Li-Fi applied to robotics : automation on disconnection

1 Introduction

This project was originally conducted by me and the other three classmates. Below I'll extract the main points which illustrates the part I took charge in specifically. This project was a collaboration with the company Lucibel that develops the Li-Fi technology (Light-Fidelity). It is a technology still little used and we collaborated with them to help develop its usage for indoor robotics.

Li-Fi is a kind of wireless communication using light to transmit data, however mobile robots frequently get disconnected from fixed sources. I was responsable for writing scripts (Python, Bash) to make our robot behave correctly during disconnection and direct itself to the nearest source. Our robot is called Turtlebot3 and has led to carry out thorough research on ROS (Robot Operating System).

Since we had some problems with the installation of Li-Fi's driver, and failed to get help from Lucibel during our project, we tried to use Wi-Fi to simulate the same behavior to get a result close to what we could have obtained with a Li-Fi connection.

2 Specificity of problem & Objectives

To stay close to the initial problems, we restricted ourselves to Li-Fi's contraints :

1. Li-Fi's covering zone is not connected, thus in some areas the robot cannot receive orders from the network : we must be able to know in real time if it is connected.
2. We need to be able to simulate disconnections and, if necessary, run the robot with an independant script at the same time.
3. We must be able to find a connection and, as soon as it is found, control our robot from the Remote PC.

Our two main objectives were :

1. Write scripts allowing us to control the robot's movement efficiently.
2. Have a global control of the robot upon connection and disconnection in real time.

The other team worked on the research of good algorithms in Python during the robot's disconnection in order to find the nearest Li-Fi source, which they simulated with circles, in the most efficient way. The final goal of our project was to combine these two parts so that the robot can behave correctly during its loss of connection.

3 Prerequisites : ROS & Gazebo

3.1 ROS (Robot operating system)

The Turtlebot3 robot uses a RaspberryPi3 card that we can use as a small computer and where we can execute scripts allowing the robot to move. The orders to the robot must be given from an external computer called Remote PC. When the connections are working properly, the robot executes the orders it receives from the remote PC via protocols that we will detail now.

The Remote PC and the robot communicate through *Topics* : a topic is a message channel on which network *Nodes* can publish or receive messages. Each topic has a well defined message type, and only messages of this type can be interpreted. This node and topic system is only effective if a main server that manages network communications is started, which is the so-called *Master*. This program is executed with the *roscore* command or by running any program *.launch*, which can be run on the Remote PC or on the robot itself. In both cases it is necessary to set the variable ROS_HOSTNAME in the file *.bashrc* to the IP address of the machine carrying the Master.

```
export ROS_HOSTNAME=172.10.20.11
```

For example, there is a default topic named *cmd_vel*, which is used to communicate the speed of the robot. The Remote PC will emit a message containing the speed to be reached on this topic, which the robot can then interpret. The messages are of type *geometry_msgs/Twist*. This could be done, for example, with :

```
rostopic pub -r 10 /cmd_vel geometry_msgs/Twist
'{linear:{x:0.1,y:0.0,z:0.0},␣angular:{x:0.0,y:0.0,z:0.0}}'
```

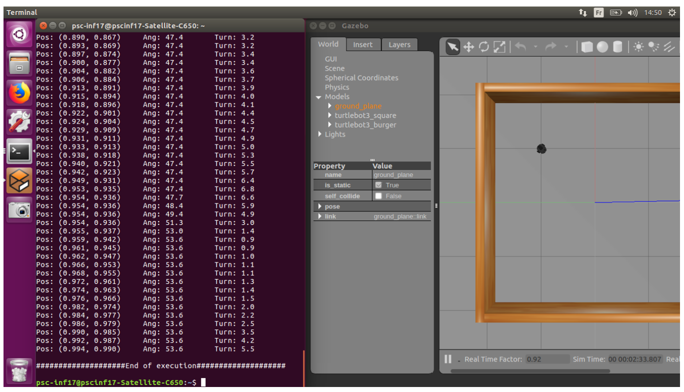
Then, Turtlebot3 has a *.launch* program (specific to ROS) called *turtlebot3_robot.launch*, which is in the package *turtlebot3_bringup*. Once this program is launched in the robot (which can be done via an ssh connection), the robot publishes and subscribes permanently to all necessary topics, including *cmd_vel* and interacts with OpenCR (the robot's motherboard) to move the robot. To launch it, we must specify the ROS_MASTER_URI in the *.bashrc* file so that it knows where the Master is located :

```
export ROS_MASTER_URI=http://172.10.20.11:11311/
```

3.2 Gazebo simulation

We used a graphic simulation of the robot named Gazebo. It is interesting to note that it works in the same way as our real robot. We have to launch the executable *turtlebot3_world.launch* situating in the package *turtlebot3_gazebo* which will allow this simulation tool to publish and/or subscribe to the topics found in *roscore* and interact with the virtual robot on the graphical interface. Thus, we do not need the real robot to test our scripts and we will find exactly the same behavior :

```
roslaunch turtlebot3_gazebo turtlebot3_world.launch
```



```
Pos: (0.890, 0.867)     Ang: 47.4       Turn: 3.2
Pos: (0.893, 0.869)     Ang: 47.4       Turn: 3.2
Pos: (0.897, 0.874)     Ang: 47.4       Turn: 3.4
Pos: (0.900, 0.877)     Ang: 47.4       Turn: 3.4
Pos: (0.904, 0.882)     Ang: 47.4       Turn: 3.6
Pos: (0.906, 0.884)     Ang: 47.4       Turn: 3.7
Pos: (0.913, 0.891)     Ang: 47.4       Turn: 3.9
Pos: (0.915, 0.894)     Ang: 47.4       Turn: 4.0
Pos: (0.918, 0.896)     Ang: 47.4       Turn: 4.1
Pos: (0.922, 0.901)     Ang: 47.4       Turn: 4.4
Pos: (0.924, 0.904)     Ang: 47.4       Turn: 4.5
Pos: (0.929, 0.909)     Ang: 47.4       Turn: 4.7
Pos: (0.931, 0.911)     Ang: 47.4       Turn: 4.9
Pos: (0.933, 0.913)     Ang: 47.4       Turn: 5.0
Pos: (0.938, 0.918)     Ang: 47.4       Turn: 5.3
Pos: (0.940, 0.921)     Ang: 47.4       Turn: 5.5
Pos: (0.942, 0.923)     Ang: 47.4       Turn: 5.7
Pos: (0.949, 0.931)     Ang: 47.4       Turn: 6.4
Pos: (0.953, 0.935)     Ang: 47.4       Turn: 6.8
Pos: (0.954, 0.936)     Ang: 47.7       Turn: 6.6
Pos: (0.954, 0.936)     Ang: 48.4       Turn: 5.9
Pos: (0.954, 0.936)     Ang: 49.4       Turn: 4.9
Pos: (0.954, 0.936)     Ang: 51.3       Turn: 3.0
Pos: (0.955, 0.937)     Ang: 53.0       Turn: 1.4
Pos: (0.959, 0.942)     Ang: 53.6       Turn: 0.9
Pos: (0.961, 0.945)     Ang: 53.6       Turn: 0.9
Pos: (0.962, 0.947)     Ang: 53.6       Turn: 1.0
Pos: (0.966, 0.953)     Ang: 53.6       Turn: 1.1
Pos: (0.968, 0.955)     Ang: 53.6       Turn: 1.1
Pos: (0.972, 0.961)     Ang: 53.6       Turn: 1.3
Pos: (0.974, 0.963)     Ang: 53.6       Turn: 1.4
Pos: (0.976, 0.966)     Ang: 53.6       Turn: 1.5
Pos: (0.982, 0.974)     Ang: 53.6       Turn: 2.0
Pos: (0.984, 0.977)     Ang: 53.6       Turn: 2.2
Pos: (0.986, 0.979)     Ang: 53.6       Turn: 2.5
Pos: (0.990, 0.985)     Ang: 53.6       Turn: 3.5
Pos: (0.992, 0.987)     Ang: 53.6       Turn: 4.2
Pos: (0.994, 0.990)     Ang: 53.6       Turn: 5.5

###################End of execution###################

psc-inf17@pscinf17-Satellite-C650:~$
```

4 Acheivements

The following is what we have been able to produce.

4.1 Scripts allowing us to control the robot's movement efficiently

4.1.1 Contents of scripts

We used the python source code of the *turtlebot3_teleop* package to better understand the syntax for managing topics. We have created a new package named *turtlebot3_automation* in which we wrote the Python scripts to make it easier to move the robot. The main scripts related to the robot's movements we developed are :

- **turtlebot3_func.py**
  All the functions needed to publish, subscribe and control the robot, we use it as a package that we import into all other scripts.

- **turtlebot3_goal.py**
  Script receiving as input the coordinates x and y of the robot's destination. It allows the robot to advance to its destination autonomously. At each moment the robot knows its position and orientation, rectifies if necessary its orientation until it is directed towards its destination, and advances straight to approach it.

- **turtlebot3_strait.py**
  Script receiving an input distance, it moves forward this distance if the input is positive and back otherwise.

- **turtlebot3_turn.py**
  Script receiving as input a number of degrees and rotates the robot by the corresponding number of degrees.

- **turtlebot3_autonomous.py**
  Script running at the time it detects that it has lost the connection, implemented in the robot. Currently the robot just rotates around itself. This is a script to be modified and to be combined with the algorithms implemented in the **Python Simulation** part (the work of the other team).

- **turtlebot3_teleop_key**
  Simplified script from the same one in the package *turtlebot3_teleop* to adapt to the French keyboard and send more useful messages, namely the robot's current position and orientation, instead of its speed.

4.1.2 Performed tests

We tested these scripts not only in simulation but also on the robot itself successfully. Here we assume that the machines are well connected, ROS_HOSTNAME and ROS_MASTER_URI have been well defined in the *.bashrc* file in Remote PC as well as in the robot. We also assume that in the same file we specified :

```
export TURTLEBOT3_MODEL=burger
```

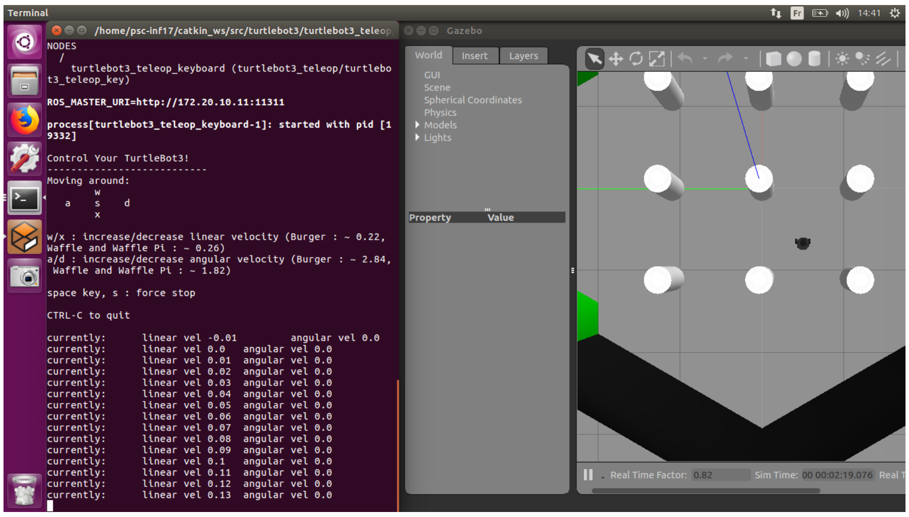
- **Simulation Gazebo (Remote PC)**
  1. Launch a terminal and type *roscore* to launch the Master.
  2. Launch another terminal and type :

     ```
     roslaunch turtlebot3_gazebo turtlebot3_world.launch
     ```

     A Gazebo graphical interface opens and we can observe the robot's movement. Note that we can also just launch other work environments in the package *turtlebot3_gazebo*.
  3. Launch un troisième terminal en tapant par exemple :

     ```
     rosrun turtlebot3_automation turtlebot3_goal.py
     ```

     The algorithm runs after specifying the coordinates x and y. The robot then moves to the desired destination. This also applies to the other scripts we have written.
  4. Observe that the robot reacts correctly on Gazebo.



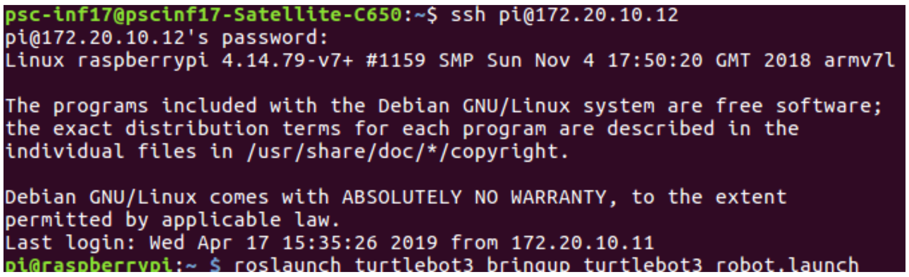
```
NODES
  /
    turtlebot3_teleop_keyboard (turtlebot3_teleop/turtlebo
t3_teleop_key)

ROS_MASTER_URI=http://172.20.10.11:11311

process[turtlebot3_teleop_keyboard-1]: started with pid [1
9332]

Control Your TurtleBot3!
---------------------------
Moving around:
        w
   a    s    d
        x

w/x : increase/decrease linear velocity (Burger : ~ 0.22,
Waffle and Waffle Pi : ~ 0.26)
a/d : increase/decrease angular velocity (Burger : ~ 2.84,
 Waffle and Waffle Pi : ~ 1.82)

space key, s : force stop

CTRL-C to quit

currently:      linear vel -0.01         angular vel 0.0
currently:      linear vel 0.0   angular vel 0.0
currently:      linear vel 0.01  angular vel 0.0
currently:      linear vel 0.02  angular vel 0.0
currently:      linear vel 0.03  angular vel 0.0
currently:      linear vel 0.04  angular vel 0.0
currently:      linear vel 0.05  angular vel 0.0
currently:      linear vel 0.06  angular vel 0.0
currently:      linear vel 0.07  angular vel 0.0
currently:      linear vel 0.08  angular vel 0.0
currently:      linear vel 0.09  angular vel 0.0
currently:      linear vel 0.1   angular vel 0.0
currently:      linear vel 0.11  angular vel 0.0
currently:      linear vel 0.12  angular vel 0.0
currently:      linear vel 0.13  angular vel 0.0
```

- **Real robot**
  We use a carpet with the grid of 2D coordinates. The robot is always placed at the origin $(0, 0)$ and oriented towards the positive direction of the axis of ordinates.
  1. Launch a terminal and type *roscore* to launch the Master from Remote PC.
  2. Two ways to do *turtlebot3_bringup* :
     - Directly on the robot with a screen, a mouse and a keyboard connected, launch a terminal and type :

       ```
       roslaunch turtlebot3_bringup turtlebot3_robot.launch
       ```

     - Via ssh from Remote PC to connect to the robot :

       ```
       ssh pi@172.20.10.12
       roslaunch turtlebot3_bringup turtlebot3_robot.launch
       ```

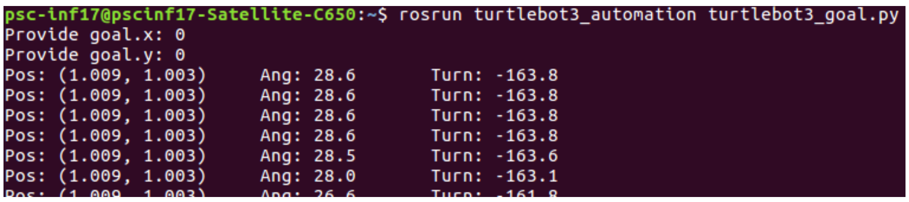


```
psc-inf17@pscinf17-Satellite-C650:~$ ssh pi@172.20.10.12
pi@172.20.10.12's password:
Linux raspberrypi 4.14.79-v7+ #1159 SMP Sun Nov 4 17:50:20 GMT 2018 armv7l

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Wed Apr 17 15:35:26 2019 from 172.20.10.11
pi@raspberrypi:~ $ roslaunch turtlebot3_bringup turtlebot3_robot.launch
```

3. Launch a second terminal from Remote PC and type for example :

```
rosrun turtlebot3_automation turtlebot3_goal.py
```

4. Observe that the robot reacts correctly on the carpet.



```
psc-inf17@pscinf17-Satellite-C650:~$ rosrun turtlebot3_automation turtlebot3_goal.py
Provide goal.x: 0
Provide goal.y: 0
Pos: (1.009, 1.003)     Ang: 28.6       Turn: -163.8
Pos: (1.009, 1.003)     Ang: 28.6       Turn: -163.8
Pos: (1.009, 1.003)     Ang: 28.6       Turn: -163.8
Pos: (1.009, 1.003)     Ang: 28.6       Turn: -163.8
Pos: (1.009, 1.003)     Ang: 28.5       Turn: -163.6
Pos: (1.009, 1.003)     Ang: 28.0       Turn: -163.1
Pos: (1.009, 1.003)     Ang: 26.6       Turn: -161.8
```

4.2 Global control of the robot upon connection and disconnection

4.2.1 Contents of scripts

We had the idea to use Network Managers like *nmcli* or *dhcpcd* to manage Network connection and disconnection on the command line to achieve the desired result. However, *nmcli* we were used to in Gnome interfaces is not the default NetworkManager in RaspberryPi3 and the use of *dhcpcd* would be an extra and tedious job. The connection interfaces must be specified during its use, so the idea was quickly abandoned because we want this project to be as feasible as possible with Li-Fi.

After some research, the easiest way we have found is the use of *ping* as a tool to detect connection and disconnection with the Remote PC. This should work even with the Li-Fi because *ping* ignores the interface we are using and focuses on the connection and disconnection themselves with a specific IP address.

For this we wrote three scripts in *Bash* :

- **autonomous.sh**
  This script is executed after the loss of connection and it aims to launch *turtlebot3_autonomous.py*. However, we met some difficulties during the loss of connection. In fact, once the connection is lost, the IP addresses no longer make sense and must be replaced by *localhost* for the robot to behave autonomously. Thus the server must be *killed* and restarted upon every disconnection, as well as all the programs that depend on it. To do this, we create additional terminals to :

  - *Kill* all programs running on the robot that send messages to the server.
  - Change ROS_MASTER_URI and ROS_HOSTNAME to *localhost*.
  - Launch *roscore* to make the robot become its own Master.
  - Start *turtlebot3_bringup* that submits to the new Master.
  - Start *turtlebot3_autonomous.py*.

- **connectagain.sh**
  This script is executed during the reconnection and continues what *autonomous.sh* was doing. We killed all running programs and create terminals to :

  - Replace ROS_MASTER_URI and ROS_HOSTNAME with the initial IP address.
  - Restart *turtlebot3_bringup* which adapts to the initial Master.

- **checkwifi.sh**
  This script runs permanently on the robot. *While loops* are implemented to continuously ping the Master. Once a connection problem is detected, the script *autonomous.sh* is launched, and we continue to ping the Master to look for reconnection while the robot moves intelligently. If this occurs, the script *connectagain.sh* is launched and the algorithm returns to its initial state.

4.2.2 Performed tests

We connected a screen, a mouse and a keyboard on the robot to simulate the connection and disconnection manually by clicking on the Wi-Fi icon. The robot is initially connected on the same network as the Remote PC where the server MASTER was launched. The following steps show the robot's behavior when it loses its connection :

1. On the Raspberry, launch a terminal and type :
   ```
   rosrun turtlebot3_automation checkwifi.sh
   ```
2. Turn off the robot's Wi-Fi by clicking on the Wi-Fi icon.
3. Observe that new terminals are launched as expected.
4. Observe that all programs running in the robot are killed and that there is no more communication between Remote PC and the robot on both screens, and that the robot continues to execute the last command received.
5. Once the new Master and *turtlebot3_bringup* are ready, the robot stops its wheels and starts to rotate around itself.
6. Reconnect the robot to the initial network.
7. Observe that the programs launched during disconnection were stopped correctly and *turtlebot3_bringup* is launched once again.
8. The robot stops rotating around itself, receives the last command sent by the Remote PC and executes it.
9. Retake control of the robot with the Remote PC.

5 Performance analysis and inconveniences

- **New programs aren't launched immediately**
  A difficulty we couldn't resolve was the fact that launching the server and programs took a considerable time (about 25 seconds), while the robot continues to execute the last received order. At the end of the projet, we found that we can, in fact, avoid changing Master's IP by assigning *raspberrypi* to it in the robot, so then we can avoid launching useless scripts. Now the robot can become autonomous after just a few seconds.

- **Manually launching the script *checkwifi.sh***
  The script *checkwifi.sh* should be run in the robot as soon as it starts. We had the idea to use *crontab* as a tool or to copy this file into */etc/init.d* to automate the launch of this script, but without success. So we still have to start it manually.

- **Inappropriate usage of *killall python* in the script *connectagain.sh***
  killall python is an efficient way to stop all programs using Python. This is the only process we have found that fits our needs, however this is dangerous : if other Python processes were needed, it may lead to many problems.

- **Inappropriate usage of *sleep* in the script *checkwifi.sh***
  The programs must follow a particular order to be launched. The use of the *sleep* that waits for the last program to launch before starting the next one is unreasonable because it is an estimated amount of time and could vary depending on the robot's capabilities, its model and other random factors. We couldn't find an efficient way to ensure one script has finished running before starting the next one.