Development of a redundancy system for autopilots

Thesis

Submitted to the Faculty of Fachhochschule Aachen, Germany
and Centro de Ingenieria y Desarrollo Industrial, Mexico

BY

M.E. Fernando Fidencio Solano Pérez

SUPERVISOR:

Dr. Antonio Estrada-Torres.

In Partial Fulfillment of the requirements for the degree
of Master of Science in Mechatronics

Santiago de Querétaro, Qro., México, February 2019

Declaration of Authorship

Hereby, I, Fernando Fidencio Solano-Perez, declare that this thesis "Development of a redundancy system for autopilots" is the result of my own work. Any part of this dissertation has not been previously submitted, in part or whole, to any university or institution for a degree or other qualification.

I confirm that all consulted work from others is attributed and the source is always given. Furthermore, the data and the software employed have all been utilized in complete agreement to the copyright rules of concerned establishments.

Signed: ______________________________

Santiago de Querétaro, México, February 2019

Acknowlegments

I would like to thank México and CONACYT for having gave me the opportunity and required funds to study a double degree program at both Centro de Ingeniería y Desarrollo Indutrial and the University of Applied Sciences FH Aachen.

I express my sincere gratitude to Germany and its people since all along the time I spent in Aachen was nothing less than a unique experience.

My sincere gratitude to both my advisor Dr. Antonio Estrada-Torres and Dr. Gengis K. Toledo-Ramirez, for their encouragement, insightful comments and for having gave me the opportunity to develop my thesis with them. My sincere thanks also goes to Prof. Dr. rer. nat. Klaus-Peter Kämper and Dr. Eng. Jörg Wollert for the knowledge which they shared with me along their lectures.

Nobody has been more important to me in the pursuit of my dreams and goals than my Family. I dedicate this work to them, my parents and siblings because they are the only four people in my life that I want to make the most proud of me.

Last but not the least, my deepest and most sincere gratitude to those who throughout their lives have shared their knowledge with the sole purpose of making the world a better place to live in ... for this and many other things thank you so much.

Abstract

The redundant autopilot hereby presented is aimed to enhance drone reliability and safety. This redundant autopilot is based on two Pixhawk autopilots and a companion computer which supervises the correct operation of the complete system. If one of the autopilots should fail, there is always a redundant autopilot to take over the control of the drone and keep it in the air.

The companion computer, runs Ubuntu 16.04, acts as supervisor, deciding which autopilot should control the drone. The communication between autopilots and the companion computer is made over the already widespread MAVLink protocol and, as a result, the redundant system is compatible with all enabled MAVLink autopilots that may exist within the market. Furthermore, this project makes use of the Robot Operating System (ROS) to acquire and process data coming from the autopilots, so that any drone can be enhanced by taking advantage of the already existing collection of tools and packages from the ROS community.

In addition, an extra computer can be used as a ground station. This computer can be then used as an extra means of controlling the drone when the RC controller is not available. These characteristics, along with the ones that ROS provides, make this a flexible and highly scalable project, which opens the opportunity to develop further projects with an enhancement in safety and reliability.

Kurzfassung

Der hier vorgestellte redundante Autopilot soll die Zuverlässigkeit und Sicherheit der Drohnen verbessern. Dieser redundante Autopilot basiert auf zwei Pixhawk-Autopiloten und einem Computer, der den korrekten Betrieb des gesamten Systems überwacht. Wenn einer der Autopiloten ausfallen sollte, gibt es jederzeit einen redundanten Autopiloten, der die Kontrolle über die Drohne übernimmt und sie in der Luft hält.

Der Companion-Computer, der das Betriebssystem Ubuntu 16.04 ausführt, entscheidet, welcher Autopilot die Drohne kontrollieren soll. Die Kommunikation zwischen Autopiloten und dem zugehörigen Computer erfolgt über das bereits weit verbreitete MAVLink-Protokoll. Daher ist das redundante System mit allen auf dem Markt erhältlichen MAVLink-Autopiloten kompatibel. Darüber hinaus nutzt dieses Projekt das Robot Operating System (ROS) zur Erfassung und Verarbeitung von Daten aus dem Autopiloten. So kann jede Drohne durch Nutzung der bereits vorhandenen Sammlung von Tools und Paketen der ROS-Community verbessert werden.

Des Weiteren kann ein zusätzlicher Computer als Bodenstation verwendet werden. Dieser Computer kann dann als zusätzliches Mittel zur Steuerung der Drohne verwendet werden, wenn der RC-Controller nicht verfügbar ist. Diese Eigenschaften sowie die von ROS bereitgestellten Merkmale machen dieses Projekt zu einem flexiblen und hoch skalierbaren Projekt, das die Möglichkeit eröffnet, komplexere Projekte mit einer verbesserten Sicherheit und Zuverlässigkeit zu entwickeln.

Resumen

El piloto automático redundante aquí presentado tiene como objetivo mejorar la confiabilidad y seguridad de los drones, este piloto automático redundante se basa en dos pilotos automáticos Pixhawk y una computadora complementaria que supervisa el funcionamiento del sistema completo. Si uno de los pilotos automáticos fallara, siempre habra un piloto automático redundante para tomar el control del drone y mantenerlo en el aire.

La computadora compañera, que corre Ubuntu 16.04, actúa como supervisor y decide qué piloto automático debe controlar el drone. La comunicación entre los pilotos automáticos y la computadora de compañía se realiza a través del ya muy extendido protocolo MAVLink. Como resultado, el sistema redundante es compatible con todos los pilotos automáticos MAVLink habilitados que puedan existir dentro del mercado. Además, este proyecto utiliza el sistema operativo de robots (ROS) para adquirir y procesar datos provenientes de los pilotos automáticos. Debido a esta razón, cualquier drone puede mejorarse aprovechando la colección ya existente de herramientas y paquetes de la comunidad ROS.

Además, se puede utilizar una computadora adicional como estación de tierra. Esta computadora se puede usar como un medio adicional para controlar el drone cuando el radio control no está disponible. Todas estas características, junto con las que proporciona ROS, hacen de este un proyecto flexible y altamente escalable, lo que abre la oportunidad de desarrollar proyectos mucho más complejos con una mejora en seguridad y confiabilidad

Contents

Declaration of authorship . . . i
Acknowlegments . . . iii
Abstract . . . v
Kurzfassung . . . vii
Resumen . . . ix
List of figures . . . xvi
List of tables . . . xviii
Acronyms . . . xix

1 Introduction **1**
1.1 Motivation . . . 6
1.2 Hypothesis . . . 6
1.3 Justification . . . 6
1.4 Objectives . . . 7
1.4.1 General objective . . . 7
1.4.2 Specific objectives . . . 7
1.5 Methodology . . . 7

2 State of the art **11**
2.1 Redundant Autopilots . . . 11
2.1.1 MicroPilot MP21283 XM . . . 11
2.1.2 Veronte Autopilot . . . 12
2.1.3 Cerberus Triple Redundant Autopilot . . . 13
2.1.4 AP-Manager . . . 14
2.1.5 Redundant Okto board . . . 15
2.1.6 Autopilot A3 PRO . . . 15
2.2 Redundant Drones . . . 16
2.2.1 Euroavionics multicopter . . . 16
2.2.2 AscTec Trinity Drone . . . 17

2.2.3 Matrice 600 Pro 18

3 Project development **20**
3.1 System definition 20
3.2 Selected elements 24
3.2.1 Flight Controller 24
3.2.2 PX4 autopilot 25
3.2.3 Companion Computer 25
3.2.4 Robot Operating System (ROS) 25
3.2.5 QGroundControl 26
3.3 Detailed design 27
3.4 The redundant system 29
3.4.1 Data streamed flow between the companion computer and the Autopilots 29
3.4.2 Data coming from Autopilots 30
3.4.3 Communication setup 31
3.4.3.1 Companion computer 31
3.4.3.2 Pixhawk Setup 31
3.4.3.3 Hardware Setup 32
3.4.4 Behind the Logic 32
3.4.5 ROS packages 37
3.4.6 Bridge Circuit 40

4 Simulation and test process **43**
4.1 HIL vs SIL 43
4.2 SIL phase. 44
4.2.1 How the simulation works. 44
4.2.2 Communication between PX4 and Gazebo. 45
4.2.3 SIL Simulation Environment 45
4.2.4 Logic for SIL. 46
4.3 HIL phase. 49
4.3.1 HIL environment 51
4.4 Setting up the SIL 54
4.4.1 Launch files 54
4.4.2 Launch files created for the project 55
4.4.3 Extra information 57
4.4.4 Running a simulation with launch files 58

4.5 Setting up the HIL 58
4.5.1 Hardware setup 60
4.5.2 Software setup 60
4.5.2.1 Pixhawk 60
4.5.2.2 Companion computer 61
4.5.2.3 Ground Station Computer 62

5 Results **65**
5.1 Communication with autopilots 65
5.2 Data transmission 66
5.2.1 QGroundControl 66
5.2.2 Ground Control Station 66
5.3 Algorithm 69
5.3.1 SIL 69
5.3.2 HIL 70

6 Conclusion **72**
6.1 Extra comments 73
6.2 Further work 73

Bibliography **80**

Appendices **81**

A Chronogram of activities **83**

B Installing the development tools **84**
B.1 Installing the development toolchain 85
B.2 Extra information 87

List of Figures

1.1 Timeline of the military and civilian uses of drones, taken from [1]. 1
1.2 Common drone's components, taken from [2] 3
1.3 CIDESI Systems Engineering's V-Model, taken from [3] 9

2.1 MP21283X autopilot, taken from [4] 12
2.2 Veronte triple redundant Autopilot, taken from [5] 13
2.3 Cerberus triple redundant autopilot, taken from [6] 13
2.4 AP-Manager, redundant board for autopilots. Taken from [7] 14
2.5 Redundant Okto board, take from [8] 15
2.6 A3 pro Autopilot, taken from [9] 16
2.7 Euroavionics multicopter, taken from [10] 17
2.8 AscTec Trinity Drone, take from [11] 17
2.9 Matrice 600 Pro Drone, taken from [12] 18

3.1 First system concept 23
3.2 Pixhawk Autopilot 24
3.3 Rasberry Pi as a Companion computer 25
3.4 QGroundControl for planning autonomous flights 26
3.5 Drone Architectural Overview 28
3.6 System data flow 29
3.7 Diagnostics message 30
3.8 Redundant Autopilot sequence diagram. 34
3.9 Ground computer Menu 35
3.10 Control control sequence diagram 36
3.11 ROS computation graph 39
3.12 Bridge circuit connection 40

4.1 MAVlink API 44
4.2 Software in the Loop (SIL) setup 46

4.3 Simulation environment . . . 47
4.4 Test for /diagnostics . . . 48
4.5 Test connection with QGroundControl . . . 49
4.6 HIL setup . . . 50
4.7 Network setup . . . 50
4.8 HIL setup and components connection. . . . 51
4.9 First test for communication . . . 52
4.10 Second test for communication . . . 53
4.11 Firmware, MAVROS and MAVlink folder in SRC workspace folder . . . 54
4.12 Example default PX4 Lauch files . . . 55
4.13 my_main_launch.launch launch file . . . 56
4.14 my_uav1_mavros_sitl.launch launch file . . . 56
4.15 Running Simulation . . . 58
4.16 FDTI module to translate from serial to USB protocol . . . 59
4.17 Serial connection between Pixhawk and Companion computer . . . 59
4.18 Configuring parameter using QGroundControl . . . 61
4.19 stablish_autopilots_connection.launch file to stablish connection with MAVROS 63

5.1 Data obtained from autopilots . . . 65
5.2 ROS starts action server and wait for a call from ground station. . . . 67
5.3 Call from Ground station and response from Companion computer . . . 67
5.4 Menu from Ground station . . . 68
5.5 Warning obtained when one autopilot has an error and connection between Ground station and companion Computer is made. . . . 68
5.6 Drone starts service in Case Emergency and waits for a failure. . . . 69
5.7 Simulating and detecting error in one Autopilot . . . 69
5.8 Test for HIL phase . . . 70

A.1 Chronogram of activities . . . 83

B.1 Folders /catkin_ws and /src/Firmware created in the Home directory. . . . 85
B.2 Folder /catkin_ws/src . . . 86
B.3 Simulation launched . . . 86

List of Tables

1.1 Main components of a Drone 2

3.1 Redundant Autopilot characteristics 21
3.2 Pixhawk Autopilot Features 24
3.3 QGroundControl Features 26
3.4 Flags used to check Autopilots status. 30
3.5 Wiring to Pixhawk (FTDI Chip USB-to-serial adapter board) 32

4.1 Gazebo characteristics 44
4.2 Wiring to Pixhawk (FTDI Chip USB-to-serial adapter board) 60

5.1 Data coming from each autopilot 66

B.1 OS for development 84

Acronyms

UAVs	Unmanned Aerial Vehicles
GCS	Ground Control Station
ESC	Electric Speed Controller
GPS	Global Position System
HIL	Hardware in the loop
GNSS	Global Navigation Satellite System
ROS	Robot Operating System
FTDI	Future Technology Devices International
SIL	Software in the Loop
RTPS	Real Time Publish Subscribe

Chapter 1

Introduction

Over the past years the use of drones or Unmanned Aerial Vehicles (UAVs), have become an outstanding emergent technology which has already opened opportunities to transform and alter several industries, and in the process, change our attitudes and behaviors regarding their impact on our daily lives. The definition of UAVs is rather extensive due to the large existing configuration that already exist in the market. In practice, any aerial vehicle that does not rely on an on-board human operator for flight, either autonomously or remotely operated, is considered a UAV [13]. The emergence of drones challenges traditional notions of safety, security, privacy, ownership, liability, and regulation [14].

Although drones were preconceived as targets for military practice, mostly in the U.S [1], they have been introduced to civilian marked and adopted in a short time (Fig. 1.1). This situation has already triggered concerns that need further attention.

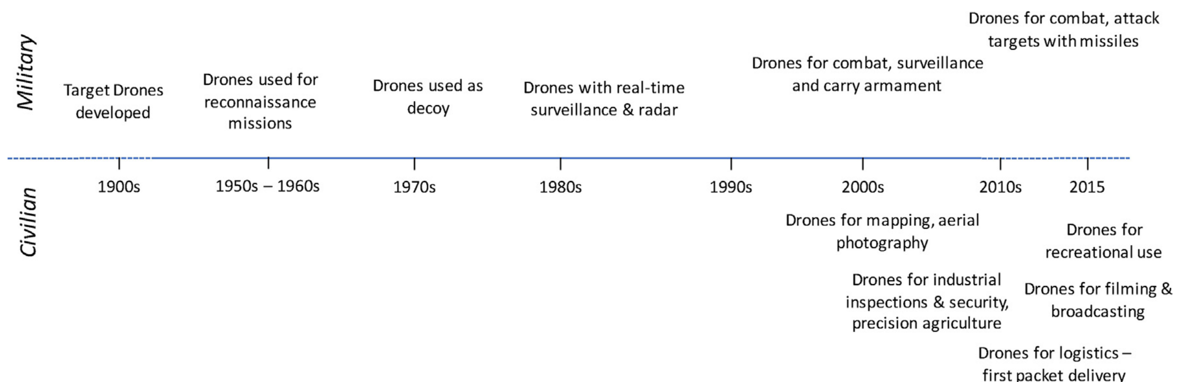

Figure 1.1: Timeline of the military and civilian uses of drones, taken from [1].

The primary criticism with the flying of commercial drones is that small mistakes could result in crashes that threaten the health, well-being and property of the public. Further-

more, if they crash into airports or other protected airspaces, it could result in dangerous scenarios that put lives in danger [14]. The impact can lead to explosions or fires, resulting in further damage. Furthermore many drones have rapidly moving parts, in the form of propellers, which are capable of causing much more substantial physical and mental trauma than the drone's mass and velocity alone suggest [15]. An out of control drone may surprise individuals in its vicinity, in some cases leading to accidents e.g. where a driver of a motor vehicle, or the pilot of another drone, loses control of their vehicle or performs a dangerous avoidance maneuver [16]. Even though drones have benefited of technological development from the twenty-first century, leading them to be small and inexpensive devices, they are not yet reliable enough to be considered completely safe devices [17].

In order to improve their reliability, it is required to focus attention on each of the main components, programming and circumstances under which they operate. All drone parts and components are vital for a safe flight. Drones in the market may have different configurations and be equipped with different types of sensors, but all of them have in common specific components that are essential to take flight and keep the drones in the air (see Table 1.1 and Fig. 1.2).

Table 1.1: Main components of a Drone

Component	Description
Propellers	The purpose of the propellers is to generate thrust and torque to keep the drone flying, and to maneuver.
Brushless Motors	The motors have to spin faster to create lift. All drones being manufactured lately use the brushless motors that are considered to be more efficient in terms of performance and operation.
Electronic Speed Controllers	It is an electronic circuit used to change the speed of an electric motor, its route and also to perform as a dynamic brake.
Autopilot and flight controller	A complete system that enables a drone to fly autonomously and keeps your aircraft stable.
Telemetry module	Telemetry is what is use to send and receive data between a drone and a ground station.
Battery.	Power source from all electronic devices.

Drones in practice, generally exhibit at least some degree of autonomy. Functions such as stabilization of altitude and latitude are delegated to electronic components [18]. As a result safety mechanisms must be applied in order to minimize risk of failures from autonomous components [19].

Figure 1.2: Common drone's components, taken from [2]

The autonomous operation generally falls on the so called autopilot, which can provide fully-autonomous and semi-autonomous operation, based on pre-programmed flight plans or more complex dynamic automation systems [20]. Nowadays there is available a vast number of autopilots from different companies and most of them can run one of the two most common softwares that controls the autopilot.

The two currently most common softwares used are PX4 and Ardupilot, both of them are open source flight control software for drones and other unmanned vehicles. The importance of autopilots is such that its use has been standardized among manned aircrafts and high performance UAVs, and as it can be expected they have several safe mechanisms to avoid compromising its proper operation, such as redundancy.

Redundancy is a common approach to improve the reliability and availability of a system. Adding redundancy increases the cost and complexity of a system design but with the high reliability of modern electrical and mechanical components, many applications do not need redundancy in order to be successful [21] [22]. However, regarding drone matters where a failure can implicate serious damages to facilities and people in general the cost of failure can be high enough to consider redundancy an attractive option.

System reliability can be improved in a desired manner by applying redundancy. There are various approaches and techniques for implementing redundancy, the following models represent the more common ones used in industry. The three main models are Standby Redundancy, N Modular Redundancy, and 1:N Redundancy [23].

The standby redundancy is an arrangement where a secondary identical unit works as a back up for the primary unit. If only one of all units which perform the required function is active and the rest of the units is inactive (they are waiting to be active if the main unit fails), this system is called a standby redundancy system [24][25]. The spare unit usually do not monitor the system before being active, it tries to recover from the last control signal that was left from the primary unit. The system cost increases about twice or less depending on the software development application [23]. In Standby redundancy there are three types: hot-Standby, cold-Standby and warm-standby.

For hot-standby redundancy, the standby components are always active and suffer from the same operational stress and have the same failure rate with the already working component. While for cold-standby redundancy, the standby components are unpowered and

shielded from the operational stress, due to this the failure rates of cold-standby components are usually supposed to be zero before being activated [26][25]. Compared with hot-standby redundancy, cold-standby needs less energy consumption and lower cost but suffers from longer restoration delay when activating the standby components. The warm-standby redundancy is an intermediate case between hot-standby and cold-standby.

The next model is the N modular redundancy, also sometimes called parallel redundancy [23]. It is a fault-tolerant form of N-modular redundancy, in which two or more units monitor a process, they are highly synchronized and their respective output or result is processed by a majority-voting system to produce a single output, this last one decides which unit should take over the control of the system when switching between units. If any one of the spare units fails, the remaining ones can correct and diminish the fault [27] [28] [29].

Deciding which unit is healthy can be challenging if there is only two units to compare. Sometimes it is only a matter of just choosing one to trust the most and this can get complicated. If there is more than two units the problem is simpler, usually the majority wins or the two that agree win [30] [31]. In N Modular redundancy, there are three main typologies: Dual modular redundancy, Triple modular redundancy, and Quadruple redundancy [23].

Redundancy 1:N is a design technique used where you have a single backup for multiple systems and this backup is able to function in the place of any single one of the active systems. This technique offers redundancy at a much lower cost than the other models by using one standby unit for several primary units [23].

The redundancy can be use in drones for the duplication of critical components, such as the autopilot, with the intention of increasing reliability of the system, in the form of a backup to improve actual system performance [32]. Nowadays, the use of redundancy in civilian drones is not widespread but this trend is changing due to the normalization of drones [33].

As the use of drones continues to proliferate, its technology still have a long evolutionary path ahead, along with regulation matters that need further attention. It is expected that drones will be part of our everyday life, just as smartphones are today, they will impact industries ranging from entertainment to agriculture, from construction to delivery market.

The First chapter presents an introduction to the topic, followed by the project motivation, objectives, hypothesis and methodology. The Second Chapter presents a brief exploration about the state of the art. The Third Chapter shows a detailed system design, going from the components selection to the system operation, focusing special attention on the redundant system. The Fourth and Fifth Chapter include information regarding the simulation, tests and the outcomes achieved from the respective phases. Finally the last Chapter shows the author's conclusion and suggestions for further work.

1.1 Motivation

Although nowadays drones have fail-proof mechanisms, with automatic routines such as return home or emergency landing, that are activated after a system failure, these systems are not completely safe and sometimes flights suffer some kind of error, resulting on the need for many drones of a redundant autopilot able to lower risk of failure [19][15][16].

Drones with redundancy safe systems are not yet standardized in the civilian market but it is clearly that this will change in the near future. Governments of different countries are focusing attention and are planning to make law enforcement regarding them [34].

Adding redundancy to drones autopilot will improve their reliability and lower possibilities of harming people. Moreover it is important to improve their technology since drones are becoming increasingly important in the field of science, technology, and society [14].

1.2 Hypothesis

If redundant Autopilot arrangement is implemented on a Drone, reliability of the system will increase and possibilities of a crashes will be lowered.

1.3 Justification

Safety is one of the most important factors when flying UAVs. Safety requirements established for civil aviation authorities, regarding commercial and professional drones operations, are becoming more stringent every day [16] [14]. Mexican government shows an example regarding UAVs law enforcement, since by the end of year 2018 Secretariat of Communications and Transportation of Mexico will start normalizing drones [35].

This comes as a direct consequence of the need for drones to operate in populated areas and due to new applications focused on the civilian market such as delivery, surveillance, inspection, among other [34].

Taking these facts into account it is reasonable to add safety mechanisms to increase the reliability of Drones avoiding possible accident that may cause injuries to the user and public in general that could be in the surrounding areas where drones operate.

1.4 Objectives

1.4.1 General objective

Development of a double redundant autopilot for drones.

1.4.2 Specific objectives

- Development of a double redundant PX4 Autopilot for drones using ROS platform.
- Establish a concept design.
- Simulation of the double redundant autopilot using Gazebo.
- Development of a ROS node that ensures the communication between two PX4 autopilots.
- Implementation of an algorithm that oversees the correct operation of each individual autopilot.
- Test and validation of the simulation by implementing a Hardware in the loop stage.
- Thesis dissertation.

1.5 Methodology

The methodology followed on this thesis is based on the so called V model which is a process model that was originated from software development and introduced as guideline by the German Government, Fig. 1.3. Because the V Model requires a permanent validation and verification of the project, dependability analysis is constantly performed throughout the project [36]. V model as a standardized and widely applied process model seems to

be a good starting point.

The projects development is divided into two main parts, namely verification phase on one side of the 'V' and validation phase on the other side. Evaluation of the redundant autopilot is done at verification phase in order to specify the requirements that have to be accomplished. At the validation phase an evaluation is done to determine if the redundant autopilot meets the initial expectations and requirements.

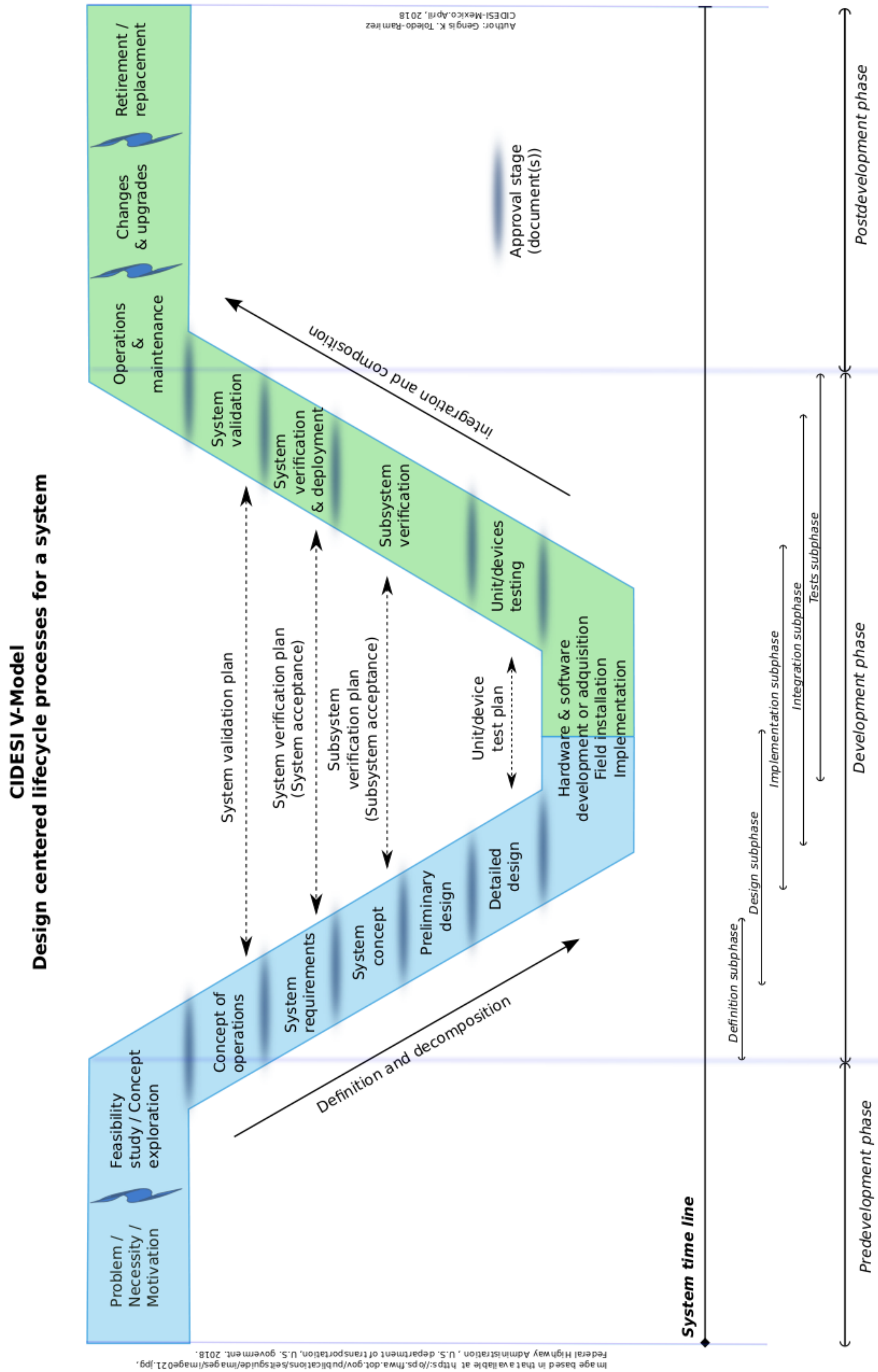

Figure 1.3: CIDESI Systems Engineering's V-Model, taken from [3]

Chapter 2

State of the art

The following chapter shows a brief description of novel Autopilots and drones that have any kind of redundant system, focusing attention on those among the civilian market. It is important to take into account that the use of redundant autopilots is not a new topic, since they have been used on military field for many years, nevertheless within the civilian market they are not widespread yet.

Military aircrafts such as the RAF's Trident fleet, used a triple redundant autoland system in the early 1960's. Ten years later, the Aérospatiale-BAC Concorde took advantage of 3X technology in its flight control system. Presently, triple redundancy is used in several manned military and commercial aircrafts [4].

2.1 Redundant Autopilots

2.1.1 MicroPilot MP21283 XM

The MP21283X autopilot (see Fig. 2.1), from MicroPilot, has a triple redundant system to increase the reliability of both fixed-wing and helicopter UAVs. If any one of the three systems fails, the remaining two take over, offering a double redundancy arrangement. If one of the other two systems should fail, the third takes over. An additional mechanism is also included to oversee these three systems. [4]

The MP21283X is comprised of three MicroPilot autopilots, mounted on a redundancy board. At the start, the autopilot in position one flies the airframe. If this autopilot should fail, the autopilot in position two takes over, and so on. The redundancy board provides several input/output (I/O) ports. The board also includes two RS232 serial

ports designed to communicate with a ground control system via radio modems. As a result of this design, users never need to work directly with bare circuit boards. Additionally, the autopilots do not have an individual casing, keeping overall weight to a minimum.

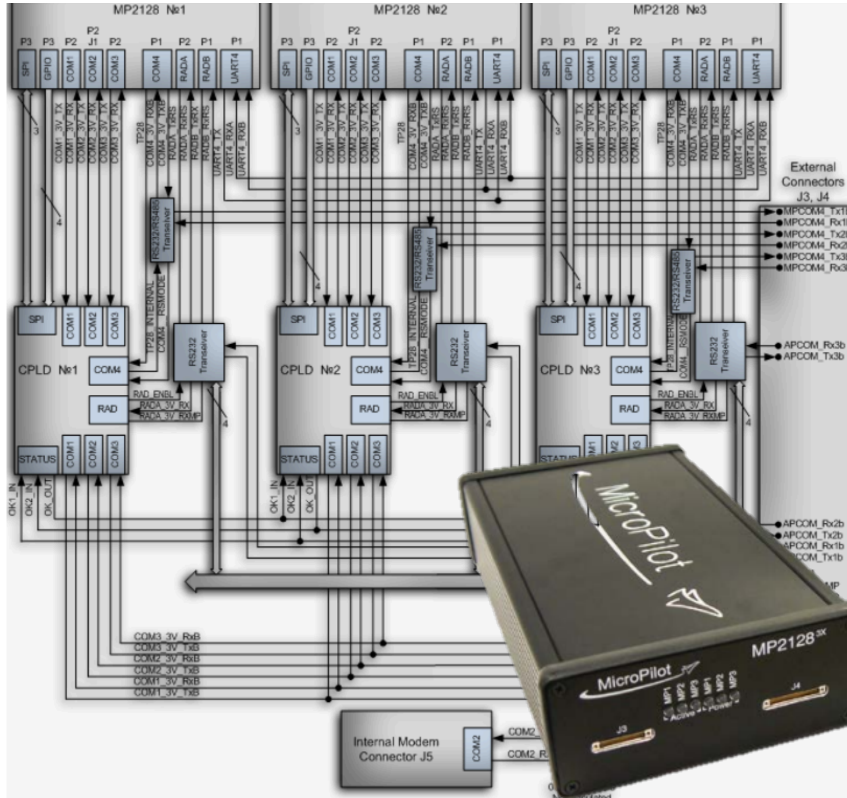

Figure 2.1: MP21283X autopilot, taken from [4]

2.1.2 Veronte Autopilot

Redundant Veronte Autopilot is a triple redundant autopilot (see Fig. 2.2). It includes three complete Veronte Autopilot modules together with an arbiter for detecting system failures and selecting the module in charge of the control. It provides a solution for controlling any unmanned vehicle such as Fixed wing, Multirotor and Helicopter.

All four modules are managed by a dissimilar microprocessor. This microprocessor compares data from all modules in the real time and processes it for discarding any autopilot module showing an undesired performance. Datalink communications are also redundant in Veronte autopilot, being possible to install 3 equal radios or to combine from compatible radio modules [5].

One spacial characteristic of this autopilot is that it has compatibility with the most demanding aeronautical regulations for onboard electronics.

Figure 2.2: Veronte triple redundant Autopilot, taken from [5]

2.1.3 Cerberus Triple Redundant Autopilot

Cerberus platform is a triple redundant autopilot from InnoFlight company (see Fig. 2.3). Cerberus enables tree Jupiter JM-1 flight autopilots, also from InnoFlight and theirs built-in IMUs, Global Position System (GPS), and compasses to work together as a true triple redundant flight control system.

The Cerberus enhances their internal sensors. As long as two Main Control Units and sensors from any module are functioning, the Cerberus would continue to provide normal flight control operations.

Figure 2.3: Cerberus triple redundant autopilot, taken from [6]

2.1.4 AP-Manager

The AP-Manager (see Fig. 2.4) is a board that allows the user to operate a DRONE safely by using two independent autopilot systems parallel. The AP-Manager is acting here as an intelligent bridge between the autopilots and switches to the alive autopilot, whenever one of the autopilots quits its operation.

The AP-Manager has implemented electronic switches, which routes the servo outputs from the autopilots to a single servo output rail depending on the desired operation mode of the AP-Manager. The AP-Manager can be operated in Manual or Automatic mode.

In both cases, the AP-Manager is monitoring the health status of the connected autopilots by checking the alive-signal of each autopilot and the servo outputs of the active This improves redundancy and therefore safety in case of malfunction in one of the autopilot systems. The AP-Manager switches to the second autopilot system and the operator is able to perform a safe landing.

An outstanding feature from the AP-Manager is that the user can mix the autopilots from different companies.

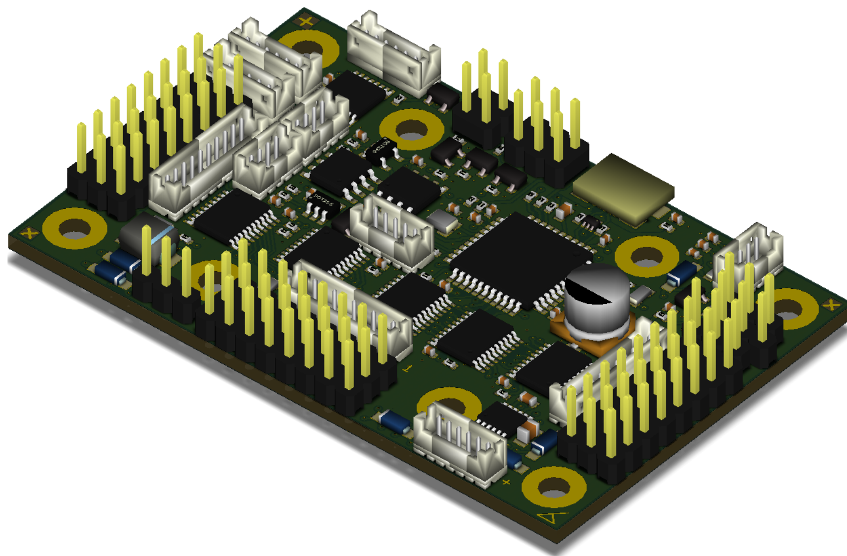

Figure 2.4: AP-Manager, redundant board for autopilots. Taken from [7]

2.1.5 Redundant Okto board

Redundant Okto (see Fig. 2.5) is a board that allows a Drone to operates with one or two autopilots in a redundancy setup. The board duplicates critical elements of the drone, so that if a problem occurs to one of them, the remaining one takes over allowing the pilot to bring the aircraft smoothly to the ground and land.

- Motor propulsion units. To ensure redundancy in propulsion, the board allows to have eight engines.
- Battery. Two batteries are required and must be connected in parallel. In this way, if one of them had to give up, the second could continue to provide enough current to allow an emergency landing
- Central unit (Autopilots). A second autopilot is installed as backup and will automatically take over, if the main autopilot should fail.

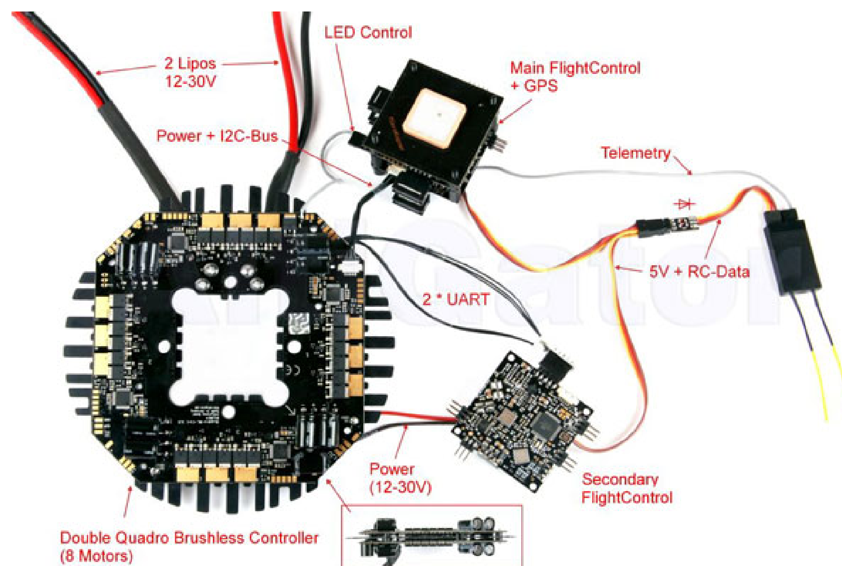

Figure 2.5: Redundant Okto board, take from [8]

2.1.6 Autopilot A3 PRO

The A3 pro an autopilot that provides reliability and precise flight control (see Fig. 2.6) , the A3 series of flight controllers focuses attention on demanding industrial and cinematographic applications where reliability and customization are essential.

The new attitude algorithms and the fusion of multiple sensors improve the control accuracy of the DJI A3. The robust control algorithm allows the A3 and A3 PRO to adapt to a wide range of drones without the need for manual adjustment [9]. It has a fault-tolerant control system, a Hexacóptero or Octocóptero can land safely even in case of failure of the propulsion system. For example, a hexacopter can even fly with 3 motors.

Figure 2.6: A3 pro Autopilot, taken from [9]

2.2 Redundant Drones

2.2.1 Euroavionics multicopter

Euroavionics multicopter (see Fig. 2.7) is a Drone with completely autonomous flight control and are capable of meeting the needs of various operators including police, fire, and infrastructure inspection. This drone has redundancy on its flight control, motors and engine control.

The multicopter drone is a professional tool which addresses the specific requirements of the police and fire brigades. Engineered to highest quality standards with built in redundancy, the system allows the police to do crowd-control, acquire post-accident data as well as perform search missions. Operators such as fire brigades will receive a comprehensive overview of the situation within minutes from arriving on site [10].

Figure 2.7: Euroavionics multicopter, taken from [10]

2.2.2 AscTec Trinity Drone

The AscTec Trinity is a triple redundant flight control (see Fig. 2.8), which is standard in commercial airliners to ensure maximum safety. Errors in flight critical attitude sensors can be identified by automatic data comparison with two redundant units [11]. The same applies to all communication systems relevant for control. Due to the adaptive flight control the Drone will stay up in the air if there is a failure in one of the autopilots. The UAVs automatically compensates for disturbances and will actively support the pilot to control the system.

Furthermore all electronic hardware devices are at least two times redundant. The drone AscTec Trinity would compensate troubles immediately. It also has a isolated operating system for more safety and functional operation of the flight system.

Figure 2.8: AscTec Trinity Drone, take from [11]

2.2.3 Matrice 600 Pro

Matrice is a professional Drone provided with redundant system. The airframe is equipped with the latest DJI technologies, including a A3 Pro flight controller, Intelligent Batteries and Battery Management system.

The autopilot A3 Pro Flight has triple modular redundancy and diagnostic algorithms that compare sensor data from three sets of Global Navigation Satellite System (GNSS). A new system for the A3's modules enable precise control of multi-rotor aircraft, providing accurate data for stable flight performance. Self-adaptive systems will automatically adjust flight parameters based on different payloads [12]. Furthermore the A3 Pro can withstand magnetic interference, providing centimeter level accuracy, suitable for various industrial applications.

Figure 2.9: Matrice 600 Pro Drone, taken from [12]

Chapter 3

Project development

The next chapter describes the project development, its detailed operation and organization. Firstly, it is presented a system definition based on the steps established in the V model from Chapter 1, followed by the main components selected to constitute the redundant system, and finally it is presented a detailed design along with a description of the system general operation where each and every one of the elements that constitute the drone are described, focusing special attention on those that are associated to the redundant system.

It is important to highlight that it is expected from the reader to have little knowledge about ROS at the moment of reading this chapter, since the vast majority of this project is developed using ROS framework.

3.1 System definition

Next is shown a brief explanation of activities, steps and procedures that form the system definition, following the V model already shown in Chapter 1.

- Problem necessity.
  Development of a redundant autopilot that will increase safety and reliability of the system, using open source software and hardware to enable the project with high scalability.

- System requirements.

  The redundant system is based on terms and set of ideas to meet the requirements and vision established in the next Table 3.1.

Table 3.1: Redundant Autopilot characteristics

Main characteristics
Redundancy in one autopilot.
Use os ROS as supervisor to monitor the correct operation of both autopilots.
Use of QGroundStation to monitor incoming data from both autopilots.
Use of a open source flight stack for Autopilots.
Mixable hardware possible (different autopilots).
Future firmware updates possible.
Use of MAVROS to establishes communication between ROS and MAVLink protocol.
Aimed to enable a scalable project.

- Preliminary design.

  The Preliminary conceptual design shows the following components (see Fig. 3.1).

  - Companion computer. It is in charge of controlling and supervising the correct operation of both autopilots. This companion computer runs ROS over ubuntu 16.4, by using ROS the drone is able to receive commands from the Ground Control Station (GCS).
  - Autopilot 1 y 2 (Pixhawk). Pixhawk autopilot is a popular general purpose flight controller based on the Pixhawk-project FMUv2 open hardware design.
  - Wi-fi module. Allows connections between GCS and the companion computer.
  - Electric Speed Controller (ESC). Controls and regulates the speed of four rotors that lift the Drone.
  - GPS. Provides drones with Global localization and has installed a compass used for navigation and orientation that shows direction relative to the geographic cardinal directions.

- Software development.
  Within this step ROS, GAZEBO and MAVROS API are used to run, test and write the CODE that will oversees, and administrate the correct operation of each Autopilots, if one of the autopilots should fail the next one will take over the control of the DRONE.

- Unit testing.
  Unit Test Plans are developed during this design phase. These unit test are executed to eliminate bugs at code level.

- System verification.
  Integration testing is associated with this phase. Integration tests are performed to test the coexistence and communication of the internal modules within the system, test are executed to check the correct communication between ROS and the autopilots.

- System deployment.
  Test the complete application with their functionality, non-functional requirements, and communication of developed application. Here it is applied a Hardware in the loop (HIL) phase by connecting the two autopilots to a computer throughout USB port.

- System validation and operation.
  This two phases will not be executed since the project is just the first approach in this line of investigation.

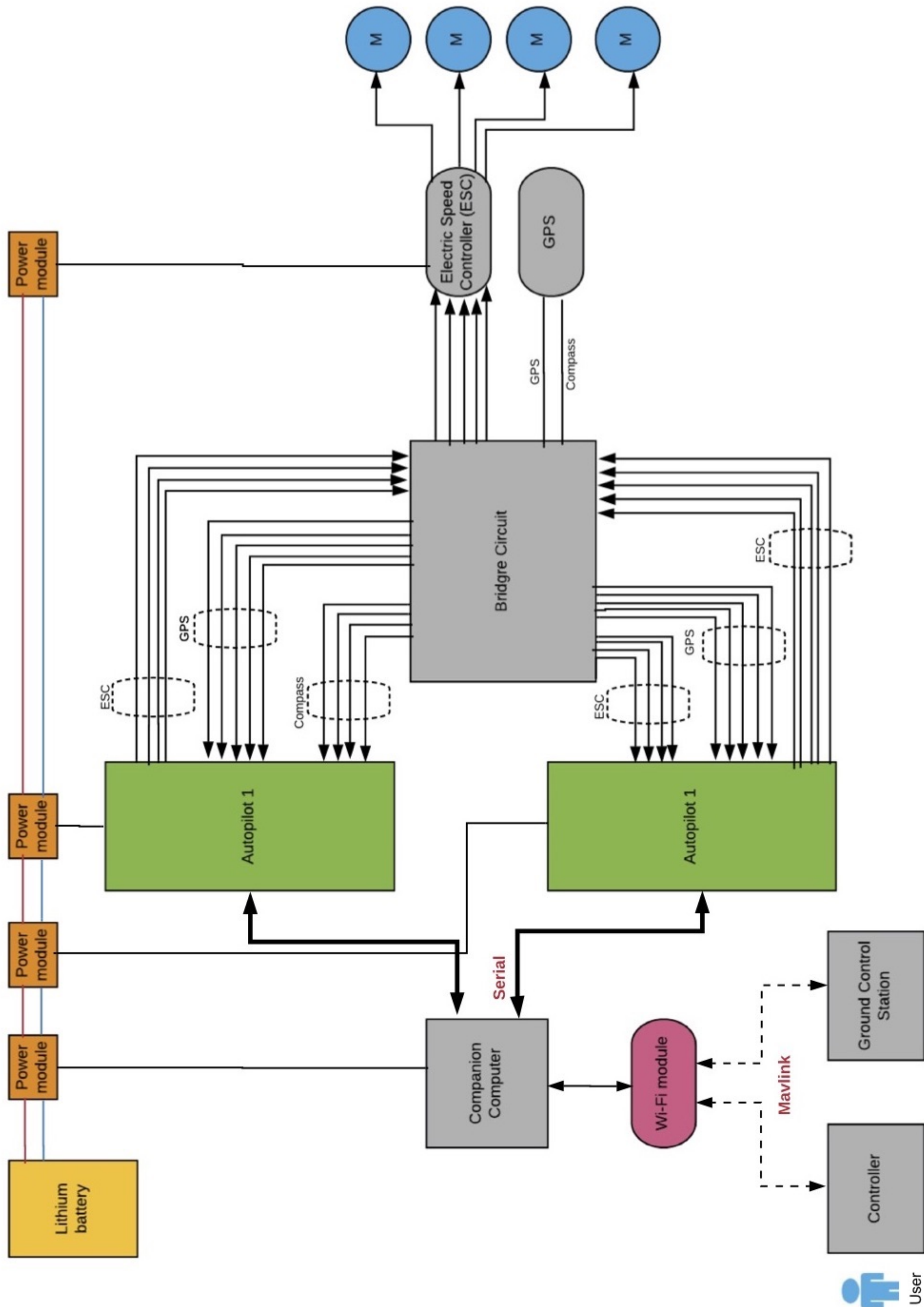

Figure 3.1: First system concept

3.2 Selected elements

All the hardware and software were selected to fit a tight budget, taken from open source projects and most of all aim to enable scalability.

3.2.1 Flight Controller

The Pixhawk® 1 autopilot (see Fig. 3.2) is a popular general purpose flight controller based on a open hardware design. Among the softwares that it runs are PX4 and Ardupilot, both also open source projects.

Figure 3.2: Pixhawk Autopilot

Table 3.2: Pixhawk Autopilot Features

	Key Feature
CPU	180 MHz ARM® Cortex® M4 with single-precision FPU
RAM	RAM: 256 KB SRAM
CONNECTIVITY	1x I2C 1x CAN (2x optional) 1x ADC 4x UART (2x with flow control) 1x Console 8x PWM with manual override 6x PWM / GPIO / PWM input S.BUS / PPM / Spektrum input S.BUS output
SENSORS	ST Micro L3GD20H 16 bit gyroscope ST Micro LSM303D 14 bit accelerometer / magnetometer Invensense MPU 6000 3-axis accelerometer/gyroscope MEAS MS5611 barometer

3.2.2 PX4 autopilot

PX4 autopilot software runs the whole time in the Pixhawk 1. PX4 is an open source flight control software that powers any vehicle from flying drones to ground vehicles. It enables developers create custom drone operating systems with a flexible core. Whether running a racing drone or a drone fleet, PX4 can be customized and tailored to fit any need.

3.2.3 Companion Computer

Raspberry pi model b+ acts as a companion computer (see Fig. 3.3). The idea behind a companion computer is to be able to control the PX4 flight stack using software running outside of the autopilot. This is done through the Mavlink protocol.

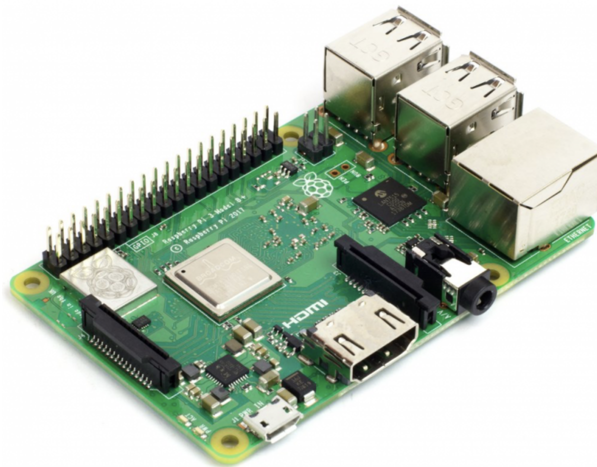

Figure 3.3: Rasberry Pi as a Companion computer

3.2.4 Robot Operating System (ROS)

Robot Operating System (ROS) is a general purpose robotics library that can be used with PX4 for offboard control. It uses the MAVROS node to communicate with PX4 running on hardware or using the Gazebo Simulator. The ROS distribution used in this project is ROS Kinetic since it is the long term support version for Ubuntu 16.04.

3.2.5 QGroundControl

QGroundControl provides full flight control and vehicle setup for PX4 or ArduPilot powered vehicles (see Fig. 3.4). It provides an easy and straightforward way to setup autonomous missions on the Autopilot (see Table 3.3).

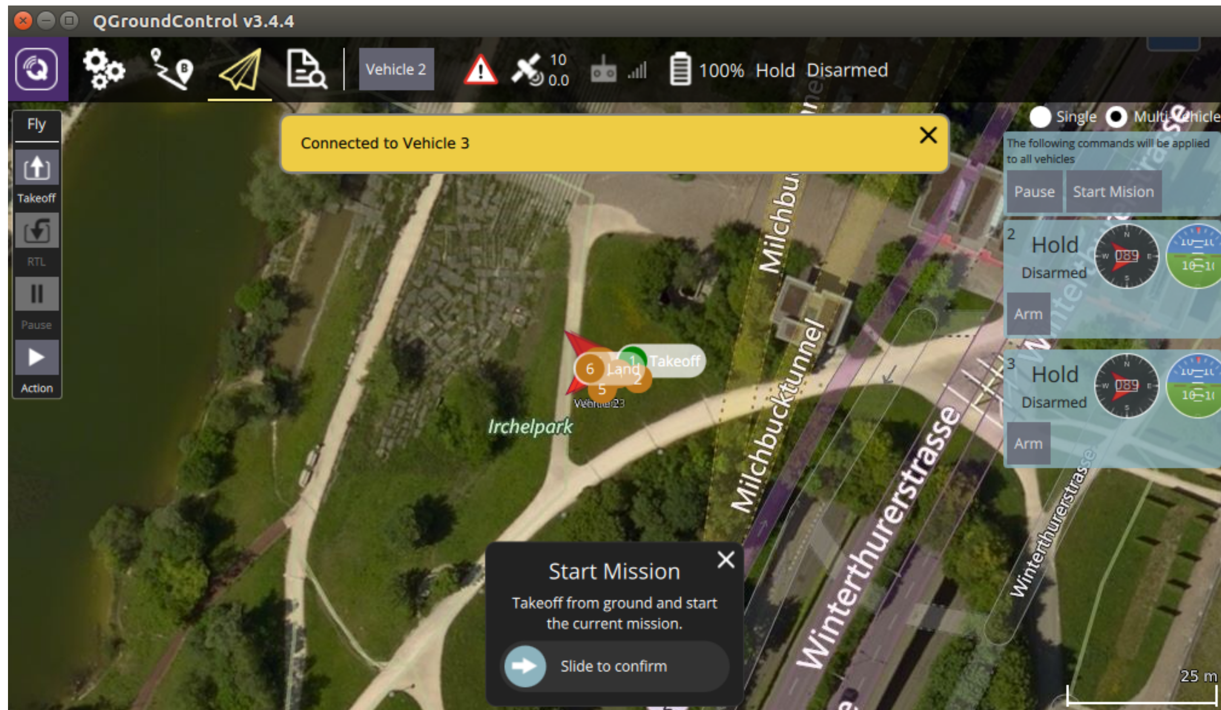

Figure 3.4: QGroundControl for planning autonomous flights

Table 3.3: QGroundControl Features

Main characteristics
Full setup/configuration of ArduPilot and PX4 Pro powered vehicles.
Flight support for vehicles running PX4 and ArduPilot (or any other autopilot that communicates using the MAVLink protocol).
Mission planning for autonomous flight.
Flight map display showing vehicle position, flight track, waypoints and vehicle instruments.
Video streaming with instrument display overlays.
Support for managing multiple vehicles.
QGC runs on Windows, OS X, Linux platforms, iOS and Android devices.
Full setup/configuration of ArduPilot and PX4 Pro powered vehicles.

3.3 Detailed design

The drone operates with two autopilots which are connected to a companion computer and communicate to it through the so called MAVLink protocol. The companion computer's function is to keep track of the operation of both autopilots and to take over the control of the so called "bridge circuit". The bridge circuit connects at all times one and just one autopilot to the ESC, by doing so there is no interference between the signals that both autopilots sends to it since both of them try to send signals to the ESC at the same time (see section 3.4.6). Recall that the ESC controls and regulates the speed the motors based on the signal that receives from an Autopilot. The component that commands the bridge circuit to connect an specific Autopilot to the ESC is as it was already mentioned the companion computer. If the autopilot currently in use should failed, the companion computer sends a signal to the bridge circuit to switch to the redundant autopilot, as a result the signals sent to the motors are never interrupt and the drone keeps in the air without any problem.

Here it is important to point out that ROS (running on the companion computer) executes the algorithm used to determinate when an autopilot has failed. ROS gets information from the autopilots through MAVROS node which in turn gets incoming data from the USB ports. The Autopilots sends its information through the serial port TELEM 2 using MAVlink protocol. The Fig. 3.5 depicts how all components interconnect with each other and next list is a summary of the main elements indispensable for the redundant system.

- **Companion computer**. Gets autopilots information.
- **Pixhawk Autopilots**. Controls the trajectory of an aircraft without constant 'hands-on' control by a human operator being required.
- **ROS**. It is a flexible framework for writing robot software. It executes the algorithm used to detect a failure in any of the autopilots.
- **MAVROS**. Enables MAVLink extendable communication between computers running ROS and MAVLink enabled autopilots. It is the Official bridge between ROS and enabled MAVlink autopilots.
- **MAVlink protocol**. MAVLink is a very lightweight messaging protocol for communicating with drones.
- **Bridge circuit**. Connects one and just one autopilot to the ESC.

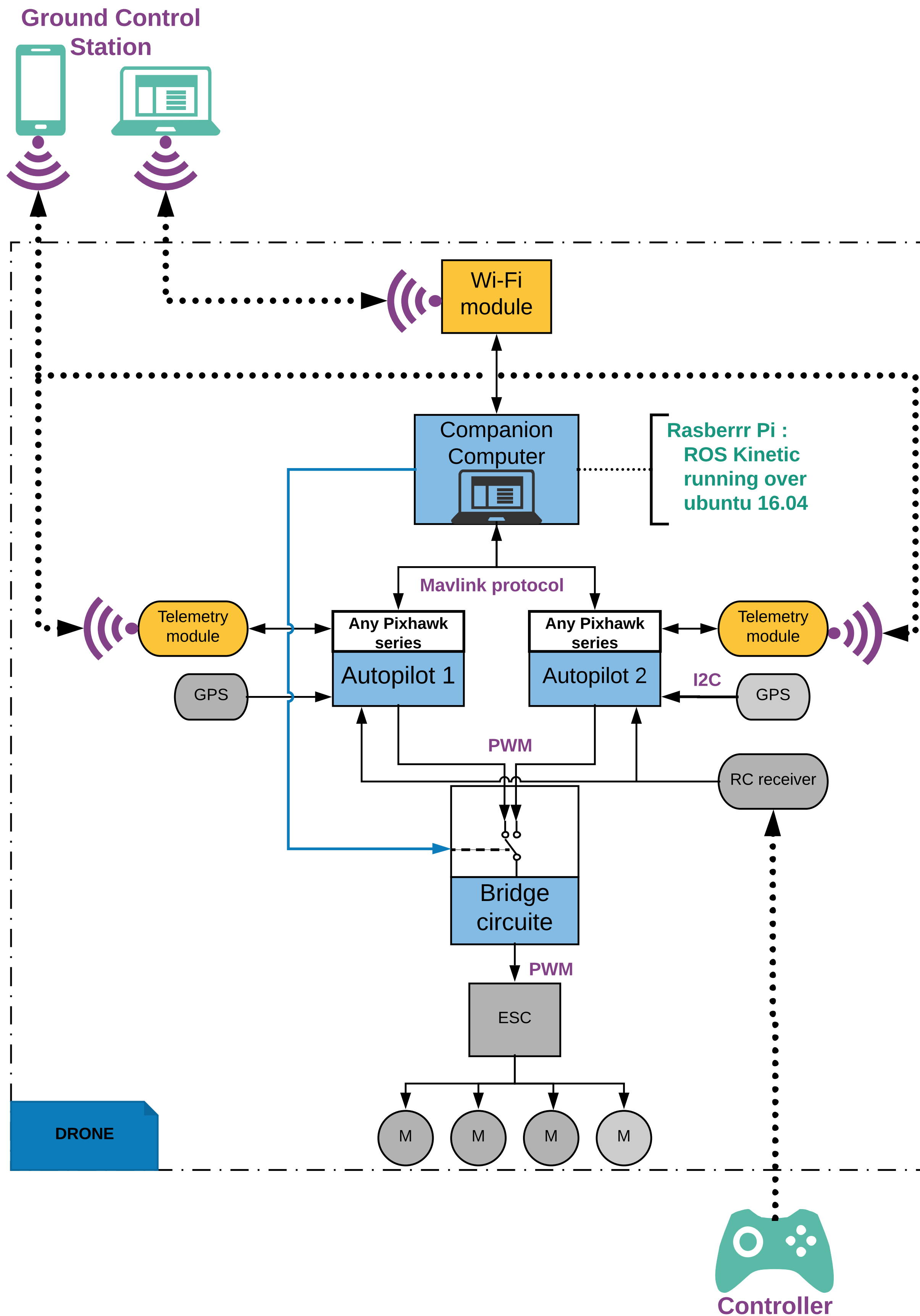

Figure 3.5: Drone Architectural Overview

3.4 The redundant system

3.4.1 Data streamed flow between the companion computer and the Autopilots

The idea behind the companion computer is to be able to get data from the PX4 stacks, coming through the MAVLink protocol, namely those related with the health of the autopilots. Here the companion computer mounted onto the drone, communicates with both autopilots using MAVROS package which in turn communicates with ROS to supervise the operation of both autopilots. Data stream flow can be seen in next Fig. 3.6.

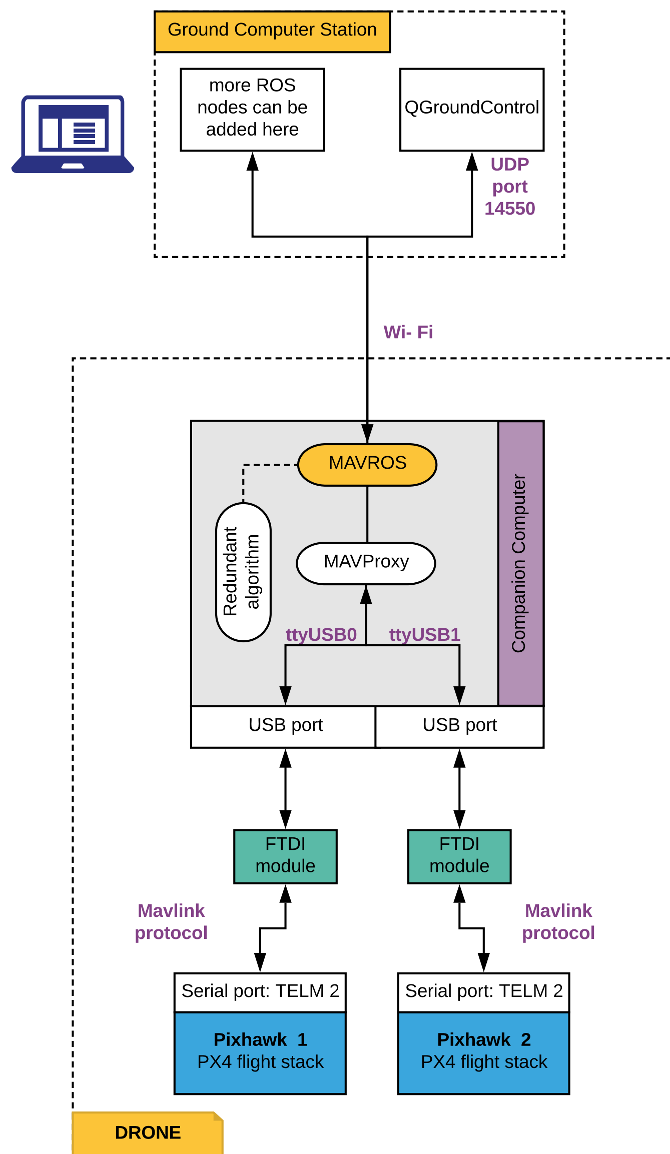

Figure 3.6: System data flow

It is necessary to use and install MAVROS since ROS does not understand MAVLink protocol coming from the PX4 stack. MAVROS is the“official” supported bridge between ROS and the MAVLink protocol. In addition MAVProxy it is also required to route the MAVLink data coming from serial ports to MAVROS node.

3.4.2 Data coming from Autopilots

MAVROS automatically publishes data coming from the autopilots into `/Diagnostics` topic. These `/Diagnostics` topic is designed to collect information from hardware drivers and robot hardware to users and operators for analysis, troubleshooting, and logging. The data (see Table 3.4) related with autopilots' health is subtracted from here.

Table 3.4: Flags used to check Autopilots status.

Flag	Description	Possible Values
FCU_status	Shows status of connection between MAVROS and the px4 stack	OK=0 WARN=1 ERROR=2 STALE=3
Heartbeat_Status	The heartbeat is used to determine whether a system is connected, and to detect when it has disconnected.	
Gyroscope	Status of gyroscope embedded sensor	
Accelerometer	Status of accelerometer embedded sensor	
Magnetometer	Status of magnetometer embedded sensor	

`/Diagnostics` publishes data using `diagnostic_msgs/DiagnosticArray` messages with the device names, status and specific data points, see Fig. 3.7.

File: diagnostic_msgs/DiagnosticStatus.msg

Raw Message Definition

```
# This message holds the status of an individual component of the robot.
#

# Possible levels of operations
byte OK=0
byte WARN=1
byte ERROR=2
byte STALE=3

byte level # level of operation enumerated above
string name # a description of the test/component reporting
string message # a description of the status
string hardware_id # a hardware unique string
KeyValue[] values # an array of values associated with the status
```

Figure 3.7: Diagnostics message

3.4.3 Communication setup

Enabling communication between a companion computer and a Pixhawk boards requires to setup the firmware on both sides and they must are interfaced using the serial port `TELEM 2`, this port is the default port used for communicating with companion computers. The following points shows a brief explanation of the main parameter to configure, it is required to go and check PX4 Development Guide if the reader wants to get a deeper understating about this matter.

3.4.3.1 Companion computer

In order to receive MAVLink data, the companion computer needs to run some software talking to the serial port, the reader can check appendix B to learn how to install this packages. They are:

- MAVROS to communicate to ROS nodes.
- C/C++/Python code to connect custom code
- MAVProxy to route MAVLink between serial and UDP.

3.4.3.2 Pixhawk Setup

It is required to enable MAVLink on the serial port that will be used to connect to a companion computer (`TELEM 2` is the default port used for companion computers). Depending on the firmware version and the hardware model used different parameter must be set before enabling MAVLink on an specific autopilot's serial port. Next parameters are the commonly used for all Pixhawk series.

- `MAV_1_CONFIG` = TELEM 2 (`MAV_1_CONFIG` is often used to map the TELEM 2 port)
- `MAV_1_MODE` = Onboard
- `SER_TEL2_BAUD` = 921600 (921600 or higher recommended for applications like log streaming or FastRTPS)

When working with the first Pixhawk model it is only necessaty to set parameter `SYS_COMPANION` to 921600. QGroundControl is used in order to set and change parameters in Pixhawk autopilots.

3.4.3.3 Hardware Setup

The serial port is wired according to the next Table 3.5. All Pixhawk serial ports operate at 3.3V and are 5V level compatible. To connect the serial port from the autopilots to the companion computer's USB ports two Future Technology Devices International (FTDI) modules are used.

Table 3.5: Wiring to Pixhawk (FTDI Chip USB-to-serial adapter board)

TELEM2		FTDI	
1	+ 5V (red)		NOT CONECTED!
2	Tx (out)	5	FTDI RX (yellow) (in)
3	Rx (in)	4	FTDI TX (orange) (out)
4	CTS (in)	3	FTDI RTS (green) (out)
5	RTS (out)	2	FTDI CTS (brown) (in)
6	GND	1	FTDI GND (black)

Section 4.5 shows how to establish connection between the autopilots and the companion computer once the previous requirements are set.

3.4.4 Behind the Logic

The programs executed by ROS can be divided into three groups for explanatory purpose, the first two run in the drone at all time (allocated withing the package `redundant_auto`) and the third one in a Ground station computer (inside the package `ground_station`). The first group of programs oversees both autopilots and switches to the redundant one in case of any problem, the second group reports data to a Ground enabled ROS computer to allow remote supervision, and the third group controls the drone by sending commands through ROS. Next sections explain the logic followed by the code. The readers should check the code line by line if they want to have a deep understanding of the logic and to know how exactly all works.

Programs inside redundant_auto package.

Next Fig. 3.8 shows the logic flow and interactions between the nodes created by the package `redundant_auto` , they are shown in the sequential order they occur, namely

`scanAutopilots`, `fromDiagnostics` and `executes_SafetyRoutine`. Each one of these nodes perform and specific task. It is important to take into account that there are a lot more nodes running along with the ones already mentioned, but for explanatory purposes and an easy understanding there is only explained the three nodes already mentioned.

- **scanAutopilots**.
  This node executes a continues loop in which requests information regarding both autopilots health (through the `getData_fromAutopiltos()` method). Here the values that can be returned are either noError, FailureOnAuto1 or FailureOnAuto2. The while loop will run indefinitely as long as the returned value is noError.

  In case of a failure the program continues by calling `switchAuto()` method. This function when executed sends a signal to the bridge circuit to immediately switch to the redundant autopilot which takes over the control of the aircraft.Afterwards a safety routing is executed. At this point the returned value can be either FailureOnAuto1 or FailureOnAuto2, in any case `call_safetyRouting()` method will be called. This last function executes a safety routing that the healthy autopilot should execute.

- **fromDiagnostics**.
  This node filters data coming from `/diagnostics` topic, and returns a message with the current state of both autopilots (`noError, FailureOnAuto1 or FailureOnAuto2`). To do this uses the data shown previously in Table 3.4

- **inCaseEmergency**.
  This node have defined a safety routing that will land the drone in case of a failure in any of the autopilots. When called it is required to pass the name of the health autopilot as an argument in order to command the correct autopilot. When executed it will wait 3 secs in the air and afterwards it starts a safety landing.

  It is important to point out that this node holds a service thus when called all process within ROS will stop and continue after the safety land has already been completed. In case the user wants to continue flying the drone, it is only required to switch to "manual mode" from the controller and the safety routing will be ignored.

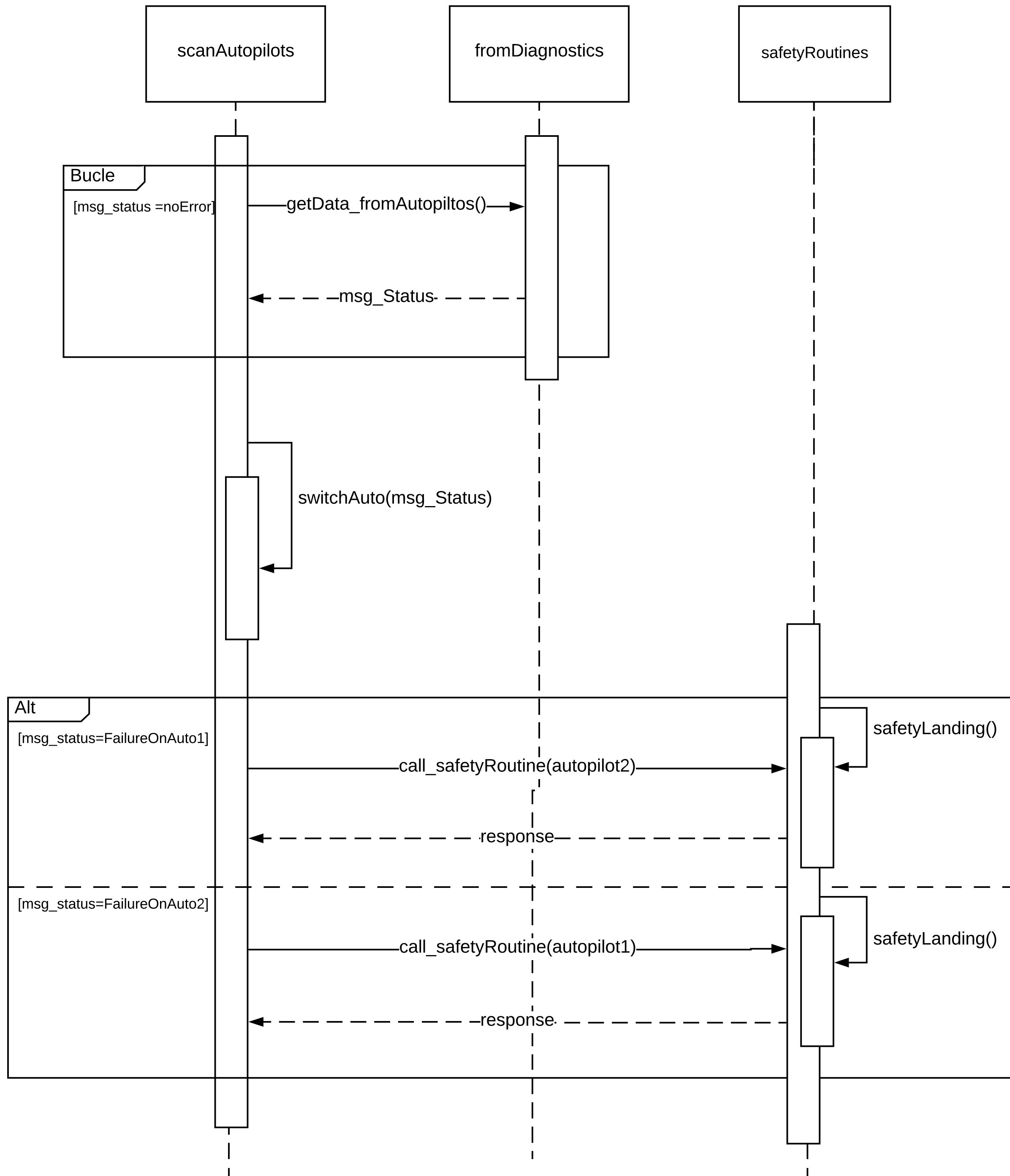

Figure 3.8: Redundant Autopilot sequence diagram.

Programs inside ground_station package.

This package works along with package redundant_auto. When the main node is started it calls the action server defined in package redundant_auto and then it displays a menu that allows the user to choose an autopilot and sends commands to the drone .

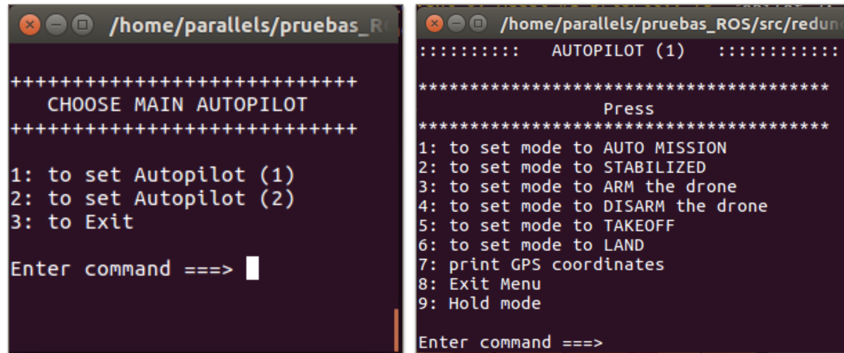

Figure 3.9: Ground computer Menu

The node“Mein_menu_node” requests through `checkAutopilots()` method the status of both autopilots, afterwards `Datato_GComputer` return a message with the current state of both autopilots (returned values are `data_fromDrone.auto1`, `data_fromDrone.auto2` or `data_fromDrone.Gstatus`), see Fig. 3.10 .

If `data_fromDrone.Gstatus`=OK is returned the program request the user to select a autopilot to control de Drone, afterwards a loop starts in which the user will sends commands as long as an exit is not requested from the user side.

If `data_fromDrone.auto1=FAIL or data_fromDrone.auto1=FAIL` are returned the program allows to sends only commands to the current autopilot being healthy through `startControlling_OneAuto()` method.

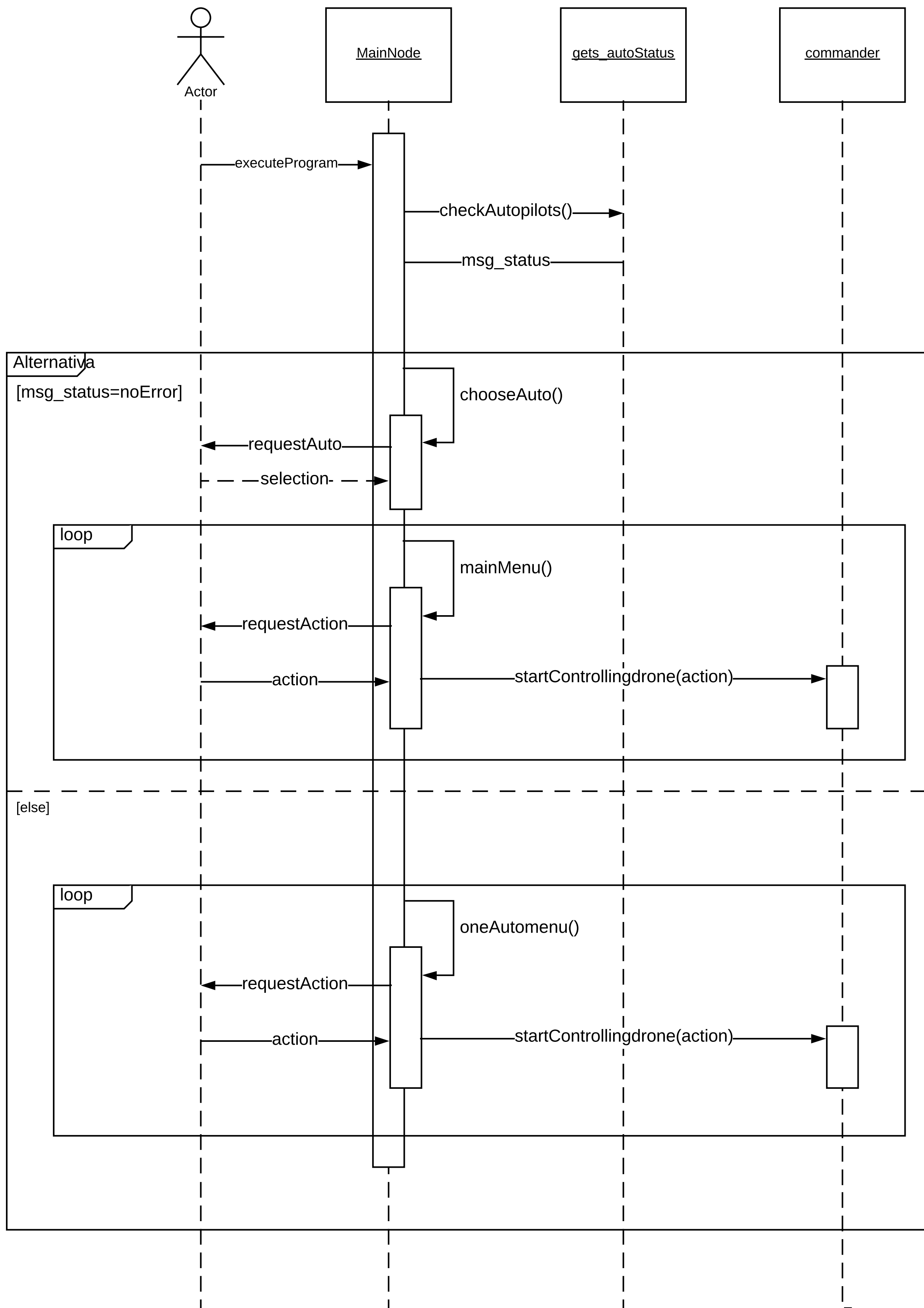

Figure 3.10: Control control sequence diagram

3.4.5 ROS packages

ROS gets information coming from the autopilots through `/diagnostics` topic, from this topic are obtained the flags that were already shown in Table 3.4. The programs follow the logic already explained in previous section 3.4.4, and are organized in two packages, one called `redundant_auto` and a second one called `ground_station`.

The package `redundant_auto` runs in the drone and follows the sequence diagram already shown in Fig. 3.8. It contains the next files.

- **read_Diagnostics.py**
  Gets information coming from the autopilots and publishes the filtered data.
  defined Node: 'fromDiagnosticsNode'
  subscribers: /diagnostics
  publishers: /autopilots_status
  –message: status_msg.msg

- **scanAutopilots_serClient.py**
  Checks continuously the the autopilots' state, in case of failure calls the service /call_safetyRouting_serServer.
  defined Node: 'scan_Autopilots'
  subscribers: /autopilots_status
  service called: /call_safetyRouting_serServer

- **execute_safetyRoutine_serServer.py**
  Holds a service which when called executes a safety routine defined in safety_routines.py.
  All process within ROS are stopped once this service is called.
  defined Node: 'executes_SafetyRoutine'
  defined service: /call_safetyRouting_serServer
  –message: emergency_msg.srv

- **safety_routins.py**
  Only safety routines are defined in this file.

- **Send_datatoGroundStation_actServer.py**
  Holds an action which when called from the ground computer provides the current status of both autopilots.
  defined Node: 'Connect_GComputer'
  subscribers: /autopilots_status

defined action: /Datato_GComputer
–message: messagetoPCAction.action

The package `ground_station` runs in a ground enabled ROS computer and executes the next programs, following the sequence diagram already shown in Fig. 3.10.

- **main_menu.py**
  When started calls call_action_getsAutopilotStatus_client.py. It shows a menu to send commands to the drone.
  defined Node: Mein_menu_node'
- **getData_fromDrone_actClient.py** Checks the current status of the autopilots by making a call to the action /Datato_GComputer.
  action called: /Datato_GComputer
- **droneController.py**
  Shows a menu with the commands available to sent to the drone.
- **commander.py**
  Holds the commands available to sent.

Next Fig. 3.11 depicts the connection between all the nodes. When the drone starts operation the node `/fromDiagnosticsNode` checks the status of the autopilots and sends this information through the topic `/autopilots_status`.

The node `/scan_Autopilots` uses data coming from `/autopilots_status` to detect any failure, in case one is detected the service `/call_safetyRouting_serServer` is called. This service sends a signal to the bridge circuit and executes a safety routine.

The last node `'Connect_GComputer'` only provides current information about the status both autopilots.

In case a ground computer is being used an additional node is added to the ROS system, this node executes a menu in which any user can commands to the drone.

It is necessary to check the code within both packages `redundant_auto` and `ground_station` if the reader wants to get a better understating of this matters.

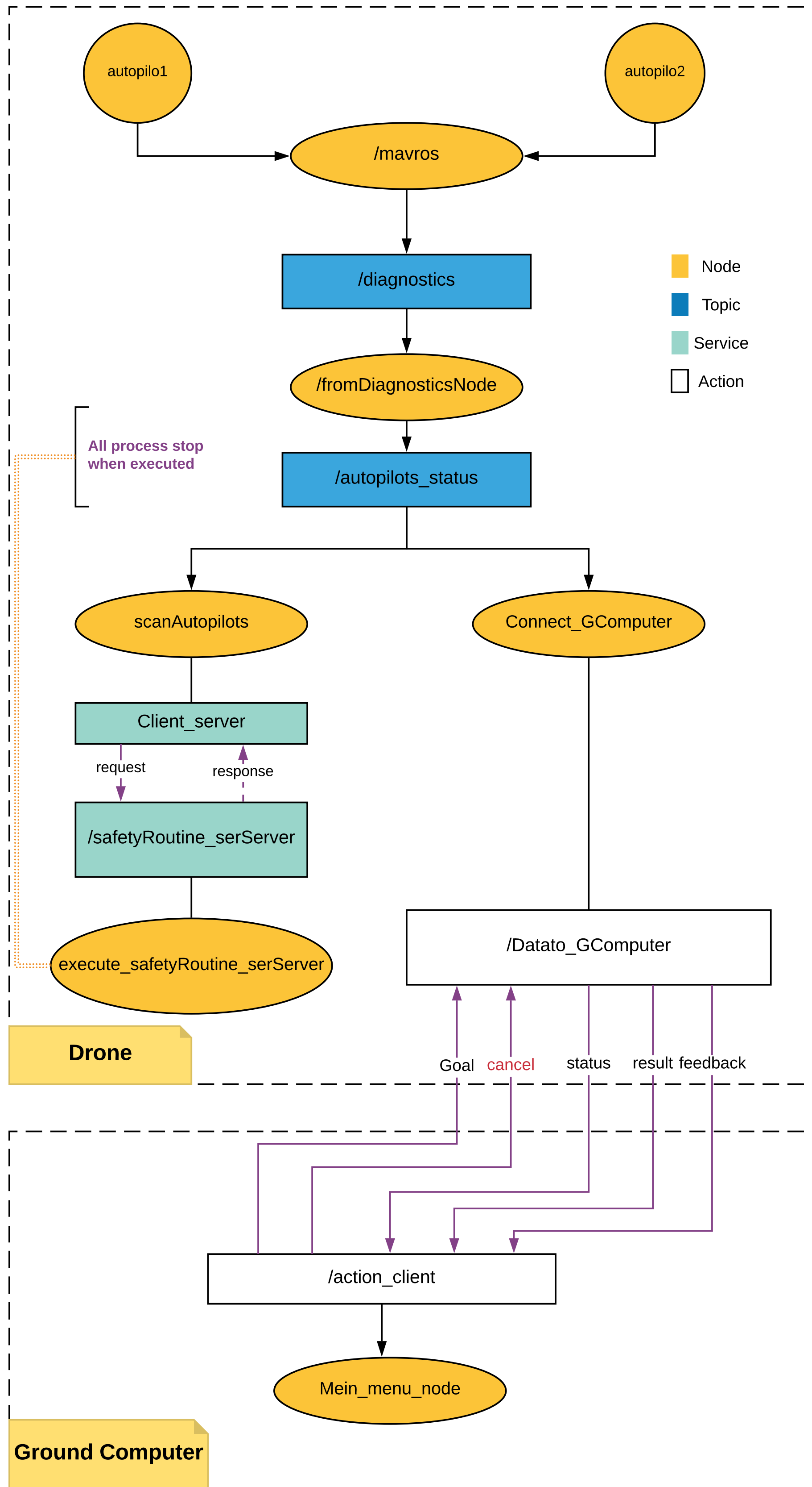

Figure 3.11: ROS computation graph

3.4.6 Bridge Circuit

The bridge circuit (see Fig. 3.12) switches connection between the autopilots and the ESC depending on the signal that it receives from the companion computer. The circuit only consist on an arrangement of 4 high speed solid stated relay with single pole and two throws.

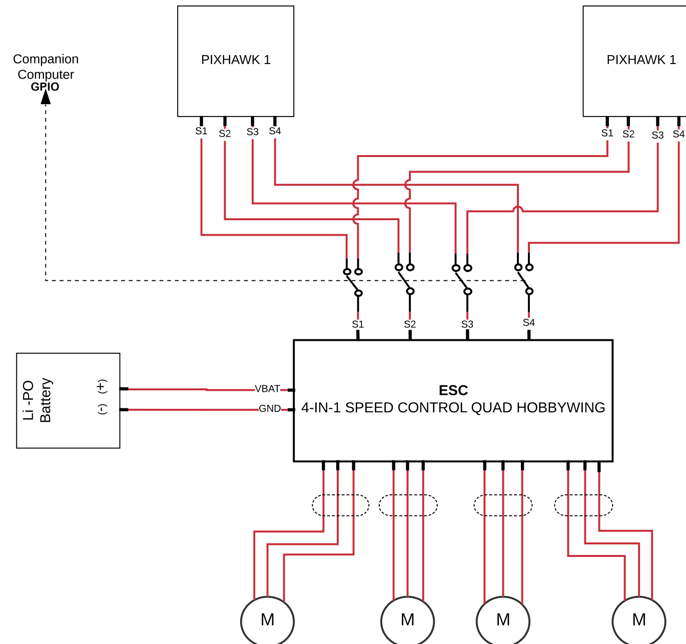

Figure 3.12: Bridge circuit connection

ESC Connection Overview.
Each PWM Electronic Speed Controller (ESC) minimally has the following wires.

- Power VBAT (usually thick and red).
- Power GND (usually thick and black).
- PWM signal for each motor (usually white or yellow).
- GND (usually black or brown)

The servo plug may also have a +5V wire (usually red or orange). The purpose of this wire and how it is connected depends on particular ESC and vehicle type.

Many PX4 drones use brushless motors that are driven by the flight controller via an Electronic Speed Controller (ESC). The ESC converts a signal from the flight controller

to an appropriate level of power delivered to the motor. PX4 supports ESCs that take a PWM input, ESCs that use the ESC OneShot standard, UAVCAN ESCs, PCA9685 ESC (via I2C), and some UART ESCs (from Yuneec).

Proposed ESC.

- ESC 20A Favourite SKY III Quattro (4 in 1). 159/5000 The electronic speed controller (ESC; Electronic Speed Controller) 4 in 1 with BEC is the perfect solution for medium sized quadricopters.
- The MAX4541–MAX4544 are precision, dual analog switches designed to operate from a single +2.7V to +12V supply. Low power consumption (5µW) makes these parts ideal for battery-powered equipment

Proposed SSR.

- The NLAS44599 is an advanced dual independent CMOS double pole double throw (DPDT) analog switch fabricated with silicon gate CMOS technology.
- The MAX4541–MAX4544 are precision, dual analog switches designed to operate from a single +2.7V to +12V supply. Low power consumption (5µW) makes these parts ideal for battery-powered equipment

Chapter 4

Simulation and test process

Next chapter provides information about the SIL simulation phase, followed by a explanation of the Hardware in the Loop process. The HIL is established to validate the correct operation of the redundant system.

PX4 supports both SIL simulation, where the flight stack runs on computer (either the same computer or another computer on the same network) and HIL simulation using a simulation firmware on a real flight controller board.

The reader can check out appendix B to learn how to install PX4 firmware, gazebo, mavros and ROS, in addition sections sections 4.4 and 4.5 show how to set up a SIL and HIL phase respectively.

4.1 HIL vs SIL

SIL runs on a development computer in a simulated environment, and uses firmware specifically generated for that environment. Other than simulation drivers to provide fake environmental data from the simulator the system behaves normally.

By contrast, HIL runs normal PX4 firmware, on normal hardware. The simulation data enters the system at a different point than for SIL. Core modules like commander and sensors have HIL modes at startup that bypass some of the normal functionality. In summary, HIL runs PX4 on the actual hardware using standard firmware, but SIL actually executes more of the standard system code.

Gazebo

Gazebo is the chosen software to perform the SIL. Gazebo is a powerful 3D simulation environment for autonomous robots that is particularly suitable for testing object-avoidance and computer vision, see Table 5.1.

Gazebo is often used with ROS, a toolkit/offboard API for automating vehicle control.

Table 4.1: Gazebo characteristics

Simulator	Description	Supported UAVs
Gazebo	A powerful 3D simulation environment that is particularly suitable for testing object-avoidance and computer vision. It can also be used for multi-vehicle simulation and is commonly used with ROS, a collection of tools for automating vehicle control. **This simulator is highly recommended.**	• Quad (Iris and Solo) • Hex (Typhoon H480) • Generic quad delta VTOL • Tailsitter • Plane • Rover • Submarine (coming soon!)

4.2 SIL phase.

4.2.1 How the simulation works.

Gazebo communicates with PX4 using the Simulator MAVLink API. This API defines a set of MAVLink messages that supply sensor data from the simulated world to PX4 and return motor and actuator values from the flight code that will be applied to the simulated vehicle. The Fig. 4.1 below shows the message flow.

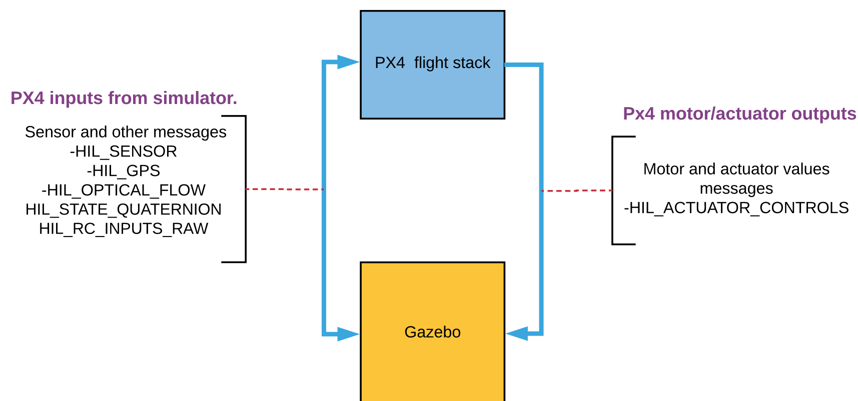

Figure 4.1: MAVlink API

4.2.2 Communication between PX4 and Gazebo.

By default, PX4 uses commonly established UDP ports for MAVLink communication with ground control stations (e.g. QGroundControl), MAVROS and simulator APIs (e.g. Gazebo). These ports are:

- Port 14540 is used for communication with offboard APIs. Offboard APIs are expected to listen for connections on this port, ROS is a perfect example for this.

- Port 14550 is used for communication with ground control stations. GCS are expected to listen for connections on this port. QGroundControl (control stations used in this project) listens to this port by default.

- Port 14560 is used for communication with simulators. PX4 listens to this port, and simulators are expected to initiate the communication by broadcasting data to this port.

4.2.3 SIL Simulation Environment

.

The Fig.4.2 shows the established SIL simulation environment for this project. The different parts of the system connect via UDP, and can be run on either the same computer or another computer on the same network.

- Two PX4 stacks use a simulation-specific module to listen on UDP ports 14561 and 14562. Gazebo connect to this ports, then exchange information using the Simulator MAVLink API described in subsection 4.2.2. Both PX4 stacks can run on either the same computer or different computers on the same network.

- A serial connection can be used to connect Joystick/Gamepad hardware via QGroundControl.

- ROS runs at all time the algorithm explained in section 3.4.4.

- Both PX4 stacks uses the normal MAVLink module to connect to QGroundControl (which listen on port 14550) and ROS (which listen on ports 14541 and 1452).

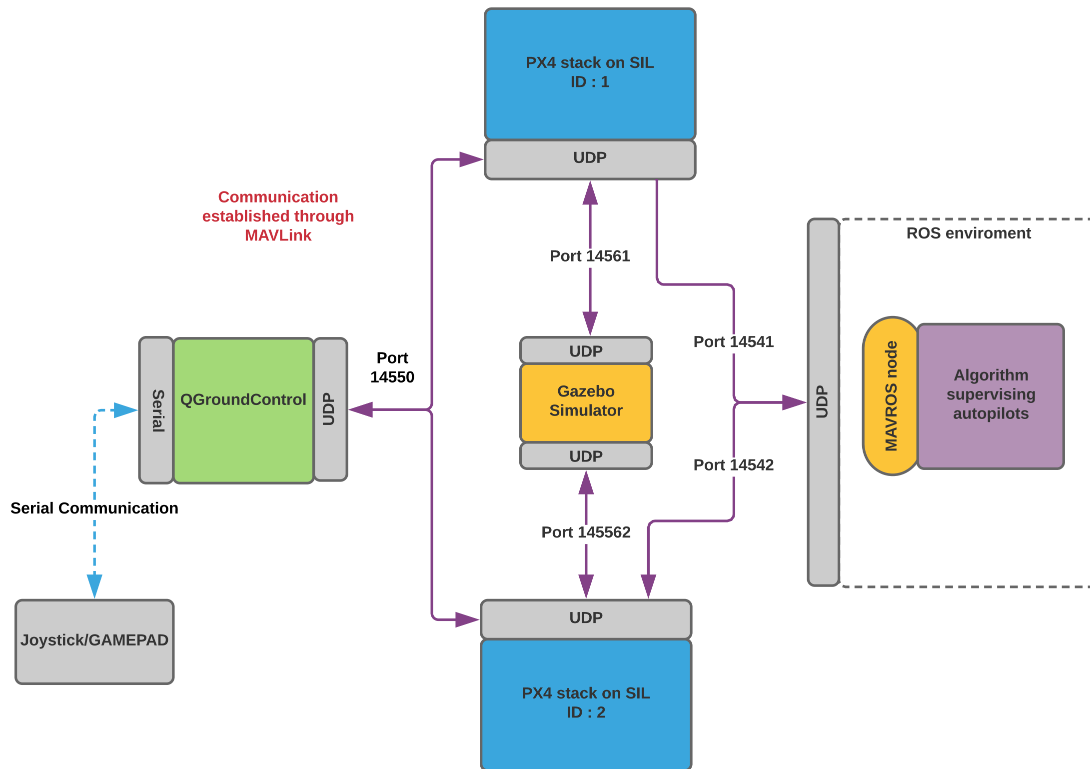

Figure 4.2: SIL setup

4.2.4 Logic for SIL.

ROS gets information from both autopilots an supervise its operation, to simulate a failure in one autopilot its daemon process is stopped, by doing so PX4 (one autopilot) loses communication with Gazebo and ROS. The redundant algorithm interprets this as a failure and immediately executes a safety routing, as it was already mentioned in subsection 3.4.4.

The original approach for the simulation was to have two autopilots connected to just one Drone running in Gazebo and to test the redundant algorithm. Unfortunately when each PX4 is launched this one creates automatically its own drone and connects to it. Due to this the redundant algorithm is tested having two drones and not just one.

It's important to point out that both autopilots when instanced must have different and unique IDs, otherwise when the simulation is launched error would pop-up saying that both autopilots can not exist at the same time in the same name space. Launch files are used to assign different ID to each autopilots as well as the UDP ports that each autopilot uses to connect with the different parts of the system. Next Fig. 4.3 depicts how the

simulation environment looks like.

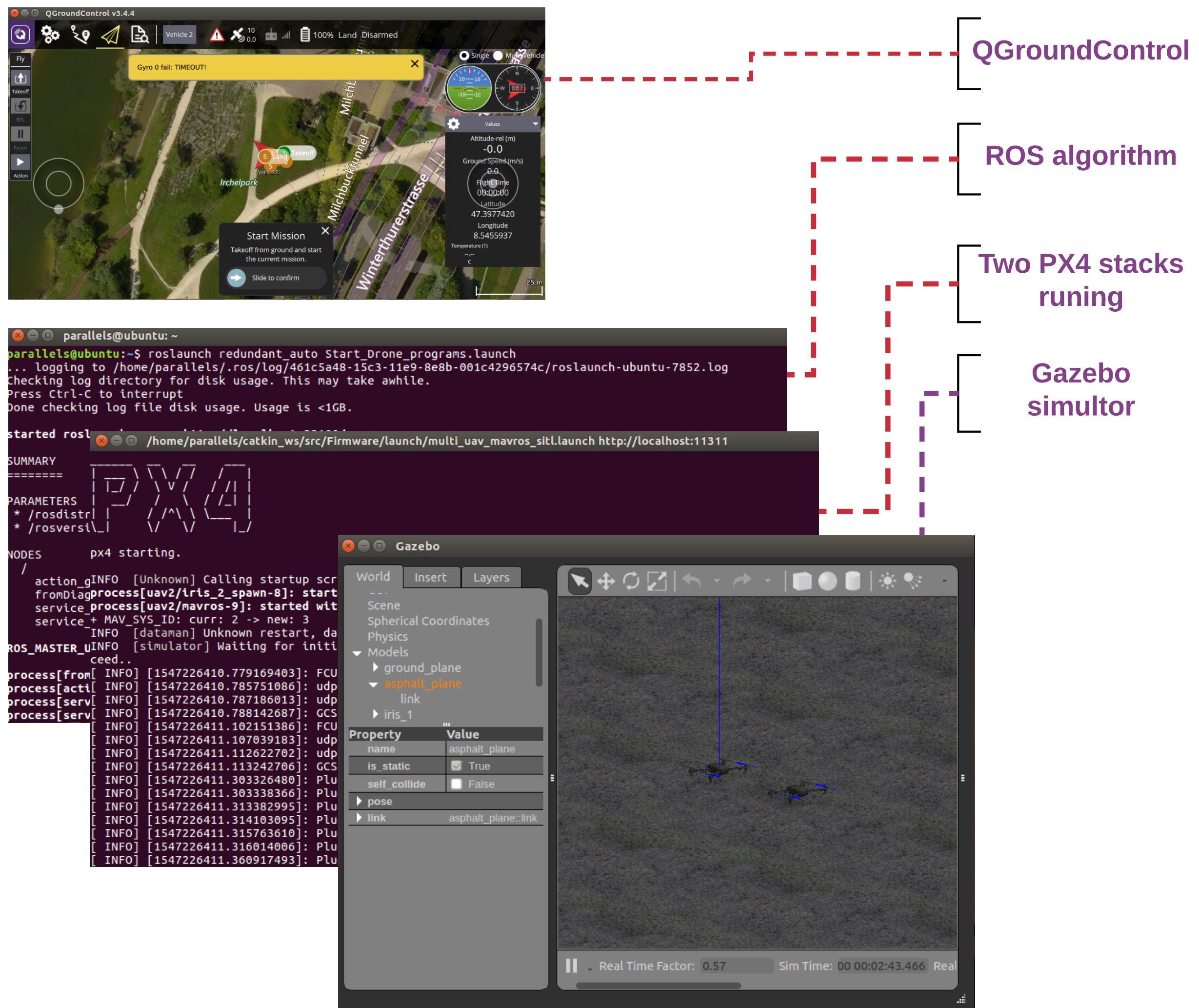

Figure 4.3: Simulation environment

Next point are checked in the listed order to validate the correct operation of the redundant system.

- Data coming from the autopilots.
  Here attention is focused on `/diagnostics` topic to check if it supplies correctly the flags mentioned in table 3.4. For this task the ROS commands `rostopic echo /diagnostics` and `rosrun rqt_runtime_monitor rqt_runtime_monitor` are used (see Fig. 4.4).

Figure 4.4: Test for /diagnostics

- Communication with QGroundControl. Both autopilots connects automatically to QGroundControl and all the parameters supplied by the autopilots are verified here, namely those related with location provided with the GPS, barometer, accelerometers and gyroscopes (see Fig. 4.5).

- Debugging the Redundant algorithm. The algorithm was compiled several times to correct different errors in the syntax. Qt creator is the compiler chosen for debugging the code.

- Safety routings.
  When a failure is simulated safety routing is executed in Gazebo, the drone with the redundant autopilot executes a safety landing when a failure occurs.

Figure 4.5: Test connection with QGroundControl

- Controlling the drone from Ground Station.
  Here the programs of the ground computer are tested. This programs control de drone when the RC control is not used.

All the previous task were tested in the same computer with all the components in the same network.

4.3 HIL phase.

Next Fig. 4.6 shows the HIL setup. Each Pixhawk autopilot is connected to the USB ports of the companion computer. The autopilots use serial communication to send data to the exterior world, due to this reason two FTDI modules are used to interface the serial TELM2 port from the autopilots to the USB ports of the companion computer. In addition an extra computer and a phone are used as Ground station. Let's remember that the Ground station is used as an extra means of control when the RC control in not being used and the phone can be used to monitor data coming from the autopilots. The companion computer creates a Hostpot to connect with the rest of the devices in the same network. Fig. 4.7 depicts the network setup.

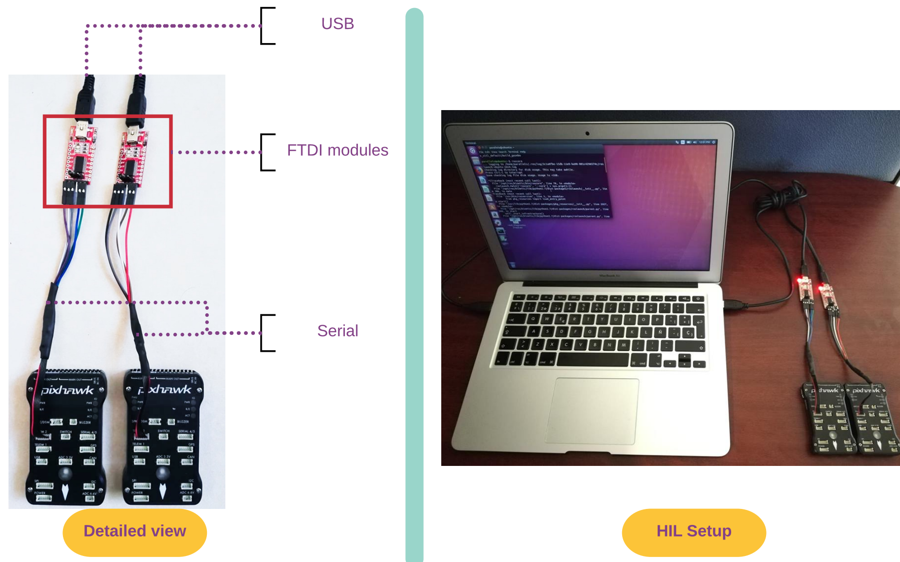

Figure 4.6: HIL setup

Spatial attention is focused on the MAV_SYS_ID parameter from each autopilot. As it was already mentioned each autopilot must have a unique ID, when simulating this parameter is set using launch files while when working with a real autopilot this parameter is set using MAV_SYS_ID parameter. To change the parameter QGroundControl must be used.

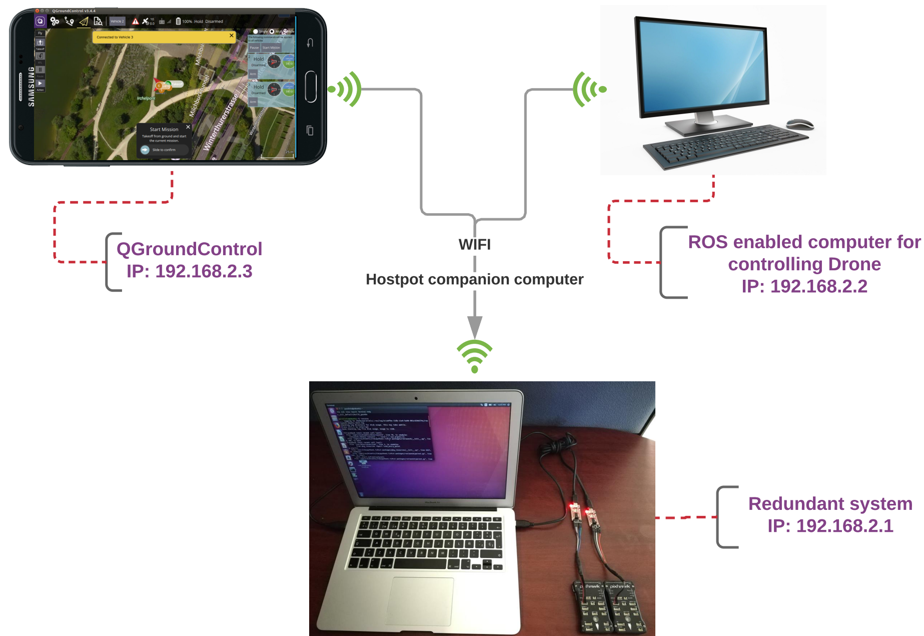

Figure 4.7: Network setup

4.3.1 HIL environment

The following Fig. 4.8 depicts the connections between all the components in the HIL phase. ROS gets information from the autopilots through UDP ports, forwards data to QGroundControl and connects to a Ground Station Computer, as long as all the devices are in the same network.

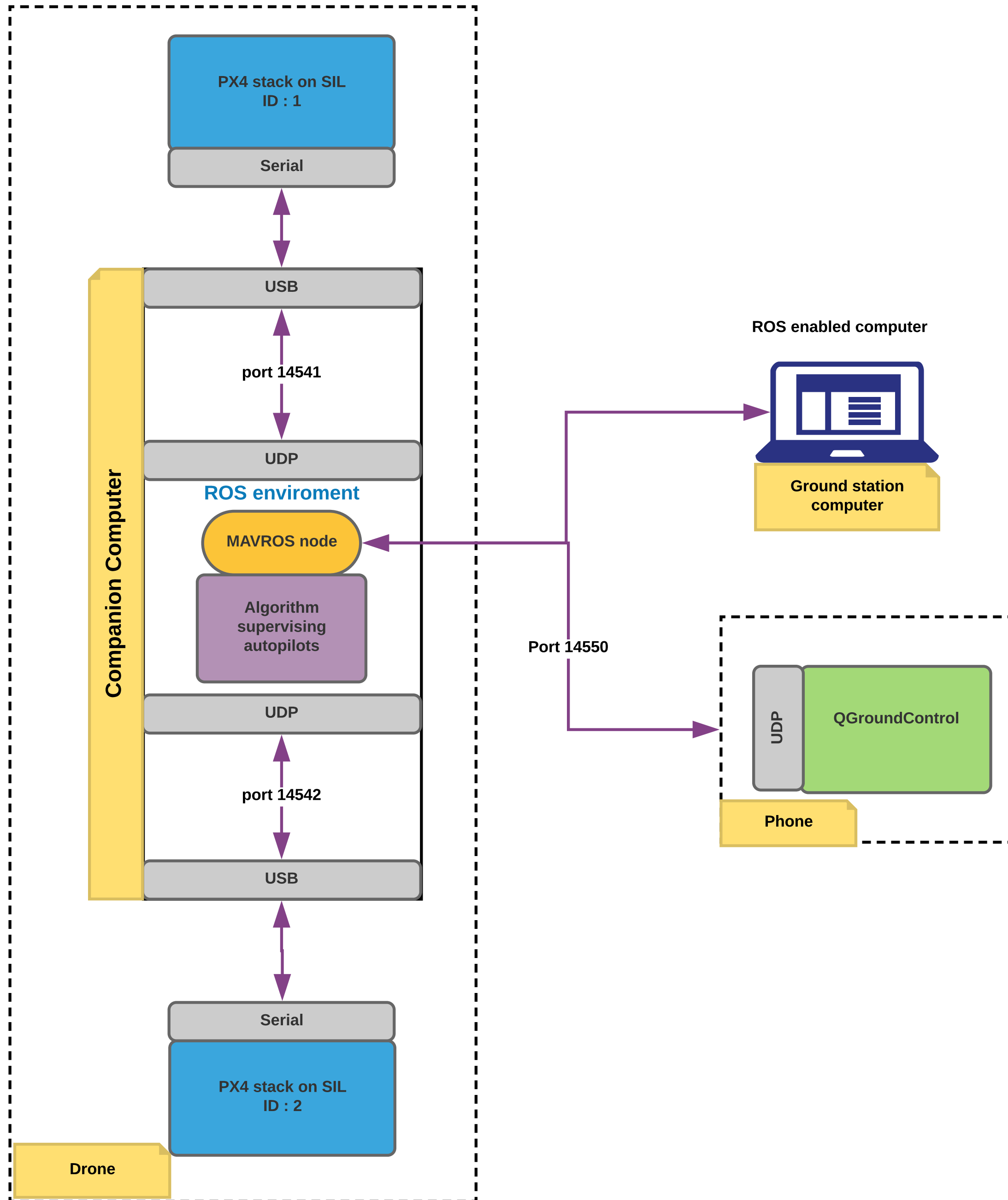

Figure 4.8: HIL setup and components connection.

Next point are checked in the listed order to validate the correct operation of the redundant system.

- Communication with the Autopilots.
  Firstly the connection was tested first using the MAVLink protocol (see Fig. 4.9):
  `mavproxy.py --master=/dev/ttyUSB0 --baudrate 921600 --aircraft MyCopter`

  This last command allows communication with the autopilot (connected in port USB0) using MAVLink commands. To test the second autopilot it is only needed to use /dev/ttyUSB1.

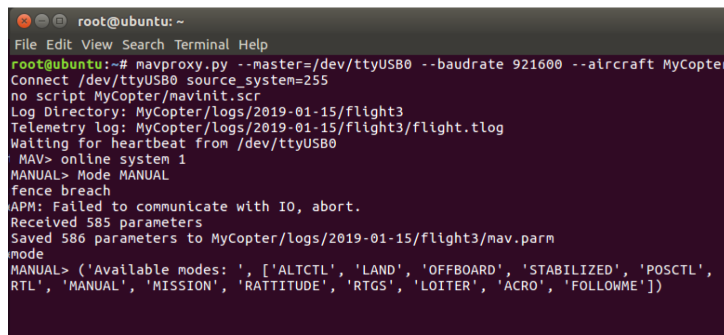

Figure 4.9: First test for communication

  Secondly it was established communication between MAVROS and the autopilots. In order to allow communication it is necessary to configure the ports trough which the companion computer uses to communicate with the autopilots, to learn how to do it the reader can check appendix B.

  After establishing connection between MAVROS and the autopilots, each one of the flags mentioned in Table. 3.4 are checked one by one to make sure that the communication with the autopilots works correctly and to make sure that ROS can supervise autopilots using `/diagnostics` topic (see Fig. 4.10).

- Communication with QGroundControl.
  The companion computer forwards the data coming from the autopilots to QGroundControl. This communication is checked focusing attention on the data received by QGroundControl, which is installed in a common phone. QGroundControl and the companion computer are set in the same network.

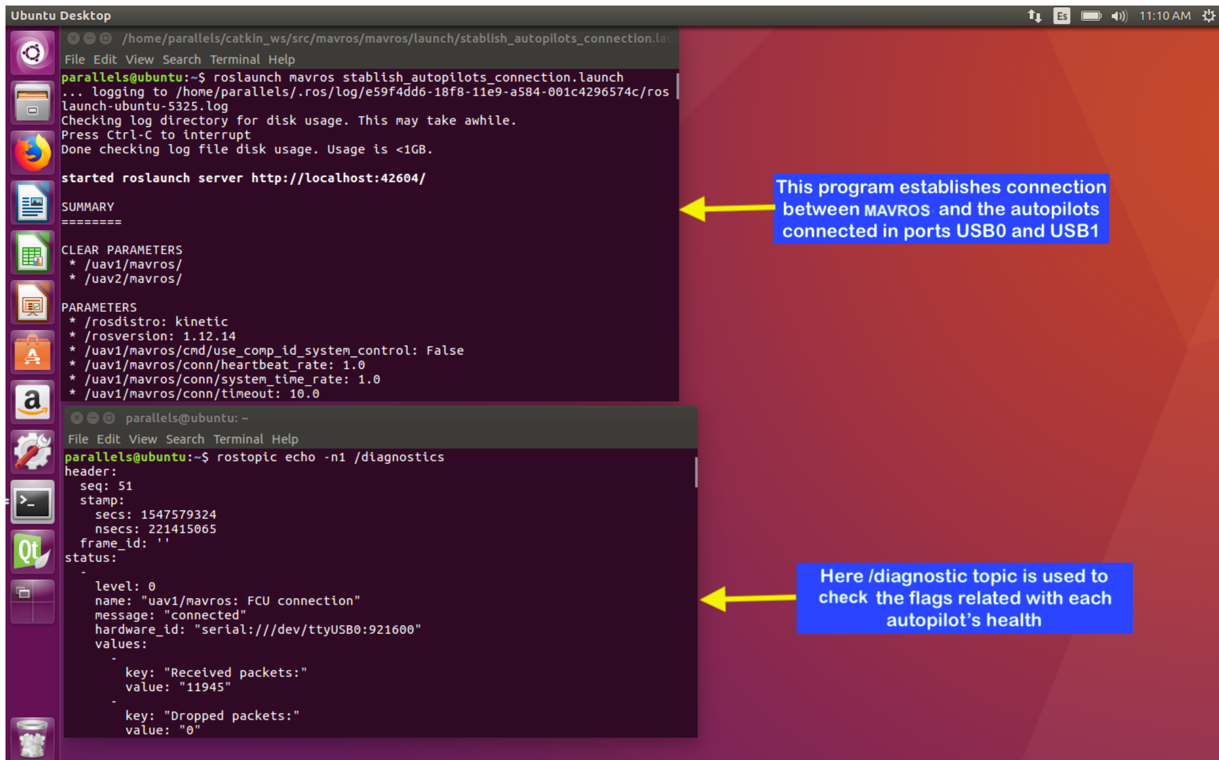

Figure 4.10: Second test for communication

- Redundant system.

  The safety routine is tested to make sure it runs when a failure has happened. In order to simulate a failure next arrangement is established.

- Communication with ground station.

  Connection is supervised by controlling the drone from a remote place. Do not forget that the companion computer and the ground station are always in the same network.

- Controlling the drone from ground station.

  In this step the drone is controlled by sending commands from QGroundControl installed on a phone.

4.4 Setting up the SIL

Before running any simulation ROS, MAVROS, MAVLink and the PX4 must be already install in the computer. The reader can install this programs one by one but PX4 development page provides a script to install all this programs by executing a file called `ubuntu_sim_ros_gazebo.sh`, appendix B shows how to use this script.

PX4 install all it's source code in folder called Firmware it's needed to make sure that this folder, MAVROS and MAVlink folders are allocated withing ROS workspace (see Fig. 4.11).

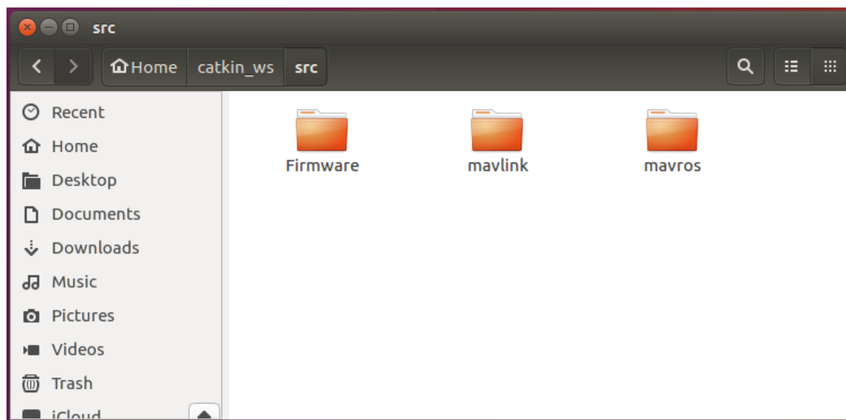

Figure 4.11: Firmware, MAVROS and MAVlink folder in SRC workspace folder

4.4.1 Launch files

.

ROS uses the so called launch files. Launch files are very common in ROS to both users and developers. They provide a convenient way to start up multiple nodes and a master, as well as other initialization requirements such as setting parameters. In this case they are used to start a SIL PX4.

PX4 has already refined launch files to start a simulation (see Fig. 4.12). Use roslaunch command to start any of them, e.g: `roslaunch px4 mavros posix sitl.launch`

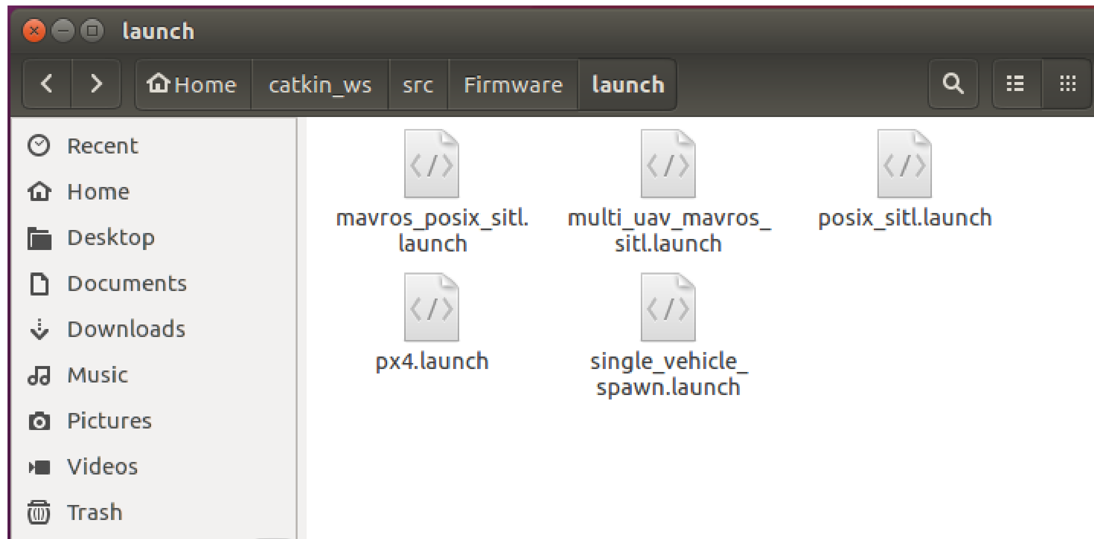

Figure 4.12: Example default PX4 Lauch files

It is important to take into account that before launching any of these lauch files the user must source the environment.

```
source  ~/catkin_ws/devel/setup.bash
cd ~/catkin_ws/src/Firmware
source Tools/setup_gazebo.bash $(pwd) $(pwd)/build/posix_sitl_default
export ROS_PACKAGE_PATH=$ROS_PACKAGE_PATH:$(pwd):$(pwd)/Tools/sitl_gazebo
```

4.4.2 Launch files created for the project

.

This project simulates two autopilots and to do so, it was created three launch files. All three have comments for a better understanding of the code.

- my_main_launch.launch. It configures and starts the Gazebo simulator
- my_uav1_mavros_sitl.launch. It starts PX4 and spawn a drone.
- my_uav2_mavros_sitl.launch. It starts a second PX4 and spawn a drone.

my_main_launch.launch

```
my_main_launch.launch (~/catkin_ws/src/Firmware/launch) - gedit
<?xml version="1.0"?>
<launch>
    <!-- PX4 SIL environment launch script -->
    <!-- Launches Gazebo environment -->
    <!-- vehicle model and world -->
    <arg name="est" default="ekf2"/>
    <arg name="vehicle" default="iris"/>
    <arg name="world" default="$(find mavlink_sitl_gazebo)/worlds/empty.world"/>
    <!-- gazebo configs -->
    <arg name="gui" default="true"/>
    <arg name="debug" default="false"/>
    <arg name="verbose" default="false"/>
    <arg name="paused" default="false"/>
    <!-- Gazebo sim -->
    <include file="$(find gazebo_ros)/launch/empty_world.launch">
        <arg name="gui" value="$(arg gui)"/>
        <arg name="world_name" value="$(arg world)"/>
        <arg name="debug" value="$(arg debug)"/>
        <arg name="verbose" value="$(arg verbose)"/>
        <arg name="paused" value="$(arg paused)"/>
    </include>

</launch>
<!-- This launch file must be the fisrt launch file
     If it's stopped subsecuente launch files will also be stopped -->
```

Figure 4.13: my_main_launch.launch launch file

my_uav1_mavros_sitl.launch and my_uav2_mavros_sitl.launch

```
my_uav1_mavros_sitl.launch (~/catkin_ws/src/Firmware/launch) - gedit
<?xml version="1.0"?>

<!-- This launch file:  -->
<!--   -Enables conection with MAVROS in UDP port: 14541  -->
<!--   -Initialize PX4 and spawns one drone  -->
<!--   -Enables communication with Gazebo in UDP port: 1461 -->

<launch>
    <arg name="vehicle" default="iris"/>
    <!-- (!) IMPORTANT (!) This PX4 works in name space UAV1  -->
    <group ns="uav1">
        <!-- PX4 ID and UPD (fcu_url) to connect with MAVROS -->
        <arg name="ID" value="1"/>
        <arg name="fcu_url" default="udp://:14541@localhost:14581"/>
        <!-- STARTS PX4 SIL and vehicle spawn -->
        <include file="$(find px4)/launch/single_vehicle_spawn.launch">
            <arg name="x" value="0"/>
            <arg name="y" value="0"/>
            <arg name="z" value="0"/>
            <arg name="R" value="0"/>
            <arg name="P" value="0"/>
            <arg name="Y" value="0"/>
            <arg name="vehicle" value="$(arg vehicle)"/>
            <arg name="mavlink_udp_port" value="14561"/>  <!-- CONNECT PX4 WITH GAZEBO IN PORT 14561 -->
            <arg name="ID" value="$(arg ID)"/>
        </include>
        <!-- STARTS MAVROS -->
        <include file="$(find mavros)/launch/px4.launch">
            <arg name="fcu_url" value="$(arg fcu_url)"/>
            <arg name="gcs_url" value=""/>
            <arg name="tgt_system" value="$(eval 1 + arg('ID'))"/>
            <arg name="tgt_component" value="1"/>
        </include>
    </group>

</launch>
<!-- to add more UAVs (up to 10):
 -Increase the id
 -Change the name space
 -Set the FCU to default="udp://:14540+id@localhost:14550+id"
 -Set the malink_udp_port to 14560+id) -->
```

Figure 4.14: my_uav1_mavros_sitl.launch launch file

The launch file for my_uav2_mavros_sitl.launch is exactly the same, next parameter changes:

- <group ns="uav2" />
- <arg name="ID" value="2"/>
- <arg name="fcu_url" default="udp://:14542@localhost:14582"/>
- <arg name="mavlink_udp_port" value="14562"/>

Not not forget that my_uav1_mavros_sitl and my_uav2_mavros_sitl establishes connection between Gazebo and MAVROS on specifics UDP ports. It is not necessary to configure the UDP port for QGroundControl since when simulating this last one connects with PX4 automatically.

4.4.3 Extra information

Next information provides a better understating about the launch files.

- Gazebo model.
  This is defined as xacro file in Firmware/Tools/sitl_gazebo/models /rotors_description /urdf/<model>_base.xacro. Currently, the model xacro file is assumed to end with base.xacro. This model should have an argument called mavlink_udp_port which defines the UDP port on which gazebo will communicate with PX4 node. The model's xacro file will be used to generate an urdf model that contains UDP port that you select. To define the UDP port, set the mavlink_udp_port in the launch file for each vehicle, see here as an example.

- PX4 node.
  This is the SITL PX4 app. It communicates with the simulator, Gazebo, through the same UDP port defined in the Gazebo vehicle model, i.e. mavlink_udp_port. To set the UDP port on the PX4 SITL app side, you need to set the SITL_UDP_PRT parameter in the startup file to match the mavlink_udp_port discussed previously, see here. The path of the startup file in the launch file is generated based on the vehicle and ID arguments, see here. The MAV_SYS_ID for each vehicle in the startup file, should match the ID for that vehicle in the launch file. This will help make sure you keep the configurations consistent between the launch file and the startup file.

- MAVROS node.
  A seperate MAVROS node can be run in the launch file, in order to connect to PX4 SITL app. It is needed to start a MAVLink stream on a unique set of ports in the

startup file. Those unique set of ports need to match those in the launch file for the MAVROS node.

4.4.4 Running a simulation with launch files

.

Open three different shell and execute roslaunch command on the three launch files already mentioned in section 4.4.2. Do not forget that the user must source the environment. After executing the launch file simulation looks like next Fig. 4.15

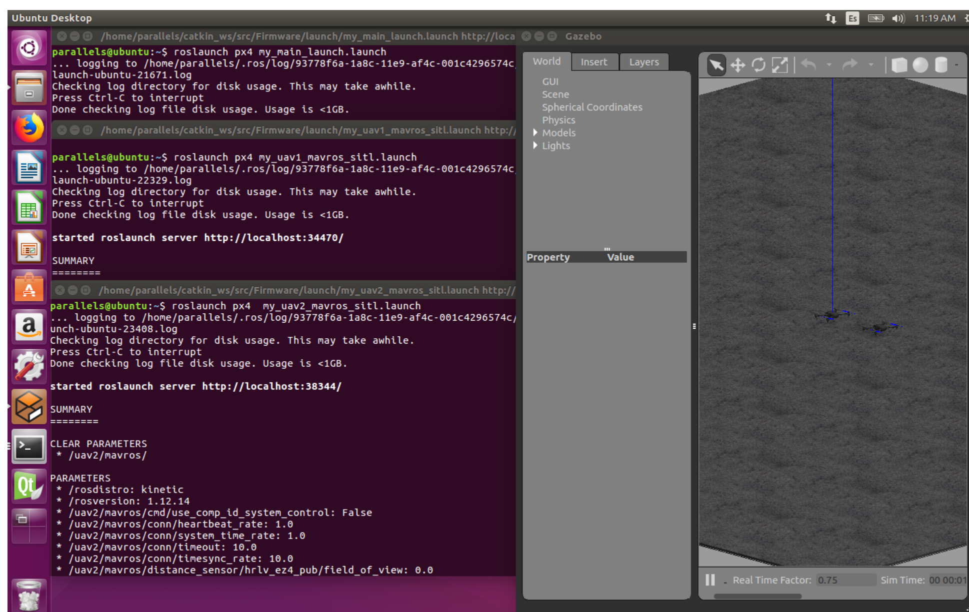

Figure 4.15: Running Simulation

4.5 Setting up the HIL

In order to enable communication between an autopilot and a companion computer it is needed to have installed MAVROS, MAVLink and MAVProxy to route MAVLink between serial and UDP. Section B describes the steps necessary to install all these programs. Pixhawk uses by default the TELM2 serial port to communicated with the exterior world. The companion computer can connect to this port by using its built-in UART or by using its USB port. Let's take into account that when using the USB port a FTDI module most be used to translate serial communication from the TELM2 port into USB protocol (See Fig. 4.16).

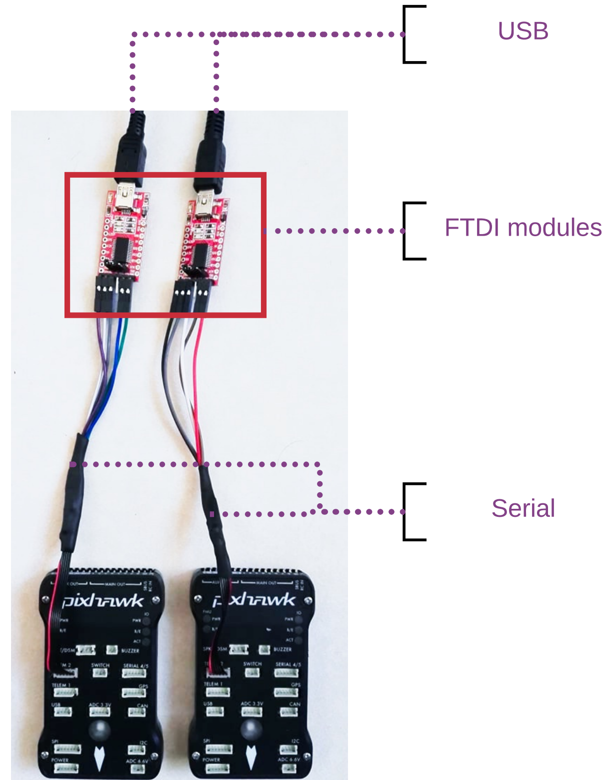

Figure 4.16: FDTI module to translate from serial to USB protocol

Fig. 4.17 depicts a typical connection using the UART port from a Raspberry PI. Unfortunately raspberry pi only has one built-in UART port and for these reason it was decided to uses two FTDI modules since two autopilots have to be connected to the companion computer at the same time.

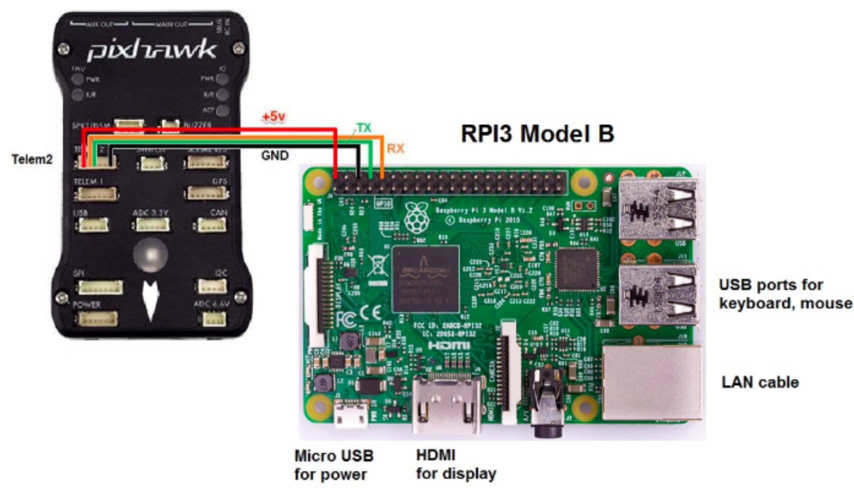

Figure 4.17: Serial connection between Pixhawk and Companion computer

4.5.1 Hardware setup

Pixhawk series uses commonly the TELM2 port for Companion computer communication, depending on the hardware version used this port may change, the reader can check PX4 Development Guide for more information about this matter. Next Table 4.2 show the typical connection used for connecting the FTDI module to the TELM2 port.

Table 4.2: Wiring to Pixhawk (FTDI Chip USB-to-serial adapter board)

TELEM2		FTDI	
1	+ 5V (red)		NOT CONECTED!
2	Tx (out)	5	FTDI RX (yellow) (in)
3	Rx (in)	4	FTDI TX (orange) (out)
4	CTS (in)	3	FTDI RTS (green) (out)
5	RTS (out)	2	FTDI CTS (brown) (in)
6	GND	1	FTDI GND (black)

All Pixhawk serial ports operate at 3.3V and are 5V level compatible.

4.5.2 Software setup

4.5.2.1 Pixhawk

Before using the TELEM2 port it must be first configured by enabling a MAVLink instance on this port, this allows communication with a companion computer. To set up the default companion computer message stream on TELEM 2, set the following parameters.

- MAV_1_CONFIG = TELEM 2 (MAV_1_CONFIG is often used to map the TELEM 2 port)
- MAV_1_MODE = Onboard
- SER_TEL2_BAUD = 921600 (921600 or higher recommended for applications like log streaming or FastRTPS)

Depending on the hardware version used the parameter that need to configured may change, for Pixhawk version 1 only the parameter SYS_COMPANION to 921600.

How to set the parameters on PORT2

All the serial drivers/ports are configured in the same way (see Fig. 4.18).

1 Set the configuration parameter for the service/peripheral to the port it will use.

2 Reboot the vehicle.

3 Set the baud rate parameter for the selected port to the desired value.

4 Configure module-specific parameters (i.e. MAVLink streams and data rate configuration).

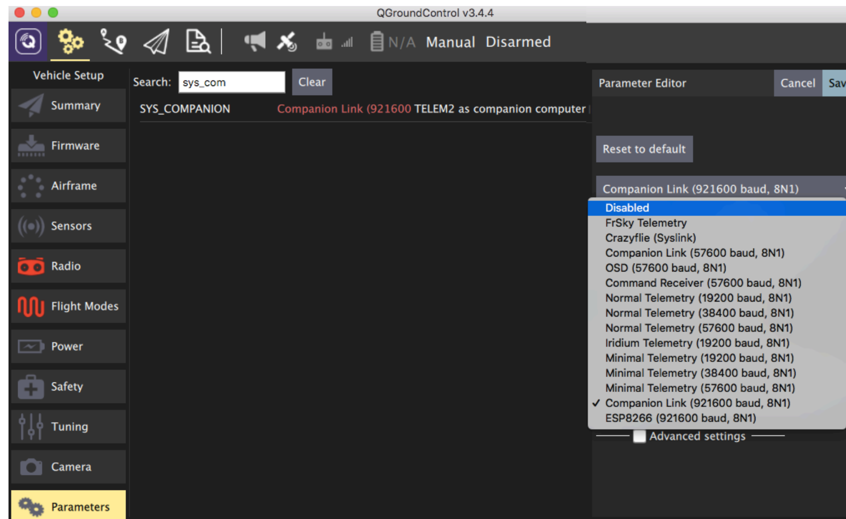

Figure 4.18: Configuring parameter using QGroundControl

4.5.2.2 Companion computer

The companion computer is expected to have installed ROS kinetic running on ubuntu 16.04, since it is the long term support version for Ubuntu 16.04.

A launch file is used to allow communication between Pixhawk and the companion computer. This launch file configures the UDP port through which MAVROS obtains data coming from the MAVProxy. Let's remember MAVProxy obtains data coming from USB ports in which the autopilots are connected.

The launch file, called stablish_autopilots_connection.launch (see Fig. 4.19), is allocated in the launch folder within the MAVROS folder. This file sets next points.

- There are declared two name spaces, each autopilots sends data to each one of them, namely uav1 and uav2. It is important to declare them like this, otherwise data conflict appears.
- Connection is established for port USB0 and USB1 at a baud rate of 921600. Line of codes for one autopilot: arg name="fcu_url" default="serial:///dev/ttyUSB0:921600"
- MAVROS forwards data received from Pixhawk to QGroundControl station with the IP 10.42.0.218 on port 14550.
  Line of code: arg name="gcs_url" default="udp://@10.42.0.218:14550"

To run the launch file it only needed to execute roslaunch command (`roslaunch mavros stablish_autopilots_connection.launch`), do not forget that the environment must be sourced before running roslaunch (`source  /catkin_ws/devel/setup.bash`). Communication must be already established after executing the launch file. Command `rosnode list` can be use to validate communication.

4.5.2.3 Ground Station Computer

Let's remember that the package redundant_auto runs in the companion computer and package ground_station runs in a Ground Station Computer, both computers must be in the same network and in the same ROS environment. Next instruction defines the step that must be fulfill in order to have a proper connection between the Ground Station Computer and the Companion computer.

The companion computer (drone) should have run the ROS master and have defined a hostname (name by which it can be addressed in the network).

- export ROS_HOSTNAME=10.0.0.1 #Companion computer's name
- export ROS_IP=10.0.0.1 #Companion computer's own IP
- roscore

The same commands must be entered in the Ground Computer.

- ROS_MASTER_URI=http://10.0.0.1:11311 #Drone's IP (Companion cumputer)
- export ROS_IP=10.0.0.2 #Ground computer's own IP

```
<launch>
	<!-- This file stablishes conection between the autopilots and ROS-->

	<!-- Data stream flows as follows:-->
	<!-- AUTOPILTO 1(Serial port) ====> FTDI module ====>Computer USB port (USB0) ====> MAVproxy
ROS -->
	<!-- AUTOPILTO 2(Serial port) ====> FTDI module ====>Computer USB port (USB1) ====> MAVproxy
ROS -->


	<!-- Stablishes conection with first Autopilot, Name space:uav1-->
	<!-- Forwards data to QGroundControl with IP: 10.42.0.218 -->
	<group ns="uav1">
	<arg name="fcu_url" default="serial:///dev/ttyUSB0:921600" />
	<arg name="gcs_url" default="udp://@10.42.0.218:14550" />
	<arg name="tgt_system" default="1" />
	<arg name="tgt_component" default="1" />
	<arg name="log_output" default="screen" />
	<arg name="fcu_protocol" default="v2.0" />
	<arg name="respawn_mavros" default="false" />

	<include file="$(find mavros)/launch/node.launch"> <!-- Creates first ROS node -->
		<arg name="pluginlists_yaml" value="$(find mavros)/launch/px4_pluginlists.yaml" />
		<arg name="config_yaml" value="$(find mavros)/launch/px4_config.yaml" />

		<arg name="fcu_url" value="$(arg fcu_url)" />
		<arg name="gcs_url" value="$(arg gcs_url)" />
		<arg name="tgt_system" value="$(arg tgt_system)" />
		<arg name="tgt_component" value="$(arg tgt_component)" />
		<arg name="log_output" value="$(arg log_output)" />
		<arg name="fcu_protocol" value="$(arg fcu_protocol)" />
		<arg name="respawn_mavros" default="$(arg respawn_mavros)" />
	</include>
	</group>

	<!-- Stablishes conection with second Autopilot, Name space:uav2-->
	<!-- Forwards data to QGroundControl with IP: 10.42.0.218 -->
	<group ns="uav2">
	<arg name="fcu_url" default="serial:///dev/ttyUSB1:921600" />
	<arg name="gcs_url" default="udp://@10.42.0.218:14550" />
	<arg name="tgt_system" default="1" />
	<arg name="tgt_component" default="1" />
	<arg name="log_output" default="screen" />
	<arg name="fcu_protocol" default="v2.0" />
	<arg name="respawn_mavros" default="false" />

	<include file="$(find mavros)/launch/node.launch"> <!-- Creates second ROS node -->
		<arg name="pluginlists_yaml" value="$(find mavros)/launch/px4_pluginlists.yaml" />
		<arg name="config_yaml" value="$(find mavros)/launch/px4_config.yaml" />

		<arg name="fcu_url" value="$(arg fcu_url)" />
		<arg name="gcs_url" value="$(arg gcs_url)" />
		<arg name="tgt_system" value="$(arg tgt_system)" />
		<arg name="tgt_component" value="$(arg tgt_component)" />
		<arg name="log_output" value="$(arg log_output)" />
		<arg name="fcu_protocol" value="$(arg fcu_protocol)" />
		<arg name="respawn_mavros" default="$(arg respawn_mavros)" />
	</include>
	</group>
</launch>
```

Figure 4.19: stablish_autopilots_connection.launch file to stablish connection with MAVROS

Chapter 5

Results

The following chapter show the outcome from the SIL and HIL phases. The results obtained from the tests are related with the communication between each autopilot and the companion computer, data transmission from the autopilots to QGroundControl and to a Ground Control Station, and finally the results obtained from the algorithm that scans autopilots status.

5.1 Communication with autopilots

Data flow received from the autopilots correspond to the ones expected. They show the current status of both autopilots. Next figure Fig. 5.1 depicts the output of commands: `rostopic echo /diagnostics` and `rosrun rqt_runtime_monitor rqt_runtime_monitor`. Table 5.1 shows data obtained.

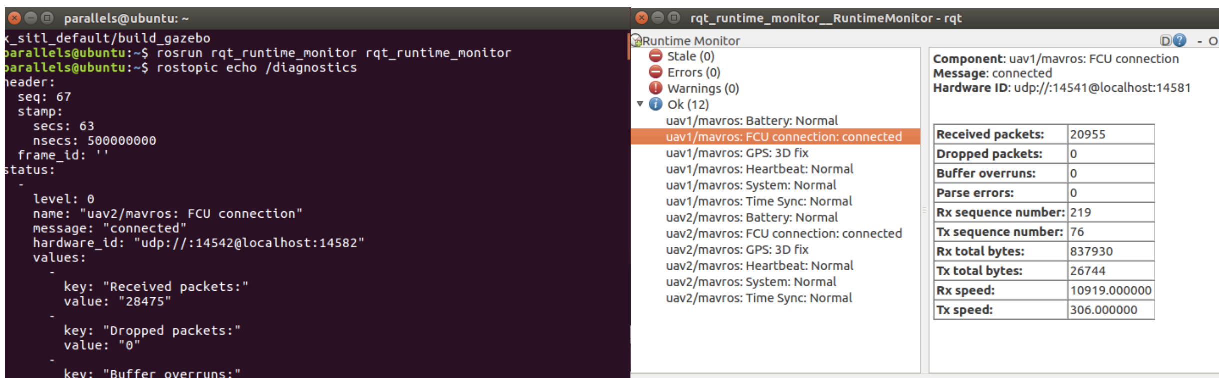

Figure 5.1: Data obtained from autopilots

Table 5.1: Data coming from each autopilot

DATA		Description	Possible Values
FCU_connection		Shows status of connection between MAVROS and the px4 stack	OK=0 WARN=1 ERROR=2 STALE=3
GPS			
Heartbeat_Status		The heartbeat is used to determine whether a system is connected, and to detect when it has disconnected.	
Battery		Show data related with battery, current and voltage	
Time Sync		Rate transmission from USB port	
System	Gyroscope	Status of gyroscope embedded sensor	
	Accelerometer	Status of accelerometer embedded sensor	
	Magnetometer	Status of magnetometer embedded sensor	

5.2 Data transmission

5.2.1 QGroundControl

Connection with QGroundControl works correctly showing data coming from both autopilots at the same time. Data received is related with:

- GPS
- Altitude
- Acceleration
- Battery
- Vibration
- Temperature

5.2.2 Ground Control Station

When the drone is started, it immediately runs the action server /action_getsAutoStatus_server that wait for a call from ground station. Next Fig. 5.2 shows the output obtained which represents the steps followed by the companion computer on board.

```
/home/parallels/pruebas_ROS/src/redundant_auto/launch/Start_Drone_programs.launch http://localho
[INFO] [1547679830.194561, 7.324000]: :::::::::::::: /action_getsAutoStatus_server :::::::::
::
[INFO] [1547679830.195071, 7.324000]:                    INITIALIZED
[INFO] [1547679830.195672, 7.324000]: Server ===>>> Waiting call from Ground Station ...
```

Figure 5.2: ROS starts action server and wait for a call from ground station.

Afterwards a call from the Ground station is made the next outputs, show in Fig. 5.3, were obtained. They show that the connection works properly. Let's remember that the ground station is in the same network along with the companion computer.

```
/home/parallels/pruebas_ROS/src/redundant_auto/launch/Start_Drone_programs.launch
[INFO] [1547742266.334861, 26.344000]: :::::::::::::: /action_getsAutoStatus_ser
ver ::::::::::
[INFO] [1547742266.336076, 26.346000]: Server ===>>> New call received ...
[INFO] [1547742266.353271, 26.354000]: Server ===>>> Checking Autopilots ...
[INFO] [1547742267.748457, 27.360000]: Server ===>>> Autopilot 1: OK
[INFO] [1547742267.749124, 27.360000]: Server ===>>> Autopilot 2: OK
[INFO] [1547742267.749903, 27.360000]: Server ===>>> Sending response ...
auto1: True
auto2: True
Gstatus: True
[INFO] [1547742267.750470, 27.360000]: Server ===>>>    ****Waiting for new call
 from Ground Station***
```

a) Response from companion computer

```
/home/parallels/pruebas_ROS/src/redundant_auto/launch/Start_Main_Menu.launch http://localho
[INFO] [1547742266.250966, 26.316000]: :::: CLIENT /action_getsAutoStatus_server :::::
[INFO] [1547742266.261342, 26.320000]: Client ===> Waiting conection with Drone...
[INFO] [1547742266.314425, 26.330000]: Client ===> Conection stablished
[INFO] [1547742266.315707, 26.330000]: Client ===> Requesting Autopilots Status
[INFO] [1547742266.317932, 26.330000]: Client ===> Waiting for response...
[WARN] [1547742269.027028, 28.330000]: Client ==> There is a warning in the Server side
(/action_getsAutoStatus_server)
auto1: True
auto2: True
Gstatus: True
[INFO] [1547742269.027616, 28.330000]: Initiating Main Menu
INICION
```

b) Call from Ground Station

Figure 5.3: Call from Ground station and response from Companion computer

After a connection with the companion computer and ground station is established, next

outputs are obtain. They are shown in Fig. 5.4.

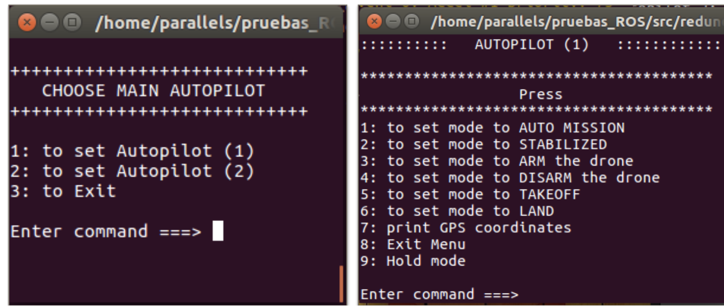

Figure 5.4: Menu from Ground station

The following commands were tested.

- Take off
- Altitude
- Auto mission
- Set stabilized mode
- Arm drone
- Disarm drone
- Land

When an error is simulated and connection with the drone is made the next menu pops up (see Fig. 5.5).

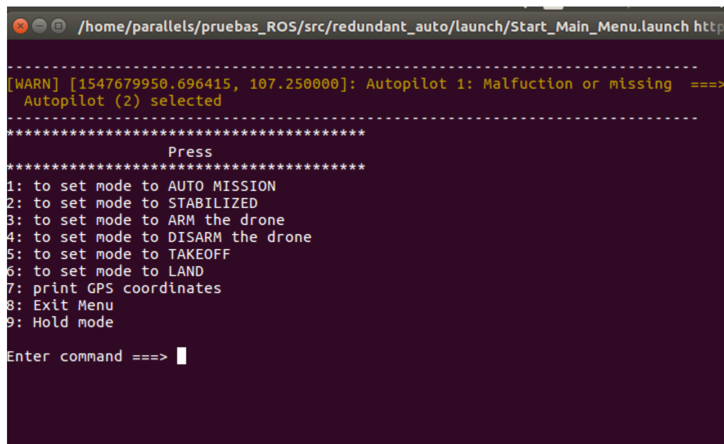

Figure 5.5: Warning obtained when one autopilot has an error and connection between Ground station and companion Computer is made.

5.3 Algorithm

The algorithm was tested firstly in the SIL and validated in the HIL phase.

5.3.1 SIL

The algorithm was tested by stopping the daemon process from each autopilot within the simulation, by doing so the algorithm interprets this as a failure in the system and immediately executes the corresponding safety routing.

Immediately after starting up the Drone it runs the services that continuously scans the autopilots status. Next Fig. 5.6 depicts a shell with the steps followed by the service.

```
/home/parallels/pruebas_ROS/src/redundant_auto/launch/Start_Drone_programs.launch

[INFO] [1547679830.584483, 0.000000]: ::::::::::::: /service_inCaseEmergency_se
rver ::::::::::
[INFO] [1547679830.587020, 0.000000]:                         INITIALIZED
[INFO] [1547679830.587287, 0.000000]: service_inCaseEmergency_server ===>>> Wait
ing for a call...
[INFO] [1547679830.621379, 7.646000]: Client ==> Conected to /service_inCaseEmer
gency_server
[INFO] [1547679830.622948, 7.648000]: Client ==> Scanning autopilots status ...
```

Figure 5.6: Drone starts service in Case Emergency and waits for a failure.

When detecting an error PX4 changes to HOLD mode and after 3 seconds starts a safety landing. Next Fig. 5.7 shows the response of the algorithm when an error has been detected in one autopilot.

```
/home/parallels/pruebas_ROS/src/redundant_auto/launch/Start_Drone_programs.launch

----------------------------------------------
                 SAFETY ROUTING
----------------------------------------------
[ERROR] [1547679847.139546, 18.006000]: Autopilot Error encountered
[WARN] [1547679847.140152, 18.006000]: REDUNDANT AUTOPILOT:  --- Triggered ---
[WARN] [1547679847.147448, 18.010000]: Switching to Hold MODE
[INFO] [1547679847.179114, 18.030000]: HOLD MODE -->  Executed
[INFO] [1547679847.181296, 18.030000]: Drone will return home in 3 secs...
[INFO] [1547679850.731972, 21.038000]: Initiating Safety Landing
[INFO] [1547679850.760159, 21.052000]: Land MODE -->  Requested
[INFO] [1547679850.760739, 21.052000]: Safety Routing ==> All steps executed wit
hout any problem
-----------------------------------------------------------------------------
[INFO] [1547679850.761333, 21.052000]: service_inCaseEmergency_server ===>>> Wai
ting for a new call...
-----------------------------------------------------------------------------
[INFO] [1547679850.762858, 21.052000]: Client ==> Emergency Routin :::Finished::
:
[service_inCaseEmergency_client_node-4] process has finished cleanly
log file: /home/parallels/.ros/log/ec242bfa-19e2-11e9-805e-001c4296574c/service_
inCaseEmergency_client_node-4*.log
```

Figure 5.7: Simulating and detecting error in one Autopilot

5.3.2 HIL

In order to test the system in the HIL phase it was used one autopilot in good condition and other one with malfunction in several sensors (see Fig. 5.8), namely the gyroscope and the accelerometer. The algorithm as usual scans both autopilots with the exception that does not supervise the complete set of flags in the autopilot which has sensor failure, it only scans all the flags when it is instructed to do so, by entering a key command given by the user. The algorithm scan the full set of flags in the healthy autopilot. This makes possible to check and supervise the response of the algorithm when it detects a failure.

The outcome from this test proved that the redundancy system works properly with a lag of 0.020 sec. It is expected that further works will improve the time of response of the algorithm.

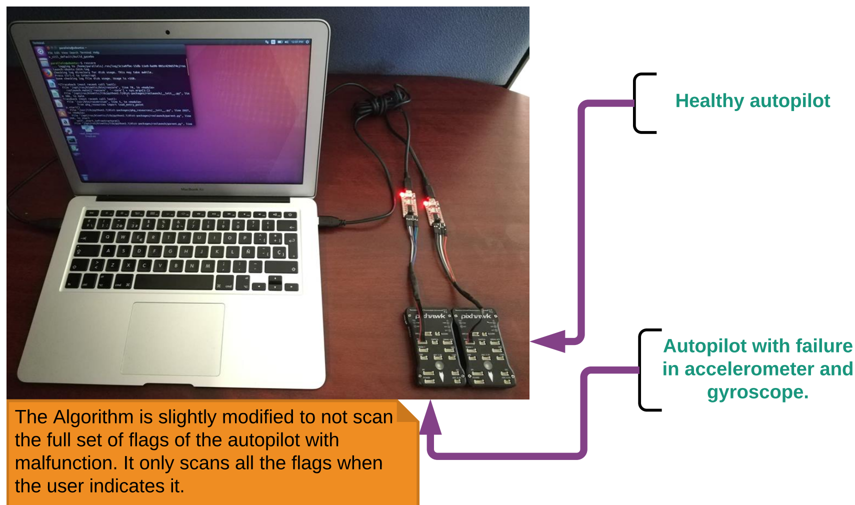

Figure 5.8: Test for HIL phase

Chapter 6

Conclusion

This work is based on the statement that redundancy increases the reliability of a drone. The hypothesis (shown in section 1.2) was achieved by having implemented and designed a hot-standby redundancy arrangement, this arrangement shortens the downtime, which in turn increases the availability of the system. In addition the statement it is supported by the already existing reliability theory. Adding a redundant autopilot to a drone permits to increase system safety and reliability. The safety offered by redundant autopilots is such that it is commonly used in several manned aircrafts and high performance unmanned systems.

The general objective (shown in section 1.4) was achieved as the redundant system hereby presented makes use of PX4 flight stack, which works as main software to control the autopilots hardware. Furthermore, ROS works as arbiter which manages two Pixhawk autopilots.

The next list shows the tasks carried out which fulfill the specific objectives shown in the chapter one (see section 1.4).

- Development and concept design of a redundant system which makes use of ROS and PX4 stack.
- Development of the algorithm that oversees the operation of the autopilots.
- Simulation of the system by using Gazebo.
- Deployment of a SIL stage followed by its validation implementing a HIL stage.
- Project's documentation.

6.1 Extra comments

The redundant system presented in this project uses ROS as arbiter which manages two Pixhawk autopilots. This redundant system not only increases safety and reliability but also adds high flexibility to a drone, by using "ROS" high scalability is achieved, as is possible to use the already existing and available packages, collection of tools and libraries among the ROS community. A drone using the redundant autopilot hereby presented can be enabled with characteristics such as:

- Object avoidance.
- Tracking.
- Mapping.
- Complex autonomous flights.
- Image recognition and more.

The redundant system also permits to independently manage both autopilots' telemetry and enables the possibility to install two independent communication links and GPS modules. This redundant autopilot can face the current situation in which drones are increasingly requiring a larger grade of safety, because of the need to operate in populated areas and due to the criticality of new applications focused on the civilian market.

6.2 Further work

Finally this chapter concludes by point out topics in which further work can focus attention.

- The reliability theory.
  The project makes use of a hot-standby redundancy arrangement and although it is based on the reliability theory there is still a lot work to do regarding mathematical matters to give this project a well establish and solid foundation.
- Modular redundancy.
  The modular redundancy arrangement it is known for having a better performance and a lower downtime that the standby redundancy. Furthermore, the reliability

factor has a better improvement in the modular redundancy than in the standby redundancy here used.

- Real time acquisition data from autopilots.
  Here Real Time Publish Subscribe (RTPS) can be used to improve data aquisition from autopilots. The PX4-FastRTPS Bridge adds a RTPS interface to PX4, enabling the exchange of uORB messages between PX4 components and (offboard) Fast RTPS applications.

- Making redundant autopilot compatible with Ardupilot.
  Nowadays PX4 and Ardupilot are the most popular a common used flight stacks, among the civilian market, by making the code compatible with Ardupilot stack the redundant system will be compatible with most of all the hardware available in the market.

- Extending code to acquire data from the complete Pixhawk series.
  Each autopilot from Pixhawk series have different flags thats shows its current status. It's necessary to add all this flags to the code since currently the flags used in the code are those related only with the first Pixhawk series, namely Pixhawk 1.

Bibliography

[1] F. Giones and A. Brem, "From toys to tools: The co-evolution of technological and entrepreneurial developments in the drone industry," *Business Horizons*, vol. 60, no. 6, pp. 875–884, 2017.

[2] jethro hazelhurst, "Detailed wiring infographic." https://docs.px4.io/en/assembly/quick_start_pixhawk.html, 2015.

[3] G. K. T. Ramirez, "Desarrollo de sistemas con la metodologiıa de sistemas de ingenierıa." internal document not published, CIDESI-CONACYT, Mexico, 2018.

[4] MicroPilot, "Triple redundant uav autopilot." https://www.micropilot.com/products-mp21283x.htm/, 2013.

[5] A. Bio-Cellular Design Aeronautics, "Redundant-autopilot." http://bda-africa.com/redundant-autopilot//, 2014.

[6] I. Company, "Triple redundant autopilot." https://www.innoflight.com/product-overview/cerberus-9248, 2014.

[7] Gizmotec, "Autopilot-manager." https://www.gizmotec.eu/product/ap-m/, 2015.

[8] A. v. a. Altigator, "Redundant okto ultra power board." https://drones.altigator.com/redundant-okto-ultra-power-board-blctrl-v30-for-8-motors-p-41747.html, 2013.

[9] D.-J. Innovations, "Redundant a3 pro." https://store.dji.com/product/a3-pro, 2015.

[10] E. UAV, "Multicopter drone." https://www.euroavionics.com/?q=multicopter-drone, 2014.

[11] A. technologies, "Asctec trinity: Advantages of the new drone." http://www.asctec.de/en/asctec-trinity/#pane-0-0, 2015.

[12] D.-J. Innovations, "Matrice 600pro." `https://www.dji.com/es/matrice600-pro`, 2014.

[13] M. Hassanalian and A. Abdelkefi, "Classifications, applications, and design challenges of drones: A review," *Progress in Aerospace Sciences*, vol. 91, no. November 2016, pp. 99–131, 2017.

[14] B. Rao, A. G. Gopi, and R. Maione, "The societal impact of commercial drones," *Technology in Society*, vol. 45, pp. 83–90, 2016.

[15] R. Luppicini and A. So, "A technoethical review of commercial drone use in the context of governance, ethics, and privacy," *Technology in Society*, vol. 46, pp. 109–119, 2016.

[16] R. Clarke, "The regulation of civilian drones' impacts on behavioural privacy," *Computer Law and Security Review*, vol. 30, no. 3, pp. 286–305, 2014.

[17] W. Yoo, E. Yu, and J. Jung, "Drone delivery: Factors affecting the public's attitude and intention to adopt," *Telematics and Informatics*, vol. 35, no. 6, pp. 1687–1700, 2018.

[18] R. Clarke, "Understanding the drone epidemic," *Computer Law and Security Review*, vol. 30, no. 3, pp. 230–246, 2014.

[19] E. Petritoli, F. Leccese, and L. Ciani, "Reliability and maintenance analysis of unmanned aerial vehicles," *Sensors (Switzerland)*, vol. 18, no. 9, pp. 1–16, 2018.

[20] I. Y. Choi, J. H. Um, J. S. Lee, and H. H. Choi, "The influence of track irregularities on the running behavior of high-speed trains," *Proceedings of the Institution of Mechanical Engineers, Part F: Journal of Rail and Rapid Transit*, vol. 227, no. 1, pp. 94–102, 2013.

[21] N. Fuqua, *Reliability Engineering for Electronic Design*. Electrical and Computer Engineering, Taylor & Francis, 1987.

[22] I. Gertsbakh and I. Gertsbakh, *Reliability Theory: With Applications to Preventive Maintenance*. Engineering online library, Springer, 2000.

[23] N. instruments, "Redundant system basic concepts." `http://www.ni.com/white-paper/6874/en/`, 2008.

[24] J. Balák and P. Ždánsky, “Modelling of Transition of System with Standby Redundancy into Failed State,” *Procedia Engineering*, vol. 192, pp. 10–15, 2017.

[25] Y. Liu, Y. Shi, X. Bai, and P. Zhan, “Reliability estimation of a N-M-cold-standby redundancy system in a multicomponent stress–strength model with generalized half-logistic distribution,” *Physica A: Statistical Mechanics and its Applications*, vol. 490, pp. 231–249, 2018.

[26] W. Wang, Z. Wu, J. Xiong, and Y. Xu, “Redundancy optimization of cold-standby systems under periodic inspection and maintenance,” *Reliability Engineering and System Safety*, vol. 180, no. August, pp. 394–402, 2018.

[27] A. H. El-maleh and F. Chikh, “Microelectronics Reliability A generalized modular redundancy scheme for enhancing fault tolerance of combinational circuits,” *Microelectronics Reliability*, vol. 54, no. 1, pp. 316–326, 2014.

[28] K. A. Hoque, O. Ait Mohamed, and Y. Savaria, “Dependability modeling and optimization of triple modular redundancy partitioning for SRAM-based FPGAs,” *Reliability Engineering and System Safety*, vol. 182, no. October 2018, pp. 107–119, 2019.

[29] D. Siemaszko and S. Pittet, “Impact of modularity and redundancy in optimising the reliability of power systems that include a large number of power converters,” *Microelectronics Reliability*, vol. 51, no. 9-11, pp. 1484–1488, 2011.

[30] S. U. S. E. L. D. S. Laboratory and J. Wakerly, *Reliability of microcomputer systems using triple modular redundancy.* Defense Technical Information Center, 1975.

[31] S. C. Anjankar, M. T. Kolte, A. Pund, P. Kolte, A. Kumar, P. Mankar, and K. Ambhore, “FPGA Based Multiple Fault Tolerant and Recoverable Technique Using Triple Modular Redundancy (FRTMR),” *Procedia Computer Science*, vol. 79, pp. 827–834, 2016.

[32] T. Nakagawa, *Maintenance Theory of Reliability.* Sieger Köder Geschenkhefte, Springer, 2005.

[33] R. Clarke, “Appropriate regulatory responses to the drone epidemic,” *Computer Law and Security Review*, vol. 32, no. 1, pp. 152–155, 2016.

[34] R. Clarke and L. Bennett Moses, “The regulation of civilian drones’ impacts on public safety,” *Computer Law and Security Review*, vol. 30, no. 3, pp. 263–285, 2014.

[35] F. Staff, "¿tienes un dron? volarlo sin licencia te costará más de 400,000 pesos.." https://www.forbes.com.mx/tienes-un-dron-volarlo-sin-licencia-te-costara-mas-de-400-mil-pesos/, 2018.

[36] P. Limbourg, *Dependability Modelling under Uncertainty: An Imprecise Probabilistic Approach*. Studies in Computational Intelligence, Springer Berlin Heidelberg, 2008.

Appendices

Appendix A

Chronogram of activities

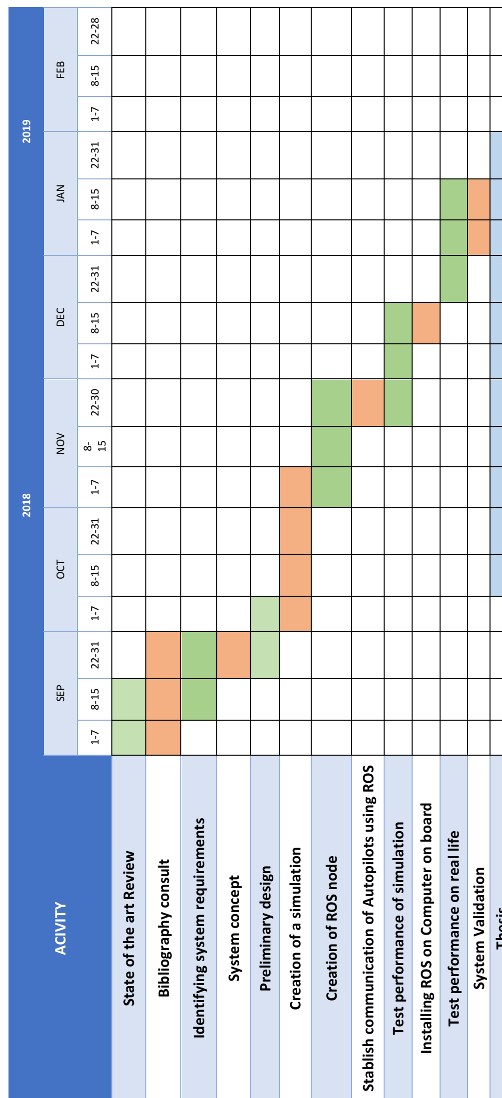

Figure A.1: Chronogram of activities

Appendix B

Installing the development tools

PX4 code can be developed on Linux or Mac OS. It is recommend to use Ubuntu Linux LTS edition as it enables building all PX4 targets, and using most simulators and ROS (see Table B.1).

Table B.1: OS for development

Target	Ubuntu	MAC	Windows
NuttX based hardware: Pixhawk Series, Crazyflie, Intel® Aero Ready to Fly Drone	x	x	x
Qualcomm Snapdragon Flight hardware	x		
Linux-based hardware: Raspberry Pi 2/3, Parrot Bebop	x		
Simulation: jMAVSim SITL	x	x	x
Simulation: Gazebo SITL	x	x	
Simulation: ROS with Gazebo	x		

Linux allows you to build for all PX4 targets (NuttX based hardware, Qualcomm Snapdragon Flight hardware, Linux-based hardware, Simulation, ROS). The following instructions explain how to set up a development environment on Ubuntu LTS using convenience bash scripts.

The scripts `ubuntu_sim_ros_gazebo.sh` install the Qt Creator IDE, Ninja Build System, Common Dependencies, FastRTPS, MAVROS, MAVLink, and also download the PX4 source to your computer (`/src/Firmware`).

B.1 Installing the development toolchain

- Download the script `ubuntu_sim_ros_gazebo.sh` from the PX4 Development page.
- In a new shell enter command: `sudo usermod -a -G dialout $USER`. Logout and login again (the change is only made after a new login).
- Execute de Script.
  It's necessary to grant the script with permission before execute it. Use command chmod +x in a shell to do so. The script will take several minutes to install all the tools. After the installation is complete the ROS folder `/catkin_ws` and `/src/Firmware` will added to the home directory (see Fig. B.1).

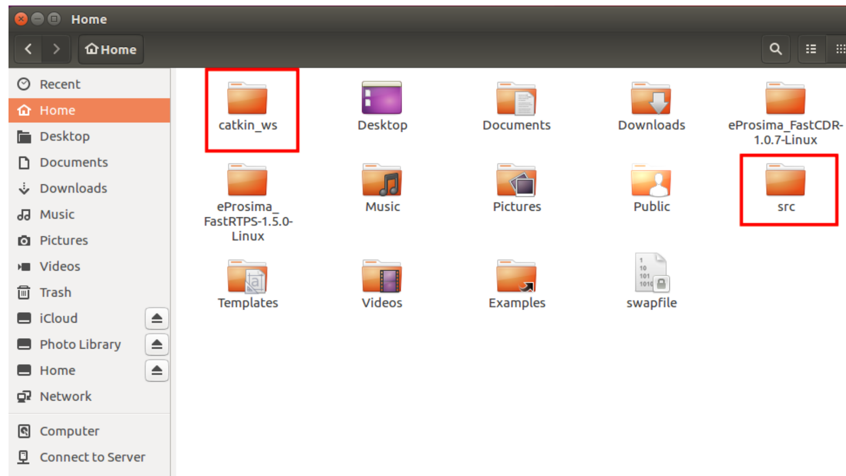

Figure B.1: Folders /catkin_ws and /src/Firmware created in the Home directory.

- Copy and paste the `/src/Firmware` into folder `/catkin_ws/src`. The folder `/catkin_ws/src` should look like Fig. B.2. The Folder `/src/Firmware` holds the PX4 firmware used for the simulation and the one running on real hardware. The folder /catkin_ws/src is the common work space used for ROS.
- Compile PX4 Firmware before launching any simulation.In a shell enter:

```
cd  /catkin_ws/src/Firmware
make px4_sitl gazebo
```

- Source your environment. Otherwise ROS got recognize the variables used to launch PX4 simulated stack. In a shell enter:

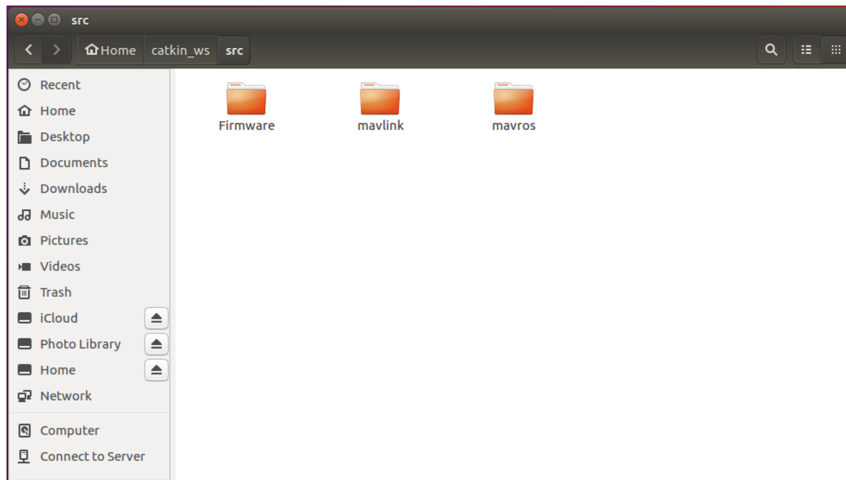

Figure B.2: Folder /catkin_ws/src

```
source Tools/setup_gazebo.bash $(pwd) $(pwd)/build/px4_sitl_default
export ROS_PACKAGE_PATH=$ROS_PACKAGE_PATH:$(pwd):$(pwd)/Tools/sitl_gazebo
```

- Now the simulation is ready to be launch. PX4 has already refined configuration examples for a few simulation. In order to star any simulation use the common ROS command `roslauch`. e.g (see Fig. B.3):
  `roslaunch px4 mavros_posix_sitl.launch`

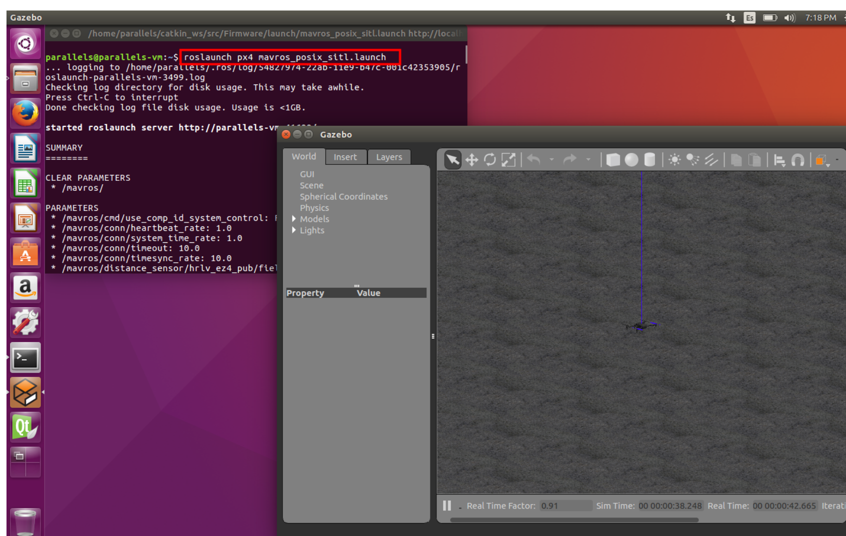

Figure B.3: Simulation launched

B.2 Extra information

The launch files are the ones in charge of executing a defined SIL configuration. They can be found in `/catkin_ws/src/Firmware/launch`. There next files are allocated:

- mavros_posix_sitl.launch
  Configures Gazebo, spawns one drone, one px4 autopilots and a connection with MAVROS.

- multi_uav_mavros_sitl.launch
  Configures Gazebo, spawns two drones, connected to two different px4 autopilots and allows connection with MAVROS.

- posix_sitl.launch
  Configures Gazebo, spawns one drone connected to one autopilot.

- px4.launch
  Base launch file for every single one PX4 autopilots running in a simulation.

- single_vehicle_spawn.launch
  Base launch file for spawning drones withing Gazebo.