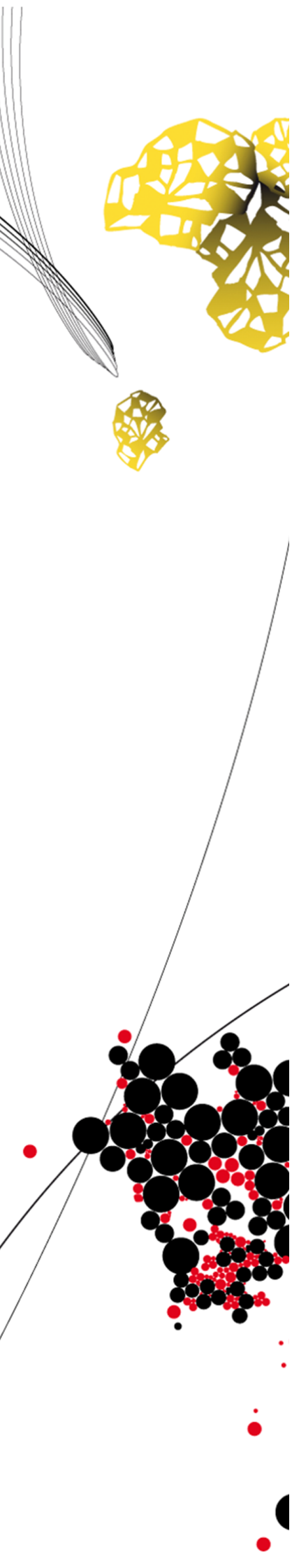

UNIVERSITY OF TWENTE.

Faculty of Electrical Engineering,
Mathematics & Computer Science

Experimental Work (191211249)

Assignment Report

Khaled Alaaeldin Abdelfattah Mustafa
s1918710
September 2018

Supervisor:
M.Sc. Ramy Hashem

Robotics and Mechatronics Group
Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

Contents

Establishing wireless communication between a hexarotor and the ground control station through ROS nodes

Khaled Mustafa

September 19, 2018

1 Introduction

1.1 Motivation

Hexarotors realize one class of unmanned aerial vehicles (UAVs) which are attractive due to their high manoeuvrability, variety of sizes, and ability to perform indoors and outdoors. Recently, research on hexarotor UAVs has grown popular for different applications, such as, e.g., inspection [1], surveillance, environment monitoring and search-and-rescue [2].
As the market develops and the variety of applications expands, more requirements are now being imposed on extending the flight range and increasing the adaptability to the complex missions of UAV systems. In missions such as surveillance and search-and-rescue, UAV systems are usually required to operate in complicated environments while still being able to maintain uninterrupted wireless communication with the ground control station for the purpose of reporting, monitoring, and control. In addition to that, wirelessly driven drones is a technology trend aiming to improve the efficiency and reliability of drones while reducing operational costs from maintenance by decreasing the amount and complexity of its wiring.

1.2 Statement of the problem

Normally, for testing purposes, the development of aerial robots starts by conducting experiments inside the lab for validation of the different system components, e.g. controller, sensor fusion subsystems ...etc. During this stage, the system runs using stationary power suppliers and wired communication with the ground control station as shown in Figure 1.

Figure 1: The hexarotor setup at RAM

However, on the other hand, when it comes to practice, a wireless communication needs to be deployed in order to increase the applicability of the system when it comes to work outdoors. Thus, the main goal of this experimental work is to present an application of wireless communication between multiple ROS nodes running on different machines to unmanned aerial vehicle control. For this reason, a bi-directional wireless communication between the hexarotor and the ground control station is established in order to continuously send information from the on-board sensors to the ground control station and at the same time by sending the control commands from the ground control station to the controller nodes on the micro-computer (odroid) mounted on the hexarotor which in turn controls the motion of the hexarotor through the PX4 flight stack. This work is run as a part of the SPECTORS project at RAM.

To achieve the aforementioned purpose, a wireless communication between a single board computer (Odroid) mounted on a tilted-rotor hexacopter (gamma) and the ground station (master PC) is established. In addition to that there is a serial communication between the Odroid and the flight controller (PX4). The framework of this work is established based on ROS. In order to realize this goal, the structure of this report is organized as follows. In section 2, a brief description of both the Odroid and pixhawk and the interconnection between them is introduced. Afterwards, the algorithm for running ROS on multiple machines is proposed in section 3.

2 Hardware Installation

In this section, the hardware architecture of the system is described. It mainly consists of the master PC that works as the ground control station. In addition to that, a single board computer (odroid) is mounted on the hexarotor and is connected to the pixhawk through a USB cable.
Odroid-XU4 is a single board octa-core (ARM® Cortex® -A15^{TM} up to 2.0GHz) computer that is more powerful than the Raspberry Pi with a 2GB high speed RAM. The boot partition can be stored on either a microSD card or the much faster embedded multi-media controller (eMMC) module that is used for this application. An eMMC module is a type of storage typically used in a smart phone, and is one of the most advanced compact media devices available. There is a boot media selector (slide-switch) from which it is possible to select either to boot from the microSD or the eMMC. The board can run various flavors of Linux, including the latest Ubuntu 16.04 that is currently used.

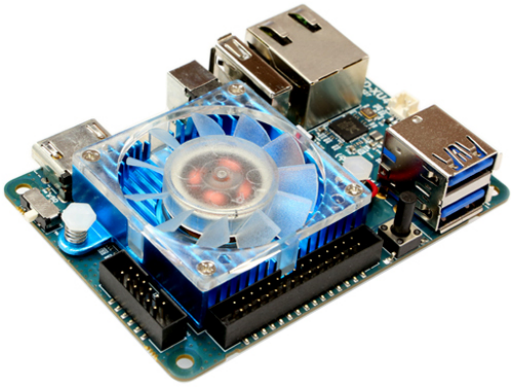

Figure 2: Odroid single board computer

Figure 3: Pixhawk autopilot

There are different types of autopilot hardware that can be used to run the PX4 flight stack. One of theses options is the pixhawk. The Pixhawk flight controller is a further evolution of the PX4 flight controller system. Pixhawk consists of a PX4 controller and a PX4-IO integrated on a single board with additional IO, Memory and other features. It is equipped with advanced sensor profile including gyroscope, accelerometer and a compass for determining orientation.
There are two possibilities for controlling the PX4. The first possibility is through the flight stack computing environment. In this case, it uses the flight controller that is already implemented. On the other hand, the second possibility, is to control the PX4 outside the flight stack through a companion computer (Odroid) or other computing environment (offboard control). In this case, a companion computer is used. This is allowed through the robotics APIs. The communication between the APIs and PX4 is done through the `MAVLink` protocol. For this reason, the `MAVROS` ROS package is used to enable the communication between the PX4 flight stack and the ROS enabled companion computer (Odroid).

There are two ways in which the companion computer (Odroid) can be connected to the autopilot (pixhawk). It can be either through the pixhawk's telemetry port `Telem2` that can be connected to one of the USB ports on the Odroid through an FTDI cable. The other way, that is implemented here, is to connect the USB ports on both the Odroid and the pixhawk by a typical USB cable.

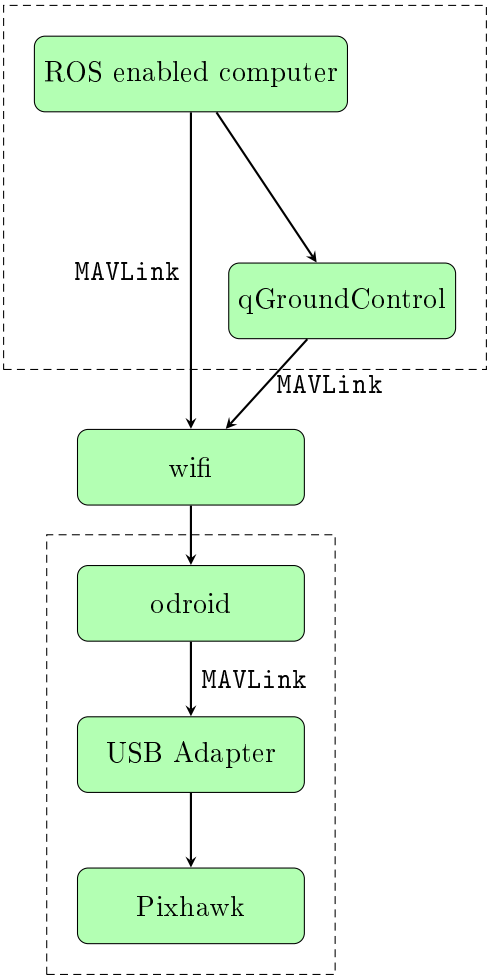

Figure 4: Flow chart illustrating the hardware architecture of the setup

Here it should be noted that `qGroundControl` cannot be installed on Odroid since its operating system is based on ARM architecture. For this reason, the calibration of the pixhawk and the flight settings should be done on the master PC first.

The new software architecture will be as follows. Only the nodes required for the offboard control will be implemented on the Odroid since it is directly connected to the pixhawk. On other hand, the nodes related to the GUI that give the commands to the the hexarotor are implemented on the main PC (master). Here, it should be noted that, due the limited capability of the Odroid's processor, it is advisable to run only the necessary nodes on the Odroid.

3 Software Architecture

In this section, a detailed description of the new adapted software architecture is given. In the first sub-section, the procedures required to run ROS on multiple machines is given. Then, an introduction of the secure shell (SSH) communication protocol is given to allow logging into the remote operating system securely. After that, it is shown how to run GUI programs remotely which is crucial for running gazebo and rviz. Afterwards, the required modification to build package on odroid is given along with how to launch mavros on ARM single board computers. Finally, two methods are provided to run the ROS nodes on the two different machines.

3.1 Running ROS on multiple machines

ROS is a distributed computing environment [3]. Based on that, it is possible that a running ROS system can compromise hundred nodes running across multiple machines. In order to do so, the following requirements need to be satisfied:

(i) All machines must be in the same local network. Thus, both the master and client servers should be connected to the same wifi network.

(ii) It is required to ensure a bi-directional connectivity between all pairs of machines on all ports.

(iii) Each machine must advertise itself by a name that all other machines can resolve.

In order for the hostname to be resolved by other machines, it is required to edit the hosts file on each machine. The host file is the file that tells each machine how to convert specific names into IP addresses. The hostname of a running machine is simply returned back by the linux command `hostname`. To edit the hosts file, the following command should be written in the terminal:

```
$ sudo gedit /etc/hosts
```

The hosts file will open and then it is possible to add the IP address of the other machine followed by its hostname that was returned back by the command `hostname`. For example, in case of the odroid, it should be something like this

```
130.89.231.112 master
```

where 130.89.231.112 is the IP address of the master PC. On the other hand, on the master, it should be something like this

```
130.89.234.31 odroid
```

where 130.89.234.31 is the IP address of the odroid. Here it should be pointed out that the IP address will differ each time the communication to the local network is lost. Thus, it is advisable to check the IP address of both machines and edit the hosts file accordingly. In order for the reader not to get confused, throughout this report, it will be assumed that it is required to run a ROS system on two machines with the following hostnames and IP addresses respectively:

```
master : 130.89.231.112
odroid : 130.89.234.31
```

To make sure that the hosts files were edited correctly, a basic `ping` test is done. Ping is a utility that is used to test the reachability of a host on an IP network by checking that Internet Control Message Protocol (ICMP) packets can get between the machines. Thus, it is needed to ping each machine from the terminal of the other machine as shown below: from master

```
$ ping odroid
```

from odroid

```
$ ping master
```

⚠If there is a problem in the ping, then it means that both machines cannot see each other. Then it is better to check the hosts file to make sure that the IP addresses and hostnames are written correctly.

3.2 Secure shell (SSH) communication protocol

`SSH` or Secure Shell is a protocol which is used to securely log into the remote operating system. It is the most common way to access the remote Linux like server. Thus, it is first required to make sure whether `SSH` is already installed on the operating system or not. This is done by just writing the following command in the terminal window [4]

```
$ ssh
```

If an error occurs by running this command, then it is required to install the `OpenSSH` client by writing the following command

```
$ sudo apt-get install openssh-client
```

By installing it, it is possible to access any operating system on the same local network by just writing

```
$ ssh user@hostname
```

Thus, in this application if it is needed to access the odroid from the master computer, write the following

```
$ ssh odroid@odroid
```

This is to verify if `SSH` client is installed. Now, let's see if the `SSH` server is installed on the operating system which is `OpenSSH` server. This is done by trying to connect to the localhost via the `SSH`. This is done by

```
$ ssh localhost
```

If the following response does appear, then it means that the `OpenSSH` is not installed on the operating system.

```
$ ssh: connect to host 127.0.0.1 port 22: Connection refused
```

Therefore, it is required to install it in order to allow the other operating system (the one on odroid for example) to access the running operating system on the master computer via the terminal. In order to install it, just write

```
$ sudo apt-get install openssh-server
```

In order to check if the `OpenSSH` server is installed correctly, the following command is written

```
$ sudo service ssh status
```

Then if in the given response, it is written that active (running), it means that it was installed correctly. This time, if the command `ssh localhost` is written again, it will be allowed to connect to the running operating system. After installing both `OpenSSH` client and server on both machines, it will be possible to access one machine from the other via `SSH`. If, for some reason, the message indicating the refusal of the connection still exists, then try to force it to reload the configuration by writing the following command in the terminal window

```
$ sudo service ssh force-reload
```

Now, after checking the reachability of the machines on the same local network, it is required to run ROS across the multiple machines. To run ROS across multiple machines, the following requirements should be satisfied:

(i) Only one master is required in order for all nodes to communicate with each other.

(ii) All nodes must be configured in such a way that they use the same master. This is done through the environmental variable `ROS_MASTER_URI`.

`ROS_MASTER_URI` is an environmental variable that is required when the ROS core is not running on the localhost [5]. It is used by ROS nodes in order to locate the master. Now, it will be shown how to run a simple talker and listener nodes on multiple machines using `rosrun` and afterwards it will be shown how to `roslaunch` a launch file on another server which is the case in this application.
As mentioned before, it is required that only one of the machines to be the master. In other words to run the `roscore` command. In this case, it will be the `master` computer. Then a new terminal should be opened and then it is needed to export the `ROS_MASTER_URI` and start the listener node

```
$ export ROS_MASTER_URI=http://master:11311
$ rosrun rospy_tutorials listener.py
```

Then, on the other server (the odroid), it is required to have the same master in order for all nodes to communicate with each other. Therefore, the same `ROS_MASTER_URI` is exported. And the talker node is started.

```
$ export ROS_MASTER_URI=http://master:11311
$ rosrun rospy_tutorials talker.py
```

Now the listener on the master computer should start receiving the messages from the talker node running on odroid. Here it should be pointed out that it is possible to add the export of the `ROS_MASTER_URI` to the /.bashrc file such that it will be exported every time a new terminal is opened without the need to write it again.

3.3 Running GUI programs remotely

In order to access the GUI programs remotely, it is required to enable X11 forwarding option in the `OpenSSH`. X11 is a network protocol to enable remote access to graphical applications. Those graphical applications include gazebo, eclipse, rviz, etc. A machine running an X windowing system can launch a program on a remote computer. All the CPU processing happens on the remote computer but the display of the application appears on the local machine. In order to activate X11 forwarding, it is required to edit the `ssh_config` file through the following command

```
$ sudo nano /etc/ssh/ssh_config
```

Then scroll down till the `Host*` does appear. The only thing that needs to be edited here is to remove the pound sign in front of both `ForwardX11` and `ForwardX11Trusted` and set both of them to yes.

```
Host *
# ForwardAgent no
   ForwardX11 yes
   ForwardX11Trusted yes
```

Then write it out and exit. Thus, now it is done for the client. The same should also be done for the daemon. This is done by

```
$ sudo nano /etc/ssh/sshd_config
```

The only difference between the client and the daemon is to replace `ssh` with `sshd`. Scroll down and make sure that both `X11Forwarding` and `X11DisplayOffset` do not have pound sign in front of them and that `X11Forwarding` is set to yes.

```
X11Forwarding yes
X11DisplayOffset 10
```

Then, again, write it down and exit. After editing `/etc/ssh/sshd_config` file, it is required to restart the `SSH` daemon. For Ubuntu 16.04 this is done by

```
$ sudo systemctl restart ssh
```

Here it should be pointed out that it is important to make sure how the `SSH` daemon is restarted since for some operating systems, the following command is used instead

```
$ sudo service ssh restart
```

After enabling X11 forwarding, the functionality of it, from the master computer for example, can be accessed by typing

```
$ ssh -X master@master
```

In order to make sure that it is allowed to access GUI programs is by typing

```
$ ech $DISPLAY
```

If the below response does appear, then it means that it is done correctly, otherwise it will not be possible to run GUI programs.

```
localhost:10.0
```

Now it is possible to run GUI programs remotely via `SSH`.

3.4 Catkin build on Odroid

Due to the limited capability of the single board computer (Odroid) compared to the main computer (master), some problems might occur while catkin build ROS packages. This is especially related to its RAM size. While building ROS packages, if the number of jobs are not specified explicitly by `-jn` for `n` jobs when calling `catkin build`, catkin gets the number of jobs from the `ROS_PARALLEL_JOBS` environmental variable. `-jn` specifies the number of jobs (commands) to run simultaneously. Excessive parallelism in a large build can exhaust system memory.
Thus if the `catkin build` command is being used on Odroid without limiting the number of jobs, most probably it will result in an internal compiler error due to the shortage of RAM size. Thus it is not possible to run various compilers in parallel This is the type of error that is expected to appear

```
g++-4.8.real: internal compiler error: Killed (program cc1plus)- it seems
    that there is not enough RAM
```

Another issue that may arise is the increase of the temperature of the Odroid causing it to shut down when it reaches the maximum temperature. A possible solution to tackle this problem is to limit the number of threads being used throughout. In this case, it might take longer to `catkin build` the packages but it will be assured not to get an internal compiler error nor an overheat of the Odroid. This is done by simply running the following command

```
catkin build -j1
```

In this case, only one thread will be run at a time. Here it should be noted that every time it is needed to build ROS packages on Odroid, it is highly recommended to replace `catkin build` with `catkin build -j1`. Another way to limit the number of threads being run is to adjust the environmental variable `ROS_PARALLEL_JOBS` since by default `catkin build` gets the number of jibs from it as mentioned before. This is done by

```
export ROS_PARALLEL_JOBS=-j1
```

But the first method is recommended.

3.5 Launching mavros on ARM single board computers

When trying to `roslaunch` the mavros launch file on Odroid, an error due to segmentation fault does appear causing `mavros_node` to die. This error actually doesn't appear when launching mavros on the master computer. Thus it is due to the difference in the operating system architecture since Odroid is based on ARM architecture.

```
[mavros-2] killing on exit
[rosout-1] killing on exit
[master] killing on exit
shutting down processing monitor...
... shutting down processing monitor complete
done
```

It turns out that this error is due to the `HIL` (hardware in the loop) plugin. A possible solution to avoid this error is to blacklist the `HIL` plugin in the `px4_pluginlists.yaml` inside the launch folder of mavros package to prevent the segfault error.

```
plugin_blacklist:
# common
- safety_area
# extras
- image_pub
- vibration
- distance_sensor
- rangefinder
- hil
plugin_whitelist: []
# 'sys_*'
```

By doing so and then relaunching `px4.launch` file, it should work without giving further errors. Here it should be pointed out that this problem will not occur if the `RotorS` launch file is used instead of the `px4` since `HIL` plugin is blacklisted in the `apm_pluginlists.yaml` file by default.

3.6 Running a launch file on a remote node

This section describes the procedures and configurations required, when a distributed system which consists of machines running ROS is developed. There are two different ways to `roslaunch` a file on a remote node (machine). Although on of these methods was not implemented successfully for some reason, both methods will be discussed into details in this section.

Method I

The first method is mainly based on the `<machine>` tag and `SSH` protocol. The `<machine>` tag declares a machine on which ROS nodes can run on. This tag is not needed if all the

nodes are launched locally. The `<machine>` tag is mainly used to declare `SSH` and ROS environmental variables settings for the remote machines.The attribute of the `<machine>` tag are the name, user and the IP address of the remote machine. In addition to that there is an `env_loader` file included in the launch file that tells ROS to run the default ROS setup on the remote machine (Odroid). An example of a simple launch file is shown below that is used just to launch a listener and talker nodes on the local (master) and remote (Odroid) machines respectively. This is the launch file attempted to be used on the master machine:

```
<launch>
  <node name="listener" pkg="agitr" type="listener" output="screen" />
  <group>
  <machine name="odroid" address="130.89.234.31" user="odroid"
      env-loader="/odroid/catkin_ws/envloader.sh" default="true" />
    <include file="$(find_agitr)/launch/test.launch" />
</group>
</launch>
```

where `test.launch` includes the talker node. Here it should be pointed out that the `<group>` tag is used in order to be able to include the launch file on the remote machine (Odroid). Thus, it is used as a container for the tags within. Due to the automated nature of `roslaunch`, it avoids executing the users `.bashrc` on remote nodes and requires an environment file to assign remote environment variables. As mentioned before,the `envloader.sh` file on Odroid should declare the ROS environmental variables including `ROS_MASTER_URI` that should have the same IP address of the master. An example of the env-loader script which is used in this case is:

```
#!/bin/bash

source /opt/ros/kinetic/setup.bash
source /home/khaled/catkin_ws/devel/setup.bash
export ROS_IP=130.89.234.31
export ROS_MASTER_URI=http://130.89.231.112:11311
export ROS_HOSTNAME="odroid"
exec "$@"
```

Save the environmental file on the remote machine (odroid) in the same path as the one written in the launch file which is, in this case, `/odroid/catkin_ws/envloader.sh`. Then, make sure that the `ROS_MASTER_URI` environmental variable is exported on the local machine (master).

```
export ROS_MASTER_URI=http://130.89.234.112:11311
```

Finally, launch the launch file `response.launch` on the local machine under the launch file directory

```
roslaunch agitr response.launch
```

By applying this method in our application, then there will be only one launch file on the local machine (master) that include all other launch files that need to be run. The launch files that should be run on the remote machine (Odroid), for example the controller node, should be written under the `<machine>` tag.

Method II

The second method, the one that was successfully implemented, is based on the splitting of the launch file into two launch files, one on the local machine (master) and the other on

the remote machine (Odroid). For this specific case, the launch files that are run on the local machine is the GUI node.

```
<?xml version="1.0"?>
<launch>

    <!--***************** Arguments **************-->

    <!-- UAV's Name -->
    <arg name="vehicle" default="betaX"/>

    <!--***************** Run GUI Node **************-->
      <include file="$(find spc_uav_comm)/launch/gui_omni_uav.launch">
        <arg name="uav_type" value="hexacopter" />
        <arg name="flightMap" value="ramlab"/>
    </include>
</launch>
```

On the other hand, the controller and simulation nodes are implemented on the remote machine (Odroid).

```
<?xml version="1.0"?>
<launch>

    <!--***************** Arguments **************-->

    <!-- UAV's Name -->
    <arg name="vehicle" default="betaX"/>
    <!-- Gazebo World FileName -->
    <arg name="world" default="$(find
        spc_uav_simulator)/worlds/wall.world"/>

    <!--***************** RUN SE3 Controller Node for rotorS simulation
        **************-->
    <include file="$(find spc_uav_control)/launch/SE3_controller.launch">
        <arg name="vehicle" value="$(arg vehicle)" />
                <arg name="uav_type" value="tilt_hexarotor" />
        <arg name="use_mavlink_interface" value="false" />
        <arg name="use_experiment_interface" value="false" />
        <arg name="dist_obs_type" value="1" />
        <arg name="odometry_topic" value="/ground_truth/odometry" />
        <arg name="dyn_reconfig_load" value="$(find
            spc_uav_control)/config/dynamic/rotors/SE3_Control_Gains_$(arg
            vehicle).yaml" />
    </include>

    <!--***************** RUN rotorS Simulation **************-->
     <include file="$(find spc_uav_simulator)/launch/rotors_gazebo.launch">
        <arg name="vehicle" value="$(arg vehicle)" />
        <arg name="world" value="$(arg world)" />
    </include>

</launch>
```

Then, from the local machine (master), two terminals are to be opened. From the first terminal, it is required to export the environmental variable `ROS_MASTER_URI`

```
export ROS_MASTER_URI=http://130.89.234.112:11311
```

Then `roslaunch` the launch file on the local machine that will launch the GUI node.

```
roslaunch spc_uav_systems offboard_demo_10.launch
```

Here it should be pointed out that it is necessary to `roslaunch` the launch file on the master machine first, otherwise, an error will occur when launching the file on the Odroid indicating that it is not possible to connect to the `roscore` running on the master computer. As a result, now, the GUI node should open on the local machine (master). Then, open the second terminal and it is required to `SSH` to the remote machine (Odroid)

```
ssh odroid@odroid
```

Then in order to be able to `roslaunch` any launch file on the Odroid, it is required to run the `rosmaster`. As mentioned before, there should be only one `rosmaster`, thus it is required to export the same `ROS_MASTER_URI` on the Odroid

```
export ROS_MASTER_URI=http://130.89.234.112:11311
```

By doing so it is now possible to `roslaunch` any launch file on the Odroid from the local machine using the processor capability of the Odroid. Thus by launching the launch file, gazebo will open on the local machine but using the processor power of the Odroid.

```
roslaunch spc_uav_systems offboard_demo_10.launch
```

As a result by adjusting the setpoint of the hexarotor from the GUI node, it will move to the same point in the gazebo animation indicating that the wireless communication is done successfully.

4 Conclusion

In this report, the experimental work for establishing a wireless communication between the ROS nodes running on the ground station computer (master) and the companion computer mounted on the hexarotor (Odroid). In addition to that the serial communication between the Odroid and the PX4. The entire interface was implemented using ROS.The communication was tested using the simulation launch file and it works successfully. This report summarizes all the necessary installations and steps that are required to establish this communication.

References

[1] L. Marconi et al., "Aerial service robots: An overview of the AIRobots activity,". in *Proceedings of the IEEE International Conference on Applied Robotics for the Power Industry*, 2012, pp. 76–77.

[2] —, "The SHERPA project: Smart collaboration between humans and ground-aerial robots for improving rescuing activities in alpine environments," in *Proceedings of the IEEE International Symposium on Safety, Security, and Rescue Robotics, 2012.*

[3] ROS Documentation,
`http://wiki.ros.org/ROS/Tutorials/MultipleMachines`

[4] ROS Documentation,
`http://wiki.ros.org/ROS/NetworkSetup`

[5] ROS Documentation,
`http://wiki.ros.org/ROS/EnvironmentVariables`