TP: A Multi-Robot Platform for Mobile Robots with ROS

Submitted to:

Prof. Assia Belbachir

Prof. Johvany Gustave

Prof. Jamy Chahal

By

Clément Besneville	20132253
Fatma Mohamed	20206429
Wojciech Stróżecki	20205182

Contents

List of Figures **4**

Chapter 1 Introduction **5**

Chapter 2 Problem Statement **7**

Chapter 3 Algorithm and obstacles **8**
3.1 Finding the flags. 9
3.2 Obstacle avoidance 12
3.3 Storing all flags. 14

Chapter 4 Results **16**

Chapter 5 References **17**

Chapter 6 Appendix **18**

List of Figures

2.1 Environment 7

3.1 Main loop algorithm 8
3.2 The polar coordinates 9
3.3 The transition point 9
3.4 Flowchart explains the algorithm 10

4.1 Results published to the console 16

Chapter 1

Introduction

At the end of the last century, industrial robotic systems were oriented toward the mass production of products in factories, where robots of high precision were destined to carry out repetitive individual work in controlled environments. However, the current tendency is toward robotic systems that must be capable of solving problems in environments that are more complex and less controlled. To this end, robotic systems are needed that are more autonomous and intelligent. Some of these systems are mobile teams of autonomous robots where a set of robots works as a group to attain a common objective. These systems require cooperative, social robots, which can move in dynamic, complex uncontrolled environments, where the capacity to understand and interpret the surrounding world is their prime challenge.1 As a consequence, the complexity of the software architecture increases and the computing needs soar. In these software architectures, the scalability, reusability, efficiency and fault tolerance play a fundamental role. The system software architecture must be designed in a distributed and modular way. These systems must nevertheless continue to take classic problems of industrial robotics into account, which are closer to the sensors and actuators, such as the real-time requirements in their lower levels. The principal characteristics, which the software architecture (and its components) of a complex robotic system must cater for, are then enumerated ;

- **Concurrent and distributed architecture:** It is necessary to be able to take advantage of all the processing units available in a concurrent way (processors, multi-processors and microcontrollers) in order to cover all the computational needs that a complex robotic system presents. Due to the consequent system complexity, a powerful remote inspection (also known as introspection) mechanism is needed.

- **Modularity:** The software architecture is formed of several components of high cohesion and low coupling. The components interact with each other; however, the dependences must be kept at a minimum in order to obtain a maintainable, scalable, and reusable architecture, which is adaptable to changes and improvements. Although these are desired features in all software architectures, they are especially important in Robotics, where the lack of standards and the closeness with the hardware have made robotics software prone to be non-reusable and non-scalable for succeeding decades. The need for a welldesigned common robot hardware interface is another related feature.

- **Robustness and fault tolerance:** The malfunction of a component must not completely block the whole system. On the other hand, the rest of the system must be capable of continuing to work as best it can with the resources available on the condition. that it is

working toward achieving the objectives. To this end, the rest of the components (still in working order) must be capable of acting on their own initiative and autonomously make decisions to overcome these situations. These decisions must be taken based on cooperation with other agents of the system or on the specific information they possess.

- **Real time and Efficiency:** The majority of robotic systems have some type of real-time constraints. These restrictions are problematic in distributed software architecture. Efficiency is also a common requirement, especially when a robotic system has limited communication and computation capacities. Hence the design of the architecture must consider the use of software, hardware, communication mechanisms and protocols that guarantee compliance with these restrictions.

Hence, the main reasons why MAS are a good choice for robotics software architecture are that, when using this approach, the resultant software is much more reusable, scalable and flexible whilst the parallelism, robustness and modularity requirements are maintained. This approach is demonstrated through its successful application in multiple areas in Robotics. Indeed, Multi-Agent Robotic Systems (MARS) have been widely studied over recent years and related events exist in the form of competitions such as RoboCup. Technologies are available that enable the development of general-purpose MAS which will be referred to as MASFs (MultiAgent System Frameworks) in this work. These technologies include: JADE, Grasshopper, JACK, Cougaar, ADE, and Mobile-C. JADE is the de facto solution for various reasons, most importantly that it was the first to implement the MAS specification defined by FIPA2 and was an Open-Source solution both accepted and supported by the MAS developers' community. These technologies have been used successfully during the past few years. However, whilst they present a valid approach for robots, their use has not been focused on the development of robotic systems but rather on other areas such as Web services, e-commerce, domotics, sensor networks, social simulation, finances and e-learning. On the other hand, "Robotics Software Frameworks" (RSFs) attempt to provide an integral solution through a set of generic tools and off-the-shelf libraries with algorithms and controllers useful to create a general-purpose robotic systems, thereby avoiding to continually reinvent the wheel. On occasions, these frameworks are known as "Robotics Middleware" or "Robotics Software System".

Chapter 2

Problem Statement

In the problem, we have the following environment

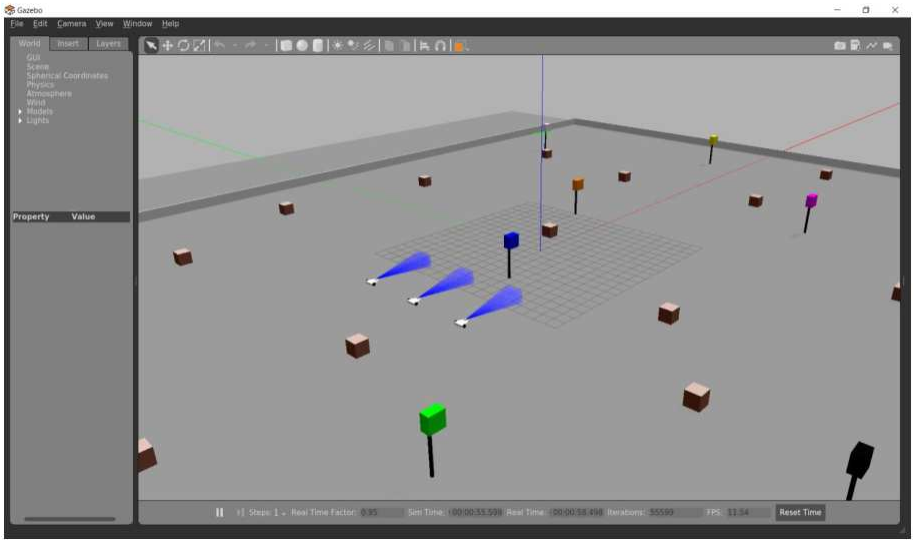

Figure 2.1: Environment

The environment is as shown above contains a fleet of at least three robots to localize cooperatively a maximal number of flags in order to win the game.

We characterize the robots as follow:

- Each robot has its own ID.
- Each robot can communicate with another one.
- Each robot embeds an ultrasonic sensor to observe the environment through a restricted range (3 meters).
- Each robot moves in the environment with two wheels The flags are placed randomly in the robot environment and send a continuous signal. Each flag has its own ID.

Chapter 3

Algorithm and obstacles

For the Algorithm, we divided the code into 3 parts, for each person to implement a part of it. So, we can split them into

1. Moving the Robots to find the Flags.
2. Obstacle avoidance to be checked before every step.
3. Storing all flags found in an array in order to make sure no flag is found twice.

For the main while loop alogirthm showed in the following figure is executed every 0.65 seconds (samplingTime).

- First the relative velocity to the nearrest flag is computed.
- Then if the flag is detected, the flag's coordinates are computed and saved.
- Additionally, In each step, the value of the sonar in front of the robot is checked in order to detect obstacles and an appropriate action is taken to prevent a collision.

```
while not rospy.is_shutdown():

    #Write here your strategy..

    samplingTime = 0.65

    # computing the relative velocity of the robot to the nearrest flag
    robot.computeVelocities(samplingTime)
    # checking if the transition from positive to negative velocity was detected
    velocity, angle = robot.velSignChange()

    sonar = float(robot.get_sonar())
    # obstacle avoidance by making a random turn
    if sonar < 4:
        robot.setOrientation(random()*2*3.1415-3.1415)

    #Finishing by publishing the desired speed. DO NOT TOUCH.
    robot.set_speed_angle(velocity,angle)
    rospy.sleep(samplingTime)
```

Figure 3.1: Main loop algorithm

3.1 Finding the flags.

The flag finding algorithm takes advantage of the fact that the distance to the flag is known at each time. This information can be used to compute the relative position of the flag to the robot in polar coordinates. The only missing data is the orientation .

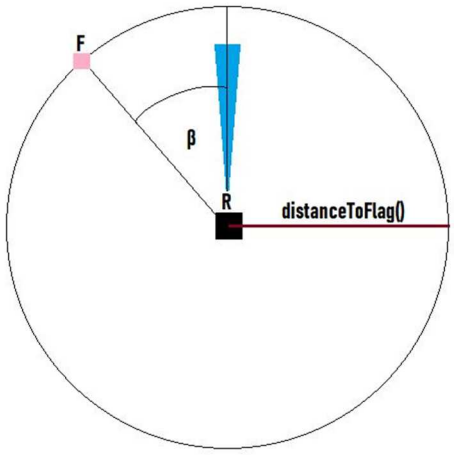

Figure 3.2: The polar coordinates

The robot is assumed to go forward on a straight line. By computing the derivative of the distance to the flag, the velocity of the robot relative to the flag can be computed. When the robot is getting closer to the flag, the velocity is assumed to be positive. Then the sign of the velocity is examined and a point of transition of the sign from positive to negative is anticipated. This point is the point of closest distance to the flag, and at that point he has the flag exactly on his lateral axis, so the orientation of the flag relative to the robot is known.

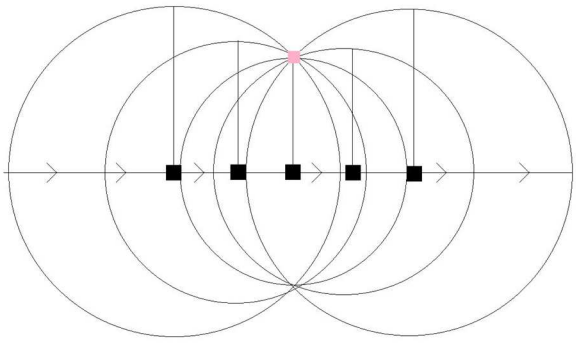

Figure 3.3: The transition point

The last step is to determine whether the flag is on the left or on the right side of the robot. To do that it turns 90 degrees to the arbitrarily chosen side and moves forward a little. If the distance to the flag got smaller, it means that the robot is pointing in the flag's direction. If the distance grew, the flag was on the opposite side. Now, the flag's coordinates can be precisely estimated and saved. The block scheme of the algorithm looks as follows:

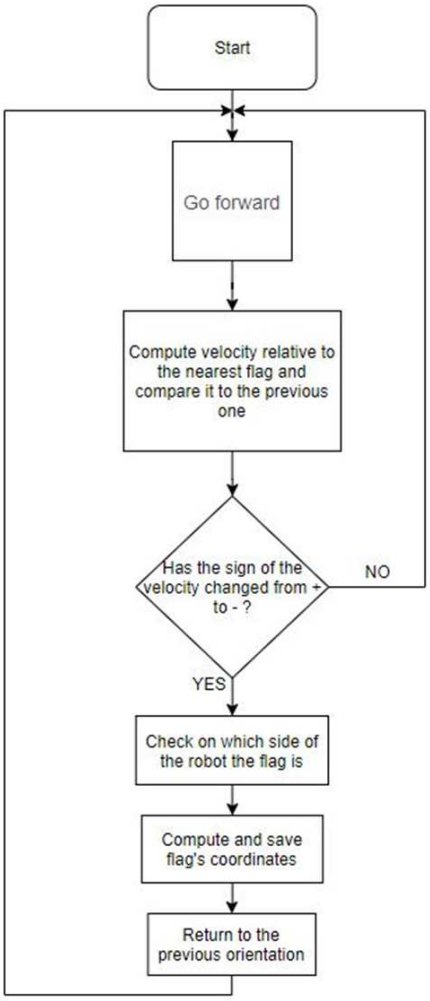

Figure 3.4: Flowchart explains the algorithm

The key function for operating the algorithm is the setOrientation(self, goal).

```
def setOrientation(self, goal):
    # this function sets robot's orientation for an angle given as an argument. Not very elegant since while is used, but it does its job well
    while goal > 3.1415:
        goal = goal - 2*3.1415
    while goal < -3.1415:
        goal = goal + 2*3.1415
    if ((self.yaw - goal) >= 0 and (self.yaw - goal) < 3.1415) or ((self.yaw - goal) >= -2*3.1415 and (self.yaw - goal) < -3.1415):
        while abs(self.yaw - goal) > 0.1:
            self.set_speed_angle(0, -self.ang)
            self.isTurning = 1
            rospy.sleep(0.01)
    else:
        while abs(self.yaw - goal) > 0.1:
            self.set_speed_angle(0, self.ang)
            self.isTurning = 1
            rospy.sleep(0.01)
    self.set_speed_angle(0, 0)
```

Its role is to set the robot's global orientation for a goal value. First, in the while loops it assures that the goal has a proper value, in the range of (-pi ; pi). Then in the if condition it checks whether it's more optimal to take a turn to the left or to the right. After it has decided, it starts turning and keeps checking if the current orientation reached the goal orientation. After it reaches it with a certain accuracy (0.1 radians) it stops the robot.

3.2 Obstacle avoidance

For the obstacle avoidance part. we tried to keep everything simple. when an obstacle is detected, the robot turns a little to take the min. deviation from its original path.

1. First, we defined the sonar function in order to call it anywhere in the code

```
def callbacksonar(self,data):
    self.sonar = data.range
```

2. then whenever its required to check for an obstacle, all we need to do is to call the function

```
sonar = float(robot.get_sonar())
```

3. If the sonar value is less than 4, we choose a random positiong for the robots.

```
if sonar < 4:
    robot.setOrientation(random()*2*3.1415-3.1415)
```

   the random function generated a random number (between 0 and 1). then we multiply it by π and subtract π from the result from, in order to give us a final value of a number (between $-\pi$ and π).

   this range is the default range of orientation that the robot can have as an input.

This function is the last agreed (and simplest function implemented) , there were previous trials to make it optimized. we will try to list the previous trials briefly.

1. First Trial: we basically wanted the robot to turn 90 degrees exactly to the left. then stop for a while, and move straight.

```
#Write here your strategy..
sonar = float(robot.get_sonar())
if sonar < 3:
    initYaw=robot.yaw
    robot.setOrientation(initYaw + 1.5708)
    robot.set_speed_angle(2,0)
    rospy.sleep(0.5)
```

   This method worked, but we choose the above method as it's more simpler, and no need to turn exact angles.

2. Second Trial: As we have explained in the flags algorithm our optimized algorithm, to be able to implement this algorithm, we needed to make an obstacle avoidance that turns the robot exactly around the obstacle and completes it's exact path. It's something close to a controller in UAVs, it's role is to get the UAV back to it's current state after a deviation due to noise or any other reason.

```
#Write here your strategy..
sonar = float(robot.get_sonar())
if sonar < 3:
    initYaw=robot.yaw
    robot.setOrientation(initYaw + 1.5708)
    robot.set_speed_angle(2,0)
    rospy.sleep(0.1)
    robot.setOrientation(initYaw - 1.5708)
    robot.set_speed_angle(2,0)
    rospy.sleep(0.1)
    robot.setOrientation(initYaw - 1.5708)
    robot.set_speed_angle(2,0)
    rospy.sleep(0.1)
    robot.setOrientation(initYaw + 1.5708)
    robot.set_speed_angle(2,0)
    rospy.sleep(0.1)
```

3.2.1 Obstacles

We can summarize the obstacles faced us into those points.

- For the optimized algorithm, the flags near the obstacles will never be discovered because we already set a path (for optimization purposes).
- For the turning angles the set.speed_angle function didn't work properly and didn't make exact turns. so we had to implement our own function (orientation function).
- The last simple code worked perfectly for one robot and we faced some errors to make it work for the three robots. but we solved them eventually.
- The sonar sensor isn't a liadr sensosr so, the robot doesn't scan the obstacles (and flags as well) in a specific range. It has to rotate, which created the turning problem that we have already mentioned above.

3.3 Storing all flags.

A simple way to make all robots having a sort of communication would have been to make a global array accessible from each robot. In order to register all flags found, store them and be able to restore them at the end we used an array. This array has to be defined as containing 8 rows (flags) and 2 or 3 columns (x, y and ID)

```
15 nbrFlagFound = 0
16 flagsArray = np.zeros((8,3), dtype='int')
```

The use of arrays needs to import the library numpy. Once the library was installed and imported, we were facing the problem of having an array accessible from all the robots. We then decided to use a global variable.

The implementation is using 3 methods:

- One is for an incrementation of the number of flags found. Each time we find a flag, we increment the global variable in order to know when all flags have been found.

```
41 def incrementFlagFound():
42     global nbrFlagFound
43     nbrFlagFound = nbrFlagFound + 1
44     return nbrFlagFound
```

- The second is about to store the flags coordinates in the array. Each time we find a flag, it is added into the array at the location corresponding to its ID

```
271         global flagsArray
272         if (flagsArray[locID-1,2] == 0):
273             flagsArray[locID-1] = (int(flg.x),int(flg.y),int(ID[1]))
```

- The last one is used to recap the flags found. Once all flags had been found, we call this method to print the contents of the array and recap the flags coordinate after saying that the goal was attempted.

```
46 def flagsRecap():
47     i=0
48     print("All flags have been found:")
49     while i<8 :
50         print("Flag N°: " +str(i+1)+ " X= " +str(flagsArray[i,0])+ " Y= " +str(flagsArray[i,1]))
51         i+=1
52     return 0
```

- Since we are running the code for each robot, and not as the whole simulation, we had to use the publisher and subscriber methodology.

As we manage to discuss about different strategies, the first we commonly choose was to define a path to each robot. In this way we would have done the task in few minutes or maybe less. In order to do that, we had to make the robots running until precise locations and do strict planning. To do this, we wrote a method called 'moveTo' which make the computation of the wright angle. Chosen by the coordinates of a goal point, the angle is calculated by the following mathematical relation:

$tan(\frac{px-gx}{py-gy}) + \pi$

With p the current point and g the goal point.

We then were able to go in the good direction, the work was now to run through the distance separating the current point from the goal point. Distance obtained by the following calculation .

$\sqrt{(px-gx)^2 + (py-gy)^2}$

A simple rospy.sleep(Distance) with a velocity of 1 m/s could have been easy. But it would have disabled the obstacle avoidance and be a huge failure... Whatever, calling this method has to face with the robots desired location. It is then called as follows:

```
304     if (robot_name == "robot_1"):
305         robot.moveTo(-47,-22)
306
307     if (robot_name == "robot_2"):
308         robot.moveTo(-47,0)
309
310     if (robot_name == "robot_3"):
311         robot.moveTo(-47,22)
```

Another problem was to have back the ID of a flag properly. The solution we are using is to split the string obtained by the getDistanceToFlag() function and then act on it as an array. As it is composed of 3 characters: a space, a number and closed parenthesis. We have the following lines of code:

```
257         s = str(self.getDistanceToFlag())
258         Distance, ID = s.split(',')
259         locID = int (ID[1])
```

Chapter 4

Results

Each Robot finds the flags and avoid obstacles successfully. The mission might take more time to be solved. All flags will be found eventually.

We had some problems with the implementation of the communication and that's mainly because our first trial was a trajectory based algorithm. The code is attached in the folder but we didn't add it to the report because we didn't implement it.

As for the Final results, we attached a video of the Robots solving the mission as the robots move randomly to scan all the map with the console open to show the flags coordinates and ID that has been saved, as in the following picture.

Figure 4.1: Results published to the console

Chapter 5

References

1. Lectures Handouts and TDs.
2. https://answers.ros.org/question/237063/howto-use-int16multiarray-in-python/
3. http://docs.ros.org/en/api/std_msgs/html/msg/Int32MultiArray.html

Chapter 6

Appendix

Python codes and Videos attached as (.py) files and (.wmv) files