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
June 17, 2018

Contents

1 Introduction **1**

2 ROS **3**
2.1 Introduction . . . 3
2.1.1 Nodes . . . 3
2.1.2 Topics . . . 3
2.1.3 Services . . . 3
2.2 Installing ROS . . . 3
2.3 Custom Message and Service Creation . . . 4
2.4 ROS Bridge . . . 5
2.5 ROS Serial . . . 5

3 Web Server **7**
3.1 Flask . . . 8
3.1.1 main.py . . . 8
3.1.2 ID_service_server.py . . . 8
3.1.3 app.py . . . 9
3.1.3.1 main . . . 11
3.1.3.2 signUpUser . . . 11
3.1.3.3 signOutUser . . . 12
3.1.3.4 updateGameStatus . . . 13
3.1.3.5 getAvailableRobots . . . 13
3.1.3.6 gameStatus . . . 13
3.1.3.7 playerReady . . . 14
3.1.3.8 incAlive . . . 15
3.1.3.9 checkAlive . . . 15
3.1.4 users.py . . . 16
3.1.4.1 User class . . . 16
3.1.4.2 Users class . . . 16
3.1.5 robot.py . . . 19
3.1.5.1 Robot class . . . 19
3.1.5.2 Robots class . . . 20
3.2 Web Application . . . 23
3.2.1 index.html . . . 24
3.2.2 about.html . . . 25
3.2.3 home.html . . . 26
3.2.3.1 Streaming video . . . 27
3.2.4 gameArea.js . . . 27
3.2.4.1 ROS client setup . . . 27
3.2.4.2 Capturing user commands . . . 28
3.2.5 manageUsers.js . . . 30
3.2.5.1 delUser . . . 30
3.2.5.2 userSignOut . . . 32
3.2.5.3 userLogin . . . 32

3.2.5.4 changeImage . . . 34
3.2.5.5 updateGameStatus . . . 34
3.2.5.6 updateAvailableRobots . . . 38
3.2.5.7 updateReady . . . 39
3.2.5.8 help . . . 40

4 Docker **41**
4.1 Introduction . . . 41
4.2 Installing Docker . . . 41
4.3 Dockerfile . . . 41
4.4 Most used Docker commands . . . 42

5 Raspberry **43**
5.1 ID_service_client.py . . . 43
5.1.1 pingThread.py . . . 44

6 Arduino **46**
6.1 Arduino Rosserial Setup . . . 46
6.2 Arduino Libraries . . . 47
6.2.1 ROS Custom Messages Integration . . . 47
6.2.2 AccelStepper Library . . . 47
6.2.3 Timer Library . . . 47
6.3 Arduino Sketch . . . 48

List of Figures

1 High Level Architecture - Whole scenario . . . 1
2 High Level Architecture - Subparts involved . . . 1
3 Web Server functions scheme . . . 7
4 Web Client functions scheme . . . 23
5 index.html - Browser view . . . 25
6 about.html - Browser view . . . 26
7 home.html - Browser view . . . 26
8 Ros network - Robot Architecture . . . 46

List of Listings

1 Robot_msg.msg . . . 4
2 AddNewRobot.srv . . . 4
3 main.py - Source code . . . 8
4 ID_service_server.py - Source code . . . 9
5 Portion of app.py - Reachable web pages . . . 10
6 portion of app.py - main function . . . 11
7 portion of app.py - signUpUser function . . . 12
8 portion of app.py - signOutUser function . . . 12
9 portion of app.py - updateGameStatus function . . . 13

10 portion of app.py - getAvailableRobots function . . . 13
11 portion of app.py - gameStatus function . . . 13
12 portion of app.py - playerReady function . . . 14
13 portion of app.py - incAlive function . . . 15
14 portion of app.py - checkAlive function . . . 15
15 Users.py - User class source code . . . 16
16 Users.py - Users class source code . . . 17
17 Users.py - users list instantiation . . . 18
18 robot.py - Robot class source code . . . 19
19 robot.py - Robots class source code . . . 20
20 robot.py - robots list instantiation . . . 22
21 index.html - ready event handler . . . 24
22 index.html - User login . . . 25
23 index.html - Robot available info . . . 25
24 home.html - Other users information . . . 27
25 gameArea.js - ROS client setup . . . 28
26 gameArea.js - User commands published on topic . . . 28
27 manageUsers.js - user object . . . 30
28 manageUsers.js - user class . . . 31
29 manageUsers.js - userSignOut . . . 32
30 manageUsers.js - userLogin . . . 32
31 manageUsers.js - changeImage . . . 34
32 manageUsers.js - updateGameStatus . . . 34
33 manageUsers.js - updateAvailableRobots . . . 38
34 manageUsers.js - updateReady . . . 39
35 manageUsers.js - help . . . 40
36 ID_service_client.py . . . 43
37 pingThread.py . . . 45
38 rosserial_node_arduino.ino - Robot Parameters . . . 48
39 rosserial_node_arduino.ino - Global variables and objects . . . 49
40 rosserial_node_arduino.ino - Subscriber callback function . . . 49
41 rosserial_node_arduino.ino - IR sensor interrupt handler . . . 50
42 rosserial_node_arduino.ino - Timer interrupt handler . . . 51
43 rosserial_node_arduino.ino - Arduino setup function . . . 51
44 rosserial_node_arduino.ino - Arduino loop function . . . 51

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.

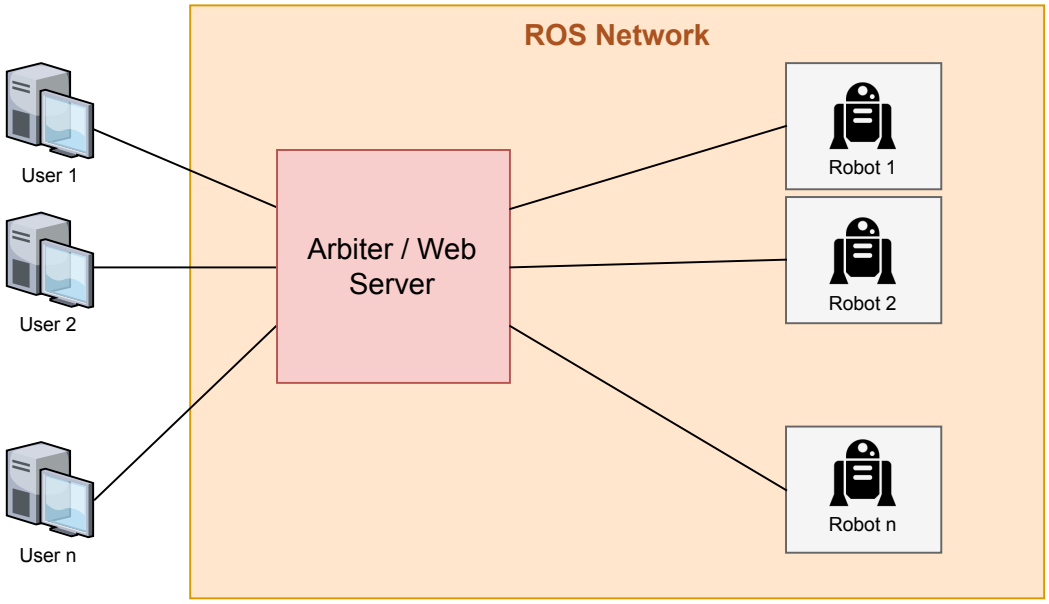

Figure 1: High Level Architecture - Whole scenario

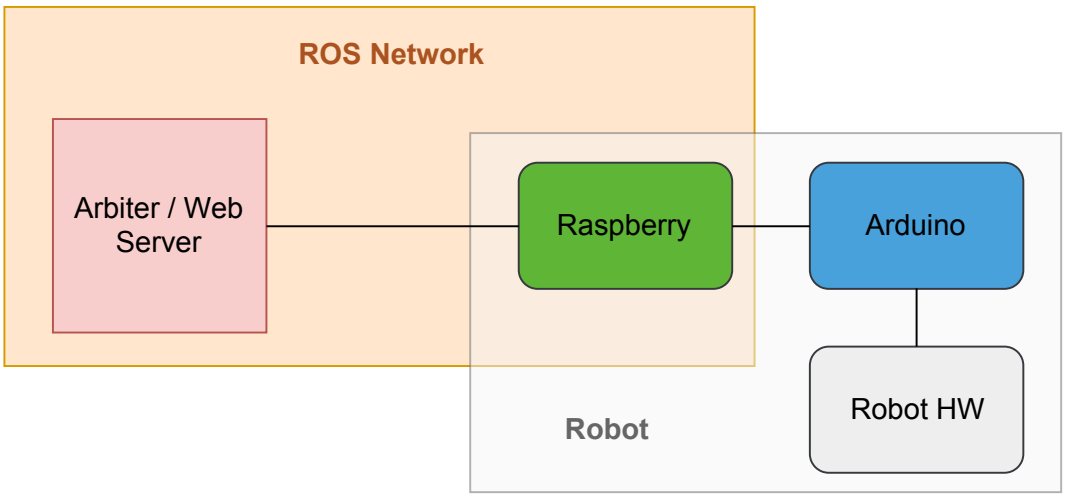

Figure 2: High Level Architecture - Subparts involved

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

- ROS section 2
- Web Server section 3
- Docker section 4
- Raspberry section 5
- Arduino section 6

2 ROS

2.1 Introduction

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.

One of the key feature that makes ROS so appealing is its communication infrastructure, commonly referred as middleware. It provides a standard message passing interface for inter-process communication and autonomously handle communications between distributed nodes via the asynchronous publish/subscribe mechanism on Topics or Services.

2.1.1 Nodes

In ROS terminology a node is nothing but an executable file within a ROS package. Nodes can publish or subscribe to Topics and can use or provide Services. *ROSCore* is the first node which has to be created when dealing with ROS. This will initialize the ROS network, allowing nodes to communicate with each other.

2.1.2 Topics

Topics are the most common mean for exchanging messages. They are named busses on which nodes can anonymously publish or subscribe. This decouple the producer from the consumer, such that each node is unaware of who is communicating with or who is receiving data from. Nodes that are interested in specific data *subscribe* to the relative topics, while nodes that generate data *publish* to related topics. Nevertheless topics are not appropriated for *Request&Respose* interaction. For this purpose *Services* have been introduced.

2.1.3 Services

Services are defined by a pair of messages: one for the *request* and one for the *response*. A service is made available by a ROS node, and a client can call the service by issuing the request message and waiting for the reply.

2.2 Installing ROS

In this project the ROS Kinetic version is used, since it is the latest LTS. ROS will be not directly installed on Raspbian (Raspberry's OS), but rather in a Docker container based on Ubuntu Xenial 16.04 (refer to section 4)
ROS Kinetic can be installed on a Ubuntu system following this guide. Note that only the basic package `ros-kinetic-ros-core` is installed (since few ROS functionalities are needed).

To setup ROS Environment, this command is executed on every opened shell or inserted into the file `.bashrc` such that it is automatically executed every time a new shell is opened :

```
$ source /opt/ros/kinetic/setup.bash
```

A workspace directory called `catkin_ws` can now be created:

```
$ mkdir -p ~/catkin_ws/src
$ cd ~/catkin_ws
$ catkin_make
```

This commands will create a `CMakeLists.txt` file in the `src` folder. Moreover it creates the `build` and `devel` folders within `catkin_ws` and will resolve all dependencies.
For this application a new ROS package, named `laser_bot_battle`, has been created. During its creation, only a few package dependencies were provided: rospy (to install Python compatibility library) and std_msgs (standard messages library).

```
$ catkin_create_pkg laser_bot_battle std_msgs rospy
$ catkin_make
$ source devel/setup.bash
```

2.3 Custom Message and Service Creation

Custom messages and custom services have to be created, since standard ones provided by ROS were not enough for this project purposes.
Communication over topics between nodes happens by sending ROS messages. Both publisher and subscriber exchanging informations on same topic must send and receive the same *type* of message. The implemented custom message is of type Robot_msg.msg. The .msg file is a simple text file that describe the fields that the new ROS message will have and it is stored in the 'msg' directory placed inside `laser_bot_battle` package. It contains two integers mapping the direction where the robots have to move and a boolean to identify if it is shooting or not. The other message type exchanged in the application is `std_msgs/Empty`, which is sent when a robot is hit by the laser.

```
1 int8 linear_x
2 int8 angular_z
3 bool shoot
```

Listing 1: Robot_msg.msg

Any `.msg` file in the msg directory will generate code for use in all supported languages. The C++ message header file will be generated in `/catkin_ws/devel/laser_bot_battle/`. This will be used in section 6.2.1.

The service introduced consist in an "identification" request from a new robot service client, and in "ID" response from the service master. The request will be performed by each robot at power on, while trying to connect to the ROS network, while the response will be provided by the network master node (Arbiter/Web Server in Figure 1). The implemented service is formatted as below:

```
1
2
3 ---
4 int8 ID
```

Listing 2: AddNewRobot.srv

In this case, the request is a simple `std_msgs/Empty` message (request is what is above "—"), since it is only needed to trigger the service. The response, instead, is an integer containing the first available ID (request is what is below "—"). For a better and deeper understanding of what the ID is used for, consult the section 5.1. The service file has been placed in a 'srv' directory within `laser_bot_battle` package.

Once both message and service has been defined, the `package.xml` and `CMakeLists.txt` files have to be modified as stated in ROS msg and srv documentation. This ensure that during the `catkin_make` process, msg and srv will be turned into Python source code. Once the make process has terminated, to be sure that everything has been done correctly and that ROS is now able to see the new message and service, these command can be issued:

```
$ rosmsg show laser_bot_battle/Robot_msg
$ rossrv show laser_bot_battle/AddNewRobot
```

These should return the fields defined in Robot_msg and AddNewRobot respectively. If this is not the case, probably some errors arose due to an incorrect modification of `package.xml` and `CMakeLists.txt` files.

2.4 ROS Bridge

ROS Bridge allows to extend ROS network to a remote Web-app. This protocol translate the WebSocket protocol using JSON standard message to a ROS-based protocol.
After the installation of ROS and rosbridge (see section 2.2), rosbridge can be run through its launch file as:

```
$ roslaunch rosbridge_server rosbridge_websocket.launch
```

This command will create two nodes:

- `rosbridge_websocket`: create a Webserver on port 9090 by default
- `rosapi`: translate messages from Websocket to ROS and viceversa

Now that rosbridge has been launched and a WebSocket connection is available, an HTML webpage has been created to send and receive messages through rosbridge.

2.5 ROS Serial

Ros serial is a protocol to serialize ROS messages and multiplex multiple topics to a specific device connected to a serial port, extending ROS functionality to embedded devices. There are different implementation of this library, each one is board-specific and contains ad-hoc extensions required to run the `rosserial_client` for the target board.
For this project, robot sensors and actuators are driven with Arduino, since it offers an easy and standard layer to interface Hardware and Software. So, rosserial_arduino has to be installed as shown in section 6.1.

Once the connection is established, Raspberry becomes the master, while Arduino becomes the slave. Control messages flow from master to slave, describing how actuators should behave, while messages containing data sensors flow from slave to master, where they will be processed.

3 Web Server

In this section the Web Server part is documented. The server consists in the host initiating the ROS network (roscore command 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). See Figure 3.

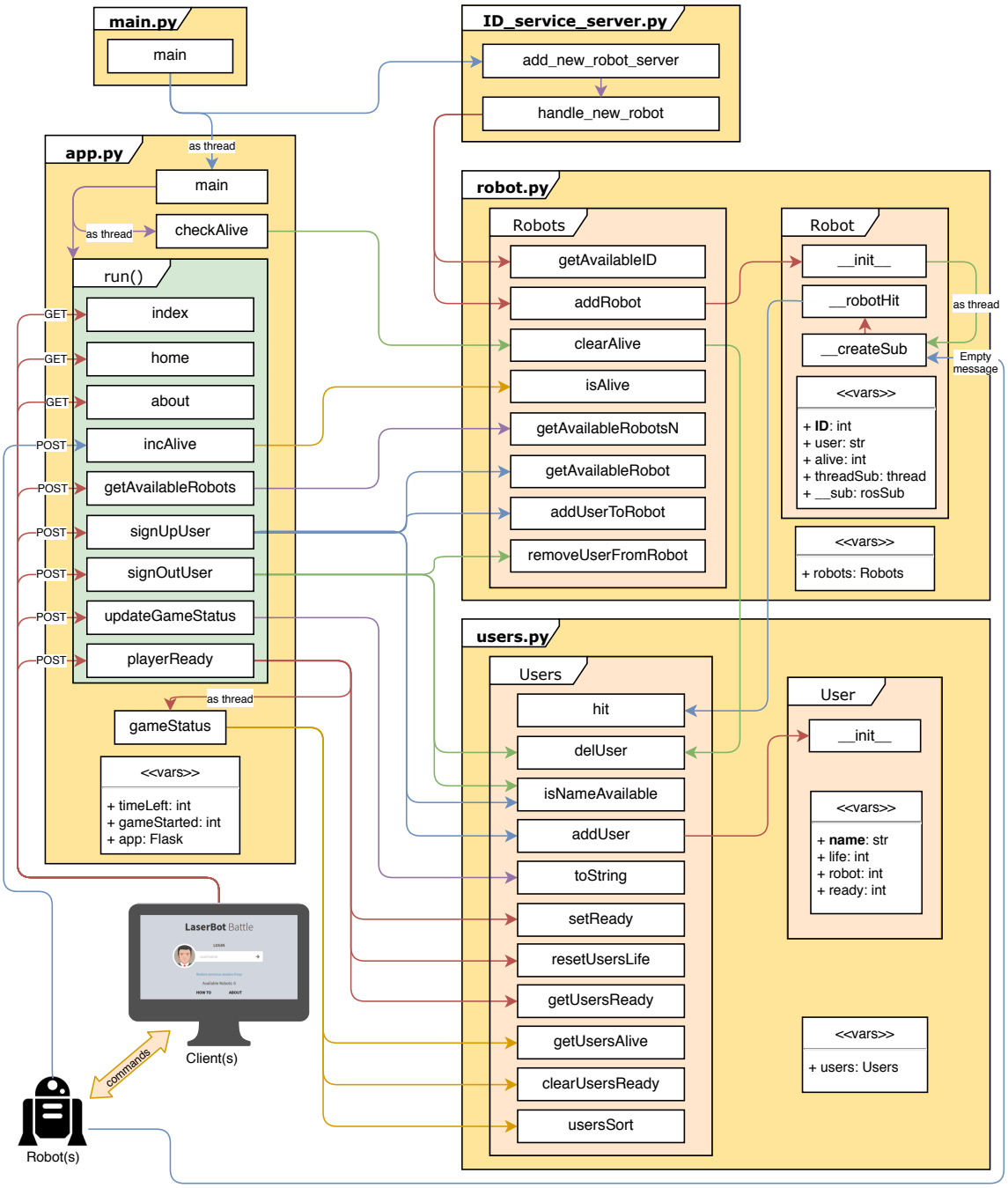

Figure 3: Web Server functions scheme

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 base version of Flask is installed in order to manage HTTP requests and tracing 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 section 3.1.1
- ID_service_server.py section 3.1.2
- app.py section 3.1.3
- robot.py section 3.1.5
- users.py section 3.1.4

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 3 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 3 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 3: 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 3 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, by invoking methods of the 'Robots' class (section 3.1.5.2), 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 by assigning to it the lowest available ID.
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 function never returns but let the server to 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 4: 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 keeps the value of the countdown before the game is started. It 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 is visible from the client by issuing 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 `"/"` (see for page description).
- Home: The home page in which the game is played, reachable at the address `"/home"` (see for page description).
- About: The about page that briefly describes the project and the developers, reachable at the address `"/about"` (see for page description)

Three more webpages are provided to manage errors 404, 405, 500.

```
#!/usr/bin/env python
from flask import Flask, render_template, request, json
from users import userDefault, users
from robot import robots
import ID_service_server
import threading
import time


timeLeft = 0
gameStarted = int(0)

app = Flask(__name__, static_folder='static', static_url_path='/static')

# Index Login Page
@app.route('/')
def index():
    return render_template('index.html')

# User home page
@app.route('/home')
def home():
    return render_template('home.html')

# Project about page
@app.route('/about')
def about():
    return render_template('about.html')

@app.errorhandler(404)
def page_not_found(e):
    return render_template('errors/404.html'), 404

@app.errorhandler(405)
def method_not_allowed(e):
    return render_template('errors/405.html'), 405

```

```
@app.errorhandler(500)
def internal_error(e):
    return render_template('errors/500.html'), 500
```

Listing 5: 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 address 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 6. All the commented lines allow to run the server on https (with TLS 1.2). This option is disabled by default because the provided certificate is not signed by an authenticated authority.

```
# main function:
def main():
    threadAlive = threading.Thread(target=checkAlive)
    # Launch checkAlive function as a separate thread
    threadAlive.start()

    '''
    CERTFILE = "src/laser_bot_battle/scripts/ssl/cert.pem"
    KEYFILE = "src/laser_bot_battle/scripts/ssl/key.pem"
    import ssl
    context = ssl.SSLContext(ssl.PROTOCOL_TLSv1_2)
    context.load_cert_chain(CERTFILE, KEYFILE)
    '''

    # Run flask web app
    app.run(debug=False, use_reloader = False, host='0.0.0.0')
    #app.run(debug=False, use_reloader = False, host='0.0.0.0', ssl_context=context)
```

Listing 6: portion of app.py - main function

3.1.3.2 signUpUser

This function manages the login request made by a user using a browser. It extracts 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 an `"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 and the ID of the robot to which the user has been associated. The function code can bee seen in Listing 7.

```
58 # signUpUser function:
59 #   add user to user list if name is available
60 @app.route('/signUpUser', methods=['POST'])
61 def signUpUser():
62     name = request.form['data']
63 
64     # if username is availabe :
65     if users.isNameAvailable(name) :
66 
67         # check first available robot id
68         robotN = robots.getAvailableRobot()
69         if robotN == -1 :
70             return json.dumps({'status':'NO_ROBOTS', 'user':name})
71 
72         # associate user to robot (check if fail)
73         if not robots.addUserToRobot(robotN, name) :
74             return json.dumps({'status':'ROBOT_UNAVAILABLE', 'robot':robotN})
75 
76         # add it to users list
77         users.addUser(name, robotN)
78         return json.dumps({'status':'OK', 'user':name, 'robot':robotN})
79 
80     else :
81         # else return UNAVAILABLE error
82         return json.dumps({'status':'UNAVAILABLE', 'user':name})
```

Listing 7: portion of app.py - signUpUser function

3.1.3.3 signOutUser

This function manages the sign out request of a 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 8.

```
85 # signOutUser function:
86 #   delete user from user list if present
87 @app.route('/signOutUser', methods=['POST'])
88 def signOutUser():
89     name = request.form['data']
90 
91     # if user is in users list :
92     if not users.isNameAvailable(name) :
93 
94         robots.removeUserFromRobot(name)
95 
96         # delete from list
97         if users.delUser(name) :
98             return json.dumps({'status':'OK', 'user':name})
99         else :
100            return json.dumps({'status':'FAILED', 'user':name})
101    else :
102        # else return UNREGISTERED error
```

```
103         return json.dumps({'status':'UNREGISTERED', 'user':name})
```

Listing 8: 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 9.

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

Listing 9: 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 10.

```
114 # getAvailableRobots function:
115 #   return number of available robots (json format)
116 @app.route('/getAvailableRobots', methods=['POST'])
117 def getAvailableRobots():
118     return json.dumps({'status':'OK', 'availableR':robots.getAvailableRobotsN()})
```

Listing 10: portion of app.py - getAvailableRobots function

3.1.3.6 gameStatus

This function manages the status of the game. It is called by a separate thread and runs in parallel with the application. Once called, it sets `gameStarted` to 1 and starts decreasing the `timeLeft` variable once every second. When it reaches 0 the game can start and `gameStarted` is set to 2. The thread checks 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 11.

```
121 # gameStatus function:
122 #   start countdown, start game and check end of game
123 def gameStatus():
124     global timeLeft
125     global gameStarted
126     gameStarted = 1
127     print "countdown started"
```

```
128
129     # countdown to game start
130     while timeLeft > 0:
131         time.sleep(1)
132         timeLeft -= 1
133         print timeLeft,
134
135     print "Starting game!"
136     gameStarted = 2
137
138     # check for game to end (only 1 player alive)
139     while users.getUsersAlive() > 1 :
140         users.usersSort()
141         time.sleep(0.5)
142
143     # game finished
144     print "Game finished"
145     gameStarted = 0
146     users.clearUsersReady()
147
148     return
```

Listing 11: portion of app.py - gameStatus function

3.1.3.7 playerReady

This function is used to set the ready status (to '1') whenever a requesting user issues a POST request (by clicking on the ready button). If the game is already started the ready button is disabled (see section 3.1.3.4, lines 380-389), such that the user cannot modify its ready status when it is already playing. Anyway, additional checks are made to guarantee the right behavior even if for some reasons the user could send the POST request when the game is in act and in this case the function will return a `"STARTED"` status. If at least 2 users are ready : 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 12.

```
151 # playerReady function:
152 #   update player ready status
153 @app.route('/playerReady', methods=['POST'])
154 def playerReady():
155     global timeLeft
156     global gameStarted
157     name = request.form['user']
158     ready = request.form['ready']
159
160     # Game already started
161     if gameStarted == 2 :
162         return json.dumps({'status':'STARTED'})
163
164     #set user ready
165     if users.setReady(name, ready) :
166         # If more than 2 players are ready
167         if users.getUsersReady() > 1 :
168             users.resetUsersLife()
169             timeLeft = 15
```

```
170             # Launch countdown to game start as a new thread
171             threadGameStatus = threading.Thread(target=gameStatus)
172             threadGameStatus.start()
173 
174         return json.dumps({'status':'OK','user':name})
175     else :
176         return json.dumps({'status':'ERROR','user':name})
```

Listing 12: portion of app.py - playerReady function

3.1.3.8 incAlive

This function is called from raspberry which sends a POST request to the `/incAlive` address. It gets the robot ID sent by the requester and calls the `isAlive` function for that robot, increasing an internal variable used by the webserver to know if the robot is alive. The code can bee seen in Listing 13.

```
179 # incAlive function:
180 #   increase alive value for robot that call this function
181 @app.route('/incAlive', methods=['POST'])
182 def incAlive():
183     robotID = int(request.form['ID'])
184 
185     if robotID != "":
186         if robots.isAlive(robotID):
187             return json.dumps({'status':'OK'})
188 
189     return json.dumps({'status':'ERROR'})
```

Listing 13: portion of app.py - incAlive function

3.1.3.9 checkAlive

This function is called by the `app.main` as a separate thread that runs 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 acting on the whole robots list. It is used to check the alive status of robots and to disconnect those robots which for some reasons become powered-off or unreachable (these are the robots which have not called the `incAlive` function at least once within 2 seconds). The code can bee seen in Listing 14.

```
192 # checkAlive function:
193 #   check every 2 sec if all connected robots are still alive, if not delete them
194 def checkAlive():
195     while True:
196         time.sleep(2)
197         robots.clearAlive()
```

Listing 14: portion of app.py - checkAlive function

3.1.4 users.py

This file contains all the classes necessary to the server to keep track 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 15: Users.py - User class source code

3.1.4.2 Users class

It is a list of "User" (section 3.1.4.1) with unique user name. 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 searches a user within the list and returns "True" if the user has been found, "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.
- `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()` : Returns the number of users that are ready (those for which Ready attribute is set at "1").
- `getUsersAlive()` : Returns the number of users that are still alive during a battle (life > 0 and ready $= 1$).
- `clearUsersReady()` : Restores the Ready status of all users to "not ready" (ready = "0").
- `resetUsersLife()` : Restores the remaining life of all users to the maximum value (life = "100").
- `hit(name)` : Implements the hit action lowering the user's life during a battle. Input parameter "name" specifies the username of the hit user.
- `userSort()` : Sorts the users involved in a battle by the life attribute in a decreasing way (the last one is the most damaged). Not playing users are placed at the bottom of the list.

The unique username is seen as an 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     # set ready status of player
54     def setReady(self, name, ready):
55         for u in self.users:
56             if u.name == name :
57                 u.ready = int(ready)
58                 return True
59         return False
60 
61     # get number of players ready to start
62     def getUsersReady(self):
```

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

Listing 16: Users.py - Users class source code

Finally, an instantiation of the users list is done (Listing 17). This is crucial since this instantiated list is global for all python scripts importing this class. An example of such import can be seen at section 3.1.3 - line 3.

```
112 users = Users()
```

Listing 17: 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, it invokes the robotHit() callback function (described above).

```
6  # Robot class
7  class Robot:
8      ID = 0
9      user = ""
10     alive = 1
11
12     def __init__(self, id, user=None):
13         self.ID = id
14         self.alive = 1
15         if user != None :
16             self.user = user
17         self.threadSub = threading.Thread(target=self.__createSub)
18
19     def __robotHit(self, msg):
20         print "Robot"+str(self.ID)+" has been hit"
21         users.hit(self.user)
22
23     def __createSub(self):
24         #print "creating node: ", "/Robot"+str(self.ID)+"_subscriber"
25         #rospy.init_node("Robot"+str(self.ID)+"_subscriber")
26
27         __sub = rospy.Subscriber("/Robot"+str(self.ID)+"/response", Empty, self.__robotHit)
28         print 'Node initialized'
29         rospy.spin()        #wait
```

Listing 18: robot.py - Robot class source code

3.1.5.2 Robots class

It is a list of "Robot" (Listing 18). All the functions needed for the list manipulation are defined in this class. A brief explanation is reported below:

- `addRobot(id)` : Adds a robot to the list. Input parameter "id" specifies the robot ID to be added.
- `getAvailableRobot()` : Returns 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()` : Returns the number of robots available (those that are registered but not assigned to a user yet).
- `getAvailableID()` : Returns the first available ID to be assigned to a new registered robot.
- `addUserToRobot(id, name)` : Associates 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)` : disassociates 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)` : Increments 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.
- `clearAlive()` : Checks 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".

```
32 # list of robots class
33 class Robots:
34     robots = []
35
36
37     # add new robot to list (and keep it sorted by ID)
38     def addRobot(self, id):
39         self.robots.append( Robot(id) )
40         self.robots.sort(key=lambda x: x.ID)
41         print "Added robot with ID", id
42
43
44     # get first available robot (ID) from robots list
45     def getAvailableRobot(self):
46         for r in self.robots:
47         #print "id: ", r.ID , "user: ", r.user
48             if r.user == "" :
49                 return r.ID
50         return -1
```

```
51
52      # get number of available robots
53      def getAvailableRobotsN(self):
54          num = 0
55          for r in self.robots:
56          #print "id: ", r.ID , "user: ", r.user
57              if r.user == "" :
58                  num += 1
59          return num
60
61      # get first unused ID starting from 0
62      def getAvailableID(self):
63          ID = 0
64          for r in self.robots:
65              if r.ID == ID :
66                  ID += 1
67              else :
68                  break
69          return ID
70
71      # associate user name to robot
72      def addUserToRobot(self, id, name):
73          for r in self.robots:
74              if r.ID == id and r.user == "" :
75                  r.user = name
76                  if not r.threadSub.isAlive():
77                      r.threadSub.start()
78                  return True
79          return False
80
81      # de-associate user from robot
82      def removeUserFromRobot(self, name):
83          for r in self.robots:
84              if r.user == name :
85                  r.user = ""
86                  return True
87          return False
88
89      # signat that robot is alive
90      def isAlive(self, id):
91          for r in self.robots:
92              if r.ID == id :
93                  r.alive += 1
94                  return True
95          return False
96
97      # clear alive status of all robots
98      def clearAlive(self):
99          for r in self.robots:
100             #print "Checking robot ", r.ID, " alive is ", r.alive
101             if r.alive == 0 :
102                 # If not delete robot and user
103                 print "Robot ", r.ID, " is dead."
104                 if r.user != "":
105                     print " Player ", r.user, " disconnected"
106                     users.delUser(r.user)
107                 self.robots.remove(r)
108             else:
109                 r.alive = 0
```

Listing 19: robot.py - Robots class source code

Finally, an instantiation of the robots list is done (Listing 20). This is crucial since this instantiated list is global and shared by all those files ".py" that import it.

```
111 robots = Robots()
```

Listing 20: robot.py - robots list instantiation

3.2 Web Application

In this section the **Web Client** part is documented. Mainly speaking, it is the web application that allows the users to log-in, command the robot and check battle status (see Figure 4). The web page structure and style have been described by HTML 5.0 and CSS. Interactive events and user's data have been handled using JavaScript. The JS code has been either written within HTML documents under the section "script" or written in a separate file and imported into HTML. In the following subsection a general explanation of the main elements of each page is reported. It is suggested to consult the source code for further information.

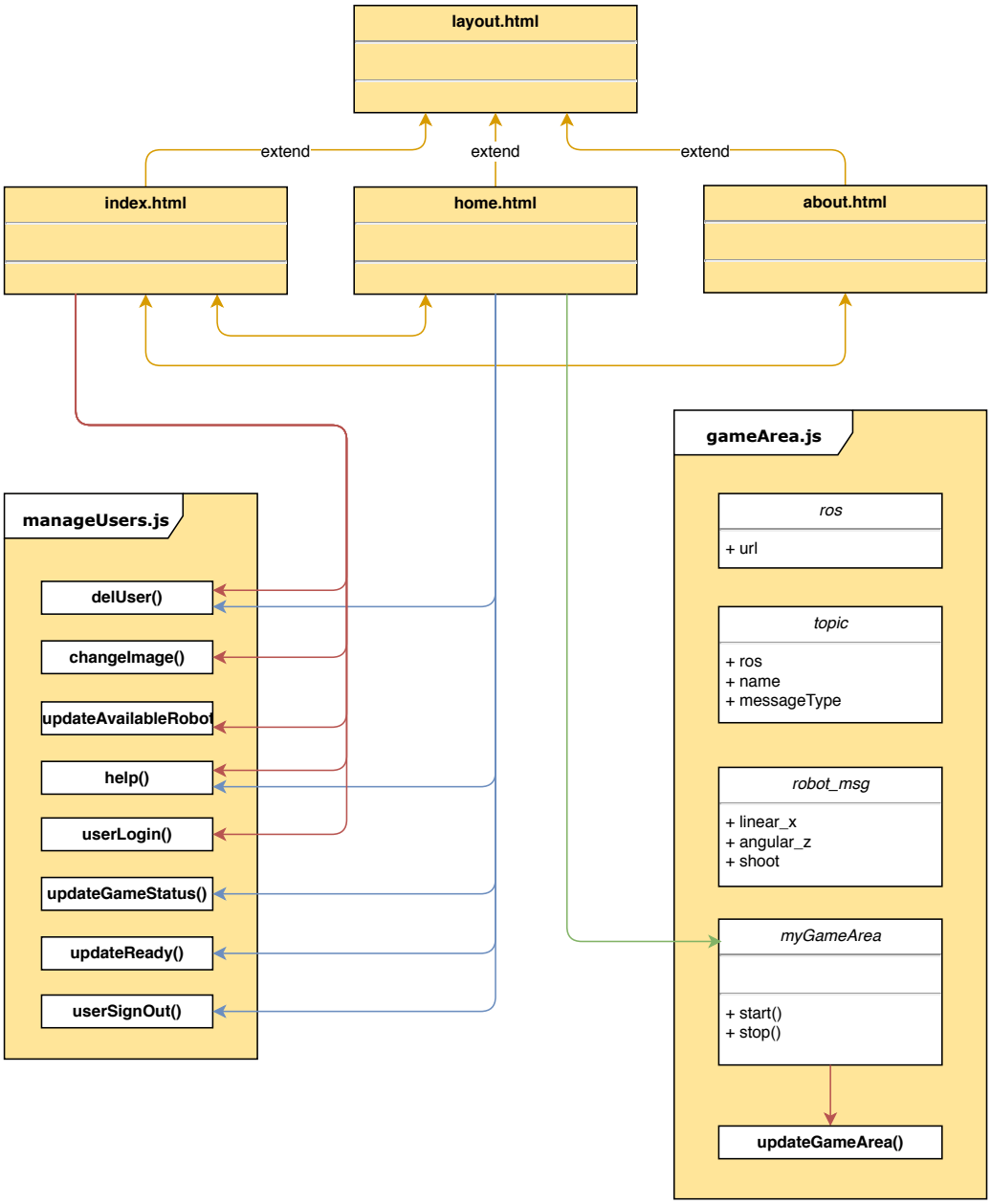

Figure 4: Web Client functions scheme

3.2.1 index.html

It is the main page to which the user is redirected after a HTTP request to <server> port 5000 is performed. From this page the user can login, restore previous sessions, consult the About page and the "How to" instructions. For these purposes, the page contains both JS and HTML code.
Main structure of the file is explained below, listing main elements and functionalities involved.

- **Ready event handler** : It is the javascript function which is called when the page is loaded (ready event). It controls whether the "user" structure has been previously stored in the localStorage. If yes, a suggestion to restore previous session is proposed to the user. If the user accepts, it is redirected to /home since it is ready to play (login has been already done), otherwise it is deleted from the list of users. Another way to restore previous session is by clicking on "Restore previous session if any". In this case a /home redirection is done. It will be the "ready event" handler within home.html that in this case checks if the user has been already registered (with a localStorage check). Moreover, it is important to point out that the data structures on which this JavaScript operates are local (i.e. Listing 21 line 35 - "user" structure). In fact, each user has its own copy containing its own information.

```
10  /* when this page is loaded */
11  $(document).ready(function(){
12
13    /* Check if user was previously logged in.
14      When the user logged in, a stringify version
15      of "user" structure is saved in the local storage. */
16    if (localStorage.getItem("user")!== null){
17
18      /* Take from the local storage the user structure */
19      user = JSON.parse(localStorage.getItem("user"));
20
21      /* Signal that a previous login has been done */
22      swal({
23        title: "You previously logged as \""+ user.name + "\".\n",
24        text: "Do you want to restore your previous session?",
25        icon: "info",
26        buttons: true,
27        dangerMode: true,
28      })
29      .then(function(isConfirm) {
30        if (isConfirm) {
31          /* Yes: restore previous session */
32          location.href = "/home";
33        } else {
34          /* No: delete current user (function delUser() described above) */
35          delUser(user.name);
36        }
37      });
38    }
```

Listing 21: index.html - ready event handler

- **Login**: The login functionality provides to register the user once that a new username has been chosen and robots are available on the network. For this purpose the userLogin()

function is called.

```
85              <button class="btn" onclick="userLogin()" title="Start!">
```

Listing 22: index.html - User login

- **How To** : An "How To" alert can be opened in order to retrieve info about game instructions.
- **"Robot available" info** : A periodic call of the function updateAvailableRobots() is created, with a period set to UPDATE_INTERVAL ms. In this way a POST request to the server asks for the number of robots available in the network. When the response is get (section 3.1.3.5 - server response), the information is written into the page. This is an important information since if no robots are available, user can not login.

```
50      /* periodically launch function to update logged users list */
51      robotUpdate = setInterval(function(){updateAvailableRobots();}, UPDATE_INTERVAL);
```

Listing 23: index.html - Robot available info

The page view is reported below:

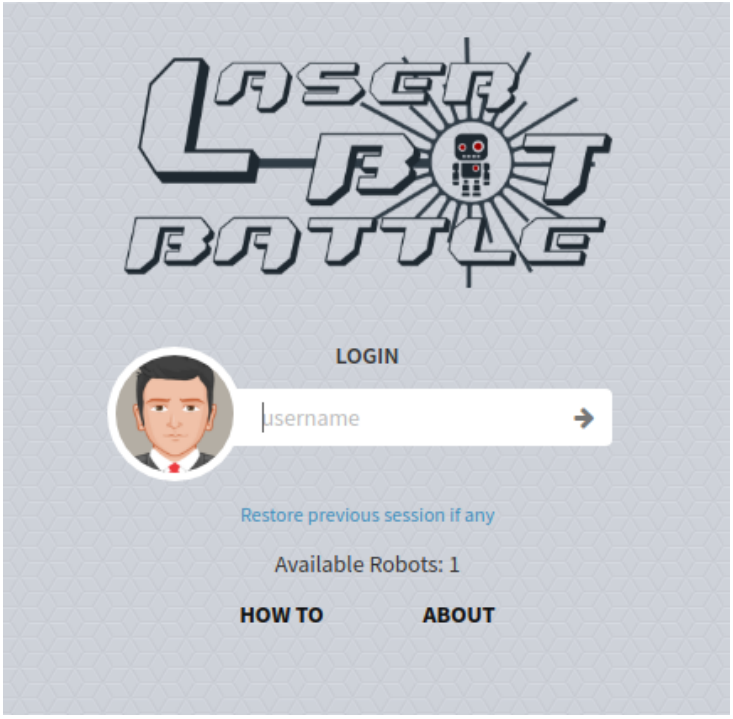

Figure 5: index.html - Browser view

3.2.2 about.html

This page contains brief description of the application and the team that realized it. The page view is reported below:

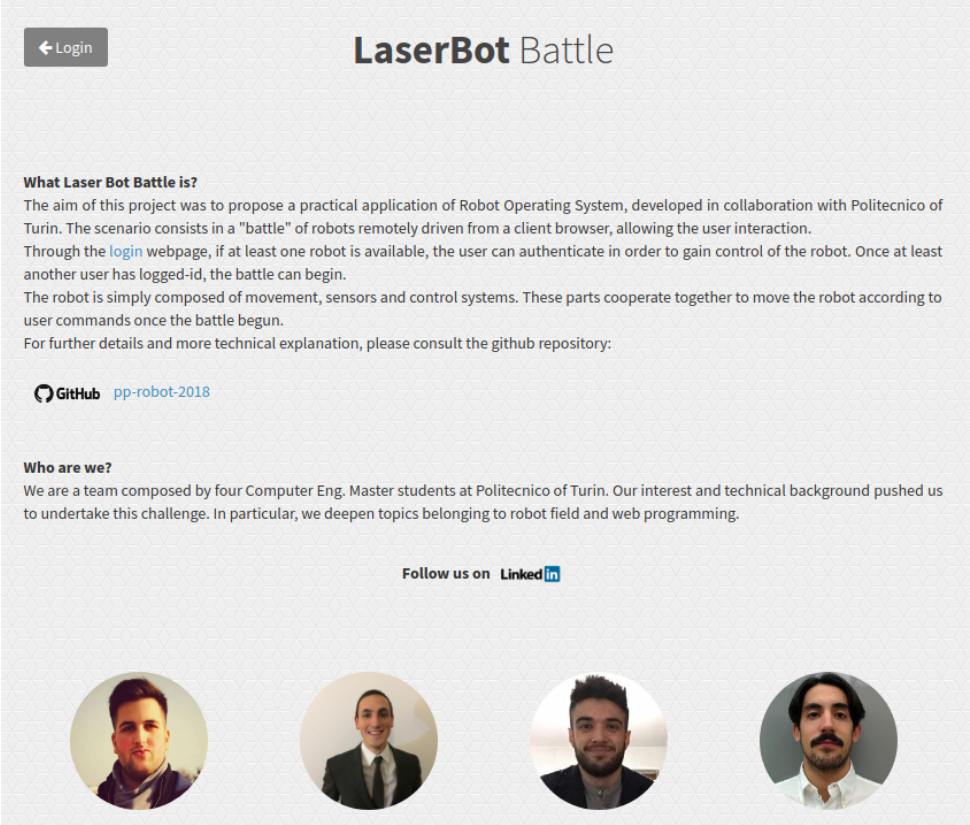

Figure 6: about.html - Browser view

3.2.3 home.html

The page view is reported below:

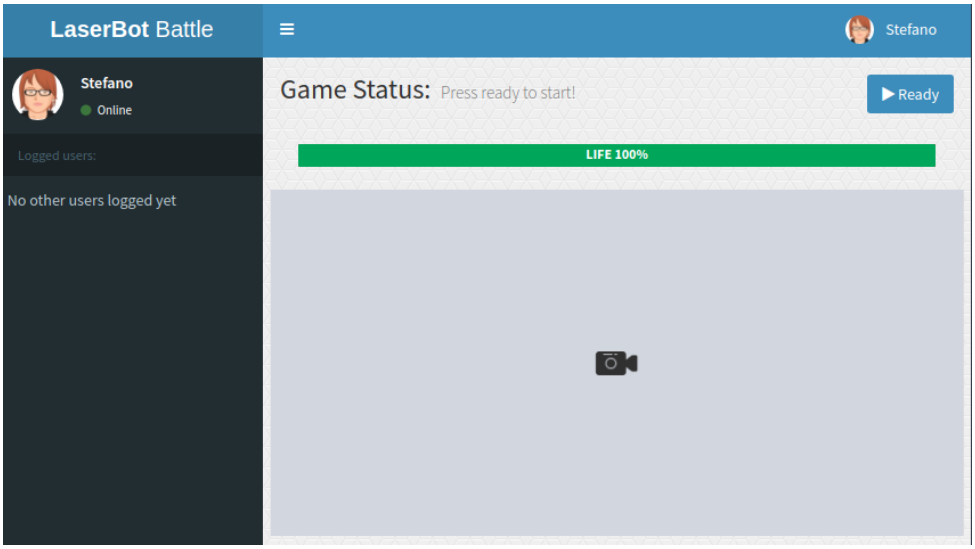

Figure 7: home.html - Browser view

This is the core page of the application. It provides to the user the possibility of sign out, initiate a battle, join a battle, command the robot during a battle, check information related to other connected users and check battle ranking at the end of the game.

- **User validation** : A handler for the "ready page" event checks if the redirection to this page is legal. A check on the localStorage controls that the user has already logged in. If not it is brought back to the Index page (section 3.2.1).
- **Get user information** : This functionality shows on the page some info related to other connected users (lifebar, status and username). Through a POST request to the server all these info are requested. When the response from server is get, these info are parsed and written into the page (Listing 11 - server response).

```
42          }, 2100);
43          return;
44        }
45
46        console.log("Logged as \"" + user.name + "\"");
47
48        /* Spawn username in all username classes */
49        $('.username').html(user.name);
50        /* Spawn robot number */
51        $('.robotN').html("Robot " + user.robotN);
52        /* Spawn date */
53        $('.date').html("Logged in: " + user.date);
54        /* Spawn user image in all userImage classes*/
55        $('.userImage img').attr("src", "static/img/avatar" + user.imgIndex + ".png");
```

Listing 24: home.html - Other users information

- All other functionalities are implemented within the file "gameArea.js" (section 3.2.4).

3.2.3.1 Streaming video

This part is still under development.

3.2.4 gameArea.js

This script plays the role of intermediary between the user and the robot (in particular the raspberry). In fact, commands from user are interpreted and packed into a message, which in turn is sent on the ROS topic related to the robot associated to the user itself. The following subsections describe the main parts of this script.

3.2.4.1 ROS client setup

This part handles the connection of the client to the ROS network. For this purpose all the required JavaScript files for the application (EventEmitter2 and roslibjs) are imported within home.html (section 3.2.3).
A ROS node object is created in order for the client to communicate with a rosbridge server. The server communicates its "domain-name=local" and "host-name=laser_bot_master" such that it is visible from all the rosbridge clients connected to its same local network. Thus,

"laser_bot_master.local" is set as server address on this client node.(Listing 25 line 11)
A topic is then created (Listing 25 lines 14-19). Messages of Robot_msg type traffic on this topic from the client to the rosbridge server. The structure of Robot_msg type can be seen in Listing 25 lines 23-27.

```
8  /* Connecting to ROS */
9
10 var ros = new ROSLIB.Ros({
11   url : 'ws://laser_bot_master.local:9090'
12 });
13
14 /* Topic definition */
15 var topic = new ROSLIB.Topic({
16   ros : ros,
17   name : '',
18   messageType : 'laser_bot_battle/Robot_msg'
19 });
20
21 /* Default message to be sent on the topic
22 It is modified later accordingly before being sent */
23 var robot_msg = new ROSLIB.Message({
24   linear_x :  0,
25   angular_z : 0,
26   shoot :     false
27 });
```

Listing 25: gameArea.js - ROS client setup

3.2.4.2 Capturing user commands

An object "myGameArea" is created. It owns two functions (start and stop).

- `start` : It calls at regular intervals the `updateGameArea()` (Listing 26 lines 72-121) function that creates the robot_msg to be sent on the topic according to user command events (Listing 26 lines 50-58). This function is invoked once the battle can begin. At regular intervals, the status is sent by the server as HTTP responses (section 3.2.5.5) at regular interval and get from manageUsers.js (section 3.2.5) in order to operate regarding on the battle status communicated by the server.
- `stop` : disables the sensitivity of the page to the events (Listing 26 lines 60-68).

The composed message is then published on the topic (Listing 26 lines 117-121).

```
43  *- start(): launch updateGameArea() each UPDATE_AREA ms
44  *            enabling event sensitivity for keyDown and keypress
45  *- stop(): stop the calling to updateGameArea() and remove
46  *          the sensitivity to the events keyDown and keyUp
47  */
48  var myGameArea = {
49
50   start : function() {
51     /* Launch updateGameArea() function every UPDATE_AREA ms */
52     this.interval = setInterval(updateGameArea, UPDATE_AREA);
53
```

```
54      /* Associating the above defined keyUp/Down() functions to the keydown/up events */
55      window.addEventListener('keydown', keyDown)
56      window.addEventListener('keyup', keyUp)
57
58    },
59
60    stop : function() {
61      /* Stop updateGameArea update */
62      window.clearInterval(this.interval);
63
64      /* Dis-associating the above defined keyUp/Down() functions to the keydown/up events */
65      window.removeEventListener('keydown', keyDown);
66      window.removeEventListener('keyup', keyUp);
67
68    }
69
70  };
71
72  /* This function is invoked every UPDATE_AREA ms.
73   *It modifies the robot_msg to be sent on the topic according
74   *to the commands received by the user (key pressures)
75   */
76   function updateGameArea() {
77    robot_msg.linear_x = 0;
78    robot_msg.angular_z = 0;
79    robot_msg.shoot = false;
80
81    /* Saving on temporal variables the intended action */
82    var left  = myGameArea.keys && ( myGameArea.keys[37] || myGameArea.keys[65]);
83    var up    = myGameArea.keys && ( myGameArea.keys[38] || myGameArea.keys[87]);
84    var right = myGameArea.keys && ( myGameArea.keys[39] || myGameArea.keys[68]);
85    var down  = myGameArea.keys && ( myGameArea.keys[40] || myGameArea.keys[83]);
86    var shoot = myGameArea.keys && ( myGameArea.keys[13] || myGameArea.keys[32]);
87
88
89    /* robot_msg to be sent modified here: */
90
91    /* LEFT */
92    if ( left && (! right) ) {
93      robot_msg.angular_z = -1;
94    }
95
96    /* RIGHT */
97    else if ( right && (! left) ) {
98      robot_msg.angular_z = 1;
99    }
100
101   /* UP */
102   if ( up && (! down) ) {
103     robot_msg.linear_x = 1;
104   }
105
106   /* DOWN */
107   else if ( down && (! up) ) {
108     robot_msg.linear_x = -1;
109   }
110
111   /* SHOOT */
112   if ( shoot ) {
113     robot_msg.shoot = true;
```

```
114   }
115
116   /* publish message only if robot is intended to act (move or shoot) */
117   if (! wasStill){
118     /* publishing the robot_msg modified above */
119     topic.publish(robot_msg);
120     console.log(JSON.stringify(robot_msg));
121   }
122
123   /* if robot is still, next time don't send message (if still again) */
124   wasStill = ( robot_msg.linear_x == 0 && robot_msg.angular_z == 0 && !robot_msg.shoot );
125 }
```

Listing 26: gameArea.js - User commands published on topic

3.2.5 manageUsers.js

This script defines structures and functions implementing the client-server communication, client side. From this file, client sends to the server the user local intents (login and sign out actions) or requests from it the game information in order to keep the page updated.

A 'user' object stores local user information, as defined below:

```
23 /* user structure:
24   - name : name chosen during login
25   - life : [0, 100]. 100 (full life), 0 (dead)
26   - robotN : associated robot ID
27   - imgIndex : index of the selected image during login
28   - date : login date
29   - ready : 0 (not ready to fight), 1 (ready to fight)
30 */
31 var user = {  name: "",
32 life: 100,
33 robotN: undefined,
34 imgIndex: 0,
35 date: "",
36 ready: 0,  };
```

Listing 27: manageUsers.js - user object

The content of this instance is an instantaneous shot of the current user situation. Comments in the source code (as reported in Listing 27) explain what does each attribute of the class represents.
Next subsections report an explanation for the functions defined in this file.

3.2.5.1 delUser

This function removes user info from both the localStorage and the 'users' list (residing on the server). When this function is invoked, it sends to the server a POST request to the /signOutUser address containing the username of the user that is going to sign out. If the response is "success", the user is removed from the localStorage. An error handling is also performed.

```
function delUser(name){
 /* An ajax post request is sent to the server.
   request: delete user named "name".
   answer:
     - success : server answers with an error or with success
     - error : due to connection problems an error answer is received
     */
 $.ajax({
   url: '/signOutUser',
   data: {'data': name},
   type: 'POST',
   success: function(response) {
     /* response from server is parsed and status is get */
     var status = JSON.parse(response).status;

     /* status "OK" : user has been deleted from server */
     if ( status == "OK" ){
       /* Signal to the user that signOut had success */
       console.log("User \"" + user.name + "\" removed");
       swal("Your session has been deleted!", "Redirecting to login page", "success");
     }
     /* status "UNREGISTERED" : user is not registered */
     else if ( status == "UNREGISTERED" ){
       /* Signal to the user that is not logged */
       swal("Server error", "The user \"" + user.name + "\" is not logged.", "error");
     }
     /* status "FAILED" : for some reasons server can not remove it */
     else if ( status == "FAILED" ){
       /* Signal to the user that it is not logged in */
       swal("Server error", "The user \"" + user.name + "\" can't be removed.", "error");
       return;
     }
     else {
       /* Signal that there has been some error */
       swal("Unknown error", "response: " + response, "error");
       return;
     }

     /* Stop users list update */
     if ( typeof(userUpdate) != "undefined" ){
       window.clearInterval(userUpdate)
     }

     /* remove user from the localStorage */
     localStorage.removeItem('user');
     user.name = "";

     /* redirect to login page. Ready for a new login*/
     setTimeout(function(){
       location.href = "/";
     }, 1500);


   },
   /* errors due to connection problems */
   error: function (xhr, ajaxOptions, thrownError) {
     console.error("ERROR " + xhr.status + ": " + thrownError);
   }
 });
}
```

Listing 28: manageUsers.js - user class

3.2.5.2 userSignOut

This function is invoked each time the user intends to sign out from the application. Basically it only invokes the function `delUser()` (section 3.2.5.1) if the user confirms to sign out.

```
57  function userSignOut() {
58   /* Sign out confirm alert */
59   swal({
60     title: "Are you sure you want to Sign Out?",
61     text: "Once deleted, you will not be able to recover this session",
62     icon: "warning",
63     buttons: true,
64     dangerMode: true
65   })
66   .then(function(isConfirm) {
67     if (isConfirm) {
68       /* call delUser() (defined above) if signOut is confirmed */
69       delUser(user.name);
70     } else {
71       swal("Your session is safe");
72     }
73   });
74
75 }
```

Listing 29: manageUsers.js - userSignOut

3.2.5.3 userLogin

When this function is invoked, a POST request is sent to the server. In case of positive response the user is added to localStorage and the "user" structure (Listing 27) is updated. An error handling is implemented.

```
155 function userLogin() {
156  /* Check if browser support local storage.
157    localStorage is used to store info about user after login */
158  if (typeof(Storage) !== "undefined") {
159
160    /* Take username value from the "username" form field of the page */
161    var tempName = document.getElementById('username').value;
162    /* remove spaces before and after string */
163    tempName = tempName.replace(/^\s+|\s+$/g, '');
164
165    /* Check if username format is valid */
166    if (tempName.match(/^[A-z0-9]+$/) == null || tempName.length > 16){
167      swal({
168        title: "Your username is not in a valid format!",
169        text: "Username can not exceed 16 char length and can not "
170        + "contain spaces and special characters.",
171        icon: "warning",
```

```
172       });
173     }
174     else {
175       /* Here if username specified is correctly formatted */
176       /* Tell python to add user to users list.
177         A post request is sent to the server (data sent: username) */
178       $.ajax({
179         url: '/signUpUser',
180         data: {'data':tempName},
181         type: 'POST',
182         dataType: "text",
183         success: function(response) {
184           /* Parse is performed from server answer */
185           var res = JSON.parse(response);
186           var status = res.status
187
188           /* Server has registered the new user */
189           if ( status == "OK" ){
190             /* Save on the localStorage the user data */
191             user.name = res.user;
192             user.robotN = res.robot;
193             user.date = (new Date()).toString().split("GMT")[0];
194             localStorage.setItem('user', JSON.stringify(user));
195             console.log("Logging in as: \"" + user.name + "\"");
196             /* Redirect to home page - login successful*/
197             location.href = "/home";
198           }
199           /* If no robots are available, login can not be done */
200           else if ( status == "NO_ROBOTS" ){
201             swal("Warning", "There are no robots available at the time!", "warning");
202           }
203           /* Robot that was associated to the user is not available */
204           else if ( status == "ROBOT_UNAVAILABLE" ){
205             swal("Server error", "The robot n. " + user.robotN + " is unavailable!",
    "error");
206           }
207           /* Username chosen is already used */
208           else if ( status == "UNAVAILABLE" ){
209             swal({
210               title: "The username \"" + tempName + "\" is already taken!",
211               text: "Please choose another one.",
212               icon: "info",
213             });
214           }
215           else {
216             swal("Unknown error", "response: " + response, "error");
217             return;
218           }
219         },
220         /* communication errors */
221         error: function (xhr, ajaxOptions, thrownError) {
222           console.error("ERROR " + xhr.status + ": " + thrownError);
223         }
224       });
225
226     }
227   }
228   else {
229     /* Sorry! No Web Storage support... */
230     swal({
```

```
231       title: "This browser does not support local storage!",
232       text: "The website can not work without local storage (Web storage)"
233       + " support. This functionality will be added later on (maybe)",
234       icon: "error",
235     });
236   }
237 }
```

Listing 30: manageUsers.js - userLogin

3.2.5.4 changeImage

During login phase, from the index.html (section 3.2.1), the user can select the avatar by clicking on it. This functionality is implemented by this function. In fact, it updates "user" class instance (Listing 27), imgIndex attribute, with the index related to the selected avatar and at the same time updates the avatar to be shown on the page.

```
248 function changeImage() {
249  /* each time a click on the avatar arises, this function is invoked.
250    It changes the avatar to be associated to the user when login will be done */
251  user.imgIndex = (user.imgIndex +1) % 5;
252  document.getElementById("userImage").src = "static/img/avatar" + user.imgIndex +".png";
253 }
```

Listing 31: manageUsers.js - changeImage

3.2.5.5 updateGameStatus

This is a crucial function which is invoked at regular intervals such that having the local view always up to date. A POST request asks to the server all information related to the current status of the session. In particular: username, life and status of each other logged user are shown; user personal lifebar is shown; info messages as final battle ranking or countdown before battle begins are shown. The commented code of this function is reported below:

```
263 function updateGameStatus() {
264
265  /* A post request to the server to obtain game status info */
266  $.ajax({
267    url: '/updateGameStatus',
268    type: 'POST',
269    data: {'data':'data'},
270    /* server answer with status information
271      Response from server has the following structure:
272      - 'status'
273      - 'users'
274      - 'game'
275      - 'timeLeft'
276    */
277    success: function(response) {
278      /* response from server is parsed and status attribute is picked up */
279      var status = JSON.parse(response).status
280
281      /* if no error arisen */
```

```
282       if ( status == "OK" ){
283         updateGameError = 0;
284
285         var res = JSON.parse(response)
286
287         /* assume my robot is disconnected */
288         var robotDead = true;
289
290         /* Taking the game status:
291           0 - none game started
292           1 - countdown - waiting for the game to start (allowing other users to join the
    game)
293           2 - game on
294         */
295         var game = res.game;
296
297         /* If user list is not empty */
298         if (res.users !== "[]"){
299           /* Inject in the page the received info from the server */
300
301           /* clean html to be injected */
302           var userList = "";
303           var userLife = "";
304
305           /* get all users info */
306           var users = JSON.parse(res.users);
307           var usersN = users.length;
308
309           for (var i = 0; i < usersN; i++) {
310
311             /* UPDATE USER STATUS */
312             if (users[i].name != user.name){  // just show info related to other users
313               userList += '<div class="user-list-div"><li><a>'
314               /* user status icon */
315               if (users[i].ready == 1){
316                 /* Users involved in a battle (ready = 1)*/
317                 if (users[i].life > 0 || game != 2)
318                   /* */
319                   userList += '<i class="fa fa-play-circle" style="color: #00a65a;"></i> '
320                 else
321                   /* user is dead & game is not ended yet - X */
322                   userList += '<i class="fa fa-times-circle" style="color: #d33724;"></i> '
323               }
324               /* users only logged in, not involved in a battle (ready = 0) */
325               else
326                 userList += '<i class="fa fa-pause-circle" style="color: #f4bc42;"></i> '
327
328               /* UPDATE USERNAME */
329               userList += '<b> '+ users[i].name + '</b></a>'
330               /* life percentage badge */
331               + ' <span class="badge user-list-badge" >'
332               + users[i].life + '%</span> </li>'
333
334               /* UPDATE LIFEBAR */
335               + ' <div class="progress progress-xs user-list-bar">'
336               + ' <div class="progress-bar progress-bar-success" style="width: '
337               + users[i].life + '%"></div></div></div><hr/>'
338             }
339             else{
340               /* If I am in the list, the robot is not disconnected */
```

```
341               user.life = users[i].life;
342               user.ready = users[i].ready;
343               robotDead = false;
344
345             }
346           }
347           /* If no other players logged */
348           if ( usersN < 2 )
349             userList = '<div class="user-list-div"><li><a>No other users logged
      yet</a></div>';
350
351           /* SHOW PERSONAL LIFEBAR */
352           /* current user life percentage badge */
353           userLife += '<div class="user-life-div">'
354           + '<span class="badge user-life-badge">LIFE ' + user.life + '%</span>'
355           /* current user life progress bar */
356           + '<div class="progress user-life-bar">'
357           + '<div class="progress-bar progress-bar-success" role="progressbar"'
358           + 'aria-valuenow="20" aria-valuemin="0" aria-valuemax="100" style="width: '
359           + user.life + '%;"></div></div></div>';
360
361           /* Append user list and user life to the page*/
362           document.getElementById('userList').innerHTML = userList;
363           document.getElementById('userLife').innerHTML = userLife;
364         }
365         /* check if robot is still connected */
366         if (robotDead) {
367           /* For some reasons robot has been disconnected */
368           swal("OPS", "Connection to robot lost", "error");
369
370           /* delete stored variables */
371           localStorage.removeItem('user');
372           user.name = "";
373
374           /* redirect to login page */
375           setTimeout(function(){
376             location.href = "/";
377           }, 2500);
378         }
379
380         /* change ready button color (ready = 1 - ready for the battle)*/
381         var button = document.getElementById("ready-btn");
382         if ( user.ready == 0 ) {
383           button.style.backgroundColor = "#3c8dbc";
384           button.innerHTML = '<i class="fa fa-play"></i> Ready';
385         }
386         else {
387           button.style.backgroundColor = "#00a65a";
388           button.innerHTML = 'Ready!';
389         }
390
391         /* Update only if:
392           - previous status is different from the current one (something has to be updated)
393           - game status is 1 (battle begin is waiting for countdown to finish)
394             because countdown must be updated */
395         if (prevGame != game || game != 2){
396           /* current state becomes old state */
397           prevGame = game;
398
399           /* *** GAME STOPPED *** */
```

```
400           if ( game == 0 ){
401             /* re enable ready button */
402             $('#ready-btn').prop('disabled', false);
403             /* If first game of user */
404             if (user.ready == 0 && !played ){
405               /* User can begin the countdown to start the battle */
406               document.getElementById('game-status').innerHTML = 'Press ready to start!';
407             }
408             /* *** GAME FINISHED *** */
409             else if (played){
410               /* battle is ended up. Since the user partecipated to it, ranking is
411               shown and settings are restored to be ready for a new battle. */
412               played = false;
413               document.getElementById('game-status').innerHTML = 'Game over. Press ready to
    start again.'
414               /* Battle is ended up. Disable events sensitivity */
415               myGameArea.stop();
416               /* Show ranking of the battle */
417               var ranking = "Ranking:";
418               var position = 1;
419
420               /* First user is the winner */
421               ranking += "\n 1 - " + users[0].name;
422
423               for (var i = 1; i < usersN; i++) {
424                 /*This user was playing and is dead*/
425                 if(users[i].life == 0)
426                   ranking += "\n" + (i+1) +" - " + users[i].name;
427
428                 /*This is my position*/
429                 if (users[i].name == user.name)
430                   position = i+1;
431               }
432               /* Alert for battle end signaling arrival position*/
433               /* game ending message */
434               if (user.life > 0 && position == 1 ){
435                 /* WIN */
436                 swal("YOU WIN!", ranking);
437               }
438               else {
439                 /* LOST */
440                 swal("YOU LOST!\nYour arriving position is " + position, ranking);
441               }
442
443             }
444             /* I am the only player ready */
445             else {
446               document.getElementById('game-status').innerHTML = 'Waiting for other
    players...'
447             }
448           }
449           /* *** COUNTDOWN *** */
450           else if ( game == 1 ){
451             /* disable ready button since countdown is already started */
452             if(user.ready == 1)
453               $('#ready-btn').prop('disabled', true);
454             /* print countdown into page
455               res is a variable storing the server response (requested at each
456               UPDATE_INTERVAL ms). This data is updated by the server.
457             */
```

```
458             document.getElementById('game-status').innerHTML = res.timeLeft + " sec to
    start."
459           }
460           /* *** GAME STARTED *** */
461           else if ( game == 2 ){
462               /* disable the ready button */
463               $('#ready-btn').prop('disabled', true);
464             /* If user was ready */
465             if ( user.ready ) {
466               /* START */
467               played = true; // to remind that user has played the battle
468               myGameArea.start(); // set sensitive the keyUp and keyDown events
469               document.getElementById('game-status').innerHTML = "PLAY!";
470             }
471             else{
472               document.getElementById('game-status').innerHTML = "The game is started
    without you";
473             }
474           }
475         }
476
477       }
478       else {
479         console.warn("response: " + response);
480       }
481     },
482     /* Communication errors get here */
483     error: function (xhr, ajaxOptions, thrownError) {
484       console.error("ERROR " + xhr.status + ": " + thrownError);
485       updateGameError ++;
486       if (updateGameError > 5){
487         swal("OPS", "Connection to webserver lost", "error");
488
489         /* delete stored variables */
490         localStorage.removeItem('user');
491         user.name = "";
492
493         /* redirect to login page */
494         setTimeout(function(){
```

Listing 32: manageUsers.js - updateGameStatus

3.2.5.6 updateAvailableRobots

When this function is invoked, a POST request is performed to ask to the server the number of robots available on the network. The response is get and used to write that data in the file "index.html" (section 3.2.1).

```
503
504 /*
505  * UPDATE AVAILABLE ROBOT NUMBER:
506  *  get number of available robots add updated html into proper location
507  * ----------------------
508  *
509  * @type function
510  * @usage updateAvailableRobots();
511  */
```

```
512 function updateAvailableRobots() {
513  /* post request to the server to receive how many Robots
514    are connected to the application */
515  $.ajax({
516    url: '/getAvailableRobots',
517    type: 'POST',
518    data: {'data':''},
519    success: function(response) {
520      /* console.log("response: " + response); */
521      var status = JSON.parse(response).status
522
523      if ( status == "OK" ){
524        /* if response was successful, write that info on the web*/
525        document.getElementById('availableR').innerHTML = JSON.parse(response).availableR;
526      }
527      else {
528        console.warn("response: " + response);
529      }
```

Listing 33: manageUsers.js - updateAvailableRobots

3.2.5.7 updateReady

A POST request is sent to the server to notify that the user has changed its ready status (0: not ready, 1: ready). The response from the server is simply an "OK" to notify that everything went well.

```
538
539 /*
540  * UPDATE READY BUTTON:
541  *  update ready status on button press
542  * ------------------------
543  *
544  * @type function
545  * @usage on click event call updateReady();
546  */
547 function updateReady() {
548  /* Signal to the server through a post request that
549   the user is ready to play */
550  $.ajax({
551    url: '/playerReady',
552    type: 'POST',
553    data: {'user': user.name, 'ready': user.ready==0 ? 1 : 0 },
554    success: function(response) {
555
556      var status = JSON.parse(response).status
557
558      if ( status == "OK" ){
559        /* OK answer from the server means that server has update
560          on its own the sent information :
561          button behavior changed in updateGameStatus function */
562
563      }
564      else if ( status == "STARTED" ){
565        swal("OPS", "The game is already started. Please wait for it to finish.", "info");
566      }
567      else {
```

```
568         console.warn("response: " + response);
569       }
```

Listing 34: manageUsers.js - updateReady

3.2.5.8 help

When invoked, it opens an alert pop-up to show game instructions to the user.

```
577
578
579 /*
580  * HELP MESSAGE
581  * ------------------------
582  *
583  * @type function
```

Listing 35: manageUsers.js - help

4 Docker

4.1 Introduction

Docker is a platform to develop, pack, distribute, and manage applications within *containers*. It allows developers to build an application image containing all the related dependencies in a single package, such that making it independent and portable across multiple platforms and OS.

4.2 Installing Docker

In order to automatically install Docker CE:

```
$ curl -fsSL get.docker.com | sudo sh
```

In order to use Docker as a non-root user, it is necessary to add the user to the "docker" group issuing:

```
$ sudo usermod -aG docker <your-user>
```

In order to make operative this change, a log-out and log-in is needed.

For uninstalling the Docker CE package:

```
$ sudo apt-get purge docker-ce
```

Images, containers, volumes, or customized configuration files on the host are not automatically removed. To delete all images, containers, and volumes:

```
$ sudo rm -rf /var/lib/docker
```

4.3 Dockerfile

It is a textual file composed by a collection of instructions that define what the image will contain and how it will behaves at runtime. The Docker Engine will be in charge of validating the Dockerfile and generating the customized image starting from a base one.
Basic docker files instructions are shown in the following :

- `FROM`: it allows to specify the Base Image from where to start to create the customized one
- `RUN`: to execute commands in the generated image (i.e. install a package). Note that each RUN command generate a new image layer wrapping the modifications introduced by the command itself. Thus, the father image remains untouched. It is good practice to collapse correlated commands within the same RUN instruction
- `ADD`, `COPY`: used to move files and directory from the build context (local host path, where the Dockerfile is) to the Filesystem of the created image.
- `ENTRYPOINT`: It allows to execute a command (or a set of commands) within the container as soon as it is started. The difference with respect to RUN is that in this case the effects of the instruction affects the container itself and not the image generating it.

Once the Dockerfile has been created, the image can be built, starting from a dedicated folder containing only the Dockerfile and needed dependencies, issuing:

```
$ docker build -t "user_name/repo_name:image_tag" .
```

The name "user_name/repo_name:image_tag" is just a name, but it is common practice to use this notation for associating a local image with a repository when the intent is to share the image. Note that the tag is referring to a specific version of an image. There could be, indeed, different images (with different components inside) branching from the same core image.
To upload the image and make it available to public, it is necessary to have an account on cloud.docker.com. Then, by issuing the command:

```
$ docker push user_name/repo_name:image_tag
```

the already compiled image will be transfered on the indicated repository, which can be downloaded later on time by simply using a pull.

4.4 Most used Docker commands

In the following list there are some of the most commonly used Docker commands:

- List docker images:

  ```
  $ docker images
  ```

- Pull or update a docker image

  ```
  $ docker pull <docker_user>/<image>:<tag>
  ```

- Run a docker image:

  ```
  $ docker run --rm -it <image>
  ```

  When this command is ran it will automatically pull the <image> (if not already present on the host), create a new container, interactively run it (-it flag) and remove the container once the it is terminated (–rm flag)

- Run an image and mount a shared folder:

  ```
  $ docker run --rm -it -v <host_dir_path>:<guest_dir_path> <image>
  ```

  This create a data volume shared between the container and the host. Volumes are a good way to persist data generated during the execution of a container. In this way, each file added in <guest_dir_path> at runtime will be visible on the localhost at <host_dir_path>.

- Build a docker image (in the current folder a dockerfile must be present):

  ```
  $ docker build -t "<docker_user>/<image>:<new_tag>" .
  ```

- Close all opened containers:

  ```
  $ docker rm $(docker ps -a -f status=exited -q)
  ```

5 Raspberry

In this section the **Raspberry** part is documented. Its main tasks are :

- Generating ROS node (robot) able to exchange messages with the ROS master (Web Server) and the Arduino board.
- Forwarding commands coming from the Server to Arduino.
- Forward notifications coming from Arduino sensors to the Server (in charge of uploading battle status accordingly).

It was necessary to use both Raspberry and Arduino since ROS needs a real Operating System to work with, and Arduino, which has limited resources, is not capable to control the robot by itself. Scripts running over raspberry board are described below.

5.1 ID_service_client.py

Here it is implemented the service *client-side*.

```
1  #!/usr/bin/env python
2
3  import sys
4  import os
5  import rospy
6  import subprocess
7  import time
8  from laser_bot_battle.srv import *
9  from pingThread import pingThread
10
11 def add_new_robot_client():
12     # Wait until the service is not activated in ID_service_server.py
13     rospy.wait_for_service('add_new_robot')
14     try:
15         if os.path.exists("/dev/ttyACM0"):
16             device = "/dev/ttyACM0"
17         elif os.path.exists("/dev/ttyUSB0"):
18             device = "/dev/ttyUSB0"
19         else :
20             print "No attached ARDUINO found"
21             return
22
23         new_robot = rospy.ServiceProxy('add_new_robot', AddNewRobot)
24         # new_robot() makes the request to the server and the returned ID is saved
25         robot_ID = new_robot()
26
27         # Create and start alive ping thread
28         ping = pingThread(robot_ID.ID)
29         ping.start()
30
31         # Wait 2 sec and check if ping is alive
32         time.sleep(2)
33         if not ping.isAlive():
34             print "Connection to webserver lost"
35             return
36
37
38         print ("\nNew robot ID = %d"%robot_ID.ID)
```

```
39          # The command to generate the ros node with unique name and topics is composed here
40          command = 'rosrun rosserial_python serial_node.py ' + str(device) + ' __name:=Robot'
    + str(robot_ID.ID)
41          command += ' command:=Robot' + str(robot_ID.ID) + '/command'
42          command += ' response:=Robot' + str(robot_ID.ID) + '/response --respawn True'
43
44          print "Running client application:\n", command
45          output = subprocess.check_call(command, shell=True)
46
47          print "Connection to Arduino lost"
48
49          # If here connection to raspberry is lost
50          # Stop sending alive ping
51          ping.join( )
52
53      except rospy.ServiceException, e:
54          print "Service call failed: %s"%e
55
56  if __name__ == "__main__":
57      while True:
58          print("\nStarting service client ...")
59          add_new_robot_client()
60          time.sleep(5)
```

Listing 36: ID_service_client.py

This script is launched at robot power-on. It calls the function add_new_robot_client() which:

- Waits for the availability of the service. This is used to block the robot initialization until it is assigned with a valid ID by the master.
- Checks on which serial port Arduino is available. This control has been implemented to cope with different USB driver Arduino can use (i.e. ttyACM0 is the standard for Linux).
- Calls new_robot() to request the Server to provide an available robot ID if possible (see section 3.1.2).
- Creates and starts a ping thread. This is used to continuously ping the Server in order to check if the robot is still connected to the network.
- Creates the serial node by launching the script *serial_node.py*. This initializes the rosserial connection between Arduino and Raspberry. The two dedicated topics, created by appending the robot ID provided by the Server are:
  - command: of type Robot_msg, containing commands for linear, angular movements and shoot.
  - response: of type std_msgs/Empty, to notify the robot has been hit.

  This allows to have dedicated topics for each node, such that messages can be exchanged point-to-point between robots and server without interfering with each other.

5.1.1 pingThread.py

In this section more details about how robot connection status is checked are provided.
For this application, understanding whether the robot is still up, connected and running is crucial.

Indeed, the ROS master has no means to understand if a robot has disconnected from the network by only leveraging on ROS base functionalities. There could be two possible scenario:

- the connection between server-Raspberry is lost
- the connection between Raspberry-Arduino crashes

A ping solution has been introduced to cope with this.

As mentioned before, prior robot initialization is completed, pingThread is started. Its argument is the ID of the robot it is called from. Every 0.4 sec it will perform a POST request, containing the ID as payload, at '/incAlive' address of the Server, signaling the robot is still alive.
If the serial_node.py for some reason crashes, and the Arduino is no more accessible from the raspberry, the pingThread is killed. In this way the Server does not receive ping anymore and it can detect that the robot is down. Consequently, the user connected to the faulty robot will be removed from the game active list by the Server.

```
#!/usr/bin/env python

import threading
import requests



class pingThread(threading.Thread):

    def __init__(self, id, name='pingThread'):
        """ constructor, setting initial variables """
        self._ID = id;
        self._stopevent = threading.Event()
        self._sleepperiod = 0.4
        threading.Thread.__init__(self, name=name)

    def run(self):
        """ main control loop """
        print "%s starts" % (self.getName(),)

        payload={'ID': self._ID}
        headers={}
        url='http://laser_bot_master.local:5000/incAlive'
        #url='http://localhost:5000/incAlive'

        while not self._stopevent.isSet():

            # Send POST request with ID data
            r = requests.post(url, data=payload, headers=headers)
            # Sleep
            self._stopevent.wait(self._sleepperiod)

        print "%s ends" % (self.getName(),)

    def join(self, timeout=None):
        """ Stop the thread and wait for it to end. """
        self._stopevent.set()
        threading.Thread.join(self, timeout)
```

Listing 37: pingThread.py

6 Arduino

In this section the **Arduino** part is documented.
Arduino was chosen since it offers a convenient interface to easily controlling actuators and reading from sensors. It is used to drive motors and IR emitter, in response of Raspberry requests, and to read the IR sensors to detect when the robot has been hit. The following figure summarizes the robot architecture, showing raspberry handling network communications, forwarding commands to Arduino which, then, properly drive the robot hardware.

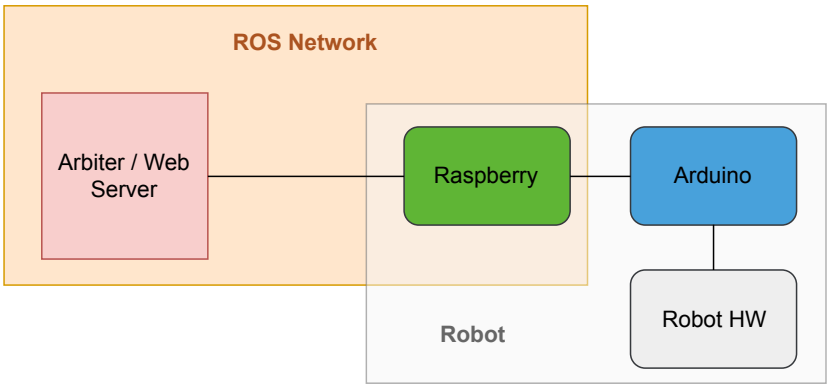

Figure 8: Ros network - Robot Architecture

6.1 Arduino Rosserial Setup

After installing Arduino IDE from the official website, check the sketchbook location by opening it and browsing into `File/Preferences` (`/home/<user>/Arduino` by default). It is required to include ros_lib into the 'libraries' folder of your sketchbook in order to use "rosserial". The commands to be issued to install Rosserial are :

```
$ sudo apt-get install ros-kinetic-rosserial-arduino
$ sudo apt-get install ros-kinetic-rosserial
```

To install ros_lib into Arduino environment it is required to firstly clean up the folder from any already existent ros_lib folder and then installing it through a script :

```
$ cd <sketchbook>/libraries
$ rm -rf ros_lib
$ rosrun rosserial_arduino make_libraries.py .
```

In order to establish the communication between an host and the arduino board through rosserial it is required to check the target port to which the board is connected, by browsing into Tools/Port in the Arduino IDE. By default this port is /dev/ttyACM0 for Linux based distributions. As a consequence the command which is issued by the raspberry to enable the serial communication will be :

```
$ rosrun rosserial_python serial_node.py /dev/ttyACM0
```

6.2 Arduino Libraries

Here it is described which libraries are required to be included in the sketchbook 'libraries' folder in order for the sketch to be executed.

6.2.1 ROS Custom Messages Integration

In order to include custom messages into ros_lib the following steps have been performed :

- Copy the previously generated custom message Robot_msg.msg into the target package laser_bot_battle.
- Build the workspace such that producing an header into the devel/include folder. Note that this is not the header Arduino will use.

```
$ catkin_make
```

- Remove ros_lib from Arduino 'libraries' folder.

```
$ rm -r ros_lib
```

- Remaining into 'libraries' folder, issue the command :

```
$ rosrun rosserial_arduino make_libraries.py .
```

  This will rebuild the arduino ros_lib library, resolving the new dependencies for the custom message, by automatically generating the header file Robot_msg.h into /Arduino/libraries/ros_lib/laser_bot_battle

- Now it is possible to include the custom message into the target arduino sketch :

```
#include <laser_bot_battle/Robot_msg.h>
```

6.2.2 AccelStepper Library

This is the library used to drive the stepper motors in charge of moving the robot wheels. First step to be performed is to download the library unzip it and copy the AccelStepper-1.57.1 folder in the sketchbook 'libraries' folder. This library provides all the methods required to make motors moving in both directions, set their speeds and stop them in target positions.

6.2.3 Timer Library

This library has been chosen in order to avoid the use of the arduino delay() function to set and unset the IR driver pin (laser) after a period of time. The difference consists in the fact that the delay function is a blocking function which causes Arduino to be not responsive to commands during the specified amount of time. In order to avoid loosing commands some methods of this Timer library can be exploited. The library can be downloaded from a github repo and useful documentation can be found on the developer website.

6.3 Arduino Sketch

In this part it is described the firmware to be uploaded to Arduino in order to manage application commands.

Some constants have been defined in order to easily change robot parameters :

- *STEPS_PER_MOVEMENT* : It defines the number of steps the stepper motor performs each time a movement command is received (angular, linear).
- *SPEED_SCALER* : It defines the divider to be applied to motor speed. It is used when the user presses l/r and f/b arrows at the same time to make the motor to curve. To allow this kind of movement some wheel speeds have to be put to lower values.
- *STEPPER_SPEED* : It defines the maximum speed at which the motor can turn (1024 is the maximum value accepted)
- *IR_XX_PIN* : Arduino pins to which IR sensor and driver are attached to.
- *IR_RAY_PERIOD* : It defines how long the IR laser stays on when a shoot command is received from the user.

```
#include <AccelStepper.h>
#include <Timer.h>
#include <ros.h>
#include <std_msgs/Empty.h>
#include <laser_bot_battle/Robot_msg.h>

// NB : The time to complete these steps should be consistent with the rate of incoming
     Robot_msgs (greater)
// Steps the motor has to perform at each Robot_msg received
#define STEPS_PER_MOVEMENT_CW 320
#define STEPS_PER_MOVEMENT_CCW -320
// Scale factor used when robot turns l/r
#define SPEED_SCALER 0.7
// Default constant speed
#define STEPPER_SPEED 1024
// Pins to manage infrared sensor and driver
#define IR_RX_PIN 2
#define IR_TX_PIN 3
// Period of the IR ray when shooting (in millis)
#define IR_RAY_PERIOD 500
```

Listing 38: rosserial_node_arduino.ino - Robot Parameters

The set of global variables and objects used is defined in the following :

- *is_ir_on* : Used to detect if the laser is already on when the user presses the button another time.
- *nh* : It is the ROS serial node instance.
- *hit* : It is a ROS empty message sent back to the raspberry when a robot is hit by the laser.

- *pubHit* : It is the ROS publisher instance, used to sent 'hit' messages over the topic 'response'.
- *sub* : It is the ROS subscriber instance, used to receive 'Robot_msg' messages over the topic 'command'. When this happens the callback function 'robot_cb' is invoked.
- *stepper_X* : These are the two instances of the AccelStepper class defining two stepper motors. HALF4WIRE is the selected mode followed by the pins to which the motor drivers are attached.
- *t* : It is the timer activated when a shoot command is received, used to manage the duration of the IR transmission.

```
21 volatile bool is_ir_on = false;
22
23 // ROS Node
24 ros::NodeHandle  nh;
25
26 // Msg to notify the robot has been hit
27 std_msgs::Empty hit;
28
29 // Used to publish to the response topic when the robot is hit by laser
30 ros::Publisher pubHit("response", &hit);
31
32 // The Robot receive a Robot_msg and actuate the stepper motor and/or to the infrared driver.
33 ros::Subscriber<laser_bot_battle::Robot_msg> sub("command", &robot_cb);
34
35 // Stepper motor instances
36 AccelStepper stepper_l(AccelStepper::HALF4WIRE, 8, 10, 9, 11);
37 AccelStepper stepper_r(AccelStepper::HALF4WIRE, 4, 6, 5, 7);
38
39 // Timer used to manage the duration of the IR transmission
40 Timer t;
```

Listing 39: rosserial_node_arduino.ino - Global variables and objects

A callback function (robot_cb()) is called when a new 'Robot_msg' command is received. It is basically used to unzip the 'Robot_msg' fields and to make the robot behaving accordingly. Here it is decided if and which one of the wheels have to move clockwise or counter-clockwise, their speeds and if the IR driver has to be turned on or not. Since well commented, by following the function structure it is possible to understand how this behavior has been achieved.

```
42 //CallBack function to manage a received msg
43 void robot_cb( const laser_bot_battle::Robot_msg& cmd_msg) {
44
45   short step_l = 0;
46   short step_r = 0;
47
48   // Set speed to constant speed
49   stepper_l.setSpeed(STEPPER_SPEED);
50   stepper_r.setSpeed(STEPPER_SPEED);
51
52   if (cmd_msg.linear_x == 0) {
53
54     // Here if turning around l/r or stopping
```

```
55      switch (cmd_msg.angular_z) {
56        case  1 :
57          // Turn around Clockwise
58          step_l = STEPS_PER_MOVEMENT_CCW;
59          step_r = STEPS_PER_MOVEMENT_CCW;
60          break;
61        case -1 :
62          // Turn around Counter Clockwise
63          step_l = STEPS_PER_MOVEMENT_CW;
64          step_r = STEPS_PER_MOVEMENT_CW;
65          break;
66      }
67    }
68    else {
69      if (cmd_msg.linear_x == 1) {
70        // Move Forward
71        step_l = STEPS_PER_MOVEMENT_CCW;
72        step_r = STEPS_PER_MOVEMENT_CW;
73      }
74      else if (cmd_msg.linear_x == -1) {
75        // Move Backward
76        step_l = STEPS_PER_MOVEMENT_CW;
77        step_r = STEPS_PER_MOVEMENT_CCW;
78      }
79
80
81      // Here if moving straight + turning r/l or stopping
82      if (cmd_msg.angular_z == 1) {
83        // Move straight + turn right
84        stepper_r.setSpeed(STEPPER_SPEED * SPEED_SCALER);
85      }
86      else if (cmd_msg.angular_z == -1) {
87        // Move straight + turn left
88        stepper_l.setSpeed(STEPPER_SPEED * SPEED_SCALER);
89      }
90    }
91
92    // Relative movement of the motor from current position by step_l/step_r
93    stepper_l.move(step_l);
94    stepper_r.move(step_r);
95
96    if(cmd_msg.shoot == true && is_ir_on == false){
97      // IR driver pin enabled
98      is_ir_on = true;
99      digitalWrite(IR_TX_PIN, LOW);
100     // timer enabled to generate an interrupt after IR_RAY_PERIOD ms
101     t.every(IR_RAY_PERIOD, resetIrRay, 1);
102   }
103 }
```

Listing 40: rosserial_node_arduino.ino - Subscriber callback function

The interrupt handler for the IR_RX_PIN has nothing to do but sending an 'hit' message to the raspberry when the IR sensor is hit by the laser.

```
105 // Hit interrupt handler
106 void sendHit() {
107   pubHit.publish(&hit);
108 }
```

Listing 41: rosserial_node_arduino.ino - IR sensor interrupt handler

The timer interrupt handler is used to disabled the IR_TX_PIN (IR driver) when IR_LASER_PERIOD expires.

```
105 void resetIrRay() {
106   // IR driver pin disabled
107   digitalWrite(IR_TX_PIN, HIGH);
108   is_ir_on = false;
109 }
```

Listing 42: rosserial_node_arduino.ino - Timer interrupt handler

Arduino setup function is used to set the stepper motors max speeds, to setup IR pins mode and interrupts and to initialize the ROS serial node and its publisher and subscriber.

```
116 void setup() {
117   // Set stepper motors max speed
118   stepper_l.setMaxSpeed(STEPPER_SPEED);
119   stepper_r.setMaxSpeed(STEPPER_SPEED);
120
121   // Setup IR pins
122   pinMode(IR_RX_PIN, INPUT_PULLUP);
123   attachInterrupt(digitalPinToInterrupt(IR_RX_PIN), sendHit, FALLING);
124   pinMode(IR_TX_PIN, OUTPUT);
125   digitalWrite(IR_TX_PIN, HIGH);
126
127   nh.initNode();
128   // Association Publisher and Subscriber
129   nh.advertise(pubHit);
130   nh.subscribe(sub);
131 }
```

Listing 43: rosserial_node_arduino.ino - Arduino setup function

Arduino loop function is the main function which is executed repeatedly until it is stopped. It is used to active subscriber callback function if pending calls are present (spinOnce() function), to move the stepper motors to the target positions and to update the timer counting value.

```
133 void loop() {
134   nh.spinOnce();
135
136   // Move the motor until the target position previously by move is reached (1 step per
        iteration)
137   stepper_l.runSpeedToPosition();
138   stepper_r.runSpeedToPosition();
139
140   // To service the pulse event associated with the timer
141   t.update();
142 }
```

Listing 44: rosserial_node_arduino.ino - Arduino loop function