POLITECNICO
DI TORINO

System Design Project - Securing Robots and Exoskeletons

LaserBot Battle - Tech Manual

Cirici Stefano	s243942
Panini Francesco	s242547
Ledda Luca	s237447
Spigarelli Edoardo	s241486

Academic Year 2017-2018
April 27, 2018

Contents

1 Introduction

The project scenario consists in a laser battle involving robots remotely driven from a client browser, which have to move and shoot in order to survive. A robot is composed by a Raspberry connected to an Arduino, whose goal is to manage IR lasers, sensors and stepper motors attached to its chassis.

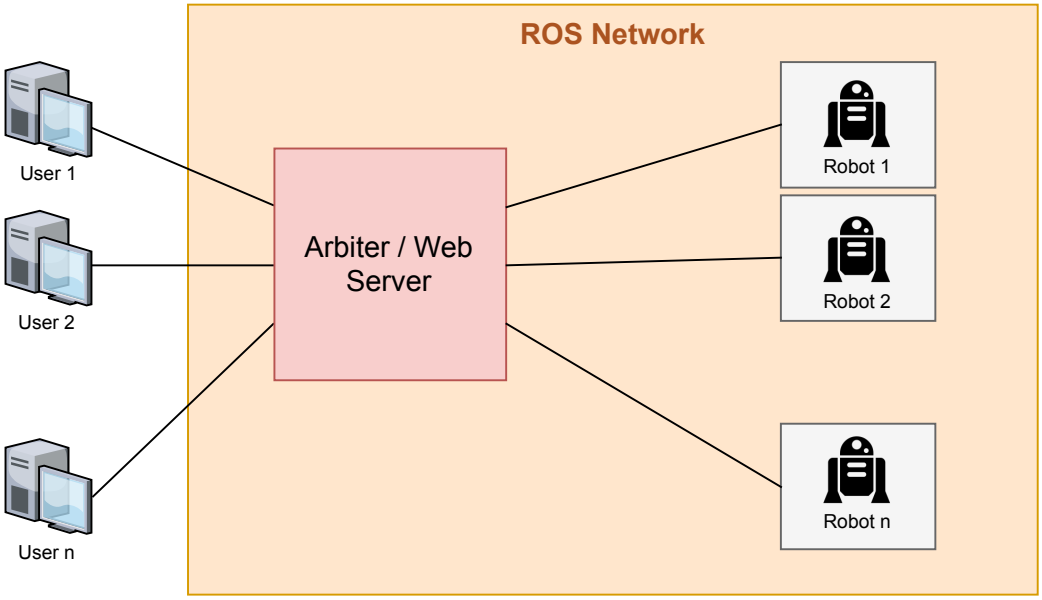

Figure 1: High Level Architecture - Whole scenario

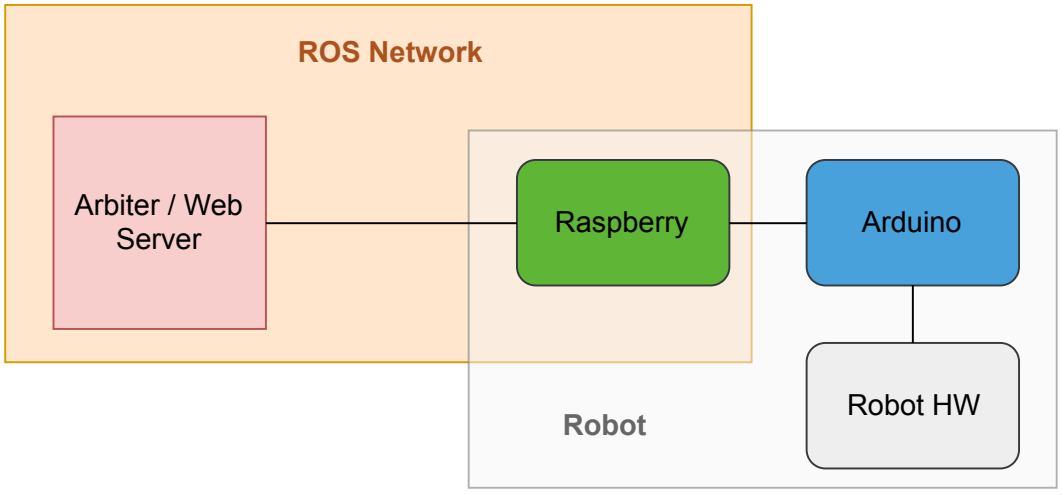

Figure 2: High Level Architecture - Subparts involved

The aim of this document is to propose a detailed documentation focused on each subpart:

- ROS
- Web Server
- Raspberry
- Arduino
- Robot assembly

2 ROS

The Robot Operating System ROS is a flexible framework for writing robot software. It is not a real operating system, but rather a collection of tools, libraries, and conventions that aim to simplify the task of creating robot applications. ROS allows to write general software, independent from the hardware, improving portability and reusability.
For this project we decide to use the Kinetic version, since it is the latest LTS. Furthermore, we did not install ROS directly on Raspbian (Raspberry's OS), but rather on Ubuntu16.04 running in a Docker container (refer to **??** and **??**)

One of the key feature that makes ROS so appealing is it communication infrastructure, commonly referred as middleware. It provides a standard message passing interface for inter-process communication. This system autonomously handle communications between distributed nodes via the asynchronous publish/subscribe mechanism on Topics or Services.
Topics are the most common mean for exchanging messages between processes. They are named busses on which nodes can anonymous publish or subscribe. This decouple the produced from the consumer, such that each node is unaware who is communicating with.

2.1 ROS Bridge

2.2 ROS Serial

3 Web Server

In this section the **Web Server** part is documented. In particular it is the host initiating the ROS network (ROScore is launched here). It is used in order to receive Post requests from clients (browsers) and to forward them to robots raspberry and vice versa in ROS messages form. Moreover, it is in charge of updating battle status (robot life, logged users and actions to be performed in response to received commands).

3.1 Flask

Flask is a micro-framework which depends on some libraries such as Werkzeug and Jinja2. In particular, Werkzeug is a toolkit for the standard Python interface between web applications and servers whereas Jinja2 renders templates. For the purposes of this project Flask has been used in its base version in charge of managing HTTP requests and keeping trace of users data during the whole game duration. However its functionalities can be further improved by adding external modules.

The Server core involves the following files:

- main.py
- ID_service_server.py
- app.py
- robot.py
- users.py

3.1.1 main.py

This is the main file of the Server core which is launched once the Server is up. It is in charge of:

- running the app that handles HTTP requests from clients in a new thread (Listing 1 line 7).
- invoking the `add_new_robot_server()` service from ID_service_server.py which handles the registration of new robots to the network (Listing 1 line 8).

It is important to notice that the `add_new_robot_server()` operates for the entire server life (the server is blocked in this function). Thus, an additional thread (app.main) is invoked in order for the server to be multi-tasking (handle both robot registration service and HTTP requests).

```
1  #!/usr/bin/env python
2  import thread
3  import app
4  import ID_service_server
5
6  def main():
7      thread.start_new_thread(app.main, () )
8      ID_service_server.add_new_robot_server()
9
10 if __name__ == '__main__':
11     main()
```

Listing 1: main.py - Source code

3.1.2 ID_service_server.py

This file contains the functions to manage the registration of new robots to the network. In details:

- From the main.py, `add_new_robot_server()` is invoked (Listing 1 line 8). This function initializes a ROS node which instantiates a service of type "AddNewRobot" and waits for a service-request to be received. When a service-request is get, the `handle_new_robot` function is called.
- `handle_new_robot()` associates an ID to the new robot and returns the program flow to `add_new_robot_server()`.

This flow is repeated each time a new robot requires to register to the network.
It is worth noticing that as said in section 3.1.1, the server program flow is blocked within the add_new_robot_server() function because of the spin() presence. Basically, the spin() never returns but let the server be reactive on service-requests.

```
1  #!/usr/bin/env python
2  from laser_bot_battle.srv import *
3  import rospy
4  from robot import robots
5
6  def handle_new_robot(req):
7      print "Request received from robot"
8
9      # Get first available robot ID
10     Robot_ID = robots.getAvailableID()
11
12     # Add robot with robotID to robot list
13     robots.addRobot(Robot_ID)
14
15     # The Robot ID is returned to the robot requiring it
16     return AddNewRobotResponse(Robot_ID)
17
18 def add_new_robot_server():
19     rospy.init_node('robots_server')
20     # The service add_new_robot is created and up to now can be required by a client
21     s = rospy.Service('add_new_robot', AddNewRobot , handle_new_robot)
22     print "Ready to add a new robot!"
23     rospy.spin()
24
25 if __name__ == "__main__":
26     add_new_robot_server()
```

Listing 2: ID_service_server.py - Source code

3.1.3 app.py

The app.py is the core of the webserver application. Two global variables are necessary to manage some internal information. The `gameStarted` keep trace of the status of the game and can assume 3 values:

- 0 : the game is stopped (is not started or is finished).
- 1 : the game is about to start (waiting for the countdown to expire).
- 2 : the game is started (users can play the game).

The `timeLeft` variable keep the count of the countdown to expires. Is set in the `playerReady` function when at least 2 players are ready (see section 3.1.3.7), and decremented in the `gameStatus` thread (see section 3.1.3.6). Its value can bee seen from the client a POST request to the `updateGameStatus` function (see section 3.1.3.4).

In the `app.py` a Flask webserver object is created with the possibility to render only three webpages:

- Index: This is the user Login webpage, reachable at the address `/` (Listing 3 line 16).
- Home: The home page in which the game is played, reachable at the address `/home` (Listing 3 line 21).
- About: The about page that briefly describes the project and the developers, reachable at the address `/about` (Listing 3 line 26)

```
10 timeLeft = 0
11 gameStarted = int(0)
12
13 app = Flask(__name__, static_folder='static', static_url_path='/static')
14
15 # Index Login Page
16 @app.route('/')
17 def index():
18     return render_template('index.html')
19
20 # User home page
21 @app.route('/home')
22 def home():
23     return render_template('home.html')
24
25 # Project about page
26 @app.route('/about')
27 def about():
28     return render_template('about.html')
```

Listing 3: Portion of app.py - Reachable web pages

Other addresses are reachable only through HTTP POST request. These implements functions that can be called from the client (through ajax javascript request) and perform operation on the server or return useful data.

3.1.3.1 main

This function is called (as a separate thread) by the `main.py` (see section 3.1.1). It launches as a separate thread the function `checkAlive` and then run the Flask app webserver. The debug and the reloader are disabled after the initial developing step of the project. the host is set to 0.0.0.0 which means that the webserver is accessible outside the local network. The function code can bee seen in Listing 4.

```
194 def main():
195     threadAlive = threading.Thread(target=checkAlive)
196     # Launch checkAlive function as a separate thread
197     threadAlive.start()
198 
199     # Run flask web app
200     app.run(debug=False, use_reloader = False, host='0.0.0.0')
```

Listing 4: portion of app.py - main function

3.1.3.2 signUpUser

This function manage the login request of an user. From the received POST request, extract the username to be added to the user list and check if it is available (there is no other user with the same name) otherwise return a "UNAVAILABLE" status. Then get the first available robot ID (if no robots are available returns a "NO_ROBOTS" status) and associate the user to the robot. If the association fails (e.g. in the meanwhile the robot has being taken by another user) return a "ROBOT_UNAVAILABLE" status. In the end add the user to the user list and return an "OK" status, the username of the user that has been added and the ID of the robot to whom the user has been associated. The function code can bee seen in Listing 5.

```
46 # signUpUser function:
47 #   add user to user list if name is available
48 @app.route('/signUpUser', methods=['POST'])
49 def signUpUser():
50     name = request.form['data']
51 
52     # if username is availabe :
53     if users.isNameAvailable(name) :
54 
55         # check first available robot id
56         robotN = robots.getAvailableRobot();
57         if robotN == -1 :
58             return json.dumps({'status':'NO_ROBOTS', 'user':name})
59 
60         # associate user to robot (check if fail)
61         if not robots.addUserToRobot(robotN, name) :
62             return json.dumps({'status':'ROBOT_UNAVAILABLE', 'robot':robotN})
63 
64         # add it to users list
65         users.addUser(name, robotN)
66         return json.dumps({'status':'OK', 'user':name, 'robot':robotN})
67 
68     else :
69         # else return UNAVAILABLE error
70         return json.dumps({'status':'UNAVAILABLE', 'user':name})
```

Listing 5: portion of app.py - signUpUser function

3.1.3.3 signOutUser

This function manage the sign out request of an user. From the received POST request, extract the username to be removed from the user list and check if it is available (if the username is available it has been registered) otherwise return a "UNREGISTERED" status. Then de-associate the user from the robot (the robot can be associated to a different user afterwards) and finally

delete the user from the list returning a "FAILED" status if the operation fails, "OK" status otherwise. The code can bee seen in Listing 6.

```
# signOutUser function:
#   delete user from user list if present
@app.route('/signOutUser', methods=['POST'])
def signOutUser():
    name = request.form['data']

    # if user is in users list :
    if not users.isNameAvailable(name) :

        robots.removeUserFromRobot(name)

        # delete from list
        if users.delUser(name) :
            return json.dumps({'status':'OK', 'user':name})
        else :
            return json.dumps({'status':'FAILED', 'user':name})
    else :
        # else return UNREGISTERED error
        return json.dumps({'status':'UNREGISTERED', 'user':name})
```

Listing 6: portion of app.py - signOutUser function

3.1.3.4 updateGameStatus

This function returns information about the game status. The complete list of users (in a JSON format), the `gameStarted` and `timeLeft` value (see section 3.1.3 for further information). The code can bee seen in Listing 7.

```
# updateGameStatus function:
#   return list of logged in users, game status and time to begin (json format)
@app.route('/updateGameStatus', methods=['POST'])
def updateGameStatus():
    return json.dumps({'status':'OK', 'users':users.toString(),
     'game': gameStarted, 'timeLeft':timeLeft}, default=userDefault)
```

Listing 7: portion of app.py - updateGameStatus function

3.1.3.5 getAvailableRobots

This function returns the number of the robots connected to the webserver that have not been assigned to an user yet. This means that are available and can be used if a new player wants to login.The code can bee seen in Listing 8.

```
# getAvailableRobots function:
#   return number of available robots (json format)
@app.route('/getAvailableRobots', methods=['POST'])
def getAvailableRobots():
    #print robots.toString()
    #print "num robots : ", robots.getAvailableRobotsN()
    return json.dumps({'status':'OK', 'availableR':robots.getAvailableRobotsN()})
```

Listing 8: portion of app.py - getAvailableRobots function

3.1.3.6 gameStatus

This function manage the status of the game. It is called by a separate thread and run in parallel with the application. Once called set the `gameStarted` to 1 and start decreasing the `timeLeft` variable once every second. When it reaches 0 the game can start, the `gameStarted` is set to 2. The thread check every half second if all players but one are dead which means the game is finished. The `gameStarted` is set back to 0 and all users ready status is reset. The code can bee seen in Listing 9.

```
111 # gameStatus function:
112 #   start countdown, start game and check end of game
113 def gameStatus():
114     global timeLeft
115     global gameStarted
116     gameStarted = 1
117     print "countdown started"
118 
119     # countdown to game start
120     while timeLeft > 0:
121         time.sleep(1)
122         timeLeft -= 1
123         print timeLeft,
124 
125     print "Starting game!"
126     gameStarted = 2
127 
128     # check for game to end (only 1 player alive)
129     while users.getUsersAlive() > 1 :
130         print "users alive:", users.getUsersAlive()
131         users.usersSort()
132         time.sleep(0.5)
133 
134     # game finished
135     print "Game finished"
136     gameStarted = 0
137     users.clearUsersReady()
```

Listing 9: portion of app.py - gameStatus function

3.1.3.7 playerReady

This function get the user that ask to set its ready status and the ready status value from the HTTP POST request. If the game is started the user cannot modify its ready status and the function will return a `"STARTED"` status. This happen because if the user ready status is 1 (ready) when the game is already started, the user is playing in the current game session (and can not unset its ready status) but if the ready status is 0 (not ready) the user can not join the game because is already started. After this, the user status is set (on failure the function return a `"ERROR"` status) and if at least 2 users are ready, the countdown is started: all users life is reset (to 100), the countdown timer is set to 15 seconds and the `gameStatus` function is called as a separate thread (see section 3.1.3.6), returning an `"OK"` status. The code can bee seen in Listing 10.

```
142 # playerReady function:
143 #   update player ready status
144 @app.route('/playerReady', methods=['POST'])
145 def playerReady():
```

```
146     global timeLeft
147     global gameStarted
148     name = request.form['user']
149     ready = request.form['ready']
150     print "name:", name, ".ready:", ready,"."
151 
152     # Game already started
153     if gameStarted == 2 :
154         return json.dumps({'status':'STARTED'})
155 
156     #set user ready
157     if users.setReady(name, ready) :
158         # If more than 2 players are ready
159         if users.getUsersReady() > 1 :
160             users.resetUsersLife()
161             timeLeft = 15
162             # Launch countdown to game start as a new thread
163             threadGameStatus = threading.Thread(target=gameStatus)
164             threadGameStatus.start()
165 
166         return json.dumps({'status':'OK','user':name})
167     else :
168         return json.dumps({'status':'ERROR','user':name})
```

Listing 10: portion of app.py - playerReady function

3.1.3.8 incAlive

This function is called from the raspberry which send a POST to the `/incAlive` address. It get the robot ID passed from the caller and calls the `isAlive` function of the robots list, increasing an internal value to let the webserver know that the robot is alive. The code can bee seen in Listing 11.

```
171 # incAlive function:
172 #   increase alive value for robot that call this function
173 @app.route('/incAlive', methods=['POST'])
174 def incAlive():
175     robotID = int(request.form['ID'])
176 
177     if robotID != "":
178         if robots.isAlive(robotID):
179             return json.dumps({'status':'OK'})
180 
181     return json.dumps({'status':'ERROR'})
```

Listing 11: portion of app.py - incAlive function

3.1.3.9 checkAlive

This function is called by the `app.main` as a separate thread that run in parallel to the webserver for the whole duration of its life (until it get killed). It simply calls, every 2 seconds, the `clearAlive` function of the robots list that clear the alive status of all robots and disconnect the dead robots (the robots that in these 2 seconds have not called the `incAlive` function at least once). The code can bee seen in Listing 12.

```
184 # checkAlive function:
```

```
185 #   check every 2 sec if all connected robots are still alive, if not delete them
186 def checkAlive():
187     while True:
188         #print "ClearAlive"
189         robots.clearAlive();
190         time.sleep(2)
```

Listing 12: portion of app.py - checkAlive function

3.1.4 users.py

This file contains all the classes necessary to the server to keep trace of user information. In particular two main classes are defined as explained below.

3.1.4.1 User class

It characterizes the User through the following properties:

- name : username chosen by the user during login phase.
- life : indicator of the remaining life of the user's robot ([0,100]).
- robot : ID of the associated robot.
- ready : specifies whether the user is willing to start a new battle or not.

```
12 # User class
13 class User:
14     name = ""
15     life = 100
16     robot = 0
17     ready = 0
18 
19     def __init__(self, name, robot):
20         self.name = name
21         self.robot = robot
22         self.life = 100
23         self.ready = 0
```

Listing 13: Users.py - User class source code

3.1.4.2 Users class

It is a list of "User" (Listing 13). All the functions needed for the list manipulation are defined in this class. A brief explanation is reported below:

- `addUser(name, robot)` : Add a user to the list. Input parameters "name" and "robot" specify the username and the associated robot ID of the user to be added.
- `delUser(name)` : Deletes a User from the list. Input parameter "name" specifies the username of the user to be deleted.
- `isNameAvailable(name)` : It search a user within the list and return "True" if the user has been found or "False" if not. Input parameter "name" specifies the username of the user to be searched.

- `toString()` : It returns a stringify JSON version of the entire users list. Useful when users info has to be sent within HTTP responses.
- `usersNum()` : Return the number of the User within the users list.
- `setReady(name, ready)` : Update the user Ready status as "ready" (0 - not ready, 1 - ready). The other input parameter "name" specifies the username of the user for which the status has to be updated.
- `getUsersReady()` : Return the list of users that are ready (those for which Ready attribute is set at "1").
- `getUsersAlive()` : Return the list of users that are alive during a battle (Ready = 1).
- `clearUsersReady()` : Restore the Ready status of all users to "not ready" (Ready = "0").
- `resetUsersLife()` : Restore the remaining life of all users to the maximum value (life = "100").
- `hit(name)` : Implement the hit action lowering the user's life during a battle. Input parameter "name" specifies the username of the hit user.
- `userSort()` : Sort the users involved in a battle by the life attribute in a decreasing way (the last one is the most damaged).

It is important to specify that Username attribute within the User structure is unique. Thus, it is seen as a ID reference for the User element within the users list.

```
26 # list of users class
27 class Users:
28     users = []
29 
30     # add new user to list
31     def addUser(self, name, robot):
32         self.users.append( User(name, robot) )
33 
34     # delete user from list
35     def delUser(self, name):
36         for u in self.users:
37             if u.name == name :
38                 self.users.remove(u)
39                 return True
40         return False
41 
42     # check if user with a given username is already in list
43     def isNameAvailable(self, name):
44         for u in self.users:
45             if u.name == name :
46                 return False
47         return True
48 
49     # return json string of users list
50     def toString(self):
51         return json.dumps(self.users, default=userDefault)
52 
53     # return num of users list
54     def usersNum(self):
55         num = 0
```

```
56          for u in self.users:
57              if u.name == "" :
58                  return -1
59              num += 1
60          return num
61
62      # set ready status of player
63      def setReady(self, name, ready):
64          for u in self.users:
65              if u.name == name :
66                  u.ready = int(ready)
67                  return True
68          return False
69
70      # get number of players ready to start
71      def getUsersReady(self):
72          num = 0
73          for u in self.users:
74              if u.ready != 0 :
75                  num += 1
76          return num
77
78      # get number of players still alive
79      def getUsersAlive(self):
80          num = 0
81          for u in self.users:
82              if u.life > 0 and u.ready == 1 :
83                  num += 1
84          return num
85
86      # clear all players ready status
87      def clearUsersReady(self):
88          for u in self.users:
89              u.ready = 0
90
91      # set (reset) all players life to 100
92      def resetUsersLife(self):
93          for u in self.users:
94              u.life = 100
95
96      # reduce user life when hit
97      def hit(self, name):
98          #print self.toString()
99          from app import gameStarted
100         if gameStarted == 2:
101             for u in self.users :
102                 #print "u.name", u.name, " name", name, "."
103                 if u.name == name :
104                     u.life -= HITDAMAGE
105                     if u.life < 0 :
106                         u.life = 0
107                     print "User", name, "has been hit. LIFE:", u.life
108                     return True
109         return False
110
111     # sort key for sorting users by life (if ready)
112     def __sortKey(self,x):
113         if x.ready == 1:
114             return x.life
115         return 101
116
117     # sort userlist
118     def userSort(self):
119         self.users.sort(key=self.__sortKey, reverse=True)
```

Listing 14: Users.py - Users class source code

Finally, an instantiation of the users list is done (Listing 15). This is crucial since this instantiated list is global and shared by all those files ".py" that import it. An example of such import can be seen at section 3.1.3 - line 3.

```
121 users = Users()
```

Listing 15: Users.py - users list instantiation

3.1.5 robot.py

This file contains all the classes necessary to the server to keep trace of robot information. Similarly to section 3.1.4, two main classes are defined:

3.1.5.1 Robot class

It characterizes the Robot through the following properties:

- ID : ID of the registered robot.
- user : username of the associated user.
- alive : indicator of the life status of the robot (live or not).

Some functions are also defined for a proper Robot manipulation:

- `robotHit(msg)` : Callback function invoked each time a message on "/response" topic is received by the server (acting as subscriber). It prints on the server terminal that robotN has been hit and calls the users.hit() function to lower the associated user's life.
- `createSub()` : Create a node subscriber on a topic involving an empty message type. Each time a robot is hit, raspberry communicates that information through this topic. When the server receives that message invokes the robotHit() callback function (described above).

```
12 # Robot class
13 class Robot:
14     ID = 0
15     user = ""
16     alive = 1
17 
18     def __init__(self, id, user=None):
19         self.ID = id
20         self.alive = 1
21         if user != None :
22             self.user = user
23         self.threadSub = threading.Thread(target=self.__createSub)
24 
25     def __robotHit(self, msg):
26         print "Robot"+str(self.ID)+" has been hit"
27         users.hit(self.user)
28 
29     def __createSub(self):
30         #print "creating node: ", "/Robot"+str(self.ID)+"_subscriber"
```

```
31          #rospy.init_node("Robot"+str(self.ID)+"_subscriber")
32
33          __sub = rospy.Subscriber("/Robot"+str(self.ID)+"/response", Empty, self.__robotHit)
34          print 'Node initialized'
35          rospy.spin()        #wait
```

Listing 16: robot.py - Robot class source code

3.1.5.2 Robots class

It is a list of "Robot" (Listing 16). All the functions needed for the list manipulation are defined in this class. A brief explanation is reported below:

- `addRobot(id)` : Add a robot to the list. Input parameter "id" specifies the robot ID to be added.
- `getAvailableRobot()` : Return the first available robot ID among all robots available (those that are registered but not assigned to a user yet). -1 is returned if no robots are available.
- `getAvailableRobotN()` : Return the number of robots available (those that are registered but not assigned to a user yet).
- `getAvailableID()` : Return the first available ID to be assigned to a new registered robot.
- `addUserToRobot(id, name)` : Associate a user to a robot. Input parameters "id" and "name" specify ID and username related to the robot and user that are going to be associated to each other.
- `removeUserFromRobot(name)` : It disassociate the user from a robot. Input parameter "name" specifies the name of the user that is going to be disassociate to its robot.
- `isAlive(id)` : Increment the Alive status for a robot (this function is invoked by the server each time a POST request on /incAlive is received from raspberry to signal that robot is alive). Input parameter "id" specifies the robot ID for which the Alive status has to be incremented.
- `delRobot(id)` : Delete a robot from the list. Input parameter "id" specifies the ID of the robot to be deleted.
- `clearAlive()` : Check which robots within the list has got Alive status equal to "0" (Alive status equal to zero means that no POST requests to /incAlive are sent from raspberry and so robot is considered not alive). These robots are first disassociated to their users and secondly deleted from the list because "not alive".
- `returnRobotList()` : Return the entire robots list.
- `toString()` : It returns a stringify JSON version of the entire robots list. Useful when robots info has to be sent within HTTP responses.

```
# list of robots class
class Robots:
    robots = []


    # add new robot to list (and keep it sorted by ID)
    def addRobot(self, id):
        self.robots.append( Robot(id) )
        self.robots.sort(key=lambda x: x.ID)
        print "Added robot with ID", id


    # get first available robot (ID) from robots list
    def getAvailableRobot(self):
        for r in self.robots:
        #print "id: ", r.ID , "user: ", r.user
            if r.user == "" :
                return r.ID
        return -1

    # get number of available robots
    def getAvailableRobotsN(self):
        num = 0
        for r in self.robots:
        #print "id: ", r.ID , "user: ", r.user
            if r.user == "" :
                num += 1
        return num

    # get first unused ID starting from 0
    def getAvailableID(self):
        ID = 0
        for r in self.robots:
            if r.ID == ID :
                ID += 1
            else :
                break
        return ID

    # associate user name to robot
    def addUserToRobot(self, id, name):
        for r in self.robots:
            if r.ID == id and r.user == "" :
                r.user = name
                if not r.threadSub.isAlive():
                    r.threadSub.start()
                return True
        return False

    # de-associate user from robot
    def removeUserFromRobot(self, name):
        for r in self.robots:
            if r.user == name :
                r.user = ""
                return True
        return False

    # delete robot from list
    def delRobot(self, id):
        for r in self.robots:
            if r.ID == id :
                self.robots.remove(r)
                return True
```

```
101         return False
102 
103     # signat that robot is alive
104     def isAlive(self, id):
105         for r in self.robots:
106             if r.ID == id :
107                 r.alive += 1
108                 return True
109         return False
110 
111     # clear alive status of all robots
112     def clearAlive(self):
113         for r in self.robots:
114             #print "Checking robot ", r.ID, " alive is ", r.alive
115             if r.alive == 0 :
116                 # If not delete robot and user
117                 print "Robot ", r.ID, " is dead."
118                 if r.user != "":
119                     print " Player ", r.user, " disconnected"
120                     users.delUser(r.user)
121                 self.robots.remove(r)
122             else:
123                 r.alive = 0
124 
125     def returnRobotList(self):
126         return self.robots
127 
128 
129     # return json string of robots list
130     def toString(self):
131         return json.dumps(self.robots, default=userDefault)
```

Listing 17: robot.py - Robots class source code

Finally, an instantiation of the robots list is done (Listing 18). This is crucial since this instantiated list is global and shared by all those files ".py" that import it.

```
133 robots = Robots()
```

Listing 18: robot.py - robots list instantiation

3.2 Web Application

4 Docker

5 Raspberry

In this section the **Raspberry** part is documented. A brief explanation on its main tasks:

- generates a ROS node (robot) exchanging messages with the ROS master and the Arduino board
- forward commands coming from the Server to Arduino (performing the proper actuations)
- retrieve sensors notification from Arduino to be forwarded to the Server (updating battle status)

5.1 ID_service_client.py

5.1.1 pingThread.py

6 Arduino

In this section the **Arduino** part is documented. In particular it is used to drive the motors and the IR emitter, in response of Raspberry requests. It is provided with IR sensors, such that detecting and notifying when a robot is hit.

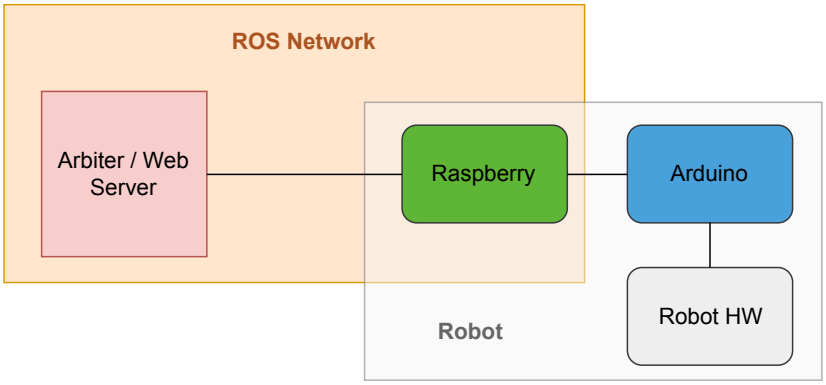

Figure 3: Ros network - Robot Architecture