TÉCNICO LISBOA

Redundant autopilot system based on COTS open source solution

Pedro Nuno Ferreira Afonso

Thesis to obtain the Master of Science Degree in

Aerospace Engineering

Supervisors: Prof. Alexandra Bento Moutinho
Eng. Renato Santos Machado

Examination Committee

Chairperson: Prof. Paulo Ramalho Oliveira
Supervisor: Prof. Alexandra Bento Moutinho
Member of the Committee: Prof. Agostinho Alves da Fonseca

December 2021

To my grandmother who I am grateful for everything and will stay alive inside the memories I shared with her...

Declaration

I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Acknowledgments

This is the final chapter of my student's life after a long journey at Instituto Superior Técnico. Not all the goals were achieved with success on the first tries. So one of the most valuable teachings I take with me is that with a lot of dedication and perseverance what seems too far away or sometimes impossible, can become a great success.

To my faculty supervisor, Prof. Alexandra Moutinho from Instituto Superior Técnico, thanks for the help and guidance since the start of the thesis and for her patience when I needed to study and learn new tools for the correct implementation and validation.

To my supervisor Eng. Renato Machado, representing CEiiA (Centre of Engineering and Product Development), who accompanied me closer at the beginning of the thesis, supporting me with his experience and precious advises on the research approaching and structuring the main steps.

I would also like to thank the Ceiia academy for accepting me on this partnership, all their people receiving me so well always with sympathy, making me to feel I belonged there and their availability to provide me all the technical resources to work on this investigation. A special mention to the Eng. João Fortuna who helped me with technical support on the programming approach.

To my parents, who have been with me through all the course, supported me at the most difficult moments and the main reason to be writing this thesis. All the possible words are not enough to thank them.

To my girlfriend, Eugénia, who have always been ready to support me in my life where she could. Thank you, specially for your understanding as well as patience on this last year with so many ups and downs.

To my friends at IST, João Filipe, João Manito, Daniel Schiffart and Francisco Castro who shared with me great moments during the course and were always available to help me with the technical and personal support, including at this last step. Thanks for your friendship.

Note regarding experimental data and source code access

In the spirit of free and unrestricted access to academic research and experimental data, all the datasets used in this thesis will be published free of charge in a Github repository, `https://github.com/stalone89/thesis-.git`, under the Creative Commons Zero v1.0 Universal license.

The source code used for this thesis will also be available in a Github repository, `https://github.com/stalone89/thesis-.git`, under the GNU General Public License v3.0.

If, for some reason, the data or source code are no longer available contact me through the e-mail pedro.afonso_trf@hotmail.com to request a copy.

Resumo

O desenvolvimento das aeronaves não tripuladas (UAV) aumentou nos anos mais recentes, devido às vastas aplicações, que estas já desempenham na sociedade. Naturalmente, o software e o hardware dos pilotos automáticos usados pelos veículos aéreos não tripulados também acompanhou a evolução, sendo que uma das preocupações destes sistemas é mitigar ou eliminar erros que possam ter consequências negativas e inesperadas em prejuízo dos utilizadores ou de terceiros. É sobre o desenvolvimento de um sistema de detecção e tolerante a falhas, aplicado no piloto automático de UAV que a presente tese incide.

A partir do estudo dos conceitos de fialibilidade e redundância justificou-se que três unidades de pilotos automáticos são suficientes para integrar o sistema.

Após análise de vários pilotos automáticos disponíveis comercialmente de código aberto, o PX4 foi o escolhido.

A biblioteca de controlo de estimação é o módulo mais complexo do PX4 firmware e por padrão, utiliza sete instâncias independentes do filtro de Kalman estendido ao mesmo tempo, oferecendo redundância ao nível do estimador e capacidade de detecção de falhas graves e leves. Cada estimador é avaliado através das suas inovações e variância das mesmas.

O algoritmo de decisão foi projectado num anel externo baseado nos resultados finais destes mecanismos redundantes existentes no firmware. A selecção do piloto automático é calculada de forma descentralizada, dentro de cada um, e em permanente comunicação com os outros dois pilotos automáticos.

O sistema de tripla redundância foi validado no software-in-the-loop com o simulador Gazebo.

Palavras-chave: Veículo aéreo não tripulado, PX4 Firmware, Filtro de Kalman estendido, Detecção de falhas, Sistema tolerante a falhas.

Abstract

Recently the unmanned air vehicle (UAV) development has been increasing due to the large number of applications these already play at the society. Naturally, the autopilots software and hardware caught up the UAV development. The mitigation or elimination of these system failures, which can imply unexpected behaviours with losses to the users and to others, has become a concern. Developing a fault detection and tolerant system to the autopilot is the main objective of this thesis.

From the reliability and redundancy concepts study, it was concluded that three autopilot units are enough to integrate the system.

After analysing some commercial-of-the-shelf open source autopilots, the PX4 firmware was chosen.

The estimation control library is the most complex module from the firmware. By default it uses seven independent extended Kalman filter instances at the same time, offering redundancy to the estimator level and capability for soft and hard fault detection. Each estimator health is evaluated through its innovations and innovations variance.

The decision algorithm was computed on the external ring based on the final results from all these redundant mechanisms existing at PX4 firmware. The selection is computed in such a decentralized way, inside each autopilot permanently on communication with the other two.

The triple redundancy system was validated running the software-in-the-loop with the gazebo simulator.

Keywords: Unmanned air vehicle, PX4 firmware, Extended Kalman filter, Fault detection, Fault tolerant system

Contents

List of Tables

List of Figures

Acronyms

ACARS Aircraft Communications Addressing and Reporting System

ADC Analog Digital Converter

AFRS Automatic Fault Reporting System

AHP Analytical Hierarchical Process

API Application Programming Interface

ARINC Aeronautical Radio, Incorporated

ARM Advanced Reduced Instruction Set Computer Machines

ASRS Aviation Safety Reporting System

ATTMS Automatic Takeoff Thrust Management System

BITE Built-In-Test Equipment

COTS Commercial-Off-The-Shelf

CPU Control Processing Unit

DDS Data Distribution Service

EEPROM Electrically Erasable Programmable Read-Only Memory

EKF Extended Kalman Filter

FD Fault Detection

FIFO First in, First Out

GCS Ground Control Station

GPCLK General Purpose Clock

GPIO General-Purpose Input/Output

GPS Global Positioning System

HITL Hardware-In-The-Loop

HUMS Health and Usage Monitoring System

I2C Inter Integrated Circuit

IP Internet Protocol

IR Interrupt Request

MAVLink Micro Air Vehicle Message Marshalling Library

MTBF Mean Time Before Fail

MMTF Mean Time To Fail

NASA National Aeronautics and Space Administration

NTSB National Transportation Safety Board

PCB Printed Circuit Board

PCM Pulse Code Modulation

PWM Pulse Width Modulation

QGC QgroundControl

RC Radio Control

RAM Random-Access Memory

ROS Robot Operating System

RTPS Real Time Publish and Subscribe

SITA *Sociétié Internationale de Télécommunications Aéronautiques*

SITL Software-In-The-Loop

SPI Serial Peripheral Interface

TCP Transfer Control Protocol

UART Universal Asynchronous Receiver/Transmitter

UAS Unmanned Aerial System

UAV Unmanned Aerial Vehicle

UDP User Datagram Protocol

USART Universal Synchronous/Asynchronous Receiver/Transmitter

USB Universal Serial Bus

VHF Very High Frequency

Nomenclature

Roman symbols

x, y, z	Cartesian components
N,E,D	Inertial navigation frame components

Superscripts

T	Transpose

Chapter 1

Introduction

This chapter provides the main goals of this thesis complemented with a research about existent systems on the market which share the same purposes of this topic. Also the motivation for the project followed by a basic outline of the structure of this thesis can be found here.

1.1 Motivation

”Automation is ubiquitous in many modern work settings, but perhaps most so in aviation. Automation in aviation, as in other domains, has increased demands on the pilot to monitor systems for possible failures. As research on vigilance has shown, this is a role for which humans are poorly suited.” [1].

To support this transcription, two examples from accidents related with automation overreliance by humans are given below. In 1979, the crew from Eastern Flight 401 failed to detect the autopilot disengaging and did not monitor altitude because they were engaged in a possible problem with the landing gear and the aircraft crashed into the Florida Everglades.

The crew from China Airways Flight 006 preoccupied with an engine problem, did not notice the autopilot gradually loosing control of the plane in 1985.

On both mentioned incidents, the crew was at the end of a long shift, was highly qualified, should have detected the occurrence, were preoccupied with another task and may have overrelied on automated systems. It can be concluded that under certain conditions, pilot overreliance on automation can make detecting failures problematic since pilots may ignore other sources of information.

The problems in monitoring automated systems are further evident when analysing pilots reports of incidents.

NASA’s Aviation Safety Reporting System (ASRS) database was examined by Mosier et al. (1994). The conclusions are: 77% of the incidents in which overreliance on automation was suspected, involved a probable vigilance failure as well as the vast majority of them occurred during cruise, when the pilot primary role was to monitor and supervise the automation.

By studying anonymous responses to questionnaires about automation-related incidents from German aviators, Gerbert and Kemmler (1986) reported the largest contributor to human error to be failures

of vigilance.

So it is assumed that automatic pilots fail as well as the people responsible for monitoring those fails. Despite of being older, these reports provide good indications to the problem. Based on the last conclusion, the research started by aircraft systems with the functionalities of acquiring and analysing the flight data looking for possible faults.

The Unmanned aircraft systems (UAS) belong to the aeronautics field as well and they have been playing an important role through the military sector. UAS include an Unmanned aerial vehicle (UAV), a ground-based controller and a system of communications between the two. The UAV came from the airplanes following an historical perspective. So the framework about automation and fault reporting systems used on airplanes plays an important role because the knowledge was adapted to develop the most recent fault detection systems used by UAV.

Large (e.g., Global Hawk, Predator) and smaller unmanned aircraft (e.g., Wasp, Nighthawk) are military examples of this technology development around the world. They are used in numerous applications such as surveillance, communication relays or reconnaissance.

Nowadays, the challenge became the UAV civil and commercial applications. Although this technology greater potential has been recognised, the society is still far from achieving all its utilities.

Some examples from UAV applications already used:

- Fire Detection - powerful to cover large areas using their mobility on the forests combining with cameras and sensors to detect fire [2];
- Rescue Operations - facilities to access inhospitable places for humans [3];
- Farming - useful for irrigation managing and vegetation monitoring [4];
- Sport coverage - guarantees a lot of new stable positions and angles for cameras [5];
- Law enforcement - using cameras as payloads allows permanent vigilance from many places or even autonomous inspection [6];
- Transportation - it allows to choose the direct routes through the atmosphere while carrying payloads [7].

It is easy to predict that in the near future the civil applications will increase because of this technology improvements.

Since these aircraft are unmanned, they are strongly related with the automation development due to the airborne software/hardware necessary to be operated.

When the automation is introduced somewhere, the faults theme will follow it, becoming crucial if the designers want the system working properly, detecting and reducing the corresponding faults as well as its amplitude to reasonable numbers.

1.2 State Of The Art

Various type of systems related with faults were studied. Mechanical faults were considered beyond the hardware and software faults.

1.2.1 Market Overview - Fault Diagnose Systems

Health and Usage Monitoring System

”Health and Usage Monitoring System provides diagnosis information required for optimum performance.”[8]

Health and performance of mission-critical components are measured by sensors and monitored by HUMS embedded diagnostic software alerting for the maintenance need of those components.

Some tasks provided by HUMS:

- Continuous vibration monitoring of drive-train;
- Performs rotor, track and balance;
- Provides actionable information for informed maintenance decisions;
- Pinpoints mechanical faults before they become catastrophic failures.

Despite of being a useful system to avoid the most dangerous scenarios, it was designed to give information to the maintenance team preventing accidents in the near future, detecting previously which are the most probable components failure.

It is not the best system to solve faults in real time, although it plays an important role on failures prevention. The system is focused in mechanical failures, it does not mention software/automatic pilot anywhere. It can be concluded from its functionality that it could be useful to this project just in case of reading some unexpected values on the automatic pilot or sensors output and comparing with the data given by HUMS to help understanding if the fault is related with a mechanical failure or if it is related with software/hardware. Since the purpose is a faster real time detection fault system, the communication and the data process with HUMS would slow in order to get just one more functionality rarely used when compared with automatic pilot control speed.

Automated Flight Data Management System

During the flight, this system [9] records the desired set of flight parameters such as airspeed, heading, fuel consumption, altitude, engine rpm, etc... It manages and generates reports based on that flight data, accessing signals obtained by the sensors installed on the aircraft and transmitted on an airborne databus.

After processing the signals, they are stored on a portable and self-protected memory device. The signals processing consists on sampling, filtering, decoding, encrypting, and subjecting to an adaptive compression.

The compression ratio varies according to the memory capacity of the device, since it decreases during the flight, to allow all the data from the entire flight to be recorded.

The inversion process to treat the flight data is applied after the flight ends and it can be analysed and it can be used to evaluate the pilot performance and monitor the operation of the aircraft over the course of the entire flight by authorised personnel.

Statistical methods or artificial intelligence based algorithms or others various data analysis techniques may be used to examine the flight data looking for any problem with either pilot or aircraft perfor-

mance.

This fault diagnosis system does not work in real time. The data is recorded and sent to the ground station for a posterior analysis.

1.2.2 Market Overview - Fault Detection Systems

Automatic Fault Reporting System

The automatic fault reporting system (AFRS) used in combination with airplanes systems reports airplane fault conditions prior to landing to the ground maintenance personnel.

The AFRS advantages consist in automatic comparing/monitoring of various aircraft data parameters during the flight supplying fault outputs to the aircraft communications addressing and reporting system (ACARS) for transmission to ground-based maintenance operations which relieves the flight crew from the responsibility of isolating and reporting Built-In-Test Equipment (BITE) detectable fault conditions. On the ground, this information can be used to assist inventory control, airplane scheduling or periodic maintenance scheduling.

Automatic fault reporting is the primary functionality of this system and it is resumed here:

- Get the most likely cause for the fault through monitoring system outputs which can be either analog or digital. Assign a fault code correspondent to that cause.
- ”On command, send ’data present’ discrete to presently installed Aeronautical Radio Inc (ARINC).” [10]
- ”On command, send fault code to ACARS which transmits data via very high frequency (VHF) communication to presently installed ARINC/Société Internationale de Télécommunications Aéronautiques (SITA) network on ground which in turn routes fault code via land lines to applicable airline.” [10]

Despite of detecting the fault ”alone” and decreasing the human factor by the employment of presently installed systems for the fault detection, AFRS does not correct the fault.

From this example it is noticed the importance of automation in the effectiveness and speed on fault detection by the automatic reporting of presently on-airplane fault information to the individual airline.

1.2.3 Market Overview - Fault Tolerant Systems

Automatic Takeoff Thrust Management System

The main point from an automatic takeoff thrust management system (ATTMS) [11] is to reduce or minimize takeoff noise in a limited takeoff field length. However its second functionality is the most interesting to the fault tolerance theme, so it will be focused here.

ATTMS comprises status sensor or set of sensors capable of detecting establishment of takeoff climb conditions and a controller coupled to them. Engine failure detectors and thrust management modules respectively coupled to at least two engines (the first ones capable of detecting engine failures and the

second ones capable of controlling the thrust of the engines) belong to this system as well as the thrust controller which restores thrust to the initial or higher schedule after detecting an engine failure.

In response to signals from the failure detectors, the computer can automatically answer to complete or partial engine failure using various techniques to control operating engine power levels. Additional sensors for redundancy and self-checking can be used to improve reliability by avoiding false positive and false negative indications.

The computer can address partial or intermittent engine failure that results in power loss by increasing the power in remaining operating engines.

Mechanical failures are detected by reading the sensor values. Designing a new controller to correct mechanical failures is out of scope from this thesis, so these will not be addressed here.

ATTMS gave some important ideas about redundancy on sensors and a conceptual schematic to apply on the detection fault architecture showing the crucial communications between sensors, computer and automatic pilot to have a response in real time when a failure occurs.

Built-In-Test Equipment and Redundancy Systems

By definition, BITE means ”any device which is part of an equipment or system and is used for the express purpose of testing the equipment or system. BITE is an identifiable unit of the equipment or system.” [12]

The built-in-test can be used in a software approach offering fault isolation, automatically complete system checks or memory capability to identify a component deterioration earlier preventing an entire system failure. Also, the software can always be adapted to the system changes if needed.

So the software approach should guarantee system data isolation from the test data because it is not desired the interfacing hardware receives system outputs as commands while testing the system. To get a greater advantage from BITE, the existing data networks should be used to test the interface circuitry too and it should be defined common monitoring points whenever possible to isolate faults while providing tests to more functional areas. Tolerance should be provided in order to monitor those functional areas while the input stimuli must not be very different from the normal accepted data in order to not create malfunctions at interfaces. [13]

In order to achieve an higher hardware fault detection capability the majority voting monitoring technique was combined with the BIT techniques. It can be assumed the probability of occurring faults simultaneously for three independent systems is very small. If the three systems are identical (software and hardware), tight synchronized and there are no faults, their behaviour and outputs should be the same when stimulated with the same input data.

The redundancy drops the need of bigger threshold values for fault judgements which could decrease the fault detect detection beyond of adding fault tolerance when switching from the master failing system to other one available.

The triple redundant architecture or triplex is very useful because the third unit can be used as a tiebreaker criteria when two systems give different outputs based on majority voting monitoring and showing which is the failed one.

As it can be seen at the figure 1.1, the monitor uses built-in-tests to get reasonable values which will be compared with the output of the real time system. Based on the comparison, the decision of using a determined lane is taken connecting or disconnecting the wire.

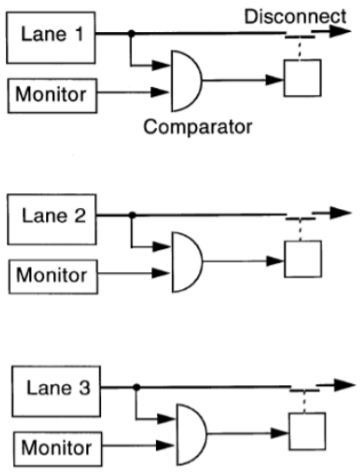


Figure 1.1: Triplex redundancy based on built-in-tests [14]

Redundant Autopilot system from Micropilot

The system functionality is based on the replacement of one failed system by another similar software and hardware system available to take over the vehicle control. In this case, there are three autopilots prioritized offering double redundancy. If the first autopilot fails, which flies the UAV, the second autopilot takes over. It is useful to detect hardware faults.

The sensor data cannot simply be compared, because of the noise or the phase shift. It could lead to a bad judgement about the sensors health.

Other concern from this triple redundancy system, MP2128^{3X}, is to assure smooth transitions between control jumps when a failure is detected on the flying autopilot. In order to avoid sudden control inputs, those jumps do not take place on flight critical phases like the takeoff.

The system has independent power supplies connected. Each one of them is connected to a single autopilot to ensure reliability on this component.

It uses a pulse based voting system instead of logic levels to choose which autopilot operates the UAV. So it decreases the chances of fail signal malfunctions.

The MP2128^{3X} consists on a decentralized system since ”the three autopilots continuous watch state information from the other two autopilots.” [15] Some of the main concepts from this example represented in the figure 1.2 will be applied on this thesis such as the decentralization and simple voting process to avoid complications at the selection.

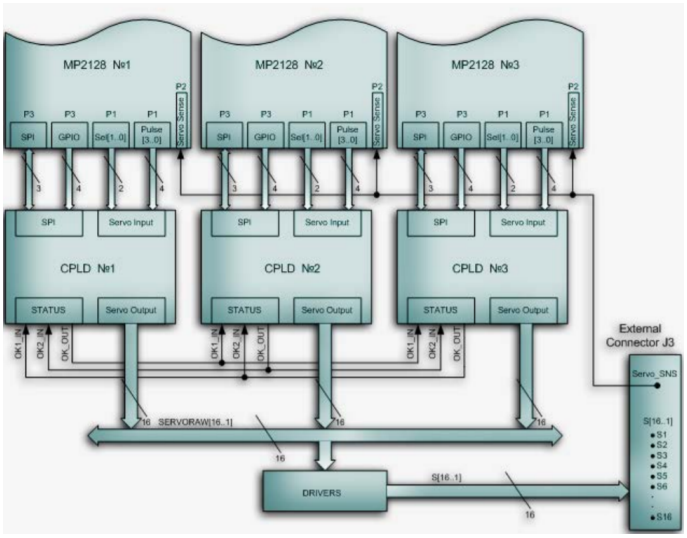


Figure 1.2: Servo signals in the MP2128^{3X} redundancy board [15]

Sensor sharing will be incorporated at MP2128^{3X} in the future to assure an autopilot to keep flying in case of loosing one of its sensors. It will allow access data from another autopilot sensor.

1.2.4 Systems Analysis Resume

The table 1.1, was filled with the features, from the mentioned systems in the section 1.2.

	HUMS	AFDMS	AFRS	ATTMS	B w/ R [1]	MSM [12]
Fault Diagnose	X	X				
Fault Detection	X	X	X	X	X	X
Fault Tolerant				X	X	X
Real Time FD [2]			X	X	X	X
Mechanical Faults	X	X	X	X		
Hardware Faults		X	X		X	X
Software Faults					X	X
Inputs	MMT[3]	FPR[4]	FPR[4]	SData[5]	DO[6]	FPR[4]
Outputs	MF[7]	PAP[8]	CF[9]	EFT[10]	Faults	AS[11]

Table 1.1: State of the Art systems characterization.

1.3 Objectives

The main purpose of this thesis is to improve an automatic pilot, decreasing its number of faults without loosing performance, always keeping in mind the time it would be necessary to create/design a new automatic pilot. There are already tested autopilots on the market with responsiveness to almost every type of aircraft like the Micropilot model.

So, instead of doing a new controller without guaranties it would be better than others available to the client, the focus was directed to build a fault detection, isolation and tolerant system. The reliability will be increased using redundancy. The fault tolerant system designed in this thesis will be applied to an UAV. Commercial-Off-The-Shelf (COTS) open source autopilots for UAV(s) are used as part of the system. This project was made in collaboration with the Centre of Engineering and Product Development, CEiiA. The institution has been working in partnership with the Portuguese Air Force doing various missions with the UAS-30, figure 1.3, using different payloads. It is intended to validate and apply the designed system to the UAS-30.

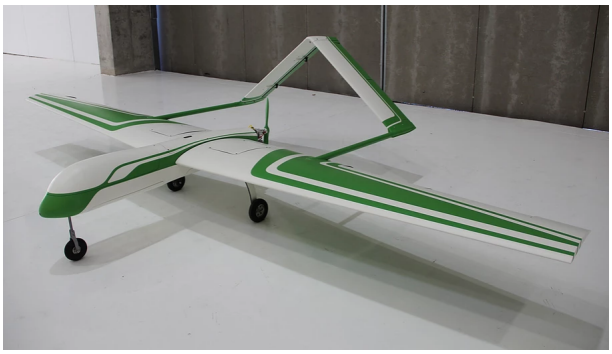


Figure 1.3: UAS30 from Ceiia [16]

Goals resume:

- Decrease the autopilot failure probability or increase its reliability
- Design a fault tolerant system with redundant autopilots integration
- Use COTS open source autopilots
- Demonstration of proof of concept of the proposed solution

[1]BITE with Redundancy
[2]Real time fault detection
[3]Mechanical monitoring Tests
[4]Flight parameters records
[5]Sensors data
[6]Devices output
[7]Mechanical faults
[8]Pilot and aircraft performance
[9]Codified faults
[10]Engine failures ans thrust
[11]Autopilot selection
[12]Micropilot System Model MP2128^{3X}

1.4 Thesis Outline

Chapter 1 refers to the main objectives from this thesis, the motivation behind it and to the state of the art.

In Chapter 2, UAS and Fault Detection systems are described focusing on the autopilot. The reliability and redundancy concepts are introduced.

In Chapter 3 the decision about the autopilot firmware and hardware used in this project is justified applying the AHP method to the autopilots available on the market.

In Chapter 4 the approach strategies to build the system are discussed and related with the PX4 software architecture to get the best solution for a fault tolerant system.

Following that, in Chapter 5 the Fault Tolerant system implementation and correspondent simulation are done to obtain the results necessary to validate the system in the Software in The Loop mode.

Finally, in Chapter 6 presents the conclusions and the following steps to validate the entire system in the UAS-30 during a real flight.

Chapter 2

Background

UAS are divided in two subsystems, the ground station and the airborne, figure 2.1. The communications are present in the two subsystem.

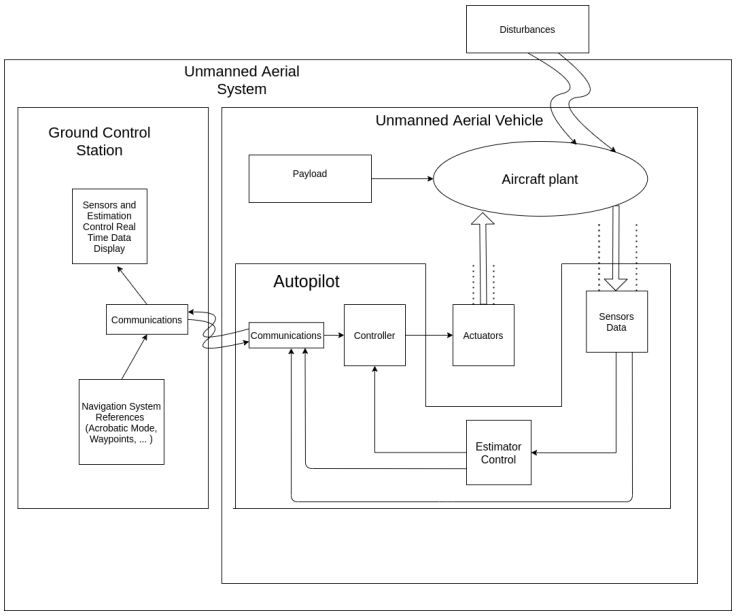
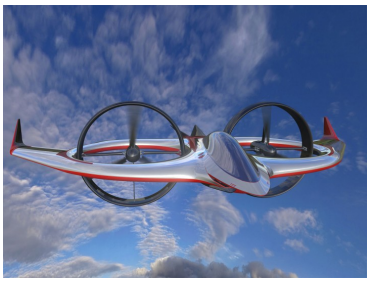


Figure 2.1: UAS

Unmanned Aerial Vehicle is a vehicle which moves trough the atmosphere without on board human crew. The technology development allowed the usage of the UAV outside from the military's domain, dropping its costs to the civilians. There are different UAV and they can be grouped by their typology: fixed wing (figure 2.2(a)), multi-rotor (figure 2.2(b)), hybrid (figure 2.2(c)) and lighter-than-air.

(a) UAS30 from Ceiia [16]

(b) Multi-rotor UAV [17]



(c) Hybrid UAV [18]

Figure 2.2: UAV typology

2.1 Unmanned Aerial System Components

2.1.1 Ground Station

An autonomous operation is characterized by the ability of keeping the flight and executing tasks as takeoff or land without any human intervention. The system navigation can change between a full autonomous operation, to a tele-operation where the UAV is commanded by an human operator on the ground using remote control. The second option gives a minimal degree of autonomy to the UAV.

The Ground Station is where the mission is planned and the operation control center. It is the machine's interface with the human regardless the flight mode operation with more or less autonomy. It has signal emitters and receptors antennas, video receptor and the software to analyse flight data through telemetry links and to be aware of some airborne components state. A picture from the Ground Station software, QgroundControl, used on this project to command the vehicle is shown in figure 2.3.

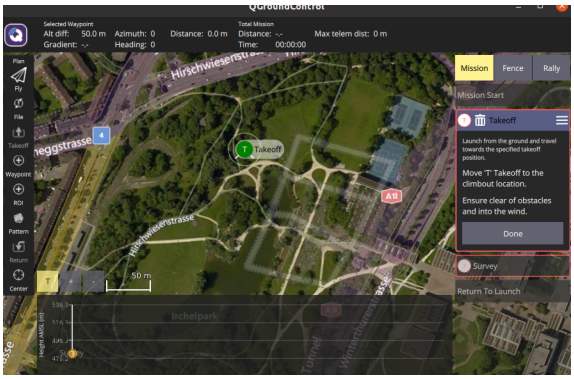


Figure 2.3: Qgroundcontrol software print screen

2.1.2 Airborne Systems

This refers to the flying equipment: wings, motors, propellers, rudders, ailerons, flight controllers, video cameras, batteries, antennas, sensors...

In figure 2.1, the two main parts of the UAS can be distinguished. In this project the autopilot will be focused, mainly the flight controller and sensors.

Usually the most important sensors for navigation have redundancy. Some examples are the accelerometers, gyroscopes, compass or pitot tubes.

Autopilot

An autopilot typical architecture is shown in figure 2.1 for a better understanding about its functionality. Other subsystems, which interact with the controller, are represented too.

The flight state estimator plays a big role on the autopilot since the states are directly used by the control unit to compute the outputs that will be sent to the actuators. Some autopilots use the Extended Kalman filter (EKF) for states estimation and it will be seen, on the next chapter 4, it is possible to evaluate the estimator itself for the fault detection purpose.

2.2 Fault Detection

2.2.1 Fault and Failure Description

The fault detection begins by analysing data communicated by the aircraft. Data can be provided by its on-board sensors, actuators as well as other internal states from the controller.

Some concepts should be clarified before starting to analyse and process the data communicated between the avionics components.

- **Fault** - ”It is a deviation (of a feature) from the acceptable, standard operational condition. (permanent, transient, ...).” [14] In figure 2.4, are demonstrated four sensor faults examples. The figure 2.5 shows actuator faults. Fault is the condition that causes the error.
- **Error** - ”Incorrect status resulting from a fault, information inaccuracy.” [14] It is the difference between actual output and the expected output. From the figure inspection 2.4, the errors are clear when it is compared the truth value (dotted line) with the measured value. Different causes (like frozen sensors or sensor drifts) could be at the error origin, so they are called faults.
- **Failure** - ”It is a permanent interruption of a system’s ability to perform a required function under operating conditions.” [14]
- **Fault Tolerant System** - ”It is a system, that is capable to perform its function properly in the presence of one or several faults.” [14]
- **Reliability** - Defines the system capability to keep running properly while being affected by faults. It is the fault tolerance measure from a system. A detailed explanation at the next section 2.2.2.
- **Redundancy** - It is the process of adding identical critical components to increase the system reliability. A more detailed explanation on the further section 2.2.3

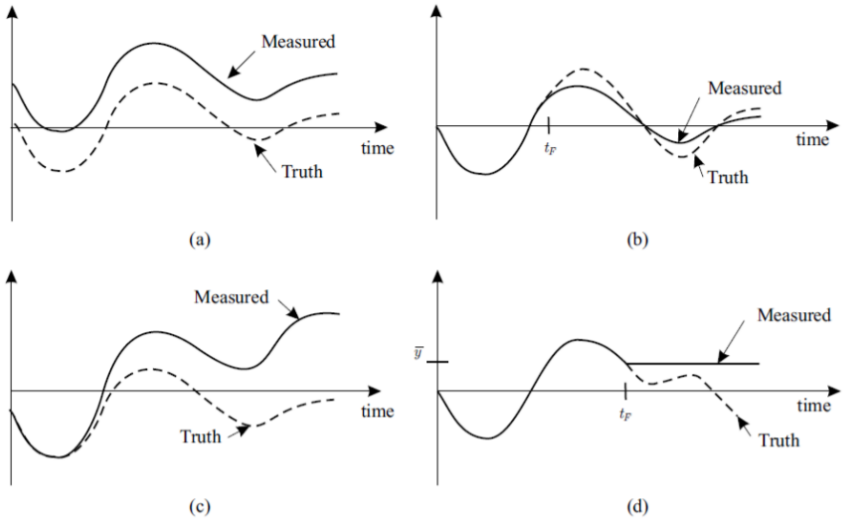


Figure 2.4: a) Sensor bias; b) Loss of accuracy or calibration error; c) Sensor drift; d) Frozen sensor; t_F represents the moment when the fault was injected; [14]

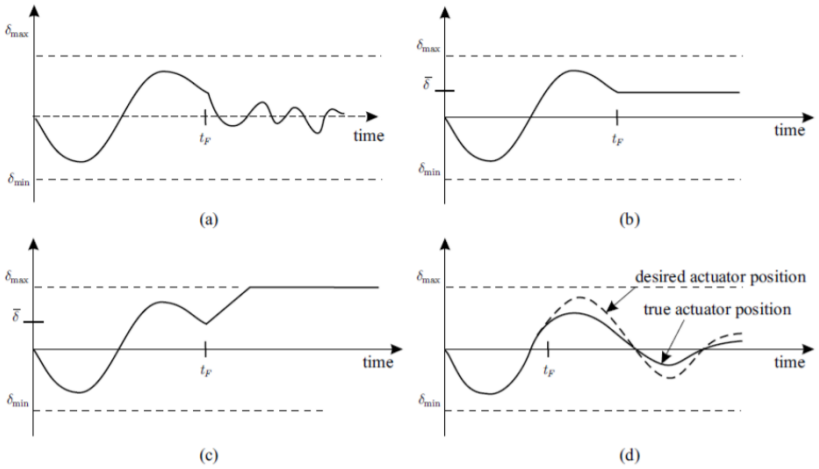


Figure 2.5: a) Floating around set point; b) Lock-in-place; c) Hard-over; d) Loss of effectiveness; t_F represents the moment when the fault was injected; [14]

2.2.2 Reliability

Reliability represents the system ability to perform its tasks without errors for a time period. It is a powerful tool to evaluate the design strategy from a fault tolerant system but other related concepts will be explained first.

The Mean time to fail, MTTF, is calculated by [14]:

$$MTTF \approx \frac{t_1 + t_2 + t_3 + ... + T_N}{N} = \hat{E}[\tau] \tag{2.1}$$

where each t_N means the time to fail or degradation from a single component under ”stress test condi-

tions", N it is the number of identical components used from a sample and $\hat{E}[\tau]$ is the empirical mean time to fail. This evaluation process is called failure characterisation.

The Mean Time Before Failure - $MTBF$ - from a system is defined by $E[\tau]$, equation 2.3. This is not an empirical variable. The reliability, $R(t)$, equation states:

$$R(t) = e^{-\int_0^t \lambda(x)dx} \tag{2.2}$$

where λ means the probability of failure which can be considered constant to simplify ($\lambda(x) = \lambda$).

In this case $E[\tau]$ can be related with Reliability, $R(t)$, through the equation:

$$MTBF = E[\tau] = \int_0^{\infty} R(x)dx \tag{2.3}$$

After introducing the $R(t)$ concept, other ways to calculate its value are defined regarding the system configuration. System reliability in a serial connection (figure 2.6):

$$R(t) = \prod_{i=1}^{n} R_i(t) = \prod_{i=1}^{n} e^{-\int_0^t \lambda_i(t)dt} \tag{2.4}$$

where $R(t)$ is the system reliability, $R_i(t)$ is the reliability from each unit, $\lambda_i(t)$ is the failure probability from each unit (which can be considered constant $\lambda_i(x) = \lambda_i$) and n is the total number of units.

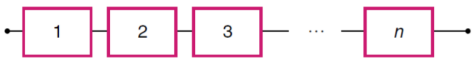


Figure 2.6: Failure in serial connection[14]

System Reliability on a parallel connection (figure 2.7):

$$R(t) = 1 - \prod_{i=1}^{n}(1 - R_i(t)) \tag{2.5}$$

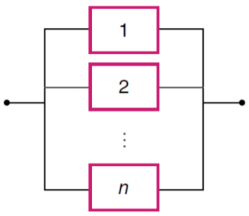


Figure 2.7: Failure in parallel connection [14]

Therefore to increase the system reliability, identical components should be added in parallel whenever possible creating redundancy.

2.2.3 Redundancy

The main purpose is to choose the best architecture option to integrate the flight controllers based on the fault detection methods and communications dependencies by duplication of the system. It will be an embedded system - computer based information processing systems that are part of a large system or equipment.

It must be a real-time system. It has time-respond constraints to complete its work/service when it is requested. There must be a predictable time interval to give the correct outputs.

First of all, a way must be figured out to evaluate if the automatic pilot is working properly. As it was noticed in the section 2.2.1, the system operation is evaluated throughout its outputs. Using the outputs from a new internal control model for comparison against the autopilot outputs is not an option (avoiding to build a new controller). So it was decided to use redundancy on the same existent model.

It takes to other questions: Which components should be redundant? Where will the monitoring point be located?

On the presence of sensor failures, wrong data is sent to the autopilot processor (the state estimator unit and after to the control unit). If the monitoring point is located at the autopilot outputs, the fault detection process becomes more difficult and lasts longer. Adding monitoring points increases the fault detection component capability. A new module was added to the system, the autopilot selection which will be connected in series with the autopilots. The autopilot selection is made based on the fault detection, so its reliability increases when the fault detection capability increases too. For the purpose of detecting the fault immediately, one monitoring point must be at the sensors level due to their multiplicity and there should be redundancy at the sensors.

Therefore, the designing strategy from this project system consists in adding more flight controllers, with a monitoring point at the sensors module, to gain redundancy on both units. [12]

An example, using serial connections between the sensors and the flight control computers in a quadruplex architecture is represented at the figure 2.8. This system uses four sets of sensors, flight control computers and actuators. Each flight controller is individually and serially connected to one sensors module.

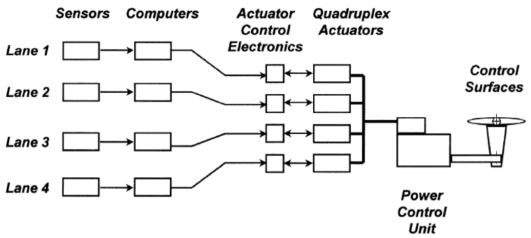


Figure 2.8: The quadruple redundant flight control system architecture.[14]

Throughout data observation and comparison from the sensor outputs, it is possible to detect a failure from one sensor unit. An hard-over failure is exemplified below on a quadruplex system, see figure 2.9.

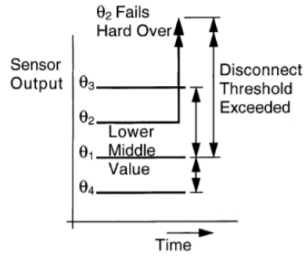


Figure 2.9: Quadruplex Hard-over failure [14]

The figure 2.9 shows an hard over failure from the second sensor θ_2. Its measurement was compared with the lower middle value from the four sensor measurements. The difference between the two sensors measurements exceeded the threshold value.

For data comparison, the system needs communication between the redundant modules. A cross communication data link can be used for data exchange between the flight controllers. For the actuators selection there must be a voting system. This example can use the flight control computers to send out the command for voting despite of the selection being made on the actuators. This system weakness resides on the serial connection between units. If either the sensor or the flight control computer fails on the same set, the entire set will fail.

The options to place the voting process and how the redundant systems communicate refers to the system architecture. Generally, there are the three types for the flight control systems:

- **Centralized architecture** - The flight control computer will be connected to all the redundant subsystems like the sensors or actuators. The wiring harness complexity increases as well as the volume, weight and the reliability requirement for the centralized system part. It has to calculate and control the entire system. Simple software design and easy maintenance beyond the need of a lower number of subsystems are the main advantages.
- **Distributed architecture** - Each subsystem do the calculations and communicates with others. The disadvantages are the increasing number of subsystems and the software design difficulty. By the other side, the dependence on a single hardware is reduced because there is not a centralized control unit. It can be demonstrated at figure 2.10 where there is no dependency from one flight control system. To be a distributed architecture the software should be decentralized too.
- **Federated architecture** - Consists in a distributed hardware design with a centralized software framework. Capability of achieving fault tolerant while reducing the design difficulty. It has a number of subsystems lower than a distributed architecture and higher than centralized architecture. Since it is a compromise between the two architectures, the point is to take main advantages from them keeping an higher reliability from the entire system.

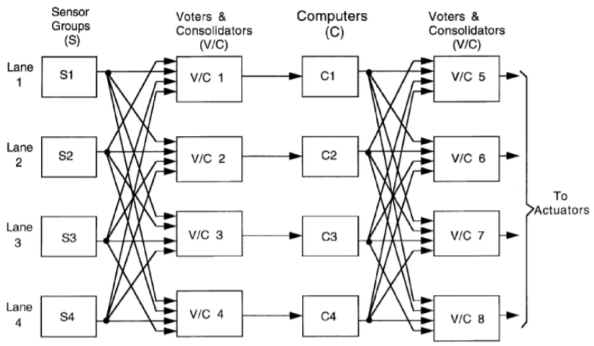


Figure 2.10: Quadruplex redundant flight control system. A possible distributed architecture with the voting mechanism system[14]

This system should have a distributed architecture ideally to achieve a greater reliability. If it is not possible to separate subsystems or the distributed system design becomes impractical because of its difficulty, the system should be federated so it does not rely on a single hardware.

It has been shown examples with four redundant sensors units and the same number of flight control pilots. However the units number should always be considered to decide the best option for the redundant system. A trade-off analysis must be performed to establish the redundancy for the electronic system. The criteria which will impact the trade-off decision to incorporate redundancy at either the system level, circuit or function level are:

- Reliability requirements;
- Testability limitations;
- Mission essential functions and criteria availability;
- Weight, size, impact on circuit functions including electromagnetic interference and cost.

From these points, it must be assured the redundancy used is the redundancy needed. In the figure 2.11, a slow over failure example occurring in the triplex architecture is represented.

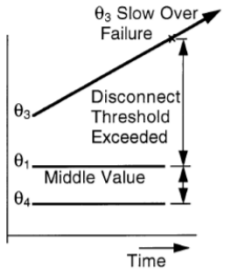


Figure 2.11: Triplex Slow-over failure[14]

Both failures, hard-over and slow-over, could be detected and isolated using the middle value read from the three outputs when compared with a threshold value imposed by the system designer. The

previous explanation is the same for the quadruplex architecture example from a sensor failure, see figure 2.9, with an exception: since the number of read values is even, the lower middle value is chosen to verify if the other sensors output exceeded the threshold to isolate it.

The autopilot failure probability is considered a small value so the chances of failing two units simultaneously are even lower. So the advantages of designing a system with four units instead of three are not enough to cover the increasing weight, size and cost.

For a majority voting system with triple modular redundancy, the system reliability is calculated by[14]:

$$R(t) = R_m \sum_{i=2}^{3} {}_3C_i R_i(t)(1 - R_i(t))^{n-i} = R_m(3R_i^2 - 2R_i^3)$$

where the R_m is the voting machine reliability and R_i is the components reliability.

The voting machine is responsible for two tasks: fault detection and autopilot selection. The autopilots outputs comparison could not be enough to detect a fault or not fast enough to avoid an accident. Therefore monitoring points should be added in internal system levels to increase the autopilot voter reliability, R_m, and so the entire system reliability, R_t. Also the equations 2.5 and 2.4 state that the total system reliability, R_t, is increased when a system component becomes redundant moving from a serial to a parallel connection.

Therefore, it should be a distributed system to reduce the computation complexity in the autopilot selection point, see figure 2.12, and to decrease its failure probability. A fault detection module will be added to all the autopilots where their internal data is shared and compared to select the best one.

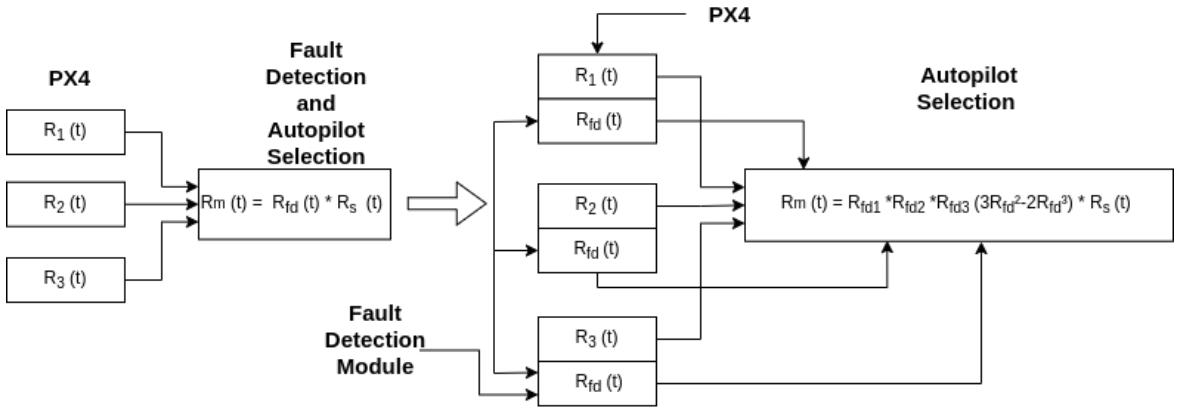


Figure 2.12: Components Reliability in the centralized (on the left) and distributed architecture (on the right).

The mechanisms to get each autopilot state must have an higher reliability: R_{fd1}, R_{fd2} and R_{fd3}. The system judgement and the autopilot selection relies on this information. The system total reliability is computed by:

$$R_{total}(t) = (3R_i^2 - 2R_i^3)R_m$$

Chapter 3

Autopilots Overview And Autopilots Selection

In this chapter, it will be given an overview from the autopilots on the market and the choice justification based on the parameters valued.

Usually, the autopilot hardware and software are sold together on the market because it is easier to improve the autopilot functionality designing a specific hardware directed to work with the target software.

Since there is compatibility with the application programming interface, open source autopilots allow the developers to change and to move the code to other microprocessor boards. However each autopilot software works better in the primarily board which was designed for.

The decision making process to choose the autopilot Software/Hardware which can be fitted on the redundant system plays an important role because the autopilot chosen will influence directly and indirectly the system functionality and all the work to build it as well. The communication depends on the protocol supported by the autopilot Software and Hardware.

There are many possibilities due to hardware and software compatibility between the autopilots. The process starts by dividing the decision making process in three parts: software, communications protocol and hardware.

From here, it will be called firmware instead of software because it is a special class of software which helps to a control device specific hardware on a low level set of instructions.

Since the most valuable practical work of this project is the fault tolerance algorithm, a ”friendly” firmware became the top priority immediately followed by an hardware board compatible with it which can accomplish all the constraints (mainly related with interfaces or communications).

3.1 Autopilot Firmware

As mentioned above, it is desired a ”friendly” firmware which means it must fit in these three conditions:

- **Application Programming Interface (API)** - the point is to simplify the implementation of the soft-

ware with a good interface or communication protocol between all the parts of a computer program. Specifications for routines, data structures, object classes or variable are frequently included in an API. Implementing and debugging ”new changes” in firmware can take a lot of time. This amount is drastically decreased if there is access to a good and well organised API documentation where it is possible to understand clearly the system architecture. Supported protocol communications like MAVLink or another one belong to the API and they are evaluated in this point.

- **Compatibility** - It refers to the variety of boards which can run the firmware chosen and it is related with its API. The microprocessor boards need to communicate with each other. Some boards are better than others on some specifications, like processing speed or different type of communications, so in this point, a firmware which can easily be ported would be a good option.
- **Support/Community** - There is another point related with the number of developers across the world testing or implementing changes in the same firmware. First, the documentation is updated regularly with the latest bugs detected and solved. Also, the probability of having someone who already overtook the same issues throughout the self experience, which the current developer is facing at the moment, increases. It is really helpful participating in the forums where the knowledge about the firmware is shared, allowing to get a lot of answers for personal programming issues.

3.1.1 Market Overview

After enumerating the criteria, a sample of the best known COTS autopilots were evaluated. The point is to use that information in the analytical hierarchy process (AHP) to choose the autopilot.

Ardupilot

ArduPilot (APM) firmware is the leading open source autopilot system considering the number of users or the community size. It means the firmware is well classified by the users and developers in a general opinion. Also APM was strongly tested having a large background across the entire world assuring to all the developers a precious help when programming the firmware or other potential issues. Its compatibility with other boards is great. Its architecture is clear and very well explained in the API documentation which is extremely detailed [19]. The APM can be used already on submarines or antenna trackers beyond the traditional multi-copters and fixed wing aircraft.

PX4

In the second place of the best known open source autopilots, comes the PX4 which is a part from the Dronecode project, Linux Foundation collaborative project. The PX4 firmware has a similar architecture and functionality with the APM.

Like APM, it is also supported by a large amount of flight control boards and by an active world wide community. It powers all kinds of vehicles from racing and cargo drones through to ground vehicles and submersibles.

However it was designed focusing on autonomous drones so it is not the best choice for first-person view users.

The advantage relatively to the APM are the protocol communications and interfaces that will be focused later in this thesis. It is known that in fault detection, the data transfer rate has a big weight on the system performance. It has the best manuals and information on the web [20]. So, it seems to have a better support nowadays than the APM.

Paparazzi

Here is another drone software and hardware project encompassing autopilot systems and ground station software.

This firmware purpose is to be used with the Paparazzi boards, despite of having been ported to other popular autopilots allowing for the use of the Paparazzi UAV system on many kind of hardware platforms. The Paparazzi designers primary focus was the autonomous flight, leaving the manual flying to a secondary plan. It can be used for multi-rotors, fixed wing, helicopters and hybrid aircraft.

The API documentation [21] is worse when compared with the PX4 or APM respective documentation. It remains less popular than the first ones, implying a smaller community and background to solve possible issues.

LibrePilot

A firmware designed specifically to the Open Pilot boards family. There is not so much support in the internet when compared with ArduPilot or PX4. One reason is due to the limited compatibility with other boards, so the LibrePilot community is smaller than other open source COT autopilots. [22]

3.1.2 Analytical Hierarchy Process (AHP)

In decision making, a process that combines the rating parameters with the firmware options available was needed. A good option to choose the best firmware is the AHP [14] application. The following steps are applied on this decision-matrix method.

A possible rating scale for the parameters, referred in the section Market Overview, is to choose the odd numbers from 1 to 9, table 3.1, to increase the difference between the parameters and so, the final computed scores.

Table 3.1: AHP Scaling

Definition	**Grade**
Equal importance - two activities contribute equally to the objective	1
Moderate importance of one activity over another	3
Essential or strong importance	5
Very strong importance	7
Extreme importance	9

Each parameter importance is compared with the others parameters importance using the grading scale from the table 3.1. From each comparison results a factor value. The criteria weigh is calculated through the geometric mean ($\sqrt[n]{a_1 * a_2 ... * a_n}$) relative to the parameter factors reducing inconsistency in the matrix. a_n represents the factor value inserted in the column n from the respective criteria resulted from the comparison between the two parameters importance. The API is the main point to choose the autopilot firmware. It was considered to grade this parameter importance by 7 when compared to the firmware compatibility importance and by 3 when compared with the support importance, first line of the table 3.2. The geometric mean from the API parameter, $API_{Geometricmean}$ is computed using its line factors:

$$API_{Geometricmean} = (\sqrt[n]{a_1 * a_2 * a_3}) = (\sqrt[3]{1 * \frac{1}{7} * 3}) = 2.47 \tag{3.1}$$

The same process was repeated for the two remaining parameters. To compute the weigh from each parameter, its geometric mean is divided by the sum from all the parameters geometric mean. The variable $FirmwareC_{Geometricm}$ represents the firmware compatibility geometric mean and the $Support_{Geometricm}$ variable represents the support parameter geometric mean. To exemplify, the API geometric mean computation is shown below:

$$API_{weigh} = \frac{API_{Geometricm}}{API_{Geometricm} + FirmwareC_{Geometricm} + Support_{Geometricm}} = \tag{3.2}$$

$$\frac{2.47}{2.47 + 0.36 + 1} = 0.45$$

The same process is applied to each parameter and the table 3.4 was filled.

Table 3.2: AHP - computing criteria weights

	API	Firm C[1]	C/S[2]	GM[3]	Weights
API	1	7	3	2.47	0.45
Firm C[1]	$\frac{1}{7}$	1	$\frac{1}{3}$	0.36	0.25
C/S[2]	$\frac{1}{3}$	3	1	1	0.3

The support parameter has more influence to the final choice than the firmware compatibility. So the support was rated by 3 when compared with the compatibility.

A similar process is done to rate each autopilot different firmware. The firmware autopilots are compared to each other considering just one parameter at a time. Each table refers to just one parameter. It is filled with the graded factor values resulted from the comparisons between the firmware autopilots relative to that parameter. Each firmware rating relative to a parameter is calculated through the geometric mean ($\sqrt[n]{a_1 * a_2 ... * a_n}$) to reduce inconsistency in the matrix. a_n represents the factor value inserted in the column n of the correspondent autopilot. For example, to compute the API rating from the PX4 on the first line of the table 3.3:

$$PX4_{Geometricmean} = (\sqrt[n]{a_1 * a_2 * a_3 * a_4}) = (\sqrt[4]{1 * 5 * 7 * 9}) = 4.21 \tag{3.3}$$

$$PX4\_rating_{API} = \frac{PX4_{Geometricm}}{PX4_{Geometricmean} + Ardupilot_{Geometricm} + Paparazzi_{Geometricm} + Librep_{Geometricm}} = \tag{3.4}$$

$$= \frac{4.21}{4.21 + 1.63 + 0.54 + 0.007} = 0.66$$

On the tables 3.3, 3.4 and 3.5 are calculated the ratings from each autopilot firmware relative to each parameter (API, Compatibility and Support).

The PX4 firmware has a better API like is demonstrated in the table 3.3. Both PX4 and Ardupilot have a good firmware compatibility, table 3.4 to run in different hardware. PX4 stands out for support, table 3.5 because of the documentation available [20].

For the total scores, each autopilot rating relative to a specific parameter is multiplied by that parameter weight. The process is repeated for all the parameters and summed to compute the autopilot firmware total score. For example:

$$PX4_{totalscore} = PX4_{API} * API_{weight} + PX4_{Firmcompat} * Firmcompat_{weight} + PX4_{Supp} * Supp_{weight} = \tag{3.5}$$

$$= 0.66 * 0.45 + 0.42 * 0.25 + 0.56 * 0.3 = 0.537$$

The firmware with the highest sum of ratings is the selected one. The choice fell in PX4 firmware justified by the total scores presented in the table 3.6. The PX4 is the best firmware autopilot and from this point, all the strategies to build the redundant system will be conditioned by this option like the hardware identification.

[1] Firmware Compatibility
[2] Community/Support
[3] Geometric Mean

Table 3.3: AHP - API ratings

	PX4	Ardupilot	Paparazzi	Librepilot	GM [1]	API R [2]
PX4	1	5	7	9	4.21	0.66
Ardupilot	$\frac{1}{5}$	1	5	7	1.63	0.26
Paparazzi	$\frac{1}{7}$	$\frac{1}{5}$	1	3	0.54	0.09
Librepilot	$\frac{1}{9}$	$\frac{1}{5}$	$\frac{1}{3}$	1	0.007	0.001

Table 3.4: AHP - Firmware Compatibility ratings

	PX4	Ardupilot	Paparazzi	Librepilot	GM [1]	CR[3]
PX4	1	1	5	7	2.43	0.42
Ardupilot	1	1	5	7	2.43	0.42
Paparazzi	$\frac{1}{5}$	$\frac{1}{5}$	1	3	0.59	0.1
Librepilot	$\frac{1}{7}$	$\frac{1}{7}$	$\frac{1}{3}$	1	0.29	0.05

Table 3.5: AHP - Support/Community ratings

	PX4	Ardupilot	Paparazzi	Librepilot	GM[1]	SR[4]
PX4	1	3	7	7	3.48	0.56
Ardupilot	$\frac{1}{3}$	1	7	7	2.01	0.32
Paparazzi	$\frac{1}{7}$	$\frac{1}{7}$	1	1	0.38	0.06
Librepilot	$\frac{1}{7}$	$\frac{1}{7}$	1	1	0.38	0.06

Table 3.6: AHP - Total scores

Criteria	Weights	PX4	Ardupilot	Paparazzi	Librepilot
API	0.45	0.66	0.26	0.09	0.001
Firm C [5]	0.25	0.42	0.42	0.1	0.05
Support/Community	0.3	0.56	0.32	0.06	0.06
Total Scores	1	0.537	0.318	0.0835	0.0309

[1] Geometric Mean
[2] API Rating
[3] Firmware Compatibility Rating
[4] Community/Support Rating
[5] Firmware Compatibility

3.2 Communication protocol

Triple redundant systems need to communicate between each other before taking the decision about which of them must be on charge. Therefore, communication is a special key to the system efficiency.

So, before deciding about the autopilot hardware, the communication protocols were studied. The protocols adopted will influence the hardware decision because of the necessary ports or interface compatibility. To choose the communication protocol some parameters will be evaluated:

- Data rate, possible simultaneous transmission directions (Duplex, Half duplex)
- Data protection - Error Checking
- Complexity - number of wires or ports

3.2.1 Universal asynchronous reception and transmission (UART)

UART is a simple serial communication protocol that supports bidirectional, asynchronous and serial data transmission.

UART has three operation modes: simplex (data transmission in one direction), half duplex (data transmission in both directions but not simultaneously) and full duplex (data transmission in both directions simultaneously).

The data is transmitted through the Tx pin while the receiving UART reads the data through its Rx pin. The transmission speed is defined by the baud rate (115,200 bits per second by default). It performs error checking with the help from a parity bit. The start and stop bit create the data packets. Those packets are sent to a UART buffer - first in, first out (FIFO) - which forces each transmitted byte to be passed to the receiving UART.

Just to summarise the main features: no clock is needed (asynchronous), error checking capability, low speed and low complexity.

3.2.2 Inter-Integrated-Circuit (I2C)

Despite of being a synchronous protocol, I2C is similar to UART. It is used for modules and sensors communication and it requires just two wires (Serial Clock Line / Serial Data line Acceptance port) to transmit information between devices connected to the bus. Many different devices, up to 128, can be connected throughout the same wires, figure 3.1). All the data is transmitted on a single wire (SDA), figure 3.2. Despite of being flexible, easy and cheap, it still have a lower speed, requires pull-up resistors and has reduced noise immunity.

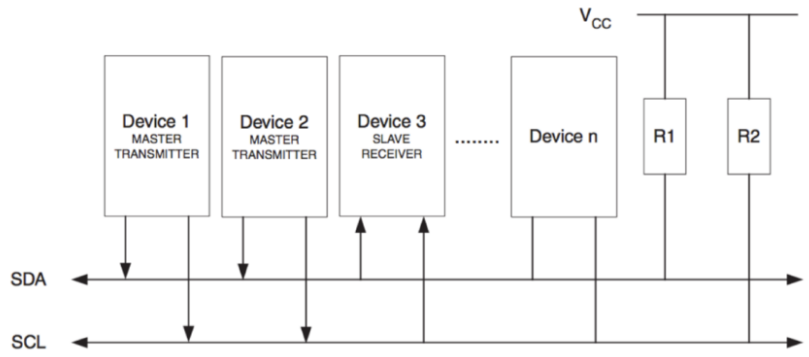


Figure 3.1: I2C wiring diagram: SDA and SCL. Multi Master and Slaves. [23]

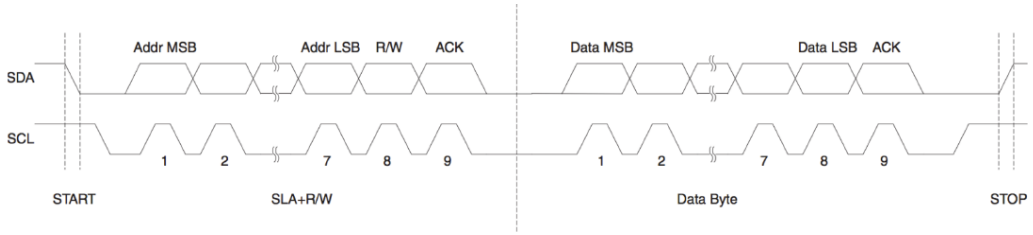


Figure 3.2: Transmission timing diagram, including the address and the acknowledgement bit[23]

3.2.3 Serial Peripheral Interface (SPI)

SPI is another synchronous protocol which operates at a full duplex mode (data can be sent and received simultaneously) and at a faster data rate transmissions (8Mbits/s). It does not require addressing but it needs slave select lines. Four ports are connected: Master Data Output, Slave Data Input (MOSI), Master Data Input and Slave Data Output (MISO), SCLK (Clock Signal) and Slave Select (SS), controlled by the master device, figure 3.3.

Despite of having good noise immunity, it is more more complex and expensive because of the pins ports occupied (slightly more complicated in hardware when compared with I2C) and there is no error checking.

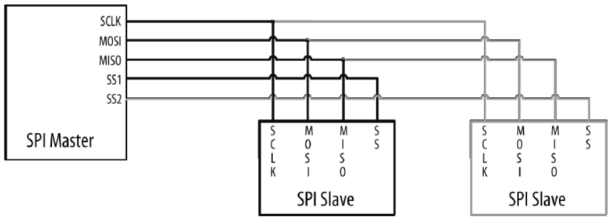


Figure 3.3: SPI wiring protocol using MOSI, SCLK, MISO and SS ports to connect the master to the slaves[23]

3.2.4 Analytical Hierarchy Process (AHP)

The AHP will be applied to the communication protocol with the same scale from the table 3.1 with the correspondent communication protocol criteria. The table 3.7 was filled with the values resultant from the criteria comparison.

Table 3.7: AHP - computing criteria weights for communication protocol

Criteria	Data Rate	Data P [1]	Complexity	GM[2]	Weights
Data Rate	1	1	3	1.44	0.43
Data P[1]	1	1	3	1.44	0.43
Complexity	$\frac{1}{3}$	$\frac{1}{3}$	1	0.48	0.14

From the weights computed in table 3.7, it is considered that complexity is not so influential as the data rate or data protection.

The communication protocols were compared with each other relatively to one parameter at a time. The tables were filled with the resulting values from the comparisons:

- Table 3.8 - Considering the data rate performance;
- Table 3.9 - Considering the data protection performance;
- Table 3.10 - Considering the complexity performance.

Table 3.8: AHP - Communication Protocol, Data Rate

	UART	I2C	SPI	GM[2]	Data rate
UART	1	$\frac{1}{3}$	$\frac{1}{5}$	0.41	0.1
I2C	3	1	$\frac{1}{3}$	1	0.26
SPI	5	3	1	2.47	0.64

Table 3.9: AHP - Communication Protocol, Data Protection

	UART	I2C	SPI	GM[2]	Data P[1]
UART	1	7	5	3.27	0.73
I2C	$\frac{1}{7}$	1	$\frac{1}{3}$	0.36	0.08
SPI	$\frac{1}{5}$	3	1	0.84	0.19

[1] Data Protection
[2] Geometric Mean

Table 3.10: AHP - Communication Protocol, Complexity

	UART	I2C	SPI	GM[1]	Complexity
UART	1	3	9	3	0.67
I2C	$\frac{1}{3}$	1	5	1.19	0.27
SPI	$\frac{1}{9}$	$\frac{1}{5}$	1	0.28	0.06

I2C is the best communication protocol when considering the data rate parameter, see table 3.8. From the table 3.9 inspection, it can be concluded UART has a better rating at data protection than the other two protocols because of its error checking capability. UART is the protocol with less wiring complexity (in this parameter, a better rate means less complexity).

The total scores were computed repeating the same process from the table 3.6, multiplying the ratings for each parameter by its weight and summing all.

Table 3.11: AHP - Communication Protocols Total Scores

Criteria	Weights	UART	I2C	SPI
Data Rate	0.43	0.1	0.26	0.64
Data Protection	0.43	0.73	0.08	0.19
Complexity	0.14	0.67	0.27	0.06
Total scores	1	0.45	0.18	0.37

From the table 3.11 scores, it was decided by the UART connection.

3.3 Message Protocols

After the hardware communication protocol, a choice has to be made about how the messages will be codified to be understood by all subsystems since the autopilots need to share information with each other as well as with other interfaces to vote.

The PX4 firmware uses two different message protocols for external communication: Mavlink and RTPS. The external communication can be with the ground control station (GCS), companion computer, other autopilots or even the simulator.

3.3.1 Micro Air Vehicle Message Marshalling Library (MAVLink)

MAVLink is used for communication between the UAS components.

[1] Geometric Mean

Python tools are used to convert the XML files into source code for the supported languages. Those XML files are defined by a message-set specifications for different systems. It is a lightweight and header-only library.

It is optimized for resource-constrained systems and implemented in C programming language.

The most popular high level protocol communication between UAV and/or ground stations. It has inter-operability interface between components from different manufacturers and it was thoroughly tested.

Since MAVLink is an higher level protocol, it must use a lower level protocol to communicate between devices. USB (to communicate with the CPU for firmware updates or simulations) and UART are the predefined ports to be used with MAVLink by the autopilots firmware. [24]

3.3.2 Real Time Publish Subscribe (RTPS)

Data Distribution Service (DDS) is a middleware protocol and API standard for data centric connectivity from the object management group (OMG) and it can be used for a real time publish subscribe interface to PX4.

RTPS is a C++ implementation of the DDS RTPS protocol, which provides a decoupled communication middleware with a model based on publishers and subscribers, over unreliable transport protocols such as UDP enabling the exchange of uORB messages between the PX4 components and (offboard) RTPS applications.

This high-performance, dependable and inter-operable data communication has been adopted as the middleware for the Robot Operating System 2 (ROS2) allowing a better integration with it using a RTPS bridge.

It is lightweight and fully open source.

As the MAVLink protocol, RTPS is an higher level protocol and it uses low level UART ports to communicate between the devices. [25]

3.3.3 Message Protocol Decision

Since there are just two options, the justification is given by an explanation comparing the two message protocols.

Fault detection systems need to reliably share time-critical/real-time information between the flight controllers and/or off board components (by sending and receiving uORB topics) which is provided by RTPS. On a first approach RTPS has advantage because of its greater speed than MAVLink.

However there is just one model available (quadcopter) to use this protocol to communicate with the simulator since this protocol is still recent. The communications between the simulator and the PX4 firmware are crucial to design the voting system and to make the selection. Any fixed-wing UAV can not be tested on the simulator using this protocol.

RTPS is a PX4 exclusive messaging protocol, so there is not the possibility to test the designed redundant system with other firmware autopilots.

Moreover MAVLink is still the most used protocol between UAS by far. There is a greater support and community which brings more safety for the developer to overtake possible errors at programming.

MAVLink will be the protocol used to communicate with external systems from each PX4.

3.4 Autopilot Hardware Identification

Due to PX4 great compatibility with many processing boards, the decision has to be made based on the parameters:

- Support, well-tested and stable;
- Flexibility in terms of hardware peripherals that can be attached by its communication protocol support and available sensors;
- Processing speed, RAM and affordability.

3.4.1 Market Overview

From the large list of compatible boards with PX4, four were chosen to be analysed and to apply the AHP to decide which hardware better fits on the PX4 firmware.

Cube Flight Controller

A flight controller which belongs to the Pixhawk Series. The CEiiA organisation already owns one which was tested in Delta-Spotter (UAV) with good performance. The technical information from this autopilot relevant for the evaluation parameters from the AHP is shown below. [26]

Processing speed: $168MHz$. RAM: $256KB$. On-board sensors: 2 Barometers, 2 compasses and 3 IMU's. Abundant connectivity options for additional peripherals (5x UART, I2C, CAN). Cost: 200€

Holybro Pixhawk 4

Pixhawk 4 is the latest update to the family of Pixhawk flight controllers. With the increased power and RAM resources, more complex algorithms and models can be implemented on the autopilot. The technical information from this autopilot relevant for the evaluation parameters from the AHP is shown below. [27]

Processing speed: $216MHz$. Ram: $512KB$. Onboard-sensors: 2 IMU's, 1 Magnetometer, 1 Barometer. Cost: 250€.

BeagleBone Blue

The all-in-one Linux-based computer has all necessary sensors and peripherals needed by a flight controller despite of being optimised for robotics. The technical information from this autopilot relevant for the evaluation parameters from the AHP is shown below. [28]

Processing speed: $1GHz$. Ram: $512MB$. Onboard-sensors: 2 IMU's, 1 Magnetometer, 1 Barometer. Cost: 250€.

Qualcomm Flight Pro

It is an onboard computer which runs the PX4 flight stack on the QuRT real time operating system. The DSPAL API is used for POSIX compatibility. It adds a camera, wifi and high-end processing power in comparison with Pixhawk autopilots. It is focused on the video and photography communications. The technical information from this autopilot relevant for the evaluation parameters from the AHP is shown below.[29]

Processing speed: $2.26GHz$. Ram: $2GB$. Onboard-sensors: 1 IMU's, 1 Magnetometer, 1 Barometer. Cost: 850€.

3.4.2 Analytical Hierarchy Process (AHP)

The AHP was applied to choose the best option. The criteria chosen to evaluate the autopilot hardware are shown on table 3.12. Since the four boards have UART ports to communicate by *Mavlink*, this criteria was not considered to make the decision. Each criteria weight was computed, repeating a similar process used to fill the tables, 3.7 and 3.2. All the criteria are compared with each other using the same scale presented on the table 3.1 to calculate each parameter weight.

Table 3.12: AHP - computing criteria weights for autopilot hardware

Criteria	AS[1]	CS/R[2]	Afford[3]	Support	GM[4]	Weights
Available Sensors	1	1	3	3	1.73	0.37
Clock speed/RAM	1	1	3	3	1.73	0.37
Affordability	$\frac{1}{3}$	$\frac{1}{3}$	1	1	0.58	0.13
Support	$\frac{1}{3}$	$\frac{1}{3}$	1	1	0.58	0.13

Available sensors and clock speed combined with RAM are the main factors which will help to decide which autopilot hardware will be used on this project, see table 3.12. Both the parameters have a bigger weight - 0.37 - than the other two parameters - 0.13.

To compute the autopilot hardware rating for each parameter, it was repeated the same process used in table 3.8. Initially the autopilots were compared with each other considering just one parameter. After filling the table with the values resultant from the comparisons, the geometric mean was calculated for each line from the table. Dividing the geometric mean from the autopilot hardware by the total geometric means sum, gives the rating for the correspondent parameter.

[1] Available Sensors
[2] Clock Speed/Ram
[3] Affordability
[4] Geometric Mean

Table 3.13: AHP - Autopilot Hardware, Available Sensors Rating

	PC[1]	PH[2]	BBB[3]	Qualcom	GM[4]	ASR[5]
Pixawk Cube	1	5	5	9	3.87	0.63
Pixawk Holybro	$\frac{1}{5}$	1	1	5	1	0.16
Beagle-Bone Blue	$\frac{1}{5}$	1	1	5	1	0.16
Qualcom	$\frac{1}{9}$	$\frac{1}{5}$	$\frac{1}{5}$	1	0.26	0.04

Table 3.14: AHP - Autopilot Hardware, Clock Speed / RAM Rating

	PC[1]	PH[2]	BBB[3]	Qualcom	GM[4]	CS/RAM
Pixawk Cube	1	3	5	9	3.41	0.58
Pixawk Holybro	$\frac{1}{3}$	1	3	5	1.5	0.26
Beagle-Bone Blue	$\frac{1}{5}$	$\frac{1}{3}$	1	3	0.67	0.11
Qualcom	$\frac{1}{9}$	$\frac{1}{5}$	$\frac{1}{3}$	1	0.3	0.05

Pixawk Cube is the autopilot hardware with more sensors available and it was demonstrated on the table 3.13. Also it has the best processor, table 3.14. By the other side, Pixawk HolyBro has the smallest cost as it can be seen from the table 3.15. Both Pixawk have a big advantage relative to the Beagle-Bone Blue or Qualcom when considering the Support and Community, table 3.16.

Since all the autopilots parameter ratings are calculated, the total scores table 3.17 can be filled with them and the corresponding parameters weight repeating the process again from the table 3.11.

Analysing the total scores table, 3.17, it is clear the Pixawk Cube is the best board to combine with the PX4 firmware.

Table 3.15: AHP - Autopilot Hardware, Affordability Rating

	PC [1]	PH[2]	BB[3]	Qualcom	GM[4]	Afford[5]
Pixawk Cube	1	$\frac{1}{3}$	1	5	1.14	0.24
Pixawk Holybro	3	1	3	7	2.28	0.47
Beagle-Bone Blue	1	$\frac{1}{3}$	1	5	1.14	0.24
Qualcom	$\frac{1}{5}$	$\frac{1}{7}$	$\frac{1}{5}$	1	0.28	0.06

[1]Pixawk Cube
[2]Pixawk Holybro
[3]Beagle-Bone Blue
[4]Geometric Mean
[5]Available Sensors Rating

Table 3.16: AHP - Autopilot Hardware, Support Rating

	PC[1]	PH[2]	BBB[3]	Qualcom	GM[4]	Support[6]
Pixawk Cube	1	1	3	3	1.73	0.37
Pixawk Holybro	1	1	3	3	1.73	0.37
Beagle-Bone Blue	$\frac{1}{3}$	$\frac{1}{3}$	1	1	0.58	0.13
Qualcom	$\frac{1}{3}$	$\frac{1}{3}$	1	1	0.58	0.13

Table 3.17: AHP - Autopilot hardware Total Scores

Criteria	Weights	PC	PH	BBB	Qualcom
Available Sensors	0.37	0.63	0.16	0.16	0.04
Clock Speed / RAM	0.37	0.58	0.26	0.11	0.05
Affordability	0.13	0.24	0.47	0.24	0.06
Support	0.13	0.37	0.37	0.13	0.13
Total Scores	1	0.527	0.27	0.148	0.07

Therefore the autopilot is already known as well as the protocols to communicate with it.

Resuming this chapter, the autopilot firmware is the PX4 which will be used on the Pixawk Cube board. It will connect to other systems through UART ports using Mavlink codified messages.

[1] Pixawk Cube
[2] Pixawk Holybro
[3] Beagle-Bone Blue
[4] Geometric Mean
[5] Available Sensors Rating
[6] Affordability Rating
[7] Support Rating

Chapter 4

Redundant System Design

It is time to materialize the ideas from the systems studied in the section 1.2, using the background knowledge explained on chapter 2 to apply on the chosen firmware autopilot. From now, when the PX4 is mentioned, it refers to the PX4 autopilot firmware .

Several strategies were adopted to build the fault tolerant system while studying the PX4 firmware code looking for the best way to detect and mitigate faults from its internal mechanisms. Therefore many changes occurred since the first approach aiming to a final effective algorithm which can add more value to the entire system and not simply replace strong mechanisms for fault detection already existent inside the autopilot.

The different approaches will be presented on this chapter, following a chronological order while a better understanding from the PX4 increases until reaching the final algorithm.

Three PX4 will be running on parallel. The number of autopilots needed to design the system is justified by the trade off between the enough redundancy for fault detection and the system complexity like it was explained on the redundancy section 2.2.3. It is considered that the failure probability of the three autopilots at the same time is very low.

4.1 First Approach

The Reliability, see definition 2.2.2, states the entire system $R(t)$ increases if its belonging subsystems $R_i(t)$ increases too. By this reason, the PX4 autopilot was divided in two subsystems to be added to the new subsystem responsible for fault tolerance:

- Sensors module;
- Flight control module.

To situate the first two modules inside the autopilot, the PX4 architecture is shown in figure 4.1 where it is noticed the estimator module between the sensors hub and the control module. On this approach, the estimator module is included on the flight control module for simplification reasons.

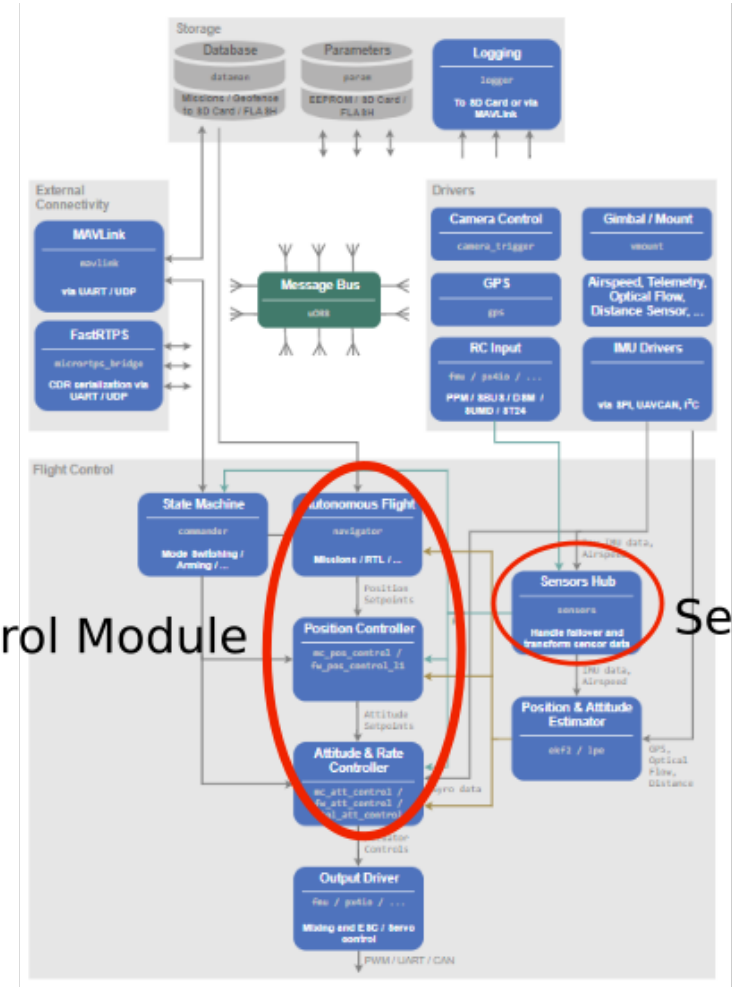


Figure 4.1: PX4 firmware architecture. [20]

The point is to raise the fault detection and autopilot selection modules reliability, adding a monitoring point after each sensor, used by the PX4, to compare the values with the same sensor type from the other two autopilots. The fault detection capability and speed increase and so the fault detection module reliability increases too.

Because of the noise, phase shifts and synchronization the comparison between sensors values it is not an easy task. It can make the system to take the wrong decision when looking for real faults. One diagram from this approach is demonstrated in figure 4.2, where is exemplified with just one sensor, how the system could share the values from that sensor, in real time. Throughout a monitoring point, before the flight control unit, it is possible to compare the referred data, looking for a sensor failure. In case of sensor failure, the entire set of the belonging sensor would be classified unhealthy and its outputs could not be selected to be sent to the actuators.

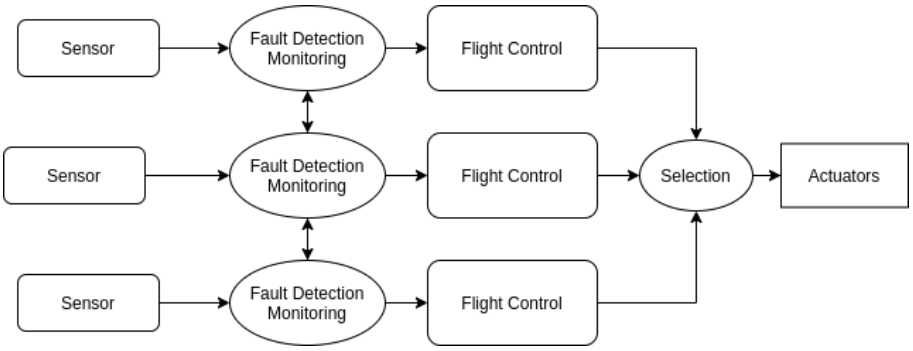


Figure 4.2: First approach diagram

While studying the sensors module from PX4 to get and treat the correspondent data in real time, it was noticed PX4 already has fault tolerant mechanisms to isolate and mitigate sensor failures. After the sensors data being treated, by low pass filters to eliminate the noise or other normal peaks, the process uses already redundancy at the sensors level. Throughout data comparison between sensors from the same typology, it classifies each sensor priority and selects the best source to use as observation, figure 4.3. The barometer was selected as an example but it could be any other redundant sensor (GPS and optical flow do not use redundancy).

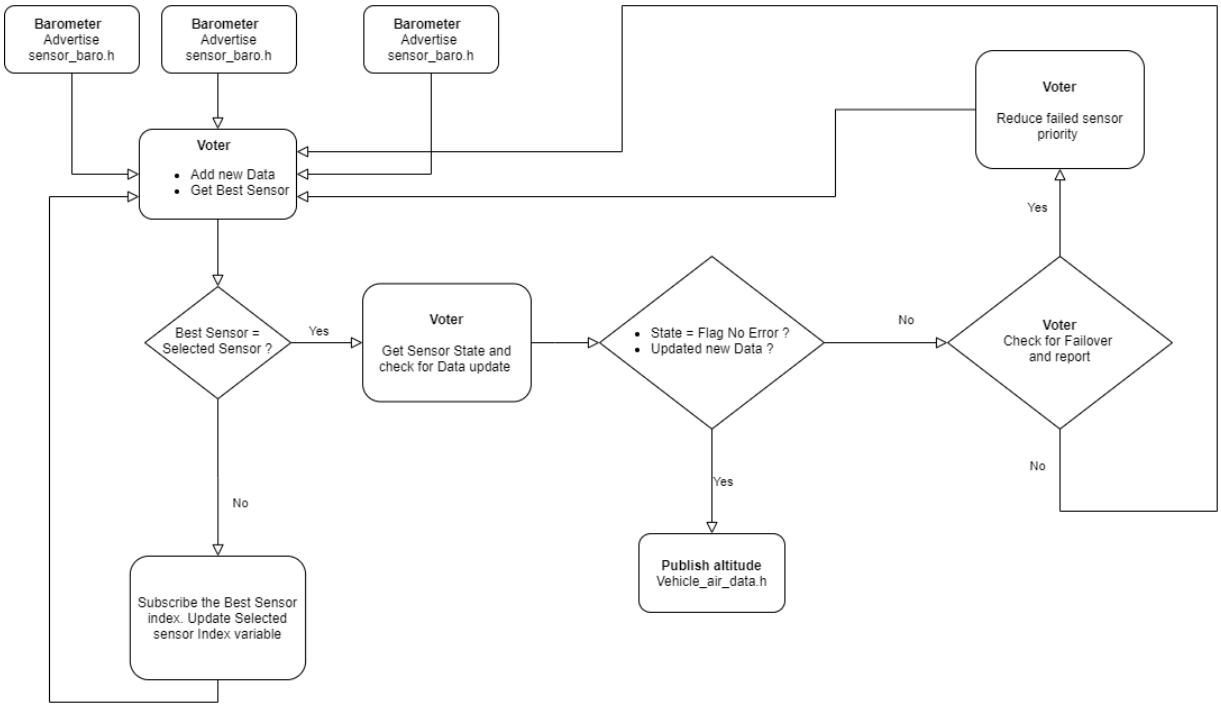


Figure 4.3: PX4 Sensor Module

If the sensor is not publishing data (error timeout) or if the sensor value overtakes the threshold from the mean of all sources data, the voter changes the sensor state to an error state, decreases the sensor

priority and it will check if the failure sensor is the selected one to change it if needed. The sensor state or priority can go up and down many times during a flight since the algorithm evaluates the sensor state in a time window constantly updated with the most recent data and dropping the record data that does not fit on the window length.

There is still the possibility of using redundancy working on an outer ring with the final values from the selected sensors.

Meanwhile the data published from the selected sensor (the best ranked sensor from its typology) follows to the Extended Kalman filter for state estimation usually as observation (exception for the IMU measurements). On the diagram 4.4 is shown an overview from the flight stack to understand where the states estimation occur and its importance for the control unit. The EKF refers to the red rectangle.

The inputs come from the sensors module while the outputs from the states estimation are used by the controller units as well as by the navigator. The mission is uploaded to the navigator module which defines the current waypoint to be achieved by the UAV and stores the next ones until the end of the mission. The navigator output is received by the Position Controller in the Mission Mode.

For fully autonomous flights, a companion computer must be used and the communication with the PX4 is made throughout the navigator module. If the user needs more freedom to control the UAV during the flight, he can choose other different modes using the radio control unit. For example, on the Acrobatic mode, more freedom is given to the autopilot allowing to send commands directly to the altitude and rate controller unit which receives the outputs from the position controller as well.

Finally the mixer translates force commands (torques, forces) to actuator commands and it hides the aircraft actuators specifications from the core controllers to ensure the different typologies do not require special case handling by them. The actuators group depend on each vehicle typology, they can be just motors like in the multi rotors case (figure 2.2(b)). They can be servos too in the fixed wing or hybrid typologies, figures 2.2(a) and 2.2(c) respectively.

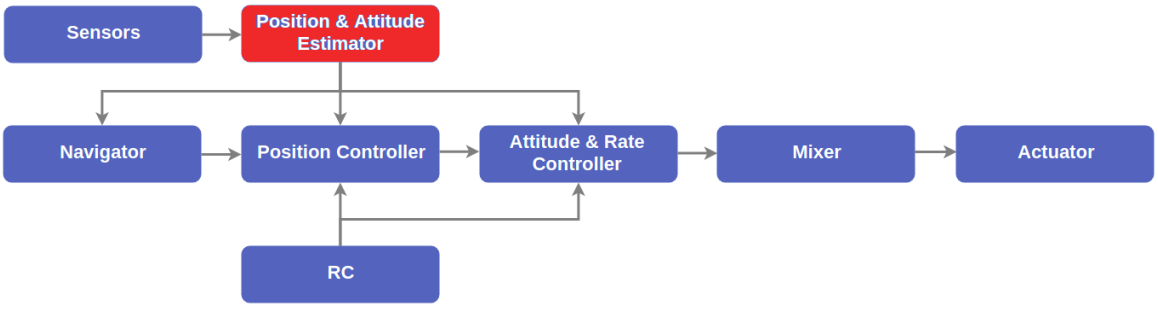


Figure 4.4: Building blocks from flight stack architecture. Estimation control library in red.[20]

So it is given a brief explanation from the EKF computing algorithm and its implementation on the PX4 for state estimation in the next section 4.1.1.

4.1.1 Extended Kalman Filter (EKF)

The EKF is implemented throughout the introduction of the system dynamics and the observations linearized model relative to the current estimated state, refining the estimation with the sensor measurements. Those states are used by the control unit [30].

The EKF computes two steps for each iteration k, since it works in discrete time:

- **Prediction** - the predicted state estimate $\hat{\mathbf{x}}_{k+1}^{-}$ is computed using the state estimate $\hat{\mathbf{x}}_{k}^{+}$ from the previous iteration and the input $u(k)$ (the input is missing in this flowchart 4.5),
- **Filtering** - the $\hat{\mathbf{x}}_{k}^{+}$ is updated using the predicted state estimate and the current observations.

In the figure 4.5, all the steps from the EKF algorithm computation are demonstrated.

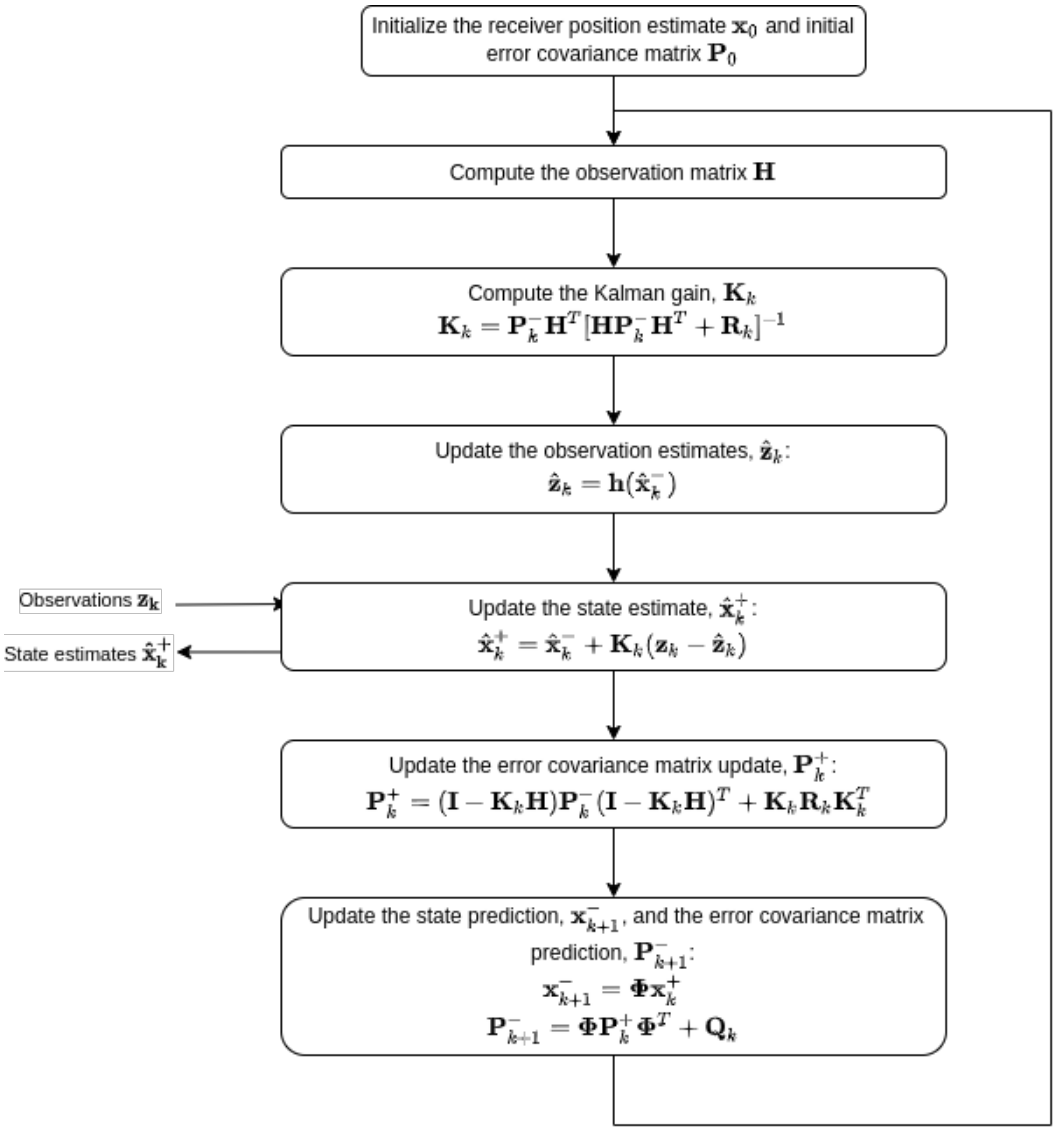


Figure 4.5: Flowchart of the EKF algorithm [30]

Dynamics model

The EKF has 24 states, equation (4.1), where the first states (q_0, q_1, q_2, q_3) correspond to the quaternions that define the angular position (rotation) from the XYZ body frame relative to the (North, East, Down) Navigation Inertial Reference Frame and the 6 next states $V_N, V_E, V_D, P_N, P_E, P_D$ refers to the velocity and position in the Navigation Inertial Reference Frame. The first 10 states capture the position information through a dynamic process model. The states $\Delta_{ang\_bias\_x}, \Delta_{ang\_bias\_y}, \Delta_{ang\_bias\_z}$ and $\Delta_{vel\_bias\_x}, \Delta_{vel\_bias\_y}, \Delta_{vel\_bias\_z}$ refer to the gyro delta angle bias and accelerometer delta velocity bias respectively (both gyro and accelerometer belong to the IMU - Inertial Measurement Unit). Following the matrix columns order, 4.1, M_N, M_E, M_D and M_X, M_Y, M_Z represent the magnetic field on the navigation inertial frame and on the body frame respectively. Finally the states V_{wind_N}, V_{wind_E} refers to the wind velocity on the Navigation Inertial Frame. The equations from this section came from the PX4 estimation control library document [31].

$$\mathbf{X} = \begin{bmatrix} q_n(4) \\ V_{(NED)} \\ P_{(NED)} \\ \Delta_{ang\_bias\_(xyz)} \\ \Delta_{vel\_bias\_(xyz)} \\ M_{(NED)} \\ M_{(XYZ)} \\ V_{wind_N} \\ V_{wind_E} \end{bmatrix} \tag{4.1}$$

From the IMU measurements (gyro + accelerometer), Δ_{ang_meas} and Δ_{vel_meas} are defined by:

$$\Delta_{ang\_meas} = \begin{bmatrix} \Delta_{ang\_meas\_x} \\ \Delta_{ang\_meas\_y} \\ \Delta_{ang\_meas\_z} \end{bmatrix} = \int_{t_k}^{t_{k+1}} \vec{w} dt \tag{4.2}$$

$$\Delta_{vel\_meas} = \begin{bmatrix} \Delta_{vel\_meas\_x} \\ \Delta_{vel\_meas\_y} \\ \Delta_{vel\_meas\_z} \end{bmatrix} = \int_{t_k}^{t_{k+1}} \vec{a} dt \tag{4.3}$$

The truth delta angles Δ_{ang_truth} are calculated from the IMU measurements and delta angle bias states Δ_{ang_bias}:

$$\Delta_{ang\_bias\_(x,y,z)} = \begin{bmatrix} \Delta_{ang\_bias\_x} \\ \Delta_{ang\_bias\_y} \\ \Delta_{ang\_bias\_z} \end{bmatrix} \tag{4.4}$$

$$\Delta_{vel\_bias\_(x,y,z))} = \begin{bmatrix} \Delta_{vel\_bias\_x} \\ \Delta_{vel\_bias\_y} \\ \Delta_{vel\_bias\_z} \end{bmatrix} \tag{4.5}$$

$$\Delta_{ang\_truth} = \Delta_{ang\_meas} - \Delta_{ang\_bias\_(x,y,z)} \tag{4.6}$$

$$\Delta_{vel\_truth} = \Delta_{vel\_meas} - \Delta_{vel\_bias\_(x,y,z)} \tag{4.7}$$

The rotation matrix from Body Frame to the Navigation frame is given by:

$$[T]_B^N = \begin{bmatrix} q_0^2 + q_1^2 - q_2^2 - q_3^2 & 2(q_1q_2 - q_0q_3) & 2(q_1q_3 + q_0q_2) \\ 2(q_1q_2 - q_0q_3) & q_0^2 - q_1^2 + q_2^2 - q_3^2 & 2(q_2q_3 - q_0q_1) \\ 2(q_1q_3 - q_0q_2) & 2(q_2q_3 + q_0q_1) & q_0^2 - q_1^2 - q_2^2 + q_3^2 \end{bmatrix} \tag{4.8}$$

The quaternion Δ_{quat} defines the rotation from frame k to $k+1$. The truth delta angle, Δ_{ang_truth}, is used to calculate Δ_{quat} using a small angle approximation:

$$\Delta_{quat} = \begin{bmatrix} \Delta q_0 \\ \Delta q_1 \\ \Delta q_2 \\ \Delta q_3 \end{bmatrix} = \begin{bmatrix} 1 \\ \frac{\Delta_{ang\_truth\_x}}{2} \\ \frac{\Delta_{ang\_truth\_y}}{2} \\ \frac{\Delta_{ang\_truth\_z}}{2} \end{bmatrix} \tag{4.9}$$

To rotate the quaternion state forward from frame k to $k+1$ the Δ_{quat} is used in the quaternion product rule:

$$\begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \end{bmatrix}_{k+1} = \begin{bmatrix} q_0\Delta q_0 - q_1\Delta q_1 - q_2\Delta q_2 - q_3\Delta q_3 \\ q_0\Delta q_1 q_0 + q_1\Delta q_0 + q_2\Delta q_3 - q_3\Delta q_2 \\ q_0\Delta q_2 + q_2\Delta q_0 - q_1\Delta q_3 - q_3\Delta q_1 \\ q_0\Delta q_3 + q_3\Delta q_0 + q_1\Delta q_2 - q_2\Delta q_1 \end{bmatrix} \tag{4.10}$$

The velocity states from frame k to $k+1$ are calculated by the truth delta velocity vector, Δ_{vel_truth}, rotated from the body frame to the Inertial Navigation frame and subtracting gravity:

$$\begin{bmatrix} V_N \\ V_E \\ V_D \end{bmatrix}_{k+1} = \begin{bmatrix} V_N \\ V_E \\ V_D \end{bmatrix}_k + [T]_B^N.\Delta_{vel\_truth} + \begin{bmatrix} 0 \\ 0 \\ g \end{bmatrix}.\Delta t \tag{4.11}$$

The position estimates are updated:

$$\begin{bmatrix} P_N \\ P_E \\ P_D \end{bmatrix}_{k+1} = \begin{bmatrix} P_N \\ P_E \\ P_D \end{bmatrix}_k + \begin{bmatrix} V_N \\ V_E \\ V_D \end{bmatrix}_k .\Delta t \tag{4.12}$$

The remaining states (IMU sensor bias, magnetic field and wind) use a static process model. They do not change from k to $k+1$ frame.

The accelerometer and Gyroscope raw data are used as input $(u(k))$ to the EKF and not as observation like the remaining sensors.

Observations Model

The EKF solves this problem by linearizing the observations model. Starting with the observations equation [30]:

$$\mathbf{z_k} = \mathbf{h}\left[\mathbf{x}(t_k)\right] + \mathbf{v}_k \tag{4.13}$$

where $\mathbf{z}_k$ is the sensor measurement vector, $\mathbf{v}_k$ the observations noise and $\mathbf{h}\left[\mathbf{x}(t_k)\right]$ is the relation between states and the observations. Some examples are given below.

The GPS position, barometer height and GPS velocity are direct observations from states, so the observation mode is trivial. The equations from this section relative to the PX4 observations model, came from the PX4 estimation control library document [31].

It is assumed the magnetometer to be aligned with the body frame and experiences a magnetic field vector which is the sum from the Navigation Inertial Frame rotted into Body Frame and a Body Frame bias:

$$\begin{bmatrix} M_X \\ M_Y \\ M_Z \end{bmatrix}_{meas} = [T]_B^N . \begin{bmatrix} M_N \\ M_E \\ M_D \end{bmatrix} + \begin{bmatrix} M_X \\ M_Y \\ M_Z \end{bmatrix}_{bias} \tag{4.14}$$

It is assumed the sensor measures the magnitude of velocity relative to the wind field by the airspeed observation equation:

$$\begin{bmatrix} V_{rel_N} \\ V_{rel_E} \\ V_{rel_D} \end{bmatrix} = \begin{bmatrix} V_N \\ V_E \\ V_D \end{bmatrix} - \begin{bmatrix} V_{wind_N} \\ V_{wind_E} \\ 0 \end{bmatrix} \tag{4.15}$$

$$TAS_{meas} = \sqrt{(V_{rel_N}^2 + V_{rel_E}^2 + V_{rel_D}^2)} \tag{4.16}$$

In order to linearize the observations equations, the Jacobian matrix of $h(x)$ is obtained. This matrix is called the observation matrix, $\mathbf{H}$, and is given by[30]:

$$\mathbf{H}_k = \left[\frac{\partial \mathbf{h}(\mathbf{x})}{\partial \mathbf{x}}\right]_{\mathbf{x}=\mathbf{x}_{\mathbf{k+1}}^-}, \tag{4.17}$$

The last observation model parameter is the observation noise covariance matrix, $\mathbf{R}_k$, which is a

$n \times n$ diagonal matrix given by [30]:

$$\mathbf{R}_k = \begin{bmatrix} \sigma_1^2 & & \mathbf{0} \\ & \sigma_2^2 & \\ & & \ddots \\ \mathbf{0} & & \sigma_n^2 \end{bmatrix} \tag{4.18}$$

Having obtained all the parameters necessary to the computation of the EKF, the 24 states can be estimated.

4.1.2 Sensor Fusion

It was noticed that PX4 can change the source of observations while running the EKF. Giving one example, the barometer is the primary source for height determination but the GPS can be used as height observation instead, figure 4.6.

In case of a sensor failure, it is replaced or corrected by another sensor source. This process is called sensor fusion and it is not related with the sensors modules which replaces sensor values with other from the same typology. Also the fault detection criteria is different from the sensors module. The observation measurements are evaluated by their innovations or residuals, the difference between the sensor measurement (Observation) and its equivalent calculated through the predicted estimated states, $\mathbf{z}_k - \hat{\mathbf{z}}_k = \mathbf{z}_k - \mathbf{h}(\hat{\mathbf{x}}_k^-)$. For a better understanding, a flowchart was designed (see figure 4.6) relative to the height source.

There are four different possible sources to get the Height observation - Barometer, GPS, Range Finder, External Vision. One of them, is the primary height source, which means that sensor publishing data is currently being fused on the EKF or its data has currently been used as observation and the correspondent innovation and innovation variance are calculated every iteration. The system checks the innovation and innovation variance, using some parameters defined and tested by the PX4 developers relative to each sensor in the calculations to conclude if the co-variance matrix is badly conditioned (filter fault, figure 4.8) through the innovation variance or if the observations will be rejected (sensor fault) through the innovations (if the sensor innovation is higher than the innovation test limit, figure 4.9).

If any of this two checks fails, the PX4 warns the user the height source will be changed and starts fusing other sensor measurements on the EKF. The fusing order is already defined at the start and it depends on the available sources. Although it is possible to define the fused sensor before or during the flight through setting specific parameters on the PX4 console as well as to know which sensor is being fused on the control mode flags from the *estimator_status* topic.

Other optional sensors not represented in the observation model can be used like the optical flow.

Doing the innovation test limit for the magnetometer x component, to exemplify:

$$mag\_test\_ratio_x = \sqrt{\frac{mag\_innov_x}{\sqrt{mag\_innov\_gate\_parameter} * mag\_innov\_var_x}} \tag{4.19}$$

where the parameter was defined by the PX4 developers, the innovation (the sensor measurement from the matrix z_k subtracted by the estimated value from the matrix $\hat{z}_k = h(\hat{x}_k^-)$) and innovation variance (from the matrix $H_k P_k^- H_k^T + R_k$) are computed using the EKF algorithm, figure 4.5. If the $mag\_test\_ratio_x > 1$ the observation will be rejected and the EKF will stop the magnetometer fusion.

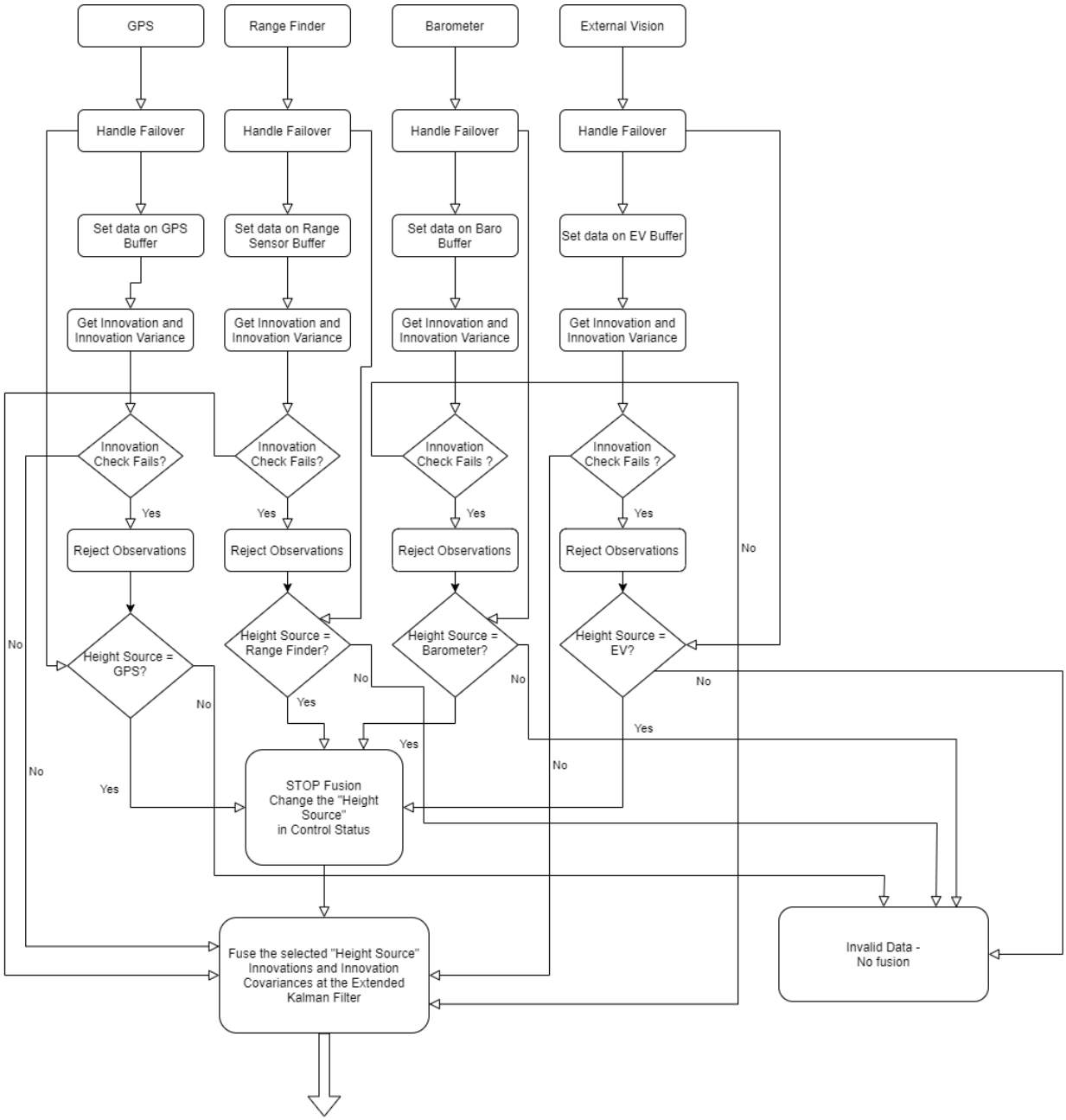


Figure 4.6: Sensor fusion example with the height source.

Different time delays on each measurement relative to the IMU are caused by different sensors. A complementary filter is needed because the EKF runs on a 'delayed fusion time horizon'. An overview of the estimation control diagram block with the complementary filter for states propagation forward from the 'fusion time horizon' to current time using the buffered IMU data is at figure 4.7. Data from each

sensor is FIFO buffered so the EKF can retrieve it from the buffer and use it at the correct time.

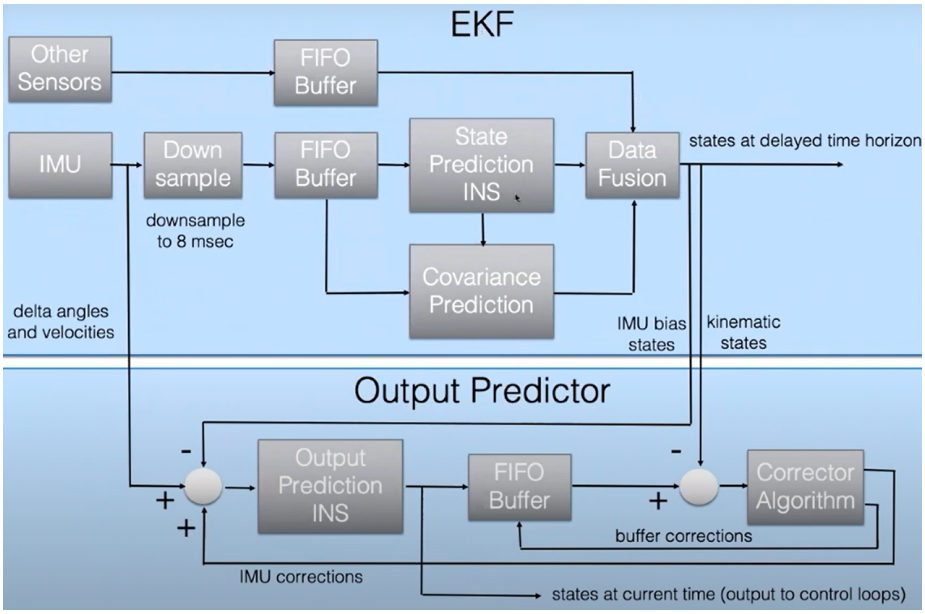


Figure 4.7: State estimator with the correction algorithm.[20]

The filter computation with the corrector algorithm will not be focused on this project since there is no need of adding more redundancy levels on the sensors level. Therefore the first approach was left.

4.2 Second Approach

Since the purpose is still to increase the fault detection subsystem $R_i(t)$, the strategy changed to work on an external ring immediately after the sensors, jumping to the filter states used by the control unit.

Instead of comparing the sensor data from the three autopilots, the point is to compare the 24 states from each autopilot correspondent EKF, adding a monitoring point after the filter. Synchronization is still a problem when comparisons are made because of the possible wrong decisions based on the states data correspondent to different timestamps. Also the processing capability needed from the PX4 stack is a problem, to keep sending and receiving the 24 states for each iteration and for each autopilot.

A solution was found inside the EKF2 module (EKF folder is just a library) from PX4. The system uses the innovations $(z_k - \hat{z}_k)$ and innovation variances $(H_k P_k^- H_k^T + R_k))$ to evaluate the filter own health. The *estimator_status* topic has flags groups that indicate observation (sensor measurements) faults, figure 4.9, or numerical errors which came from filter faults, figure 4.8.

```
uint32 filter_fault_flags       # Bitmask to indicate EKF internal faults
# 0 - true if the fusion of the magnetometer X-axis has encountered a numerical error
# 1 - true if the fusion of the magnetometer Y-axis has encountered a numerical error
# 2 - true if the fusion of the magnetometer Z-axis has encountered a numerical error
# 3 - true if the fusion of the magnetic heading has encountered a numerical error
# 4 - true if the fusion of the magnetic declination has encountered a numerical error
# 5 - true if fusion of the airspeed has encountered a numerical error
# 6 - true if fusion of the synthetic sideslip constraint has encountered a numerical error
# 7 - true if fusion of the optical flow X axis has encountered a numerical error
# 8 - true if fusion of the optical flow Y axis has encountered a numerical error
# 9 - true if fusion of the North velocity has encountered a numerical error
# 10 - true if fusion of the East velocity has encountered a numerical error
# 11 - true if fusion of the Down velocity has encountered a numerical error
# 12 - true if fusion of the North position has encountered a numerical error
# 13 - true if fusion of the East position has encountered a numerical error
# 14 - true if fusion of the Down position has encountered a numerical error
# 15 - true if bad delta velocity bias estimates have been detected
# 16 - true if bad vertical accelerometer data has been detected
# 17 - true if delta velocity data contains clipping (asymmetric railing)
```

Figure 4.8: Filter internal fault flags resulted from the innovation variance, *estimator_status* topic [32]

```
float32 pos_horiz_accuracy # 1-Sigma estimated horizontal position accuracy relative to the estimators origin (m)
float32 pos_vert_accuracy # 1-Sigma estimated vertical position accuracy relative to the estimators origin (m)
uint16 innovation_check_flags # Bitmask to indicate pass/fail status of innovation consistency checks
# 0 - true if velocity observations have been rejected
# 1 - true if horizontal position observations have been rejected
# 2 - true if true if vertical position observations have been rejected
# 3 - true if the X magnetometer observation has been rejected
# 4 - true if the Y magnetometer observation has been rejected
# 5 - true if the Z magnetometer observation has been rejected
# 6 - true if the yaw observation has been rejected
# 7 - true if the airspeed observation has been rejected
# 8 - true if the synthetic sideslip observation has been rejected
# 9 - true if the height above ground observation has been rejected
# 10 - true if the X optical flow observation has been rejected
# 11 - true if the Y optical flow observation has been rejected
```

Figure 4.9: Rejecting observation flags resulted from the innovation test limits, *estimator_status* topic [32]

The combination from this two flags groups gives origin to a third group of flags which contains a bitmask indicating which filter kinematic state outputs are valid for flight control use (*solution_status_flags*). The values from the ratios of the largest sensor innovation component to the innovation test limit (the innovation check) are represented too in this topic to understand the observation error dimension, figure 4.10. The GPS has its own bitmask to indicate specific faults (minimal required satellite count fail, maximum allowed PDOP fail, ...) since the PX4 navigation system depends a lot from it.

If trustful data referring to the estimator health already exists, based on different sensor measurements comparisons with the estimated data from the EKF, there is no need of an heavy computational operation for states comparison between autopilots to achieve similar conclusions. Each PX4 can watch

```
float32 mag_test_ratio # ratio of the largest magnetometer innovation component to the innovation test limit
float32 vel_test_ratio # ratio of the largest velocity innovation component to the innovation test limit
float32 pos_test_ratio # ratio of the largest horizontal position innovation component to the innovation test limit
float32 hgt_test_ratio # ratio of the vertical position innovation to the innovation test limit
float32 tas_test_ratio # ratio of the true airspeed innovation to the innovation test limit
float32 hagl_test_ratio # ratio of the height above ground innovation to the innovation test limit
float32 beta_test_ratio # ratio of the synthetic sideslip innovation to the innovation test limit
```

Figure 4.10: Ratio of the largest sensor innovation component to the innovation test limit, *estimator_status* topic [32]

out his own estimator flags and share the information with the other two autopilots, between defined time periods, in order to catch permanent faults and not punctual faults caused by a single bad sensor measurement reflected in the innovations.

The PX4 keeps running the default mechanisms from 1 autopilot explained on the previous section 4.1, including the sensors module which uses redundancy to vote on the sensor with the highest priority (applied specially to the Inertial Measurement Unit sensors - accelerometers and gyroscopes) followed by the EKF. The sensor fusion selects the best source of the same sensor typology throughout the innovations analysis from observations.

As EKF input, the IMU are the most relevant sensors, so special attention is given to the accelerometer and gyroscope. With less than 3 gyroscopes, it's impossible to determine which is the failing gyro if one fails because there is not a tiebreaker. The same applies to the accelerometer. Therefore the autopilot with just 2 healthy IMU must change to prevent the worst case scenario.

Also the tilt and yaw alignment flags must be true to consider the EKF healthy. Those flags belong to the the topic where the innovation flags can be found.

So the designed system uses the *estimator_status* topic to get the filter Bitmask flags correspondent to the innovation checks and EKF internal faults to decide if the estimated states are good or bad. If the estimated results are bad and do not change during a time period, it is decided to decrease the autopilot priority and to request for another autopilot to take over (in case if the autopilot being on charge).

For a while this algorithm had been the best choice however the discovery of the Multi-EKF operation mode forced a turn back on the main ideas.

4.3 Third Approach

The most recent fault detection method from PX4 uses 7 (by default) EKF running on parallel. Each EKF is attached to one accelerometer, one gyroscope and one magnetometer and it cannot change any of these sensors but the same sensor can be attached to more than one EKF, figure 4.11.

The Multi-EKF allows soft failures detection beyond the hard faults which were detected until now by the sensors modules. The EKF selector module judges the EKF health through the innovation test ratios and the Bitmask flags correspondent to the filter internal faults (numerical errors).

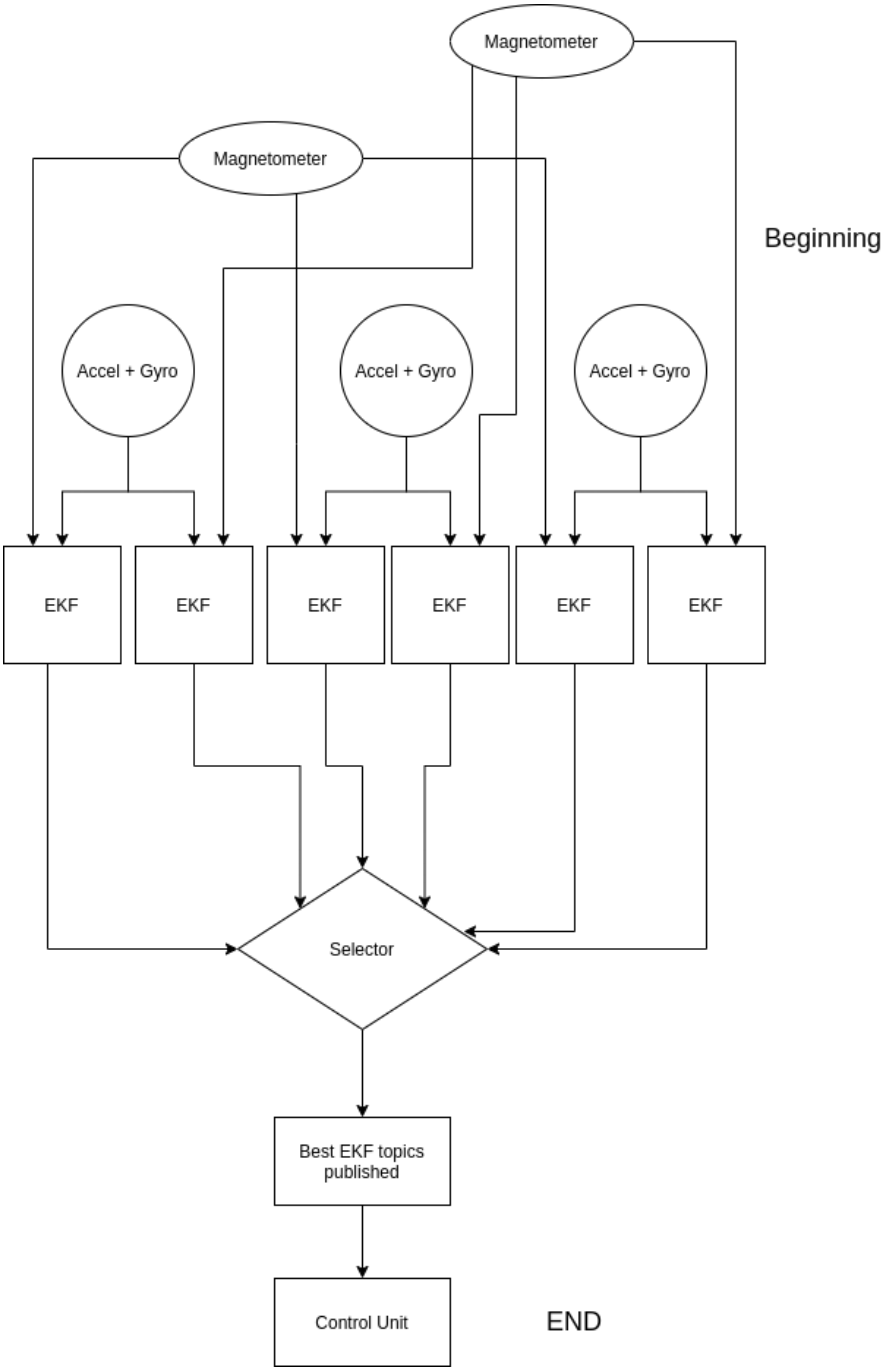


Figure 4.11: PX4 Multi-EKF mode flowchart.

The chosen EKF is the one classified as ”healthy” with the best (the smallest) $combined\_test\_ratio$ which corresponds to:

$$combined\_test\_ratio = \max(0.5 * (vel\_test\_ratio + pos\_test\_ratio), hgt\_test\_ratio) \tag{4.20}$$

where vel_test_ratio, pos_test_ratio and hgt_test_ratio belong to the innovation tests ratios done by each EKF like it was exemplified with the magnetometer in equation 4.23.

Ratio of the largest velocity innovation ”component” to the innovation test limit:

$$vel\_test\_ratio = \sqrt{\frac{vel\_innov}{\sqrt{vel\_innov\_gate\_parameter} * vel\_innov\_var}} \tag{4.21}$$

Ratio of the largest horizontal position innovation component to the innovation test limit:

$$pos\_test\_ratio = \sqrt{\frac{pos\_innov}{\sqrt{pos\_innov\_gate\_parameter} * pos\_innov\_var}} \tag{4.22}$$

Ratio of the vertical position innovation to the innovation test limit:

$$hgt\_test\_ratio = \sqrt{\frac{hgt\_innov}{\sqrt{hgt\_innov\_gate\_parameter} * hgt\_innov\_var}} \tag{4.23}$$

To prevent unnecessary selection changes between EKF instances, the $relative\_test\_ratio$ is calculated through the difference between the actual selected EKF instance $combined\_test\_ratio\_s$ and the other EKF instance $combined\_test\_ratio\_i$. If the difference module overtakes a threshold, the parameter *EKF2_SEL_ERR_RED*, the change occurs:

$$|relative\_test\_ratio_i| = |combined\_test\_ratio\_s - combined\_test\_ratio\_i| > \textit{EKF2\_SEL\_ERR\_RED} \tag{4.24}$$

The fault tree for an ”unhealthy” EKF classification was designed, figure 4.12. Any of these five events leads to an ”unhealthy” judgement of the EKF:

- Filter internal faults
- Faulty gyroscope or faulty accelerometer
- $combined\_test\_ratio < 0$
- EKF publishing topics timeouts

The Multi-EFK mode does a similar process from which was planned on the last approach algorithm, section 4.2. It evaluates the estimator to select the best one. The main difference resides the estimators are running on the same autopilot instead of running on different boards.

One possible approach would be to evaluate just the selected EKF flags from each autopilot. Although there is another module that does already the task.

The Commander Module relates the chosen/best EKF published *estimator status* topic data with time periods to conclude if the autopilot has detected an hard failure and if it has to trigger the failsafe mode. It verifies if the innovation test ratios for the velocity and position components are higher than 1. In that case the Commander Module considers the navigation is failing. If the navigation keeps failing for more than 2 seconds, the commander module changes the state to ”navigation failure”. That decision will change three flags in the ”vehicle_status” topic published by this module: *condition_local_position_valid*, *condition_local_velocity_valid* and *condition_global_position_valid*, see figure 4.13. A fault tree using the example from the global and local position flag is represented in the figure 4.15.

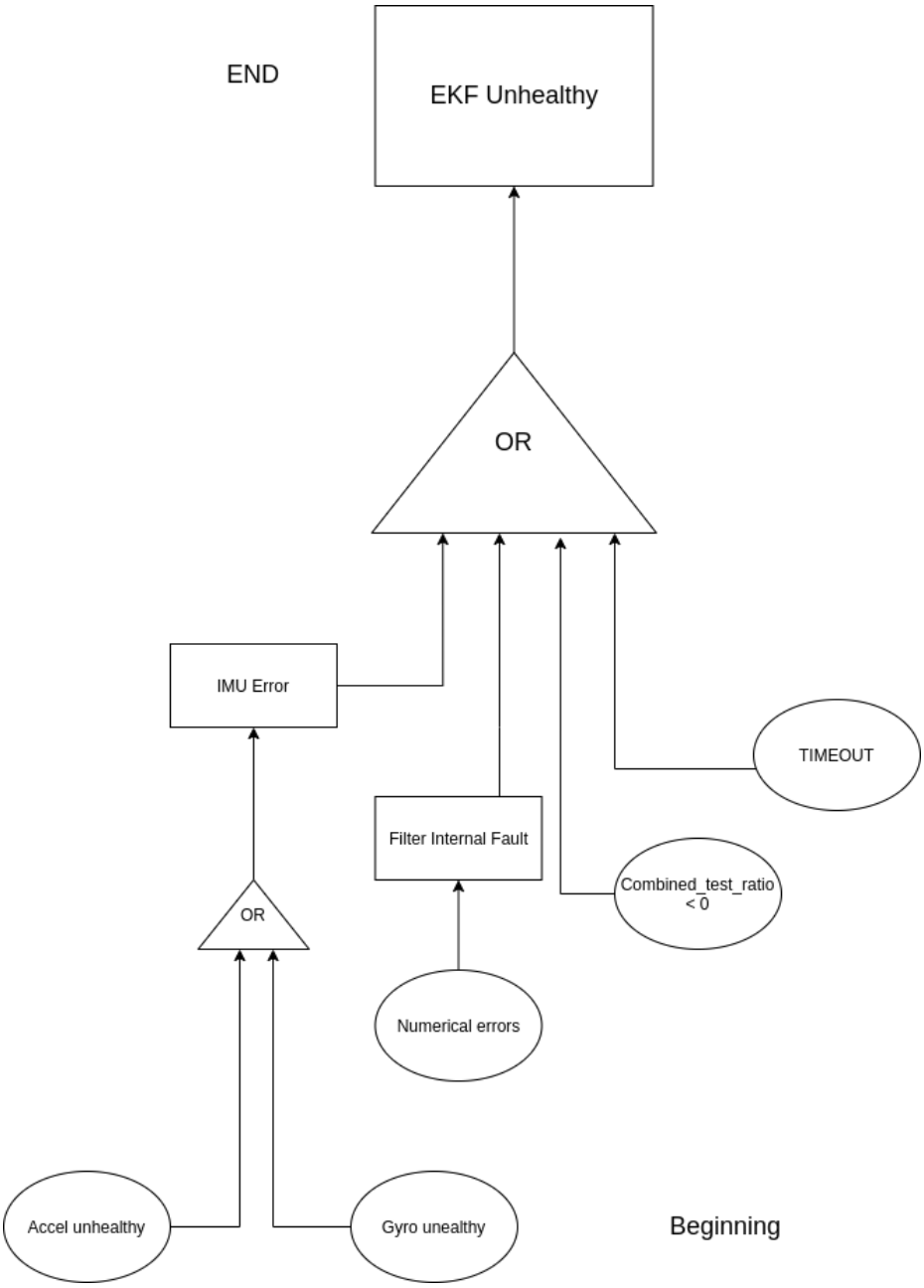


Figure 4.12: EKF Fault tree used by the EKF Selector Module.

```
bool condition_local_position_valid          # set to true by the commander app if the quality of the local position estimate is good enough to use for navigation
bool condition_local_velocity_valid          # set to true by the commander app if the quality of the local horizontal velocity data is good enough to use for navi
bool condition_global_position_valid         # set to true by the commander app if the quality of the global position estimate is good enough to use for navigation
```

Figure 4.13: Global position, local position and local velocity validity checks flags. [32]

Those flags are not exclusively dependent from the ”navigation failure” decision. They are set to be ”false” if the dead reckoning time is exceeded, if standard deviations errors from the correspondent estimates exceeds the limit imposed by the PX4 parameters (figure 4.14) or if the messages published by the selected EKF lasts longer than 1 second (timeout error).

Also all the estimated states values are checked to be finite, looking for software faults.

```
float32 eph                     # Standard deviation of horizontal position error, (metres)
float32 epv                     # Standard deviation of vertical position error, (metres)
float32 evh                     # Standard deviation of horizontal velocity error, (metres/sec)
float32 evv                     # Standard deviation of horizontal velocity error, (metres/sec)
```

Figure 4.14: Standard deviation errors. [32]

The standard deviation error of the horizontal and vertical local position (meters) is calculated by:

$$eph = \sqrt{P_k^-(7,7) + P_k^-(8,8)} \tag{4.25}$$

$$epv = \sqrt{P^-k(9,9)} \tag{4.26}$$

where P_k^- is the error covariance estimate matrix from the EKF algorithm.

The standard deviation error of the horizontal and vertical global position (meters) is calculated by:

$$eph\_g = \sqrt{P_k^-(7,7) + P_k^-(8,8) + gps\_origin\_eph^2} \tag{4.27}$$

$$epv\_g = \sqrt{P_k^-(9,9) + gps\_origin\_epv^2} \tag{4.28}$$

The standard deviation error of the horizontal and vertical velocity meters per second) is calculated by:

$$evh = \sqrt{P_k^-(4,4) + P_k^-(5,5)} \tag{4.29}$$

$$evv = \sqrt{P_k^-(6,6)} \tag{4.30}$$

These values are compared with the required accuracy from the PX4 defined parameter.

The fault tree relative to the *global* or *local position validation* variable is demonstrated in figure 4.15, as an example for a better understanding of the PX4 coded functions and to relate the estimator faults with possible sensor failures from any kind.

More three Boolean flags are determined by the Commander Module: *condition_angular_velocity_valid*, *condition_attitude_valid* and *condition_local_altitude_valid*. The angular velocity, the attitude and the local altitude are verified due to their importance to the navigation. The Commander Module checks if those variables are updated (less than 1 second from the last publish) as well as if their values are finite.

So the 6 error flags were considered relevant for fault detection about one PX4 unit:

- *condition_local_position_valid*
- *condition_local_velocity_valid*
- *condition_global_position_valid*
- *condition_angular_velocity_valid*

- *condition_attitude_valid*
- *condition_local_altitude_valid*

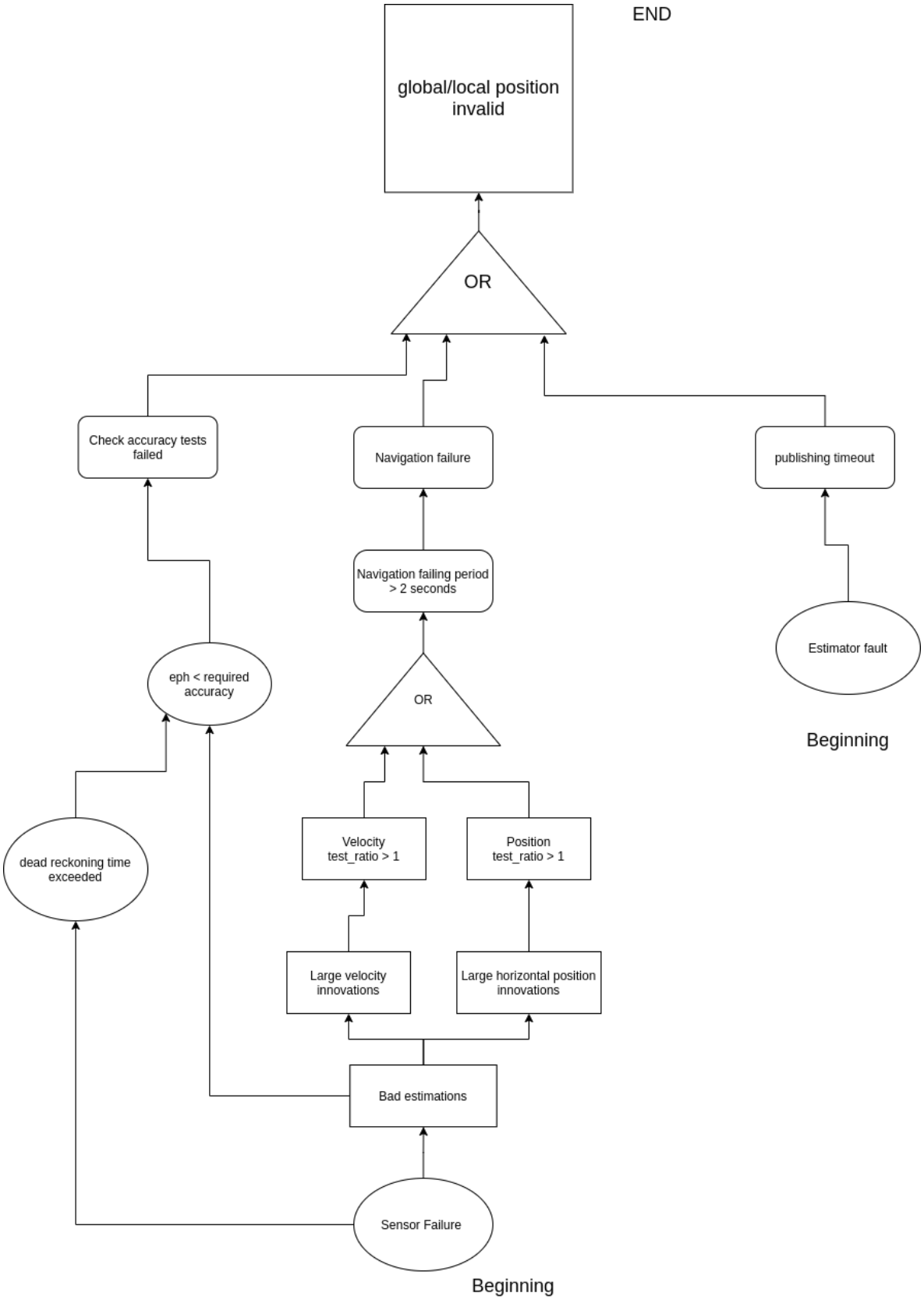


Figure 4.15: Global or Local position validation fault tree used by the Commander module.

The commander module publishes other important flags not related with the estimator. They are suitable to evaluate crucial subsystems for the autopilot operations:

- *battery_healthy* (set to true if the battery is available - not low)
- *condition_system_sensors_initialized* (check if the sensors are initialized)
- *failsafe* (true if system is in failsafe state (e.g.:RTL, Hover, Terminate, ...) - a security mechanism which is triggered when an hard failure is detected
- *data_link_lost* (datalink to GCS lost)

The triple redundant system provides the possibility to detect and mitigate faults by comparing the system or subsystem states. When two of the three systems agree about its state, the third one which disagrees is probably failing. Because the chances of two systems taking the wrong decision at the same time are much lower. This logic will guide the system fault detection.

Therefore the mentioned six flags relative to the estimator/sensor health plus the other 4 flags, relative to other autopilot subsystems, were chosen to be monitored and compared with the respective flags from the others redundant autopilots. As an example, the comparison logic between different PX4 instances is demonstrated with the *battery_healthy* flag, on the diagram 4.16.

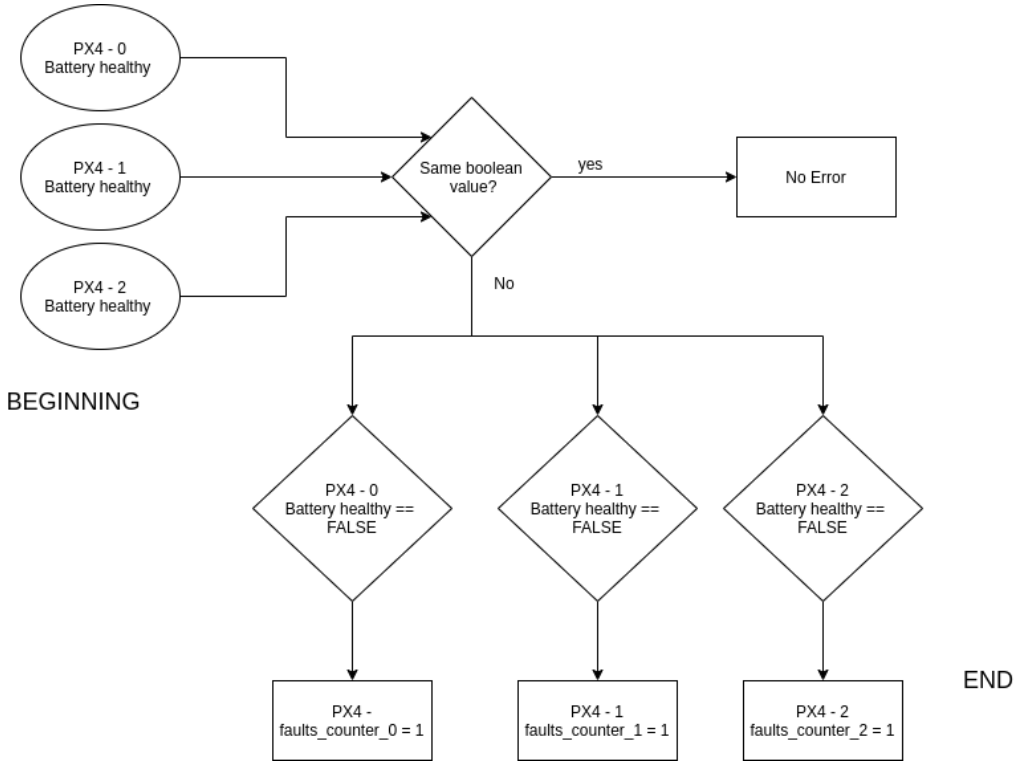


Figure 4.16: Comparison diagram of the redundant Boolean battery healthy flag and failure counter for each PX4 instance.

Error/failing counters are defined by the sum of all faults from one PX4 unit. All the faults have the same weight with exception to the communication faults between the autopilots that will be covered next.

The *battery healthy* used in this example, figure 4.16, is the first variable to be compared so the failing/error counter from each instance is equalized to one instead of being incremented when the

correspondent instance *battery_healthy* state is 'FALSE'. When the three PX4 instances flag have the same Boolean value, the system is considered to be healthy even if the flag indicates a faulty subsystem ('FALSE' state). In that case, it is considered an external problem to the autopilot and out of range for fault tolerance.

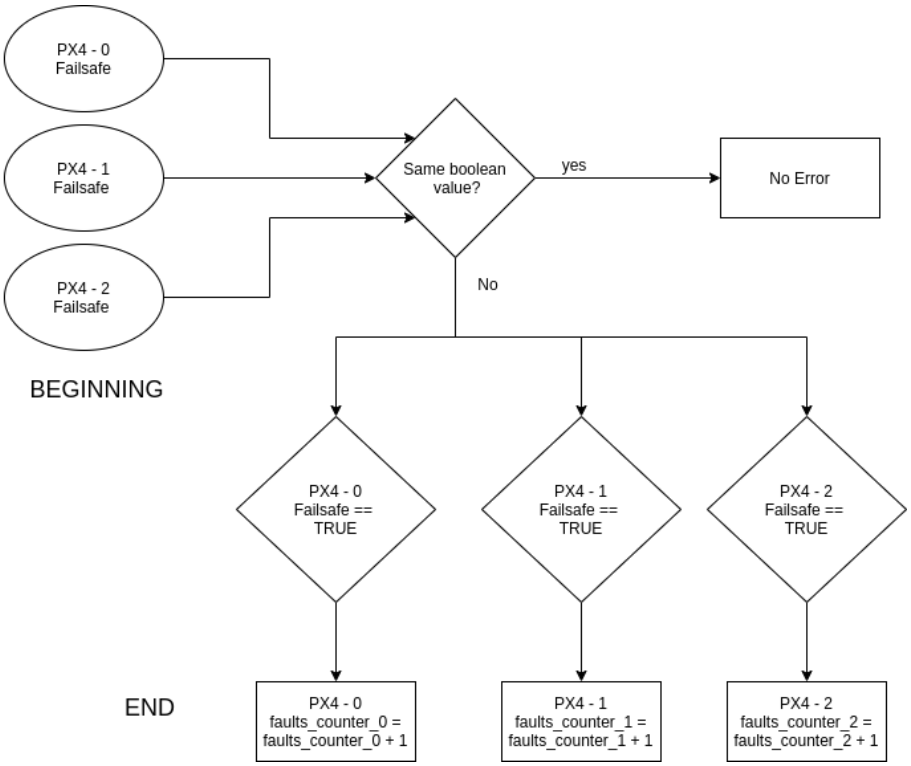


Figure 4.17: Comparison diagram of the redundant Boolean failsafe flag and failure counter for each PX4 instance.

Another diagram, figure 4.17, refers to the failsafe variable comparison between autopilots. At this time a 'FALSE' Boolean flag state means there is no error. The failsafe is triggered when an hard failure is detected by the PX4 itself. Like any error flags, when the fault is detected, the respective error counter from the autopilot is increased.

The comparisons are done inside each PX4, since it is a decentralized and distributed system. An extra Module was added to the Firmware code, the *redundancy_manager.cpp*. Also the voting process is decentralized, each PX4 has autonomy and computes its own vote on the best autopilot based on the data shared between them. All the autopilots vote on the respective autopilot with the lowest error/faults counter computed by themselves based on the information they received about the 10 error flags states from the other 2 PX4. The error counters values are reflected on the autopilot priority (another created variable) unless the communications between the PX4 stand as the error origin. The autopilot which does not communicate for more than 3 seconds, has its priority immediately changed and becomes the last choice regardless the other two PX4 error counters values.

The autopilot priorities values are reflected on the voted autopilot by each instance.

The first autopilot (PX4 instance 0) computation algorithm and the external selection using the other

two autopilots decisions can be visualized on diagram 4.18.

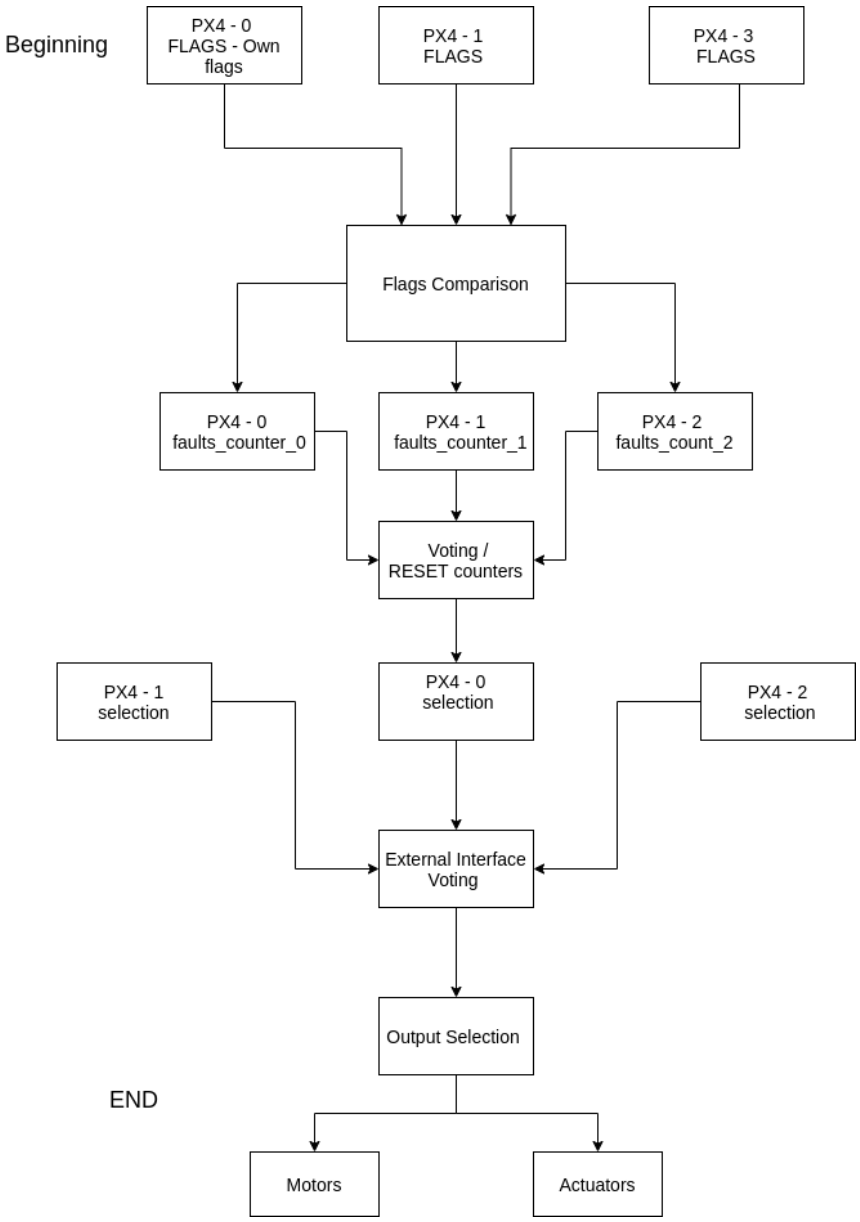


Figure 4.18: Decentralized voting algorithm for the first autopilot.

The autopilots instances are numbered from 0 to 2:

- First autopilot - Instance 0
- Second autopilot - Instance 1
- Third autopilot - Instance 2

When the autopilots computed priorities are the same (equal error counters values), each autopilot votes on the autopilot numbered with the smallest value.

The voted autopilot will be the selected to take over. In case of all the autopilots having one vote, which is not a good sign, the first autopilot or the instance 0 takes over the UAV control.

The triple redundant algorithm computation is just allowed to work 30 seconds after take off to prevent sudden jumps at flight critical phases.

The third approach was the chosen one to be validated since it benefits from the existent fault tolerant systems which are similar with the two first approaches systems. This approach is based on the commander module which works as the upper layer and last fault tolerant system from the estimation control library. In the next chapter 5 all the computations will be clear and justified from this programmed algorithm.

Chapter 5

Experimental Results

In this chapter, the system implementation and experimental results are described. All the programs were specifically designed for the Software-In-The-Loop (SITL) simulation. SITL consists of running the three flight stacks and executing other compiled source code belonging to the fault tolerant system on computer using a modeled vehicle in a simulated world (mathematical model simulation). After the experimental setup, the results are presented and analyzed. Gazebo was the simulator software used on this project while the GCS software to interact with the PX4 was the QgroundControl, see figure 5.1.

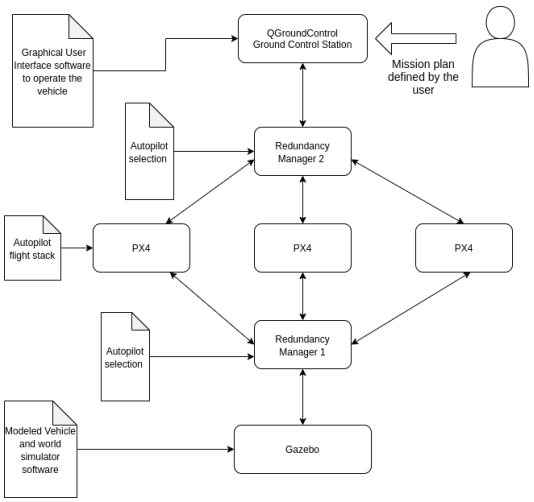


Figure 5.1: Description of the redundancy system software running on SITL.

5.1 Problem Description

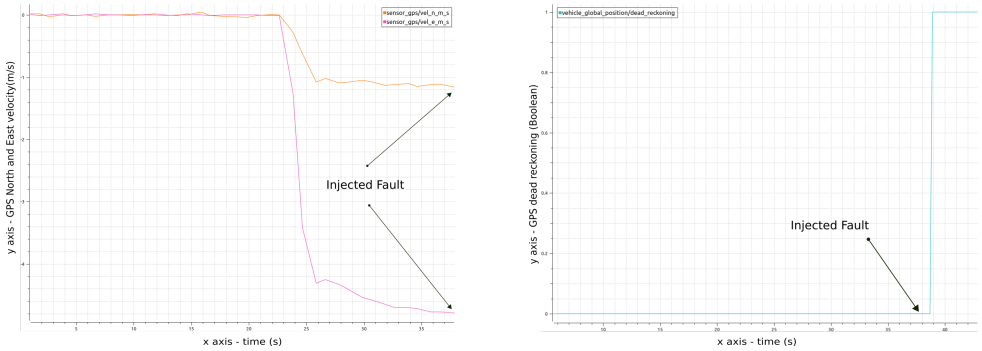
The main goal of this thesis is to build a fault tolerant system for an autopilot. The system was designed based on redundancy and using internal variables from the autopilot to increase the fault detection reliability and so the entire system reliability too.

To this end, sensor faults were artificially injected in the autopilot for time periods through the System Failure Injection module, on the PX4 console. Those faults consequences are observed through logged

internal topics as well as the system reaction and fault mitigation in order to validate the system.

The example from the figure 5.2(a) shows when the GPS signal stopped to publish data. It occurred 37 seconds after the simulation start. The time window is automatically adjusted to the published values, so the graph ends when the last North and East velocities values were published. It is confirmed by the triggered dead reckoning flag 2 seconds later, see figure 5.2(b).

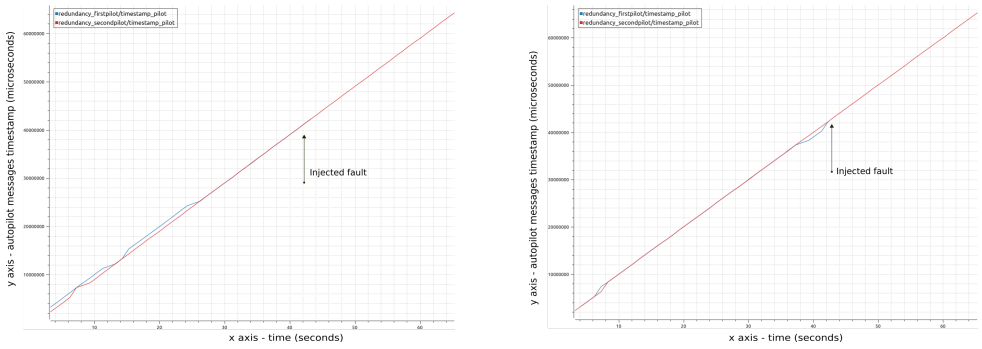
To simulate the communication failure between autopilots, the instance 0 stopped sending messages to the other two instances. On both figures 5.3(a) and 5.3(b), the blue line is not published anymore after the 42 seconds. It represents the time of the last message received from the instance 0.



(a) PX4 instance 0, GPS North and East velocity.

(b) PX4 instance 0, GPS dead reckoning.

Figure 5.2: GPS signal turned off using the failure injection command.



(a) PX4 instance 1, received messages timestamp from other autopilots.

(b) PX4 instance 2, received messages timestamp from other autopilots.

Figure 5.3: Broken communications from the PX4 instance 0.

Stuck magnetometer values are represented in figure 5.21(a) when the fault occurred (2 minutes and 27 seconds after the simulation start). This case was studied in the section 5.3.2.

5.2 Experimental Setup

For this thesis were implemented three different programs to allow the autopilot selection. Beyond the module added to the PX4 firmware, two interface software were designed to manage the communications between the autopilot, simulator and ground control station. Both interface software forward data

just from the selected PX4 instance, accordingly to the voting system, to prevent communications conflicts on the simulator and GCS ports. In order to prevent bugs and problems which are not matter of interest of this project context, the complexity factor of running multiple PX4 SITL instances cannot be exposed to the Gazebo and to the QGC. So the redundancy manager programs were designed to be the the only one communication contact point from both GCS software and simulator and to make the three PX4 instances to be invisible to them both. Also it is desired the user experience to be transparent (one vehicle - one autopilot) although the developer knows that three autopilot instances are running.

5.2.1 Software-In-The-Loop (SITL)

In order to validate the fault tolerant algorithm, three PX4 SITL stacks ran independently at the same machine. The SITL architecture diagram for one flight stack is below, figure 5.4.

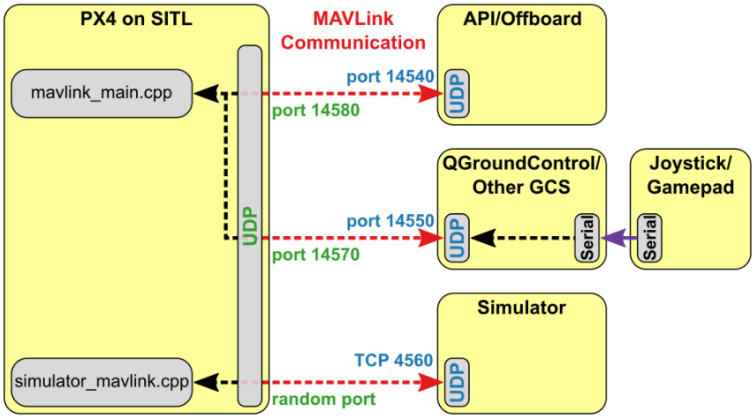


Figure 5.4: Software In The Loop simulation. [20]

The simulator communicates with PX4 using the simulator MAVLink API and it supplies sensor data to the PX4 SITL instance while this one returns actuators and motor values to be applied to the simulated vehicle though the API referred. The API creates the sockets and codifies the messages to MAVLink. The offboard API is not used. The ports were changed at the PX4 SITL side to connect to the interfaces programs instead of direct connection to the Simulator and the Ground Control Station.

5.2.2 Simulator

The decision fell on Gazebo because of its support, Robot Operating System and multi-vehicle compatibility . The Gazebo communication process with PX4 was carefully studied to select easily the PX4 instance outputs through ROS before becoming simulator inputs. Unfortunately the gazebo plugins designed to interface with PX4 were not build with ROS libraries so it is impossible to redirect them through this tool. Gazebo can also be used for multi-vehicle simulations which could be interesting to run multiple flight stacks on the same model multiplied (twins) when injecting faults to analyse the ”same” UAV behaviour response to the different stimulus. However just one model instance was ran on the Gazebo

to test the voting system and selection process robustness to mitigate the faults and capability to keep the same model running with a different flight stack after fault detection. Jmavsim could also be an option, although a major experience and knowledge about Gazebo became a factor to continue with this simulator. It communicates by TCP/IP socket protocol. The simulation ran on the different machine, remote server, to improve the PX4 SITL performance since it consumes a lot of processing from the same computer when all the three autopilots are running at the same time. The simulated UAV model chosen is a simple quadcopter named ”iris”.

5.2.3 Gazebo Redundancy Manager

Before implementing the redundancy manager, it must be understood how the communication between the PX4 SITL and the simulator is established, figure 5.5. This is based on TCP/IP protocol where PX4 SITL is the client and the simulator behaves like server.

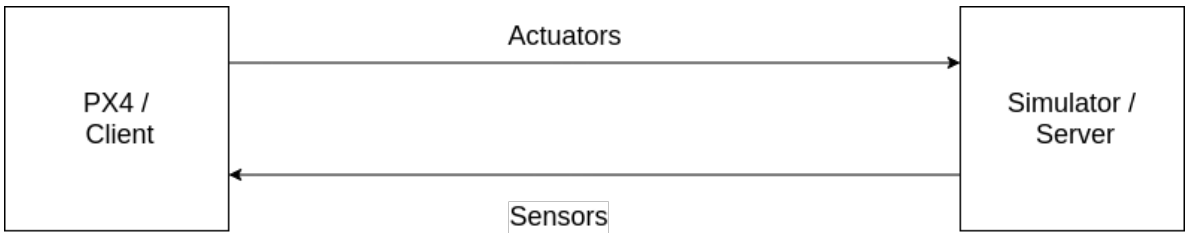


Figure 5.5: Communication protocol between PX4 and Simulator.

The simulator redundancy manager must behave like a server listening for connections while communicating with PX4 SITL. By the other side, it must behaves like a client to connect to the simulator. So it was designed to poll messages continuously from both sides without interrupting the cycles using multi threading techniques in C language.

It is intended to run three PX4 SITL at the same time where just one of them sends the computed actuators and motor values to the simulated vehicle. The program must select the outputs accordingly to the voting system and forward to the simulator as well as it must forward also the sensor values from the simulated world to all the three PX4 SITL. Once again, multi threading the program allowed to create more two servers listening on different ports (referred at the diagram 5.6).

Despite of not being represented at the diagram 5.6, there are three more socket connections between the PX4 and the interface software (one more for each PX4 SILT). Those communication channels refer to the vote result computed in each autopilot. The selection program counts the votes and selects the most voted to forward its messages to the simulator.

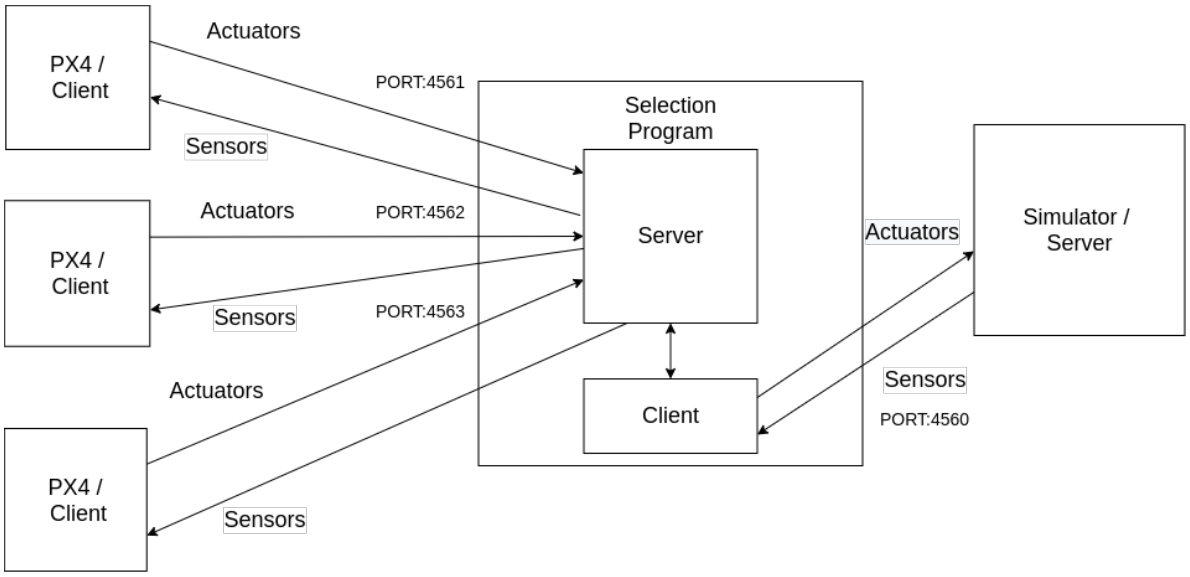


Figure 5.6: Communication protocol between PX4, Simulator Redundancy Manager and Simulator.

5.2.4 QgroundControl

The GCS software mostly indicated to work with PX4 is the QgroundControl. It communicates by UDP/IP protocol. There is no offboard control, the mission is settled on the QgroundControl. This software was used to command the vehicle (where missions were defined) and to monitor the vehicle throughout telemetry. A mission plan taken from the QGC window is represented in the section 5.2.6, in figure 5.8.

All the PX4 SITL connect to the GCS sending their own heartbeats Mavlink messages to the same port, 14550. However the QGC answers to each PX4 SITL local port.

5.2.5 QgroundControl Redundancy Manager

It is intended the QGC to send the same mission or commands to the three PX4 instances and just to receive telemetry from one of them, accordingly to the redundant voting system.

An similar process, figure 5.6, was done to create the redundancy manager program. Now the papers are inverted, each PX4 SITL behaves like server and the QGC like client. So the Redundancy manager has to behave like a client to communicate with the PX4 instances or like a server if the communication refers to the QGC. In the UDP communication protocol, the local ports are defined by default (14580, 14581,14582) from the PX4 SITL side to allow the Redundancy manager (or the QGC when not running the redundant system) to answer to the autopilots, diagram 5.7. In multi-vehicle simulations, all the PX4 SITL instances send messages to the QGC through the same port (14550). That port was divided on three different ports (145555,1556, 14557) to prevent congestion in a single port when polling messages. This program was written in C language with multi threaded cycles to poll messages from the QGC and the 3 autopilots continuously. More three TCP connections were added to the QGC Redundancy Manager, not represented at the diagram 5.7, to receive the computed votes from each autopilot instance.

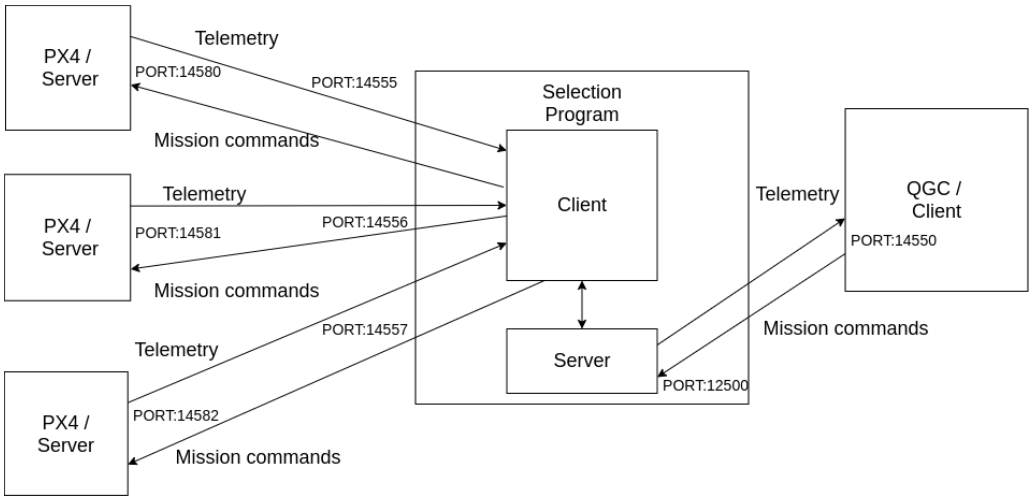


Figure 5.7: Communication protocol between PX4, QGC Redundancy Manager and QGC.

5.2.6 Mission plan

The mission takes place in the PX4 default location (Zurich, Switzerland). There is no need to change the location since all the simulated sensors values, including GPS, are not real and are provided by the Gazebo while running on SITL or HITL simulations. It was chosen a typical survey mission where the UAV, after the takeoff, follows the Waypoints number order to up and down throughout vertical lines on the map. The waypoints form a circular pattern, so the vehicle covers the respective area, figure 5.8. As it will be seen on the section 5.3, most of the vehicle simulations did not perform the full mission plan since it was considered enough data was taken to validate the redundant system operation.

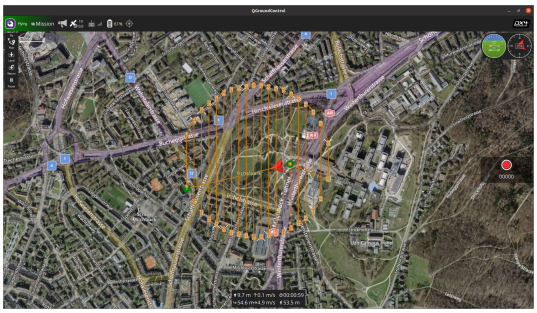


Figure 5.8: Mission plan view from QgroundControl.

5.3 Results

In order to demonstrate the fault tolerant redundant system working properly, artificial faults must be injected on the PX4 which is on charge. By default, the instance 0 takes over the vehicle control when all the instances remain at the same healthy condition. The instance 1 takes over if the instance 0 is classified unhealthy when compared with the other two instances available. This means the instance 3 just takes over the control when both first two instances are classified by a worst condition than the last

instance.

Faults will not be introduced by the Gazebo plugins since the three autopilots are connected to the same simulated vehicle. So the faults would reach the three autopilots and it could not be possible to differentiate any of them. The failure command, available on the PX4 console, was useful to simulate GPS, barometer and magnetometer faults since it introduces them on the correspondent PX4 instance sensor messages. The failure injection is defined inside the PX4 own simulator module which means its action resides just after the data being received from Gazebo allowing to differentiate the failing instance.

For the flight log analysis were chosen two different tools:

- The Flight Review [33] for the general topics and its graphs visualization. It is not possible to visualize different topics from the default.
- The Plotjuggler [34] for specific topics inspection like the added messages from the redundancy algorithm or other messages containing flags considered important for this project.

5.3.1 GPS Failure

A GPS failure was injected 27 seconds after the takeoff as it can be read on the PX4 console or in the recorded messages in the log from the PX4 console, figures 5.9 and 5.10. At the appendix A, is shown the dead reckoning start around the 37 seconds on the instance 0, see figure A.4(a). Also in the section 5.1, this example was given to demonstrate the moment when the GPS signal was turned off, figures 5.2(a) and 5.2(b).

Since the tests were done in a simulated vehicle and there was no risk of loosing the UAV on a crash, the timestamp for voting and selection between different autopilots was reduced to 10 seconds after the critical flight phases, like the takeoff. It is intended to do not waste time during the simulations to get the data to validate the system. In real flights that value becomes equal to 30 seconds to increase the vehicle security.

75	0:00:09	INFO	[redundancy_manager] sent message to the first socket
76	0:00:10	INFO	[commander] Armed by external command
77	0:00:10	INFO	[navigator] Takeoff to 10.0 meters above home.

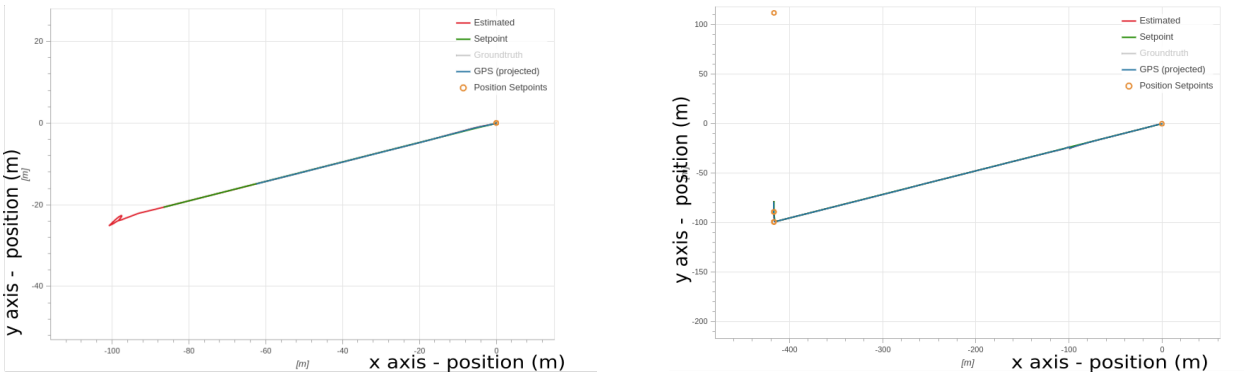
Figure 5.9: Takeoff command on the log recorded messages from PX4 console.

215	0:00:37	WARNING	[failure] inject failure unit: gps (4), type: off (1), instance: 0
216	0:00:37	WARNING	[simulator] CMD_INJECT_FAILURE, GPS off
217	0:00:38	INFO	[redundancy_manager] Published first pilot
218	0:00:38	INFO	[redundancy_manager] sent message to the first socket

Figure 5.10: GPS failure injection command on the log recorded messages from PX4 console.

The PX4 trajectories represented in figure 5.11(a) and 5.11(b) were designed throughout the provided data (GPS, setpoints/waypoints, estimated position) from the PX4 instances. The graphs belonging to the third PX4 autopilot, correspondent to the instance 2, are not shown below because they are similar to the second autopilot, instance 1.

When comparing the computed trajectories it is noticed the instance 0 did not accomplish the mission continued by the other two instances. After the takeoff both instances towards to the same first position setpoint despite of instance 0 not achieving it. The GPS projected trajectory is coincident with the estimated and setpoint trajectory on the instance 1 and 2, figure 5.11(b). While the instance 0, figure 5.11(a), stopped the GPS navigation (blue line) because of the blocked GPS sensor messages.



(a) PX4 instance 0, waypoints were not reached.

(b) PX4 instance 1 and 2, waypoints reached after takeoff.

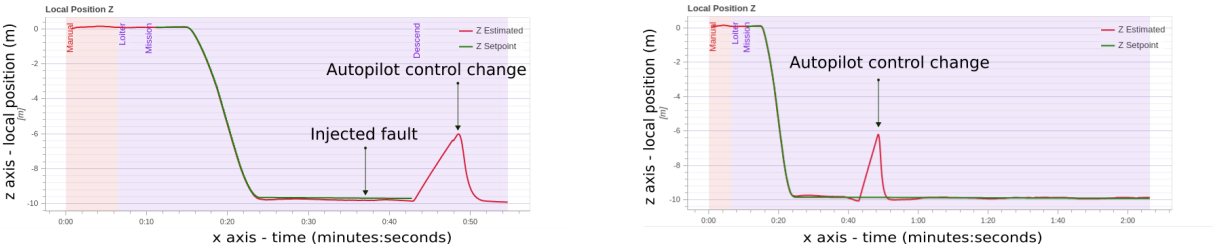
Figure 5.11: Two dimensions UAV mission trajectories from the top view

The three autopilot instances refers to the same simulated vehicle in space time, so the quadcopter travelled the same true trajectory 5.11(b) regardless the PX4 instance. The reason of the false trajectory from instance 0, figure 5.11(a), is because of the PX4 low capability to estimate the position seconds after the GPS being turned off, specially when the direction needed to change as well as the forced navigation stop.

The figure 5.12(a) shows the UAV position along the Z axis during the simulation. The ground level is stated by 0 value and the Z axis points to down which means the relative altitude to the ground is negative. The UAV climbs to 10 meters after the takeoff and stays at the same altitude until it receives the land command from the navigator module. Despite of not receiving new commands to abort the mission, the vehicle started the descend maneuver to land 42 seconds after the simulation start. While the instance 0 has been taking over the vehicle, the failsafe mechanism was triggered after the GPS fault injection. The autopilot took 5 seconds to consider the fault as an hard failure. The redundant system algorithm is based on the commander module flags. Some of them, like the global and local position valid flags or the failsafe flag, take the same time to change as the PX4 to be declared faulty. Therefore the redundant algorithm caught the first errors when the autopilot started to descend and they were compared with the other two instances. For the autopilot to be declared faulty by the algorithm, the same flags should remain faulty more than 1 second when compared again. The purpose is to gain consistency on the decision about the running PX4 instances and to avoid punctual errors which could lead to bad evaluations about the autopilots and provoke constant changes between autopilots during a flight. It is a trade off between the decision consistency and the reaction time to a real failure.

Before the 50 seconds, the instance 1 takes over and the vehicle starts to climb again to the 10 meters to complete the mission following to the next set point, figure 5.12(b). The difference between

the setpoint and the estimated Z position is clear when the UAV descends.

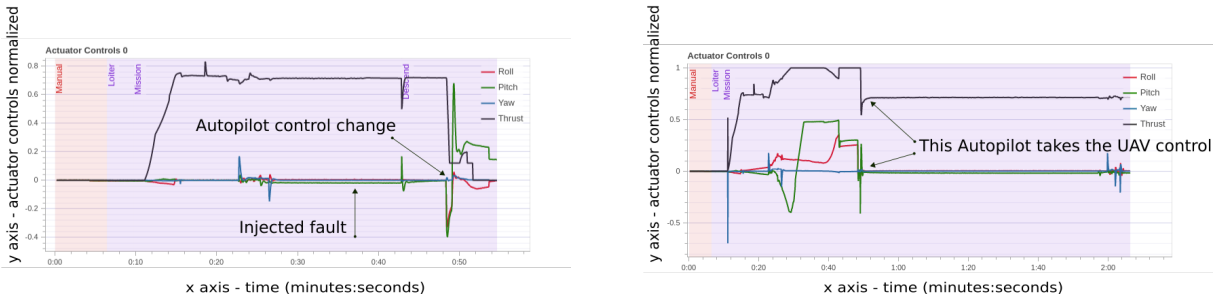


(a) PX4 instance 0.

(b) PX4 instance 1 and 2.

Figure 5.12: PX4 set and estimated position projected on the Z axis (negative altitude relative to the ground) during the flight in the Inertial Frame from the different instances

On the figures 5.13(a) and 5.13(b) are represented the actuator controls values from both PX4 instances during the SITL simulation. The values from the PX4 instance 2 actuator controls were ignored because it does not take over the UAV control in this simulation. The instance 1 actuator controls begin with instability while the instance 0 is the only autopilot controlling the UAV, so the other instances are receiving the feedback from the other autopilot control. It is not a problem since their actuators values are being ignored. When the instance 0 initiates the landing phase, the other two instances show opposite values to the actuators since they are not aware of the triggered failsafe and they will do an effort to keep the UAV on the previous route to reach the next waypoint. Fifty seconds after the simulation start, the actuator controls from the instance 1 stabilize since it has taken over the UAV and it receives now the correct feed backs from the vehicle answers as well as it turns back to the right position to accomplish the mission.



(a) PX4 instance 0.

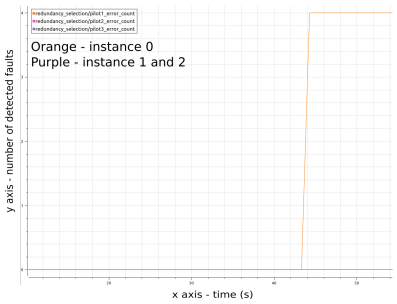
(b) PX4 instance 1.

Figure 5.13: PX4 actuator controls (Roll, Pitch, Yaw and Thrust) from the different instances during the simulation time.

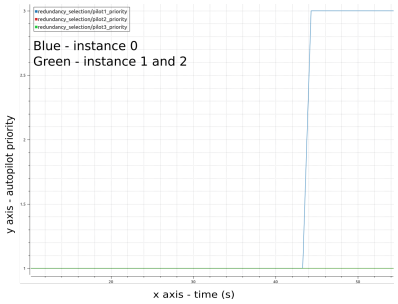
The redundancy manager module publishes a topic, *redundancy_selection*, where the voting selection can be followed any time by the user. The message topic is recorded on the logger as well as its selection criteria.

The error provoked by the GPS fault had been detected around the 43 seconds on the instance 0 where the fault was injected, figure 5.14(a). Three seconds later it had been detected on the other

two instances (figures 5.15(a) and 5.16(a)). All the autopilots error counters are represented on the graphs which proofs the decentralized capability at any autopilot. Each autopilot number line refers to the autopilot instance plus 1 because the autopilots were numbered from 1 to 3 instead of 0 to 2. The line of the autopilot 2 (instance 1) is not visible because of the autopilot 3 (instance 2) line, which has exactly the same form and it superimposes the last one.

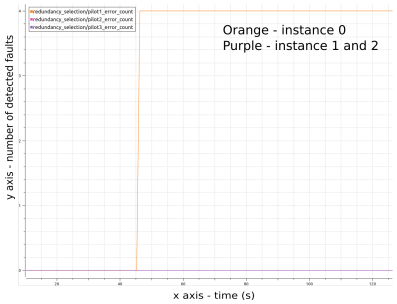


(a) Error counters from the three autopilots computed by the instance 0.

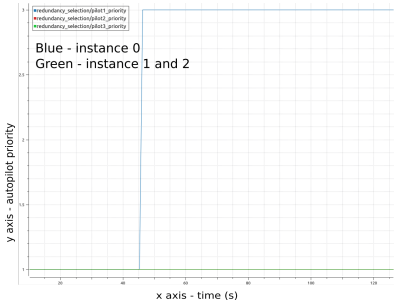


(b) The three autopilots priorities computed by the instance 0.

Figure 5.14: PX4 instance 0 computed variables by the redundancy algorithm during the simulation.

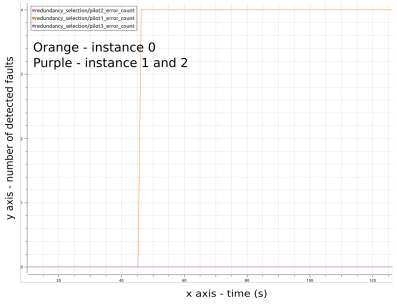


(a) Error counters from the three autopilots computed by the instance 1.

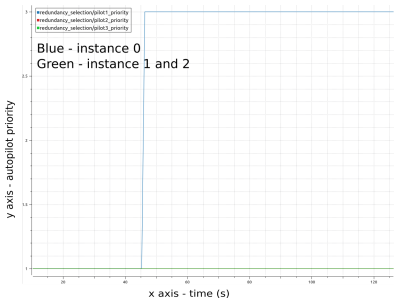


(b) The three autopilots priorities computed by the instance 1.

Figure 5.15: PX4 instance 1 computed variables by the redundancy algorithm during the simulation.



(a) Error counters from the three autopilots



(b) The three autopilots priorities

Figure 5.16: PX4 instance 2 computed variables by the redundancy algorithm during the simulation.

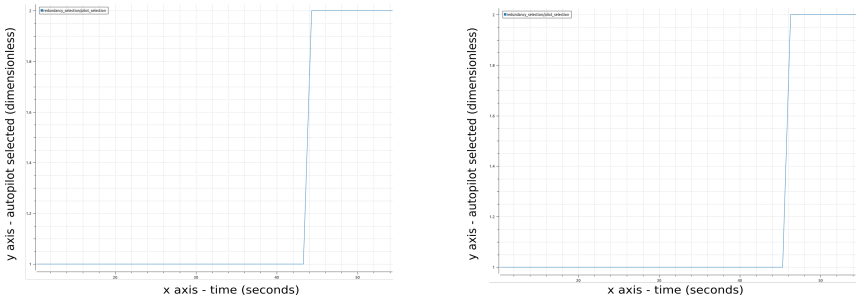
The redundancy algorithm belonging to the respective autopilot refreshes the autopilots priorities which is based on the error counters, figures: 5.14(b), 5.15(b) and 5.16(b). However the voting selection

is made following a descending order, so the autopilot selected has the lowest priority. The default priority is 1. The priority varies until 3 if the error counter from the correspondent autopilot increases when compared with the other two autopilots. When the autopilots have the same error counter value, the same priority number is given to both or to all (in case of the three having the same error counter value). In the case of having the same priority, the tiebreaker criteria for the selection is the instance/autopilot number. The autopilot with a lower instance number, is the chosen one. That is the reason why the instance 0 always starts the flight.

Since the autopilot priorities computation is based on the respective error counters and it is executed immediately after computing the error counters, they changed both at the same time. When the first autopilot error counter increased to 4, its priority increased to 3.

The voting selection from the instance 0 is shown in the figure 5.17(a). Again the instance 0 or autopilot 1, is the first to change the voted autopilot, around 43 seconds accordingly with the computed priority from the correspondent instance. The other two instances changed the voted autopilot at the same time, 45 seconds, so the instance 2 is omitted because the graph is equal to the instance 1, figure 5.17(b). From the figure 5.12(b) it can be concluded the changing between the autopilots is not instantaneous, because of the heavy computational multiple threads occurring at the same time on the interface selection programs. The autopilot 2 just took over the control 48 seconds after the simulation start.

The redundancy manager interface programs are running on the background decoding and coding every MAVLink message exchanged between the simulator, the three PX4 instances and the Qground-Control so the voting thread took 3 seconds to process the votes from all the instances to change the autopilot which is sending the actuator values to be read by the UAV on the simulator.



(a) PX4 instance 0 voter. (b) PX4 instance 1/2 voters.

Figure 5.17: Autopilot voted by the redundancy algorithm during the simulation.

The error flags whose sum defines the error counters if they are constant for a time period, are shown at the console in real time when they occur or on the logged messages part from the log file, figure 5.18.

However the error flags counters cannot reflect sometimes the error flags shown at the console because of its consistency. Regardless the error vanishment from the console, if it had been constant for more than one second, it will be taken in account for the error counter during at least 600 seconds. On the contrary, when the faults are detected at the first time, they are not immediately reflected on the error counters or priorities since they have to keep the same error more than one second. It is just the

case of instance 0, figure 5.18. The failsafe was triggered just in the instance 0, so the fault was noticed and printed on the console. The same happened for the condition_local_position error flag despite of not having its values reflected on the error counters and the autopilots priorities.

The autopilot priority_1 vector refers to the computed autopilot priorities. This vector is ordered differently from the default way (autopilot_priority) and it is relative to each autopilot. It should not be used for comparisons between autopilots. The autopilot_priority is ordered on the same default way to all the instances and it can be compared with the other autopilots same vector.

The flags used to compute the error flags from each instance can be accessed through the *vehicle_status.msg* and the *vehicle_status_flags.msg* topics as mentioned in 4.3.

246	0:00:43	INFO	[redundancy_manager] Pilot[0] failsafe error
247	0:00:43	INFO	[redundancy_manager] Testing error counters.counter_pilotfailing[0]: 0, counter_pilotfailing[1]: 0, counter_pilotfailing
248	0:00:43	INFO	[redundancy_manager] Testing priorities. Pilot_priority_1[0]: 1 . Pilot_priority_1[1]: 1. Pilot_priority_1[2]: 1.
249	0:00:43	INFO	[redundancy_manager] Testing priorities. Pilot_priority[0]: 1 . Pilot_priority[1]: 1. Pilot_priority[2]: 1.
250	0:00:44	INFO	[redundancy_manager] Pilot[0] condition_local_position error

Figure 5.18: Console messages display with error flags from instance 0 or autopilot 1.

Since the fault was injected it took 6 seconds to be detected by the first autopilot and 8 seconds by both second and third autopilots. Globally it took 11 seconds for the second autopilot to take over the vehicle after the fault injection.

The relation between the estimations quality with the GPS failure is analyzed through the innovation test ratios graphs in the appendix A.1. Also the standard deviation errors and the dead reckoning time graphs demonstrate to be the cause for triggering some error flags.

5.3.2 Magnetometer Failure

Two stuck magnetometers connected to the first autopilot were simulated, two minutes and 27 seconds since the simulation start, throughout the failure injection command leaving the PX4 instance 0 without any working magnetometer available.

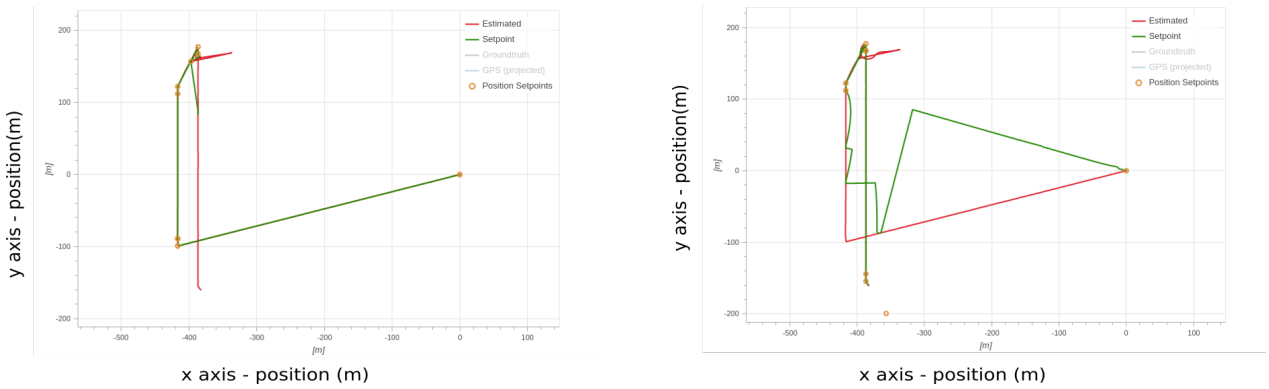
0:02:27	INFO	[redundancy_manager] Testing priorities. Pilot_priority[0]: 1 . Pilot_priority[1]: 1. Pilot_priority[2]: 1.
0:02:27	WARNING	[failure] inject failure unit: mag (2), type: stuck (2), instance: 0
0:02:27	WARNING	[simulator] CMD_INJECT_FAILURE, mag stuck
0:02:28	INFO	[redundancy_manager] PX4 sending selection pilot: 1 to the simulator

Figure 5.19: Magnetometer failure injection command on the log recorded messages from PX4 console at 2 minutes and 27 seconds.

In the figure 5.20(a) is drawn the trajectory made by the vehicle from the point of view of the first autopilot or instance 0. The vehicle must follow the green line, which defines the mission connected set points on the correct order. The answer of the vehicle to the setpoints is represented by the red line which represents the estimated vehicle trajectory. The green line is superimposed to the red line in the first part of the trajectory. Then the second autopilot took over because of the fault detection. Since the

first autopilot triggered the failsafe, the mission was aborted and the green line vanished at that moment. Although the second autopilot or instance 1, has continued the mission so the red line which estimates the vehicle position is correct. The estimated and mission trajectory from that autopilot is drawn in the figure 5.20(b). The third autopilot or instance 2 is omitted because of the lack of space, the similar behaviour with the instance 1 and the point of not taking over the vehicle control during the flight time.

There is a great difference between the estimated trajectory and the mission trajectory on the first part of the trajectory. The first setpoint chosen to start the mission corresponds to the second setpoint in the first autopilot, ignoring the first one, by unknown reasons since the same mission was sent to the three autopilots from the QGC. Apart from that issue, the vehicle estimated trajectory differs from its mission trajectory when the vehicle reaches the 180 metes along the y axis. That is because the first autopilot is still on charge and the UAV started the landing movement determined by the failsafe mechanism. Meanwhile the autopilot 2 has taken over the vehicle control and it recovers the position as well as the mission setpoints which make the red line being superimposed by the green line.



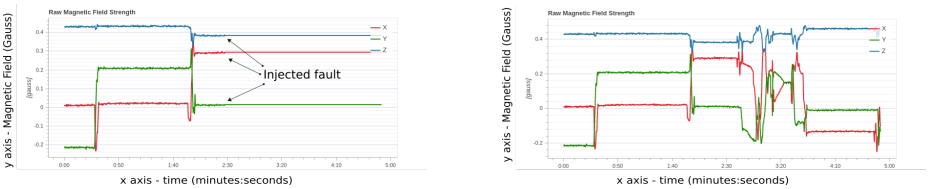
(a) PX4 instance 0.

(b) PX4 instance 1.

Figure 5.20: 2D trajectory in the inertial frame of reference (top view).

The magnetic field measured by both instances is shown below providing a visualisation of the fault injected at the magnetometers. In figure 5.21(a) is shown the raw magnetic field strength from the instance 0 along the three axis with different colours. Around the 30 seconds the UAV took off and some variations were noticed on the magnetic field. Even after being stabilized, the measurements always have some noise. At 2 minutes and 27 seconds after the simulation start, the raw magnetic field strength values had become stuck like it was commanded on the PX4 console, figure 5.19.

By the other side, on the instance 1 (figure 5.21(b)) some perturbations on the magnetic field are noticed between the third and fourth minute when the vehicle is still confused because of the fault injected and the consequent decision to land caused by the failsafe mechanism. When the control change between autopilots is made, the magnetic field and the vehicle stabilize again. The magnetic field strength measured values from the instance 1 are not stuck, since they keep showing noise like before the fault injection.

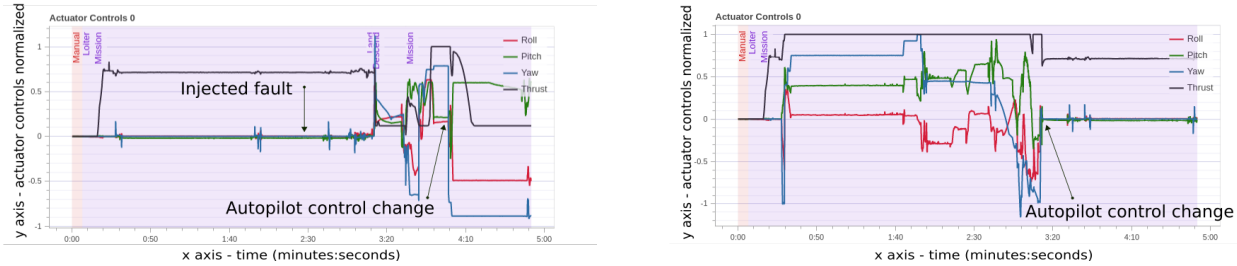


(a) PX4 instance 0.

(b) PX4 instance 1. X axis - time in minutes:seconds.

Figure 5.21: Magnetic field strength measured by the PX4 magnetometers during the simulation.

The figures below were chosen to demonstrate the control change between autopilots during the flight using the actuator controls. The difference between the pitch, yaw and thrust actuator control values from instance 0 (figure 5.22(a)) to the instance 1 (figure 5.22(b)) until the third minute are explained by a different first Waypoint on the uploaded mission by the QGC, nothing related with faults. The landing maneuver is visible on the first autopilot in figure 5.22(a) at 3 minutes and 10 seconds when the thrust is decreased. The landing phase also confuses the second autopilot, figure 5.22(b), which tries to contradict the maneuver not commanded by itself. After taking over the vehicle control, the thrust returns to same initial value from the instance 0 as well as the roll, pitch and yaw which turns 0. Now it is the instance 0 turn to try contradicting the maneuver commanded from the instance 1.



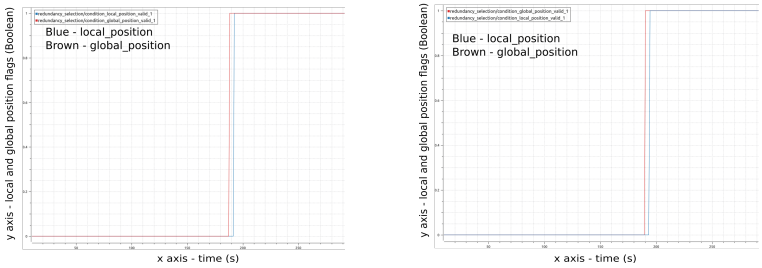
(a) PX4 instance 0.

(b) PX4 instance 1.

Figure 5.22: PX4 actuator controls (Roll, Pitch, Yaw and Thrust) from the different instances during the simulation time.

For this simulation the error flags used on the computed selection were recorded. Just the four flags which changed over time are shown below. The flags were recorded from the first autopilot (see figures 5.23(a) and 5.24(a)) and from the second autopilot (see figures 5.23(b) and 5.24(b)).

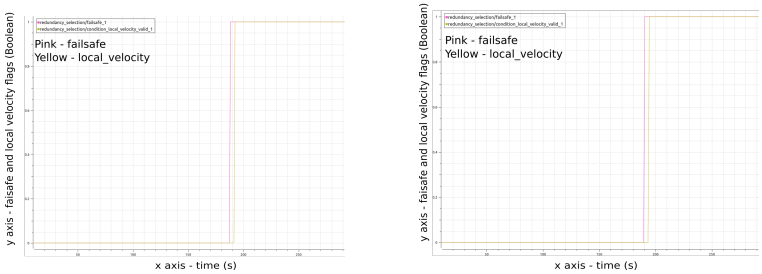
The instance 0 detects the global position error flag firstly followed by the local position, figure 5.23(a), around the 190 seconds. Both error flags are detected approximately 3 seconds earlier than the instance 1, figure 5.23(b). A similar behaviour is observed with the failsafe and the local velocity error flags. The instance 0 detects the failsafe flag at the same time as the global position error flag is detected, figure 5.24(a). Again the instance 1 detects those error flags, figure 5.24(b) approximately 3 seconds after of being detected on the instance 0.



(a) PX4 instance 0. (b) PX4 instance 1.

Figure 5.23: Local and global position error flags.

These error flags are relative to the first autopilot behaviour, although the results shown came from two different autopilots. The third autopilot results were omitted because they are similar to the second autopilot.



(a) PX4 instance 0. (b) PX4 instance 1.

Figure 5.24: Failsafe and local velocity error flags.

Some of the error flags relative to the second autopilot behaviour are described for comparison - figure 5.25(a) from the first autopilot point of view and figure 5.25(b) from the second autopilot point of view.

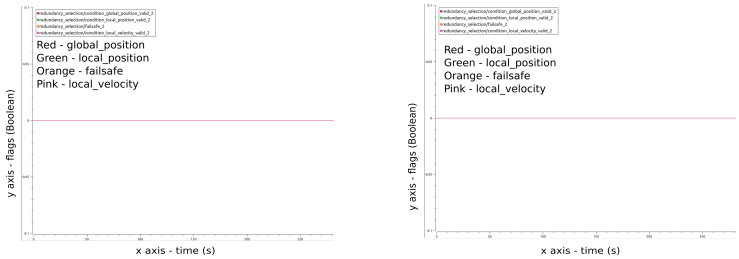
The error flags computed by the two instances do not change over time, figures 5.25(a) and 5.25(b), resulting in zero accountable errors for the second autopilot.

The error counters relative to three autopilots but computed on the different autopilots during the flight simulation are represented in figures 5.26(a) and 5.26(b). The third autopilot or instance 2 was omitted because of its similarity with the instance 1. So the instance 0 computed error counters, figure 5.26(a), shows two detected faults around the 187 seconds when the first two error flags changed. Three seconds later, the first autopilot error counters increases to 4 while the other two autopilots keep the error flags unchanged and the error counters at 0.

While the second and third autopilot keep the same priority level at 1, the first autopilot priority number increases to 3 (remember this system priority follows a descending order). The figure 5.27(a) refers to the instance 0 computed priorities relative to the three autopilots while the instance 1 computed priorities are represented in the figure 5.27(b).

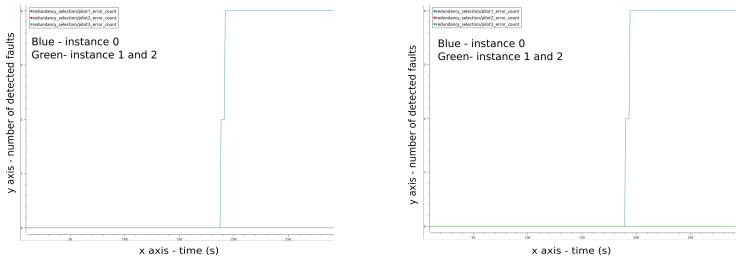
Around the 190 seconds the priority of the second autopilot is classified by 3 at the same time of the error flags changes detection. So the instance 1 changes the second autopilot priority 3 seconds after

the instance 0 like the error flags behaviour observed before in figures 5.23(b) and 5.24(b).



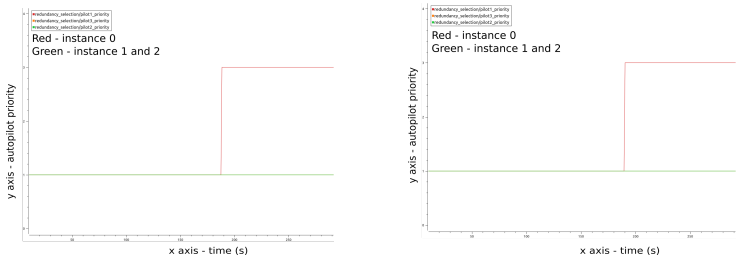
(a) PX4 instance 0 observation. (b) PX4 instance 1 observation.

Figure 5.25: Local velocity, local position, global position and failsafe error flags behaviour from the second autopilot observed by two different instances.



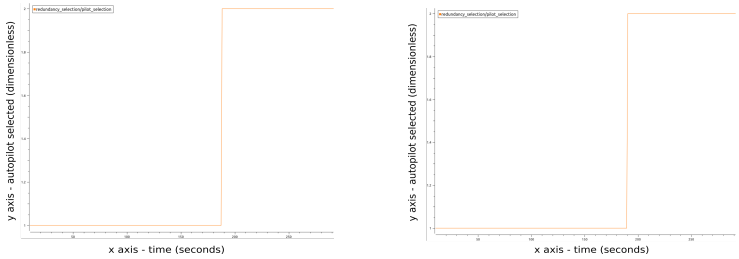
(a) PX4 instance 0. (b) PX4 instance 1.

Figure 5.26: Error counters from the three autopilots computed by the first two instances.



(a) PX4 instance 0 priorities. (b) PX4 instance 1 priorities.

Figure 5.27: Computed priorities over time from both autopilots (instance 0 and instance 1). Instance 3 omitted. Priority scale from 1 to 3.



(a) PX4 instance 0. (b) PX4 instance 1 and 2.

Figure 5.28: Autopilot voted by the redundancy algorithm during the simulation.

From the past results analysis about the autopilots priorities, error counters and the respective error flags, it is possible to guess the vote changing time from the first to the second autopilot. Since the voting selection is made immediately after the priorities computation, around the 190 seconds the vote changed to the second autopilot in the first instance, figure 5.28(a). Like the error counters and the error flags, the computed vote from the second autopilot was three seconds later approximately, figure 5.28(b). Therefore the vote selection accompanied the first change in the error counters which means when the first two error flags changed. The other two error flags triggered later did not influence the voting since the other two autopilots kept its error counters at 0.

Since the magnetometers faults were injected, it took approximately 40 seconds until the system declare it as a failure. However the UAV was not in a real danger all that time while waiting for other autopilot to take over. The PX4 maintained the UAV on the same route 40 seconds after the failure injection with no fault detection because of its estimators ability to work temporarily with just other sensor data sources.

So it starts to struggle because of the bad estimations provoked by the magnetometers faults and it took more 25 seconds until the error flags being triggered as well as the failsafe mechanism to initiate the landing. So the system catches the error flags values and it votes on another autopilot and approximately three seconds later the second autopilot takes over.

Finally the relation between the estimator health with the magnetometer failure is analyzed through the innovation test ratios graphs in the appendix A.2. Some of the error flags triggered caused by the Magnetometer failure are a direct consequence from great innovations during the flight.

5.3.3 PX4 Communication Failure

That scenario occurs when a autopilot turns off for some reason or when it becomes unable to communicate with the others. In the second case, despite of the autopilot being healthy at the present, it can avoid the system fault detection ability in the future because the autopilots rely on real time data to compute the priorities and to vote on the right autopilot.

To simulate a communication breakdown at the autopilot on charge, the first autopilot was programmed to stop sending messages to the other two autopilots 30 seconds after the take off.

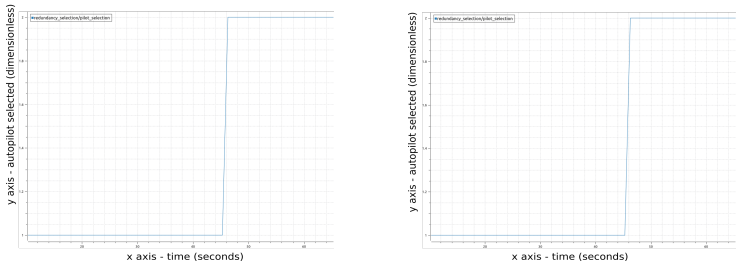
The UAV took off 11 seconds after the simulation start, figure 5.29. Therefore after 41 seconds from the simulation start, the second and third autopilot had not received any messages from the first one. The figures 5.3(a) and 5.3(b) in the section 5.1 refers to this fault. All the autopilots have a 3 seconds margin to wait for other autopilot messages which corresponds to three iterations because the shared messages between autopilots have a rate of 1 per second.

79	0:00:10	INFO	[redundancy_manager] Testing priorities. Pilot_priority[0]: 1 . Pilot_priority[1]: 1. Pilot_priority[2]: 1.
80	0:00:11	INFO	[commander] Armed by external command
81	0:00:11	INFO	[navigator] Takeoff to 10.0 meters above home.
82	0:00:11	INFO	[redundancy_manager] PX4 sending selection pilot: 1 to the simulator

Figure 5.29: Takeoff on the logged messages from PX4 console at 11 seconds.

The instance 1 (second autopilot, figure 5.30(a)) and the instance 2 (third autopilot, figure 5.30(b)), react to the communication failure almost at the same time, approximately at 45 seconds, changing the first autopilot priority and the voting autopilot. The graph relative to the instance 0 (first autopilot) voting selection was omitted because it stands the vote in the first autopilot all the simulation, since it keeps receiving the messages from the other two autopilots and does not know the other two autopilots are not receiving its messages.

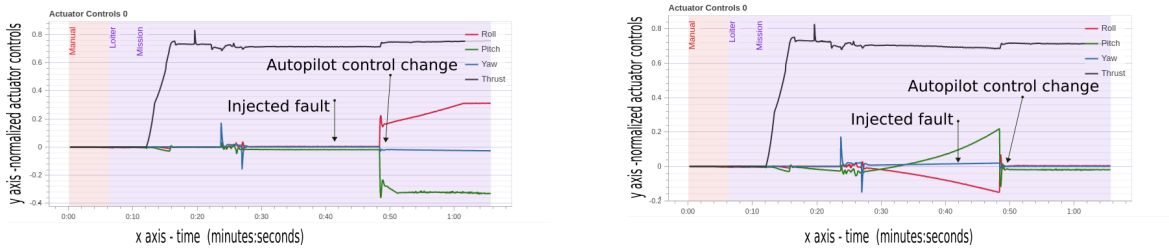
Through the actuator controls pitch and roll during the flight, it is clear the second autopilot took over the control from the UAV around the 48 seconds, see figures 5.31(a) and 5.31(b).



(a) PX4 instance 1 voter. (b) PX4 instance 2 voter.

Figure 5.30: Autopilot voted through the redundancy algorithm during the simulation.

From the 30 seconds up to 48 seconds, the first autopilot on charge has its actuator controls pitch and roll stabilized with low values, figure 5.31(a). However the second autopilot actuator controls pitch and roll are increasing because of a small divergence on the UAV control, figure 5.31(b), which obeys just to the first autopilot by now. When the UAV control is transferred to the second autopilot, at 48 seconds, the opposite situation occurs. The first autopilot actuator controls pitch and roll rise, figure 5.31(a), as well as the respective variables from the second autopilot stabilizes, figure 5.31(b). Now the UAV obey to the second autopilot.



(a) PX4 instance 0. (b) PX4 instance 1.

Figure 5.31: PX4 actuator controls (Roll, Pitch, Yaw and Thrust) from the different instances during the simulation time.

So it took approximately 3 seconds to the system to change the autopilot on charge since the decision was made inside each autopilot. The total time the system spent to answer to the communication failure was 7 seconds. In a real flight, it is not known if this failure refers just to the communications or if the entire autopilot turned off. Therefore the system reaction time to an hard failure from this kind is not smaller enough to be considered safe or to avoid a crash.

Chapter 6

Conclusions

This thesis was developed with the goal of designing a redundant autopilot system to increase its reliability when compared with a single autopilot. To this end, the reliability and redundancy concepts were studied in Chapter 2 to choose the best architecture and the number of units in order to get the best results considering the trade-off with complexity and weight. In the end, a triple redundancy system with distributed architecture was chosen as the best approach.

In the Chapter 3 several pilots available at the market were analyzed. The PX4 was considered the best pilot to fit on this triple redundancy system. MAVLink protocol was chosen for the pilots external communications.

The system design was implemented on Chapter 4. An introduction was given about the estimation control library from PX4. Different approaches were attempted before reaching a solution which benefits from the fault detection systems existing at PX4 such as the sensor fusion and the Multi-EKF.

Afterwards, on Chapter 5 can be found the simulation results obtained from the designed system of this project when GPS or Magnetometer failures occurred as well as the simulation environment implementation. For a better understanding of the steps needed to run the simulations, the commands and routines are detailed in the appendix B.

This project has shown that three PX4 units combined, allows to detect and mitigate faults increasing the system total Reliability.

In the case of GPS and Magnetometer failures, the PX4 failsafe mechanisms were triggered before the pilot actuators control changed. Some of the error flags used by the implemented system are also used by the PX4 itself for own fault detection to trigger the failsafe. The situation caused perturbations along the flight trajectory. The pilot which receives corrupted data from the faulty sensors, initiates an emergency landing maneuver to avoid greater damages while the new pilot have not taken over the vehicle control. Two seconds are needed, at least, to update the error counters and the correspondent vote. The heavy computational communications from the interface programs limit the available computer resources. Just running the software on a powerful processor allows to reduce the time interval between iterations in real time without loosing program threads. To avoid the trajectory deviation, the fault detection speed of the system has to increase and so the time between the iterations to check the error flags

and to communicate with other pilots is decreased. While running the software on Hardware-In-The-Loop simulation mode or in real flights, the freedom to change the target variables will depend on the board processor clock speed since the interface programs are just needed for simulations (Software-In-The-Loop and Hardware-In-The-Loop) allocating resources just on the computer and not on the boards.

For a comparison between different failure causes tested on the system, the table 6.1 was filled with the time periods for fault detection followed by the pilot replacement. The magnetometer failure took more 37 seconds to be detected than the GPS failure meaning the PX4 relies more on GPS than on Magnetometer. The PX4 was able to control the vehicle properly for a longer period without magnetometers. Although any of these sensors can not be replaced by other sensor sources throughout the sensor fusion, since the pilot error flags were triggered continuously. Apart from the PX4 estimators capability to keep the flight stabilized with sensor failures, the needed time for the system to detect the fault should be reduced like it was explained on the previous paragraph. Consequently the time needed for the pilot change is reduced too because it fully depends on fault detection, usually 3 or 4 seconds after.

The PX4 communication failure is the fastest to be detected. However 7 seconds is still a lot of time in flight for the actuators to be locked by an autopilot without heartbeat. In this case, there is no error flags dependency from the PX4, so the fault detection time period will decrease considerably to 1 second if the iterations rate of the program increases. Therefore the pilot change time period could decrease to 3 seconds or less.

Table 6.1: Fault tolerance reaction performance of the system.

	Failure Typology		
Time period	**GPS**	**Magnetometer**	**PX4 communication**
Failure Detection (seconds)	6	43	4
PX4 On Charge Replacement (seconds)	10	46	7

6.1 Future Work

This section identifies some directions to improve the current work:

- System validation for accelerometer and gyroscope faults:

  Beyond the GPS and Magnetometer faults, the system was tested for Barometer fault without any consequence for the pilot selection since the PX4 changed the height source to the GPS throughout the sensor fusion using the GPS as a primary height reference.

  Both accelerometer and gyroscope faults were introduced by the failure injection command (stuck and off modes) without success because the simulation broke down every attempt for some reason that it could not be figured out. There is triple redundancy for any of these sensors, making this entire group of sensors extremely unlikely to fail.

More faults can be introduced in another sensors used by the PX4 for validation.

- Validate the system for QGC communication fault:

  The QGC communication and battery faults were not simulated yet despite of the system algorithm being already implemented to mitigate them through the error flags ”mission failure”, ”data link lost” and ”battery_healthy” observation from each PX4. So the system still needs to be validated for those faults. For the QGroundControl communication fault, it can be done modifying the interface program between the pilot instances and the QGC in such a way that one or more pilots will not be able to receive messages from the QGC.

- System validation running on HITL simulation mode:

  The next step is to validate the system running it on the Hardware-In-The-Loop simulation mode, where the software is tested on a real flight controller hardware. The interface programs must be modified and adapted to the new architecture, replacing the three TCP sockets relative to each pilot instance by the three USB connections. CEIIA has already bought 3 Hex Cube Black Flight Controller [26] for this purpose.

- Validate the system in a real flight with a low cost UAV, including already the battery and engine fault tolerance:

  The fault tolerant system will be tested on a real flight with a low cost UAV where new functionalities will be added to the redundant algorithm, like the ”engine failure” error flag observation. It will allow to verify if there are faults related with the wire connection to a specific autopilot and to distinguishing them from external faults related with the engine (when the same fault is detected on the three autopilots). The same process applies to the battery with the ”battery healthy” error flag observation.

  When testing the system on a real flight, the MAVLink API module from PX4 is not needed anymore because there is no connection between the pilots and the simulator. Therefore the interface selection programs are no more suitable. The pilots outputs are sent to the actuators by PWM throughout wires. An analog multiplexer connected to the three pilots output selects the right one to the actuators based on the final selection. A micro controller will be needed to receive and decode the MAVlink messages with the votes from each pilot, to select the most voted sending the decision to the analog multiplexer.

- Validate the system on the UAS-30

Bibliography

[1] R. Molloy and R. Parasuraman. Monitoring an automated system for a single failure: Vigilance and task complexity effects. *Human Factors*, June 1996. doi: 10.1177/001872089606380211.

[2] C. S. K. S. Sudhakar, V. Vijayakumar. Unmanned aerial vehicle (uav) based forest fire detection and monitoring for reducing false alarms in forest-fires. *Pre-proof*, 2019. doi: https://doi.org/10.1016/j.comcom.2019.10.007.

[3] L. K. Marzena Pólkaa, Szymon Ptaka. The use of uav's for search and rescue operations. In *TRANSCOM 2017: International scientific conference on sustainable, modern and safe transport*. Elsevier Ltd, 2017. doi: https://doi.org/10.1016/j.proeng.2017.06.129.

[4] P. S. Panagiotis Radoglou-Grammatikis. A compilation of uav applications for precision agriculture. *Pre-proof*, 2020. doi: https://doi.org/10.1016/j.comnet.2020.107148.

[5] B. Lynn. Rethinking drone usage on live broadcasts. `https://www.sportsvideo.org/2019/10/30/rethinking-drone-usage-on-live-broadcasts`. Accessed on: 28-11-2019.

[6] DRONEFIY. Police drone infographic. `https://www.dronefly.com/police-drone-infographic`. Accessed on: 28-11-2019.

[7] J. C. G. Emmanouil N. Barmpounakis, Eleni I. Vlahogianni. Unmanned aerial systems for transportation engineering: 4 current practice and future challenges. *International Journal of Transportation Science and Technology*, 2017. doi: https://doi.org/10.1016/j.ijtst.2017.02.001.

[8] Health and usage monitoring system (hums). Application note, Honeywell, 2020. `https://aerospace.honeywell.com/en/~/media/aerospace/files/brochures/n61-1549-000-000-onboardvibrationmonitoringsystemhums_bro.pdf`.

[9] J. L. Jiang. Automated flight data management system. Patent, United States Patent, August 2001. `https://patentimages.storage.googleapis.com/96/ed/96/c6ecddc983e559/US6278913.pdf`.

[10] C. E. C. K. N. W. Bellevue. Automatic fault reporting system. Patent, United States Patent, June 1987. `https://patentimages.storage.googleapis.com/0b/c1/50/3f5baa075f34fa/US4675675.pdf`.

[11] D. P. Todd Wilkinson. Automatic takeoff thrust management system. Patent, United States Patent, April 2005. https://patentimages.storage.googleapis.com/98/c8/96/79175b99a0c53f/US6880784.pdf.

[12] T.-Y. H. Chang-Han Wu, Ti-Jui Chen and S.-H. Tsai. Design of applying flexray-bus to federated architecture for triple redundant reliable uav flight control system. *Institute of Electrical and Electronics Engineers*, 2017. doi: 10.1109/desec.2017.8073836.

[13] Design guide, built-in-test and built-in-test equipment (bite) for army missile systems final report. Application note, US Army Missile Command, Sperry Corporation, April 2005. https://apps.dtic.mil/dtic/tr/fulltext/u2/a101130.pdf.

[14] B. D. da Costa and A. da Fonseca. *Support text slides from Integrated Avionic Systems course*. Instituto Superior Técnico, Universidade de Lisboa, 2017/2018.

[15] Micropilot. Micropilot whitepaper. Application Note 21283x, Micropilot, 2020. https://www.micropilot.com/pdf/white-papers/mp21283x.pdf.

[16] Ceiia. Uas-30. https://www.ceiia.com/single-post/2017/09/15/SEGUNDA-GERA%5C%C3%5C%87%5C%C3%5C%83O-DO-UAS30-DO-CEIIA-J%5C%C3%5C%81-TEM-ASAS-PARA-VOAR. Accessed on: 28-11-2019.

[17] Pixabay. Multi-rotor. https://pixabay.com/photos/drone-flying-camera-remote-control-1245980/. Accessed on: 14-07-2021.

[18] AugustaWestland. Project 0 tilt rotor. https://www.carbodydesign.com/media/2013/06/Project-Zero-tilt-rotor-aircraft-01-720x540.jpg. Accessed on: 28-1-2021.

[19] *ArduPilot Developer Guide*. https://ardupilot.org/dev/index.html, .

[20] *PX4 User/Developer Guide*. https://docs.px4.io/master/en/, .

[21] *Paparazzi Autopilot Developer Guide*. https://wiki.paparazziuav.org/wiki/Developer_Guide, .

[22] *LibrePilot Developer Guide*. https://librepilot.atlassian.net/wiki/spaces/LPDOC/pages/2818105/Welcome, .

[23] A. J. M. Bernardino. *Support text slides from Real Time Distributed Control Systems course*. Instituto Superior Técnico, Universidade de Lisboa, 2017/2018.

[24] *Micro Air Vehicle Message Marshalling Library Aviation, Volume I - Radio Navigation Aids*. https://mavlink.io/en/guide/define_xml_element.html, .

[25] *Real Time Publishers and Subscribers - ROS2 Interface*. http://dev.px4.io/v1.9.0/en/middleware/micrortps.html, .

[26] Cube autopilot technical report. Technical report, Pixhawk, https://ardupilot.org/copter/docs/common-thecube-overview.html, .

[27] Holybro pixhawk autopilot technical report. Technical report, Pixhawk, https://github.com/ArduPilot/ardupilot/blob/master/libraries/AP_HAL_ChibiOS/hwdef/Pixhawk4/README.md, .

[28] Beagle bone blue technical report. Technical report, Beagle Boards, https://beagleboard.org/blue, .

[29] Qualcom technical report. Technical report, Lantronix, https://www.intrinsyc.com/qualcomm-flight-pro-development-kit/, .

[30] F. D. Nunes. *Sistemas de Controlo de Tráfego - Apontamentos das Aulas*. Instituto Superior Técnico, Universidade de Lisboa, 2017.

[31] Estimation Control Library Dynamics Model Github page. https://github.com/PX4/PX4-ECL/blob/master/EKF/documentation/Process%20and%20Observation%20Models.pdf.

[32] Auterion. Px4 firmware open source. https://github.com/PX4/PX4-Autopilot.

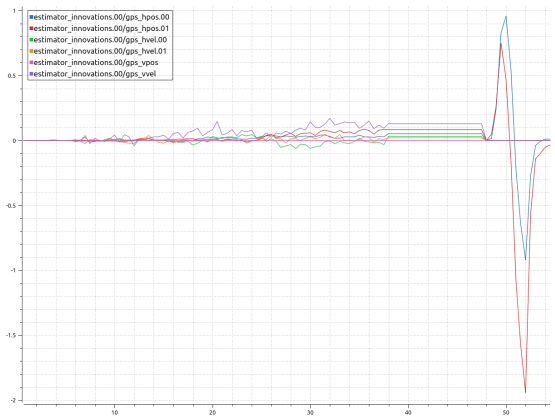
[33] PX4. Flight review. https://review.px4.io/.

[34] Marxlp. Px4 flight log visual analysis tool. https://plotjuggler.io/#one.

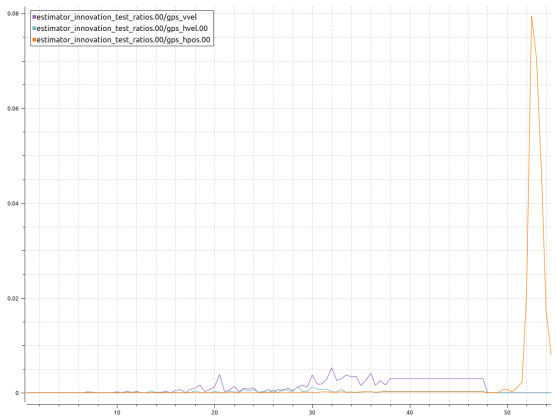
Appendix A

Extra Simulation Data - Innovations

A.1 GPS Failure

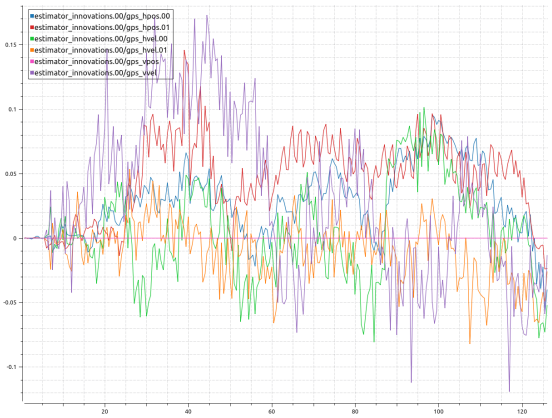


(a) PX4 instance 0. X axis - time in seconds. Y axis - GPS position innovations (meters), GPS velocity innovations (meters/second).

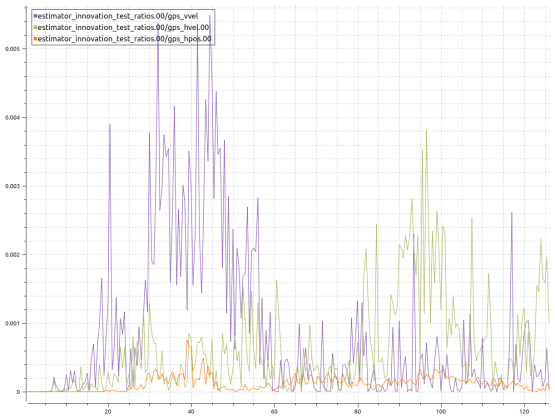


(b) PX4 instance 0. X axis - time in seconds. Y - GPS Innovations test ratios (dimensionless).

Figure A.1: PX4 Instance 0 - GPS innovations and innovation test ratios.

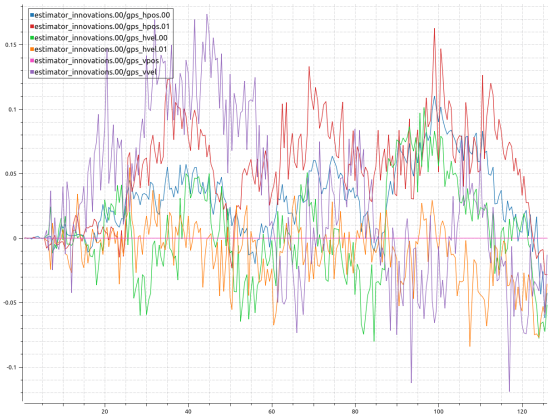


(a) PX4 instance 1. X axis - time in seconds. Y axis - GPS position innovations (meters), GPS velocity innovations (meters/second).

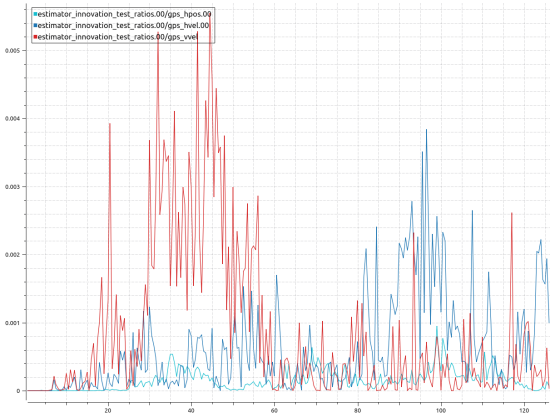


(b) PX4 instance 1. X axis - time in seconds. Y - GPS Innovations test ratios (dimensionless).

Figure A.2: PX4 Instance 1 - GPS innovations and innovation test ratios.



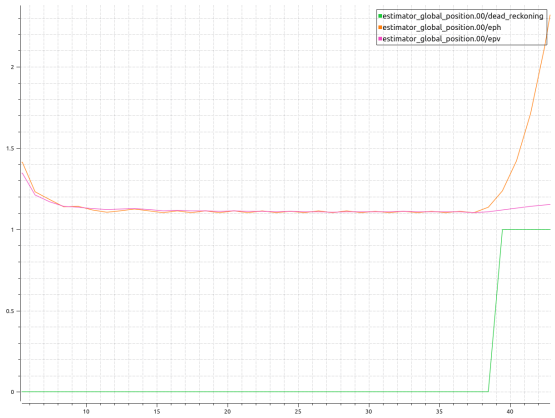
(a) PX4 instance 2. X axis - time in seconds. Y axis - GPS position innovations (meters), GPS velocity innovations (meters/second).



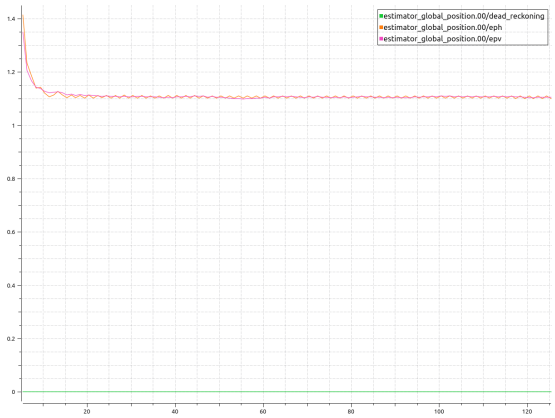
(b) PX4 instance 2. X axis - time in seconds. Y - GPS innovations test ratios (dimensionless).

Figure A.3: PX4 Instance 2 - GPS innovations and innovation test ratios.

The GPS failure does not provoke greater innovation test ratios from the GPS horizontal and vertical position or velocity, figure A.1(b). A peak is noticed after the autopilot loosing the UAV control for the second autopilot (instance 1) and so is not related with the fault injected. To be considered navigation failure, the innovation test ratio must be greater than 1 while the peak is around 0.08. For the other autopilots, in figures A.2(b) and A.3(b), the peak is around 0.005. In this case the fault was not detected because of the innovations values.



(a) PX4 instance 0. X axis - time in seconds. Y axis - Dead Reckoning (Boolean), horizontal and vertical standard deviation errors (meters).

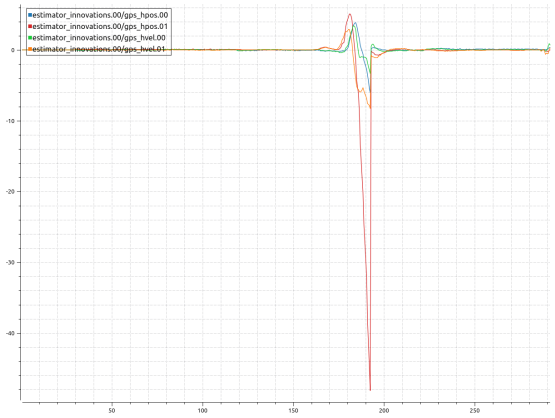


(b) PX4 instance 1. X axis - time in seconds. Y axis - Dead Reckoning (Boolean), horizontal and vertical standard deviation errors (meters).

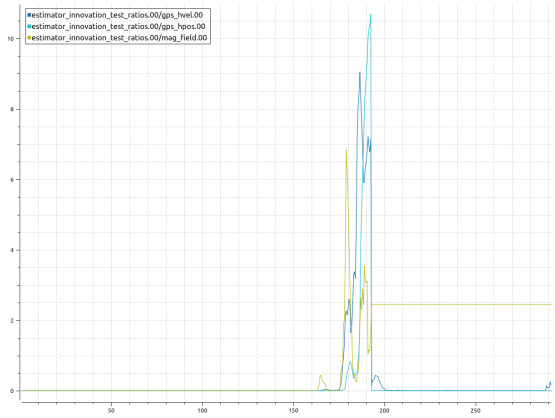
Figure A.4: Standard deviation errors and Dead Reckoning.

The dead reckoning was triggered after the 37 seconds when the fault was injected in the instance 0, figure A.4(a). Therefore this is the cause to the error flags change and to the fault being detected. The horizontal standard deviation error also increases until the end of the simulation. The instance 1 standard deviation errors and dead reckoning flag is represented in figure A.4(b), similar to the instance 2. The standard deviation errors keep constant along the simulation time and the dead reckoning is never triggered.

A.2 Magnetometer Failure

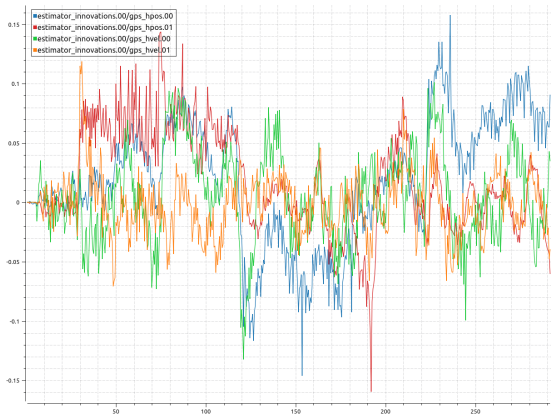


(a) PX4 instance 0. X axis - time in seconds. Y axis - GPS position innovations (meters), GPS velocity innovations (meters/second).

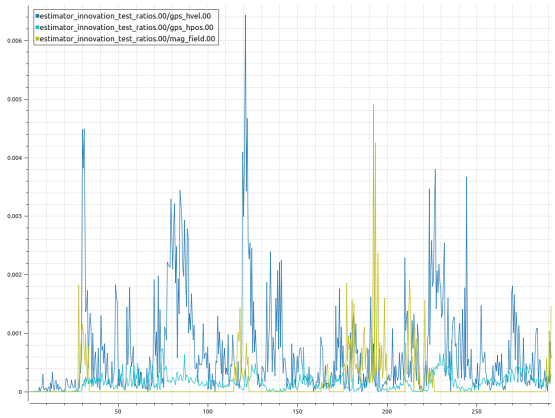


(b) PX4 instance 0. X axis - time in seconds. Y axis - GPS and Magnetometer Innovations test ratios (dimensionless).

Figure A.5: PX4 Instance 0 - GPS innovations and GPS plus MAG innovation test ratios.

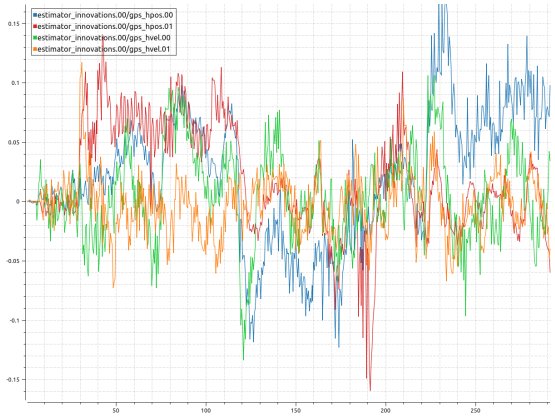


(a) PX4 instance 1. X axis - time in seconds. Y axis - GPS position innovations (meters), GPS velocity innovations (meters/second.

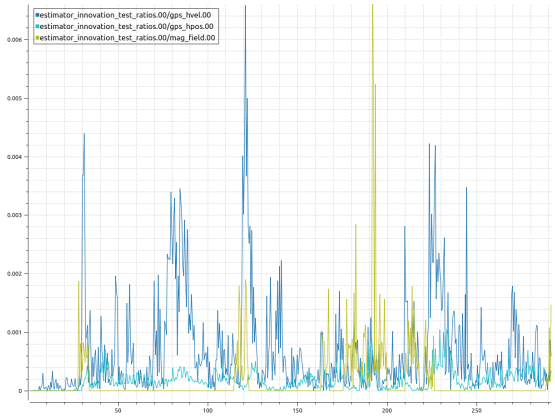


(b) PX4 instance 1. X axis - time in seconds. Y axis - GPS and Magnetometer Innovations test ratios (dimensionless).

Figure A.6: PX4 Instance 1 - GPS innovations and GPS plus MAG innovation test ratios.



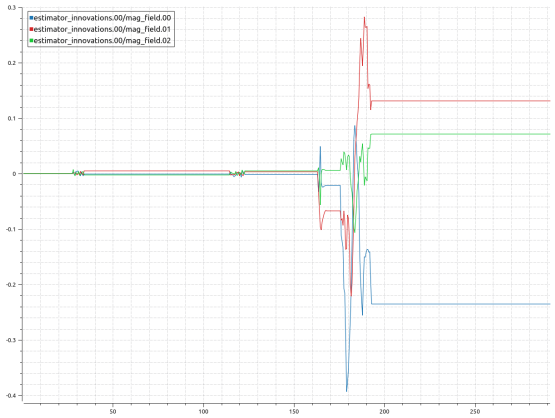
(a) PX4 instance 2. X axis - time in seconds. Y axis - GPS position innovations (meters), GPS velocity innovations (meters/second.



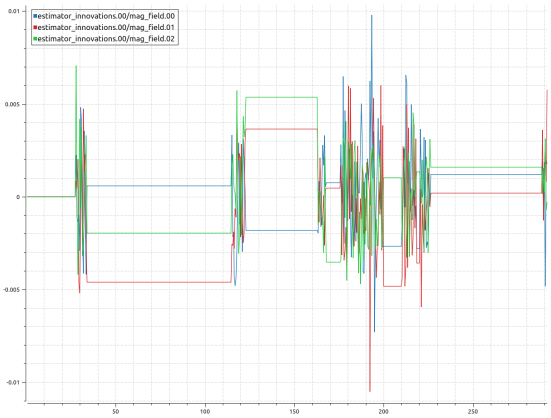
(b) PX4 instance 2. X axis - time in seconds. Y axis - GPS and Magnetometer Innovations test ratios (dimensionless).

Figure A.7: PX4 Instance 2 - GPS innovations and GPS plus MAG innovation test ratios.

The GPS and Magnetometer innovations test ratios from instance 0, figure A.5(b), are greater than 1 for a time period of 20 seconds approximately. So the fault detection is triggered by the *global_position_valid* and *local_position_valid* flags. Both reflect the respective GPS innovations, figure A.5(a) and the Magnetometer innovations A.8(a). While the other two instances, figures A.6(b) and A.7(b), keep their innovations test ratio without overtaking 1. It means the estimator is working properly.



(a) PX4 instance 0. X axis - time in seconds. Y axis - Magnetometer Innovations in the three axis (Gauss).



(b) PX4 instance 1. X axis - time in seconds. Y axis - Magnetometer Innovations in the three axis (Gauss).

Figure A.8: Magnetometer innovations.

Beyond the GPS innovations already shown, the magnetometer innovations in the three dimensions are represented in figures A.8(a) (instance 0) and A.8(b) (instance 1), which are reflected in the magnetometer test ratio for the instance 0, figure A.5(b), and for the instance 1, figure A.8(b), respectively.

Appendix B

Simulation environment procedures

In this chapter, a more in-depth explanation of the procedures to realise the simulations on the Linux operating system. Gazebo had been the simulator chosen to run the default UAV model, a quadcopter called iris. QGroundControl had been the GCS software to control the desired trajectory. Two interface programs had run to redirect the communications between the PX4 instances and QGC/Gazebo, selecting the most voted instance to control the vehicle. Finally the Mavlink protocol was updated to define new message structures for specific data communicated between PX4 instances, the states relative to each autopilot which allow to compute the error flags.

B.1 PX4 instances

Firstly, the PX4 code was downloaded using the terminal command:

```
1 git clone https://github.com/PX4/PX4-Autopilot.git --recursive
```

The redundancy manager directory from the `https://github.com/stalone89/thesis-.git` was added to the *modules* directory. Also the *px4-rc.simulator* and the *px4-rc.mavlink* run control script files from the same repository were changed in the */PX4-Autopilot/ROMFS/px4fmu_common/init.d-posix* directory because the TCP ports are defined on these files.

In order to run 3 independent PX4 instances, they must run on different terminals. On the first terminal, the command make is run with the simulation mode (SITL) and the simulator name, so the Gazebo automatically initiates with the configured default model (quadcopter - ”iris”) while the PX4 binaries had been created inside the build directory.

```
1 make px4_sitl gazebo
```

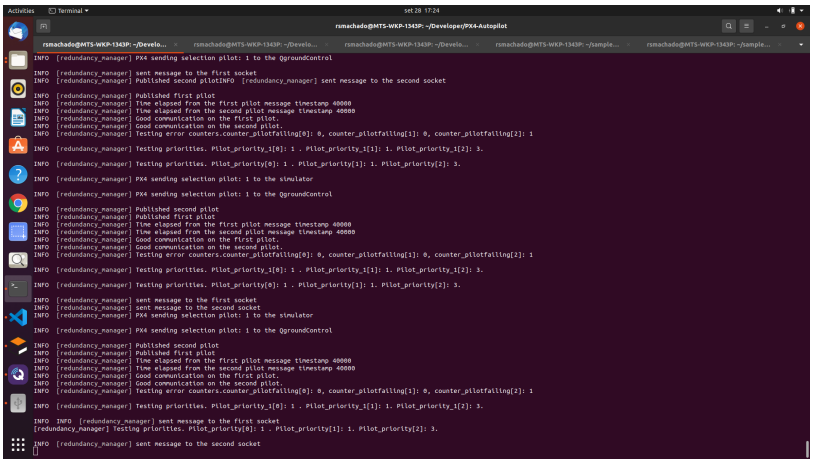


Figure B.1: PX4 instance 0 or first autopilot running on terminal.

After the simulation start, this is a typical example from the first terminal visualization, figure B.1. A lot of printed computations relative to the new added module redundancy_manager are shown on the print screen.

The second and third autopilots were called by their binaries. The ”-i” option represents the instance number. The second autopilot is associated to the instance 1, since the first autopilot was initiated by the ”make” command corresponding to the instance 0. The ”-s” option represents the Run Control file script used for the PX4 initialization. The PX4 binary is found using the *cd* command from *home* location. The PX4 model has to be set because both instance 1 and 2 do not have the environment variable initiated like the instance 0.

```
export PX4_SIM_MODEL="iris"
cd Developer/PX4-Autopilot/build/px4_sitl_default/instance_1
../bin/px4 -i 1 ~/Developper/PX4-Autopilot/build/px4_sitl_default/etc/ -s etc/init.d-posix/rcS -t
↪  "/home/pedro/Developper/PX4-Autopilot"/test_data
```

```
export PX4_SIM_MODEL="iris"
cd Developer/PX4-Autopilot/build/px4_sitl_default/instance_2
../bin/px4 -i 2 ~/Developper/PX4-Autopilot/build/px4_sitl_default/etc/ -s etc/init.d-posix/rcS -t
↪  "/home/pedro/Developper/PX4-Autopilot"/test_data
```

Both print screens are relative to the terminals where the PX4 instance 1 (figure B.2) and PX4 instance 2 (figure B.3) had run.

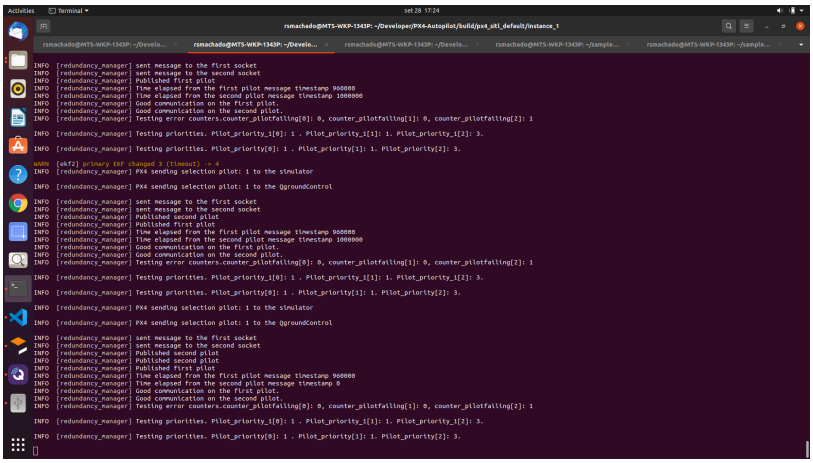


Figure B.2: PX4 instance 1 or second autopilot running on terminal.

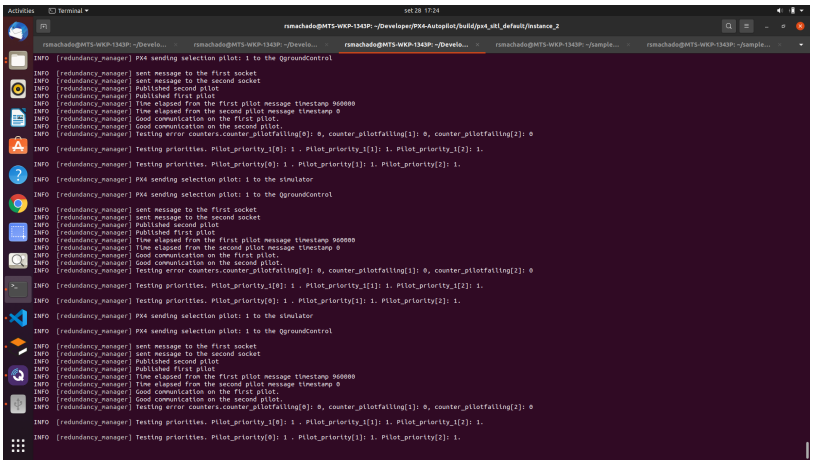


Figure B.3: PX4 instance 2 or third autopilot running on terminal.

B.2 QGC / Gazebo

The iris model flying following the Waypoints is shown in the figure B.4.

The Waypoints were defined on the mission at the QGC. In the figure B.5 is shown the UAV following the Waypoints during a mission.

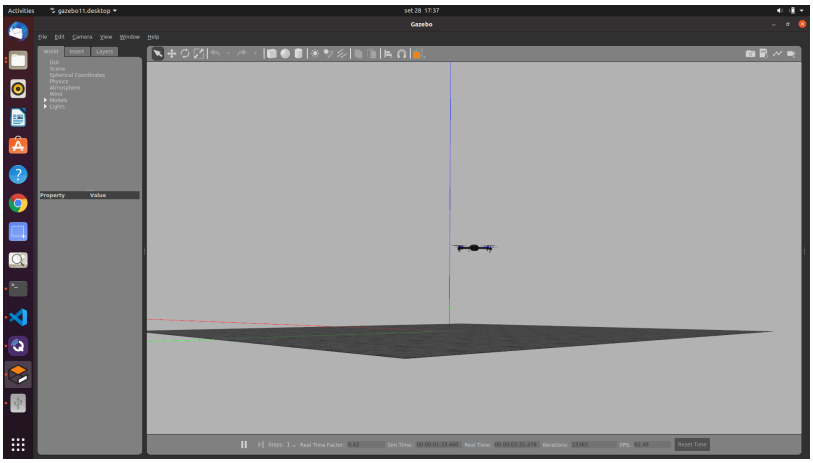


Figure B.4: Gazebo simulator print screen during the simulation.

To run the QGC, the QGroundControl.AppImage was downloaded from the website: `https://docs.qgroundcontrol.com/master/en/getting_started/download_and_install.html`

After use these two terminal commands:

```
chmod +x ./QGroundControl.AppImage
./QGroundControl.AppImage
```

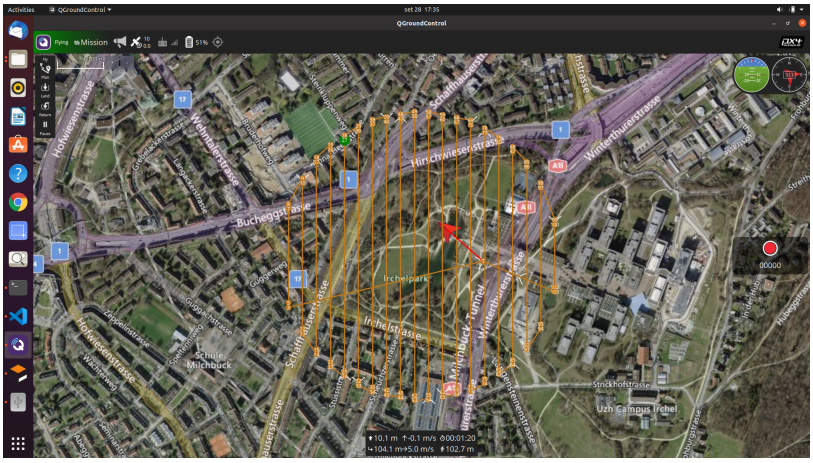


Figure B.5: QGroundControl print screen during the simulation.

B.3 Redundant Interface programs

The redundant interface programs are available in the repository `https://github.com/stalone89/thesis-.git`. The figure B.6 represents the interface program forwarding messages from the simulator to the three PX4 instances as well as forwarding messages from the first autopilot (the selected at the moment) to the simulator.

The figure B.7 represents the interface program forwarding messages from the QGC to the three PX4 instances as well as forwarding messages from the first autopilot (the selected at the moment) to

the QGC.

When a message is received from any of the sides, the message origin and its target is printed on the terminal. It is useful to track the communications and to know if the messages had been correctly redirected. Specially when the fault tolerant system had been designed and tested.

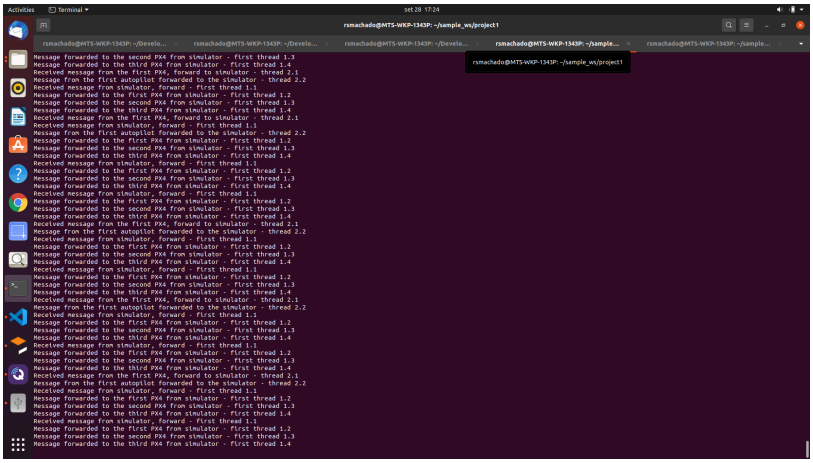


Figure B.6: Redundant interface program between the PX4 instances and Gazebo.

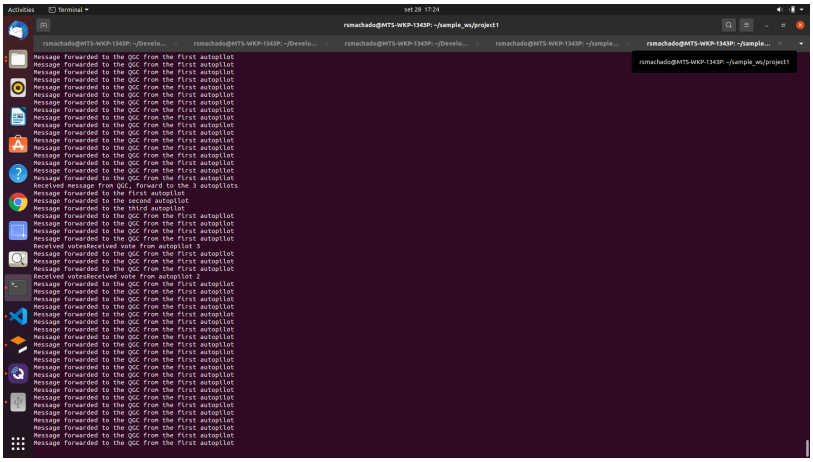


Figure B.7: Redundant interface program between the PX4 instances and QGroundControl.

B.4 Mavlink protocol update

In order to understand the communication data between autopilots, the new message structures must be updated to the default Mavlink structures already existent. The *PX4-Autopilot/mavlink/include/mavlink/v2.0/common* directory must be replaced by the ”new common” directory available in the thesis repository `https://github.com/stalone89/thesis-.git`. The new message definitions were defined in the dialect file ”.xml” and the libraries in C++ were generated through the python *mavgen* tool [24].