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:

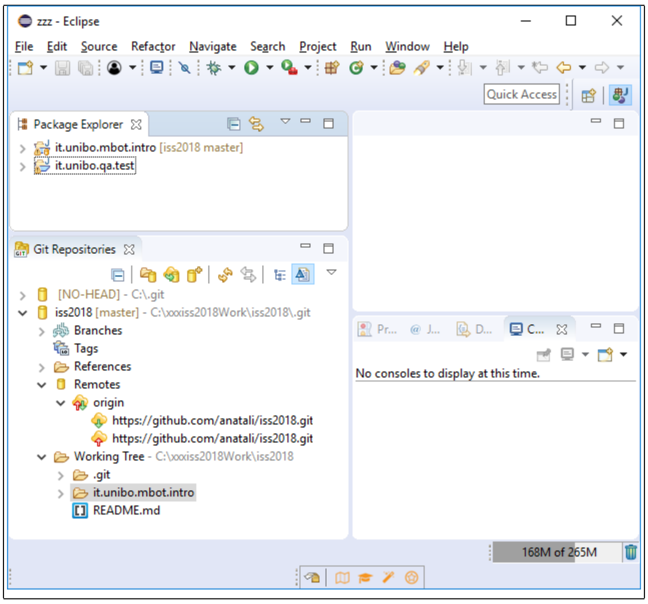
```
git clone https://github.com/anatali/iss2018Lab.git
```

   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)
```

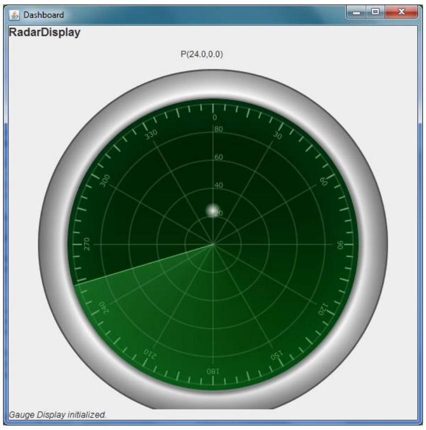
3. Import (by copying) into the current working space[1] the project it.unibo.mbot.intro. The result is:



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.

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:

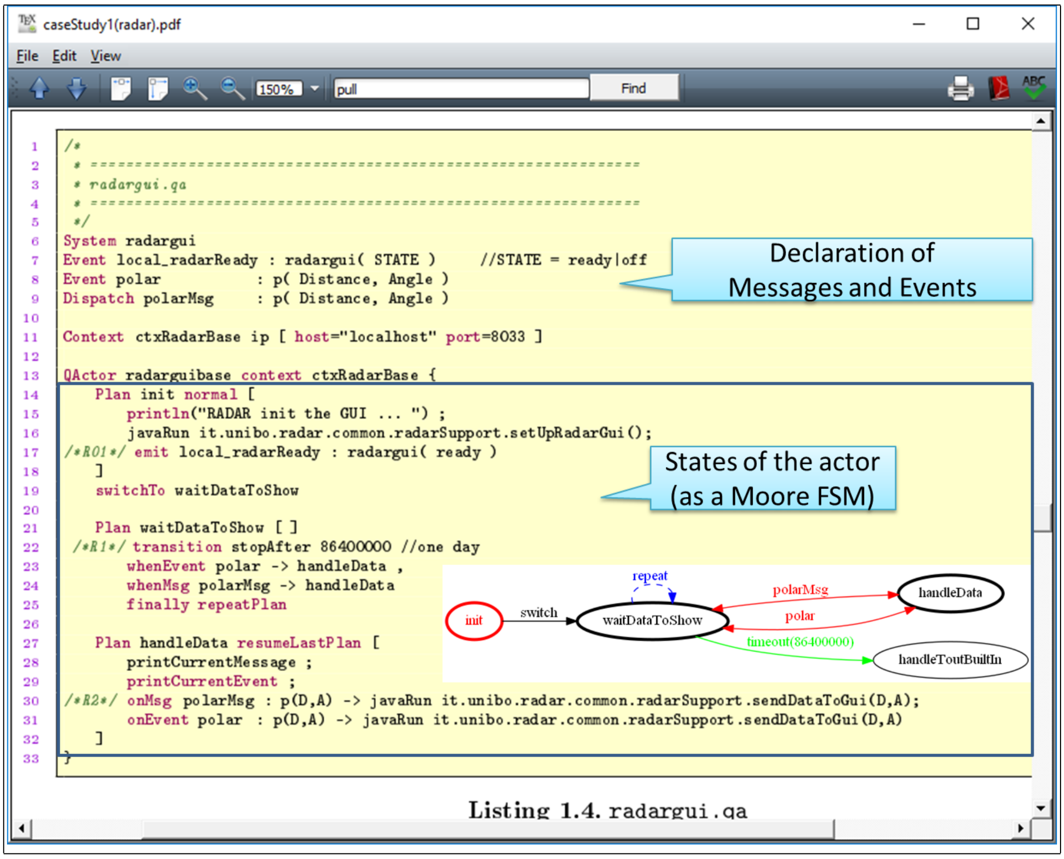
```
connecting to localhost:8033
SENDING msg(polarMsg,dispatch,jsSource,radarguibase, p(50,30),1)
SENDING msg(polarMsg,dispatch,jsSource,radarguibase, p(50,90),2)
SENDING msg(polarMsg,dispatch,jsSource,radarguibase, p(50,150),3)
connection is ended
connection is closed
```

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:



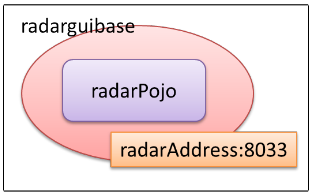
```
/*
 * ================================================================
 * radargui.qa
 * ================================================================
 */
System radargui
Event local_radarReady : radargui( STATE )     //STATE = ready|off
Event polar            : p( Distance, Angle )
Dispatch polarMsg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

```
public static void setUpRadarGui( QActor qa ) {
    try {
        radarControl = new RadarControl( qa.getOutputEnvView() );
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

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]:

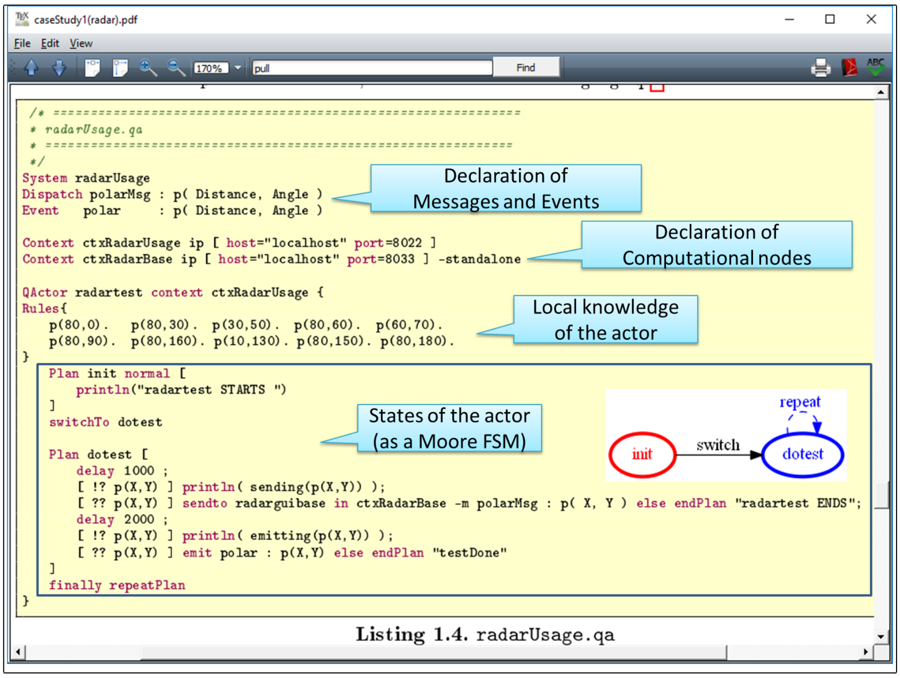
```
Dispatch polarMsg : p( Distance, Angle )
Event    polar    : p( Distance, Angle )
```

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.




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

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

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

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

Listing 1.4. radarUsage.qa

The model states that:

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:

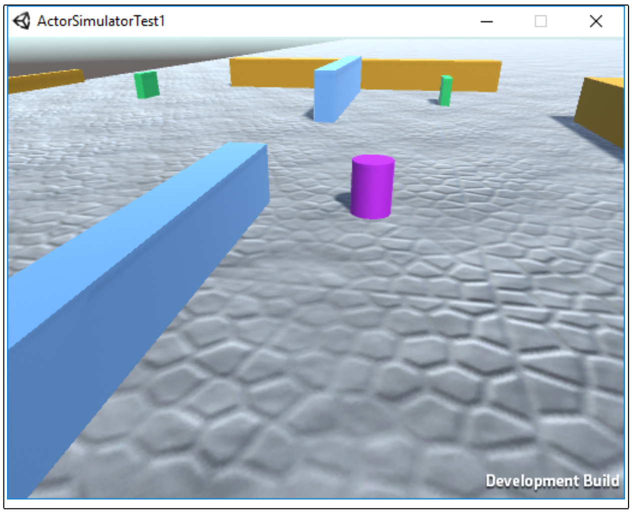
1. what is a `mbot`?
2. what is the virtual environment?
3. in which way we can obtain data from a virtual or from a real sonar?

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].

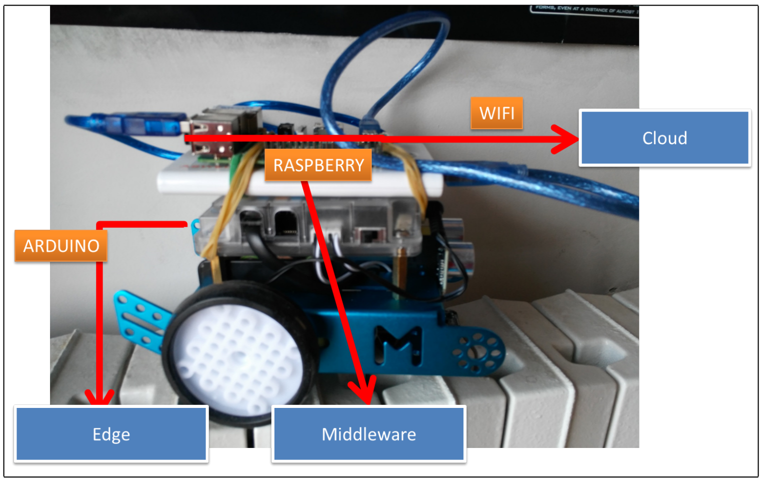
[4] The file is available in `it.unibo.issMaterial/issdocs/Material`.

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.

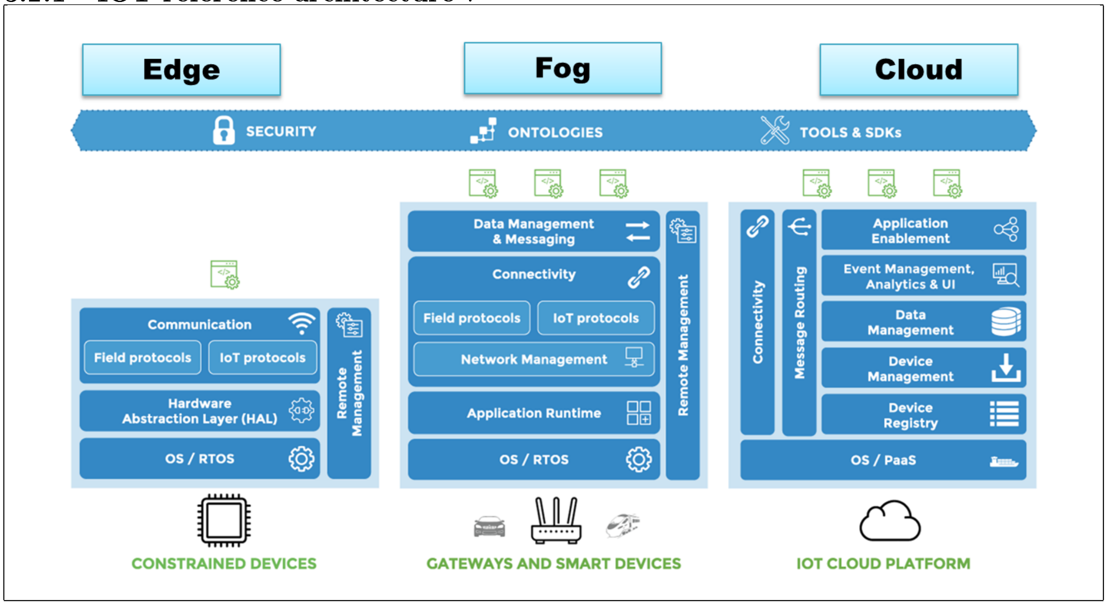
As regards the other questions, our goal is to build a model for the mbot and a model for the real sonar. The goal of these models is to clarify how we can exchange information with the corresponding entities of our application domain. The internal structure and the internal behaviour of the entities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

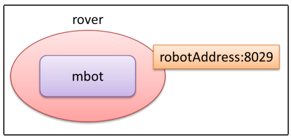
```
Plan connectToUnity resumeLastPlan[
    onEvent usercmd : usercmd(robotgui(unityAddr(ADDR))) -> println(ADDR);//ADDR=localhost in this version
    [ !? unityOn ] println( "UNITY already connected" )
]
switchTo [ not !? unityOn ] doconnectToUnity

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.

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

Listing 1.10. `mbotControl.qa`: the emitter

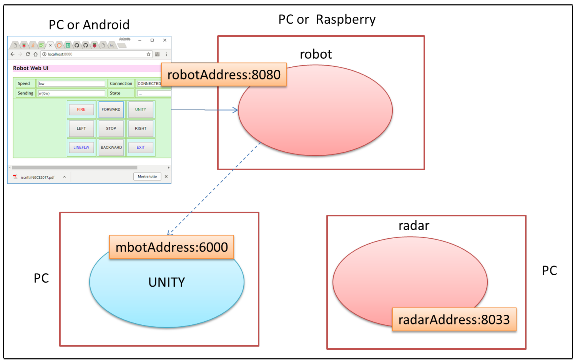
Note that the choice to emit events (and not to send messages) avoid the need to known the name of any destination actor.

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.

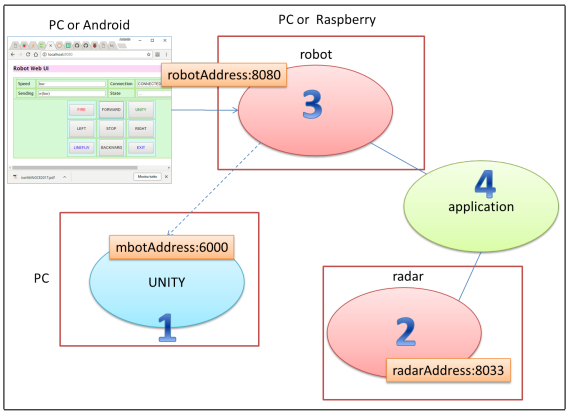
4 Problem analysis

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

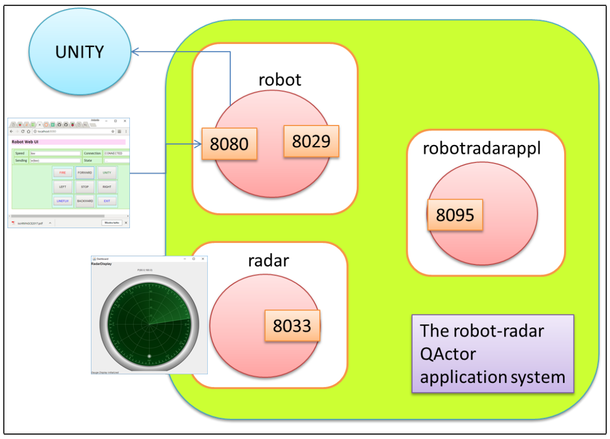
The robot is a source of information expressed as *QActor* events of the form `polar:p(DISTANCE,ANGLE)`). This kind of information is available by any other software component belonging to a *QActor* system. Since also the radar is implemented as a *QActor* application, we can state that the logical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

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

Context ctxRobotRadarAppl ip [ host="localhost" port=8095 ]
Context ctxMbotControl    ip [ host="localhost" port=8029 ]
Context ctxRadarBase      ip [ host="localhost" port=8033 ]

QActor robotradarappl context ctxRobotRadarAppl{
```

```
    Plan init normal [
        println("robotradarappl STARTS ") ;
        delay 600000
    ]
}
```

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.

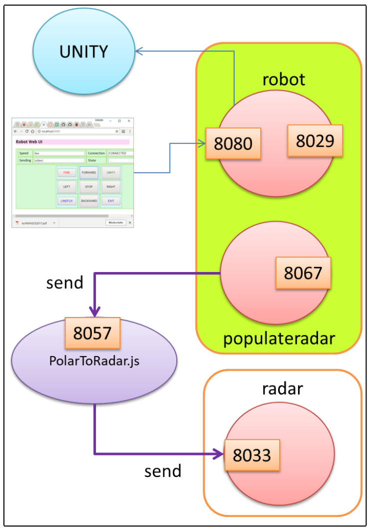
5.2 A *QActor* system as an adapter

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`:

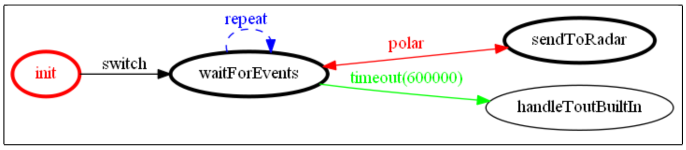
```
System robotToRadarAdapter
Event   polar : p( Distance, Angle )

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:

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

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`	performs a REST GET operation.
`sendRestPut urlString -m msgString`	performs a REST PUT operation.
— UNITY —	
`connectUnity addrString`	connects to the modified Unity virtual environment. At the moment, the addrString is ignored, since the "localhost" address is used.
`createUnityObject "rover" ofclass "Prefabs/CustomActor"`	creates 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`	connects as publisher to the given topic of the specifier MQTT server.
`connectAsSubscriber topicString`	connects 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.
— ANSWER TO EXTERNAL —	
`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:

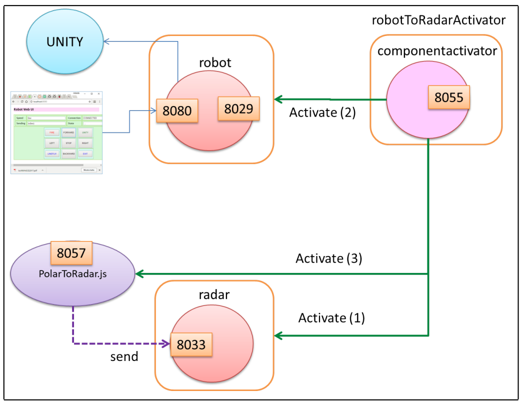
```
    public static void customExecute(QActor qa, String cmd) {
        try {
            Runtime.getRuntime().exec(cmd);
        } catch (IOException e) { e.printStackTrace(); }
    }
```

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 C:\Didattica2018Run\it.unibo.ctxRadarBase.MainCtxRadarBase-1.0
 START "" java -jar it.unibo.qactor.radar-1.0.jar > logRadar.txt
 cd ..
 echo please press CR
 PAUSE
 echo START THE ROBOT
 cd C:\Didattica2018Run\it.unibo.ctxMbotControl.MainCtxMbotControl-1.0
 START "" java -jar it.unibo.mbot.intro-1.0.jar > logRobot.txt
 cd ..
 PAUSE
 echo START THE ROBOT-RADAR APPLICATION
 cd C:\Didattica2018Run\it.unibo.ctxRobotRadarAppl.MainCtxRobotRadarAppl-1.0
 START "" java -jar it.unibo.mbot.intro-1.0.jar > logRadarRobotAppl.txt
 cd ..
```

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

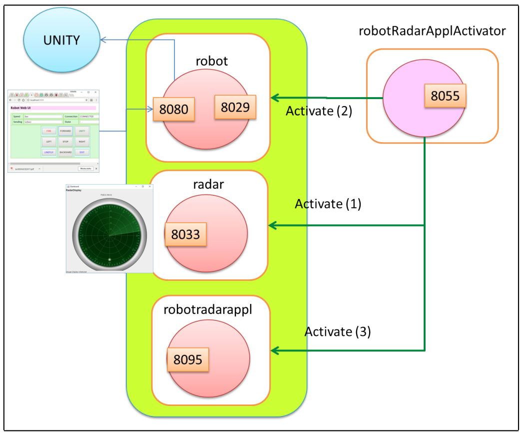
```
System robotRadarApplActivator

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") ;

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

        delay 10000;
        println("ACTIVATING THE SYSTEM") ;
        javaRun it.unibo.commToRadar.polarToRadar.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:

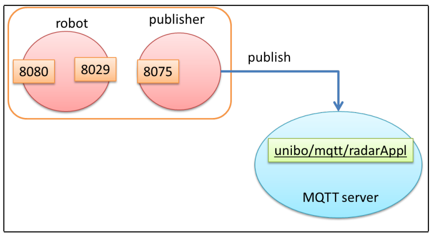
1. Introduce proper extensions like done in Subsection 5.2.
2. Modify the behaviour of the components by accessing their source code

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
}
```

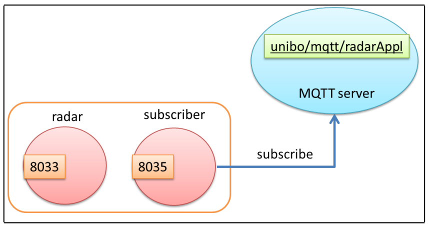
Listing 1.19. `robotToPublisher.qa`



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`.

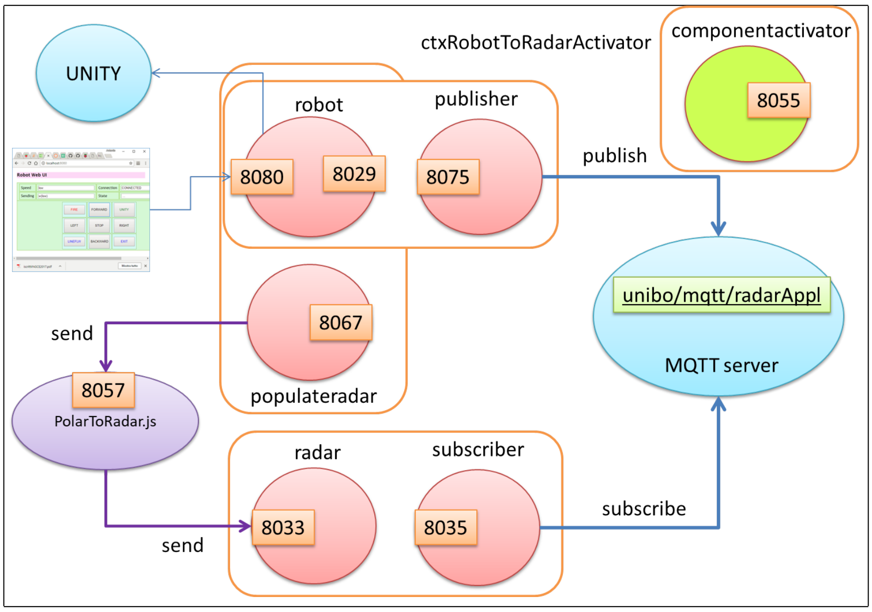
[7] Note that the `VPN` (if any) must be disabled.

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 7, in order to exchange sensor data by using the `MQTT` protocol.
   - Execute the `ctxRobotToRadarAdapter` of Subsection 5.2 in order to send sensor data by using the Node server.

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



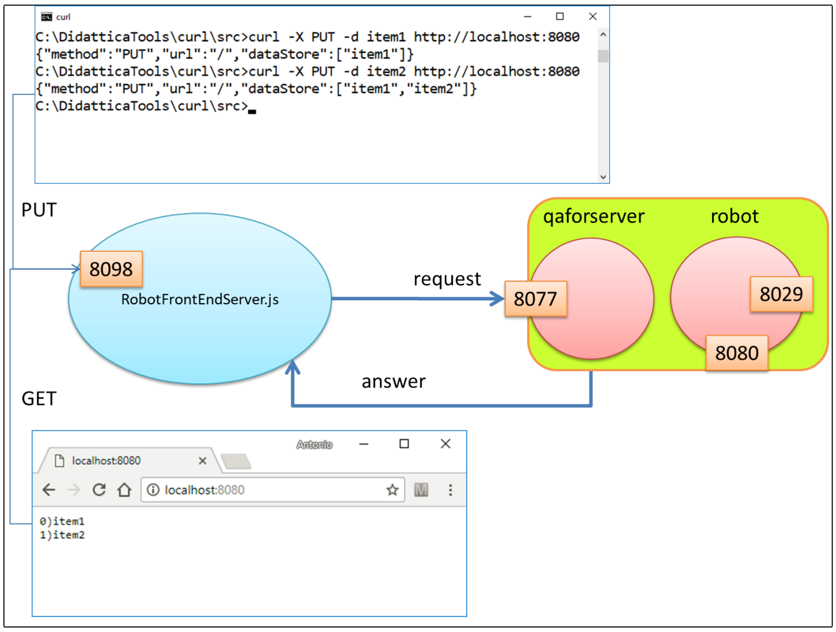
Thus, our final, concrete project architecture will be different from the logical architecture of our first prototype introduced in Subsection 4.1. However all the constraints imposed by the logical architecture are preserved. Moreover, several components introduced in the logical architecture are reused,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

9 A front-end written 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 the software designers could decide to build a more advanced server, supporting features like user authentication, security, logging, etc. To build such a new server, the `Node.js` platform and related frameworks, such as `Express`, `MongoDB`, `Angular` could be selected as the reference implementation technology. 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, we have to extend one of our working systems (see Subsection 4.1, Subsection 5.2 and Section 8) with a new feature: the possibility to receive requests from our new front-end server and to sent answers to it.



9.1 The front-end server

Our front-end server answers to `PUT/GET` requests. For each operation, the server logically emits a *QActor* event of the form:

```
serverRequest : serverRequest(data(SENDER,VALUE))
```

For example:

- when `GET` (via a browser): emits `serverRequest:serverRequest(data(frontend,get))`
- when `PUT` (or `POST`): emit `serverRequest:serverRequest(data(frontend,item1))`

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

9.2 The robot adapter

Our robot adapter is an 'extension' of the robot that waits for `serverRequest` events emitted by the front-end server.

```
/*
 * ==============================================================
 * robotToServerAdapter.qa
 * ==============================================================
 */
System robotTServerAdapter
Event polar        :  p( Distance, Angle )
Event serverrequest : serverrequest( CONTENT )

Context ctxRobotToServerAdapter ip [ host="localhost" port=8077 ]
Context ctxMbotControl       ip [ host="localhost"  port=8029 ]  -standalone

QActor qafrontendactivator context ctxRobotToServerAdapter{
    Plan init normal [
        println("ACTIVATING NODE SERVER") ;
        nodeOp "./nodeCode/RobotFrontEndServer.js -o" //WARNING: the actor is engaged since PolarToRadar.js waits
    ]
}
QActor qaforserver context ctxRobotToServerAdapter{
    Plan init normal [
    ]
    switchTo waitForEvents

    Plan waitForEvents[ printCurrentEvent ]
    transition stopAfter 600000
        whenEvent serverrequest -> handleServerRequest ,
        whenEvent polar : p( D, A ) do printCurrentEvent //TODO ...
    finally repeatPlan

    Plan  handleServerRequest resumeLastPlan [
        printCurrentEvent ;
        onEvent serverrequest : serverrequest( data(SENDER,DATA) ) -> sendToExternalServer SENDER -m DATA
    ]
}
```

Listing 1.21. `it.unibo.commToRadar.robotToServerAdapter.qa`

Since the adapter is useless without the front-end server, it defines also an actor (`qafrontendactivator`) that activates the server. Thus, to run the system we can:

– execute the robot : `src-gen/it/unibo/cctxMbotControl/MainCtxMbotControl.java`.
– execute the robot-to-server adapter: `src-gen/it/unibo/ctxRobotToServerAdapter.java` (that in its turn activates the front-end server: `nodeCode/robotFrontEndServer.js`).
– perform some `PUT/GET` to port 8098. For example:

```
        curl -X PUT -d item1 http://localhost:8098
        curl -X PUT -d item2 http://localhost:8098
        curl http://localhost:8098
```

9.3 The front-end server implementation

Let us define a front-end server that implements a simple data store and that answers to `PUT/GET` requests on port `8098`. In this version the server will respond to a `HTTP` request with its own answer. In a later version it should respond with an answer built by the *QActor* application.

9.3.1 Connect with the *QActor* application .

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

```
var net      = require('net');
var http     = require("http");
var utils    = require("./utils"); //to handle uncaughtException

var host        = "localhost";
var port        = 8098;
var qaport      = 8077;
var socketToQaCtx = null;
var dataStore = []; //Array of Buffers

//Connect to the QActor application
function connectToQaNode(){
    try{
        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 1.22. `RobotFrontEndServer.js`: connect to *QActor* application

9.3.2 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 1.23. `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.

9.3.3 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);
                emitQaEvent( item );
                buildHtmlResponse( url, method, dataStore, response );
                //WARNING: The response should be given by the qa
          });
    }//if POST PUT
}).listen( port, function(){ console.log('bound to port ' + port);} );
console.log('Server running on ' + port);
```

Listing 1.24. `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.

9.3.4 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:

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

Listing 1.25. `RobotFrontEndServer.js`: emit a *QActor* event

9.3.5 Build an (HTML) response .

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

```
        response.on('error', function(err) { console.error(err); });
        response.statusCode = 200;
        response.setHeader('Content-Type', 'application/json');
        //response.writeHead(200, {'Content-Type': 'application/json'}) //compact form
        var responseBody = {
          //headers: headers, //comment, so to reduce output
          method: method,
          url:   url,
          dataStore:  dataStore
        };
        response.write( JSON.stringify(responseBody) );
        response.end();
        //response.end(JSON.stringify(responseBody)) //compact form
}
```

Listing 1.26. `RobotFrontEndServer.js`: build an (`HTML`) response