



UNIVERSITY OF THESSALY
SCHOOL OF ENGINEERING
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

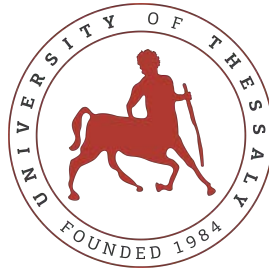
Fault-Tolerant Autopilot Operation Through Active Replication

Diploma Thesis

Maris Athanasios

Supervisor: Spyros Lalis

Month 2025



UNIVERSITY OF THESSALY
SCHOOL OF ENGINEERING
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Fault-Tolerant Autopilot Operation Through Active Replication

Diploma Thesis

Maris Athanasios

Supervisor: Spyros Lalis

Month 2025



ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

**Ανοχή Βλαβών Αυτόματου Πιλότου μέσω Ενεργού
Πλεονασμού**

Διπλωματική Εργασία

Μάρης Αθανάσιος

Επιβλέπων: Σπύρος Λάλης

Μήνας 2025

Approved by the Examination Committee:

Supervisor **Spyros Lalis**

Professor, Department of Electrical and Computer Engineering, University of Thessaly

Member **Christos D. Antonopoulos**

Professor, Department of Electrical and Computer Engineering, University of Thessaly

Member **Nikolaos Bellas**

Professor, Department of Electrical and Computer Engineering, University of Thessaly

Acknowledgements

First and foremost, I would like to thank my supervisor Professor Spyros Lalis for giving me the opportunity to tackle this thesis. Without his meticulous planning, his constant guidance, and his invaluable advice, this thesis wouldn't be possible.

Also, I would like to thank Professor Christos D. Antonopoulos and Professor Nikolaos Bellas for their participation in this thesis.

In addition, I would like to thank Manos Koutsoubelias, who provided me with important feedback and assistance during the technical parts of this thesis.

Furthermore, I would like to thank my family, who provided me with love and constantly supported me through this journey.

Last but not least, I would like to thank my friends. The fun times we've had, made this journey unforgettable.

DISCLAIMER ON ACADEMIC ETHICS AND INTELLECTUAL PROPERTY RIGHTS

«Being fully aware of the implications of copyright laws, I expressly state that this diploma thesis, as well as the electronic files and source codes developed or modified in the course of this thesis, are solely the product of my personal work and do not infringe any rights of intellectual property, personality and personal data of third parties, do not contain work / contributions of third parties for which the permission of the authors / beneficiaries is required and are not a product of partial or complete plagiarism, while the sources used are limited to the bibliographic references only and meet the rules of scientific citing. The points where I have used ideas, text, files and / or sources of other authors are clearly mentioned in the text with the appropriate citation and the relevant complete reference is included in the bibliographic references section. I also declare that the results of the work have not been used to obtain another degree. I fully, individually and personally undertake all legal and administrative consequences that may arise in the event that it is proven, in the course of time, that this thesis or part of it does not belong to me because it is a product of plagiarism».

The declarant

Maris Athanasios

Diploma Thesis

Fault-Tolerant Autopilot Operation Through Active Replication

Maris Athanasios

Abstract

Last decade, drones have infiltrated various aspects of life, providing effective aid to a plethora of tasks. Drones usually execute their mission by having one Autopilot, from which they receive instructions. However, the Autopilot can be jammed, be hacked, or break down, causing the drone to either stop working or malfunction. This shows the vulnerability of 1-Autopilot systems. To combat this problem, we developed a system in which more than one Autopilots are used by a single drone. We studied how the communication is achieved between the Autopilot and the simulation software. We then constructed an intermediate unit between the Autopilot and the simulation software that we selected, essentially orchestrating the transmission of the drone's motor commands and the simulator's sensor data from one side to the other. This intermediate unit was then extended to support multiple Autopilots and more features such as crash tolerance. Every feature was tested through an extended mission that forces the drone to go through many flight states. The tolerance of our system under the influence of noise was also tested.

Διπλωματική Εργασία

Ανοχή Βλαβών Αυτόματου Πιλότου μέσω Ενεργού Πλεονασμού

Μάρης Αθανάσιος

Περίληψη

Την τελευταία δεκαετία, τα drones έχουν διεισδύσει σε διάφορες πτυχές της ζωής, παρέχοντας αποτελεσματική βοήθεια σε μια πληθώρα εργασιών. Τα drones συνήθως εκτελούν την αποστολή τους έχοντας έναν αυτόματο πιλότο, από τον οποίο λαμβάνουν οδηγίες. Ωστόσο, ο αυτόματος πιλότος μπορεί να μπλοκαριστεί, να χακαριστεί ή να χαλάσει, με αποτέλεσμα το drone είτε να σταματήσει να λειτουργεί είτε να δυσλειτουργεί. Αυτό δείχνει την ευπάθεια των συστημάτων με έναν αυτόματο πιλότο. Για να καταπολεμήσουμε αυτό το πρόβλημα, αναπτύξαμε ένα σύστημα στο οποίο χρησιμοποιούνται περισσότεροι από ένας αυτόματοι πιλότοι σε ένα drone. Μελετήσαμε πώς επιτυγχάνεται η επικοινωνία μεταξύ του αυτόματου πιλότου και του λογισμικού προσομοίωσης. Στη συνέχεια, κατασκευάσαμε μια ενδιάμεση μονάδα μεταξύ του αυτόματου πιλότου και του λογισμικού προσομοίωσης που επιλέξαμε, ενορχηστρώνοντας ουσιαστικά τη μετάδοση των εντολών του κινητήρα του drone και των δεδομένων του αισθητήρα του προσομοιωτή από τη μια πλευρά στην άλλη. Αυτή η ενδιάμεση μονάδα επεκτάθηκε στη συνέχεια για να υποστηρίξει δύο αυτόματους πιλότους και περισσότερες δυνατότητες, όπως η ανοχή σύγκρουσης. Κάθε feature δοκιμάστηκε μέσω μιας εκτεταμένης αποστολής που αναγκάζει το drone να περάσει από πολλές καταστάσεις πτήσης. Δοκιμάστηκε επίσης η ανοχή του συστήματός μας υπό την επίδραση θορύβου.

Table of contents

Acknowledgements	ix
Abstract	xii
Περίληψη	xiii
Table of contents	xv
List of figures	xix
List of tables	xxi
Abbreviations	xxiii
1 Introduction	1
1.1 Thesis Subject	1
1.1.1 Contribution	2
1.2 Thesis Structure	2
2 Drones, Crash Failures, Byzantine Failures	3
2.1 Uses	4
2.2 Crash Failures	6
2.3 Byzantine Failures	6
2.3.1 Byzantine failures in drones	9
2.3.2 Theoretical Findings and Proposed Solutions	10
3 The Ardupilot Framework: Flow Of Execution	13
3.1 Flow of Execution and Motor Command Calculation	14

4	Gazebo Integration with Ardupilot	21
4.1	SITL and Gazebo Integration	21
4.2	Gazebo Plugin: A closer look	22
4.3	Ardupilot → Gazebo	25
4.3.1	Ardupilot Side	25
4.3.2	Gazebo Side	26
4.4	Gazebo → Ardupilot	27
4.4.1	Gazebo Side	27
4.4.2	Ardupilot Side	27
4.5	Lockstep vs Asynchronous Behavior	28
4.5.1	Frame Counter	29
4.5.2	Simple Mavlink Mission	29
4.6	Ardupilot and Gazebo Socket Communication	29
4.6.1	Ardupilot → Gazebo	30
4.6.2	Gazebo → Ardupilot	30
4.6.3	Observations	31
4.6.4	SITL Standalone	31
5	Autopilot Replication	35
5.1	Forwarder	35
5.1.1	Implementation Details	35
5.1.2	Program Usage	36
5.1.3	Validation Experiment	37
5.2	Synchronizer	38
5.2.1	Implementation Details	38
5.2.2	Program Usage	40
5.2.3	Basic Synchronizer Experiment	40
5.2.4	Extensive Synchronizer Experiment	41
5.3	SITL Behavior	45
5.3.1	SITL Code Examination	45
5.3.2	SITL Behavior Experiment	45
5.4	Tmux Propagation Delay	48
5.4.1	Automated Experiment	49

5.5	PyMavlink Library	51
6	Tolerating a Faulty Autopilot Replica	53
6.1	Crash Tolerance	53
6.2	Noise Tolerance	54
6.3	Servo Behavior	59
6.4	Visualization of the results	61
6.5	Transient errors	62
6.6	Mission Completion Under Noise	65
7	Conclusion	69
	Bibliography	71

List of figures

2.1	The Byzantine Generals model [1]	7
2.2	Representation of fault categories [2]	8
2.3	Change of states leading to system failure [2]	8
3.1	Visualized flow from target angle to motor output [3]	15
3.2	Ardupilot Command Calculation Flow	19
4.1	Gazebo plugin world (iris runway)	24
4.2	Intermediate storing of sensor data	28
4.3	Autopilot-Gazebo data exchange visualization	28
4.4	Detection of Autopilot by Gazebo	32
5.1	The Forwarder Unit	36
5.2	The Synchronizer Unit	40
5.3	First pattern of motor value discrepancies	43
5.4	Second pattern of motor value discrepancies	44
5.5	Plan layout for the packet comparisons	46
5.6	Results of the experiment	47
5.7	True representation of the packet comparisons	48
5.8	Ardupilot-Gazebo flow with Tmux	48
5.9	Ardupilot-Gazebo flow after Mavlink file upload	50
5.10	Motor values with zero discrepancy	50
6.1	Noise-Tolerant Synchronizer logic	54
6.2	Noise Introduction in the Flight Phases (Red Line)	55
6.3	Servo Behavior in the Go_Home phase	59
6.4	Servo Behavior in the Loiter phase	60

6.5	Average error of the flight phases in respect to noise level	61
6.6	Average error of policy and tolerance combination in respect to noise level	62
6.7	Average error of policy and tolerance combination in respect to noise level	65
6.8	Trajectory of the drone without noise	66
6.9	Left: Pol = 1, Tol = 30, Phase = Go_wp. Right: Pol = 2, Tol = 60, Phase = Loiter	66
6.10	Left: Pol = 1, Tol = 60, Phase = Loiter. Right: Pol = 2, Tol = 30, Phase = Go_home.	67
6.11	Left: Pol = 1, Tol = 30, Phase = Go_home. Right: Pol = 2, Tol = 60, Phase = Go_wp.	67
6.12	Left: Low Freq Center: Medium Freq Right: High Freq	68

List of tables

4.1	Differences in SITL vs SITL-Gazebo	33
5.1	Statistical results of first pattern	43
5.2	Statistical results of second pattern	44
6.1	Results of experiment sorted by ascending error	56
6.2	Best policy for each combination based on average error	58
6.3	Results of experiment sorted by ascending error	64
6.4	Best policy for each combination based on average error	64

Abbreviations

UAV	Unmanned Aerial Vehicle
ESC	Electronic Speed Controller
PWM	Pulse Width Modulation
IMU	Inertial Measurement Unit
GPS	Global Positioning System
RC	Remote Controlled
SITL	Software In The Loop
GCS	Ground Control Station
JSON	JavaScript Object Notation
WP	Waypoint
RAID	Robotic AI Intrusion Detection

Chapter 1

Introduction

Unmanned Aerial Vehicles (UAV), also known as drones, are used in many different areas of life, offering help whenever needed. In automated missions, the function of the drone solely relies on its Autopilot firmware. If something happens to that single Autopilot instance, the drone will not be able to complete the mission. Therefore, we can see that single Autopilot systems are prone to external parameters, such as noise and jamming. Because of this weakness, there is an urgent need for the development of dual Autopilot systems, increasing the drone's resilience in the possibility of malicious (or not) activity during its flight. In order to contribute to this attempt, we created a simulated dual Autopilot system using active replication, so that a possible failure in one of the two Autopilot instances can be detected and tolerated.

1.1 Thesis Subject

The subject of this thesis is to create the necessary modules, so that a simulated dual Autopilot system can be facilitated. It aims to counteract the vulnerability of the drone, when exposed to noise and other adversities. We implemented a dual Autopilot system by studying the communication between the Autopilot firmware and the simulation software, and then by creating an intermediary module between them. This module was then expanded accordingly to accommodate, at first, the introduction of a second Autopilot unit, later a crash tolerance mechanism, and finally, a fault tolerance mechanism.

1.1.1 Contribution

The contribution of our thesis can be summarized as:

1. We analyzed the calculation of motor commands in Ardupilot (the Autopilot firmware), as well as the communication between Ardupilot and Gazebo (our simulation software).
2. Several modules were implemented between Ardupilot and Gazebo, including a forwarder, a synchronizer, and a crash tolerance module.
3. All the above modules were tested through the appropriate experiments for each module.
4. We evaluated the tolerance of the system under the influence of noise, for different noise levels, flight phases, and policies.

1.2 Thesis Structure

The thesis is structured in the following way. Chapter 2 provides an overview of drones, their uses, and the concepts of Crash Failure and Byzantine Failure. Chapter 3 investigates the flow of the Ardupilot Framework and the process of motor command calculation. Chapter 4 describes the simulation software used (Gazebo) and analyzes the Ardupilot - Gazebo communication and data exchange. Chapter 5 describes the replication process along with all the modules that we developed. In Chapter 6 we evaluate the tolerance of our system by adding noise to our second Autopilot. Lastly, Chapter 7 concludes our work and provides ideas for a future implementation.

Chapter 2

Drones, Crash Failures, Byzantine Failures

An Unmanned Aerial Vehicle (UAV), is an aircraft without any human personnel on-board [4]. Drones are usually of small or of medium size. Since a pilot is not present, the classic way by which a drone can be controlled and maneuvered, is through an operator. Utilizing 4G (and nowadays also 5G) connectivity, the drone executes the commands that it receives. A possible problem arises when the network is slow. In that case, the operator issues a command, and the drone executes it later than expected, possibly after a new command has been issued, making the drone appear out of sync. Therefore, a solid connection must be established to ensure low latency and a smooth flight.

Once a command has been issued to the drone, how does the drone know which steps to take, in order to execute this command successfully? This crucial information is given by the Autopilot. In simple terms, the autopilot is a piece of firmware. What this firmware contains is hundreds of thousands of lines of code, which are responsible for the correct functionality of the drone. For example, this functionality includes, but is not limited to, reading sensor data correctly, producing the correct motor commands and deploying fail-safe mechanisms when the drone is met with unexpected -environmental or not- adversities.

The firmware is then compiled and installed on the autopilot board. The autopilot board is a hardware device, that is placed on the drone. This hardware device contains a microprocessor, sensors (accelerometers, gyroscopes, Inertial Measurement Units) and other useful modules such as RC Channels and GPS receivers). Once the setup has been completed, the

firmware starts running. We can assume that the drone is not moving yet, therefore what the firmware does first is to initialize the components of the drone. During the initialization procedure, the commands that it can accept from the user are very limited and involve the selection of the flight mode. Flight mode is the way the autopilot adds certain fixed properties to the flight [5]. It can change mid-flight depending on the purpose of the mission. Once the initialization has been completed, the drone is ready to receive commands from the operator and execute them.

When the drone is in a ready state, the autopilot is responsible for correctly directing the flow of the execution to the hardware components of the drone. There are hardware components that are imperative for the correct function of the drone, and which are being used in every command. The sensors are a good example of such components. Others, like a camera, are more supplementary in nature and can be used in specialized operations. We can see autopilot as the “decision maker”, (lower-level decisions than the operator’s decisions) while the hardware executes the steps to fulfill that decision.

Every x hz (x can vary depending on the board), the main autopilot flow is executed. The flow begins with reading data from the sensors and ends with Pulse-Width Modulating signals (PWM) being sent to the Electronic Speed Controllers (ESCs). In a later chapter, we will study this flow in more detail.

2.1 Uses

We have briefly described how a drone works. Due to their complexity, they are utilized in different aspects of life, in many different ways [6]. We will discuss some of them.

- **Surveillance.** Police departments can utilize drones to keep an eye on possible crimes or infringements. While many highways can be covered through surveillance cameras, countryside roads and hard-to-reach areas like the woods could greatly benefit from a drone presence (arson protection). Private properties can also use drones for surveillance. Through a technique called RAID [7], which stands for Robotic AI Intrusion Detection, the drone circles the property and is able to distinguish the authorized personnel from an intruder, issuing an alert only on the latter case.

- **Agriculture.** The way farmers manage their operations has drastically changed with the evolution of drones. Many tasks that were done manually by the farmers, for so long, are now done by agricultural-specific drones. In crop harvesting, a drone, using computer vision, scans the crop, successfully determining whether it is ready to be harvested. In greenhouses, drones can be used to check the temperature and the humidity and keep them at the appropriate levels. Using aerial imaging, the farmer can inspect his vegetables from the comfort of his home, having to intervene only if a problem is detected. The pesticide spraying process is also automated [8].
- **Photography.** Incorporating a camera to a drone opens up new possibilities, through a technique called photogrammetry. This technique involves utilizing a drone to take multiple photos of a certain area from different angles. Through these photos, a three-dimensional model of the area can be constructed, and it can be later used for an in-depth analysis. Photogrammetry finds applications in many fields, examples being construction and civil engineering. An example case is visual odometry. If the drone's GPS malfunctions, the drone can calculate its position by taking pictures of its environment.
- **Product Delivery.** Drones can also be used for transporting goods, resulting in faster delivery times, as traffic in the world of aviation is practically non-existent. They are much better for the environment as they emit significantly fewer polluting substances than the traditional trucks. Also, drones can easily access otherwise inaccessible areas, providing important products to the people living there, such as medicine. It has to be noted that during the delivery process, drones are connected to the network, and therefore can be hijacked, along with what they carry. Therefore, security concerns must be addressed.
- **Rescue Missions.** In a lot of natural disasters or accidents, due to the severity and the dangerousness of the situation, human personnel cannot be blindly sent in to help, but nonetheless a set of actions must be taken. In cases like these, drones use cameras and radars to provide important information to the rescuers, or to the rescue robots. Techniques like thermal imaging and signal detection are used, with the drone serving as a guide of how the situation must be approached.

2.2 Crash Failures

A crash failure is when a process, unexpectedly, loses its ability to function. They can occur either from a hardware or from a software malfunction. An example of a hardware malfunction is a RAM with corrupted memory, while an example of a software malfunction is a bug in the logic of the program. Frequent crash failures can have a crippling effect on the afflicted system. For example, if a server of a renowned company often crashes, customers will slowly turn away, resulting in a loss of reputation for the company. Crash failures can be mainly categorized to **permanent** and **transient** crashes [9]. As the names of both categories suggest, a permanent crash means that the system will not recover on its own, and will recover only if the issue is found and resolved. On the other side, a transient crash means that the system briefly lost functionality, but managed to resolve the issue itself. In order for crashes to be mitigated, diligent testing of the system must be ensured. However, given their unexpected nature, a fault tolerant approach is an even better safety measure, with the replicated module taking over when the original module crashes.

If a crash failure happens to a drone mid-flight, the outcome, unlike a byzantine failure, is pretty straightforward. The drone will fall to the ground, deemed unusable after the fall. In some cases, a pilot might take over, after a notification is sent from the Ground Control Station. However, such an approach isn't viable for automated missions, as a pilot is rarely available.

2.3 Byzantine Failures

Before we discuss the concept of Byzantine failure, we first need to define what a Byzantine fault is. A **Byzantine fault** is a state of a system, in which a part of the system may start to misbehave and there is not sufficient information on whether that part has failed. It goes hand in hand with the Byzantine Generals problem, which we will discuss later.

The lack of information regarding the byzantine fault comes from the fact that the possibly faulty component, presents different symptoms to different observers. Therefore, a consensus between the other parts of the system cannot be reached and the component cannot be officially declared as faulty.

The above is clearly illustrated by the **Byzantine Generals Problem** [1]. In its simple form, the problem states the following: A group of generals want to attack a castle. The attack will be successful only if all generals agree on it. If at least one general decides to opt out of the attack, but the decision to attack has been made, then the attack will fail. The generals choosing to retreat, instead of attacking, is also a valid solution, as long as it is unanimous. A general is called a **traitor** if he breaks the unanimity of the decision. A traitor, proposing an attack to some generals, and retreat to some others, can be seen as a component with faulty behavior in a distributed system.

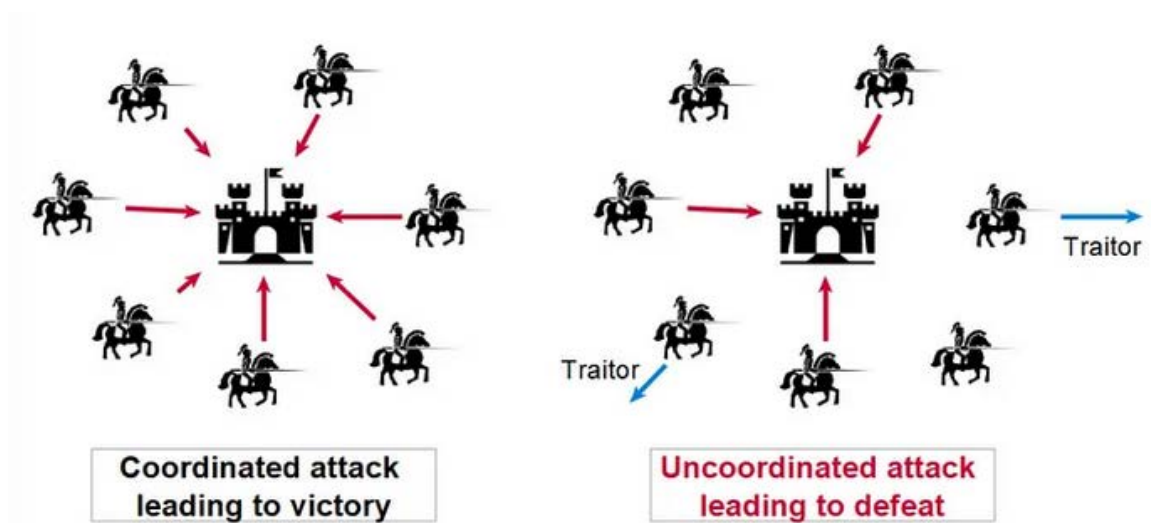


Figure 2.1: The Byzantine Generals model [1]

Byzantine faults belong in a broader category of faults, called system faults. System faults are broken down into 4 categories, transient, intermittent, permanent, and byzantine faults [2]. Briefly:

A **transient** fault is a non-recurring temporary fault and can be resolved with a simple action, such as a system restart. An example of a transient fault is a small network connectivity issue.

An **intermittent** fault is a recurring temporary fault which may require complex actions and meticulous debugging to be solved. If they remain unresolved, they can slowly harm a long-running system.

A **permanent** fault stays in the system until it gets fixed. They result in system failure, but treating them is normally an easy process.

A **byzantine** fault, with characteristics as described earlier. The hardest category, mainly because these faults are hard to identify, given the unstable behavior of the examined component.

The visual representation of these categories is presented in 2.2

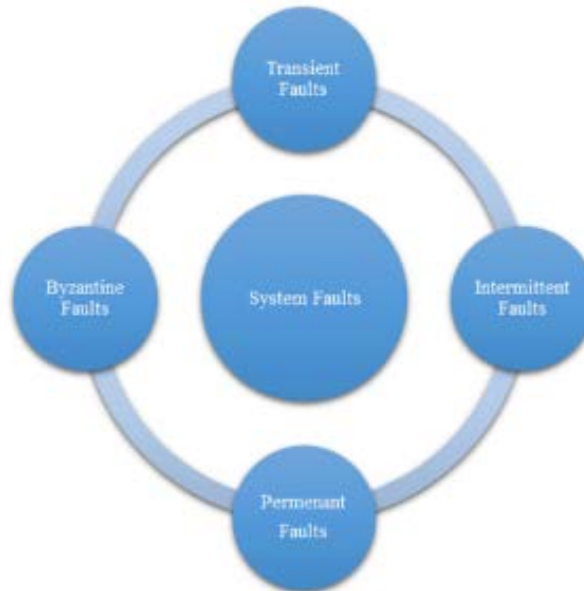


Figure 2.2: Representation of fault categories [2]

If the fault is not countered quickly, the system moves from the fault state to the error state. The error state is when the outcome of the system starts to deviate from the correct outcome. This can be managed, as long as the error is within the expected tolerance. If the error gets significantly larger than the tolerance, then the system has moved to the failure state. This change of states can be seen below:



Figure 2.3: Change of states leading to system failure [2]

Therefore, a **Byzantine failure** is the breakdown of the system, which occurred by intolerable errors, which errors, in turn, occurred by a **Byzantine fault** (component with unstable behavior). We can compare Byzantine Failure with the case of a failed attack in our Byzantine Generals model.

A real-world example of a technology that is prone to and must tolerate Byzantine faults

is **Blockchain** [10]. In Blockchain, because it is a decentralized system, consensus must be reached among nodes, for an action to be taken (e.g. seller verification). However, some nodes might act maliciously (on purpose or because of defection), making consensus harder to be reached. In a rare-case scenario, if malicious nodes are over half of the nodes in the network (51%), then consensus is reached, and fraudulent transactions can be verified. This poses a serious threat to the financial integrity of the platform, so detection mechanisms must be applied.

2.3.1 Byzantine failures in drones

Byzantine faults can appear in drones, either when a drone works standalone, or when a group of them are connected in the same network. In the latter case, they can perform coordinated activities, forming a swarm. We will discuss some of the problems that can occur due to a byzantine fault and their possible repercussions to the operation [11], [12]:

Falsely reporting position. In a swarm operation, it is obvious that each drone should be aware of the positions of the other drones. However, during a byzantine fault, a drone can send two different coordinates of his location, to two different drones, just how a traitor can propose both attack and retreat, to different generals. This discrepancy regarding the drone's location can cause a crash between the drones. This problem can also occur during a malicious attack, where a false position is deliberately shared across all drones in the swarm, with the hijacker attempting to misguide or break up the swarm.

Falsely reading sensor data. Sensors are crucial for the drone's function. GPS, IMU, and accelerometer sensors' data, are used by the autopilot to produce the motor commands. Therefore, if one of these sensors becomes faulty, then the motor commands produced will not correspond to the actual state of the drone. This can create a dangerous situation for the drone. For example, a drone might think that it's flying above an obstacle, while in reality, flies directly at it, and possibly crashing head-on.

Failure in reporting task completion accurately. When several drones work together, the operation may require that tasks are divided between the drones. For example, in a search and rescue mission, a drone has to scan the area of interest for possible dangers, before another drone can go in. If, due to a fault, the drone completes the scan, but does not report it back, the second drone will falsely wait, essentially delaying the operation. If the scan is reported,

but not completed, then the second drone is in active danger, as it is falsely led to believe that it is entering a safe area. We can see that both cases are seriously disruptive for the mission at hand.

2.3.2 Theoretical Findings and Proposed Solutions

In the original paper [13], in which the Byzantine general model was presented, the authors figured that instead of having n generals of equal rank, the problem can be reduced to a commander general, who commands several lieutenant generals (subordinates). If the commander is loyal, then all loyal lieutenants must obey the order the commander sent to them, reaching a unanimous decision. Using this new model, they prove that, in a scheme of 3 generals, of which 1 is a traitor, no solution can be found, if the communication is achieved through oral messages. Then, using the previous proof as a contradiction, they proceed to show that if there are m **traitors** in the system, in order for a solution to exist, at least $3 \cdot m + 1$ generals are needed. Furthermore, they propose an algorithm, called the **Oral Message** algorithm (OM), which solves the Byzantine Generals problem for $3 \cdot m + 1$ or more generals, when there are at most m traitors in the system.

By using oral messages, we saw that the problem becomes difficult to solve, because the traitors can either lie, or fabricate the messages that they receive, towards other lieutenants. Therefore, the authors introduce the concept of **unforgeable signed messages**. This concept includes the assumptions that the signature of a loyal general cannot be forged, and that the editing of his message by a third party can be detected. They go on to propose an algorithm called the **Signed Messages** algorithm (SM), which breaks the limitation of the previous algorithm and is able to deal with m traitors for any number of generals.

Another important outcome is that there is no completely asynchronous consensus protocol which can withstand at least 1 silent process death. This means that there is no general solution for an asynchronous system that is characterized by a byzantine failure, unless we make the problem more specific, and we do not impose strict regulations to our solution. We will later show that our execution flow between autopilot and the physics system is synchronous (lockstep), making a general consensus protocol not impossible.

A recent solution, that involves technologies of Machine Learning and that has been tested

on a swarm of robots, is reducing the consensus problem on a **Best-of-n Problem** [14]. In the mentioned problem, we have n options, with each option having a **quality function** and a **cost function**. These are multi-variable functions, with the variables being the features of the testing environment. The robots have to decide for the option that maximizes the quality function and minimizes the cost function. Unless a single option manages that, a trade-off between quality and cost ensues, choosing one over the other depending on the needs of the given simulation.

Chapter 3

The Ardupilot Framework: Flow Of Execution

In order to study how the autopilot works, we decided to use the Ardupilot framework. It is a very popular framework that has applications in many vehicles like copters, planes, subs, and rovers. Since the motor values that Ardupilot produces will play an integral role later in our project, we will analyze how these motor values are produced, by following the flow of the framework. Starting from a high-level Mavlink command, we will see how the command is being broken down, reaching the ESCs [3]. The important folders of the framework are:

AntennaTracker: Contains code that is executed when the object of reference is an antenna tracker.

ArduCopter: Contains code that is executed when the object of reference is a copter (drone).

ArduPlane: Contains code that is executed when the object of reference is an airplane.

ArduSub: Contains code that is executed when the object of reference is a submarine.

Blimp: Contains code that is executed when the object of reference is blimp.

Build: Contains files relative to the compilation of SITL.

Libraries: Contains the implementation of all the ardupilot libraries. This folder is the core of the framework and is used by all vehicles.

Modules: Contains external third-party modules that are used by the project.

Rover: Contains code that is executed when the object of reference is a rover.

Tools: Contains miscellaneous support directories

The folders that we will mostly use in our analysis are the ArduCopter and the Libraries folder.

3.1 Flow of Execution and Motor Command Calculation

The execution begins by specifying the mode in which the drone will run. The modes are enumerated in the **mode.h** [15] file in an enum called **Number**. Below is a snippet of that enum:

```
enum class Number : uint8_t {
    STABILIZE = 0,
    ACRO = 1,
    ALT_HOLD = 2,
    AUTO = 3,
    ...
}
```

The enum continues until all 26 modes are declared.

The way that a mode is being selected, is through the **set_mode()** function in the **mode.cpp** [16] file:

```
bool Copter::set_mode(Mode::Number mode, ModeReason reason)
```

The mode that we want ends up being stored in the parameter that is responsible for the flight mode of the drone, called **flightmode**:

```
Mode *new_flightmode = mode_from_mode_num(mode);
flightmode = new_flightmode;
```

After a mode has been selected, a command can be placed, and the execution moves to the respective .cpp file of the selected mode. There is a .cpp file for every mode, in the pattern of mode_name_of_mode.cpp (e.g. mode_drift.cpp). For our analysis we will assume that the selected mode is Guided Mode.

In the **mode_guided.cpp** [17] file, the guided controller **run()** is called periodically. This controller contains sub-controllers, which control the position, velocity, and acceleration of the drone, depending on which sub-mode the drone is on. What these controllers have in common is that the last instruction is a call to the **AC_Attitude_Control** library. The function that is called may differ depending on the sub-controller, but because it is a function of **AC_Attitude_Control**, the execution moves to that library. For example, in the **pos_control_run()** sub-controller:

```
attitude_control->input_thrust_vector_heading();
```

where **input_thrust_vector_heading** is an **AC_Attitude_Control** function.

AC_Attitude_Control is responsible for taking as input a desired angle or rate, calculating the error from the actual angle or rate, and reducing that error to roll, pitch, and yaw values (motor requests). While the processing of the input may differ from function to function, depending on the form the input is given in (euler frame, body frame etc.), the common thing most functions have is that they call the **attitude_controller_run_quat()** [18] function, which calculates the angular velocity corrections.

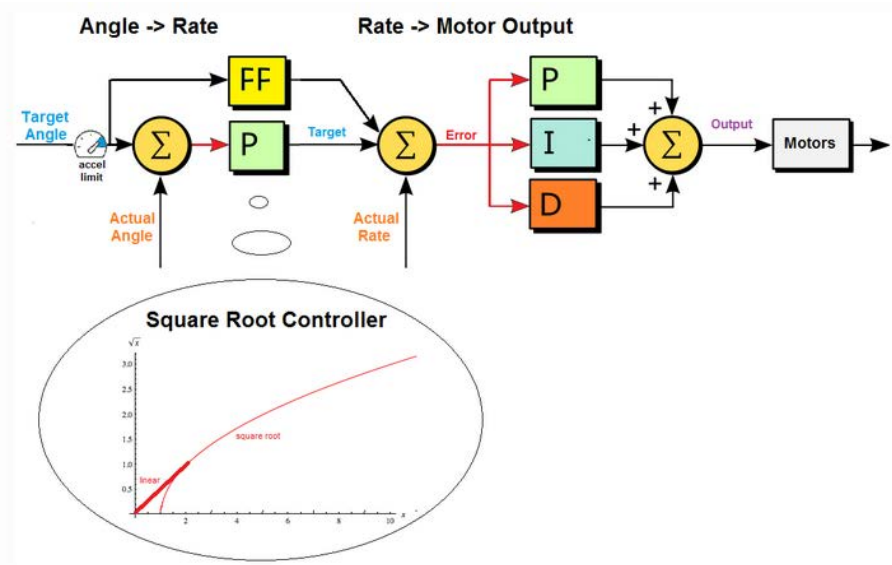


Figure 3.1: Visualized flow from target angle to motor output [3]

The flow of the **attitude_controller_run_quat()** is the following:

The attitude error is calculated:

```
thrust_heading_rotation_angles(_attitude_target,
attitude_body, attitude error, _thrust_angle,
_thrust_error_angle);
```

Using the attitude error, the angular velocity corrections are calculated:

```
_ang_vel_body = update_ang_vel_target_from_att_error(attitude_error);
```

The above function uses 3 Square Root Controllers (seen as P in 3.1) and after calculations, the output of these controllers (one for each axis) ends up being the angular velocity corrections.

We then calculate the feedforward:

```
ang_vel_body_feedforward = rotation_target * _ang_vel_target;
```

And we add it to our angular velocity corrections, just like 3.1):

```
_ang_vel_body.x += ang_vel_body_feedforward.x * _feedforward_scalar;
_ang_vel_body.y += ang_vel_body_feedforward.y * _feedforward_scalar;
_ang_vel_body.z += ang_vel_body_feedforward.z;
```

where **_feedforward_scalar** is the percentage of the feedforward velocity used in the calculation.

Once we have our corrections, the execution moves to the **rate_controller_run()** function of the **AC_AttitudeControl_Multi.cpp** [19] file. This rate controller is multicopter-specific, runs periodically, and its purpose is to convert the corrections to roll, pitch, and yaw and to store them to a **motor class** object.

We will examine how roll is set, and the process is exactly the same for pitch and yaw:

```
_motors.set_roll(get_rate_roll_pid().update_all(
_ang_vel_body.x, gyro_latest.x, _dt, _motors.
limit.roll, _pd_scale.x) + _actuator_sysid.x);
```

get_rate_roll_pid() returns an object of type **AC_PID**, which is a classic PID controller:

```
AC_PID& get_rate_roll_pid() override {return _pid_rate_roll;}
```

update_all [20] is the function that calculates the roll from the angular velocity corrections. It uses a PID controller, and after complex calculations, it calculates the proportional term **P**, the derivative term **D**, and the integral term **I**, and adds these three terms together (just like we see in the second half of 3.1:

```
return P_out + D_out + _integrator;
```

The above term is passed as input to the **set_roll()** function and eventually gets stored in the **_roll_in** variable, which represents the motor roll, in the range of [-1,1]:

```
void set_roll(float roll_in) { _roll_in = roll_in; };
```

The exact same procedure is followed for the pitch and yaw variables.

The **set_roll**, **set_pitch**, and **set_yaw** functions are part of the **AP_Motors_Class.h** [21] file, which belongs in the **AP_Motors** library, and as we will see, the flow of execution has now moved to the **AP_Motors** library.

The function **output()** in the **AP_MotorsMulticopter.cpp** [22] file processes roll, pitch, and yaw, transforms them to motor values, and sends them to the ESCs, by calling several other functions:

update_throttle_filter(): Updates the throttle filter, which is responsible for avoiding abrupt changes in e.g. velocity of the drone.

output_logic(): This function runs the spooling logic. It runs the required logic, depending on the spooling state that the drone is on (idle, spooling up, spooling down, or throttle unlimited). The RPY commands are not used here.

output_armed_stabilizing(): In this function, the roll, pitch and yaw are used to produce the thrust output for each motor. Specifically:

```
_thrust_rpyt_out[i] = roll_thrust * _roll_factor[i];
_thrust_rpyt_out[i] = _thrust_rpyt_out[i] + pitch_thrust * _pitch_factor[i];
_thrust_rpyt_out[i] = _thrust_rpyt_out[i] + yaw_thrust * _yaw_factor[i];
```

roll_factor[i], **pitch_factor[i]**, and **yaw_factor[i]** are the *i*th motor's contribution to roll, pitch, and yaw respectively and are relative to the motor frame type that the drone uses.

As for the thrust variables, they are produced using our original roll, pitch, and yaw values, along with their feedforward values:

```
float roll_thrust = (_roll_in + _roll_in_ff) * compensation_gain;
float pitch_thrust = (_pitch_in + _pitch_in_ff) * compensation_gain;
float yaw_thrust = (_yaw_in + _yaw_in_ff) * compensation_gain;
```

Lastly, we incorporate throttle in our thrust output:

```
_thrust_rpyt_out[i] = (throttle_thrust_best_plus_adj *
_throttle_factor[i]) + (rpy_scale * _thrust_rpyt_out[i]);
```

output_to_motors(): Once we have our thrust output, this function transforms it to an actuator through `thrust_to_actuator()`, and then calculates raw PWM from that actuator using `output_to_pwm()`:

```
set_actuator_with_slew(_actuator[i],thrust_to_actuator(_thrust_rpyt_out[i]));

for (i = 0; i < motors; i++) {
    rc_write(i, output_to_pwm(_actuator[i]));
}
```

The formula used for the conversion is: $\text{pwm} = \text{pwm_min} + (\text{pwm_max} - \text{pwm_min}) * \text{actuator}$.

rc_write(): This function moves the execution to the **SRV_Channel** library, as this function is responsible for sending the output to the communication channels. We see inside **rc_write()** that:

```
SRV_Channels::set_output_pwm(function, pwm);
```

where **function** is an enum that shows what channel we want to write our PWM value to.

set_output_pwm(): This function goes through all the available channels and stores the PWM value to the appropriate channel [23]:

```
for (uint8_t i = 0; i < NUM_SERVO_CHANNELS; i++) {
    if (channels[i].function == function) {
        channels[i].set_output_pwm(value);
        channels[i].output_ch();
    }
}
```

In the above code, if the examined channel is a motor channel, then we store the PWM value to that channel. We then call **output_ch()** to push the channel value to the ESCs.

output_ch(): This function is responsible for pushing the channel value to the ESCs. Specifically:

```
hal.rcout->write(ch_num, output_pwm);
```

where **ch_num** is the number of the channel and **output_pwm** is a member variable of the channel, in which a PWM value is stored. **hal.rcout -> write** is a virtual function, therefore its implementation depends on the HAL of the system.

What we described above is how a high-level user input is turned into motor commands through the flow of the framework. The above flow is parameterizable in the sense that different commands and different modes can be activated, but the core logic stays the same. Overall, it can be visually depicted in 3.2.

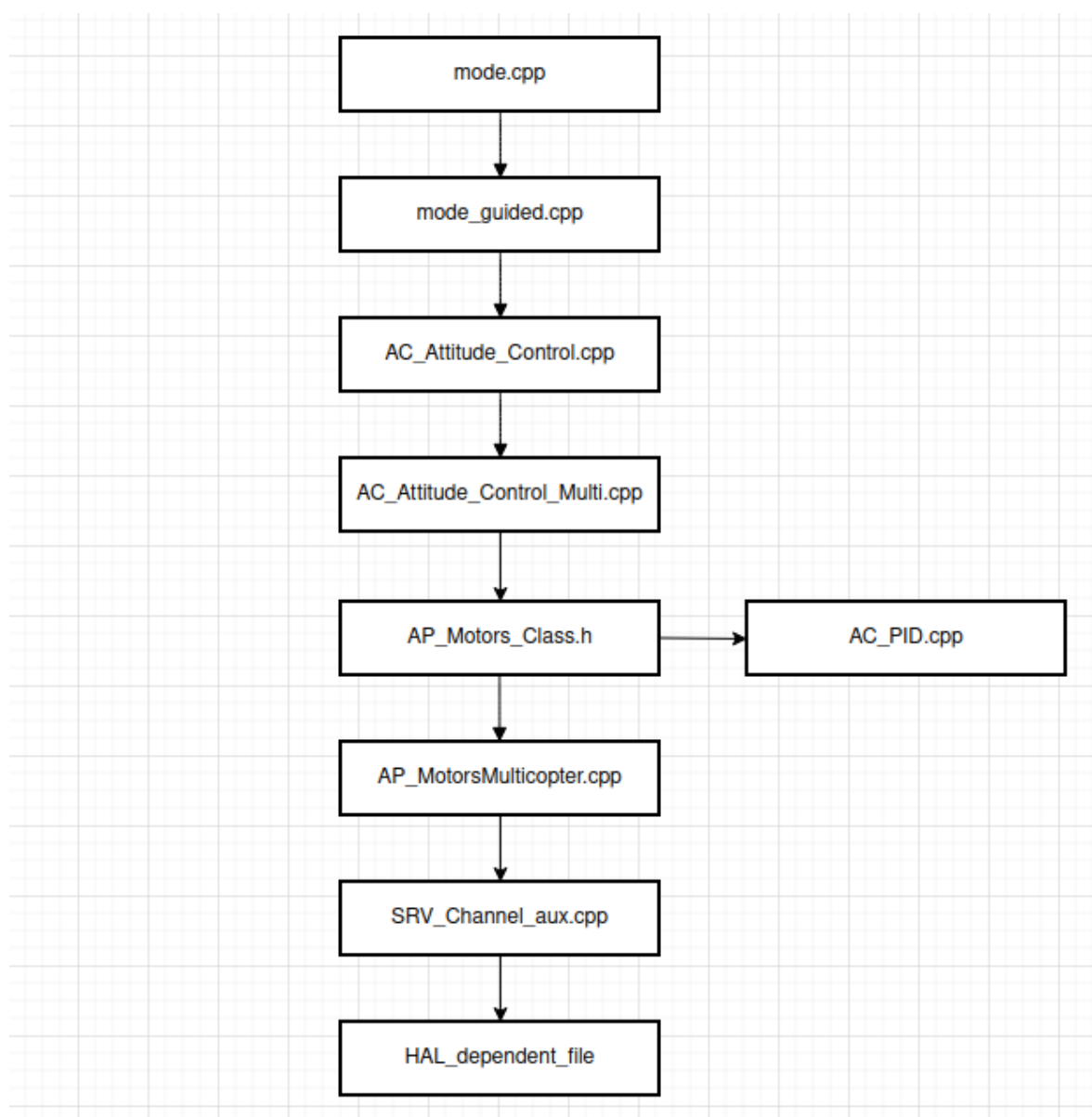


Figure 3.2: Ardupilot Command Calculation Flow

Chapter 4

Gazebo Integration with Ardupilot

4.1 SITL and Gazebo Integration

SITL (Software In The Loop) is a great way to run the Autopilot firmware without needing actual hardware (an actual drone). The Autopilot firmware runs on a computer and its features can be tested virtually. Because of the lack of the sensors, the sensor data are produced by SITL itself, based on the simulated environment that we're in. It is also compatible with Ground Control Stations(GCS) so the developer can check on crucial drone data through telemetry.

However, the graphic capabilities of SITL are very limited, and it does not provide a physics engine, so using it standalone, although the motor commands are properly calculated, we are not provided with a very accurate representation of what the drone actually does. Therefore, we need an advanced graphic simulator that would have the capability to simulate the physics of the drone and provide the sensor data, which the autopilot will work with. We will specifically use Gazebo.

Gazebo is an advanced open-source physics simulator which is used to simulate situations that include one or more(swarm) of robots [24]. It is written in C++ and mainly used with Linux. Some of its features are:

- **Sensors.** Gazebo offers a lot of sensors, including depth cameras, Inertial Measurement Unit (IMU), Light Detection and Ranging Sensors (LIDAR), altimeter, accelerometer

and more. They could be used without noise, or a noise parameter can be introduced, to simulate malicious or non-malicious interference.

- **3D Graphics.** Gazebo uses the OGRE 2.1(Object-Oriented Graphics Rendering Engine) Engine for the production of graphics. OGRE offers materials, shaders, meshes, animations, and special effects, all of which contribute to an accurate depiction of the simulated environment.
- **Physics.** Physics are simulated by the DART (Dynamic Animation and Robotics Toolkit) physics engine. Dart is a physics engine especially useful for robotics and computer animation. Some of its features involve support of SDF (Simulation Description Format), which is the main file format that Gazebo uses, collision detection through multiple detector types, a plethora of kinematic states, and powerful support for joint dynamics.
- **Support For Plugins.** While Gazebo offers some basic libraries, code that is written by other people can also work with these libraries in the form of a plugin. These plugins can introduce new models and new worlds that are compatible with specific types of simulations (e.g. drone simulation).

So, in our case, Gazebo will be used in combination with SITL, Gazebo providing the graphics and SITL providing the motor commands.

We will use a plugin which connects Ardupilot to Gazebo. Namely, autopilot sends a packet, that contains the motor commands, to the Gazebo plugin, the plugin processes the packet, moves the drone in the simulated environment, produces the sensor data, and sends the sensor data back to autopilot. Autopilot then uses these sensor data to produce the next motor commands packet.

4.2 Gazebo Plugin: A closer look

From now on, we will refer to the Gazebo plugin [25] we described earlier as Gazebo, for convenience purposes. The plugin has 3 important functions, **configure()**, **preupdate()**, and **postupdate()**.

Configure(): This function is called once, and is responsible for initializing the simu-

lation environment, along with the sensors. It creates the model, which in our case, is the **Iris Quadcopter**, the specs of which we will reference later. It then creates the simulation world, in our case that is **iris_runway.sdf**, a simple runway. It changes the Gazebo frame to the NED frame, a frame commonly used in aviation. After that, it initializes the sensors, containing IMU, GPS, range and wind sensors. Lastly, it initializes the socket logic, so that connection between Gazebo and Autopilot can be established.

PreUpdate(): This function is called in a loop. It is responsible for receiving a packet from the autopilot and making changes to the simulation, in accordance with the PWM commands it received from the packet. Initially, it verifies that the sensors are in a working condition by performing some checks. Then, it receives the packet through socket logic and extracts the PWM commands of the incoming packet. After some calculations, they get stored internally in Gazebo in an array of controllers. That array of controllers is then used to update the Gazebo PID controllers and make changes in the simulation environment.

PostUpdate(): This function is called in a loop and after PreUpdate(). It is responsible for sending the new state to the autopilot. Initially, it gathers the sensor data given the current position of the drone. Then, it creates a JSON file and writes that sensor data to the JSON file. If some of the sensors are missing, the corresponding values are not written in the JSON file. After the JSON file has been written, it is sent to the autopilot through socket logic.

So, we can view the flow of execution like this:

```
Configure();           // Executing only once

while (autopilot_online) { // Main loop
    PreUpdate();

    PostUpdate();
}
```

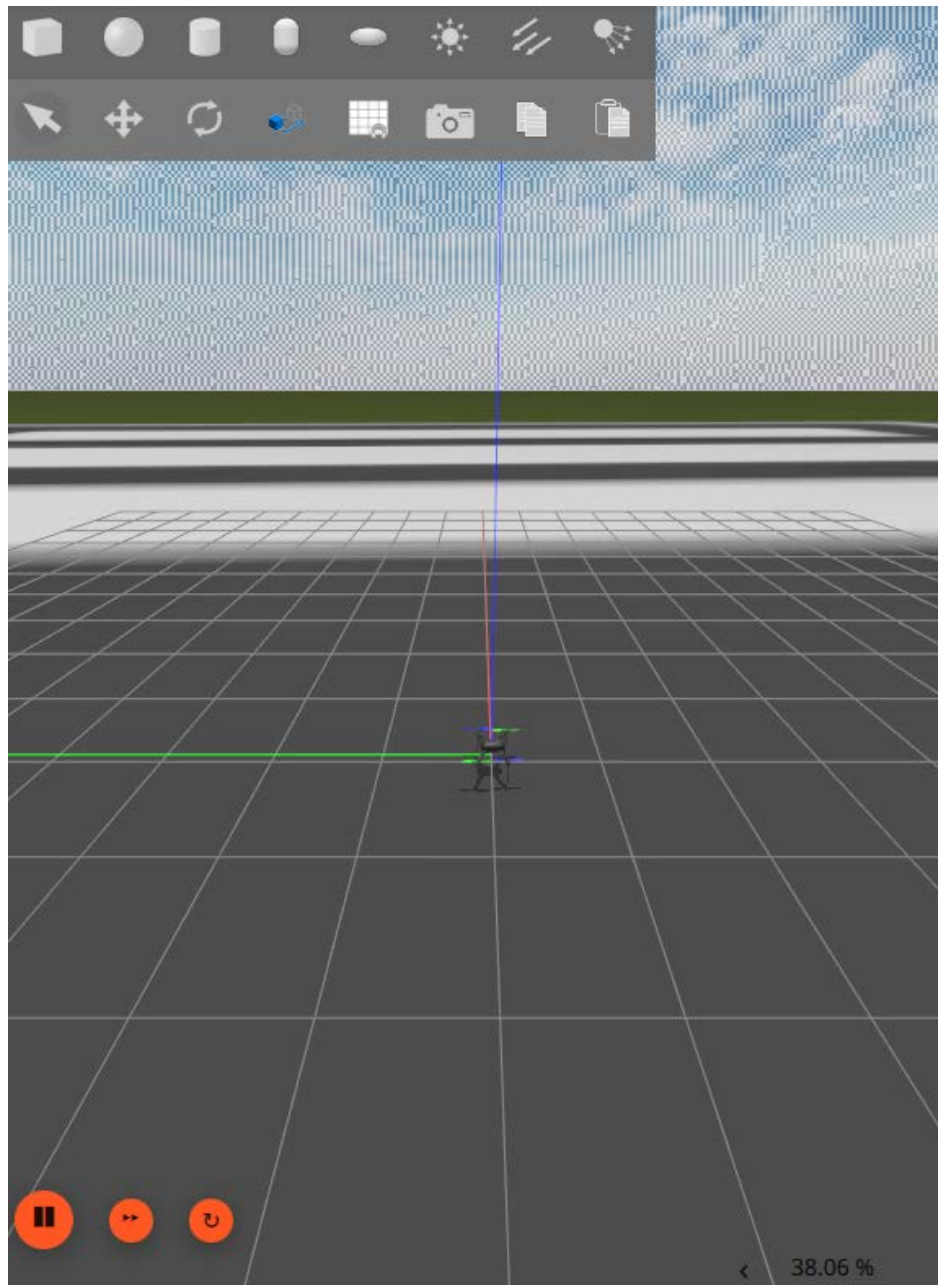


Figure 4.1: Gazebo plugin world (iris runway)

As a next step, we will go through the communication between the two sides, as understanding it is crucial for the replication process. From the Autopilot side, the file **SIM_JSON.cpp** [26] is activated and from the Gazebo side, the file **ArdupilotPlugin.cc** [27] is activated. Therefore, the functions mentioned can be found in these files.

4.3 Ardupilot → Gazebo

4.3.1 Ardupilot Side

The packet that the Autopilot sends to Gazebo is the following:

```
struct servo_packet_16 {

    uint16_t magic = 18458; // constant magic value
    uint16_t frame_rate;
    uint32_t frame_count;
    uint16_t pwm[16];

};
```

We notice that in SITL, the PWM commands are sent directly to Gazebo, without getting sent to the RC Channels or the ESCs. This is happening because both of these concepts are useful only in a real drone scenario, hence not used by SITL. The variable **magic** is used for packet authentication by Gazebo and has a fixed value, the variable **frame_rate** shows the frequency of a packet sent in Hz, the variable **frame_count** shows how many Autopilot packets were received by Gazebo, and the array **pwm[16]** carries the PWM commands for each servo.

For the other fields:

```
pkt.frame_rate = rate_hz;
pkt.frame_count = frame_counter;
```

Rate_hz and **frame_counter** are SITL member variables. **Rate_hz** is initialized to 1200 Hz and **frame_counter** is initialized to 0.

```
for (uint8_t i = 0; i < 16; i++) {
```

```

    pkt.pwm[i] = input.servos[i];
}

```

Pwm[16] takes its values from the **servos[32]** array, which contains the calculated motor commands. Because the package that we are sending deals with 16 servos at max, we only need the first 16 values of the array, and only the first 4 are actually used, since our drone is equipped with 4 motors.

In this way, every field has been set, and the packet is sent to Gazebo through a socket.

4.3.2 Gazebo Side

The function **ReceiveServoPacket()** is responsible for packet decoding in Gazebo side.

```

pkt_magic = pkt.magic;
pkt_frame_rate = pkt.frame_rate;
pkt_frame_count = pkt.frame_count;
std::copy(std::begin(pkt.pwm), std::end(pkt.pwm), std::begin(pkt_pwm));

uint16_t magic = this->dataPtr->have32Channels ? magic_32 : magic_16;
if (magic != pkt_magic) {
    ...
}

this->dataPtr->fcu_frame_rate = pkt_frame_rate;
this->dataPtr->fcu_frame_count = pkt_frame_count;

```

Pkt_magic is used for packet authentication as mentioned before, while **pkt_frame_rate** and **packet_frame_count** are stored in the corresponding Gazebo member variables.

The motor commands also end up getting stored in a Gazebo member variable, specifically an array of controllers, after some processing:

```

this->dataPtr->controls[i].cmd = multiplier * (raw_cmd + offset);

```

Multiplier and **offset** are fixed for each motor, however **raw_cmd** is dependent on the PWM value received from Autopilot for the *i*'th motor.

4.4 Gazebo → Ardupilot

4.4.1 Gazebo Side

Gazebo sends a JSON file to the Autopilot, containing the sensor data it produced. The function responsible for the creation of the JSON file is **CreateStateJson()**. Initially, IMU data, linear acceleration, angular velocity, pose (position and orientation), velocity, range sensor data and wind data are gathered. We should notice that some sensors (e.g. range sensors) may not be available in the model that we are using. That is not a problem, as long as the sensors deemed necessary by Autopilot are available. Then, the JSON building begins.

The **rapidjson** library is used for the JSON generation. Rapidjson offers many advantages, as its memory requirements are minimum, while its performance is astonishing.

```
rapidjson::StringBuffer s;  
rapidjson::Writer<rapidjson::StringBuffer> writer(s);
```

The collected data are written in a **StringBuffer** object with the help of a **Writer** object, for more convenience. Once the data have been written, they are extracted through the **GetString()** function and are stored in the form of a continuous string:

```
this->dataPtr->json_str = std::string(s.GetString());
```

That string is then sent to Autopilot using a socket.

4.4.2 Ardupilot Side

The JSON file gets parsed in the **parse_sensors()** function through a keytable, which gives information regarding the type and the necessity of the data being parsed. Autopilot can handle lack of data from some sensors, as long as **timestamp**, **gyroscope data**, **acceleration**, **position** and **velocity** are provided. During the parsing of the values, they are stored in a temporary struct **state** before they get stored in the corresponding SITL member variables. **State** can be seen as an intermediary, which ensures that SITL is not in a modified state, if something goes wrong during parsing. This is clarified further through 4.2.

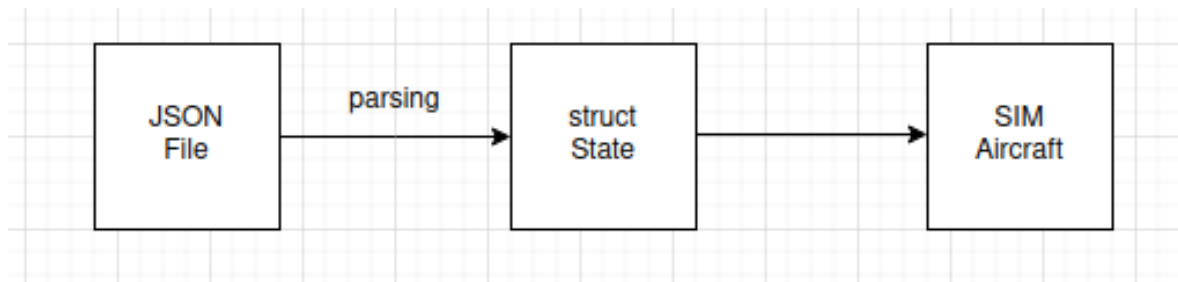


Figure 4.2: Intermediate storing of sensor data

In 4.3, a diagram of the data flow between the two sides is presented. The Autopilot blue square represents the calculation of the motor commands (treated as a black box) and the Gazebo blue square represents the changes made to the simulation (treated as a black box).

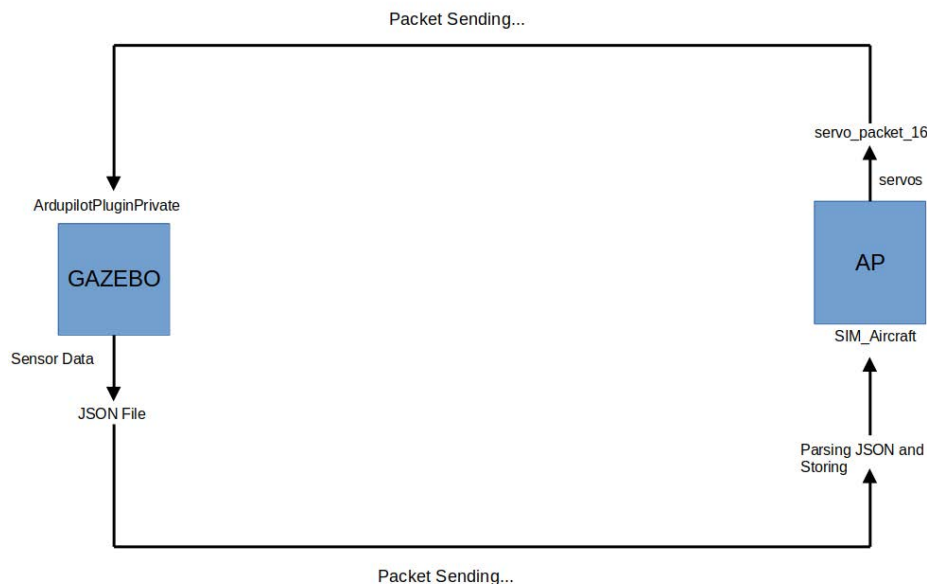


Figure 4.3: Autopilot-Gazebo data exchange visualization

4.5 Lockstep vs Asynchronous Behavior

After establishing the data flow, the next step is to find out whether our model is **lockstep** or not. A model is said to be lockstep when, during its execution, the previous function must finish before the next function starts. This behavior allows the program to run in a synchronized manner, effectively making it easier to witness an expected outcome. A model is said to be **asynchronous**, when functions can proceed independently, without following a linear path. We tested our model by introducing a small delay (3 seconds) after each packet

transmission on each side separately. We utilized the frame counter and a simple Mavlink mission, to prove what category our model belongs in.

4.5.1 Frame Counter

The frame counter of Autopilot shows us the total number of packages the Autopilot received. We injected the delay from the Autopilot side. If the frame counter kept increasing with the same rhythm as without the delay, this essentially means that Autopilot is receiving packets without sending packets, making our model asynchronous. If, on the other hand, the frame counter conformed with the delay, Autopilot received a packet only after it sent a packet, therefore a linear path is followed, and our model is lockstep. We noticed that the frame counter did conform with our delay, indicative of a lockstep model. We tried different delays to make sure it is delay-dependent.

We conducted the same experiment, this time injecting the delay from the Gazebo side, and as before, the Gazebo frame counter conformed with the delay. Therefore, both sides exhibit lockstep behavior, and our model is **lockstep**.

4.5.2 Simple Mavlink Mission

We decided to verify the lockstep model by running a Mavlink mission (`arm()`; `takeoff()`; `goto WP()`; `RTL()`;) with a small delay, to see if the mission will be completed successfully, albeit in slow motion. We started with a delay of 1 sec, but we had to reduce it because each side considered the other as offline. For a delay of 10ms, the mission was tested 2 times, injecting the delay on Autopilot and Gazebo separately. Both of these times, the mission was successfully completed in slow motion, reinforcing our suspicions that the model is **lockstep**.

4.6 Ardupilot and Gazebo Socket Communication

Autopilot and Gazebo exchange their data through sockets. It is important to understand and analyze how this communication is achieved.

4.6.1 Ardupilot → Gazebo

Autopilot sends its package through a **sendto()** function:

```
sock.sendto(&pkt, pkt_size, target_ip, control_port);
```

Target_ip is the ip address of the recipient and **control_port** is the port the recipient listens to. In our case, Gazebo is the recipient. The variables are hard-coded to the values 127.0.0.1 (callback address) and 9002 respectively. The callback address is used because Autopilot and Gazebo run to the same machine. Had that not been the case, the ip address of the recipient's machine would be used.

On the Gazebo side, we have:

```
public: std::string fdm_address{"127.0.0.1"};
public: uint16_t fdm_port_in{9002};

this->dataPtr->sock.bind(fdm_address.c_str(), fdm_port_in);
```

Therefore, Gazebo binds to port 9002 and Autopilot sends the packet to port 9002, guaranteeing proper communication.

4.6.2 Gazebo → Ardupilot

Gazebo sends the JSON file through a **sendto()** function:

```
this->dataPtr->sock.sendto(
    this->dataPtr->json_str.c_str(),
    this->dataPtr->json_str.size(),
    this->dataPtr->fcu_address,
    this->dataPtr->fcu_port_out);
```

Fcu_address is the ip address of the recipient and **fcu_port_out** is the port the recipient listens to. Unlike Autopilot, Gazebo does not know these values beforehand. It discovers these values from the autopilot packet it receives, through the **last_rcv_address()** function:

```
void SocketAPM::last_rcv_address(const char *&ip_addr, uint16_t &port)
{
```

```
ip_addr = inet_ntoa(in_addr.sin_addr);  
port = ntohs(in_addr.sin_port);  
}
```

The ip address and the port are extracted from the struct **in_addr** which contains the sender's information. The function is called accordingly:

```
_sock.last_recv_address(_fcu_address, _fcu_port_out);
```

In this way, Gazebo knows exactly where to send the JSON file, ensuring proper communication.

4.6.3 Observations

Given the fact that Gazebo makes his port and address known beforehand, while Autopilot doesn't, we can induce that Gazebo acts as a server and Autopilot acts like a client. Also, the port that Autopilot listens to, changes in every execution of the simulation. That means that Autopilot does not have a fixed port like Gazebo. A temporary port is assigned to it by the operating system when it tries to send a packet for the first time. 4.4 shows the communication and detection between the two sides.

4.6.4 SITL Standalone

SITL can also work in the absence of Gazebo through **SIM_Multicopter.cpp**, the default SITL file. The main difference is that the sensor data aren't provided to SITL, therefore it has to produce them on its own. It produces them through some simple equations, which work like a very simple FDM inside SITL. We checked to see which variables of the SITL-Gazebo model are used in the pure SITL model in Table 4.1.

We can see that the sensor data used in SITL is generally on par with SITL-Gazebo, however the accuracy of that data is lower, due to the simpler computational model used in SITL.

Note that, not all the SITL-Gazebo variables are present in the JSON file that Gazebo sends. The data that are actually being sent depend on the sensor capabilities of the drone.

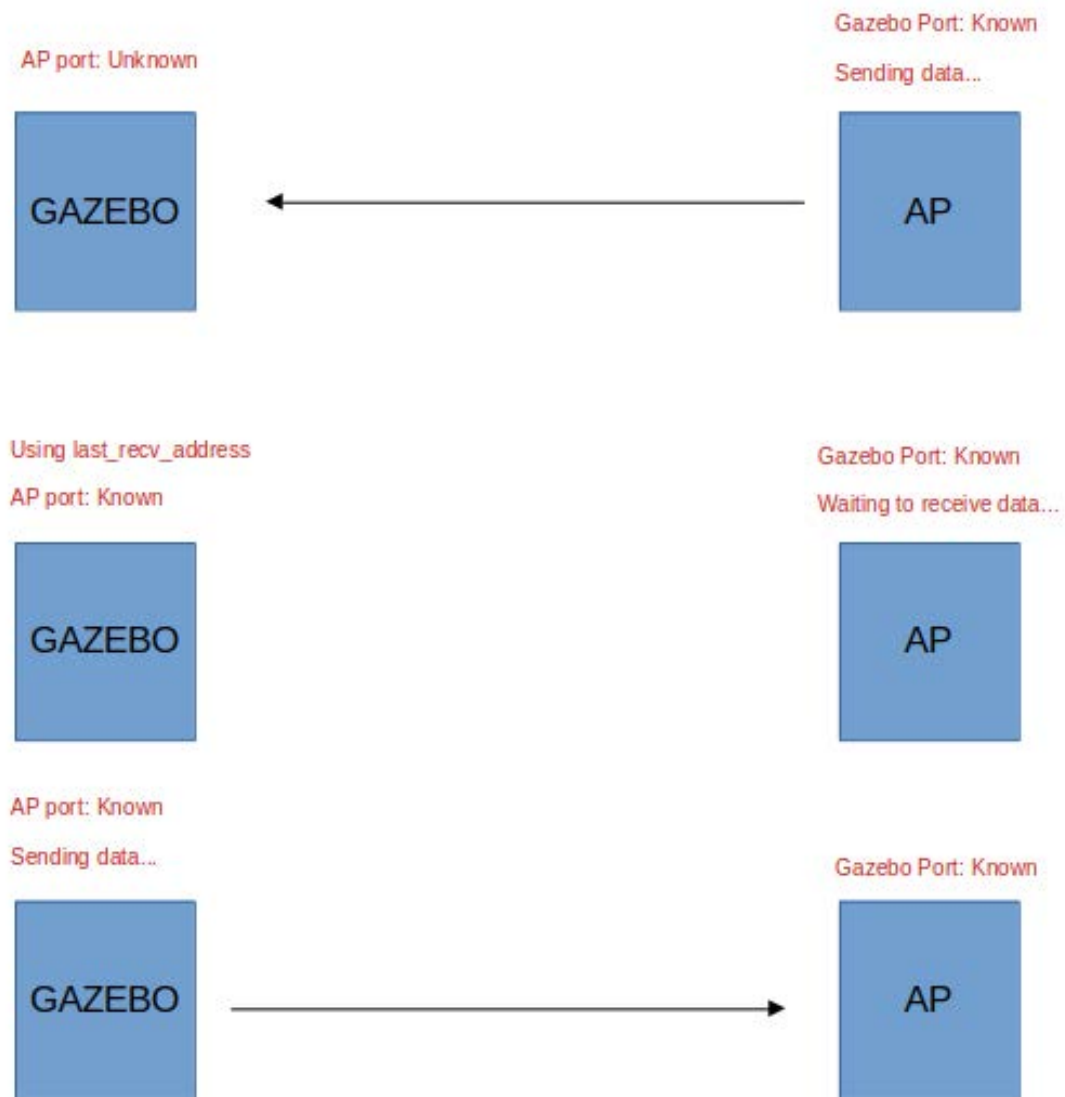


Figure 4.4: Detection of Autopilot by Gazebo

SITL-Gazebo Variables	Used in SITL Standalone
accel_body	Yes
gyro	Yes
velocity_ef	Yes
position	Yes
use_time_sync	Yes
dcm	Yes
velocity_air_bf	Yes
airspeed	Yes
airspeed_pitot	Yes
rangefinder_m	Yes
wind_vane_apparent direction	No (Simulator only)
wind_vane_apparent speed	No (Simulator only)
last_timestamp_s	No (Simulator only)

Table 4.1: Differences in SITL vs SITL-Gazebo

Chapter 5

Autopilot Replication

In this chapter, we will initially go through the replication process and later present our experimental results to evaluate our attempt.

5.1 Forwarder

Before we add a second autopilot to the equation, it is important to validate that the communication between Autopilot and Gazebo can work as expected through an intermediary program, in our case, the forwarder. In a high level, the forwarding loop will work like this:

1. Initially, the forwarder receives a motor commands packet from Autopilot.
2. Once the packet has been received, the forwarder will send the packet to Gazebo.
3. After Gazebo changes the drone's state and produces the sensor data, the forwarder receives the sensor data from Gazebo.
4. Lastly, the forwarder sends the sensor data to Autopilot, so that the new packet can be produced.

5.1.1 Implementation Details

Autopilot and Gazebo communicate using the UDP protocol. Since our forwarder acts as the middleman, it must also adhere to the UDP protocol, therefore that is the protocol we used. So, whenever we had to receive data, we used the **recvfrom()** function and whenever

we had to send data, we used the **sendto()** function.

Regarding the ports, we know that Autopilot sends the packet to port 9002, given that Gazebo listens to that port. So, we made the forwarder listen to port 9002 and Gazebo listen to another port (9022), so that Autopilot sends its packet to the forwarder and not to Gazebo directly. So, from a lower-level port perspective, the forwarding loop becomes as follows:

1. Autopilot sends the packet to port 9002 (forwarder).
2. The forwarder (9002) receives the packet and sends it to port 9022 (Gazebo).
3. Gazebo (9022) receives the packet and sends the sensor data back to port 9002 (forwarder).
4. The forwarder (9002) receives the sensor data and sends it to port X (Autopilot). We know port X from the sockaddr structure that comes along with the data from Autopilot.

Schematically, the forwarder is presented in 5.1:

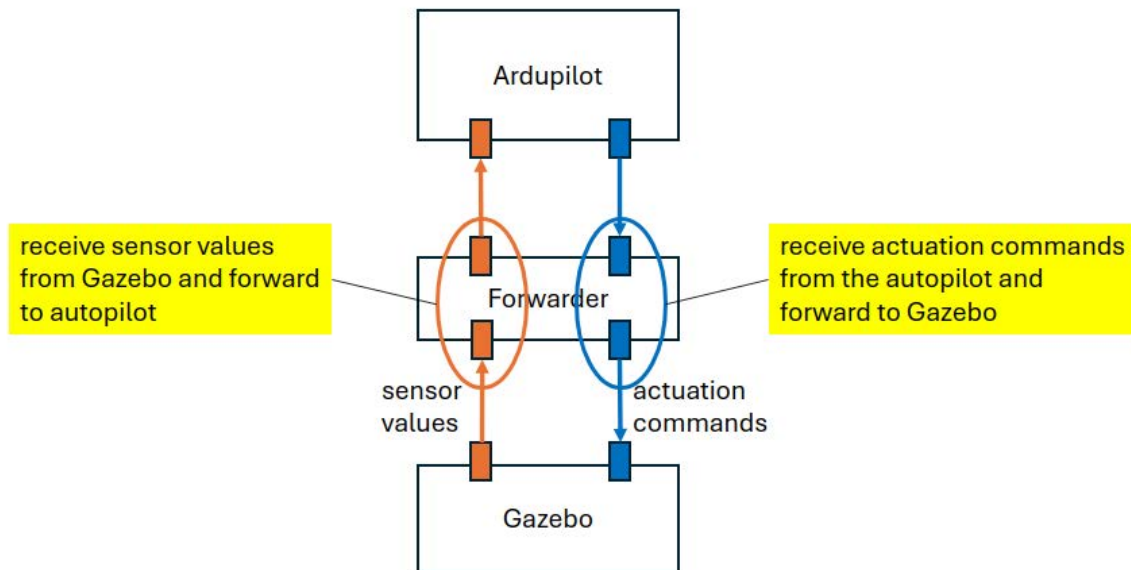


Figure 5.1: The Forwarder Unit

5.1.2 Program Usage

In order to use the forwarder as the means of communication:

1. We run the forwarder program. The program then rests to the first `recvfrom()` function (waiting packet from Autopilot).

2. We start the Gazebo plugin. The communication is not initiated at that point, as the Autopilot has not started working yet.
3. We start the Autopilot. The communication is initiated and kept on going through the forwarder.

5.1.3 Validation Experiment

We conducted a small experiment to verify the correct function of the forwarder. The experiment was the following:

```
arm();  
takeoff();  
goto wp();  
mode rtl();
```

The takeoff altitude was 2 meters, and the waypoint was approximately 10 seconds away from the drone's home position and we used MavProxy as our Ground Control Station. The drone completed the experiment successfully, and the communication between the two sides involving the forwarder, worked as expected.

5.1.3.1 Iris Quadcopter

For the above experiment and for the rest of the experiments, we used a simulated version of the Iris Quadcopter [28], [29]. It is a popular copter that runs on the Pixhawk autopilot system. It has many automatic capabilities such as takeoff, loitering, RTL (Return To Launch), and landing. Its height is only 10cm, making it very portable, and suitable for tight-spaced areas. It supports different kinds of cameras, it can handle harsh conditions, extreme temperatures, and even a low-speed impact. Moreover, it can be easily integrated with a ground station for mission planning and flight analysis. The average flight time of the drone is 10-15 minutes but the battery in the simulated version does not drop, allowing us to simulate longer flight times, if necessary. Given the above benefits, the Iris Quadcopter is a good choice, and more than enough for our simulation needs.

5.2 Synchronizer

Once the communication through an intermediary has been established, it is time to begin the replication process. The first thing we want to do, is to have the ability to add a second autopilot to our flow. This is possible through the **instance** (-I) command line parameter of Autopilot. The default setting with only one autopilot implies that instance is 0 (-I0). Therefore, by specifying the instance parameter to be 1 (-I1), we are adding another Autopilot instance. From now on, we will refer to our first Autopilot(-I0) as **Autopilot 1**, and to our second autopilot(-I1) as **Autopilot 2**.

The goal of the synchronizer, which connects both Autopilots to the same Gazebo environment (aka same drone), is to make sure that the 2 autopilot instances produce the same motor commands throughout the whole duration of the flight, assuming no external factors affect the Autopilots. That way, if one or both Autopilots malfunction (due to malicious intent or not), we will be immediately notified due to the discrepancy in the motor commands. In a high level, the synchronizing loop will work like this:

1. Autopilot 1 sends a packet to the synchronizer.
2. Autopilot 2 sends a packet to the synchronizer.
3. The synchronizer compares the motor commands that the two packets contain (for all the servos).
4. The synchronizer then sends one of the two packages (if the motor commands are the same, we don't care which) to Gazebo.
5. Gazebo processes the packet and sends the produced sensor data to the synchronizer.
6. The synchronizer fans out the sensor data to both Autopilots.

5.2.1 Implementation Details

The UDP Protocol was once again used, along with the **recvfrom()** and **sendto()** functions. For the 2 Autopilots to send their data, we created 2 sockets in the synchronizer, one in port 9002 and one in port 9012. These numbers were not assigned randomly, as in AP_HAL_SITL library, we can see that:

```
case 'I': {
    _instance = atoi(gopt.optarg);
    if (simulator_port_out == SIM_OUT_PORT) {
        simulator_port_out += _instance * 10;
    }
}
```

Since `SIM_OUT_PORT` is 9002 by default and the instance number for Autopilot 2 is 1, we notice that Autopilot 2 will send its packets in port 9012. Generally, we can deduct that for each Autopilot we add, the destination port number increases by 10. Using this information, we connected our synchronizer directly with the two Autopilots. From a lower-level port perspective, the synchronizer loop becomes as follows:

1. Autopilot 1 sends a packet to 9002 (synchronizer).
2. Autopilot 2 sends a packet to 9012 (synchronizer).
3. The synchronizer compares the packets and sends one to Gazebo (9022).
4. Gazebo sends the sensor data back to 9002 (9012 could also be used).
5. The synchronizer sends the sensor data to both Autopilots (port X and port Y). Port X and Y are ephemeral ports that are assigned to the 2 Autopilots by the operating system.

A slight variation of this implementation would be to use a single port (9002) to receive the packets of both autopilots. In that case, the two packets would be queued and would be processed sequentially. A drawback would be that the queue capacity would be cut in half, meaning that only half of the packets of each Autopilot would remain in the queue before they start getting drained. This would matter only in a case of a severely delayed network, where the two Autopilots would not get a response from Gazebo and would have to send their packet multiple times.

Two important checks are made to ensure that we are comparing the correct packets. In order for the comparison to be correct, the two packets must come from a **different port** (comparing packets from different Autopilots) and the two packets must have the same **frame counter** (comparing packets in the same Autopilot state). Therefore, if in a certain synchronizer iteration, the packages we are receiving come either from the same Autopilot, or have different frame counters, we get notified that we are comparing the wrong packets.

Schematically, the synchronizer is presented in 5.2:

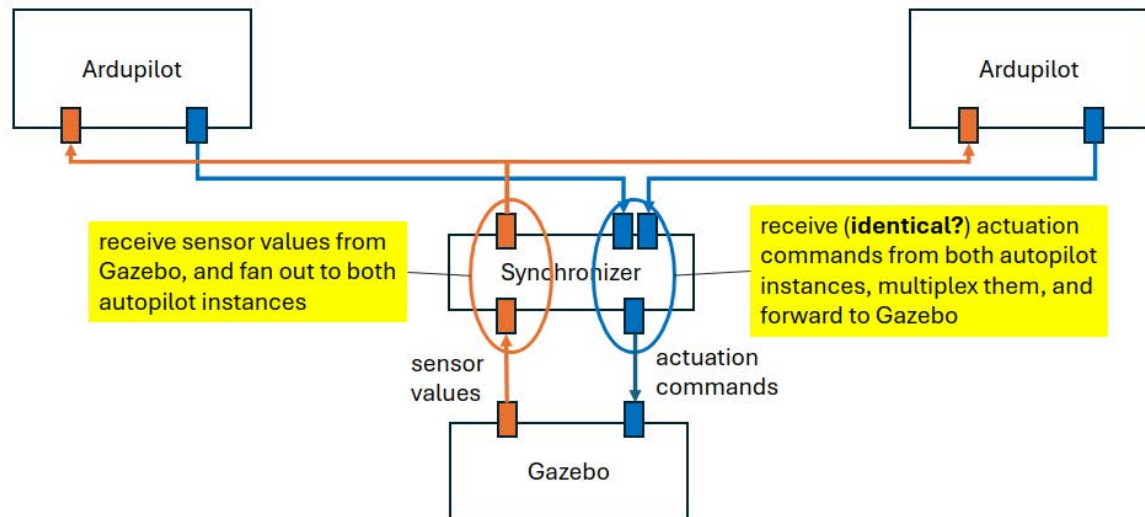


Figure 5.2: The Synchronizer Unit

5.2.2 Program Usage

In order to use the synchronizer as the means of communication:

1. We run the synchronizer program. The program then rests to the first `recvfrom()` function (waiting packet from one Autopilot).
2. We start the Gazebo plugin.
3. We start both Autopilots. In order to start both Autopilots close to simultaneously, we use Tmux (for now). Tmux is a terminal multiplexer that supports multiple terminal sessions in a single window. Pane synchronization can be enabled, and commands can be run simultaneously across all terminal sessions.
4. We keep giving commands to the Autopilots, with synchronization kept enabled.

5.2.3 Basic Synchronizer Experiment

We conducted the forwarder experiment to the synchronizer, but this time our purpose was to get an overview of the motor commands' behavior. Each time the motor commands we were comparing were different, we wrote their values to a log file. Contrary to what we were expecting, the motor commands of the two Autopilots were **different**, even though we were comparing the correct packets (the checks mentioned earlier were passing).

From the log file, we could obtain some general information regarding the motor commands. Their range is between 1000 and 2000 PWM. 1000 means the drone is immobilized (not necessarily inactive). Once the drone starts arming, the motors climb to 1100, staying there until another command is given or until the drone disarms. In the latter case, motors drop back to 1000. When takeoff is initiated, motors climb from 1100, to approximately 1400. During takeoff, reposition, and hovering states, the motors seem to oscillate from 1400 to 1700. We did not encounter values from 1800 or above, as that would indicate that either the drone is approaching its limits, or that there is a possible malfunction.

Given that the phases of the experiment were short, and therefore not discrete enough to extract any valuable information, we decided to conduct a bigger experiment, to study this unexpected behavior further.

5.2.4 Extensive Synchronizer Experiment

This experiment was designed to be longer and with discrete phases, in order to examine the behavior of the motor commands in each phase. The experiment was the following:

```
arm();  
takeoff();  
goto wp();  
loiter();  
goto home();  
land();  
disarm();
```

The takeoff altitude was 2 meters. The waypoint was approximately 40 seconds away from the drone's home position and the loitering time was 30 seconds.

Apart from logging the different motor commands, we also gathered several statistical information for each run of the experiment. Specifically, we logged the **average error**, the **minimum error**, the **maximum error** and the **variance** of the error, for all four servos, for a certain run. We define error as the absolute value of the difference of the motor commands in a certain packet comparison. By logging the error for every packet exchanged during a run, and the number of packets exchanged, we can obtain the metrics referenced above.

Furthermore, in this experiment, we logged the **average error** for every second of a run, for every servo. This is useful because we can see how the error fluctuates during a run. Initially, we created a stats function which logged the current second, and the average error for that second. In order to achieve asynchronous behavior (calling the function every 1 second), we passed the stats function as parameter to another function, which was calling the stats function in a thread independent of the main thread (detached thread). Providing a snippet:

```
std::thread([after, task]() {  
    while (true) {  
        std::this_thread::sleep_for(std::chrono::milliseconds(after));  
        task(); // Execute the task  
    }  
}).detach();
```

As we can see, the detached thread is responsible for two things: **Sleeping** for the specified amount of time (in our case 1 second) and **executing** the task given (in our case the stats function).

Once we had our data, we created a program which plotted our average error, so we could analyze it visually. Four plots are produced, one for each servo. The x axis contains the time in seconds, as our run progresses, with labels being placed on the start of every phase of the run (arm, takeoff etc.). The y axis contains the average error for every second of the run. We ran the experiment 5 times, however the general pattern of the plots falls into one of two categories. An indicative run of each of the two categories is presented in 5.3 and 5.4:

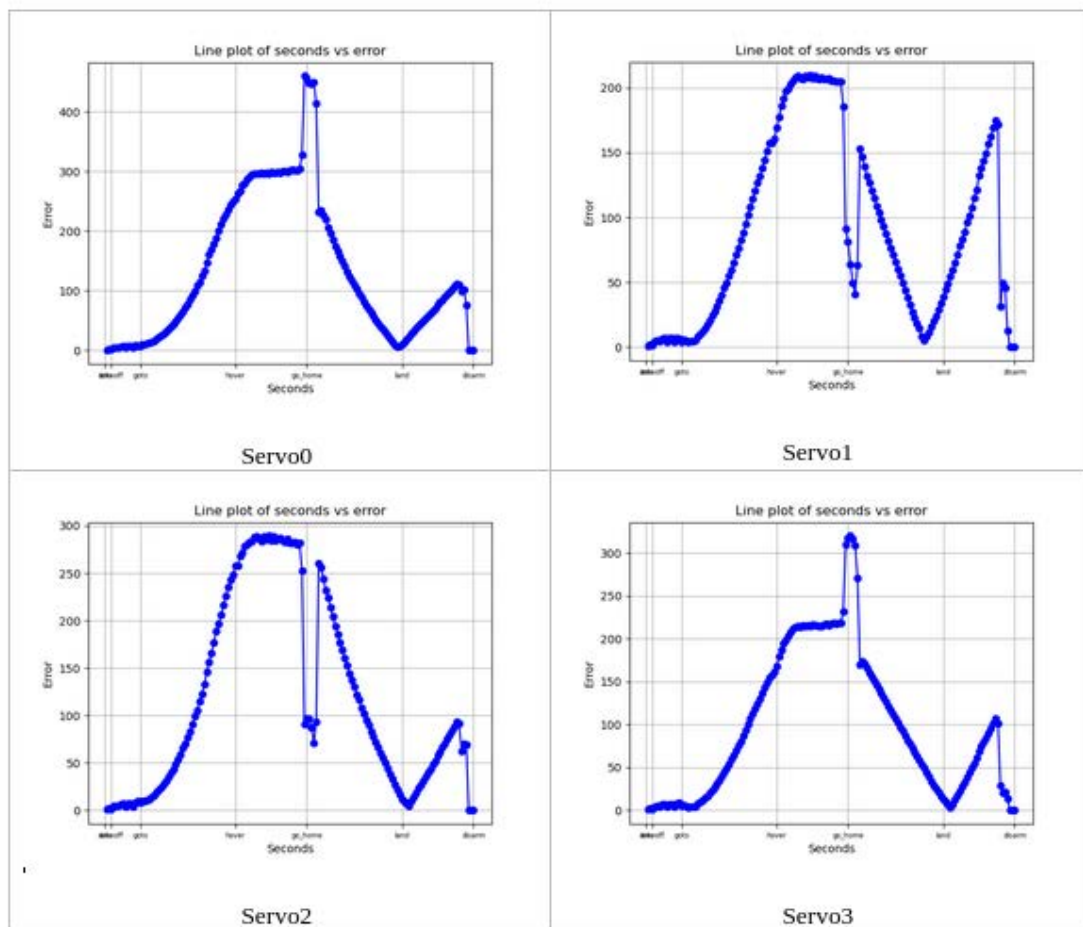


Figure 5.3: First pattern of motor value discrepancies

Diff\servo	Avg	Min	Max	Std Dev
0	71.1884	1.0	540.0	144.68
1	47.3584	1.0	303.0	87.27
2	63.2165	1.0	400.0	124.44
3	52.6916	1.0	357.0	101.16

Table 5.1: Statistical results of first pattern

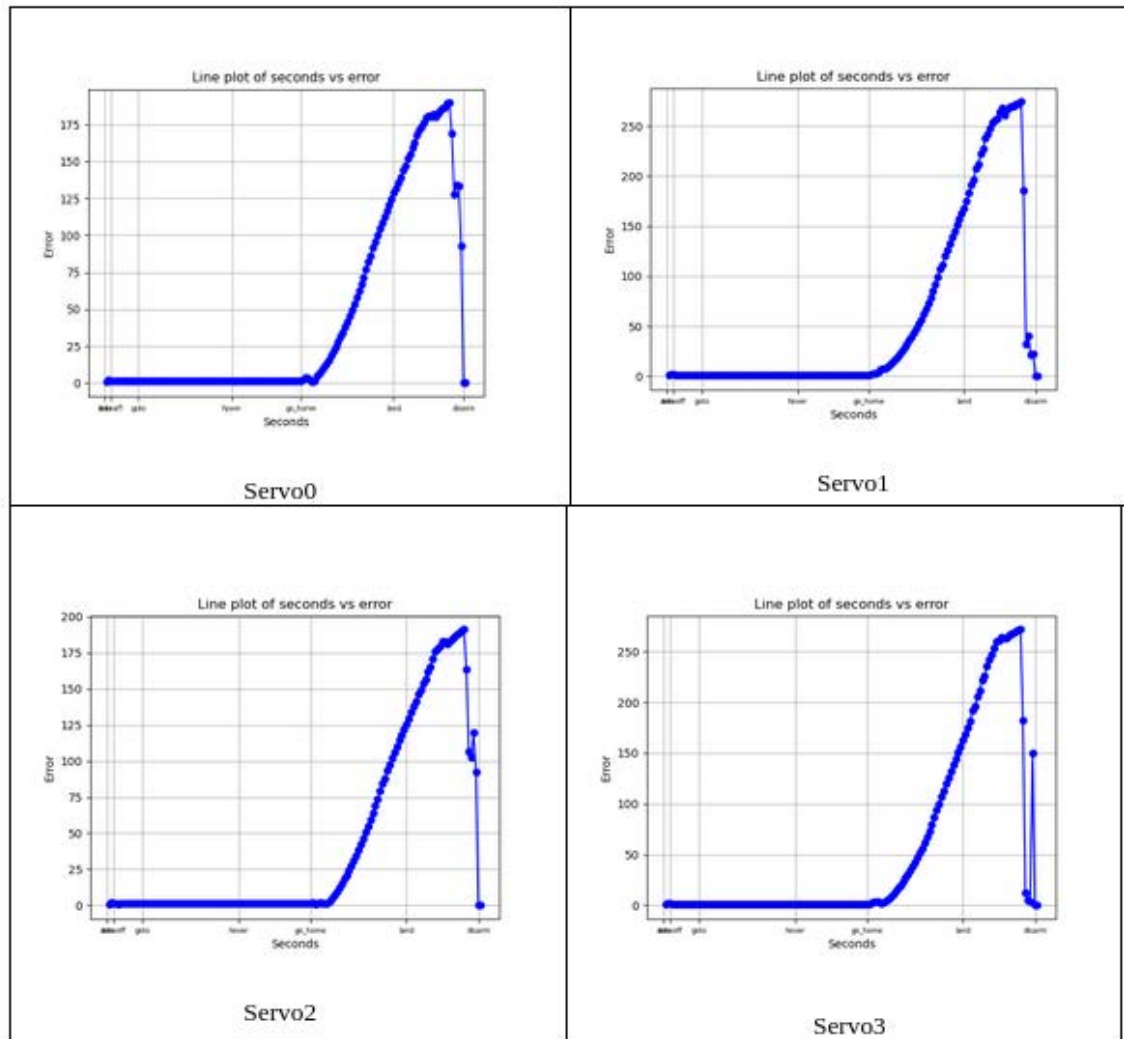


Figure 5.4: Second pattern of motor value discrepancies

Diff\servo	Avg	Min	Max	Std Dev
0	71.1884	1.0	540.0	144.68
1	47.3584	1.0	303.0	87.27
2	63.2165	1.0	400.0	124.44
3	52.6916	1.0	357.0	101.16

Table 5.2: Statistical results of second pattern

As we can observe, the error can either appear in the beginning of a run (after takeoff) or at some point during the run. When the error appears early, it contains significant fluctuations, contrary to the error appearing at a later time. Therefore, despite our experiment being the same, the error behavior is not the same during all runs of the experiment.

5.3 SITL Behavior

Since the behavior of the synchronizer was somewhat randomized, we decided to check whether SITL contains a factor that randomized its behavior. So, our end goal was to see if SITL is deterministic or not. If SITL was, in fact, proven to be non-deterministic then that would explain the discrepancy in the motor commands. We checked the behavior of SITL both in theory (examining the code) and in practice (by running an investigative experiment).

5.3.1 SITL Code Examination

We first checked whether noise was added to the sensor data once they are received but that wasn't the case. Then, we located the functions in the SITL folder that contained randomized variables. Many of them were disabled by default in the configuration settings. For example, the function **gps_jamming()** introduces randomization, but it is turned off by default. For the rest of the functions, we replaced the variables that introduced randomization with something deterministic (a specific number) and we ran our experiment again. We didn't notice any changes and the randomized elements we encountered do not affect the Ardupilot – Gazebo connection. Therefore, a non-deterministic factor wasn't found through this approach.

5.3.2 SITL Behavior Experiment

Our experiment was the following:

1. Run a simple (arm, takeoff,hover) experiment with 1 Autopilot.
2. Run the same experiment again, with 1 Autopilot.
3. If we get different motor commands, SITL is non-deterministic.

Since 1 Autopilot is enough for this experiment, the forwarder was used. Our plan of execution is depicted through 5.5:

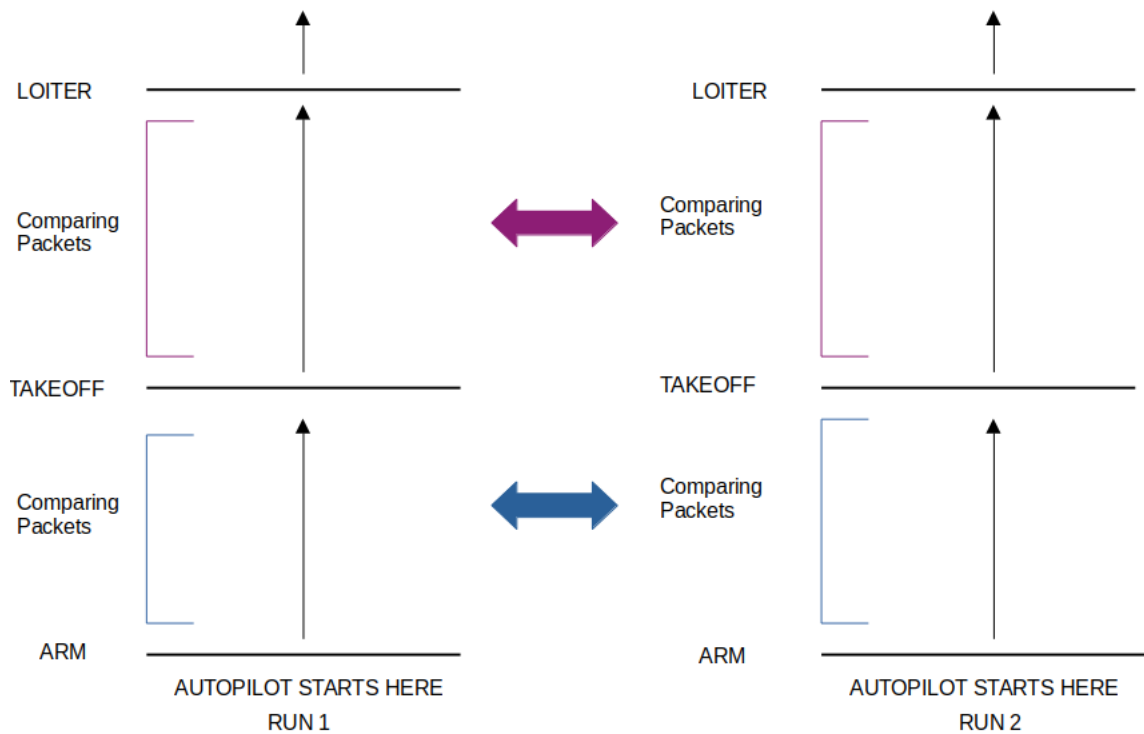


Figure 5.5: Plan layout for the packet comparisons

First, we compare the arm packets of the two Autopilots between the two runs. Then, we compare the packets **after** the takeoff command has been issued, and we ignore the packets sent between arm and takeoff. We ignore these packets because they are inherently the same (value 1100) and because their quantity differs between the two runs, creating a mismatch in overall packet comparison.

After we had collected our data, we extracted the error for **each packet** (not the average error) by subtracting the values of the corresponding packets of the two runs. Subsequently, we created four plots, one for each servo, to observe the behavior of the error. The x axis contains the packet number, as our run progresses, with a label being present every 1000 packets. We kept informaton for 6000 packets, which corresponds to the drone arming, taking off, and hovering for a little. The y axis contains the error for every packet comparison. Therefore, in 5.6 we have:

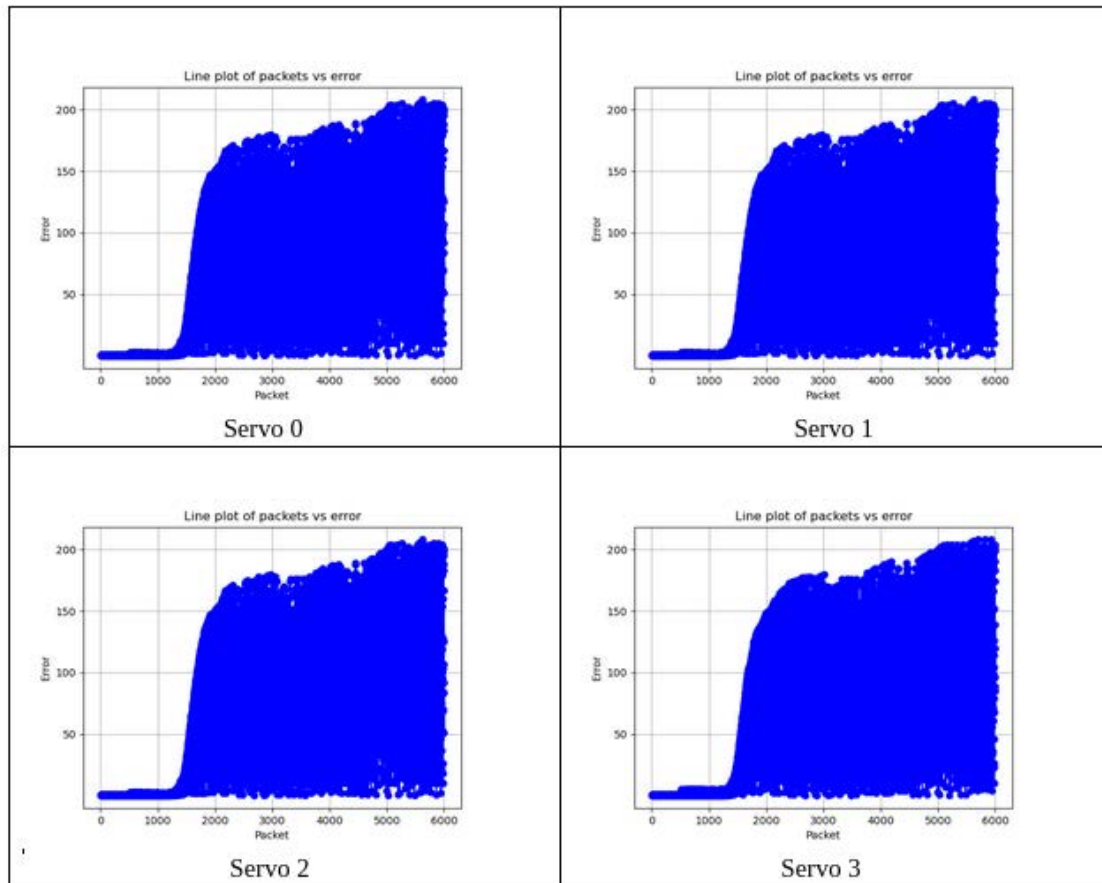


Figure 5.6: Results of the experiment

From the plots, we can see that the error appears in most of the packets, and the error is ranging from 0 to 200. Since the error is present in most of the packets, someone could deduce that SITL is non-deterministic. However, a crucial piece of information prevents us from drawing such a conclusion. After some research, we found out that the Autopilot begins running **from the moment** the SITL-Gazebo connection opens (and therefore before giving the arm command). Essentially, this means that, since the arm commands between the two runs are given with a certain time difference from the time the Autopilot starts in each run, the two Autopilots are in a **different** state when the arm command is issued. Therefore, the arm commands (and consequently the takeoff commands), will change in a different way as the run moves forward. Schematically, it can be depicted as in 5.7:

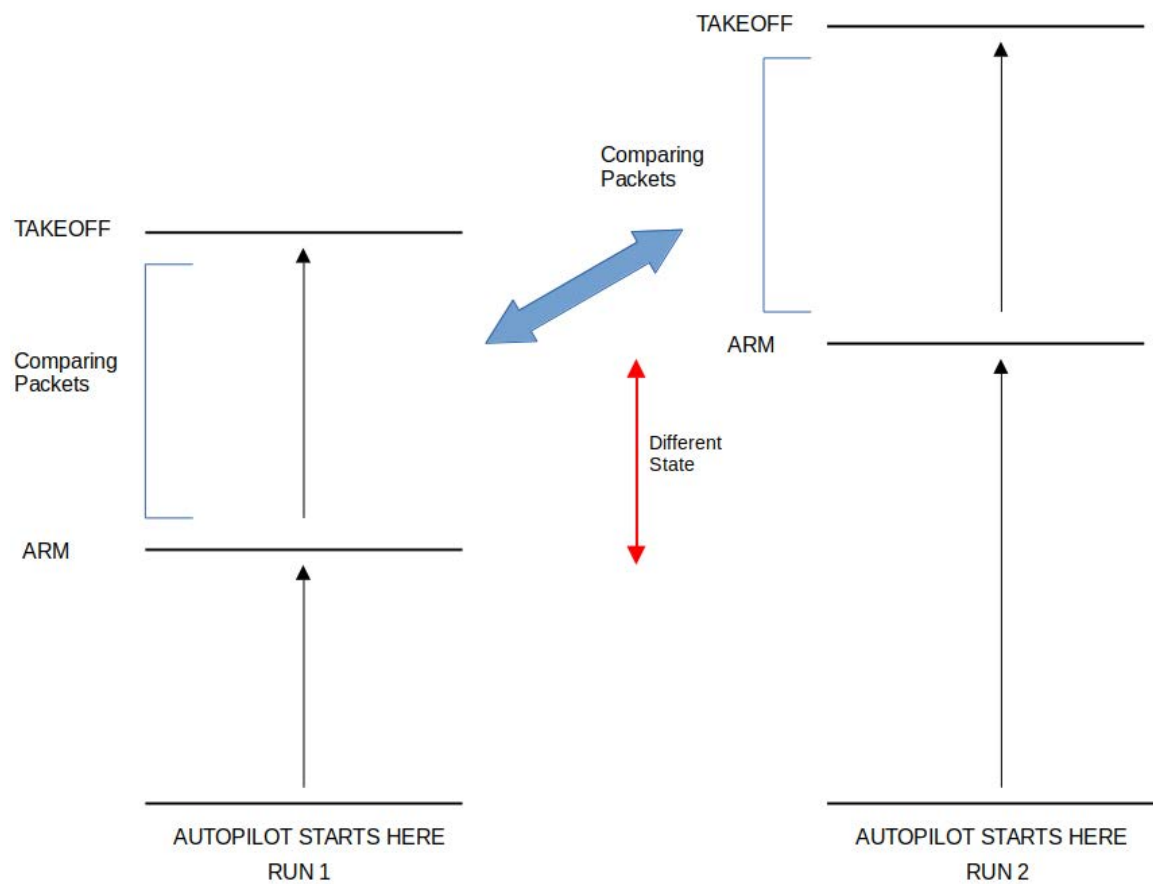


Figure 5.7: True representation of the packet comparisons

Through the above experiment, we have **proven** that we can't prove experimentally whether SITL is deterministic or not, as long as we can't specify the exact time (and even at which exact packet) the arm command can be given. Through the investigative means of 6.3.1, we are led to believe that SITL is **deterministic**.

5.4 Tmux Propagation Delay

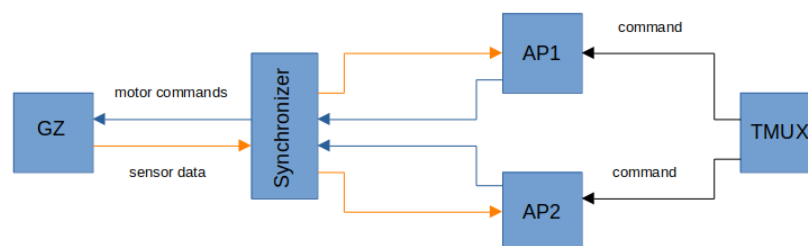


Figure 5.8: Ardupilot-Gazebo flow with Tmux

We provided the commands to the drone so far, using Tmux. However, an observation was made, that Tmux has a propagation delay of a couple of milliseconds. Let's examine a scenario where both drones are in state A, and a command (e.g. relocation command) forces both drones to a state B. Because of the propagation delay, one drone will receive the command a couple of packets quicker than the other. Therefore, during that delay, the quicker drone will send motor commands in order to get to state B, while the slower drone, that hasn't received the command yet, will send motor commands that correspond to state A. This mismatch could be responsible for the discrepancy that we saw in our earlier experiments.

5.4.1 Automated Experiment

For this experiment, we decided to avoid interference with Tmux as much as possible and created an automated mission using the Mavlink file format. Essentially, a Mavlink file is a simple text file that contains drone commands in the form of Mavlink messages [30]. We can then run the mission through Autopilot's **mode auto**. The general form of a Mavlink command is the following [31]:

```
seq starting frame code P1 P2 P3 P4 P5 P6 P7 1
```

In more detail:

seq: The sequence number of the command. It starts at 0 and for every command we add, it is incremented by 1.

starting: This shows what command is the starting command for the mission. Usually 1 when seq is 0, and 0 for all other values of seq.

frame: Specifies the coordinate frame.

code: A unique identifier used to identify the command.

P1-P7: Command parameters. 7 parameters are the maximum a command can have. Some of them can be empty.

1: Signifies the end of a command.

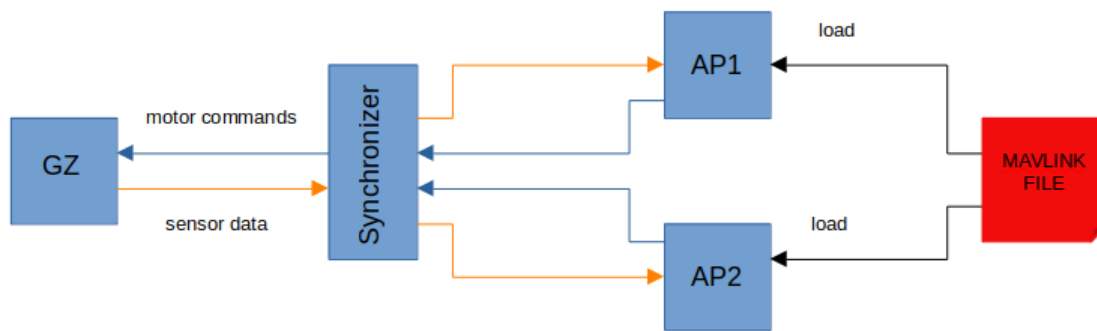


Figure 5.9: Ardupilot-Gazebo flow after Mavlink file upload

We created a Mavlink file reconstructing our initial synchronizer mission, without incorporating the arm command (the mission became unsupported when arm was integrated into the mission file). Tmux was still used only for the arm command. If the propagation delay during arm is small enough, the experimented is completed error-free, as also indicated by the following plots in 5.10:

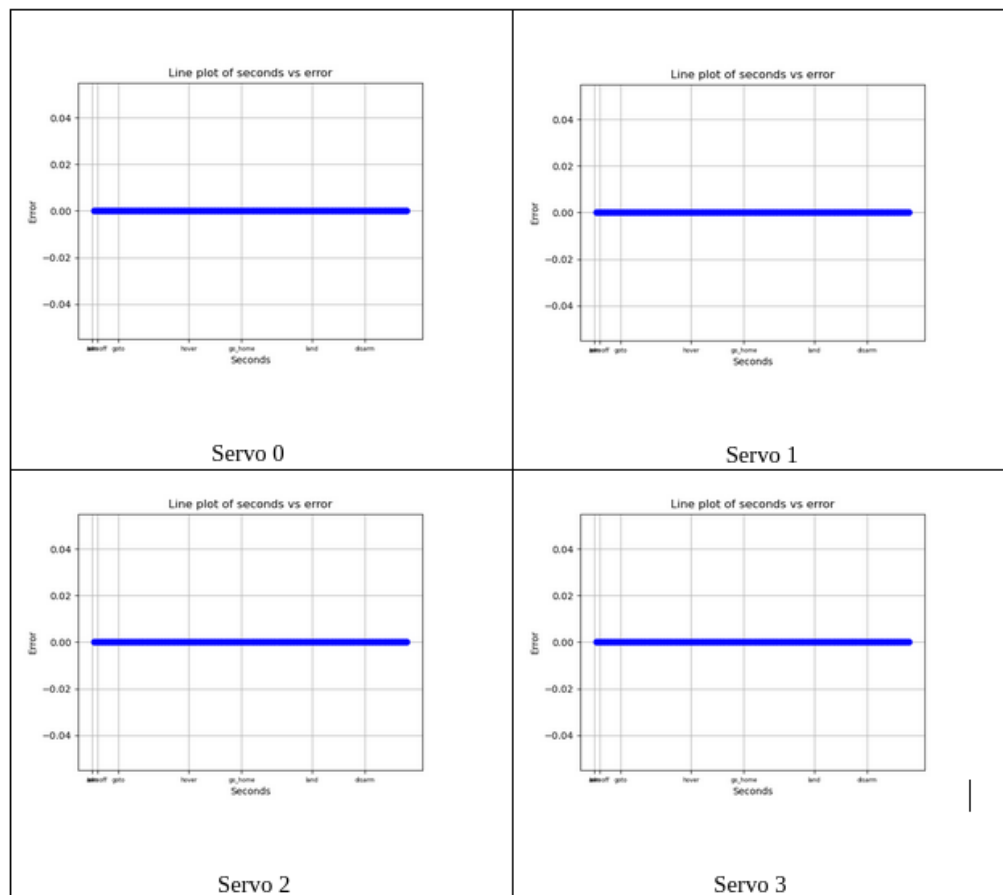


Figure 5.10: Motor values with zero discrepancy

The plots verify our original expectations, and the motor commands produced by the two Autopilots are **the same** throughout the duration of the experiment.

5.5 PyMavlink Library

Since arm couldn't be given as a mission item, we tried to send it through the Pymavlink library [32]. Pymavlink is a low-level Python library that is used to send Mavlink commands to a Ground Control Station and to other Mavlink applications.

We sent the arm command using the COMMAND_LONG message. Its format is similar to the format of a Mavlink command [33]:

target_system target_component code confirmation P1 P2 P3 P4 P5 P6 P7

Namely:

target_system: The system to execute the command

target_component: The component to execute the command

code: Command ID (as previously)

confirmation: Indicator whether the command is sent for the first time.

P1-P7: Command parameters (as previously).

We created a program utilizing pymavlink, which sends the arm commands to the two drones concurrently. The commands were sent to port 14550 and 14560 for Autopilot 1 and Autopilot 2 respectively, as the Ardupilot convention indicates that $\text{port}[i] = \text{port}[i-1] + 10$ (just like we mentioned earlier). For the rest of the commands, we use the Mavlink file that we created.

With our current setup, the synchronizer is significantly less dependent on the propagation delay, and we can achieve results like those of 5.10 much more frequently.

Chapter 6

Tolerating a Faulty Autopilot Replica

6.1 Crash Tolerance

Once the replication is complete, the next step is to develop a simple Crash Tolerance System, which can handle the case of one Autopilot crashing. If one Autopilot crashes mid-flight, the second, functional Autopilot, should take over, so that the mission can continue seamlessly. This is achieved by using a timeout mechanism. Specifically, after the first packet has been sent, (so that we are sure that both Autopilots have started) we introduce a small socket timeout to both sockets that are used for receiving the motor commands from the Autopilots. The timeout should be big enough to justify small delays in the network, but also small enough, so that the timing of the control loop is not affected. If a crash is detected, we close the socket responsible for the crashed Autopilot. Then, we make sure that the control loop works only with the functional Autopilot, bypassing the parts that involve the closed socket. Essentially, our synchronizer seamlessly transitions to a forwarder.

We decided to test crash tolerance in 2 different ways, locally, and over a network. Locally, we ran the 2 Autopilot instances and at some point during the mission, we killed one of the two Autopilot processes. After that, the other Autopilot was successfully taking over, and the mission was completed successfully with one Autopilot.

For our network setup, we used two different machines. Gazebo, the synchronizer, and one of the two Autopilot instances were running on one machine (local), while the second instance was running on the other machine (remote). Our plan of action was to disconnect the

remote Autopilot from the Wi-Fi and see if it was successfully detected as a crash from our module. Indeed, we disconnected the remote device without closing the remote Autopilot's process, and the crash was successfully detected.

6.2 Noise Tolerance

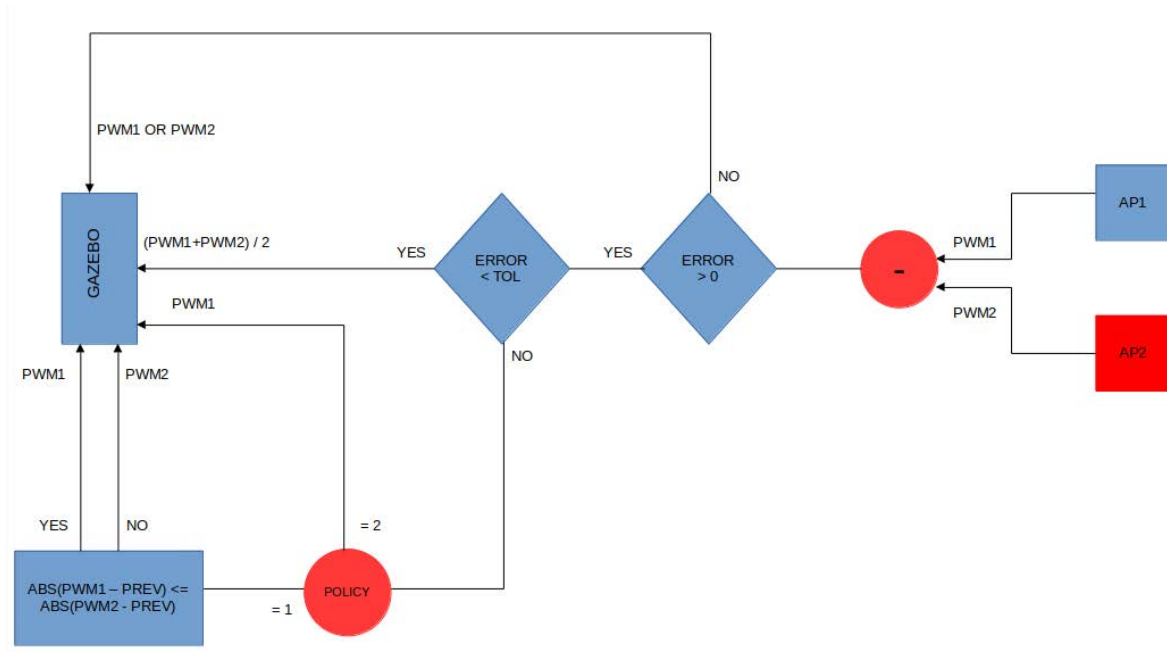


Figure 6.1: Noise-Tolerant Synchronizer logic

As a next step, we decided to analyze the tolerance capabilities and the behavior of the system, if noise was introduced to the second Autopilot. We conducted an experiment that involved many different parameters along many different runs, and we examined the average error of each run, the best policy for each combination of parameters, as well as the visual representation of the drone in Gazebo to watch for some possible flight anomalies.

The parameters that we used are the following:

Tolerance. The upper bound of error discrepancy that we can tolerate. After that, a policy that we have specified takes over. While we are under the tolerance limit, the motor command that we send to Gazebo is the average of the two motor commands that the Autopilots have produced. The values of tolerance that we chose for the experiment are 30 and 60.

Policy. Once the tolerance limit has been surpassed, one of our two policies takes over. Policy 1 states that the motor command that we send to Gazebo is the one that is closer to the previous value that we sent to Gazebo. Policy 2 states that the motor command that we send to Gazebo is the motor command that the first Autopilot (not affected by the noise) produced. The values of policy for this experiment are 1 and 2.

Noise Level. Noise level defines how strong the noise that we apply, will be. Ranging from 0 to 1, we decided to use 3 different noise levels, 0.2 (low), 0.4 (medium), and 0.7 (high). We can see where the noise was introduced (red line) in each flight phase in 6.2.

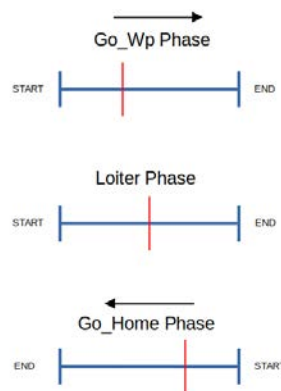


Figure 6.2: Noise Introduction in the Flight Phases (Red Line)

Flight Phase. We applied the noise at different stages of the flight in each run. Specifically, our examined flight stages were 3, going to waypoint (go_wp), loitering (loiter), and returning home (go_home).

Noise is added in the angular velocity vector and in the acceleration vector. This change affects the motor commands that are produced for all 4 motors. The noise values are produced using normal (Gaussian) distribution and the range of the noise depends on the noise level that we selected.

We tested every combination of the above parameters, resulting in 36 runs (2 policies · 2 tolerance_values · 3 noise_levels · 3 phases). For each run, we plotted the average error for every servo during the run, along with statistics (average,max,min,variation) for each servo. First, we present a table that shows the average error (average across all servos) of each run, sorted from lowest to highest error:

Table 6.1: Results of experiment sorted by ascending error

Policy	Tolerance	Noise	Phase	Average Error
2	30	0.2	Go_home	3.83
1	30	0.4	Go_home	16.54
1	30	0.2	Go_home	16.77
2	60	0.2	Go_home	21.16
1	60	0.2	Go_home	22.84
2	60	0.4	Go_home	36.31
1	60	0.4	Go_home	36.87
2	30	0.7	Go_home	40.32
1	60	0.7	Go_home	42.71
1	60	0.2	Go_wp	43.27
2	60	0.2	Go_wp	49.53
1	30	0.7	Go_home	51.16
2	30	0.2	Go_wp	52.54
1	30	0.2	Go_wp	56.52
2	60	0.4	Go_wp	56.77
2	30	0.4	Go_home	56.82
2	60	0.4	Loiter	57.47
1	30	0.2	Loiter	59.54
1	60	0.4	Go_wp	64.13
2	60	0.7	Go_home	66.71
2	60	0.2	Loiter	72.55
1	30	0.4	Go_wp	72.57
2	30	0.4	Go_wp	80.03
1	30	0.4	Loiter	80.04
1	60	0.4	Loiter	81.32
1	30	0.7	Go_wp	83.53
1	60	0.7	Loiter	84.52
2	30	0.4	Loiter	84.74
2	60	0.7	Go_wp	85.5

Policy	Tolerance	Noise	Phase	Average Error
1	60	0.7	Go_wp	90.99
2	30	0.2	Loiter	96.04
1	60	0.2	Loiter	97.57
2	60	0.7	Loiter	103.36
1	30	0.7	Loiter	104.89
2	30	0.7	Go_wp	109.62
2	30	0.7	Loiter	114.88

As we can see, the **time** in which the noise was introduced affects the error. Specifically, noise introduced at the go_home phase leads to lower error rates than the other phases. This is logical, because the go_home phase comes after the other two phases and therefore the drone is under the influence of noise for less time. However, **time** is not the determining factor, because as we can see, the loiter phase leads to higher error than the go_wp phase, even though noise is applied to the drone for less time. Therefore, the **flight phase** at which the noise was introduced, is also of critical importance.

Next, we examined which policy results in the least error for each combination of noise, tolerance and flight phase. The results are shown in Table 6.2.

Out of the 18 different combinations, policy 1 was considered better **9 times** and policy 2 was also considered better **9 times**. This shows that the two policies are overall equally powerful in regard to system tolerance. However, we can see that policy 1 is performing better with tolerance 30 and policy 2 is performing better with tolerance 60. Therefore, we can pair them accordingly to achieve better system robustness.

Tolerance	Noise	Phase	Best Policy	Error Between Policies
30	0.2	Go_wp	2	3.98
30	0.2	Loiter	1	36.5
30	0.2	Go_home	2	12.94
30	0.4	Go_wp	1	7.46
30	0.4	Loiter	1	4.7
30	0.4	Go_home	1	40.28
30	0.7	Go_wp	1	26.09
30	0.7	Loiter	1	9.99
30	0.7	Go_home	2	10.84
60	0.2	Go_wp	1	6.26
60	0.2	Loiter	2	25.02
60	0.2	Go_home	2	1.68
60	0.4	Go_wp	2	7.36
60	0.4	Loiter	2	23.85
60	0.4	Go_home	2	0.56
60	0.7	Go_wp	2	5.49
60	0.7	Loiter	1	18.84
60	0.7	Go_home	1	24

Table 6.2: Best policy for each combination based on average error

6.3 Servo Behavior

From the plots that we have produced, we can extract some information regarding the behavior of the motors. Servo behavior is different, depending on the flight phase parameter. It is not affected as much by the other 3 parameters. Specifically, in the `go_home` phase, we can see across the plots that all 4 servos behave the same. An indicative example of a run follows in 6.3:

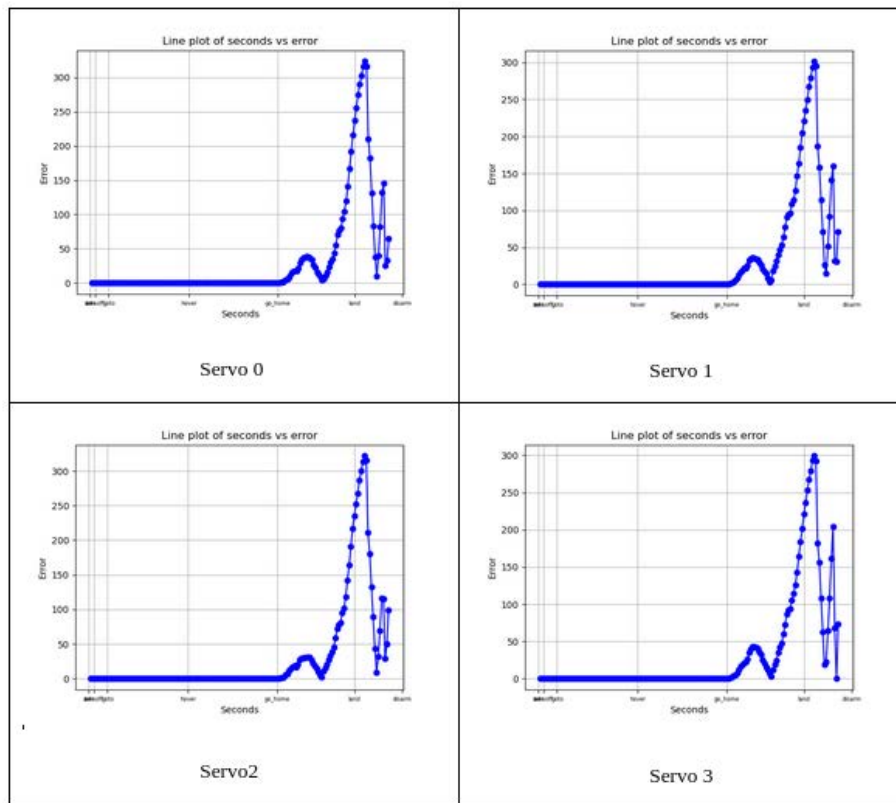


Figure 6.3: Servo Behavior in the Go_Home phase

As we can see, the plots are really similar to each other. This can be explained in the following way. From the moment the noise is introduced, the 4 motors are giving out the exact same thrust, since the drone just goes straight and then lands. Therefore, the relative error stays the same in the 4 motors.

In the **Loiter** phase, things are a little different. Through the plots we noticed that servos behave the same in 2 groups. Specifically, servos 0 and 2 behave the same, and servos 1 and 3 behave the same. By again presenting an indicative example of a run, the pattern looks like 6.4:

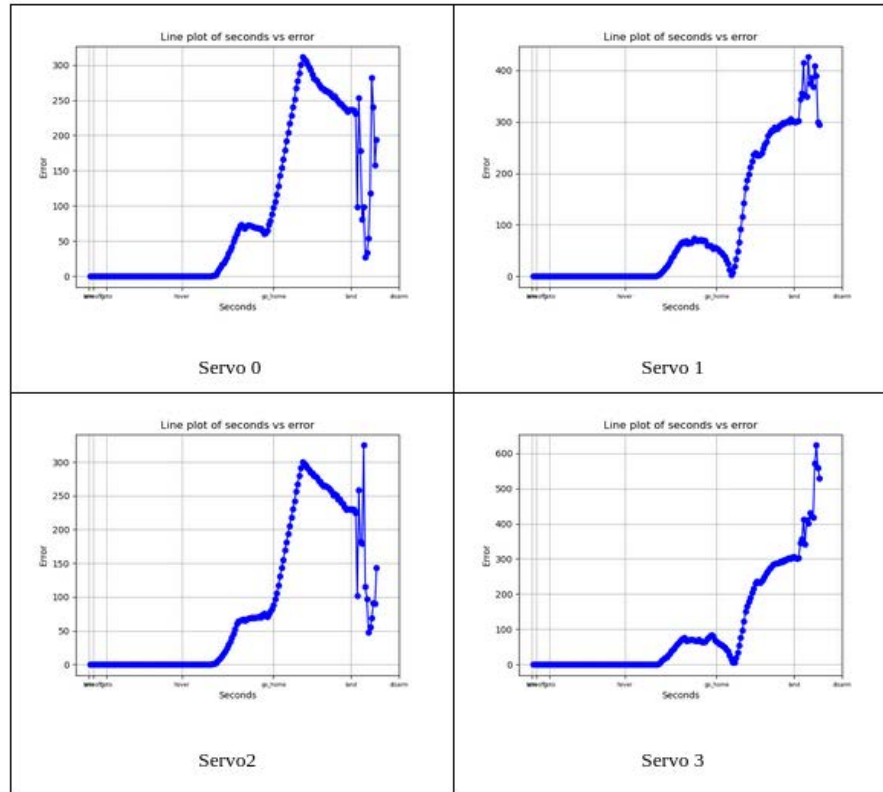


Figure 6.4: Servo Behavior in the Loiter phase

Like we described, we can see that the pattern is followed in pairs (0 and 2, 1 and 3). This can also be explained. The noise is introduced in the middle of the Loiter phase. At some point, the drone has to change its state from static (loitering), to mobile (heading home). So, its pitch must be increased, and the drone leans forward. Through that leaning movement, more thrust is generated by the front motors (0 and 2) than the back motors (1 and 3), therefore leading to a different relative error between the motor pairs.

In the **Go_Wp** phase, the pattern followed is mainly similar to that of the Loiter phase. The noise is introduced in the middle of the Go_Wp phase, so the drone has to lean backwards to stop and then loiter (different thrust between the front and back motors) and then follows the path we described in the previous paragraph, essentially creating once again a distinction between the front and back motors.

6.4 Visualization of the results

We created some plots to have a better view of the results produced by the experiment. Namely, on the x-axis we have the 3 different noise levels, and on the y-axis, we have the average error that corresponds to each flight phase parameter, across the 3 noise levels. We have:

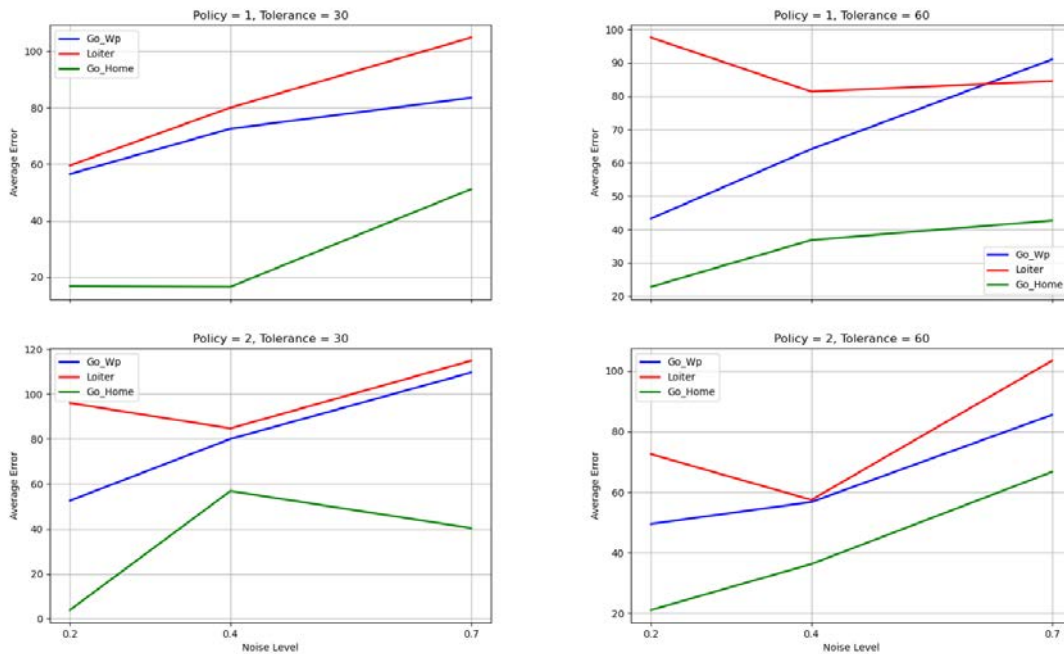


Figure 6.5: Average error of the flight phases in respect to noise level

These plots point out the important role of the noise level, because, apart from a few exceptions, the average error increases as the noise level increases. Also, noise introduced in **Loiter** phase produces the biggest error of all the phases we tested, which was also pointed out in table 6.1.

Furthermore, we plotted the average error of a given flight phase, across the 3 different noise levels, for the different combinations of policy and tolerance:

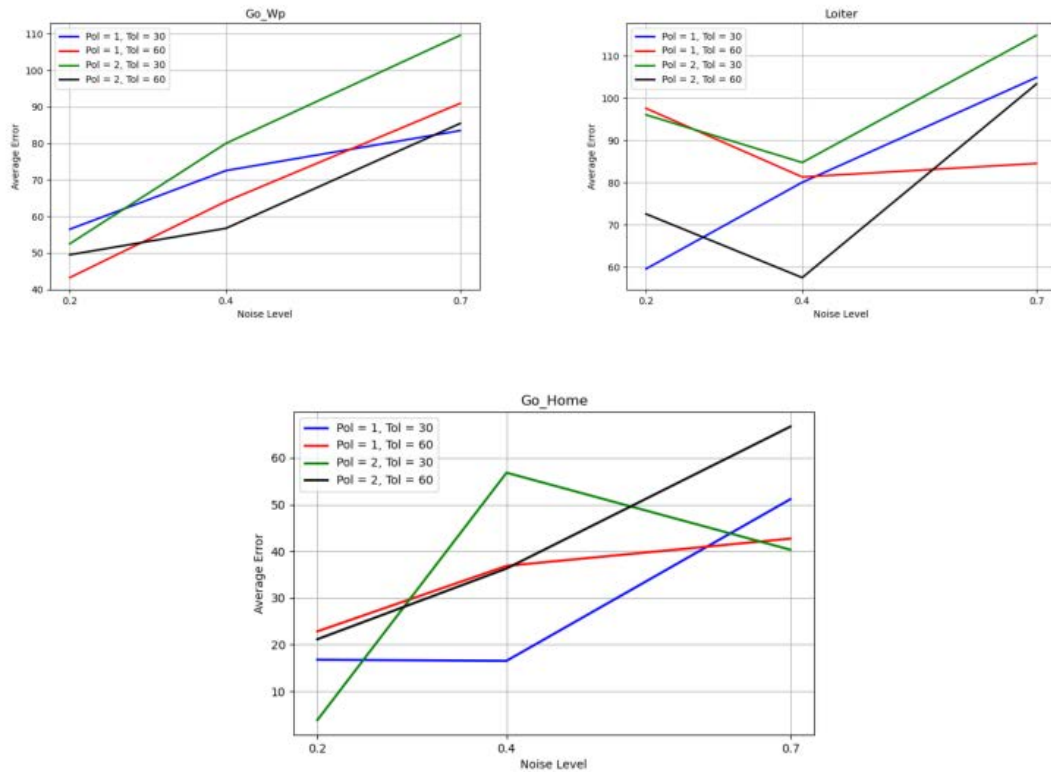


Figure 6.6: Average error of policy and tolerance combination in respect to noise level

We can see that the combination of policy and tolerance reacts differently under different noise levels, however, based on the plots, the most favorable combination for each flight phase is:

Go_Wp phase: Policy = **2**, Tolerance = **60**

Loiter phase: Policy = **2**, Tolerance = **60**

Go_Home phase: (Policy = **1**, Tolerance = **30**) or (Policy = **2**, Tolerance = **30**) (Tied Performance)

6.5 Transient errors

So far, the noise introduced in our experiments was permanent, which means the drone was constantly under the influence of noise (from the moment that noise was introduced, until the end of the mission). In this section, we investigate errors that appear periodically.

We thought of 3 scenarios, that deal with low, medium, and high noise injection frequency:

Low: Noise appears for 2000 frames, every 10000 frames.

Medium: Noise appears for 4000 frames, every 10000 frames.

High: Noise appears for 7000 frames, every 10000 frames.

We decided to keep the noise level steady (0.4), to focus on what happens when the noise frequency changes. The noise starts appearing during the Go_Wp phase, in the moment indicated by the red line in Go_Wp phase in 6.2.

We tested transient errors in 12 runs (3 frequencies * 2 tolerance values * 2 policies). We will perform a similar analysis as the one we did for permanent errors. In 6.3 we present a table with the runs, sorted from lowest to highest error. We can see that, as the frequency of the noise increases, the error also increases. This is logical, because a higher frequency means that noise is applied to the drone for more time during the overall mission. Furthermore, we can see that both tolerance values are equally powerful, as they are equally scattered between the two halves of the array.

We continue by showing the best policy for each combination of frequency and tolerance in 6.4. Through this table, we can see that Policy 2 is slightly better than Policy 1, and we can also observe that Policy 2 is always better than Policy 1 when tolerance is 30.

Regarding the servo behavior, we noticed a similar behavior as the Go_Wp phase in the permanent errors. Namely, the servos 0 and 2 followed a different error pattern than servos 1 and 3, because the thrust applied to each pair of motors is different, as we have mentioned earlier.

Next, we plotted the error of each combination of policy and tolerance, as the frequency changes in 6.7:

Frequency	Policy	Tolerance	Error
Low	2	30	32.81
Low	1	60	48.60
Medium	2	30	56.42
Low	1	30	61.27
Low	2	60	61.88
Medium	1	60	66.34
Medium	2	60	72.07
Medium	1	30	73.87
High	2	30	78.63
High	2	60	84.80
High	1	60	97.38
High	1	30	99.28

Table 6.3: Results of experiment sorted by ascending error

Frequency	Tolerance	Best Policy	Error Between Policies
Low	30	2	28.46
Low	60	1	13.28
Medium	30	2	17.45
Medium	60	1	5.73
High	30	2	20.65
High	60	2	12.58

Table 6.4: Best policy for each combination based on average error

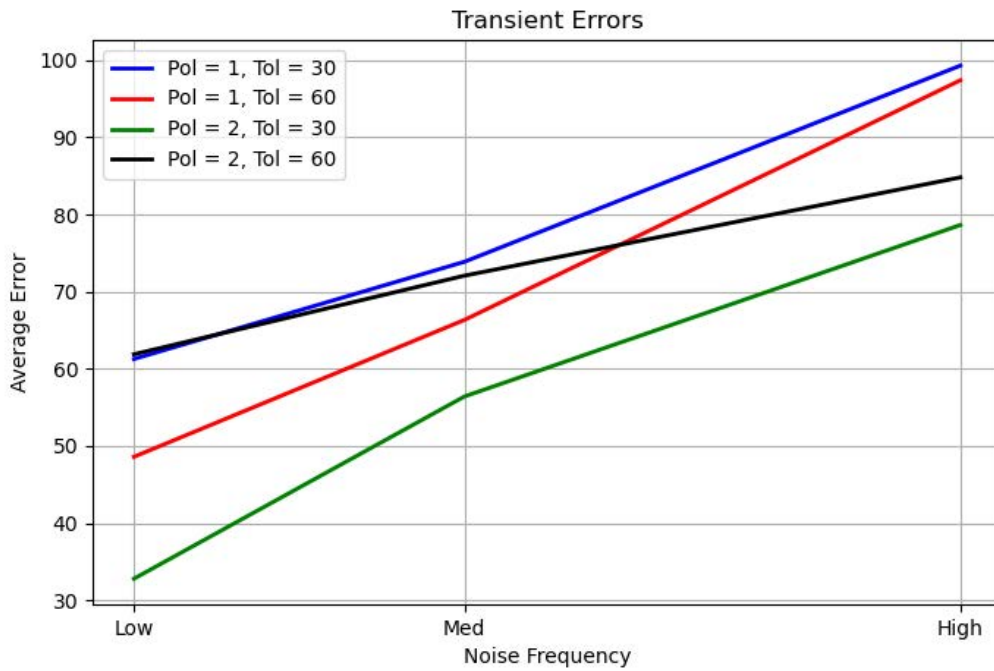


Figure 6.7: Average error of policy and tolerance combination in respect to noise level

Through the above diagram, we can see that indeed, the error increases as the noise frequency increases, in all cases.

Also, we can observe that the best combination of policy and tolerance for transient errors, is by far **Policy = 2, Tolerance = 30**, as it performs better across all noise frequencies.

6.6 Mission Completion Under Noise

We also monitored for every run, whether the drone managed to execute the mission successfully.

For the permanent errors, the drone completed the mission for all 36 runs of the experiment. This means that our policies and tolerance values can handle malfunction in small missions. We have also plotted indicative trajectories for some of the experiments. But first, let's see the trajectory of the drone without the influence of noise.

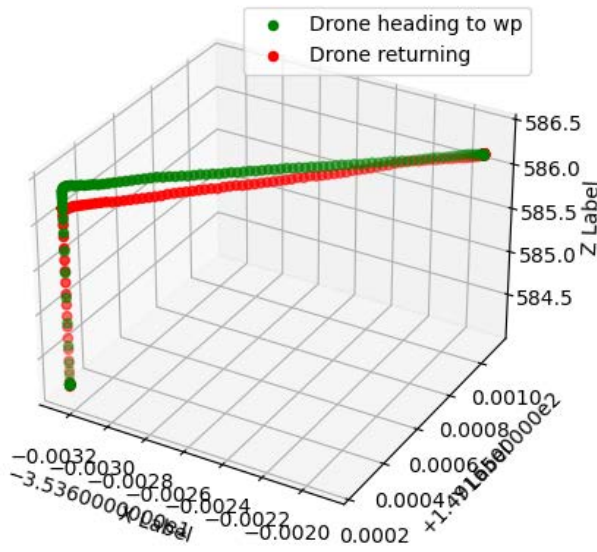
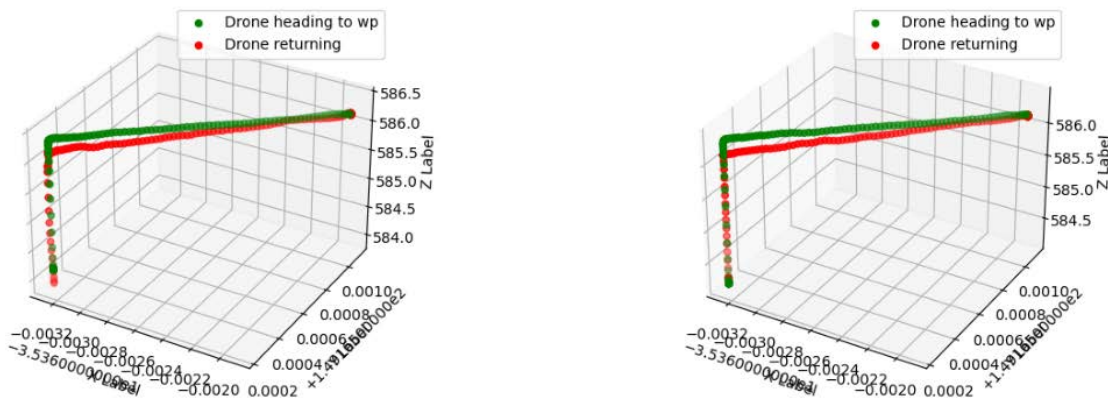


Figure 6.8: Trajectory of the drone without noise

The green line formed, is the trajectory of the drone towards the waypoint, and the red line formed, is the trajectory of the drone towards its home position.

Now, we will present indicative trajectories for low (0.2), medium (0.4), and high (0.7) noise level, in 6.9, 6.10, and 6.11 respectively:

We can see that the trajectories match the non-noisy trajectory (except really small deviations), so we can see the drone completing the mission successfully.

Figure 6.9: **Left:** Pol = 1, Tol = 30, Phase = Go_wp. **Right:** Pol = 2, Tol = 60, Phase = Loiter

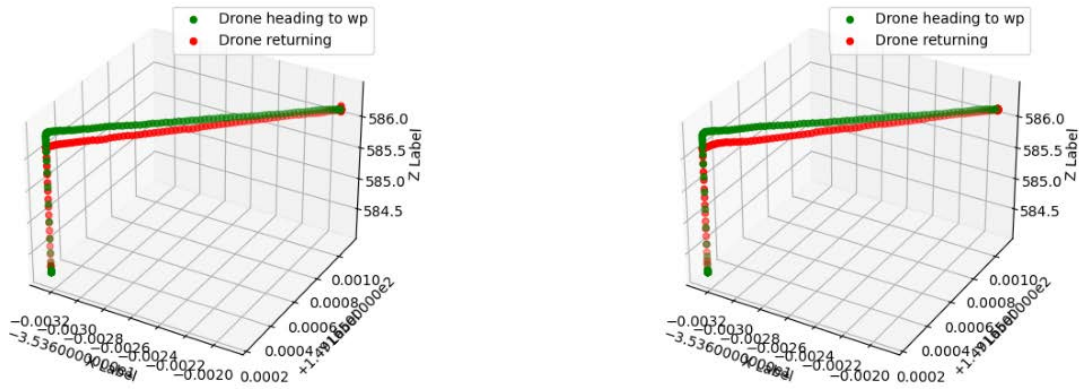


Figure 6.10: **Left:** Pol = 1, Tol = 60, Phase = Loiter. **Right:** Pol = 2, Tol = 30, Phase = Go_home.

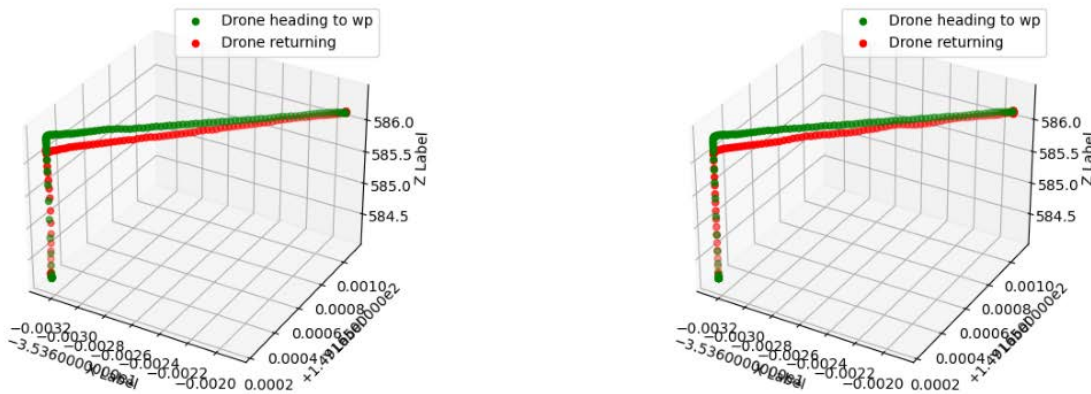


Figure 6.11: **Left:** Pol = 1, Tol = 30, Phase = Go_home. **Right:** Pol = 2, Tol = 60, Phase = Go_wp.

An observation was made regarding the landing phase of the drone. In policy 2, the drone landed straight down, as usual, but in policy 1, in some of the runs, the drone landed after it formed a small curve. This can be depicted in the left of 6.9 and in the left of 6.11, where the green dots during takeoff don't align exactly with the red dots during landing. This means that a slight deviation took place (a small curve in our case).

For the transient errors, the drone completed the mission for all 12 runs of the experiment. This means that our noise tolerance system can also handle transient errors in small missions. We will present indicative trajectories for low, medium, and high noise frequency, in 6.12:

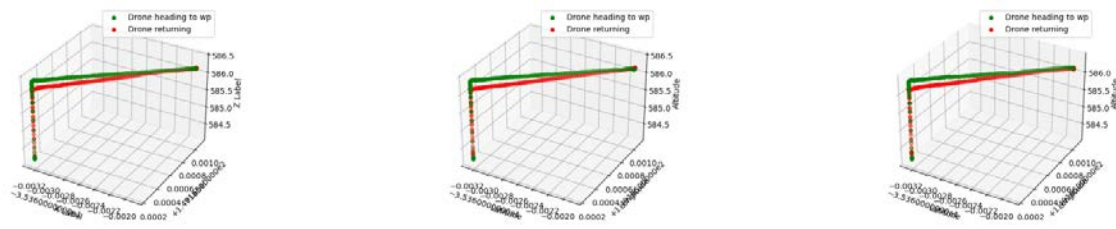


Figure 6.12: **Left:** Low Freq **Center:** Medium Freq **Right:** High Freq

Just like permanent errors, we can see that the trajectories match the non-noisy trajectory, essentially showing the ability of the drone to complete the mission under the influence of recurring noise.

Chapter 7

Conclusion

In this thesis, we have created a simulated dual Autopilot system through the process of active replication. The process is based on a synchronizer, which receives the motor commands from the 2 Autopilots, sends one of them to Gazebo, and then fans out the sensor data produced by Gazebo to both Autopilots. As a result, the drone is virtually equipped with 2 Autopilots and is able to follow an automated mission. It is also able to detect crashes and survive them. We also presented the testing procedure each module went through, to ensure proper functionality. Lastly, we showcased the model's tolerance to external factors (e.g. noise).

Following the findings of this thesis, several ideas for future implementation have entered the picture. The first idea is to try the setup we created on a real drone. There will be several differences to the mechanics, as the drone takes its data from actual sensors, but the core logic of keeping the 2 Autopilots in sync with each other can be applied. Another interesting idea is to try and implement a system with 3 Autopilots (in simulation first). In such a system, we could detect the faulty Autopilot and cut it off from the system, using majority voting. Therefore, tolerating the error of a single autopilot is not necessary anymore.

Bibliography

- [1] Byzantine Generals Problem. <https://medium.com/@ayogun/byzantine-generals-problem-a47b33ef87fc>.
- [2] M.A. Shahid, N. Islam, M.M. Alam, M.S. Mazliham, and S. Musa. Towards resilient method: An exhaustive survey of fault tolerance methods in the cloud computing environment. *Computer Science Review*, 40, 2021.
- [3] Copter Overview. <https://ardupilot.org/dev/docs/apmcopter-code-overview.html>.
- [4] UAV Definition. https://en.wikipedia.org/wiki/Unmanned_aerial_vehicle.
- [5] Flight Modes. <https://ardupilot.org/copter/docs/flight-modes.html#flight-modes>.
- [6] Drone Uses. <https://builtin.com/drones>.
- [7] Robotic AI Intrusion Detection. <https://www.nightingalesecurity.com/>.
- [8] K.S. Subramanian, S. Pazhanivelan, G. Srinivasan, R Santhi, and N. Sathiah. Drones in Insect Pest Management. *Frontiers In Agronomy*, 3, 2021.
- [9] Crash failure in system design. <https://www.geeksforgeeks.org/crash-failure-in-system-design/>.
- [10] What is Blockchain. <https://www.mckinsey.com/featured-insights/mckinsey-explainers/what-is-blockchain>.

- [11] J.K. Medhi, R. Liu, Q. Wang, and X. Chen. Robust Multiagent Reinforcement Learning for UAV Systems: Countering Byzantine Attacks. *Cyber Security and the Internet of Things (IoT): Theories, Approaches, Smart Applications, and New Trends*, 14, 2023.
- [12] N.M. Rodday, R.O. Schmidt, and A. Pras. Exploring security vulnerabilities of unmanned aerial vehicles. In *NOMS 2016 - 2016 IEEE/IFIP Network Operations and Management Symposium*, pages 993–994, Istanbul, Turkey, Apr 2016.
- [13] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4:382–401, 1982.
- [14] G. Valentini, E. Ferrante, and M. Dorigo. The Best-of-n Problem in Robot Swarms: Formalization, State of the Art, and Novel Perspectives. *Frontiers in Robotics and AI*, 4, 2017.
- [15] Mode.h. <https://github.com/ArduPilot/ardupilot/blob/master/ArduCopter/mode.h>.
- [16] Mode.cpp. <https://github.com/ArduPilot/ardupilot/blob/master/ArduCopter/mode.cpp>.
- [17] Mode_Guided.cpp. https://github.com/ArduPilot/ardupilot/blob/master/ArduCopter/mode_guided.cpp.
- [18] AC_AttitudeControl.cpp. https://github.com/ArduPilot/ardupilot/blob/master/libraries/AC_AttitudeControl/AC_AttitudeControl.cpp.
- [19] AC_AttitudeControl_Multi.cpp. https://github.com/ArduPilot/ardupilot/blob/master/libraries/AC_AttitudeControl/AC_AttitudeControl_Multi.cpp.
- [20] AC_PID.cpp. https://github.com/ArduPilot/ardupilot/blob/master/libraries/AC_PID/AC_PID.cpp.
- [21] AP_Motors_Class.h. https://github.com/ArduPilot/ardupilot/blob/master/libraries/AP_Motors/AP_Motors_Class.h.

- [22] AP_MotorsMulticopter.cpp. https://github.com/ArduPilot/ardupilot/blob/master/libraries/AP_Motors/AP_MotorsMulticopter.cpp.
- [23] SRV_Channel_aux.cpp. https://github.com/ArduPilot/ardupilot/blob/master/libraries/SRV_Channel/SRV_Channel_aux.cpp.
- [24] Gazebo. <https://gazebo.org/>.
- [25] Gazebo Plugin. https://github.com/ArduPilot/ardupilot_gazebo/tree/main.
- [26] SIM_JSON.cpp. https://github.com/ArduPilot/ardupilot/blob/master/libraries/SITL/SIM_JSON.cpp.
- [27] Ardupilotplugin.cc. https://github.com/ArduPilot/ardupilot_gazebo/blob/main/src/ArduPilotPlugin.cc.
- [28] Iris Copter. <https://www.adafruit.com/product/2199>.
- [29] Iris Copter. https://www.gabahobby.com/index.php?route=product/product&product_id=157964.
- [30] Mavlink Message Definitions. https://github.com/ArduPilot/mavlink/blob/master/message_definitions/v1.0/common.xml.
- [31] Mavlink File Format. https://mavlink.io/en/file_formats/.
- [32] PyMavlink Library. https://mavlink.io/en/mavgen_python/.
- [33] Command Long Format. <https://mavlink.io/en/messages/common.html#messages>.