Exploring a New Method of Controlling Fleets of Low Power Quadcopter Drones

COMP3003
psyjg12@nottingham.ac.uk – 20180775
Supervised by Dr Steve Bagley (pszsrb@exmail.nottingham.ac.uk)

April 2022

All work in the following dissertation is my own except for when stated otherwise.

J.G.

Contents

1 Introduction & Motivation

In this dissertation we explore the possibilities of a *dumb drone*[1] - that is, a drone which relies on a server to do all (or the vast majority) of its calculations for it, including not only navigational things like pathfinding, but even calculating how to set each of the drone's motors to move in the required way.

This can be seen as analogous to a "thin client", which refers to a device which has either no, or very little computing power, and relies on a host system for any processing it needs done. A now obsolete but once popular example of this is a dumb terminal, which sends key presses directly to a computer somewhere, and simply prints the characters it receives from the server on its screen. This has advantages and disadvantages: the hardware is a lot cheaper because of the simpler design

[1]The term "drone" can refer to lots of different types of unmanned vehicles. In this dissertation we use "drone" to refer to specifically quadcopter drones, unless otherwise specified.

and less advanced processor, and as a result also uses less electrical power, while sacrificing various useful features that a user might expect from a terminal.

Dumb terminals went out of fashion because the extra processing power needed for a smarter terminal became cheap to implement, but the idea is still interesting, and seems that it could be beneficial for modern day drones.

A typical arm microprocessor that might be used onboard a drone can consume around 7 watts.[2] A typical drone battery may be in the order of 1000mAh in capacity - maybe a little larger than this, but not by much because of the increased weight of higher capacity batteries. Furthermore, a typical microprocessor runs at 5V, which for this purpose can easily draw at least 1A of current. This means that a typical drone battery can't run even just the processor on its own for much more than an hour, without even taking into account the power drawn by the motors, sensors, and RF transceiver.

Offloading all of the calculations to a central server has the advantage of allowing the drone's power supply to be dedicated almost entirely to running its motors, only needing to have the capability to read data from the sensors, pass that data directly to the server, and feed the response it gets from the server to its motors. This has two main advantages: each drone can have a slower, cheaper processor, saving significant cost if a large fleet of drones is required, and, each drone will use less power to operate, leading to more flight time and less time charging.

2 Related Work

To calculate what these drones should set their motor speeds to, we'll employ a popular algorithm called PID control.

There are countless papers published about PID control in general. The first paper which proposed the PID algorithm was written in 1922[8]. It proposes a system which extended the already existing *proportional control*, which is where the input into a system is the difference between the target value (the *error*) and the measured value. PID is an extension to this, where the output of the controller also is affected by the sum of the previous error values over time, and the derivative of the error. This allows the system to have better stability once it reaches the target value, and avoid overshoot.

The PID algorithm requires a gain parameter to be set for each of the terms (proportional, integral, and derivative), which can be difficult to choose manually, so there has been a significant amount of work in various ways of "tuning" those parameters. One such method is the Ziegler-Nichols method[17], which is still a manual method but makes it a lot easier than simply guessing.

Another method for PID tuning was proposed by Åström & Hägglund in 1984[2], and is fully automatic. This can be very useful for complex systems involving lots of interacting PID controllers, but for my project which will have only three *independant* PID controllers, I think it would be faster to use something along the lines of the Ziegler-Nichols method.

There has been some work into systems for controlling multiple drones from a central server, similar to what I'm doing. For example, a paper was written by Paula and Areias in 2019 about a 'ground control' system to facilitate multiple drones performing autonomous missions[12]. The architecture they propose involves what they call a "Ground Broker", which is a server that communicates with all the drones in the system, as shown in Figure 1. It's made up of multiple components including a "Mission Planner", which communicates with software on the drone called the "Mission Worker" to carry out missions which are defined by the user. The missions in this system are written in an XML format and parsed by the ground broker.

This paper has some really interesting ideas, such as a "drone self-replacement" system which can replace a drone carrying out a mission with a new drone which can continue the mission, if the first drone has some sort of malfunction.

The difference between this and my project is that the drones in this one each have an onboard flight controller (the OpenPilot Revolution controller, specifically), unlike my drones where all the motor control will be done from the server.

Another paper I looked at was titled "Development of Multiple AR.Drone Control System for Indoor Aerial Choreography", and talks about making a system of drones which can perform a sort of choreographed dance together[9]. The client-server model used in this paper is really similar to

[2]For a rough estimate I'm using Raspberry Pi 4 benchmarks.[14] A Raspberry Pi 4B can consume 6.4W when running at full utilisation, which is likely comparable to a drone's CPU usage because of the large amount of calculation required.

the last one I discussed - each drone connects to a ground control server, and each drone does its own calculations for its motion.

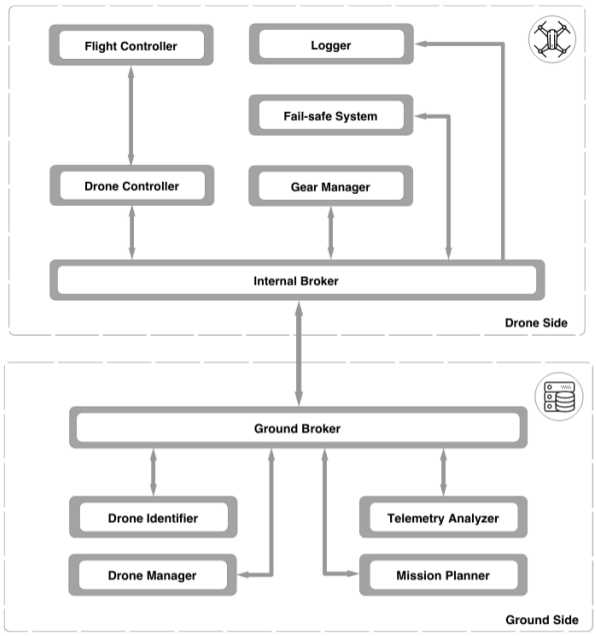

Figure 1: Paula and Areias System Architecture

This paper mainly focuses on coordination and location of all the drones in the system. It uses a large array of computer-readable codes on the floor of the room the drones fly in (the system is designed for interior use), where each code encodes the position it is placed. This way, a drone flying above which has a down-facing camera can scan the codes below itself and work out where it is. Of course, the drone also has on-board sensors (accelerometer, gyroscope, etc.), which can be used to also give a rough idea of where it is in the room, but neither this nor the codes on their own would be fully accurate. The paper doesn't explicitly state this, but I imagine they use an algorithm such as a Kalman filter to combine the sensor readings and the localisation information from the codes to get a far more accurate location reading. This method is commonly used in other drones, but combining readings from the motion sensors as well as an onboard GPS receiver instead. I imagine that the reason they didn't use GPS in these drones is because GPS is very unreliable inside buildings.

There has been a lot of work on simulating drones. One such example is Microsoft AirSim[16]. This is a package for Unreal Engine (and also for Unity) which simulates drones in a realistic environment. It also includes flight simulator software. This package is designed primarily for the purpose of training AIs that will run on drones.

Another similar package is developed by MathWorks, who offer a drone simulator addon for Matlab. This package is designed mainly for development of new drones, and allows the user to tune controller parameters, and develop algorithms for navigation and such.

3 Description of the Work

The aim of this work is to find out if it's possible to use a single centralised server to offload the slow calculations from many drones all onto the server itself. Instead of the drones calculating the speed to set each of its motors to, the server will know the position it's supposed to be going to and will calculate the necessary speeds on its own much faster processor. This also has the scope of using multiprocessing to perform these calculations when many drones are connected to the server.

To do this, the following things will be needed:

- A server which drones can connect to, which will take sensor data from the drones and calculate values for them to set their motors to. This server should have a way of a human operator deciding where the drones should go, and provide some status readout. The server should also be able create artificial and user configurable latency, to check how well the control system works in real life environments.
- A physics based drone simulator, which has variables for the speed of each motor. This simulator should also be able to simulate sensors such as a gyroscope, accelerometer, and a GPS reader. The simulator should be able to run multiple drones simulataneously.
- "Firmware" to run on the virtual drone, which will read its virtual sensors, send them to the server, and set its motors in accordance to what the server says.
- A comprehensive network specification for messages between the server and the drones. This should specify everything from the precise format of all messages, to the timings that they

have to be sent in, so that anyone is able to write firmware for their own drones which can interact with this server.

4 Methodology

4.1 Software Used

First, we needed to decide which software to use for the drone simulator. Two possible software packages were considered: Microsoft AirSim, and Mathworks' Drone Simulator package. Another option was to use a pre-existing physics engine and use it to simulate drones.

Microsoft AirSim isn't ideal for the project because it doesn't have an easy way to modify the flight controller code; it's designed for training machine learning algorithms on data from simulated drone cameras, and for programming navigation algorithms, etc., and not for creating the actual drone control algorithms, which is what we will be experimenting with.

Mathworks' Drone Simulator package looks better, but it's both prohibitively expensive and still not simple to modify the flight controller code. For these reasons, we will write a drone simulator using *Godot*[3]. Godot has a built in physics engine on top of which the drone simulator can be built.

Due to how the drone will be simulated, Godot's physics engine won't handle yaw rotation itself (which is achieved by counter-torque produced by the motors) since, for simplicity, we'll model each motor as simply an object which applies a force to the body. This is not a problem though, since the yaw rotation can be modelled with some maths, which we will discuss later.

4.2 Drone Principles of Operation

In this dissertation we're dealing primarily with quadcopter drones, but the way in which these work extends to multicopters with any amount of rotors.

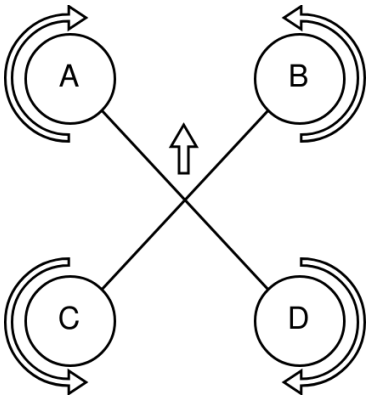

Figure 2: A possible arrangement of motors on a quadcopter.

A quadcopter consists of four propellers on motors, connected to a body. This is shown in Figure 2. The most important thing to note is that these motors do not all spin in the same direction; A and D spin clockwise, whereas B and C spin anticlockwise. Motors on a drone don't change direction, the only aspect that is changed is how fast each one spins.

Each motor produces both a lift force as well as torque. If the sum of the speeds of the two clockwise motors are greater than the sum of the speeds of the two anticlockwise ones, the net torque on the drone body will be in the anticlockwise direction, because a spinning motor causes counter-torque in the opposite direction. This is the basis for producing yaw rotation.

To change the amount of lift force on the drone, we can simply adjust the speed of every motor by the same amount. If all the motors spin at the same speed and produce a sufficient net force, the drone will rise upwards evenly. If the motors slow down sufficiently, then the drone will fall downwards due to gravity.

Pitch and roll rotation are achieved in similar ways: if we set the speeds of motors C and D to be greater than those of A and B, then the back of drone will rise up. This will cause the motors to point forwards slightly, so the forces from the motors will have a small forwards component. This rising of the back of the drone is called pitch rotation, and is how we make our drone move forwards. In an equivalent manner, roll rotation allows the drone to move side-to-side.

It's important that the arrangement of motors in a quadcopter is such that the two clockwise motors are diagonally opposite each other. This is so that the drone can perform yaw rotation independant of pitch and roll. To see why this is the case, imagine that motors A and C were the two clockwise motors, and B and D were anticlockwise. If we wanted to yaw, we would have to have the speeds of A + C be greater than B + D: this would provide the necessary net torque. The problem here is that as mentioned previously, having a greater net speed on one side of the drone causes

[3]Godot, pronounced GOD-oh, is a multimedia and game development software package.

either a pitch or roll rotation. Adding the same speed to two diagonally opposite motors keeps the net speed of each side of the drone equal relative to each other.

4.3 The PID Algorithm

The PID algorithm is a very important part of this project. This is the algorithm we use to set the input values to the simulation (namely the speeds of each of the motors on the drones) based on a target state. To explain how this algorithm works, we can first imagine how we might achieve this without such an algorithm. Let's also simplify the problem to more clearly explain what the algorithm is doing. We'll reduce the amount of dimensions from three to one, and imagine a one-dimensional drone. It might look something like Figure 3.

This is essentially just a motor on top of a ball. If the motor speeds up, the drone will accelerate upwards, and if the motor slows down or stops it will accelerate downwards due to gravity. Since this drone exists in a 1-dimensional world, we don't have to worry right now about the drone veering off to the side. Now, let's imagine that we want to control the drone's motor so that it stays at a specific altitude of, let's say, 100. We will call the target altitude our *setpoint*. If our drone starts at altitude 0 (resting on the floor), it's clear that we need to increase the motor speed, so we need to calculate what to increase it by.

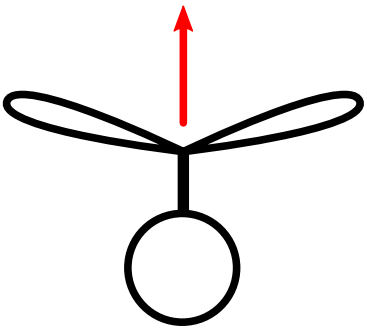

Figure 3: 1-dimensional drone.

The simplest method of doing this would be to set the motor to 100% of its maximum output when the measured altitude is below the setpoint, and set it to 0% when it's above. A drone using this rule would lift off of the ground and constantly accelerate until it reached the setpoint, at which point it would stop accelerating upwards. It would still have some upwards momentum though, which would carry it past the setpoint until the acceleration due to gravity started moving it down again. Once it falls down below the setpoint, its motors would start again and slowly start to move it upward again. This solution will never stay at the setpoint, and instead continuously oscillate around it.

An improvement to this is to make the motor speed *proportional* to the difference between the current position and the setpoint, which we'll call the *error*. This way, when the drone is on the ground, far away from the setpoint, the motors will spin close to their maximum speed. This is good, because the drone can reduce the error faster. As it comes closer to the setpoint, it slows down. This means that it will overshoot by less, causing less oscillation. For this reason, using a proportional term is preferable to not, but a system using *only* a proportional term normally will not settle on the exact target value. This is called a steady-state error.

To try to eliminate this steady-state error, we can introduce an *integral* term. This is simply another value that we calculate that will affect the controller's output, and is the sum of all previous errors. When the drone starts on the ground, the error is high, and so the integral error is also high (as at this point it is just the error). As the drone approaches the target, the error decreases, but the integral error keeps increasing until it overshoots the target. Once it overshoots, the integral error will begin decreasing, since the error here is negative. At some point, the drone will fall down to the setpoint. It will still oscillate a little bit, but the steady-state error will be eliminated. This is due to the fact that if there *is* a steady-state error, this will be reflected in the integral error, which will tend towards that error being reduced.

With this integral term, the system can still take some time to settle. To improve this, we need to implement some sort of damping, to limit how rapidly the altitude can change. If it can't change as quickly, it will overshoot the target by less, causing less oscillation. For this damping, we introduce the third letter of PID - the derivative term. To calculate this, we get the difference between the current and the previous altitudes and add this value to the total error term. If the drone is accelerating rapidly towards the target, this derivative error will be high, which will cause the controller to reduce the motor speed.

The PID controller can be represented with the following equation[4]:[4]

$$u(t) = K_p e + K_i \int_0^t e dt + K_d \frac{de}{dt}$$

[4]This equation has been adapted from the cited page.

In this equation, K_p, K_i, and K_d are constants which affect how much each term contributes to the result. $u(t)$ is the value that is fed back into the system (so in this example, the speed which the motor will be set to).

1-dimensional PID controllers are widely used in modern day control systems, as it's often required to accurately and efficiently control a single value. Common exampled are temperature[1] and flow rate[7]. To extent this concept to three dimensions, you can simply create a 1-dimensional PID controller for each dimension. Since a PID controller operates as a closed loop controller (where the output of the system is fed back into the input), the controllers can all be independant objects which only interact with each other through their shared manipulation of the system as a whole.

4.4 Networking Technologies

There are two distinct interfaces in this project: communication between a drone and the control server, and communication between the control server and the user control panel web page.

4.4.1 Physical Drone

The drones in this dissertation are, as mentioned, simulated in software. Despite this, we will discuss physical network interfaces as a starting point for future research as well as to inform decisions in this dissertation pertaining to simulation of real life network environments (like latency, packet loss, etc.)

One possible option for this communication is the LoRa radio protocol. With line-of-sight, these transceivers can reach ranges in excess of 1 - 3km[5]. The particular LoRa transceiver discussed here can transmit data at up to 300kb/s. A LoRa transceiver can operate on multiple channels (not simultaneously, but it is able to switch which channel it's using).

Despite having a range on the order of kilometers when the transmitter has direct line of sight to the receiver, LoRa transceivers often have their range reduced to around 150m[3] in a heavily built up environment. Although this is a large reduction, it's better than most similar RF transceiver types, for example the widely used ZigBee protocol. These typically have a range of just 10-100m *with line of sight*, and around 20m in a built up area[3].

Martin Bor et al.'s paper on LoRa[3] claims they could get LoRa nodes to communicate with each other with 80% packet reliability. Their nodes achieved an impressive "potential lifetime of 2 years on 2 AA batteries", but it seems this is because they were sending the data at a very slow rate, which is not feasible for a system that needs to act quickly in realtime, such as the drone motor controller server. The specific LoRa transceiver mentioned earlier[5] can send data suitably fast, at the expense of a higher power draw.

4.5 Interface for simulation

The simulation itself runs as a Godot application, whereas the motor control server is a Python application. Writing the server in Python rather than integrating the motor control server into the Godot application has two major advantages. Firstly, separating the server-side and client-side into two distinct programs means that if a physical drone was made using this system, it would be trivial to adapt the server software to communicate with it.

There are two possible ways to do this: the simplest would be to replace the methods in the server code which transmit data over the network and establish connections, and change them to interact with the specific network hardware that is chosen for the drone-server communication. For example, to allow the drones to communicate over a LoRa network, the Python code would need to access the LoRa transceiver hardware. This can be done either using a transceiver with a USB interface, or by running the Python code on a single board computer with general purpose input-output (GPIO).

Another method would be to either use a Wi-Fi network card on each drone (which would mean that the onboard processor needs to have the capability to demodulate and modulate Wi-Fi signals, and therefore would not be a lower power processor, defeating the purpose of this method), or to create a form of "repeater" between the server and the drones, which reads all relevant network packets, and sends them out using a protocol such as LoRa. This would work, but would be likely to greatly increase the latency of messages between the drones and the server.

The other advantage of separating the server code and drone code is that the server, when talking to a simulated drone, should have the option of introducing latency and packet loss into the

connection in order to test the performance of the system. If there's a definite and clear interface between the two pieces of code, it will be easier to implement these parameters.

For communicating between the control server and the drone simulator, a decision has to be made between TCP and UDP sockets. In a TCP connection, a server program listens on a specific port for clients to connect. If a client connects, the server establishes a connection. When sending data, TCP splits the data into *packets*, which are sent to the receiving socket. Each packet is sent with a sequence number. If the receiver gets the packets in a different order, they will know (because of the sequence number), and can rearrange them. If any packets are lost between being sent and received (due to a faulty or intermittent network connection, for example), TCP will ensure that those packets are resent. Each packet's contents is checked with a checksum to ensure that the data hasn't been corrupted.

UDP, on the other hand, is far simpler. There isn't a concept of a *connection* between two machines - rather, they can simply send packets of data to each other, without any preamble required. The packets that are sent may or may not arrive at the receiver's socket, and if they do, they may or may not be corrupted. This is because, unlike TCP, UDP does no error checking, nor does it resend packets. UDP makes up for its shortcomings by being significantly faster.

UDP makes the most sense for this application, since latency and speed are my primary concerns. If a packet is dropped whilst travelling over the network, it is not optimal to try and resend the packet. This is because by the time the packet got resent, it would not be up to date anymore, and so it would be better to simply keep trying to send newly calculated packets.

4.6 Emulating aspects of real network interfaces

As mentioned, UDP connections do not guarantee that packets get delivered. For the drone simulation, the UDP connection is fully contained within one computer, so it should be extremely close to 100% reliable, but since our aim is to investigate how well this system performs for *real* drones, we should introduce artificial latency and packet loss into the system.

5 Drone Simulator

5.1 Design

Since we're using Godot's built-in physics to simulate most of the movement of the drones, designing the drone simulator is not as difficult as it could otherwise be. First, we have to decide how the user is able to interact with the simulation. The user of this drone simulation would likely be someone continuing this research, and so the system should be easy to use for this purpose. Of course, it'll also be used for the evaluation stage of this dissertation, so there should be a way of retrieving numerical output which can then be used to produce graphs. As well as getting data out of the simulation, it should be easy for the user to visually see how the drones are moving around. Finally, the user should have a method by which they can they control the drones (so, setting positions for the drones to travel to).

The specific data which the program should give to the user for evaluation purposes are:

- The path each drone has taken (as a list of coordinates and directions).
- How the PID controller errors have varied with time.

This will allow the user to plot graphs to analyse how well the PID controller is tuned.

To make it easy to see a drone moving, the camera should follow it, and give the user the option to rotate the camera around the drone. If there are multiple drones in the scene, then there should be a way of switching which drone is being viewed. So, the camera will contain parameters for how far away from the drone it should stay, as well as its rotation relative to it (stored as yaw and pitch). Whenever the drone moves, we can simply calculate its position and rotation by imagining it being *locked* to a sphere centred on the drone, and using trigonometry to work out where it is. The equations we can use for this are:

$$p_x = cos\theta * cos\alpha$$
$$p_y = sin\alpha$$
$$p_z = sin\theta * cos\alpha$$

5.2 Implementation

First of all, we needed a 3d model to represent the drones in the world. For this, we created a model of a drone in Blender (Figure 4).

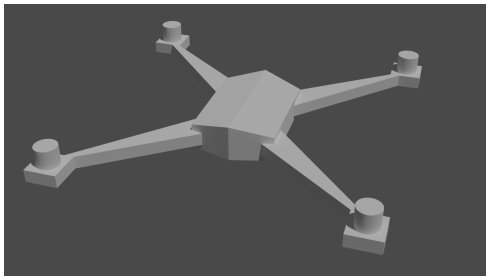

Figure 4: Drone model within Blender.

In terms of the functionality of the software, it does not matter what the drone looks like, because the physics engine which is being used doesn't simulate turbulence or air resistance. For this reason, the 3d model we use doesn't have to be aerodynamic. After making this model, we had to load it into Godot, and set up our scene.

5.2.1 Godot

Godot offers a hierarchical method of organising the program. The programmer can create several "scenes", each of which consist of a root node with a hierarchy of child nodes. The root node specifies whether the scene is 3d or 2d, so a program can consist of scenes of varying dimensions. One scene must be set as the "Main Scene", and this is the scene that is displayed when the program starts.

5.2.2 Scene Layout

The main scene in this program will contain several drones. Additionally, the main scene will contain a camera object, as well as some objects to create an environment for the drones to fly around in. This hierarchy will look similar to this:

- Root Node
  - Camera
  - Floor Plane
  - Drone List
    - Drone 1
    - Drone 2
    - etc.

And within this tree, the Drone nodes are all instances of the same scene. The Drone scene will look similar to this:

- Drone
  - 3D Rigid Body
    - Drone Mesh
    - Collision Shape

A "rigid body" is a particular type of physics object; the other available types are soft bodies, kinematic bodies, and static bodies[13]. A rigid body is a physics object whose shape is fixed (i.e. it doesn't deform when pushed). It has various physical properties such as friction, mass, and restitution. When writing a program to control a rigid body, you don't directly set its position – instead, you apply forces to it, and the physics engine will *integrate* these forces to calculate where the object should be, as well as how it should rotate.

A soft body is similar to a rigid body. The difference is that its shape can deform. This is useful for simulating objects such as cloth or balloons. We won't be using any soft bodies in this program, since although technically drone objects *can* deform in real life (for example, if they crash), there is no use simulating this because it won't affect the aspects of the drone control that we're testing.

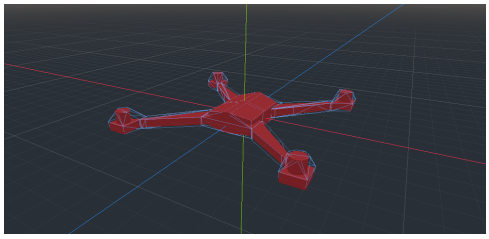

Figure 5: A screenshot of the Drone scene.

The third type of physics body that Godot offers is a kinematic body. This type is different to the first two because its movement is not handled by the physics engine - instead, the programmer can manually move it by exact amounts. It does still interact with the physics world though. Finally, a static body is similar to a kinematic body, except that it isn't intended to move.

To begin making the Drone scene, we first imported our drone mesh from Blender into Godot. As mentioned, we need a collision shape for the drone. A simple option is to use the bounding box of the drone as its collision shape. This is more efficient to simulate, and simpler to create, but has the major drawback that for a complex model such as this drone, the collisions will be wildly inaccurate.

Luckily, Godot provides functionality to generate collision shapes for any given mesh, which is what we use here. In Godot, collision shapes must be convex bodies (which can be thought of as meaning you can draw a straight line from anywhere inside the shape to any of its vertices. My drone mesh is concave, however, and so Godot has to create a number of convex collision shapes to represent it.

Once we have a mesh and a collision shape, we can create a *RigidBody* node. This node must have at least one collision shape as a child node so that the physics engine knows the shape of that particular body. Note that child nodes in Godot inherit any transformations applied to their parent, which in this case means that when the physics engine translates the *RigidBody*, the mesh and collision shapes within it are automatically moved.

Putting this all together, we have a Drone scene as shown in Figure 5. The collision shape is represented by the blue wireframe surrounding the drone mesh, which is red. Notice that Godot uses a Z-forward and Y-up coordinate system. The tree for this scene is shown in Figure 6.

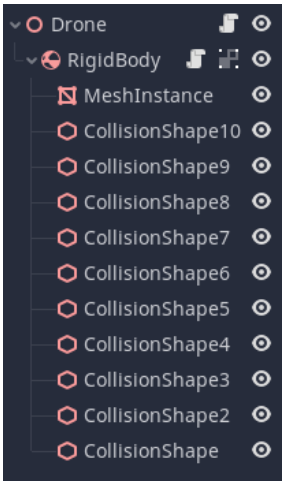

Figure 6: The tree of the Drone scene.

Note that the scroll icon next to the *RigidBody* in Fig.6 means that it has a script attached to it, which is just a source code file which contains functions that are run at various points during the program execution. This will be explained in much more detail later on.

Now that we have a *Drone* scene available to us, we create our *Main* scene, which will be the scene that is directly run when the program starts. We constructed this scene as described in Figure 8.

Most notably, here we have an instance of the *Drone* scene, as described earlier. When we instantiate a scene in Godot, we're able to position it anywhere we like, so we raised it slightly above the floor. It's relevant that the Drone instance also has a script attached to it - all *Drone* instances need to have a different network port that they use to communicate with the central server, and this additional script is used to expose the port property to the Godot GUI to make it simple for the user to modify the port of each *Drone* as they wish. At this point, the *Main* scene looks the screenshot in Figure 7.

The floor here is implemented as a *StaticBody* node which, just like the *RigidBody* node from earlier, needs a collision shape and mesh within it to function properly. Finally, we need a camera which is where Godot renders the scene from. Later on, we can make the camera follow a drone.

We will implement the motors in a way slightly different to how the motors work in real life. In a real drone, each of the drone's motors turns a propeller, which pushes air down causing lift. Our physics simulation doesn't simulate airflow, though, and so if we simulated four spinning propellers, no lift would be created. The workaround for this is to represent each motor as simply a point which applies a force to the drone. To implement this, we first can define some arrays:

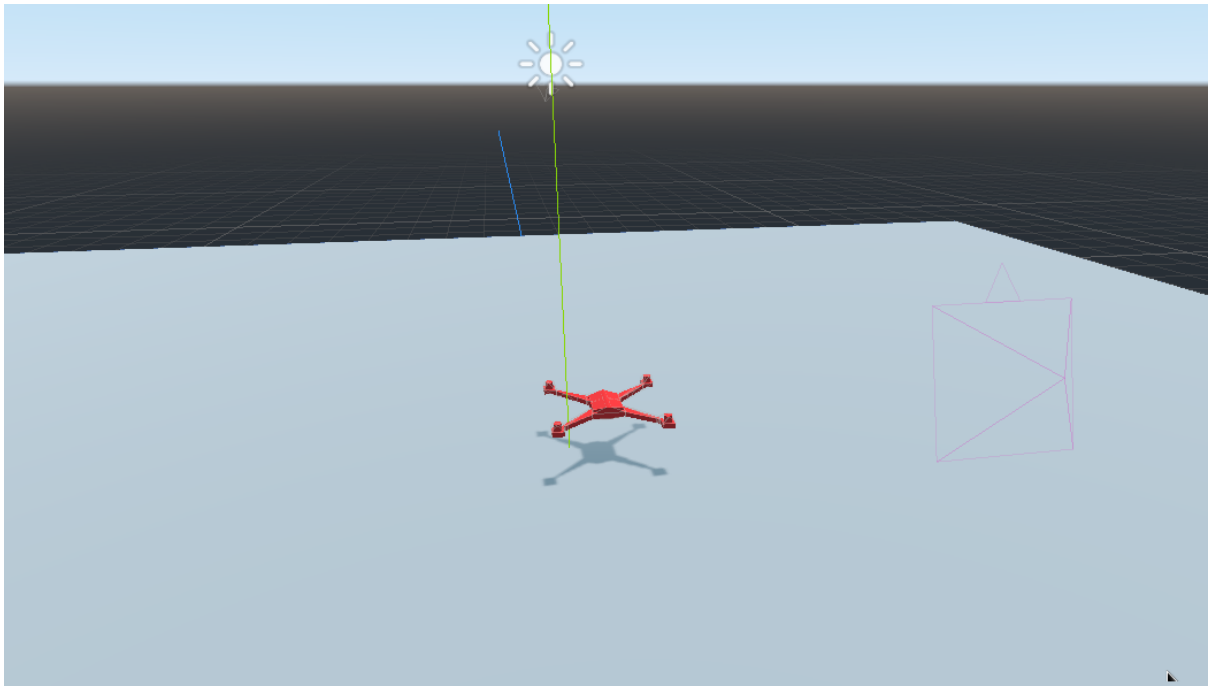

Figure 7: A screenshot of the Main scene within the editor.

```
var mots = [
        0,0,
        0,0
]

var mot_dirs = [ # 1 = clockwise, -1 = counter-clockwise
        1, -1,
        -1, 1
]

var mot_offsets = [
        Vector3(4.5,0,4.5), Vector3(-4.5,0,4.5),
        Vector3(4.5,0,-4.5), Vector3(-4.5,0,-4.5)
]
```

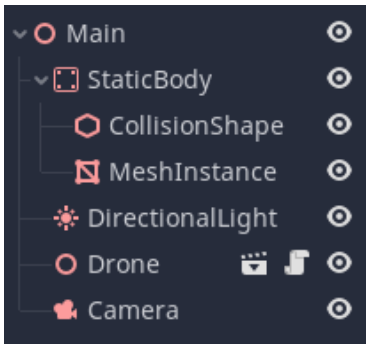

Figure 8: The tree of the Main scene.

Firstly, note that this code is written in Godot's GDScript. For the most part it's very similar to Python, but a big difference here is that we have to use the *var* keyword to declare variables.

The *mots* variable will represent the power of each motor. These arrays are all arranged as if looking at the drone from the top: mots[0] is the front left motor, mots[1] is the front right, mots[2] is the back left, and mots[3] is the back right. This pattern also holds for *mot_dirs* and *mot_offsets*.

Since we want this drone system to be as general purpose as possible, it makes sense not to hard-code the direction which each motor spins in. This is what the *mot_dirs* variable is for: each motor has an assigned direction, which is either 1 or -1 to represent clockwise or counter-clockwise. Since we are not simulating the motors as propellers that actually spin, this will have no effect visually on the program, but the counter-rotating propellers are required for the drone to be able to control its yaw rotation, as explained in Section 4.2.

The last thing that is needed to fully represent the motor setup on the drone is where the motors are relative to the drone's centre. This is needed when applying the motor forces, since a motor that's

further from the centre will create a greater moment with an equivalent force. These positions need to be set properly since otherwise the drone's pitch and roll rotation will not work as expected. For this particular drone, the horizontal and vertical spacing between the motors is 9 units, which we used to work out the motor positions given that they're symmetrical around the centre (which is at the position (0,0)).

Now that we have this set up, we can simply simulate the motors by applying the correct force at each motor's position, with the following code:

```
for i in range(4):
    if mots[i] < 0: mots[i] = 0
    add_force_local(
        Vector3(0,mots[i],0),
        mot_offsets[i]
    )
```

We can test if this works by setting *mots* to *[10, 10, 10, 10]*, to turn every motor on at a fairly high power. If we then run the program we can observe that our drone moves upwards, as expected. Similarly, we can set both of the left-hand motors to 10 and both of the right-hand motors to 3 to cause our drone to roll in a clockwise direction. Using similar methods, we can cause the drone to perform any pitch or roll rotations we want, as well as combinations of those two types.

With this current method, our drone cannot perform yaw rotations[5]. This is due to the fact that, as mentioned earlier, yaw rotation comes from the net torque of the drone which is controlled by the rotation of the motors. If the two clockwise motors spin faster than the two counterclockwise moors, the drone will undergo a counterclockwise yaw rotation. The formula for calculating the torque from one motor is $\tau = Fd$, where F is the perpendicular force, and d is the distance from the centre of the body to the position that the force is applied to. We can calculate the total torque on the drone by simply calculating this for each motor and adding them up.

We run into a problem here, however. We don't know what the force applied by the propellers is. Force can be calculated using the equation $F = ma$, but we don't have any values for mass or acceleration. This is because so far in this simulation we have not taken into account the specific values of our physical properties, and since we're not modelling our drone on any one real drone it's impossible to know the exact values we should use here.

We can realise, however, that in the equation for torque, F is the only variable that will change: d remains constant, since the motors are always the same distance from the centre of the body. Furthermore, since we're not trying to simulate any specific real drone, the exact force that the propeller exerts on the drone doesn't matter as long as it produces realistic motion. We don't attempt to make our drone's behaviour identically match specific real drone, because every model of drone will have a different mass and motors which supply different amounts of force. It would be useless to go out of our way to simulate one of these exactly, because the parameters used for that couldn't be used for any other drone.

We shall change the previous code to the following:

```
var torque = 0
for i in range(4):
    if mots[i] < 0: mots[i] = 0
    torque += -mots[i] * mot_dirs[i] * 20
    add_force_local(
        Vector3(0,mots[i],0),
        mot_offsets[i]
    )
add_torque(Vector3(0,torque,0))
```

The difference here is that for each motor, as well as adding its lifting force to the drone, the torque is calculated. We're using a value of 20 for the constant part of the equation (essentially $m * d = 20$). This value was found empirically by testing various values to see what looks the most realistic in terms of angular acceleration. Since the amount of torque is proportional to the speed of the motor, we can use *mots[i]* as the force in the torque equation (remembering to multiply this by *mot_dirs[i]* to take the direction of the motor into account). We are getting the negative version

[5]Recall that a yaw rotation is a rotation around the vertical axis

of the motor speed because if a motor's torque is in the clockwise direction, the resultant rotation on the drone will be counterclockwise.

At this point, we now have a working drone simulator. Our drone is capable of independent yaw, pitch, and roll rotation.

6 Drone "Firmware"

6.1 Design

Our drone simulator so far is equivalent to the electronics on-board a real drone - specifically, the electronic speed controllers (ESCs) which allow a microcontroller to control how fast each motor spins. To complete our implementation of the drone, we now must write the equivalent of the firmware that would be running on the microcontroller. The behaviour of this firmware is summarised in Figure 9.

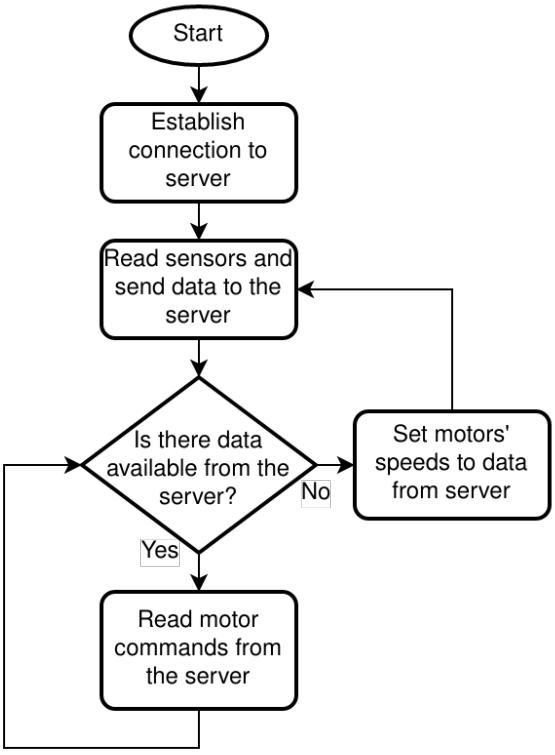

Figure 9: A flowchart of the drone's behaviour.

This can be thought of as some initialisation followed by a main loop. When the drone first boots up, it needs to connect to a server and say that it's a new drone that wants to be controlled. To decide how this communication will happen, we first need to design a network protocol between the drone and the server. This protocol will describe every type of message possible between the drone and the server, including the initialisation, how the drone should tell the server what its sensors read, and how the server should tell the drone what to set its motors to.

6.1.1 Protocol Design

There are several types of message that will need to be sent in this system. They are:

- Drone to Server
  - Registering a new drone
  - Sending sensor readings
  - Informing the server of low battery
- Server to Drone
  - Sending motor commands

There are a few guidelines we should follow when designing this protocol. Firstly, we should keep in mind that the protocol must work with real drones using real life interfaces. Looking at the datasheet of a fairly standard LoRa transceiver (the *easyRadio eRA-LoRa RF transceiver*), we can see that the data sending is split into packets (similar to UDP), and that there's a maximum of 180 bytes per packet. Because of this, we should limit the amount of bytes in each message sent to 180 bytes.

First, when a new drone wants to join the network, it must inform the server of this. Since we're not using an interface which has a concept of connections (UDP for the simulation, and potentially LoRa for future work on a physical implementation), this isn't handled for us as it would be with TCP.

The way that a drone signals to the server that it wants to be controlled is by sending a message consisting of a single 'N' character. If a server receives messages from a drone before that drone has registered itself, it will simply drop them.

We want to keep the format for sending sensor data as adaptable and drone-independent as possible. This means that it should support drones with varying numbers of propellers (even though at present this program won't support that), as well as varying sensors. For example, one drone may have four propellers where another has six (called a hexacopter), and one drone may have a gyroscope, accelerometer and barometer whereas another might have an onboard inertial measurement unit module (IMU) which aggregates data from these sensors internally and provides simple position and rotation measurements.

The different sensors we decided to allow drones to send data from are:

- Gyroscope
- Accelerometer
- Barometer
- GPS
- Translation
- Rotation (yaw, pitch, and roll)

The thought behind this is that if the server knows the readings of the accelerometer, it can make an (inaccurate) estimate of where the drone is through a technique known as dead reckoning. This means that if we have a reading of the acceleration of the drone, we can integrate this data (by adding up each data point we receive) to get the drone's velocity. We can then also integrate the velocity to get the position. This way of calculating the drone's position is very inaccurate, though, which is why other sensors are useful. A barometer can give us a rough reading of altitude (since atmospheric pressure becomes lower the higher you go, on average). Using this data along with the dead reckoning approach, we can gain a more accurate picture of the drone's position using a Kalman filter. This is an algorithm which can be used to mix readings from various sensors to produce a more accurate state estimation. The GPS data can also be used here to supplement readings from the accelerometer. There are plenty of low power chips that can be used to do this[10].

The *Translation* and *Rotation* "sensors" can be used if the drone performs sensor fusion onboard, allowing it to pass what it thinks its precise transformation is to the server. This is also useful for the drone simulator, which *does* know it's exact position and rotation.

The message format I decided on to accommodate these different varieties is defined as follows: *'S' followed by at least one sensor, written as that sensor's corresponding letter followed by all of that sensor's values, separated by '/'s. Each sensor is separated by ':'s*. A table of the available sensors, their corresponding letters, and the amount of values each one should by represented by is presented below.

Sensor	Letter	Values	Unit(s)
Gyroscope	G	3 (Yaw, Pitch, Roll)	rad s^{-2}
Accelerometer	A	3 (X, Y, Z)	ms^{-2}
Barometer	B	1 (Pressure)	atm
GPS	P	2 (Lat, Long)	°
Translation	T	3 (X, Y, Z)	m
Rotation	R	3 (Yaw, Pitch, Roll)	rad

To clarify how this format works, here are some possible sensor messages:

- *SG10.2/3.51/18.055:A0.1/-0.3/-0.8:P70.2321 /50.23012/801.32* – This defines a gyroscope reading (10.2, 3.51, -11.8), an accelerometer reading (0.1, -0.3, -8.8), and a GPS receiver reading (70.2321°, 50.23012°, 801.32m).
- *ST10.2/8.3/20.8:R0.1/2.212/-3.001* – This defines an absolute position of (10.2, 8.3, 20.8), and an absolute rotation of (0.1, 2.212, -3.001).

If a drone senses that its battery is low, it has the option of informing the server of this. It can do this by sending a message which begins with the character 'L'. If a server receives this message from a drone, it will attempt to guide the drone to safety so that it doesn't fall out of the sky.

The final message type is from the server to the drone, setting its motor speeds. The format of this is: the character 'M' followed by each motor speed, separated by '/' characters. Each motor speed shall be written in a human readable ASCII format, such as '6.532'. Each number shall be written to at *most* four decimal places, as this is plenty accurate enough without making the message too long. We want to keep the message length below 180 bytes (since this is the bytes per packet of the LoRa transceiver we considered previously[5]). Each number must be between 0 and 100 since they are percentages of the motor's total power.

All of the message types in our protocol are sent as plain ASCII text. At this stage, the protocol does not support application layer encryption, but if the physical or transport layers of the network stack support encryption then there's no reason it wouldn't work. Later on, we will discuss possibilities of encrypting these messages in the application layer if we're using protocols such as LoRa which don't offer an in-built method of encryption. We will also discuss other issues related to security, such as making sure that only the desired server can control a given drone.

6.2 Implementation

To program the drone's firmware, we need to add some code to the script on the drone's *RigidBody* node which we wrote in the section on drone simulation. First of all, we need to set up a UDP socket to communicate with the server. In GDScript, we can do this simply - first, we create our socket object, as well as some variables which shall be used shortly:

```
var server_addr = "127.0.0.1"
var server_port = 14444
var sock = PacketPeerUDP.new()
```

The above code is written outside of any functions. These variables are global only within the scope of this script, but *not* global to the entire application, so we won't run into common problems with global variables here. Godot calls these sorts of variables *member variables*[6].

When the program is run, the drone will try to connect to the server. We can do this with the following code:

```
if sock.listen(get_parent().client_port,
        server_addr) != OK:
    print("Error listening")
else:
    sock.set_dest_address(server_addr,
        server_port)
    sock.put_packet("N".to_ascii())
```

Here we're trying to create a socket to listen to messages from the control server. If we're able to do this[6], we can then set the destination address for any messages we send. At this point, we are listening to messages from the server, and ready to send messages back. As mentioned previously, we then have to send the 'N' character to the server to signify that we're a new drone trying to be controlled by it.

Now that the initialisation of the drone is done, we need to work out how our main loop will look. There are two things we need to do: send readings from our sensors to the server, and respond to any control commands from the server.

6.2.1 Sending sensor readings

For the simulated drone, we will send more sensor types than required just to ensure that each type can be read by the server. We *could* send only the absolute translation and rotation of the drone – this would be enough for the server to control the drone just fine, but we'll send more sensor types than this for testing.

Another consideration is how frequently to send sensor readings. There's clearly a trade off, where too high a frequency could overload the server, but too low a frequency would negatively affect how responsive the system is. For now we'll report our sensor readings at a frequency of 20Hz, but

[6]The primary reason such an operation would fail is if the port is already in use. Since UDP doesn't actually establish a connection, this operation will succeed even if the remote server isn't up - this is the purpose of the drone reporting its presence to the server manually.

this is something we will analyse during the evaluation stage to determine a more optimal value. The sensor frequency is determined by the variable *sensrate* so that it's easier to change during the evaluation phase. This avoids any hardcoded numbers. Each time the main loop runs, we add the delta time[7] to an elapsed time variable and check if that is greater than $1/sensrate$. If it is, then we can send the sensor data.

The code for sending the sensor data is as follows:

```
sock.put_packet(("SP" + str(self.translation[0])
    + "/" + str(self.translation[2])
    + ":G" + str(self.rotacc[0]) + "/" + str(self.rotacc[1]) + "/"
        + str(self.rotacc[2])
    + ":A" + str(self.linacc[0]) + "/" + str(self.linacc[1]) + "/"
        + str(self.linacc[2])
    + ":R" + str(self.rotation[1] + 3.14159) + "/"
        + str(self.rotation[0]) + "/" + str(self.rotation[2])
    + ":T" + str(self.translation[0]) + "/" + str(self.translation[1]) + "/"
        + str(self.translation[2])
).to_ascii())
```

Looking at this from the top down, we first see that we're using the *put_packet* function on our drone's socket object. As expected, this function simply sends a data packet to the remote address which we specified previously. We then need to construct a sensor message string. First recall that a sensor message has to start with the character 'S', followed by each sensor we want to send.

First, we'll send the GPS position with the 'P' character (which is why the message starts with 'SP') - GPS sends only the latitude and longitude, which are not applicable to our simulated environment because it doesn't exist on a spherical planet. For this reason, we'll just send the X and Z components of the drone's translation here, mainly to see if the server is able to properly read this data. The server won't need to use this data for the simulation anyway, because we'll also be sending our absolute position with the 'T' sensor.

The other sensors are sent in the same manner, but one notable difference is when sending the drone's rotation: we have to add π to the yaw rotation (which is the rotation about the Y-axis) to match Godot's 0° rotation orientation to that of the control server.

Now that we've sent our sensor readings to the server, our drone needs to check if it's received any commands from the server. Godot's socket class will keep a buffer of any packets that its received, and so if in the time period since the drone last checked for incoming messages it received *more than one*, we want to only consider one of them. When considering how best to do this, we had the option of considering either the most recent or the oldest message in the buffer – we could of course pick a message not at either end, but there is no reason to pick an arbitrary one in the middle. Seemingly another option is to consider *every* message in the buffer but since each message *sets* the motor speeds instead of modifying them, this would be equivalent to taking only the last considered message into account.

There seems to be functionally very little difference between considering the most recent or the oldest message, because for two messages to be in the buffer together they must have been received both within a very short timeframe (i.e. the time that it takes to go through the main loop once). We'll consider the newest message for two reasons: it makes the code slightly simpler, and having a more up to date message is better even if it's only very slightly newer.

```
var done = false
for _i in range(sock.get_available_packet_count()):
    var msg = sock.get_packet().get_string_from_ascii()
    if not done:
        if msg[0] == 'M':
            var val_strings = msg.right(1).split("/")
            mots = []
            for v in val_strings:
                mots.append((max(min(float(v), 100),0)/100) * power)
        done = true
```

[7] The time since the last loop.

The logic of this code may at first seem inefficient and/or convoluted because we're iterating through every packet even when we only act on one, but actually this is necessary. This is because to clear a message from the packet buffer, you have to call the *get_packet* function, which acts as a pop off of a stack[8].

In the above code we're, as mentioned, iterating through every available message, but as soon as we've read the motor controls from the first one we set the *done* flag to ensure that we don't act on any more.

This concludes the implementation of our drone's simulated firmware. The logic behind it is not complex, due to the nature of our *dumb drone* architecture; the control server handles most of the complex logic and control.

7 Drone Control Server

7.1 Design

A high-level overview of the control server is that it must have the following functionalities:

- Receive sensor readings from all connected drones.
- Calculate how the drones should set their motor speeds to reach the desired states.
- Accept new drones who want to connect.
- Allow a human operator to set the drones desired states.

As mentioned previously, our control server is written in Python. The control server's execution will be split into three threads. The main thread (the one which is created immediately when the program is run) is responsible for receiving commands from an operator[9] using a command line. There's a great advantage of taking drone control instructions from the command line: a third party program can pipe commands into the program while it's running. This fits in nicely with the UNIX principle of "Do one thing and do it well"[15]: our control server does as minimal of a job as possible, limiting its operation to what's necessary for controlling drones. It doesn't concern itself with anything like pathfinding or collision avoidance because these features aren't suitable for all models of drones. Separating the software that decides what the drones *should* do from the software that actually makes the drones do that makes the software far more modular and adaptable.

To see where this architecture is useful, we can imagine a use case. For example, a university might want a system consisting of a fleet of drones to deliver books from several campus libraries to several drop-off locations. The developers of this system would not have to worry about the low-level control of the drone's motors – they could simply write another program, in any language they like, which sends control commands to an instance of our drone control server. Their program would most likely communicate with a university website which would allow staff members and students to request specific books, which would then prompt a librarian to place that book on a specially designed podium. The designers of this system would have to devise some sort of claw actuator or suction cup to allow the drone to pick up the book, and then simply send control commands to send the drone to the drop-off location. This is just one example application; it's clear to see that this control system lends itself to very easy modification.

While this is running, we have another thread which is constantly waiting for messages from drones. This thread is responsible for two things: accepting new drones into the system, and handling messages from drones. This thread shall consist of an infinite loop which waits for incoming messages. When it receives one, it decodes it and reacts accordingly.

If this thread gets a message informing it of a new drone (i.e. one starting with the 'N' character), it will need to add the sender address of that message to a list of drones. This list is used whenever other messages are received to check if that drone has registered itself. The specific format of this address, for the simulated system, is a pair of an IPv4 address and a port number. If the message begins with an 'S' character then it shall be interpreted as a sensor report. To decode a message of this format, we can first cut the first character off of the string (the 'S') to be left with just the data portion. We can then divide the rest of the string into each sensor by using the ':' characters as

[8] The packet buffer is actually implemented as a ring buffer.[11]

[9] We use "operator" here to refer to a human controlling the drones.

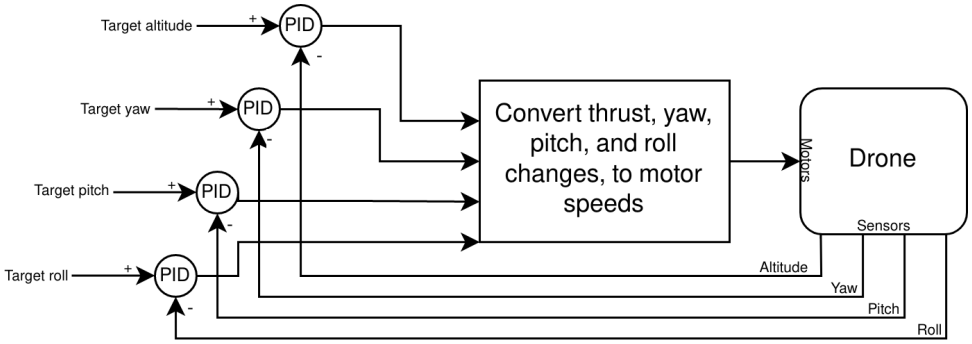

Figure 10: A diagram of the main drone control system.

delimiters. Each sensor may be decoded in different ways, in theory, but all of the sensors supported so far are 1 to 3 float values separated by '/' characters. These sensor values can then be loaded into a drone object.

A drone object is an instance of a drone class, which we'll have to define. First of all, this class needs to contain data for each of the drone's sensors. Each drone also has an ID, which is used by operator commands to select which drone to control. The drone class is also what stores the address of the drone. The drone class also contains a function which sends a message to the actual drone that it represents, telling it the speeds that it should set its motors to, as well as a function that computes those motor speeds. Encapsulating the logic for calculating the motor speeds within the drone class (as opposed to within the main server code), but also allows the potential for the system to support different types of vehicle in the future, such as a radio controlled car.

The method by which the motor speeds are calculated is described in Figure 10. This diagram represents the control system for one drone, to move that drone to a target altitude, yaw, pitch, and roll. The large box in the centre is for converting desired motor movement in the alt-yaw-pitch-roll form to a single speed for each of the drone's motors. For this proof-of-concept, we will only allow drones to have four motors. In the future, it would be useful to allow drones to specify their number of motors and configuration[10] of such when they register with the server.

The conversion algorithm is defined in the following way:

```
A:   yaw + pitch + roll + thrust
B: - yaw + pitch - roll + thrust
C: - yaw - pitch + roll + thrust
D:   yaw - pitch - roll + thrust
```

The motors here are labelled as described in Figure 2. For an example, we can see that if pitch is positive (i.e. we want to tilt our nose upwards), it's intuitive that the front motors (A & B) must speed up while C & D slow down. This behaviour can be seen from the signs in front of *pitch* in each element of the return of this function.

The PID circles in the Figure 10 take two inputs and provide an output using the PID algorithm, discussed in Section 4.3. Next to each input is a positive or negative sign, representing how the error is worked out. For example, if an input a has a negative sign (-), and an input b has a positive sign (+), then the error used by the PID controller is $b - a$, as it's just the sum of the inputs with their sign.

With this system, we could get our drone to hover in place by setting the target altitude to, for example, 10m, and then the yaw pitch and roll all to 0°. The problem with this is that if a gust of wind blows the drone off course, it will either start flipping over until it crashes into the ground, or drift away from the target point. To correct for this, we can add a stage before the target orientation which would use *another* PID controller to set the target angles to get back to the correct position.

We now need to design a class for a PID controller. An elegant way to achieve this is to have one PID object act as one PID controller, and contain everything needed for that itself. Our PID controller class first needs to store K_p, K_i, and K_d, to allow it to calculate the system's error. Another value that is essential to the operation of a PID controller is its setpoint – the value that the controller is trying to reach. It can also be useful to limit the output value of the controller to certain

[10]This means the direction (clockwise or anticlockwise)

bounds (for example if the PID controller's output must be a percentage between 0% and 100%), so our PID class will include fields for lower and upper bounds to clamp the output to.

There are a few more features that we'll implement to make our controller as useful as possible. For one, simply updating our integral and derivative errors as frequently as possible is not ideal - it's more desirable to update the controller at a fixed frequency to ensure that the error accumulation is consistent. Even if the controller is running at a fixed frequency, it won't necessary be able to update exactly at this frequency perfectly. This is because we will only call the function to update it every time the control server goes through the main loop (the controller will then check if the elapsed time since the last update has been long enough to perform another update), and the downside of this is that if one update takes a little longer to get to than the last, the error may affect the proportional term disproportionately. To fix this, we can simply multiply our proportional error by the elapsed time since the previous update each time it's added to the integral term.

7.2 Implementation

We can begin the implementation stage by defining the drone class. Its constructor should take an address (which in Python is a tuple of an IP address and a port number), and an ID number. Within the constructor we can store these values away in member variables in standard Python fashion:

```
class Drone:
    def __init__(self, addr, id):
        self.id = id
        self.addr = addr
```

We also need a few more member variables, which we can initialise to some default values:

```
        self.translation = [0,0,0]
        self.accelerometer = [0,0,0]
        self.gyroscope = [0,0,0]
        self.barometer = 0
        self.gps = [0,0]
        self.ypr = [0,0,0]
```

These variables are all used to store the last known readings of various sensors that the drone supports. These are all set to zero-values, and will be set by the control server when it receives incoming communications from the drone that is represented by a certain instance of this class.

Now that we have a representation of our drone, we need a way of linking this to an actual drone within the simulation (or potentially a physical one). For this, we need a function that can take a list of motor speeds and send them over a network to the target drone. We define this function like so:

```
    def set_motors(self, sock, mots):
        sock.sendto(bytes("M"+"/".join([str(x) for x in mots]), "ascii"),
            self.addr)
```

This function does a lot in a small amount of code. It iterates the list of motor speeds and for each value, it converts it into a string so that we can begin constructing our message. It then creates a single string by joining all of these values together with '/' characters, using Python's *join* function. After appending this string to the 'M' character, our message is fully constructed. For the meaning behind this specific format, refer to Section 6.1.1. We then need to convert that string into a *bytes* type, since a string in Python may be stored in various ways, and we need to make sure that it's in a suitable format that can be understood by any client on the network. We use the ASCII encoding as specified when designing the protocol. Once that has been done, we simply send this message to the socket connected to the target drone.

We're now at a position where we can command a drone to set its motors to a specific speed. We now need to implement our PID controller which can work out *what* we should set the motor speeds to. Let's start off by writing the constructor a PID controller class:

```
class PidController:
    def __init__(self, kp, ki, kd, setpoint, freq, minval, maxval,
            dowrap=False, minwrap=0, maxwrap=0):
        self.kp = kp
```

```
        self.ki = ki
        self.kd = kd
        self.minval = minval
        self.maxval = maxval
        self.setpoint = setpoint # Target value
        self.timeperiod = 1/freq
        self.err_i = 0 # Error integral (sum of previous errors)
        self.errprev = 0 # Previous error
        self.timeprev = time.time()
        self.output = 0
        self.dowrap = dowrap
        self.minwrap = minwrap
        self.maxwrap = maxwrap
```

At the moment, all this class does is store some values. Let's go through what some of the less self-explanatory of these are for. timeperiod is a variable that represents the minimum amount of elapsed time before the system will update its output. It's calculated from the frequency parameter that is given when a PID controller is constructed. Frequency is a more natural way to specify this value, but the period is more useful when writing code to check if the controller should update, which is the reason for this conversion.

err_i is used to store the integral error - it's essentially an accumulator where each time the controller updates, the current error is added to (or subtracted from) this value. Similarly, errprev stores the measured error from the last time the controller updated, so that we can calculate the derivative error.

All of the variables ending in "wrap" are used together to implement what we call input wrapping. We'll explain what this means in the context of the yaw rotation PID controller, which is the way we use this functionality in our program. The drone's yaw ranges from 0 to 2π, since it's in radians, but once its rotation passes 2π it loops back around to 0. This can cause an issue in a PID controller: the controller has no idea that a value of 0 is an equivalent rotation to 2π, and so if the current rotation was at 0.1π and the setpoint was 1.9π, the controller would cause a large positive error of 1.8π, whereas a far more efficient way of reaching the setpoint would be a negative error of -0.2π. To implement this behaviour, we can define the minimum and maximum bounds where the controller wraps around, and then the only aspect of the PID control calculation we need to change is how the proportional error is calculated, as this error is propogated to the other types of error.

To do this, we can simplify the problem. If input wrapping is enabled, we want to take the shortest route from the current value to the setpoint. This route could take one of three distinct forms:

- Current value $\rightarrow$ Setpoint
- Current value $\rightarrow$ Lower bound $\rightarrow$ Setpoint
- Current value $\rightarrow$ Upper bound $\rightarrow$ Setpoint

If we calculate the length of each of these routes, we should pick the one whose absolute magnitude is the smallest as the actual error.

We can now implement a function which calculates an output value using the PID algorithm. Let's start by implementing the most basic form of the algorithm, which simply computes the value every time it's called. In the following code, we're simply calculating a proportional error, adding it to the integral error, and working out the derivative error. We then multiply these different errors by their respective constants, and clamp the output withing the allowed output range. This works, but lacks the advantage of only updating the output at a fixed frequency.

```
def compute(self, value):
    err_p = self.setpoint - value
    self.err_i += err_p
    err_d = err_p - self.errprev

    self.errprev = err_p
```

```
        self.output = self.kp * err_p + self.ki * self.err_i + self.kd * err_d
        self.output = max(min(self.output, self.maxval), self.minval)

        return self.output
```

We can assume that this compute function will be called as frequently as possible, so we can change the function to the following to restrict the frequency that the output updates:

```
def compute(self, value):
    t = time.time()
    if t - self.timeprev >= self.timeperiod:
        dt = t - self.timeprev
        self.timeprev = t

        err_p = self.setpoint - value
        self.err_i += dt * err_p
        err_d = err_p - self.errprev

        self.errprev = err_p

        self.output = self.kp * err_p + self.ki * self.err_i + self.kd * err_d
        self.output = max(min(self.output, self.maxval), self.minval)

    return self.output
```

Each time the above function is called, it checks if the difference between the current time and the time that the controller last updated is enough that it should be calculated again. This ensures that the update frequency is at *most* the target frequency, however it could be a slower frequency if the compute function is not called frequently enough.

In addition to limiting the frequency of updates, we also scale the proportional error by the elapsed time (known as the delta time), to make sure the integral error isn't skewed if the update frequency happens to be too low.

All that's left now for the PID controller class is to implement the input wrapping. We implemented this by inserting the following code after the line where we calculate err_p:

```
if self.dowrap:
    # Error of: value -> left bound (right bound) -> setpoint
    err_left = (self.minwrap - value) + (self.setpoint - self.maxwrap)

    # Error of: value -> right bound (left bound) -> setpoint
    err_right = (self.setpoint - self.minwrap) + (self.maxwrap - value)

    err_abs = {abs(err_p):err_p, abs(err_left):err_left,
        abs(err_right):err_right}
    err_p = err_abs[min(err_abs)]
```

Here we first calculate the two alternate distances from the current value to the setpoint. Then, we construct a dictionary mapping absolute magnitudes of the distances to the raw distances themselves. This is to solve the problem that we want to pick the distance that has the smallest magnitude, but the value we want to actually use is the signed version. This dictionary solves this by allowing us to get the minimum magnitude and then simply use that as a key to find the value we want.

At this point, our PID controller is fully functional, and ready to integrate into our drone control code. First of all, we should construct the PID controllers that we'll need according to Figure 10.

```
    self.pid_yaw   = PidController(1.5, 0.2, 80, math.pi, 50, -200, 200,
        True, 0, 2 * math.pi)
    self.pid_pitch = PidController(10, 1, 200, 0, 50, -200,200)
    self.pid_roll  = PidController(10, 1, 200, 0, 50, -200,200)
    self.pid_alt   = PidController(8, 3, 200, 15, 50, -200,200)
```

As shown in the PID controller's constructor, the first three parameters for each controller refer to K_p, K_i, and K_d. At this time, these values have been found empirically. A future extension to this work could involve automatic PID tuning to find these values. Following those parameters is the setpoint: the beginning setpoint for the drone is set here to have an altitude of 15 units, 0 pitch and roll, and a yaw of π. These are simply sensible defaults, and can of course be changed by the operator at runtime, or programatically by accessing the PID controllers as member variables within the drone.

All of the controllers operate at the same frequency of 50Hz. Too high a value here can lead to congestion, but too low a value could potentially cause the control to be less precise and stable. The PID update frequency is something we will test in the evaluation stage to see how much it affects the performance. We will try to find the lowest possible frequency that still allows the drone good movement, because the less messages we send, the less power used by the transceiver.

For the yaw controller, we use the constructor's optional parameters to enable input wrapping. As mentioned earlier, we want this to wrap around the full circle, which is $0 \rightarrow 2\pi$ radians. It would be reasonable to use the same input wrapping scheme for pitch and roll, but during normal operation these values should never have to wrap. If they did, it would mean the drone had been upside down, which we don't want.

Now that we have these controllers, we can write a function to compute and set the motors of a particular drone:

```
def compute(self, sock):
    yaw = self.pid_yaw.compute(self.ypr[0])
    pitch = self.pid_pitch.compute(self.ypr[1])
    roll = self.pid_roll.compute(self.ypr[2])
    thrust = self.pid_alt.compute(self.translation[1])

    mots = typr_to_motors(thrust, yaw, pitch, roll)
    mots = [round(m, 3) for m in mots]
    self.set_motors(sock, mots)
```

First of all, we have to update all of the PID controllers using their compute methods. Since this is where we tell the controllers to update, the drone's compute method must therefore be called as often as manageable to ensure the PID controllers can update whenever it's time.

Once we have the desired values for yaw, pitch, roll, and thrust, we convert these into motor speeds. The method for this conversion has already been explained in the design stage of the control server. The implementation of typr_to_motors is as follows, and is almost identical to the equations shown previously when explaining the conversion.

```
def typr_to_motors(thrust, yaw, pitch, roll):
    return [
          yaw - pitch + roll + thrust,
        - yaw - pitch - roll + thrust,
        - yaw + pitch + roll + thrust,
          yaw + pitch - roll + thrust
    ]
```

We also have to round the motor speeds to three decimal places before sending them to the drone over the network. This was specified in the protocol we designed earlier, to make sure that the message will never be too long to send if we want to use a LoRa transceiver in the future. After we've done that, we invoke the set_motors method to send the computed list of motor speeds to the corresponding drone.

Now that our drone class is complete, we can work on incorporating it into the drone control server itself. We'll make a class for the server called DroneServer. The most important member variables that this contains are: drones, which is a list of Drone instances; and sock, which is the socket that the server listens for new drone connections on.

When a drone server is instantiated, it binds a new socket on a port that is supplied as a parameter to its constructor. It then spawns a thread which executes a function named run. This function goes into an infinite loop which constantly waits for incoming messages. Each time it gets a message, it determines which type it is.

If the message is a new drone requesting to be controlled, the thread makes a few checks to determine whether that drone should be added to the list of drones. First, the server has a maxi-

mum number of drones it can control, that can be set when an instance of the server is created. If there is no space for another drone according to this number, then the join request will be ignored. Furthermore, the server needs to ensure that there is no existing drone in the list with the same address (matching IP address *and* port) as the new one. If both of these checks are passed, then we send the string "OK" to the drone to let it know it's been accepted to join the system, and to expect motor control messages soon. We also create a new Drone instance, and add it to the list of drones.

When we create a new drone instance, we must supply it with a unique ID. Our function for this is very simple:

```
def get_unique_id(self):
    self.d_id += 1
    return self.d_id
```

d_id is an integer member variable which, when the server is created, is initialised to 0.

If, when the server receives a message, it instead begins with the character 'S', we interpret it as sensor readings. Before decoding the message, we first check that the address that we received the message from is present in our list of drones. This ensures that the drone sending the message has joined the system in a correct way. The code for decoding a sensor message is as follows:

```
spl = msg[1:].split(":")
for sens in spl:
    if sens[0] == "G": # Gyroscope
        sender.gyroscope = [float(n) for n in sens[1:].split("/")]
    if sens[0] == "A": # Accelerometer
        sender.accelerometer = [float(n) for n in sens[1:].split("/")]
    if sens[0] == "B": # Barometer
        sender.barometer = float(sens[1:])
    if sens[0] == "P": # GPS
        sender.gps = [float(n) for n in sens[1:].split("/")]
    if sens[0] == "R": # Rotation (yaw/pitch/roll)
        sender.ypr = [float(n) for n in sens[1:].split("/")]
    if sens[0] == "T": # Translation
        sender.translation = [float(n) for n in sens[1:].split("/")]
```

The first line here takes the whole message excluding the first, 'S', character, and splits it into a list, separating elements wherever the ':' character is found. The logic for decoding most of the sensor types is identical: split the sensor substring with the '/' character as a delimiter, convert each element to a float, and set the corresponding member variable in the object that represents the drone which the message was received from. The only sensor that deviates from this pattern is the barometer – it's only one value, so there's no need to turn it into an array.

The server creates another thread just after it starts this one: a thread to continuously update the motors of all of the connected drones. This thread has a far shorter implementation than the other one, because all of the motor control logic is contained within the drone class. Here's this thread's code:

```
def compute_drones(self):
    while True:
        for drone in self.drones:
            drone.compute(self.sock)
```

This thread, like the first one, goes into an infinite loop. Each time it loops, it simply calls each drone in the list to update itself.

At this point, we have all of the basic functionality implemented: our control server is able to accept new drones to the system, read sensor messages from any of them, and compute and send motor commands back to them. The only functionality that is currently missing is the ability for an operator to control the setpoints of the drones. First, we have to decide on a format for the messages that the operator sends. The format needs to convey three things: the ID of the drone to control, the particular setpoint to change, and the value to change it to. The message format we decided on is the following: "<drone ID> <identifier of setpoint to change> <value>". The drone ID should be able to be converted into an integer. The setpoint identifier is a single character - one of:

- 'y' - to set the target yaw.

- 'p' - to set the target pitch.
- 'r' - to set the target roll.
- 'a' - to set the target altitude.

The value is simply a number that the operator wants to set that setpoint to. An example message would be "2 r 0.2", which would set the target roll orientation of the drone with an ID of 2 to 0.2. This message is written as simple human readable ASCII so that the system can be used both by someone simply sitting at a terminal typing commands, or equally by someone writing a program to generate these commands and send them.

In our control server class, we use the main thread to continuously wait for command line input, and then decode it according to this message format. The code for this is as follows:

```
inps = inp.split(" ")
value = float(inps[2])
drone = None
for d in self.drones:
    if d.id == int(inps[0]):
        drone = d
if drone != None:
    if inps[1] == "y":
        drone.pid_yaw.setpoint = value
    if inps[1] == "p":
        drone.pid_pitch.setpoint = value
    if inps[1] == "r":
        drone.pid_roll.setpoint = value
    if inps[1] == "a":
        drone.pid_alt.setpoint = value
```

First of all, this message format is delimited by space characters, so we split it into a list of substrings with this delimiter. The first element of that list is expected to be the ID of the drone, so we iterate the list of drones until we find one whose ID matches. If one isn't found, we simply ignore the rest of the message, since we have then been asked to control a drone that doesn't exist. If we *did* find a drone, we look at the second element in the list, which is the identifier of the setpoint we want to change. Depending on this, we set different member variables of the drone object that we found in the first stage to the value, which is the third element of the list. Before using the value, we have to convert it to a floating point.

We also implemented another type of message: if the operator sends the 'q' character, the server quits cleanly by exiting the infinite loop.

At this point, our control server implementation is complete, and we're ready to test it and evaluate how well it works in varying situations.

8 Evaluation

There are several aspects of the system that we want to test:

- How packet loss affects the performance.
- How latency affects the performance.
- How latency jitter[11] affects the performance.
- Which PID controller parameters work best.

Latency and packet loss are useful to test because we expect these will be the main differences between a real life drone system and our simulated one. The latency and packet loss of the simulated system that we've developed in this dissertation is negligible because, although the messages are sent over a network interface, they go between processes on the same computer. In the real world

[11] Jitter is the range over which the latency varies.

we expect that we would use LoRa to send data between the drones and the control server, which as discussed earlier can reach a packet loss of around 20%[3]. The latency of a LoRa transceiver is very low since its carried over radio waves, which move at the speed of light. Latency here is caused instead by processing time on each end: if a server takes a long time to prepare the data to send, the latency is effectively higher. We don't expect the latency to be higher than around 5ms, but we will test much higher latencies than this to see how well the system would perform if it used a potentially higher latency interface.

Since the network interface we are using for the simulation has such a low latency, we will need to introduce artificial latency to the system. We previously discussed methods of achieving this by adding delays either in the server code or in the drone code, but there is another way to consider.

The Linux kernel contains a component called *netem*, which can be used to simulate varying amounts of latency and packet loss when sending packets to an interface. This component is controlled via a commandline tool called *traffic controller* (the executable is named *tc*), which can use netem as part of the queue discipline (known as the qdisc) of any given interface. A qdisc can be thought of as a scheduler, which buffers packets sent to an interface and decides when (or if) they should be sent. The default one which is normally used acts just like a FIFO (first-in, first-out) buffer, but we can tell the system to use a netem qdisc instead, allowing us to modify these network parameters. To do this, we ues this command:

```
$ sudo tc qdisc add dev lo root netem delay 20ms loss 2%
```

This commands job can be written in plain english as "add a netem qdisc to the device called 'lo' (the loopback device) to add a delay of 20ms and a packet loss of 2%". The loopback device is that which is used when we try and send a packet through a socket which has both ends on the same device. It's worth noting that the latency only affects outgoing messages; qdiscs are not designed for affecting incoming messages. We can see if this has worked by pinging our own computer:

```
$ ping 127.0.0.1
PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=40.2 ms
64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=40.1 ms
64 bytes from 127.0.0.1: icmp_seq=4 ttl=64 time=40.2 ms
64 bytes from 127.0.0.1: icmp_seq=5 ttl=64 time=40.2 ms
64 bytes from 127.0.0.1: icmp_seq=6 ttl=64 time=40.3 ms
64 bytes from 127.0.0.1: icmp_seq=7 ttl=64 time=40.2 ms
```

A normal time for pinging ones own computer is far less than 1ms. We can see that it's worked, but it may seem surprising that the time is actually around 40ms instead of 20ms even though only outbound packets are delayed. This is because we not only have to wait for our ping request packet to get sent, but we also need to wait for the ping response.

8.1 Evaluation Metrics

We need to plan some metrics which we can use to usefully evaluate the drone control system. These should test aspects of the system that are relevant to its operation in the real world. The evaluation metrics that we plan to use are listed in Table 1. These metrics will all be tested with varying amounts of packet loss, latency, and jitter. After this, we will use a different evaluation method to determine better parameters for the PID controllers.

8.2 Setting up the evaluation environment

For each test we need to ensure that the only differences are the network parameters that we're investigating. Luckily since we're simulating the drones this is easy to do: we can simply use the same initial drone position and orientation each time, and send the same commands at the same times.

We need to decide which combinations of network parameters we'll test. It would be useful to test ranges of each parameter separately so that it's clearer to see how they affect the drone performance, but it would also be interesting to see how they act together. We will test values of latency from 0ms to 800ms, running more tests at the lower end of that range as we expect that will be where there's the most change in behaviour. We will test packet loss percentages from 0% to 30% – packet loss

#	Metric	Purpose	Method to test
1	Is the drone able to hover in place?	Check if the drone can perform the most basic flying.	Qualitatively determine whether the drone can hover in place. If it can hover for one minute without falling over or moving, it passes.
2	How quickly can the drone yaw $180°$	We want to see how varying network conditions affect the responsiveness of the system. This metric allows us to see if these conditions affect the time it takes for one of the PID controllers to reach its setpoint.	Once the drone is hovering stably, at a yaw of $0°$, we will change its yaw setpoint to $180°$. We'll start a timer when we issue the command, and stop the timer when the drone's yaw oscillations are all within $\pm 5°$ of the setpoint.

Table 1: The evaluation metrics that we will use.

higher than 20% isn't expected with a LoRa transceiver setup, but it is possible. Finally, we'll test jitter values from 0ms to 100ms.

To perform these evaluations, we added some code to the drone controller class that can write sensor data to a file every time it receives it from a drone. Using this data, we can plot graphs of the drone's trajectory after the simulation has ended.

8.3 The Results

The results of running the above tests for many various combinations of network parameters are presented in Table 2.

Although it's common to perform three trials for tests like these, our simulator is deterministic if it has the same starting state. For this reason, each of these tests was only performed once.

The raw data used to calculate these results is available as an appendix. Each file of results is formatted as a long list of pairs of timestamps and yaw rotations.

Control Variables			Metric Results	
Latency* (ms)	Packet Loss* (%)	Jitter* (ms)	1	2
0	0	0	Pass	4.2672
20	0	0	Pass	4.5021
100	0	0	Pass	6.7467
800	0	0	Fail	–
0	5	0	Pass	4.1539
0	15	0	Pass	4.3170
0	30	0	Pass	7.2107
0	0	10	Pass	4.3750
0	0	25	Pass	4.6402
0	0	100	Pass	5.9888
20	5	0	Pass	5.7300
20	30	0	Pass	4.3243
100	30	0	Fail	–
800	5	0	Fail	–
20	0	10	Pass	4.4344
100	0	100	Fail	–

Table 2: The results of the evaluation of how varying network parameters affects the system performance.

*Note that these are the network problems that we have *artificially introduced*. This is on top of the very small (roughly 0.05ms) latency present on my system. Under normal circumstances there is no packet loss on the loopback network interface, which we're using.

8.4 Analysis of Results

First, we'll discuss *why* each network condition may affect the performance of the drone.

If the network has a high latency, it will be less responsive. As an example, if a drone is trying to stay straight but is starting to slightly tilt, its sensor readings that it sends to the server will say that it's slightly off-centre. The problem is that by the time the server receives this message, the drone will have tilted even more, meaning that the motor control messages that the server send back to the drone will be out of date by the time they reach it.

A packet loss low enough will not affect the functionality of the drone too much because of the high rate at which messages are sent between the server and the drone. If the packet loss is too high, however, we run into a similar problem as with high latencies: the server takes too long to respond to the drone if many of the messages are not being received.

Jitter can affect the performance of the drone control in a more subtle way. Since so many packets are being sent in a short period of time, the variance in latency can effectively *reorder* the sensor and motor control packets quite dramatically. If the packets containing sensor readings are reordered, the control server will think the drone is moving in a way different to how it is. If the motor control packets are rearranged, the drone will be controlled to move in an erratic way, which can be seen watching the drone try to fly with a high jitter.

From the results we got, we can see that at a reasonable latency that we might expect from a LoRa transceiver the drone should be capable of flying in a stable manner. It's only when the latency reaches hundreds of milliseconds (which should not be the case for our drone in the real world under normal circumstances) that, paired with a high packet loss, the drone can become unstable and not able to fly.

The only time that latency alone caused the drone to become unstable was when it reached 800ms, which being almost a second is far higher than we would expect to see in a real life situation.

Peculiarly, when the only problem with the network was a packet loss of 5%, the time for the drone to settle to its setpoint was actually *faster*. It's unclear what causes this to happen, but our theory is that with just a *few* packets lost, the drone may in fact overshoot its target by less, causing it to settle down faster. This result was surprising, though, and so it was exciting to see how a 5% packet loss affected network conditions with varying amounts of latency. Once the packet loss reached 15%, the settling time was worse than with no packet loss. This is what we expected. We also see a steep rise in settling time when the packet loss reaches 30%. This is due to the yaw having to oscillate one more time than otherwise since it just barely overshot the 5° limit for us to say that the yaw has settled sufficiently.

The results when jitter is the only condition that we changed are closer to what we would expect. When we introduce a small amount of jitter, the settling time slightly increases. It then simply increases more as we raise the jitter amount.

Recalling that a 5% packet loss improved the settling time when it was the only condition, we might expect a 5% packet loss paired with a 20ms latency to be better than 20ms latency on its own. This isn't the result we get though: the settling time for this combination is around a second higher than 20ms latency on its own. It's quite unclear why a 5% packet loss only improves the performance at very low latencies. If our previous theory about the slight packet loss causing the PID controller to overshoot by less, then even a slightly high latency could counteract this benefit by delaying the "pulling back" of the yaw rotation, causing it to overshoot more than it otherwise would.

Overall, these results show promise that a system like this one would scale well to a real world application. Further work would need to be done to find out if this is truly feasible - this would involve building a drone and writing firmware for it to communicate with this control server. There is no reason, however, to think that it wouldn't work.

9 Reflections

Overall, we achieved the majority of what we set out to do with this project. Of course some aspects when better than others, which we shall discuss.

9.1 The Most Successful Aspects

After much research into related work, it appears that this idea has not been looked into previously. From our testing of our simulated system as well as research into hardware that could be used for a real world drone, it seems that this control method could be genuinely useful where a fleet of low-cost drones are needed to fly for as long as possible. One use case of this was discussed earlier: a delivery system. There are many other use cases, for example a light show using drones with coloured LEDs on them, as well as a fleet of drones used to follow someone around and video them. Our research does indeed show that this method could well replace traditional models of drone control.

I was also very pleased with the elegant method of allowing third party software to interface with the drone control server, by piping data to the server's *stdin* buffer. It's very useful to allow people to use our work without having to read and modify our code, and without even having to use the same programming language.

It was also satisfying to write an implementation of the PID algorithm. This is something that I haven't needed to do before, but it has inspired me to plan several other projects using similar control systems for the future (one of which being a physical drone for which I can implement our system).

9.2 Less Successful Aspects

It was unfortunate that we weren't able to build a real world drone for this dissertation. It was simply out of scope since it would be primarily electronic and mechanical engineering, and we sadly did not have time. We worked around this by still considering how this system *would* scale to a physical drone, like planning what sort of physical interface it would use to communicate with the control server.

Another aspect of the system that would have been nice to implement was an extra step in the PID control chain to allow the drone to automatically move to a given set of coordinates, rather than controlling only the yaw, pitch, and roll. This would not be complex to do - for the X and Z axes we would introduce new PID controllers whose outputs affect both pitch and roll to cause the drone to move around.

9.3 Future Developments

This dissertation does not mark the end of development on this idea. There are many improvements and new features that we plan to add in the future.

As it stands, our network protocol is insecure. There are no security features built in, and so a malicious actor could trivially pose as a control server and control all of the connected drones. For our research purposes this is not a problem, but if this system is to replace conventional methods we need to discuss how it could be made more secure.

This system is a lot easier to secure than, for example, a client-server connection for an email server. This is because the server operator has physical access to each drone that will connect to it. If we want to encrypt all of the messages travelling over the network, we don't have to use more complex systems like public-key encryption and Diffie-Hellman key exchange, because each drone can simply be programmed directly with a key. Symmetric key encryption can then be used, where this key is used both for encryption and decryption of messages. This even prevents man-in-the-middle attacks, since for someone to run such an attack on this system they would need to intercept the process of loading the key onto the drone. This would be prohibitively difficult because the drone operator would be physically next to the drone while doing this.

Another improvement to the system would be to allow more customisation on a per-drone basis. As it stands, the system only works with one type of drone: a quadcopter with specific mass and motor powers that work well with the PID parameters we've picked. Our system would ideally be able to support drones with various amounts of motors (quad-, hexa-, and octacopters, and perhaps even regular helicopters). For it to work with different drones, different PID parameters would have to be able to be supplied for each type of drone. A database could be created so that the system has some predetermined parameters for popular drone models to make it easier for new users, for example.

On the topic of PID control, it would be useful if there was some setting where the control server would attempt to automatically tune the PID controller parameters for a given drone. It can be

tricky to do this automatically, since if the server tries some parameters unguided by a human it could lead to the drone crashing and breaking.

Finally, it would be useful if when a drone starts up it could search for the control server it needs. IP addresses can change, but even if the server has a static IP address, it can be clunky for the drone operator to need to modify the drone firmware to change which IP it connects to if needed. To help with this, we could devise a new protocol for drones to discover which control servers exist. If the communication happened over a LoRa connection, for example, the drone could scan each LoRa channel and this new protocol would allow servers to give themselves a name. A drone could then, when powered on, scan the network for a control server of a specific name to connect to.

References

[1] Yugal K. Singh et al. “Temperature Control System and its Control using PID Controller”. In: *INTERNATIONAL JOURNAL OF ENGINEERING RESEARCH & TECHNOLOGY (IJERT)* (2016). URL: https://www.ijert.org/temperature-control-system-and-its-control-using-pid-controller.

[2] Karl Johan Åström and Tore Hägglund. *Automatic Tuning of Simple Regulators*. eng. Tech. rep. Department of Automatic Control, Lund Institute of Technology (LTH), 1984. URL: https://lup.lub.lu.se/search/files/47932505/7269.pdf.

[3] Martin Bor, John Edward Vidler, and Utz Roedig. “LoRa for the Internet of Things”. In: *EWSN ’16 Proceedings of the 2016 International Conference on Embedded Wireless Systems and Networks*. 2016.

[4] *Different PID Equations*. URL: https://control.com/textbook/closed-loop-control/different-pid-equations/.

[5] *eRA-LoRa Long Range Datasheet*. URL: https://docs.rs-online.com/87a2/0900766b8162322c.pdf.

[6] *GDScript basics — Godot Engine (stable) documentation in English*. URL: https://docs.godotengine.org/en/stable/tutorials/scripting/gdscript/gdscript_basics.html.

[7] R. Kebriaei et al. “Numerical modelling of powder metallurgical coatings on ring-shaped parts integrated with ring rolling”. In: *Journal of Materials Processing Technology* 213.11 (2013), pp. 2015–2032. ISSN: 0924-0136. DOI: https://doi.org/10.1016/j.jmatprotec.2013.05.023. URL: https://www.sciencedirect.com/science/article/pii/S0924013613001908.

[8] N. Minorsky. “DIRECTIONAL STABILITY OF AUTOMATICALLY STEERED BODIES”. In: *Journal of the American Society for Naval Engineers* 34.2 (1922), pp. 280–309.

[9] SungTae MOON et al. “Development of Multiple AR.Drone Control System for Indoor Aerial Choreography”. In: *TRANSACTIONS OF THE JAPAN SOCIETY FOR AERONAUTICAL AND SPACE SCIENCES, AEROSPACE TECHNOLOGY JAPAN* 12.APISAT-2013 (2014), a59–a67. DOI: 10.2322/tastj.12.a59.

[10] *MPU-6050 Invensense, MEMS Module, MotionTracking Series, 3-Axis Gyroscope/Accelerometer | Farnell*. URL: https://uk.farnell.com/invensense/mpu-6050/gyro-accel-6-axis-i2c-qfn-24/dp/1864742.

[11] *PacketPeer — Godot Engine (stable) documentation in English*. URL: https://docs.godotengine.org/en/stable/classes/class_packetpeer.html#class-packetpeer-method-get-available-packet-count.

[12] Nuno Paula et al. “Multi-drone Control with Autonomous Mission Support”. In: *2019 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*. 2019, pp. 918–923. DOI: 10.1109/PERCOMW.2019.8730844.

[13] *PhysicsBody - Godot Engine*. URL: https://docs.godotengine.org/en/stable/classes/class_physicsbody.html#class-physicsbody.

[14] *Power Consumption Benchmarks | Raspberry Pi Dramble*. URL: https://www.pidramble.com/wiki/benchmarks/power-consumption.

[15] Eric S. Raymond. *Basics of the Unix Philosophy*. URL: http://www.catb.org/~esr/writings/taoup/html/ch01s06.html.

[16] Shital Shah et al. “AirSim: High-Fidelity Visual and Physical Simulation for Autonomous Vehicles”. In: *Field and Service Robotics*. 2017. eprint: arXiv:1705.05065. URL: https://arxiv.org/abs/1705.05065.

[17] J G Ziegler and N B Nichols. “Optimum Settings for Automatic Controllers”. In: (). DOI: https://web.archive.org/web/20170918055307/http://staff.guilan.ac.ir/staff/users/chaibakhsh/fckeditor_repo/file/documents/Optimum%20Settings%20for%20Automatic%20Controllers%20(Ziegler%20and%20Nichols,%201942).pdf.