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

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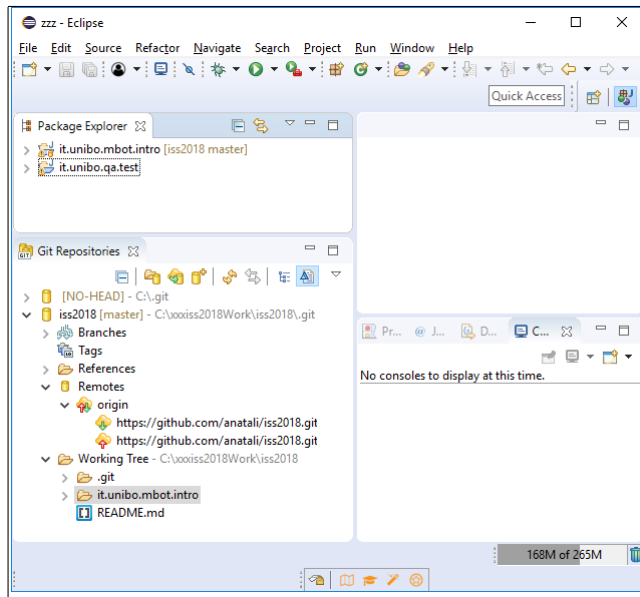
   To update the repository, the command is `git pull`.

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:

---
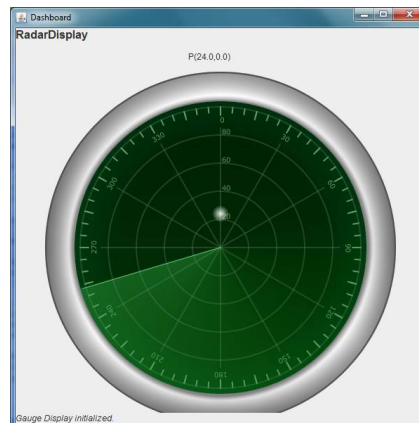
[1] To avoid any conflict in project updating.

2

1.1 Interacting with the radar

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

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

where

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

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

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

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

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

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

Listing 1.2. `TcpClientToRadar.js: utility`

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

Finally, we send some data to the radar:

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

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

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

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

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

1.2   Using the radar: a model-based approach

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

But, fortunately, there exist a high level description of the radar system, expressed in the high-level, custom modelling language *QActor*. This description is a (executable) model defined as follows:
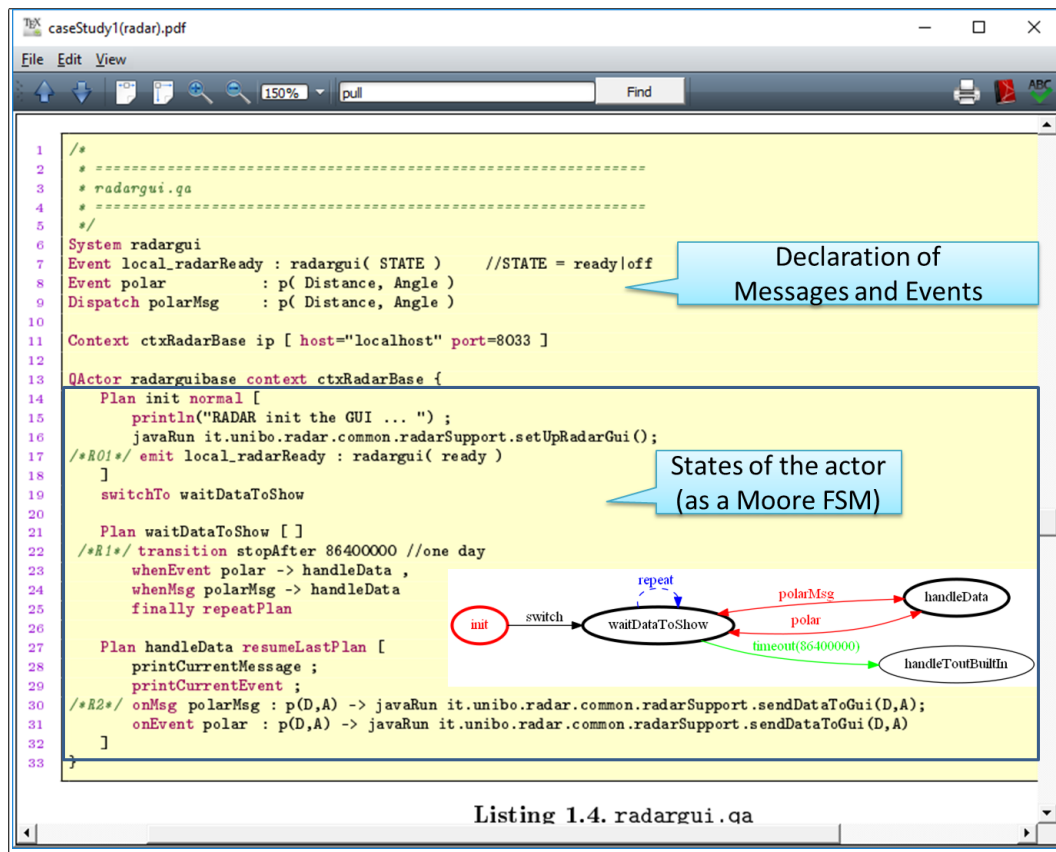
The model describes the structure, the interaction and the behaviour of the radar. More specifically, it shows that the radar is able to handle messages and events *explicitly declared* at the very beginning of the model[2]:

```
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'.

―――――――――

[2] The event `local_radarReady` is a local event used internally.

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]:

Listing 1.4. radarUsage.qa
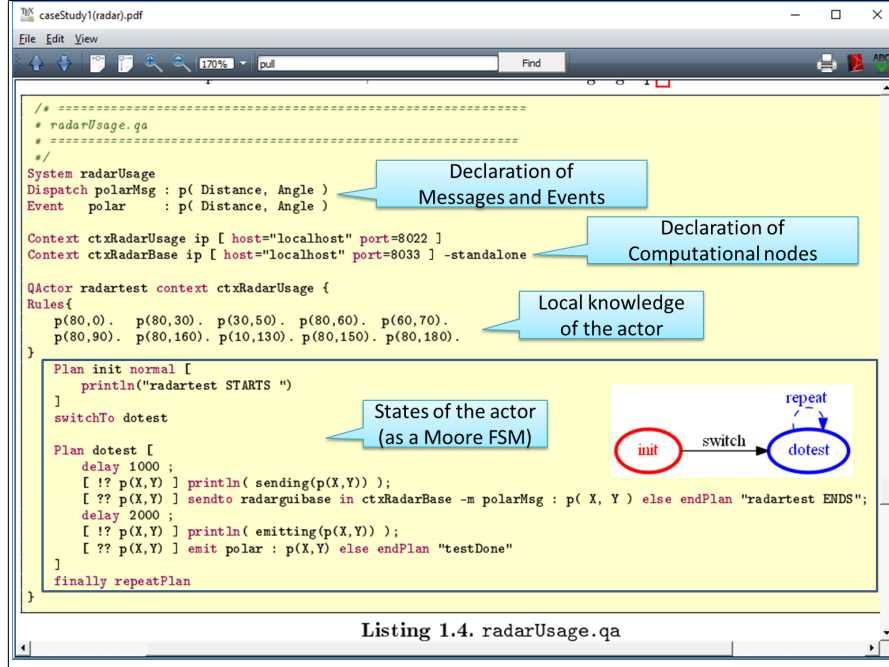
The model states that:

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 1.2.1.
4. The messages and events involved in our system are *the same* defined in the radar model:

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

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:

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

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

1.2.1 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:

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

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

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

---

[3] The code is in the file `it.unibo.mbot.intro/src/radarUsage.qa`.

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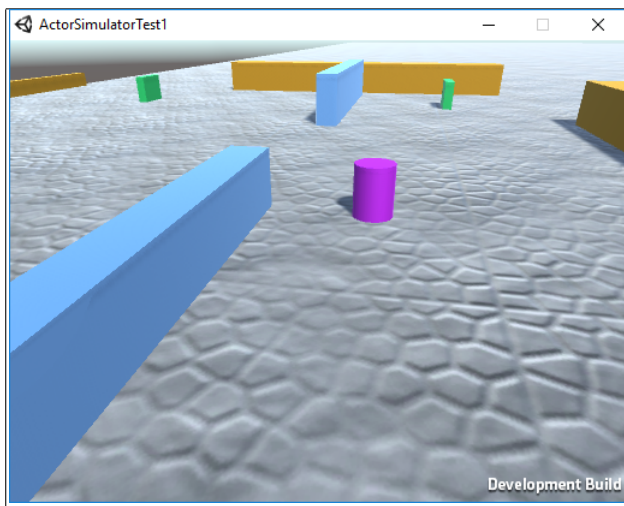
The original Unity environment has been modified to interact with *QActor* systems. Details about this point are given in `IntroductionQa2017.pdf` (section 12)[4].

---

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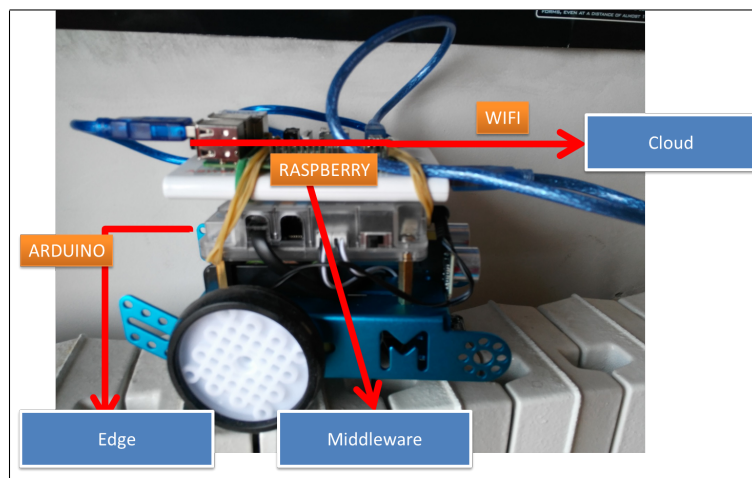
3.2.1 IOT reference architecture .
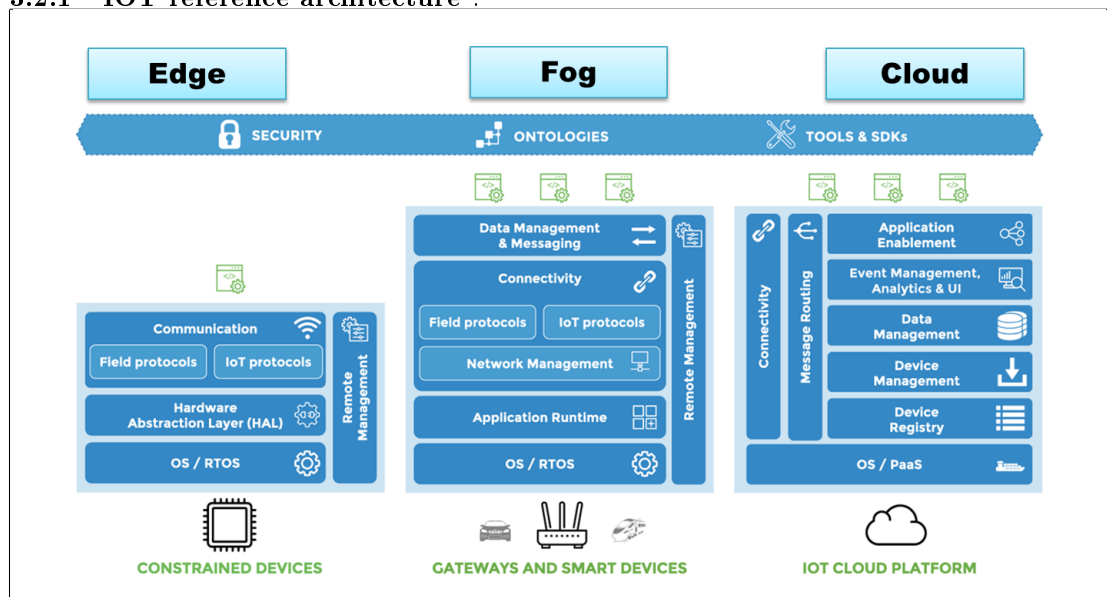
Arduino handles physical devices such as motors and sensors, while RaspberryPi provides support for interaction with a remote node. More precisely, as regards the interactions:

- 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:

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

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

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

3.3.1 The rover as a command interpreter .

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

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

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

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

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

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

Listing 1.6. `mbotControl.qa`: command intepreter

3.3.2   The connection to the Unity virtual environment .

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

```
1    Plan connectToUnity resumeLastPlan[
2        onEvent usercmd : usercmd( robotgui(unityAddr(ADDR)) ) -> println(ADDR) ;
3        connectUnity "localhost" ;
4        addRule unityOn ;
5        createUnityObject "rover" ofclass "Prefabs/CustomActor" ;
6        backwards 70 time ( 800 ) ;
7        right 70 time ( 1000 ) //position
```

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

3.3.3   Activating the motors for a prefixed time: reactive actions .

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

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

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

```
1        or whenEvent usercmd -> execMove
2    Plan turnLeft resumeLastPlan [
3        javaRun it.unibo.rover.mbotConnArduino.mbotLeft();
4        [ !? unityOn ] left 40 time(750) else delay 900;
5        javaRun it.unibo.rover.mbotConnArduino.mbotStop()
6    ]
7    Plan turnRight resumeLastPlan [
8        javaRun it.unibo.rover.mbotConnArduino.mbotRight();
9        [ !? unityOn ] right 40 time(750) else delay 900;
10        javaRun it.unibo.rover.mbotConnArduino.mbotStop()
11   ]
12   Plan stopTheRobot resumeLastPlan[
13        stop 40 time ( 10 );
14        javaRun it.unibo.rover.mbotConnArduino.mbotStop()
```

Listing 1.8. `mbotControl.qa: simple moves`

3.3.4   Activating the motors for a long time .

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:

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

Listing 1.9. `mbotControl.qa`

3.3.5   The emitter .

Another component that must run on the RaspberryPi is an actor able to mapssonar events into events that can be understood by the radar:

```
1
2  QActor sonardetector context ctxMbotControl{
3      Plan init normal [ ]
4      switchTo waitForEvents
5
6      Plan waitForEvents[ ]
7      transition stopAfter 600000
8          whenEvent sonar      -> sendToRadar,
9          whenEvent sonarDetect -> sendToRadar,
10         whenEvent realSonar -> sendToRadar
11     finally repeatPlan
12
13     Plan sendToRadar resumeLastPlan [
14         printCurrentEvent;
15         onEvent realSonar : sonar( DISTANCE )              -> emit polar : p(DISTANCE, 0) ;
16         onEvent sonar     : sonar(sonar1, TARGET, DISTANCE ) -> emit polar : p(DISTANCE,30) ;
17         onEvent sonar     : sonar(sonar2, TARGET, DISTANCE ) -> emit polar : p(DISTANCE,120) ;
18         onEvent sonarDetect : sonarDetect(TARGET)          -> switchTo showObstcle
19     ]
20
21     Plan showObstcle resumeLastPlan[
22         println( "found obstacle" );
23         emit polar : p(30,90)
24 //      sendto radarguibase in ctxRadarBase -m polarMsg : p( 30, 90 )
25     ]
26 }
```

Listing 1.10. `mbotControl.qa`

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.

## 3.4 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.
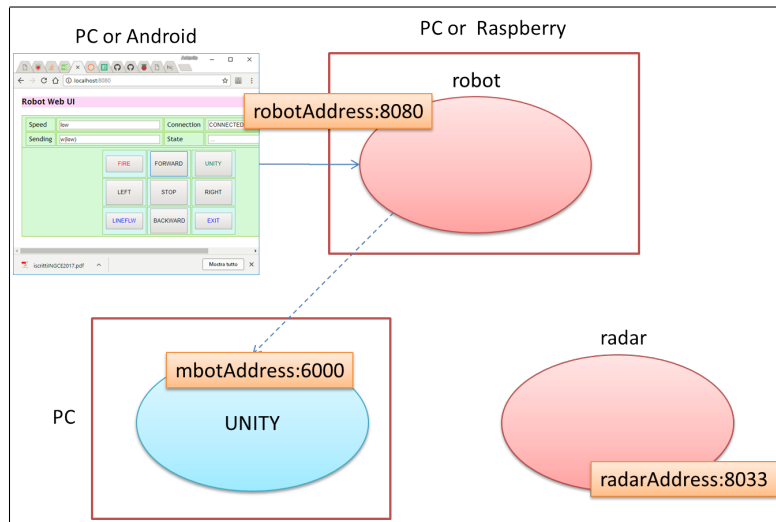
Now, we have to *understand* the problem posed by the requirements, the problematic issues involved by the problem and the constraints imposed by the problem or by the context. This is the main goal of the problem analysis phase, whose outcome is the logical architecture involved by the problem itself.

The following picture shows, in a informal way, a possible logical architecture of the system that arises from our current 'technology assumptions':
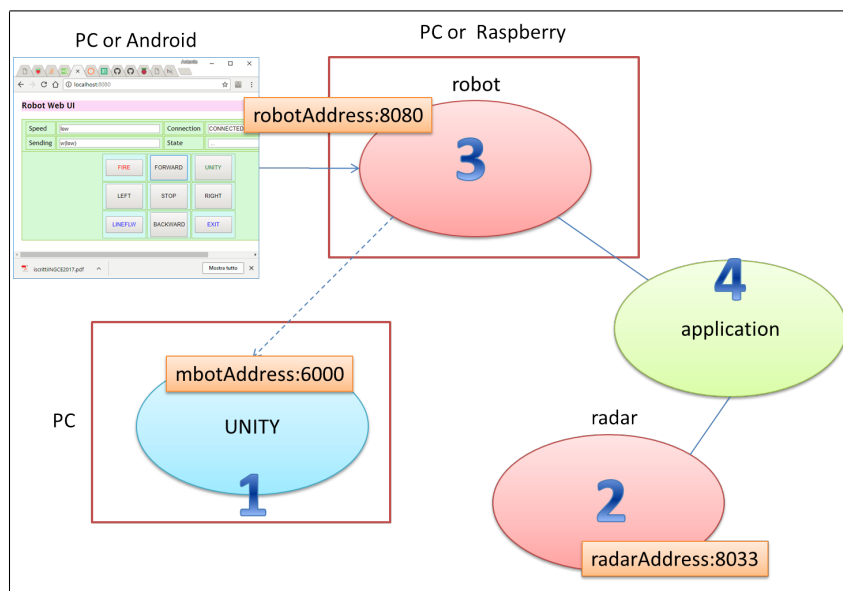
Our goal now are to describe such an architecture in a formal (i.e. machine-executable) way and to build a prototype to be shown to the customer as soon as possible.