  : p( Distance, Angle )

Context ctxRobotToRadarAdapter ip [ host="localhost" port=8067 ]
Context ctxMbotControl      ip [ host="localhost"  port=8029 ]

QActor populateradar context ctxRobotToRadarAdapter{
    Plan init normal [ ]
    switchTo waitForEvents

    Plan waitForEvents[ printCurrentEvent ]
    transition stopAfter 600000
        whenEvent polar : p( D, A ) do
            javaRun it.unibo.commToRadar.polarToRadar.sendPolarToNodeServer( D, A )
    finally repeatPlan
}
```

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

```
System robotToRadarActivator

Context ctxRobotToRadarActivator ip [ host="localhost" port=8055 ]

QActor componentactivator context ctxRobotToRadarActivator {
    Plan init normal [
        //The radar takes some time to start and to end its testing phase;
        println("ACTIVATING RADAR") ;
        javaRun it.unibo.commToRadar.polarToRadar.customExecute("C:/Didattica2018Run/radarStart.bat") ;

        //The robot activates Unity, if required;
        delay 1000;
        println("ACTIVATING ROBOT") ;
        javaRun it.unibo.commToRadar.polarToRadar.customExecute("C:/Didattica2018Run/robotStart.bat") ;

        //PolarToRadar requires the radar;
        delay 10000;
        println("ACTIVATING NODE APPLICATION") ;
        nodeOp "./nodeCode/PolarToRadar.js -o" //WARNING: the actor is engaged since PolarToRadar.js waits
    ]
}
```

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:

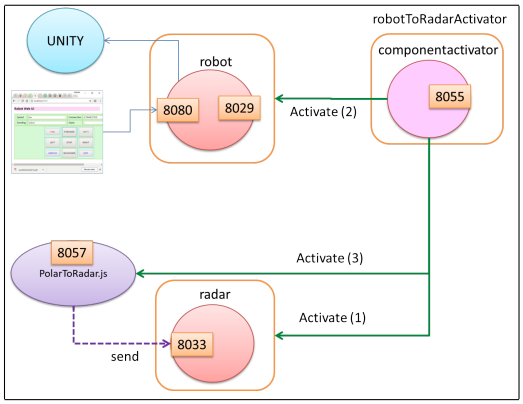
```
    public static void customExecute(QActor qa, String cmd) {
        try {
            Runtime.getRuntime().exec(cmd);
        } catch (IOException e) { e.printStackTrace(); }
    }
```

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:

```
echo off
title radarStart
echo START RADAR
cd C:\Didattica2018Run\it.unibo.ctxRadarBase.MainCtxRadarBase-1.0
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) [6].
2. Run the application of Subsection 4.1.

A batch file that does this work could be:

```
echo off
 title robotRadarApplStart
 echo START THE RADAR
 cd .\it.unibo.ctxRadarBase.MainCtxRadarBase-1.0
 START /B java -jar it.unibo.qactor.radar-1.0.jar > ../logRadar.txt
 cd ..
 echo please press CR
 PAUSE

 echo START THE ROBOT
 cd .\it.unibo.ctxMbotControl.MainCtxMbotControl-1.0
 START /B java -jar it.unibo.mbot.intro-1.0.jar > ../logRobot.txt
 cd ..
 PAUSE

 echo START THE ROBOT-RADAR APPLICATION
 cd .\it.unibo.ctxRobotRadarAppl.MainCtxRobotRadarAppl-1.0
 START /B java -jar it.unibo.robotRadarAppl-1.0.jar > ../logRadarRobotAppl.txt
 cd ..
 PAUSE
```

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

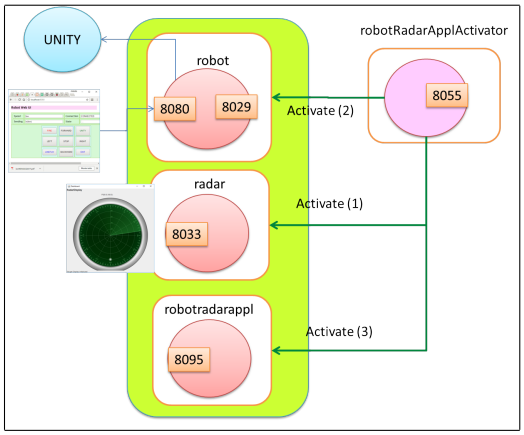
```
System robotRadarApplActivator

Context ctxRobtRadarApplActivator ip [ host="localhost" port=8078 ]

QActor robotradarapplactivator context ctxRobtRadarApplActivator {
   Plan init normal [
       //The radar takes some time to start and to end its testing phase;
       println("ACTIVATING RADAR") ;
//     javaRun it.unibo.commToRadar.polarToRadar.customExecute("C:/Didattica2018Run/radarStart.bat") ;
       javaOp "customExecute(\"C:/Didattica2018Run/radarStart.bat\")" ;

       //The robot activates Unity, if required;
       delay 1000;
       println("ACTIVATING ROBOT") ;
//     javaRun it.unibo.commToRadar.polarToRadar.customExecute("C:/Didattica2018Run/robotStart.bat") ;
       javaOp "customExecute(\"C:/Didattica2018Run/robotStart.bat\")" ;

       delay 10000;
       println("ACTIVATING THE SYSTEM") ;
//     javaRun it.unibo.commToRadar.polarToRadar.customExecute("C:/Didattica2018Run/robotRadarStart.bat")
       javaOp "customExecute(\"C:/Didattica2018Run/robotRadarStart.bat\")"
   ]
}
```

Listing 1.18. `robotRadarApplActivator.qa`

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

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

[6] Note that the activation of the `PolartoRadar` server is useless if we do not use the `robotToRadarAdpater` extension.


UNITY
robot
8080
8029
robotRadarApplActivator
8055
Activate (2)
radar
8033
Activate (1)
robotradarappl
8095
Activate (3)

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:

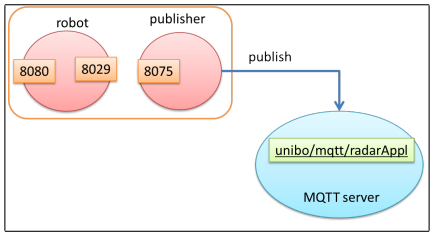
1. Introduce proper extensions like done in Subsection 5.2.
2. Modify the behaviour of the components by accessing their source code

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

7.1 Extending the robot as a publisher of events

```
System robotToPublisher
Event polar : p( Distance, Angle )

//pubSubServer "tcp://m2m.eclipse.org:1883"
pubSubServer "tcp://test.mosquitto.org:1883"

Context ctxRobotToPublisher ip [ host="localhost" port=8075 ] -g green
Context ctxMbotControl    ip [ host="localhost"  port=8029 ]  -standalone

QActor roboteventpublisher context ctxRobotToPublisher{
    Plan init normal [
        connectAsPublisher "unibo/mqtt/radarAppl" ;
        println("roboteventpublisher STARTED")
    ]
    switchTo waitForEvents

    Plan waitForEvents[ printCurrentEvent ]
    transition stopAfter 600000
        whenEvent polar : p( Distance, Angle ) do //println( p( Distance, Angle ) )
            publishEvent "unibo/mqtt/radarAppl" -e polar : p( Distance, Angle )
    finally repeatPlan
}
```

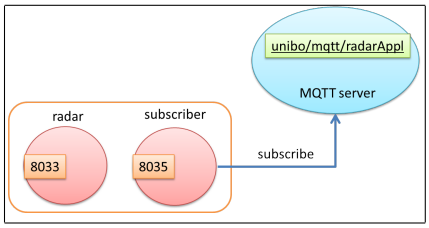
Listing 1.19. `robotToPublisher.qa`



7.2 Extending the radar as a subscriber of events

```
System radarToSubscriber
Event   polar  : p( Distance, Angle )

//pubSubServer "tcp://m2m.eclipse.org:1883"
pubSubServer "tcp://test.mosquitto.org:1883"

Context ctxRadarToSubscriber ip [ host="localhost" port=8035 ]
Context ctxRadar ip [ host="localhost" port=8033 ] -standalone

QActor radareventsubscriber context ctxRadarToSubscriber{
    Plan init normal [
        connectAsSubscriber "unibo/mqtt/radarAppl" ;
        println("radareventsubscriber START")
    ]
    switchTo waitForEvents

    Plan waitForEvents[
        printCurrentEvent;
        delay 1000
    ]
    transition stopAfter 600000
        whenEvent polar : p( D, A )  do println ( perceived( p( D, A ) ) )
    finally repeatPlan
}
```

Listing 1.20. `robotToPublisher.qa`



To run the system we [7]:

- 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`.

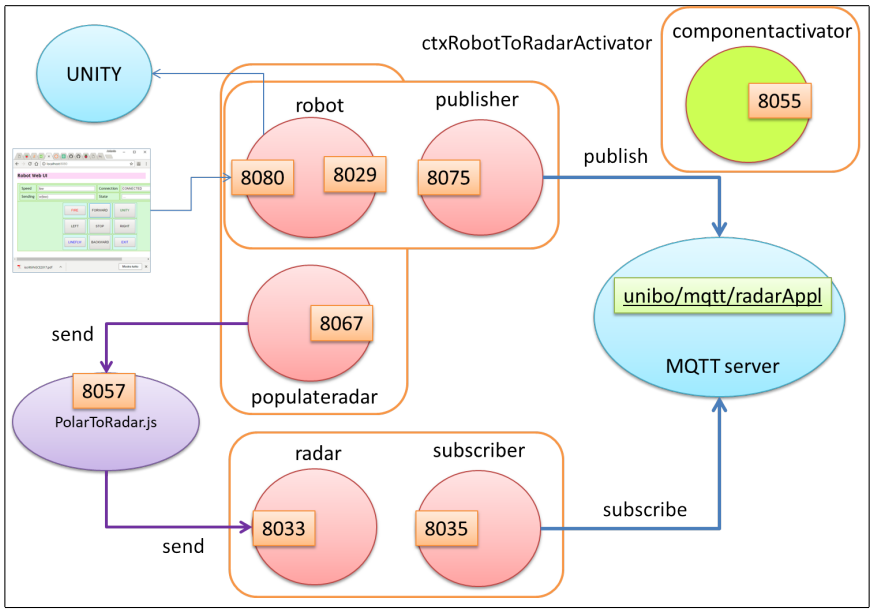
[7] Note that the VPN (if any) must be disabled.

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 **??**, 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 :



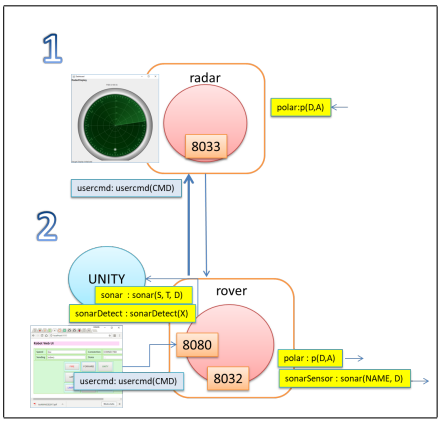
Thus, our final, concrete project architecture will be different from the logical architecture of our first prototype introduced in Subsection 4.1. However all the constraints imposed by the logical architecture are preserved. Moreover, several components introduced in the logical architecture are reused,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

In the following sections we will introduce other project architectures, in order to give to our solution more modularity and more flexibility.

9 A first 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* (named `rover`) on the (modified) Unity system. See Subsection 9.1.
2. Allow a robot executor to work on a `RaspberryPi` and to interact with an `Arduino` microcontroller that manages the motors and the sensors. See Subsection 9.1. At this stage, a radar-robot system like that analysed of Section 4 can be represented as in the following picture:



3. Define a robot agent, i.e. a component (microservice) that exploits the robot executor to implement some application. For example,

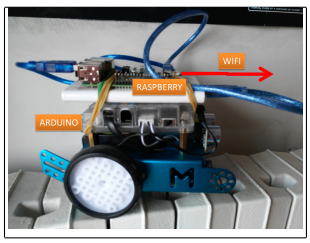
   > we want to move the robot in a controlled way, by introducing some obstacle avoidance policy and some movement constraint, e.g. stop the robot when it is detected by a fixed virtual sonar.

   The intent is to make the specification of the behavior as simple and clear as possible, by recurring to *reactive actions* (as done in the model of Subsection 3.3.4) only when they are really needed. See Subsection 9.2.

In the following, we will use the version `1.5.13` of the *QActor* plugins that allows us to express `guarded action sequences`.

9.1 The robot command executor

Our new robot executor must be able to handle both a *real* robot built on a `RaspberryPi + Arduino` platform and a virtual robot *avatar* (named `rover`) running on the (modified) Unity system.



During the testing phase, it can be useful to connect Arduino to the PC used for software development, rather than to the RaspberryPi.

Since we want solve the same problem of Section 2, our software system requires also the `radar`:

```
System mbotExecutor
Event usercmd    : usercmd(CMD)                //from web gui
Event sonar      : sonar(SONAR, TARGET, DISTANCE) //From (virtual) sonar
Event sonarDetect : sonarDetect(X)             //From (virtual robot) sonar
Event realSonar  : sonar( DISTANCE )           //From real sonar on real robot
Event   polar    : p( Distance, Angle )

Dispatch moveRover : cmd( CMD )                //from usercmdmanager

Context ctxMbotExecutor ip [ host="localhost" port=8029 ] -httpserver
Context ctxRadarBase  ip [ host="localhost" port=8033 ] -standalone
```

Listing 1.21. `mbotExecutor.qa`

9.1.1 Capturing user-commands .

The re-factoring of the robot executor of Subsection 3.3 starts with introducing a component that transforms a user-command event into a message sent to the robot.

```
QActor usercmdmanager context ctxMbotExecutor {
   Plan init normal[ ]
   transition stopAfter 3600000  //1h
       whenEvent usercmd -> execMove
       finally repeatPlan

   Plan execMove resumeLastPlan[
       onEvent usercmd : usercmd( robotgui(w(X)) ) -> forward rover -m moveRover : cmd(moveForward);
       onEvent usercmd : usercmd( robotgui(s(X)) ) -> forward rover -m moveRover : cmd(moveBackward);
       onEvent usercmd : usercmd( robotgui(a(X)) ) -> forward rover -m moveRover : cmd(turnLeft);
       onEvent usercmd : usercmd( robotgui(d(X)) ) -> forward rover -m moveRover : cmd(turnRight);
       onEvent usercmd : usercmd( robotgui(h(X)) ) -> forward rover -m moveRover : cmd(moveStop);
       onEvent usercmd : usercmd( robotgui(f(X)) ) -> forward rover -m moveRover : cmd(followLine);
       onEvent usercmd : usercmd( robotgui(x(X)) ) -> { //action sequence
           forward rover -m moveRover : cmd(moveStop);
           actorOp terminateSystem
       } ;
       onEvent usercmd : usercmd( robotgui(unityAddr(X)) ) ->
                                              forward rover -m moveRover : cmd(connectToUnity)
   ]
}
```

Listing 1.22. `mbotExecutor.qa`

9.1.2 A message-based command executor .

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

```
QActor rover context ctxMbotExecutor {
   Rules{   //onRaspberry.
            unityConfig("localhost", "unityStart.bat"). //(1) for testing
            //unityConfig("192.168.43.229", "").      //(2) if onRaspberry is on
   }
   Plan init normal [
       [ !? onRaspberry] javaRun it.unibo.rover.mbotConnArduino.initRasp() ;
       println("rover START")
   ]
   switchTo waitForCmd

   Plan waitForCmd[ ]
   transition stopAfter 3600000  //1h
       whenMsg moveRover -> execMove
```

Listing 1.23. `mbotExecutor.qa`: the interpreter

9.1.3 Virtual robots and real robots .

The fact onRaspberry should be present in the knowledge base of the rover actor when the code has to be deployed on the RaspberryPi connected with the Arduino-based mbot (see Subsection 3.2 and Subsection 3.3.3). The class it.unibo.rover.mbotConnArduino is defined by the application designer as a 'bridge' to the Arduino controller; for the details see Section 11.

The fact unityConfig/2 declares the address of the program that activates the virtual Unity environment. The case (1), makes reference to a batch file; thus, it is significant only when the executor runs on a PC[8]. The case (2) must be used when the executor runs on the RaspberryPi; in this case Uity hase to be started manually on the PC.

9.1.4 The command interpreter .

Now, the command interpreter can be defined into a state that executes different action sequences on behalf of the current received message:

```
1   Plan execMove resumeLastPlan[
2       printCurrentMessage;
3       onMsg moveRover : cmd(moveForward) -> javaRun it.unibo.utils.robotMixMoves.moveRobotAndAvatar("forward","40","0");
4       onMsg moveRover : cmd(moveBackward) -> javaRun
              it.unibo.utils.robotMixMoves.moveRobotAndAvatar("backward","40","0");
5       onMsg moveRover : cmd(turnLeft)   -> {
6           javaRun it.unibo.rover.mbotConnArduino.mbotLeft();
7           [ !? unityOn ] left 40 time(750) else delay 900;
8           javaRun it.unibo.rover.mbotConnArduino.mbotStop()
9       };
10      onMsg moveRover : cmd(turnRight)  -> {
11          javaRun it.unibo.rover.mbotConnArduino.mbotRight();
12          [ !? unityOn ] right 40 time(750) else delay 900;
13          javaRun it.unibo.rover.mbotConnArduino.mbotStop()
14      } ;
15      onMsg moveRover : cmd(moveStop)   -> javaRun it.unibo.utils.robotMixMoves.moveRobotAndAvatar("stop","40","0") ;
16      onMsg moveRover : cmd(connectToUnity) -> {
17           [ not !? unityOn ] {
18              //activate Unity [ only in case (1) ], else simply connect to UNITYADDR:6000
19              [ !? unityConfig(UNITYADDR, BATCH) ]
20                     javaRun it.unibo.utils.external.connectRoverToUnity( UNITYADDR, BATCH ) ;
21              addRule unityOn ;
22              //setAvatarInitialPosition
23               backwards 70 time ( 1000 ) ;
24               right    70 time ( 1000 )
25          }else println("UNITY ALREADY ACTIVE")
26       };
27      onMsg moveRover : cmd(followLine) -> javaRun it.unibo.rover.mbotConnArduino.mbotLinefollow()
28  ]
29  switchTo waitForCmd
```

Listing 1.24. `mbotExecutor.qa`: the interpreter

9.1.5 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 of the following form:

```
1 echo off
2  title startUnity
3  echo START UNITY
4  cd C:\Didattica2018Run\UnityRobot
5  VirtualRobotE80.exe
```

Listing 1.25. `unityStart.bat`

[8] We consider not appropriate to run Unity on the RaspberryPi.

A said before, we do not intend to run Unity on the RaspberryPi; thus the command moveRover : cmd(connectToUnity) is appropriate only when the executor runs on a PC. The user-defined utility operation it.unibo.utils.connectRoverToUnity does not activate Unity if the batch file is "" (case (1)) of unitConfig/2), but always performs a connection to Unity. Thus, in case (2)) the executor running on the real robot can move also the virtual robot.

A further re-factoring should make things more clean, by splitting the real-robot case from the virtual-robot case[9].

9.1.6 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 (lines 5 and 10 of the interpreter). This in order to show the move policy at model level and to show the user-defined classes that move the *real* robot (the class it.unibo.rover.mbotConnArduino) and the built-in moves for the *avatar* (ahead, backwards, left, right, stop) .

As an alternative, we could delegate this 'complex move' to a user-defined method, as done for the other commands (see Subsection 9.1.7).

9.1.7 Composed robot moves .

The user-defined class it.unibo.utils.robotMixMoves.java defines a method that allows to move both the *real* robot (if any) and the *avatar*.

```
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;
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 1.26. robotMixMoves.java

[9] This re-factoring will be done in Section 10.

The last argument denotes the time of the move; if it is "0", we intend a 'forever' move[10]. Important to note that the moveRobotAndAvatar operation immediately returns the control if the timeStr == "0", otherwise the operation keeps the actor blocked for the specified time (in milliseconds).

9.1.8 Emitting polar events .

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

```
QActor sonardetector context ctxMbotExecutor{
    Plan init normal [
        println("sonardetector STARTS ")
    ]
    switchTo waitForEvents

    Plan waitForEvents[ ]
    transition stopAfter 3600000  //1h
        whenEvent sonar     ->  sendToRadar,
        whenEvent sonarDetect -> showObstcle,
        whenEvent realSonar ->  handleRealSonar
    finally repeatPlan

    Plan sendToRadar resumeLastPlan [
        printCurrentEvent;
        onEvent sonar : sonar(sonar1, TARGET, DISTANCE ) -> emit polar : p(DISTANCE,30) ;
        onEvent sonar : sonar(sonar2, TARGET, DISTANCE ) -> emit polar : p(DISTANCE,120)
    ]
    Plan handleRealSonar resumeLastPlan[
        printCurrentEvent;
        onEvent realSonar : sonar( DISTANCE ) -> emit polar : p(DISTANCE, 0)
    ]
    Plan showObstcle resumeLastPlan[
        println( "found obstacle" );
        emit polar : p(30,90)
    ]
}
```

Listing 1.27. mbotExecutor.qa: the sonardetect actor

9.2 A robot application agent

Now let us introduce a component that is able to move the robot by considering also the events emitted by the sonars:

```
System mbotAgent
Event usercmd    : usercmd(CMD)
Event sonar      : sonar(SONAR, TARGET, DISTANCE) //From (virtual) sonar
Event sonarDetect : sonarDetect(X)            //From (virtual robot) sonar
Event realSonar  : sonar( DISTANCE )          //From real sonar on real robot
Event   polar    : p( Distance, Angle )       //Just to perform an experiment

Context ctxMbotAgent  ip [ host="localhost" port=8039 ]
Context ctxMbotExecutor ip [ host="localhost" port=8029 ] -standalone //(0)

QActor roveragent context ctxMbotAgent {
    Plan init normal [
        println("roveragent STARTS")
//      emit usercmd : usercmd( robotgui(w(low)) )
    ]
    switchTo lookAtSonars
    Plan lookAtSonars[
//      emit polar : p( 70, 160 );  //(2) event propagation experiment
        println("roveragent lookAtSonars ")
    ]
    transition stopAfter 600000 //10 min
        whenEvent sonar      -> handleFixedSonar ,
```

[10] In practice, we set a time of 36000 secs, i.e. 10 hours.

```
23          whenEvent sonarDetect -> handleSonar ,
24          whenEvent realSonar  -> handleSonar
25
26      //The robot is moving either forward or backward: move it out of sonar range.
27      Plan handleFixedSonar [
28              delay 500 ;   //avoid to raise sonar events again
29              emit usercmd : usercmd( robotgui(h(low)) )
30      ]
31      switchTo lookAtSonars
32
33      //Obstacle
34      Plan handleSonar [
35          printCurrentEvent ;
36          emit usercmd : usercmd( robotgui(s(low)) ) ; //retrogress
37          delay 500 ;   //avoid to raise sonar events again
38          emit usercmd : usercmd( robotgui(h(low)) )
39      ]
40      switchTo lookAtSonars
41  }//roveragent
```

Listing 1.28. `mbotAgent.qa`

Reactive actions are not required here, since all the commands that move a robot (real or virtual) are operations logically executed in 'asynchronous way', i.e. they return immediately the control.

9.2.1 An event logger .

Let us introduce in the agent also an event logger as an actor that registers the important events occurring in the system:

```
1   QActor evlogagent context ctxMbotAgent -g yellow {
2      Plan init normal [ println("evlogagent STARTS") ]
3      switchTo doWork
4
5      Plan doWork[]
6      transition  stopAfter 60000 //1min
7          whenEvent polar       -> logEvent,
8          whenEvent usercmd     -> logEvent,
9          whenEvent sonar       -> logEvent,
10         whenEvent realSonar   -> logEvent,
11         whenEvent sonarDetect -> logEvent
12     finally repeatPlan
13
14     Plan  logEvent resumeLastPlan[
15         printCurrentEvent
16     ]
17  }
```

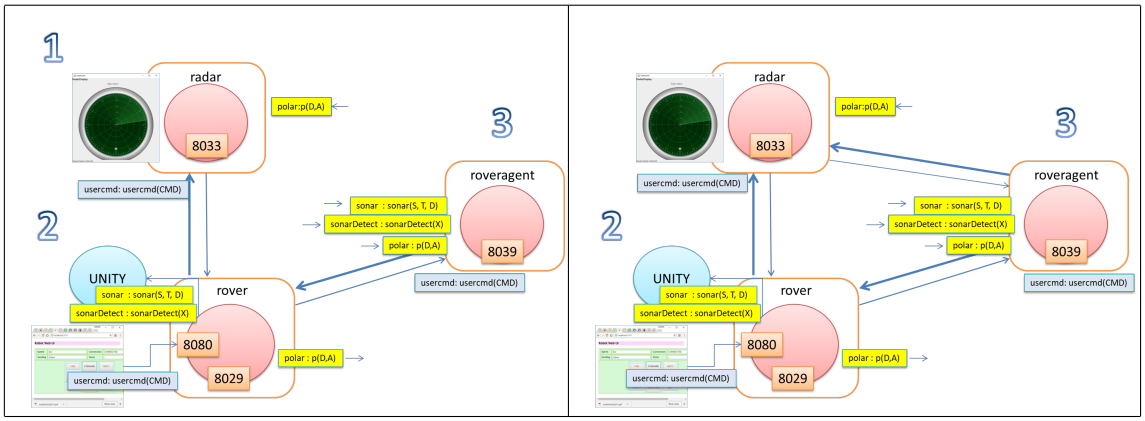
Listing 1.29. `mbotAgent.qa`: an event logger

9.3 From components to systems

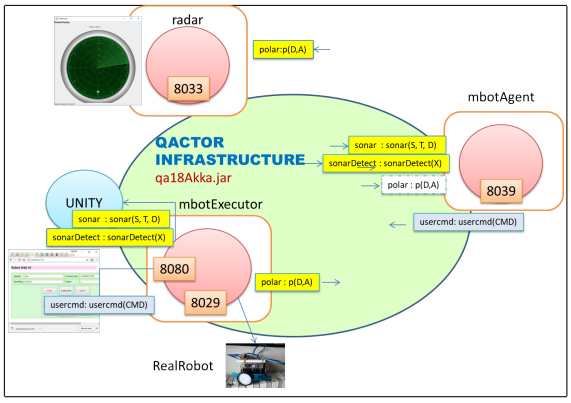
After the re-factoring introduced in this section, the logical architecture of Section 4 is still valid. However the steps that lead to a concrete working systems can be summarized as follows:

1. First of all we must activate the `radar` component.
2. Afterwards, we activate the `mbotExecutor` system (`it.unibo.ctxMbotExecutor.MainCtxMbotExecutor`) that will form a 'system' with the radar.
3. Finaly, we activate the agent (the main is `it.unibo.ctxMbotAgent.MainCtxMbotAgent`) that will form a 'subsystem' with the `mbotExecutor`.

The system we have built is shown in the following picture:



Since we work upon the *QActor* infrastructure, an event raised in one of the component of a subsystem can be perceived by the other components of the same subsystem. In fact, the agent perceives the events emitted by the rover and controls the robot by raising `usercmd` events. Moreover, if we uncomment the sentence `(2)` in the agent of Subsection 9.2, we will raise a `polar` event from the agent and see the effect on the radar.



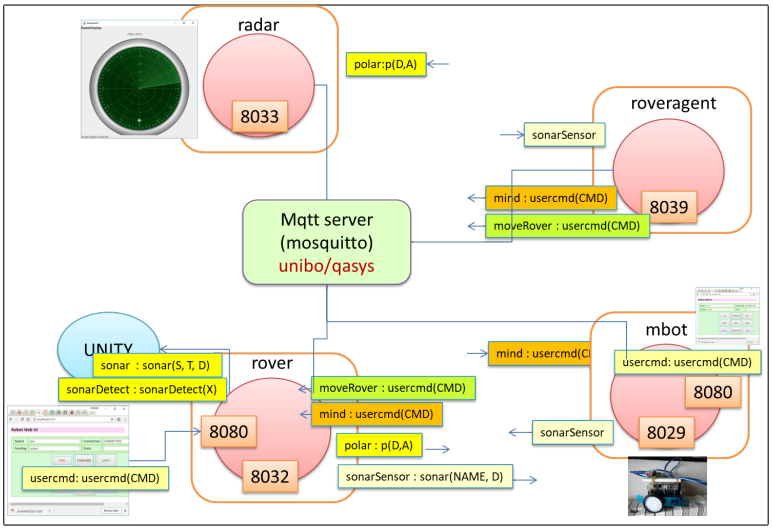
The `roveragent` of Subsection 9.2 handles the events `sonar` and `sonarDetect` that are always emitted in different situations, i.e. they are never emitted 'at the same time'. The `polar` event however, is always emitted together with a `sonar` or a `sonarDetect` event. The `roveragent` is in principle able to handle also such a `polar` event, but only if it is not already engaged in handling a `sonar` or a `sonarDetect` event. In fact, the behavior of a *QActor* is *event-based* and `not` *event-driven* (see `IntroductionQa2017.pdf` - section 4).

10 Another refactoring: Mqtt-based components

In Section 7 we supposed to exploit the Mqtt protocol in order to make a robot a publisher of events and a radar a subscriber. In this section we will assume an stronger position, by posing the following problem:

> Is it possible to build a concrete distributed system working without the *QActor* infrastructural support? For example, is it possible to design and build a system in which the components (microservices) interact by using the Mqtt protocol?

The goal of this section is to re-factor our robot-radar system as shown in the following picture:



With respect to the architecture of Section 9, there are now several important differences:

1. Each component is completely unaware of the others.
2. Each component has only one (`TCP`) connection to the Mqtt server and no connections to the nodes of the other components.
3. The Mqtt server is a centralization component that plays the role of 'system integrator' since it provides the support that makes independent components able to exchange information and to form a system[11].

10.1 The *QActor* extensions for Mqtt-interaction

To facilitate the transition form a (logical) prototype built upon the *QActor* infrastructure to a concrete system working with Mqtt, the version `1.5.13` of the *QActor* plugins provides the following features:

1. A *QActor* associated with the `-pubsub` flag includes in its initial plan actions to register itself (both as publisher and as subscriber) to topic named `unibo/qasys` of the Mqtt server specified in the `pubSubServer` declaration.
2. A *QActor* associated with the `-pubsub` flag re-implements event emission (`emit` operation) by publishing low-level messages[12] on the `unibo/qasys` topic. The low-level messages are Strings of the form introduced in Subsection 1.1:

```
1 msg(MSGID,  dispatch, SENDER, RECEIVERID, MSGPAYLOAD, MSGNUM)
2 msg(EVENTID, event,  EMITTER, none,      EVENTPAYLOAD, MSGNUM)
```

[11] We could view the Mqtt server as a pivot, like the radar of Subsection ??. However, the Mqtt server works as message broker rather than as a (replicated) repository of knowledge about the system.

[12] Events whose names starts with `local` are `not` sent via Mqtt.

3. When a event is *published* on the `unibo/qasys` topic, a proper `callback` will be activated by the Mqtt support in all the subscribers. Such a callback (method `messageArrived` of class *it.unibo.qactors.mqtt.MqttUtils*) is defined as an extension of the *QActor* support that converts the received String into a *QActor* message or a *QActor* event.

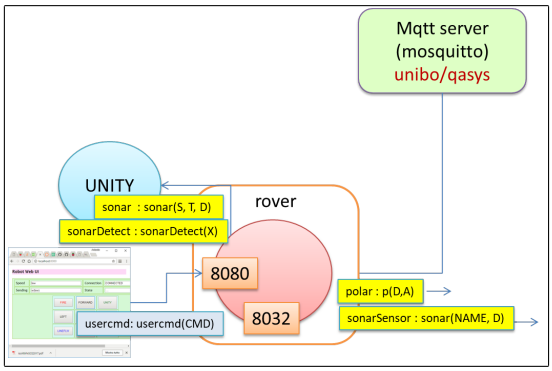
> IMPORTANT NOTE: all the actors with the `-pubsub` flag that make reference to the same Mqtt server must have different names.

We note also that we can publish messages, but only by recurring to lower-level operations, since the `forward` high-level operation requires explicit knowledge of the destination at model level. For example (see also Subsection 10.3.5):

```
publishMsg "unibo/qasys" for "DESTNAME" -m MSGID : MSGPAYLOAD;
javaOp "sendMsg(\"MSGID\", \"DESTNAME\", \"dispatch\", \"MSGPAYLOAD\")";
```

10.2 The mqtt-based robot executor

The robot executor is now defined as a virtual robot connected to the Mqtt server:




```
System roverExecutor
Event usercmd     : usercmd(CMD)                //from web gui
Event alarm       : alarm( X )                  //from web gui
Event alarmev     : alarm( X )                  //same payload as alarm
Event sonar       : sonar(SONAR, TARGET, DISTANCE) //From (virtual) sonar
Event sonarSensor : sonar(NAME, DISTANCE)       ////NAME= sonar1 | sonar2 | roversonar
Event sonarDetect : sonarDetect(X)              //From (virtual robot) sonar
Event   polar     : p( Distance, Angle )
Event mindcmd     : usercmd(CMD)                //to command a real robot

Dispatch moveRover : usercmd(CMD)

//pubSubServer "tcp://192.168.43.229:1883"
pubSubServer "tcp://localhost:1883"        //for testing

Context ctxRoverExecutor ip [ host="localhost" port=8032 ] -g white -httpserver
EventHandler evh for usercmd, mindcmd -print { //event-driven ; no Mqtt support yet
   forwardEvent rover -m moveRover  //from event to message
```

Listing 1.30. `roverExecutor.qa`: using a Mqtt server

In Subsection 9.1, we introduced an actor that transforms a user command event into a message sent to the rover robot. Now we take the following position:

10.2.1 Mapping events into massages .

In the *QActor* framework, an event is lost if no actor is waiting for it. Since `usercmd` events are our main way to manage a robot, we want to avoid to miss them. To achieve this goal, we introduce an `event-driven` code section that transforms `usercmd` events into dispatches (`moveRover`) sent to the `rover` actor.

The *QActor* feature that helps us in mapping events into message is the `EventHandler` introduced in `IntroductionQa2017.pdf` - section `4.2`. In our case, the event-driven handler `evh` (that works at context level) executes a `forwardEvent` operation to generate a `moveRover` dispatch with the same payload of the `usercmd` event[13].

10.2.2 The robot command executor .

Differently from Subsection 9.1.4, the rover now works only with reference to the virtual robot: its goal is just to move the *avatar* according to commands received through the `moveRover` dispatch:

```
1
2  QActor mindtobody context ctxRoverExecutor -pubsub {
3      Plan init normal [ println("mindtobody STARTS ") ]
4  // switchTo doWork
5  //
6  // Plan doWork[ println("mindtobody WAITS ... ") ]
7  // transition stopAfter 3600000  //1h
8  //         whenEvent usercmd -> emitMindEvent
9  //     finally repeatPlan
10 //
11 //     Plan emitMindEvent resumeLastPlan[
12 //         printCurrentEvent;
13 //         onEvent usercmd : usercmd(CMD) -> emit mindcmd : usercmd(CMD)
14 //     ]
```

Listing 1.31. `roverExecutor.qa`: Mqtt version

The rover actor waits not only for a `moveRover` dispatch; it also waits for an `alarm` event in order to emit anther event `alarmev`. This new event is transmitted via Mqtt[14] and constitutes a command for the real robot (see Section 11).

The local interpreter now sends commands to the virtual robot:

```
1
2  /*
3   * ------------------------------------------------------
4   * The rover is a an interpreter of moveRover dispatch
5   * ------------------------------------------------------
6   */
7  QActor rover context ctxRoverExecutor -pubsub {
8      Rules{ //unityOn. //set by connectToUnity
9             unityConfig( "unityStart.bat").
10     }
11     Plan init normal [ println("rover START") ]
12     switchTo waitForCmd
13
14     Plan waitForCmd[ ]
15     transition stopAfter 3600000  //1h
16         whenEvent alarm : alarm(X) do {
17             emit alarmev : alarm(X) //emit via Mqtt a new event (to avoid recursion);
18         },
19         whenMsg moveRover -> execMove
20     finally repeatPlan
21
22     Plan execMove resumeLastPlan[
23         printCurrentMessage;
24         onMsg moveRover : usercmd( robotgui(h(X)) ) -> stop 40 time(0) ;
```

[13] Thus, the payload of the event and of the corresponding dispatch must have the same structure.

[14] The Mqtt emission of non-local events done by an `EventHandler` is not supported.

```
        onMsg moveRover : usercmd( robotgui(w(X)) ) -> onward 40 time(0) ; //0 means asynch
        onMsg moveRover : usercmd( robotgui(s(X)) ) -> backwards 40 time(0) ;
        onMsg moveRover : usercmd( robotgui(a(X)) ) -> left 40 time(750) ;
        onMsg moveRover : usercmd( robotgui(d(X)) ) -> right 40 time(750) ;
        onMsg moveRover : usercmd( robotgui(x(X)) ) -> { actorOp terminateSystem() }; //TODO
        onMsg moveRover : usercmd( robotgui(unityAddr(X)) ) -> {
            [ not !? unityOn ] {
                [ !? unityConfig( BATCH) ] //Unity is on the current host
                      javaRun it.unibo.utils.external.connectRoverToUnity(BATCH);
                addRule unityOn ;
                //setAvatarInitialPosition
                backwards 70 time ( 1000 ) ;
                right    70 time ( 1000 )
            } else println("UNITY ALREADY ACTIVATED")
        }
    ]
```

Listing 1.32. `roverExecutor.qa`: interpreter

10.2.3 The sonardetector .

As done in Subsection 9.1.8, the virtual robot component includes an actor that converts `sonar` data into `polar` events. In this case however, this `sonardetector` actor propagates via Mqtt a 'standard-sonar' event of the form[15]:

```
sonarSensor : sonar( NAME, DISTANCE )       //NAME= sonar1 | sonar2 | roversonar
```

```
 */
QActor sonardetector context ctxRoverExecutor -pubsub {
    Plan init normal [ println("sonardetector STARTS ") ]
    switchTo waitForEvents

    Plan waitForEvents[ ]
    transition stopAfter 3600000  //1h
        //these events can be captured because sonardetector is in the context ctxRoverExecutor
        whenEvent sonar     ->  sendToRadar,
        whenEvent sonarDetect -> showObstcle
    finally repeatPlan

    Plan sendToRadar resumeLastPlan [
        onEvent sonar : sonar(NAME, TARGET, DISTANCE ) -> emit sonarSensor : sonar(NAME, DISTANCE) ;
        onEvent sonar : sonar(sonar1, TARGET, DISTANCE ) -> emit polar : p(DISTANCE,30) ;
        onEvent sonar : sonar(sonar2, TARGET, DISTANCE ) -> emit polar : p(DISTANCE,120)
    ]
    Plan showObstcle resumeLastPlan[
        println( "found obstacle" );
        onEvent sonarDetect : sonarDetect( TARGET ) -> emit sonarSensor : sonar(roversonar, 30) ;
        emit polar : p(30,90)
    ]
}
```

Listing 1.33. `roverExecutor.qa`: the sonardetector

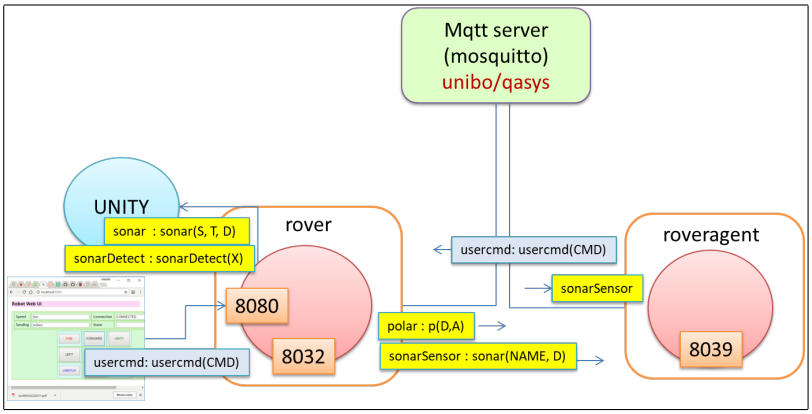
10.3 The mqtt-based robot agent

Let us define now a new (Mqtt-based) version of the component `roveragent` (the previous one was in Subsection 9.2) that looks at the `sonarSensor` events (emitted by the `roverExecutor` of Subsection 10.2.3) and performs some work at 'application level'. For example:

- stops the robot when the rover is detected by a fixed virtual sonar (`sonar1` or `sonar2`);
- stops the robot when an `alarm:alarm(X)` event is emitted.
- implements some *obstacle-avoidance policy*;
- takes a log of all the events perceived (different from `usercmd`);

[15] Note that the events `sonar` and `sonarDetect` cannot be perceived by the other components.

– ...



The new `roveragent` component can be viewed as the 'mind' that moves the robot by emitting `usercmd` events, according to application requirements.

```
System mbotAgent
Event mindcmd     : usercmd(CMD)
Event alarmev     : alarm( X )
Event sonarSensor : sonar( NAME, DISTANCE ) //From rover or real robot

Dispatch alarmmsg : alarm( X )  //same payload as alarmev
Dispatch moveRover : usercmd(CMD)

pubSubServer "tcp://localhost:1883"
```

Listing 1.34. `mbotAgent.qa`: Mqtt version

The `roveragent` must now face a new problem:

> The `alarmev` event can be raised while the actor is handling the `sonarSensor` event, i.e. while it is performing its application logic.

10.3.1 Proactive-reactive systems .

The problem above is typical of systems that must exhibit both a `proactive` and a `reactive` behaviour and implies that more precise requirements must be specified. For example, in order of importance:

1. we must avoid to miss `alarm` events;
2. we should handle an `alarmev` as soon as possible.

10.3.2 Non real-time systems .

The second requirement means that we are not (fortunately) involved in the design of a real-time systems. However the meaning of 'as soon as possible' should be better clarified. Since, the maximum delay in handling an `alarm` is related to the execution time of the most lasting (synchronous) application operation, we will assume here that such a delay can be always tolerated[16].

[16] In a further version of our application agent, we will exploit the idea of `reactive actions` introduced in the model of Subsection 3.3.4 (see also `IntroductionQa2017.pdf` - section 14).

10.3.3 Avoiding the loss of alarm events .

Avoid the loss of alarm events seems more difficult to achieve, since the computational model of a *QActor* is event-based and not event-driven (see IntroductionQa2017.pdf - section 4). In other words, the usual finite-state machine behavior:

```
    Plan init normal [ ]
    switchTo lookAtSonars
    Plan lookAtSonars[ ]
    transition stopAfter 6000000
        whenEvent alarm     -> handleAlarm,
        whenEvent sonarSensor -> handleSonars
    finally repeatPlan
```

does not work, since an alarm is missed if it occurs while the actor is working in the handleSonars state.

To overcome this problem we can introduce an event-driven part (see IntroductionQa2017.pdf - section 4.2) with the very limited goal of transforming events into dispatches as done in Subsection 10.2.1:

```

Context ctxMbotAgent ip [ host="localhost" port=8039 ] -g cyan //192.168.43.229
EventHandler evh for alarmev -print { //event-driven
    forwardEvent roveragent -m alarmmsg
```

Listing 1.35. mbotAgent.qa: EventHandler

10.3.4 Event processing .

The behaviour of the roveragent consists in handling sonarSensor events and alarmmsg messages:

```

QActor roveragent context ctxMbotAgent -pubsub{
    Plan init normal [
        println("roveragent STARTS")
    ]
    switchTo doWork
    Plan doWork[
        println("roveragent WAITS")
    ]
    transition  stopAfter 6000000
        whenMsg alarmmsg      -> alarmHandlePolicy ,
        whenEvent sonarSensor -> handleSonarEvents
```

Listing 1.36. mbotAgent.qa: handling sonarSensor

The handling of sonarSensor events is implemented by the handleSonarEvents state. For each kind of events a sequence[17] of (terminating) actions (see IntroductionQa2017.pdf - section 2.4) is executed:

```
    Plan handleSonarEvents resumeLastPlan[
        printCurrentEvent;
        onEvent sonarSensor : sonar(realsonar, DISTANCE) -> {
            //real obstacle
            println( roversonar );
            emit mindcmd : usercmd( robotgui(s(low)) ) ; //retrogress
            delay 700 ;  //continue to move to stop sonar events
            emit mindcmd : usercmd( robotgui(h(low)) ) ;
            delay 1500  ; //in this time, an alarm could be emitted
            emit mindcmd : usercmd( robotgui(s(low)) ) //retrogress and stop if alarm
        };
        onEvent sonarSensor : sonar(roversonar, DISTANCE) -> {
            //virtual obstacle
            println( roversonar );
            emit mindcmd : usercmd( robotgui(s(low)) ) ; //retrogress
            delay 700 ;  //continue to move to stop sonar events
            emit mindcmd : usercmd( robotgui(h(low)) ) ;
            println("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% LONG ACTION");
```

[17] This feature has been introduce in version 1.5.13.

```
19          //in this time an alarm could be emitted
20      emit mindcmd : usercmd( robotgui(a(low)) ) ; //750 / 680 msec
21      emit mindcmd : usercmd( robotgui(a(low)) ) ; //750 / 680 msec
22          println("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% GOING ON");
23      emit mindcmd : usercmd( robotgui(w(low)) ) ; //ahead
24          println("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% LONG ACTION END")
25      };
26      onEvent sonarSensor : sonar(sonar1, DISTANCE) -> {
27          //fixed sonar (moving ahead or backwards)
28          println( sonar1 );  //no variable substitution, sorry
29          delay 500 ;   //continue to move to stop sonar events
30          emit mindcmd : usercmd( robotgui(h(low)) )
31      };
32      onEvent sonarSensor : sonar(sonar2, DISTANCE) -> {
33          //fixed sonar (moving ahead or backwards)
34          println( sonar2 );  //no variable substitution, sorry
35          delay 500 ;   //continue to move to stop sonar events
36          emit mindcmd : usercmd( robotgui(h(low)) )
37      }
38  ]
39  }//roveragent
```

Listing 1.37. `mbotAgent.qa: handleSonarEvents`

Note that the action sequence corresponding to the `sonarSensor:sonar(roversonar,D)` event includes a 'long-lasting' action, during which an `alarm` event could be raised.

10.3.5 Alarm handling .

The agent alarm-handling policy is implemented by the `alarmHandlePolicy` state:

```
1
2   Plan alarmHandlePolicy resumeLastPlan[
3       printCurrentMessage;
4       println("roveragent ALARM HANDLING POLICY (turn left)... ") ;
5       //NOTE : we interact in this case by sending a message (and by going to a LOWER-LEVEL)
6       publishMsg "unibo/qasys" for "rover" -m moveRover : usercmd( robotgui(a(low)) );
7       javaOp "sendMsg(\"moveRover\", \"rover\", \"dispatch\", \"usercmd( robotgui(d(low)) )\")";
8       println("roveragent ALARM HANDLING POLICY DONE ")
```

Listing 1.38. `mbotAgent.qa`: alarmHandlePolicy

10.4 Testing the application agent

The application agent can be tested by an actor that generates a sequence of events:

```
1  System testMbotAgent
2  Event usercmd    : usercmd(CMD)
3  Event alarmev    : alarm( X )
4  Event sonarSensor : sonar( NAME, DISTANCE ) //From rover or real robot
5
6  pubSubServer "tcp://192.168.43.229:1883"
7
8  Context ctxTestMbotAgent ip [ host="1ocalhost" port=8043 ]
9
10 QActor testmbotagent context ctxTestMbotAgent -pubsub{
11     Plan init normal [
12         println("testmbotagent STARTS")
13     ]
14     switchTo doWork
```

Listing 1.39. `testMbotAgent.qa`

First of all our testing actor generates `sonarSensor` events that activate actions in the `roveragent` and then an `alarmev` event:

```
1   Plan doWork[
2       demo assign(n,9);
3       emit sonarSensor : sonar( sonar1, 10 ) ;
4       delay 1000;
5       emit sonarSensor : sonar( roversonar, 20 ) ; //long action
6       emit alarmev    : alarm( fire)
7   ]
8   switchTo raiseSeries
```

Listing 1.40. `testMbotAgent.qa`

In this way we can test if the `alarmev` event is not lost when executing the long-lasting action.
Afterwards, the actor generate a series of `sonarSensor` events, in order to see that several of them are lost:

```
1   Plan raiseSeries[
2       [ !? inc(n,1,V)] emit sonarSensor : sonar( sonar2, V ) ; //> 500msec
3       [ !? getVal(n,V)] println( emitted( sonar( sonar2, V ) ));
4       delay 50
5   ]
6   finally repeatPlan 30
```

Listing 1.41. `testMbotAgent.qa`

The built-in actions used in the guards are introduced in `IntroductionQa2017.pdf` - section `2.8`.
The problem to deal with now is:

> How can we automate the testing activity?

11 Using a real robot

The 'middleware' working on the `RaspberryPi` is modelled as a *QActor* system that reacts to events coming from a user web interface (`usercmd` event) and from a robot 'mind' (`mindcmd` event) by mapping these events into a `moveMbot` dispatch:

```
System mbotExecutor
Event usercmd    : usercmd(CMD)             //from web gui
Event realSonar  : sonar( DISTANCE )        //From real sonar on real robot
Event   polar    : p( Distance, Angle )
Event sonarSensor : sonar( NAME, DISTANCE )
Event mindcmd    : usercmd(CMD)             //from the appl agent (the 'mind')

Dispatch moveMbot : usercmd( CMD )

pubSubServer "tcp://192.168.43.229:1883"

Context ctxMbotExecutor ip [ host="192.168.43.67" port=8029 ] -httpserver
EventHandler evh for usercmd , mindcmd -print { //event-driven
    forwardEvent mbot -m moveMbot  //from event to message
};
```

Listing 1.42. `mbotExecutor.qa`

The 'middleware' includes also an interpreter of `moveMbot` messages that calls the 'bridge' to Arduino introduced in Subsection 3.3.3:

```
QActor mbot context ctxMbotExecutor {
    Rules{
        onRaspberry.
        foundObstacle :- retract( realDistance(D) ), eval( lt, D, 20 ).
    }
    Plan init normal [
        [ !? onRaspberry] javaRun it.unibo.mbot.mbotConnArduino.initRasp() ;
        println("mbot START")
    ]
    switchTo waitForCmd

    Plan waitForCmd[ ]
    transition stopAfter 3600000  //1h
        whenMsg moveMbot -> execMove
    finally repeatPlan

    Plan execMove resumeLastPlan[
        printCurrentMessage;
        onMsg moveMbot : usercmd( robotgui(h(X)) ) -> javaRun it.unibo.mbot.mbotConnArduino.mbotStop();
        onMsg moveMbot : usercmd( robotgui(w(X)) ) -> javaRun it.unibo.mbot.mbotConnArduino.mbotForward();
        onMsg moveMbot : usercmd( robotgui(s(X)) ) -> javaRun it.unibo.mbot.mbotConnArduino.mbotBackward();
        onMsg moveMbot : usercmd( robotgui(a(X)) ) -> {
            javaRun it.unibo.mbot.mbotConnArduino.mbotLeft();
            delay 680; //TODO: use some configuration parameter
            javaRun it.unibo.mbot.mbotConnArduino.mbotStop()
        };
        onMsg moveMbot : usercmd( robotgui(d(X)) ) -> {
            javaRun it.unibo.mbot.mbotConnArduino.mbotRight();
            delay 680;//TODO: use some configuration parameter
            javaRun it.unibo.mbot.mbotConnArduino.mbotStop()
        }  ;
        onMsg moveMbot : usercmd( robotgui(f(X)) ) -> javaRun it.unibo.mbot.mbotConnArduino.mbotLinefollow()
    ]
}
```

Listing 1.43. `mbotExecutor.qa`: the command executor

Finally, the 'middleware' defines an actor that captures the data coming from the real sonar and emits the 'standard' `sonarSensor` events:

```
QActor realsonardetector context ctxMbotExecutor -pubsub{
    Plan init normal [
        println("sonardetector STARTS ")
    ]
```

```
5     switchTo waitForEvents
6
7     Plan waitForEvents[ ]
8     transition stopAfter 3600000  //1h
9         whenEvent realSonar -> handleRealSonar
10
11    Plan handleRealSonar [
12        printCurrentEvent;
13        onEvent realSonar  : sonar( DISTANCE ) ->
14            emit sonarSensor : sonar( realsonar, DISTANCE )  ; //convert in int???
15        onEvent realSonar : sonar( DISTANCE ) ->
16            emit polar   : p( DISTANCE,0 )
17    ]
18    switchTo waitForEvents
19 }
```

Listing 1.44. `mbotExecutor.qa`: handling real sonar

11.1 Running the system

Now our system is composed of a set of independent components, each designed as a micro-service. To set-up the system, we must activate these components. No prefixed order is required, but the premise is:

1. Activate the Wi-Fi hotspot and connect the PC.
2. Activate the Mqtt server (on a PC in the virtual network).

Afterwards we can follow - from a logical point of view - the following activation sequence:

1. Optional: activate the mbotExecutor on the `RaspberryPi`.
2. Activate the radar (in the Mqtt version).
3. Activate the rover executor of Subsection 10.2.
4. Select the Unity button and set-up the virtual environment.
5. Activate the application agent of Subsection 10.3.
6. Use the web GUI of the rover executor.

12 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 or Subsection 10.2. 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:

- activate the robot system (see for example Subsection 11.1).
- activate the Node.js server working on port 8098.
- perform some PUT/GET to port 8098.

12.1 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.

12.1.1 Connect with the *QActor* application .

Let us start the code of the server by setting up the connection to the *QActor* application (that must be running):

```
1  var net      = require('net');
2  var http     = require("http");
3  var utils    = require("./utils"); //to handle uncaughtException
4
5  var host        = "localhost";
6  var port        = 8098;
7  var qaport      = 8077;
8  var socketToQaCtx = null;
9  var dataStore = []; //Array of Buffers
10
11 //Connect to the QActor application
12 function connectToQaNode(){
13     try{
14         socketToQaCtx = net.connect({ port: qaport, host: host });
```

```
        socketToQaCtx.setEncoding('utf8');
        console.log('connected to qa node:' + host + ":" + port);
    }catch(e){
        console.log(" ------- connectToQaNode ERROR " + e + " socketToQaCtx=" + socketToQaCtx);
    }
}
connectToQaNode();
```

Listing 1.45. `RobotFrontEndServer.js`: connect to *QActor* application

12.1.2 Handle the answer from the *QActor* node .

Now we write the code that handles the answers sent by the *QActor* application.

```
var qaanswer = "";
    console.log("NodeServer SETUP socketToQaCtx ---- ");
    socketToQaCtx.on('data',function(data) {
        qaanswer = qaanswer + data;
        if( qaanswer === "" ) return;
        if( data.includes("\n") ){
            console.log( "RobotFrontEndServer received from qa: "+qaanswer );
            qaanswer = "";
        }
    });
    socketToQaCtx.on('close',function() {
        console.log('connection is closed');
    });
    socketToQaCtx.on('end',function() {
        console.log('connection is ended');
    });
}
```

Listing 1.46. `RobotFrontEndServer.js`: handle a *QActor* answer

In this version, we simply show the answer in the console. The reader should consider the possibility that this answer is also the answer sent by the server to the `HTTP` client.

12.1.3 Create the server .

Now it is the time to write the part that waits for `HTTP` requests and sends some answer:

```
    //The request object is an instance of IncomingMessage (a ReadableStream; it's also an EventEmitter)
    var headers  = request.headers;
    var method   = request.method;
    var url      = request.url;
    //console.log('method=' + method );
    if( method == 'GET'){
        var outS = "";
        if( dataStore.length == 0 ) outS ="no data";
        dataStore.forEach( function(v,i){
            outS = outS + i + ")" + v + "\n"
        });
        response.write( outS )
        response.end();       //qaanswer to the caller
        emitQaEvent( "get" ); //emit event serverRequest
        //WARNING: The response should be built by the qa
    }//if GET
    if( method == 'POST' || method == 'PUT'){
        var item = '';
        request.setEncoding("utf8"); //a chunk is a utf8 string instead of a Buffer
        request.on('error', function(err) {
            console.error(err);
          });
        request.on('data', function(chunk) { //a chunck is a byte array
            item = item + chunk;
            console.log('method=' + method + " data=" + item);
         });
        request.on('end', function() {
                dataStore.push(item);
```

```
29              emitQaEvent( item );
30              buildHtmlResponse( url, method, dataStore, response );
31              //WARNING: The response should be given by the qa
32          });
33      }//if POST PUT
34  }).listen( port, function(){ console.log('bound to port ' + port);} );
35  console.log('Server running on ' + port);
```

Listing 1.47. `RobotFrontEndServer.js`: the server

The answer to a `GET` is a list of data currently stored by the server. The answer to a `PUT` (or `POST`) is an `HTML` page.

12.1.4 Emit an event for the QActor application .

For each request, the server builds the low-level representation of a |qa event and sends it to the *QActor* application:

```
1   function emitQaEvent( payload ) {
2       try{
3           var msg = "msg(serverrequest,event,frontend,none,serverrequest( data(frontend,'" + payload +"') )," + msgNum++
                +")";
4           if(socketToQaCtx !== null ){
5               console.log('emitQaEvent mmsg=' + msg );
6               socketToQaCtx.write(msg+"\n");
7               //TDOO: wait an qaanswer from the qa before responding to the HTTP user
8           }
9       }catch(e){
10          console.log("WARNING: " + e );
11      }
12  }
```

Listing 1.48. `RobotFrontEndServer.js`: emit a *QActor* event

12.1.5 Build an (HTML) response .

The (HTML) response shows to the user the list of data currently stored by the server.

```
1           response.on('error', function(err) { console.error(err); });
2           response.statusCode = 200;
3           response.setHeader('Content-Type', 'application/json');
4           //response.writeHead(200, {'Content-Type': 'application/json'}) //compact form
5           var responseBody = {
6             //headers: headers, //comment, so to reduce output
7             method: method,
8             url:   url,
9             dataStore:  dataStore
10          };
11          response.write( JSON.stringify(responseBody) );
12          response.end();
13          //response.end(JSON.stringify(responseBody)) //compact form
14  }
```

Listing 1.49. `RobotFrontEndServer.js`: build an (`HTML`) response