Thales – Stay Connected

Adela Simion - s2676451
Lars Wijntjes - s2551071
Hayo van de Werfhorst - s2592975
Lars Gortemaker - s2977516
Huanbo Meng - s2653168
Luca Vlăsceanu - s2991772

Abstract

Naval communication systems using, for example, satellites, are costly, prone to delays, and have limited bandwidth. To overcome these limitations, Thales proposes a drone-relay system within MaritimeManet leveraging deployable UAVs to dynamically extend and stabilize communication links between ships. This paper presents the design and implementation of a distributed drone relay system within a virtualized simulator developed to test concepts for marine communication technology. The core goal of this project is a fully automated drone control algorithm that offers new smart functionality for both ships and drones.

This project demonstrates that using drones is a viable alternative to traditional naval communications systems, allowing ships to communicate even at great distances.

1. Introduction

The project "Stay Connected" proposed by Thales aims to maintain ship-to-ship connectivity with the help of deployable drones that act as relays. Marine operations refer to the planning, execution, and management of activities conducted on water and coastal environments using ships and other related infrastructure such as coastal control towers. In this environment, communication becomes challenging because on-land antennas cannot be reached from deep-sea locations. A good mission entails constant communication between ships and other units. To speed up the testing of new marine communication technologies at Thales, a virtual simulator has been developed. This paper describes the implementation of a new autonomous drone deployment system within this existing virtual simulator.

1.1. Company Background

Thales Group is a global technology company focused on offering defense solutions across maritime, aerial, and digital spaces. It designs and integrates advanced systems that fuse sensors, secure communications, cybersecurity, and AI to help users detect, decide, and act in complex environments. From avionics and radars to rail signaling, satellite payloads, and digital identity solutions, Thales supports civilian and military customers worldwide with a strong emphasis on safety, reliability, and sovereignty. Our client is from the Hengelo branch, which specializes in marine operations.

1.2. Existing Solution Overview

Maritime operations face the challenge of losing connectivity between ships due to vast distances and constantly changing network topology. Current solutions like satellite connectivity depend on third-party providers, cost significantly, and suffer from delays, making them inadequate for the substantial bandwidth required by ships.

Networking technologies such as MaritimeManet, developed by Thales, expand a ship's connectivity range using Multi-Beam Antennas. Because of its limited range, the traditional 360-degree Wi-Fi radius offered by an antenna is compressed into a beam to extend the range. For example, using this different approach, a small ship that would have had quite a limited 360-degree radius now has six antennas with each of its beams angled at 60 degrees to achieve a full, much more potent 360-degree radius. Off-the-shelf Wi-Fi modules are low-cost and are able to provide sufficient bandwidth for long-distance connectivity. With the use of the BATMAN routing algorithm and MANET's (Mobile Ad-Hoc Network) adaptability to topology change, the network dynamically configures based on the position of the ships.

The traditional solutions suffer from delays because they rely on satellite connections, as well as being expensive. Even with the use of directional antennas, the bandwidth is still highly dependent on the distance, as well as rotation between peers. If two ships are in connection and then move away from each other, the connection will get weaker or break completely. A possible solution to prevent such a loss of connection and keep the ships connected is to deploy a drone as a relay.

1.3. Virtual simulator

Because of the expensive and time-consuming nature of MaritimeManet, a Digital Twin (simulator) was developed that uses virtual machines (VMs) to simulate network behavior. The simulator illustrates how ships equipped with omnidirectional MBAs interact in the ocean. The communication between the ships is managed by MaritimeManet, which is a novel Mobile Ad Hoc Network (MANET) that uses a modified version of BATMAN-adv that operates on the MAC layer. Those technologies ensure reliable communication over the ocean's large distances without the need for additional physical actors. The simulator contains two parts: front-end and back-end. It consists of a web interface where the ships are added, and their settings configured (change speed, direction, number of antennas, etc.), as well as adding "waypoints" which represent the destination of a ship. After clicking the "start" button, the backend interprets the data from the front-end and allows the ships to start moving to their waypoints, and when in range, communicate using the MaritimeManet networking technology. Fig. 1 shows a screenshot of the simulator during the set-up stage, with two small, one medium, and one large ship. In this

example, the large ship has a more complex route than the rest of the ships. After clicking “Simulate”, Fig. 2 shows the new shape, the triangle, which represents drones. There is no manual control over the drones as all actions occur autonomously.

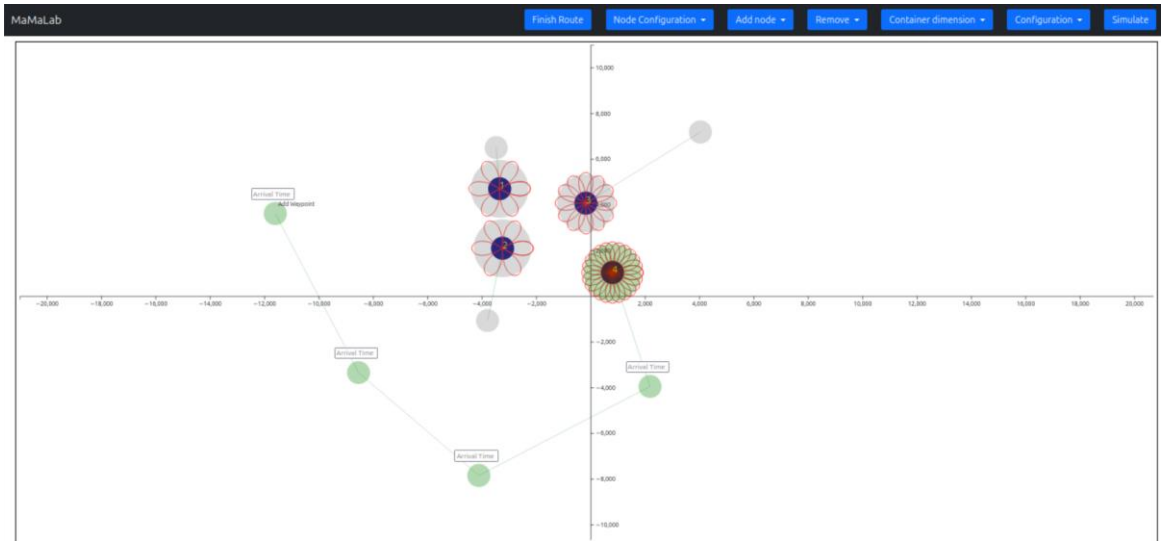

Figure 1 Virtualized Simulator – Set up

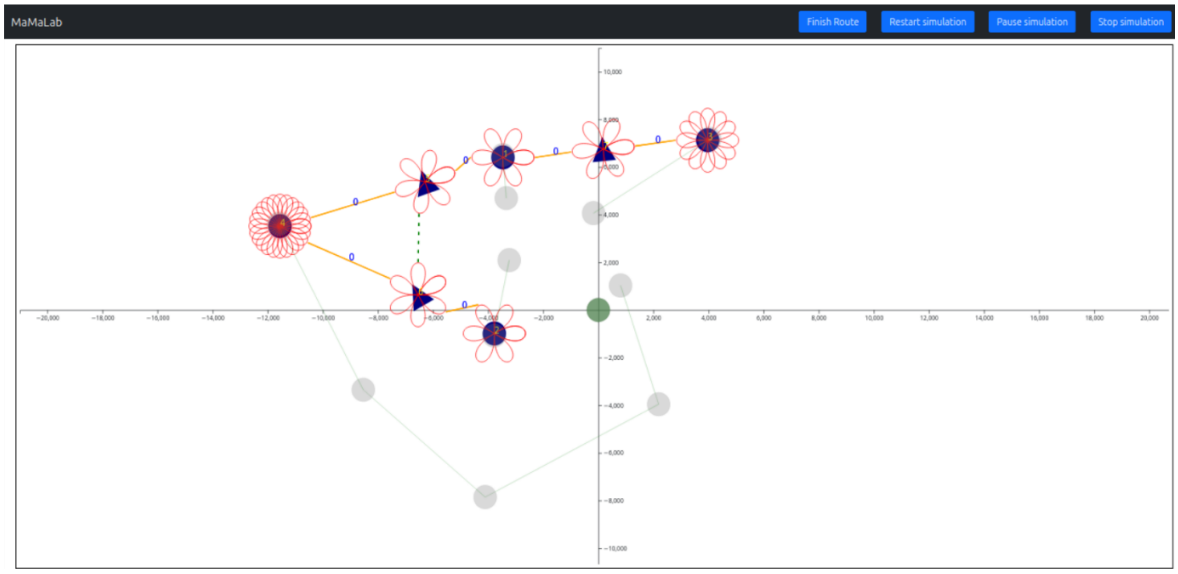

Figure 2 Virtualized Simulator – Ongoing Simulation

1.4. Assignment description

Our assignment is to design and implement an autonomous drone control system that maximizes communication capacity between ships. We were tasked with designing an algorithm that runs on each drone. Separately from the algorithm, drones should monitor the state of nearby peers, share that state with other drones, process this data internally, and act upon this, such as repositioning based on this data. Implementation will use the existing virtualized simulator, and a simple demonstration will verify correct operation.

We will start by describing the system requirements as they were drafted together with the client, then we will explain the chosen design based on these requirements. After this, we will dig into the code and show how the components interact together to achieve our desired functionality, as well as the testing methodology used. We will then describe some potential further work and finally evaluate our performance as a team, by both comparing our proposed implementation trajectory with the actual trajectory as well as reflecting upon our experience.

1.5. Terminology

Before going into the technical details, we will establish some definitions to help the reader understand our terminology:

1. **Node –** unit**,** ship or drone
2. **Peer** – nodes which we have a wireless connection with
3. **Link** – wireless connection between two units
4. **Path** – possible link between two units, consisting of one or more links
5. **Graph(s) -** Network, a group of connected units
6. **Cut link -** a path for which, if disconnection occurs, the network splits into two isolated graphs (subnets)
7. **Mothership –** the source ship, the original ship a drone has been deployed from
8. **Target ship –** the destination ship, the ship a drone is moving towards
9. **Trajectory** – the path or route a ship sail over time
10. **Responsibility** – nodes a drone is responsible for relaying data for
11. **RSSI** = Received Signal Strength Indicator
12. **Deploy Threshold** – the minimum RSSI value at which a drone should be deployed
13. **Recall –** drone returns to its mothership

In our project, specifically in the Digital Twin, ships are represented by circles with six, eight, or twelve beams (small, medium, or large, respectively). Drones are represented by triangles and, similar to small ships, have six beams and the same communication range. Below in Fig. 3, we illustrate our system, with the relevant definitions attached as numbers.

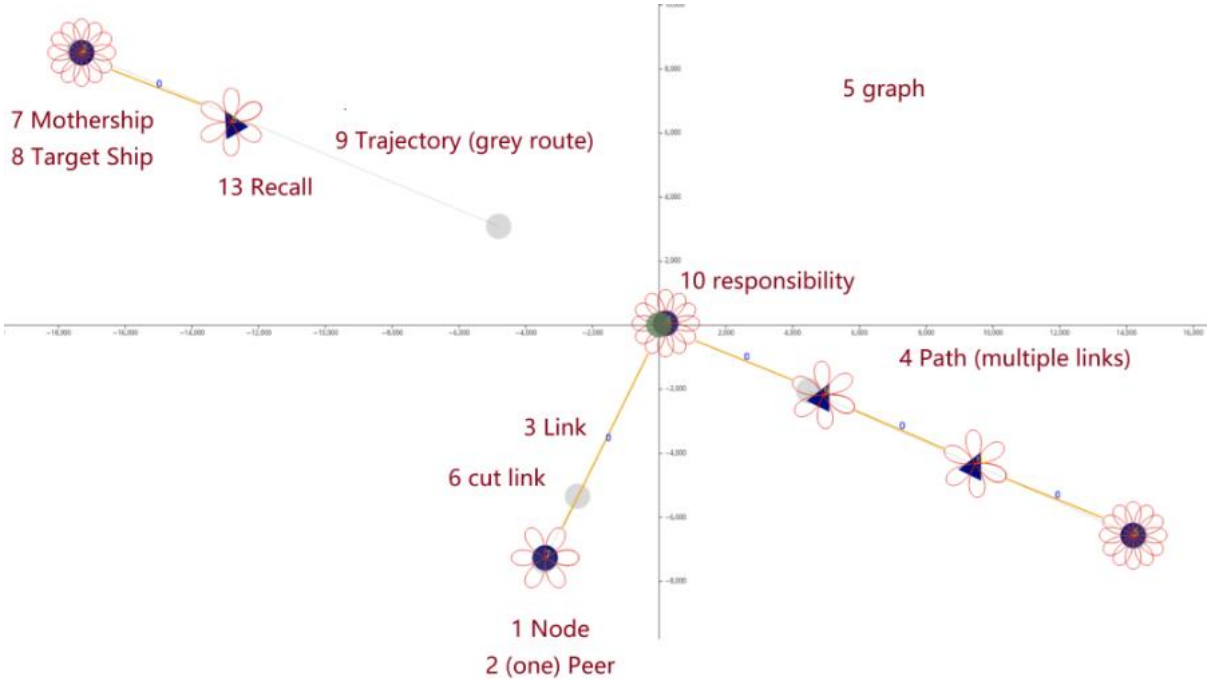

Figure 3 Terminology Example

2. System Requirements

Autonomous drone deployment within the existing simulator requires changes on two main fronts: modifying the existing virtualized simulator and implementing a new distributed drone algorithm. We first drafted these requirements with the client during the project proposal phase, and the following design is based on these. The system requirements cover both the simulator and the drone algorithm.

They are organized using the MoSCoW method. The labeling for each functional requirement ID is done using the format:

"Algorithm/Simulator - #id". For example, the id "A-2" means that it is a requirement related to the Algorithm and is number #2.

Nonfunctional requirement IDs don't have such labels.

2.1. Must Have

ID	Requirement	Description
A-1	When a link is weak, a drone must be deployed to relay the connection.	This is the main goal of the system.

A-2	A drone must adapt its position autonomously.	This is one of the main goals of the system.
A-3	A drone must provide the maximum available bandwidth.	This is one of the main goals of the system.
S-1	There must be new drone nodes as part of the simulator backend.	This is a necessary precursor to implementing an algorithm.
S-2	Drones and ships should have separate functionalities as part of the simulation.	This is a necessary precursor to implementing a drone specific algorithm.
S-3	The simulator must allow drones to move based on an algorithm instead of a waypoint route.	This is a necessary precursor to implementing a drone specific algorithm.
S-4	The simulator must allow drones to have knowledge of the signal strength and connection state of peer nodes.	This is a necessary precursor to implementing a drone specific algorithm.
S-5	A drone must be deployed only when there is an already existing connection between two ships.	This is necessary based on the existing MaritimeManet functionality.

Table 1 Must have requirements

2.2. Should Have

ID	Requirement	Description
A-4	If a deployed drone has only one connection, it should return.	This is needed so drones don't remain "stranded". This would result in them getting lost in real life.
A-5	When a ship has a better connection on another path than through its drone, the drone should be recalled.	This is such that we don't let drones roam if they are not needed anymore.
A-6	Drones should deploy only when the breaking link is a cut link.	This is such that we don't deploy drones if ships have another communication path.
A-7	Drones should autonomously find the position with the maximal signal strength.	This is one of the main requests of the client for this algorithm.

S-6	A drone should have the ability to launch in a specified direction as part of the simulation.	This is a necessary precursor to allow drones to run autonomously.

Table 2 Should have requirements

2.3. Could Have

ID	Requirement	Description
A-8	Drones return to their mothership when their energy level is low.	This was one of the ideas proposed by the client at the beginning of the project.
A-9	Drones should coordinate and move autonomously to the most efficient positions as a group.	This was one of the ideas proposed by the client at the beginning of the project. (future works)
A-10	If a connection can benefit from a larger bandwidth, a drone could be deployed.	Right now, we deploy on an almost breaking threshold, but it could be deployed on a more sensitive threshold based on this requirement.

Table 3 Could have requirements

2.4. Won't have

ID	Requirement
S-1	The system won't be tested in a real-life scenario.

Table 4 Won't have requirements
For simplicity, we omit the rest of possible "won't" haves.

2.5. Non-functional

ID	Requirement	Description
1	The system must be able to maximize the connection of at least two ships.	At least two, a requirement of the main problem
2	When a drone is deployed, it will seek to restore bandwidth to a minimum of 1Mb/s. If this is not possible with the current number of drones, an additional drone should be deployed.	Client says that ships do video communications, and these usually require 1Mb/s for optimal relay.
3	Drones must look different than ships in the simulator.	There must be a visual difference, so they are differentiable in the simulation
4	Drones should monitor the connection state at a rate of 500ms.	This is a requirement which may be more relevant in real life. In the

		simulation it happens almost constantly
5	The simulator should allow at least one drone onboard a ship.	There should be at least one drone. In our case it's one for any type of ship
6	The drone should have the same number of beams as a small sized ship.	The configuration of a drone is the same as that of a small ship

Table 5 Non-functional requirements

3. Approach

This chapter is dedicated to describing a high-level overview of how the solution to the main goal of the project will be achieved. After a brief introduction to the infrastructure of the existing system, we will discuss the design process of the simulator and algorithm, respectively.

3.1. Existing architecture

At first, it was difficult to understand how the simulator works. From reading previous reports, we found that the architecture of the simulator is data centered. The rationale behind this design decision was to have consistent data across the system and minimal component coupling with maximum scalability. The figure below shows how the simulator was structured prior to our changes.

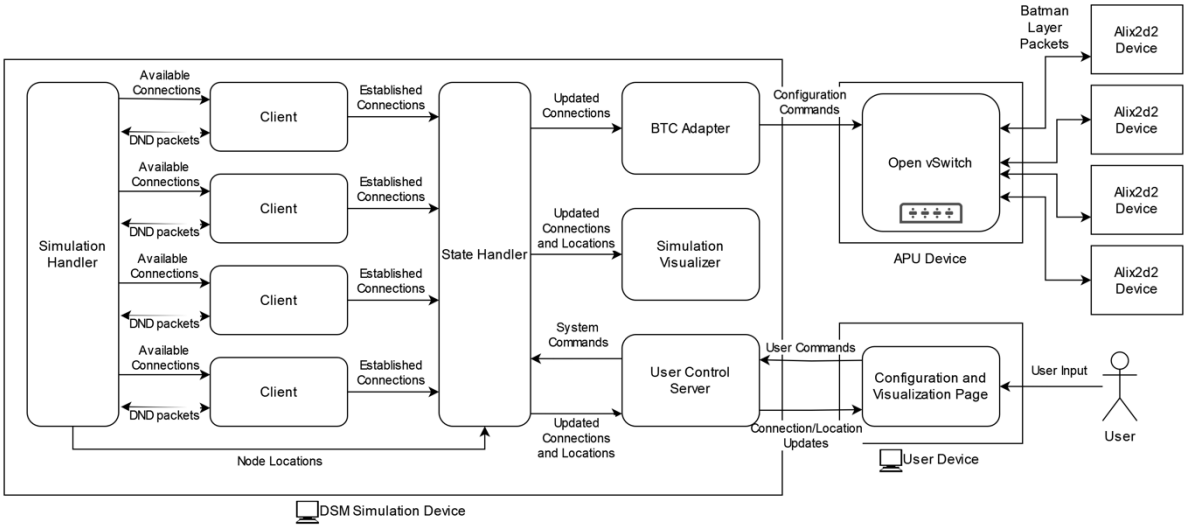

Figure 4 Overall Simulator architecture

We shall describe the main pre-existing components, emphasizing the most important aspects for the realization of our project's main goal. We broke down the simulator into two parts. The first part is the interface, where connections are established between the nodes and drawn, and the second part represents how the components will be made in the backend.

Interface

Simulation Handler

1. Calculates valid radio paths that are sent to the client
2. Calculates the movement of each node, this is interesting to use since the node will need to update its location automatically, without a predefined path

The client

1. Communicates with other clients through the valid paths it receives
2. Sends peer nodes to the State Handler.

State Handler

1. Central repository of the system
2. Sim Handler sends positions and receives connection information from clients
3. Sends data to the front-end and BTC handler

Physical devices

In the figure below (Figure 5), we illustrate how the simulator functions in the physical layer. Because BATMAN-adv uses the MAC layer, the Digital Twin implements MaritimeManet on a set of VMs. From the figure, we can observe how the following components:

Batman topology configurator (BTC) Handler

1. The BTC handler is responsible for configuring the OpenVSwitch based on the information it receives from the State Handler
2. Translates network configurations for the NC

Network controller (NC)

1. Routes the inter-node communication

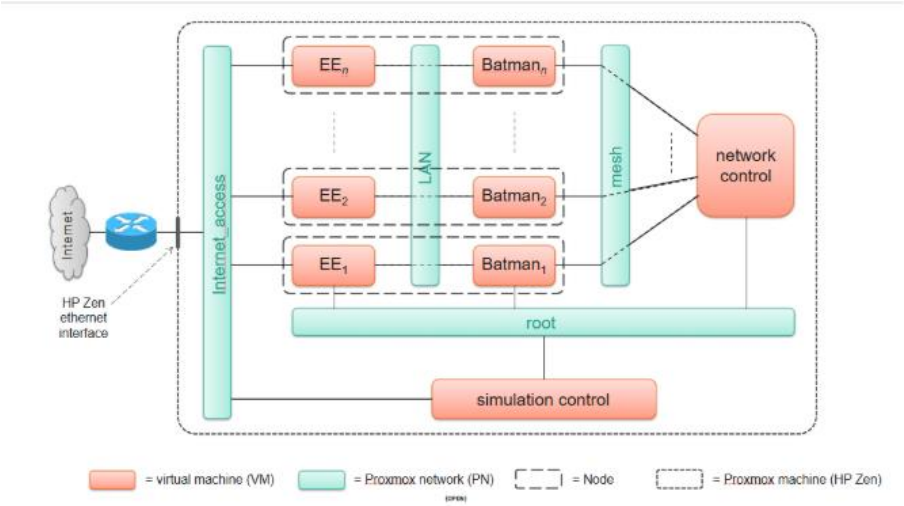

Figure 5 Simulator VM architecture

Both of the components represent the Digital Twin is presented. The interface simulates how drones move on the canvas, while the physical layers ensures that the sockets connect accordingly, and packets are sent from socket-to-socket. The figure below represents a high-level overview of the key class diagrams. In section 5 (System Architecture) we will show what class components we have modified to achieve the project's main goal.

3.2. Simulator Design

In this section, we will illustrate the justification of our design choices on the simulator side. The simulator's main task can be divided into three smaller responsibilities: drone creation, drone deployment, and autonomous drone movement. For each of these stages, we will present several approaches and justify the one that requires minimal changes to the existing framework. Before explaining how we modified the simulator to accommodate drones, we will first present how a ship is created (Fig. 6).

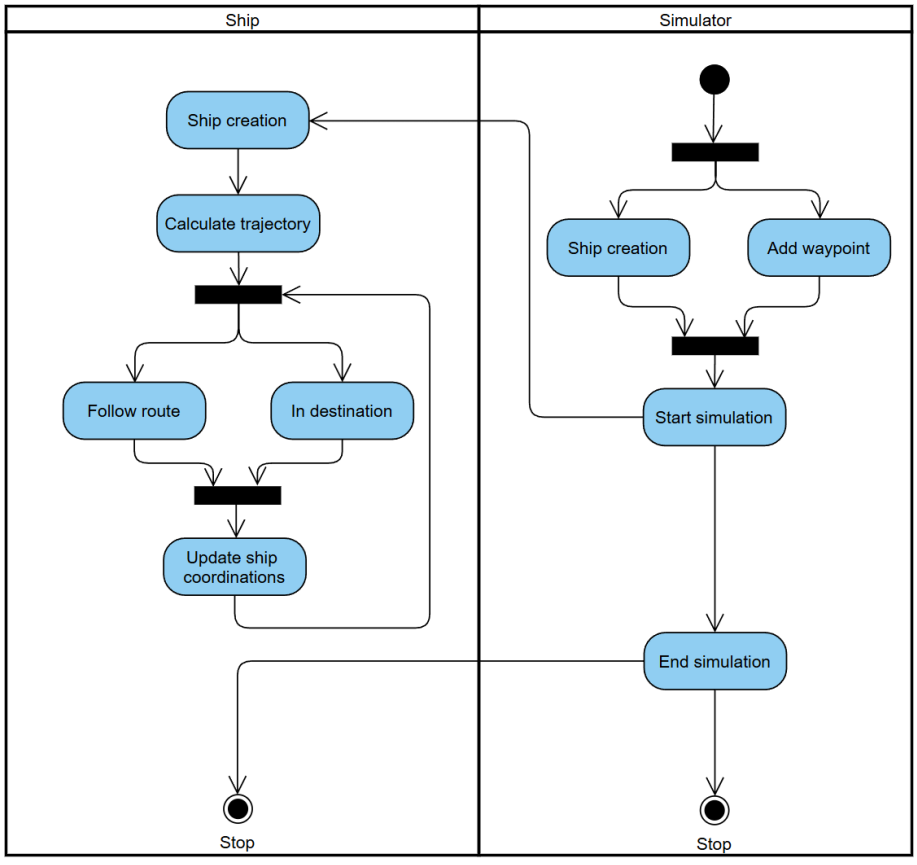

Figure 6 Existing simulator activities

3.2.1. Drone creation

This section presents the design rationale of the drone-creation process. At the beginning of each simulation, every ship is seen as a node in the backend. All the nodes read the configuration from the "Constants" class (Figure above). This component describes the characteristics of each node.

Unified Node

One of the favored solutions being to unify node objects. Both ships and drones read the same configuration, but the drone object has a "is_drone" flag set to "True", as opposed to the "False" value that would be assigned to a ship.

Separate Drone Class

The other possible approach is to separate the drone object into a different class. Since the basic functionality of a drone is the same as a ship (automatically establish connection to reachable peer nodes), and both ship and drone nodes share the same VM initialization, it's not necessary to separate them into different classes.

Drone-ship Aggregation Class

Another procedure worth mentioning is to create an aggregation class that combines the drone and its mothership internally. This approach is out of scope for current system design but has potential for future work, so it will be discussed in the future work section.

Decision

The group decided to adopt the "Unified drone" option, where ships and drones have a similar base, but are differentiated by a flag upon initialization. This approach satisfies the project's requirements while minimizing the change of structure and staying aligned with the design of the previous groups.

3.2.2. Number of drones per ship

After finalizing the drone class design, drone creation was implemented on both frontend and backend. The discussion also focused on determining the appropriate number of drones per ship. In practice, ships should carry different numbers of drones depending on their size and tasks. But in the simulator design phase, considering the time constraint and the scenario complexity, one drone for each ship would satisfy the demand from our client.

3.2.3. Binding Drone IDs to Motherships

In the current system architecture, ships are placed on the canvas according to user-defined configurations. After the simulation starts, the ship virtual machine will be initialized, then the internal ID allocation for each ship node occurs. Therefore, it

would be straightforward to map the drone ID to its mothership ID during the initialization.

3.2.4. Drone Launching Criteria and Autonomy

Drone launching decisions can follow many potential criteria, but the takeoff command should be determined by the mothership. Since our goal is to relay weak connections between ships, the launch condition is designed based on the RSSI strength. By setting up a certain threshold, a drone will be launched by its mothership to prevent potential disconnection between ships.
Because the signal strengths are known, ships are able to acquire sufficient enough information that they can control the movement of the drones. A sufficient and efficient algorithm can be drafted based on this. However, the client addressed that the drone should be autonomous and should have the minimal communication with its mothership, therefore, we decided to design the drone node in such a way that it can change its own velocity, direction and rotation in the "sim_handler" and receive information from the "state_handler", the details of which will be discussed in section 5.2.2.

3.2.5. Communication after launching

Early Phase - Foundation

At the initial stage of the design, we opted to establish a fully functional prototype with the minimum modification of the existing project. Therefore, in the choice of information shared between peer nodes, we decided to only share essential information such as locations of peers. With this approach, we can provide an efficient and clear structure for the upcoming algorithm implementation.

Final Phase - Refinement

Regarding the early-stage design, along with the need for more dedicated drone behavior, we decided to change the information structure shared in the current communication system. Instead of only the locations being shared with direct peers, they now share "peerinfo" objects which contain the location, drone deployment status, deployment timestamp, and responsibilities. This approach can improve the autonomous decision making of the drones, such as in the case where multiple drones are needed to maintain a connection between 2 ships. The details will be discussed in section 5.3.1.

3.2.6. Justification Wrap Up

Regarding the justification on simulator design, through several team meetings and client meetings, we reached an agreement that the existing architecture would be modified as shown in Fig. 7, such that drone creation happens along with the creation of ships. Drones and ships share the same class but distinguish with a flag.

Exactly one drone node will be automatically created when a ship is created at the beginning of the simulation. This will allow us to map the relation between a drone and its mothership. Drones will be activated once certain conditions are met (for example, weakening signal strength). Once deployed (activated) the drone will receive information and will be able to move independently based on algorithmic decisions.

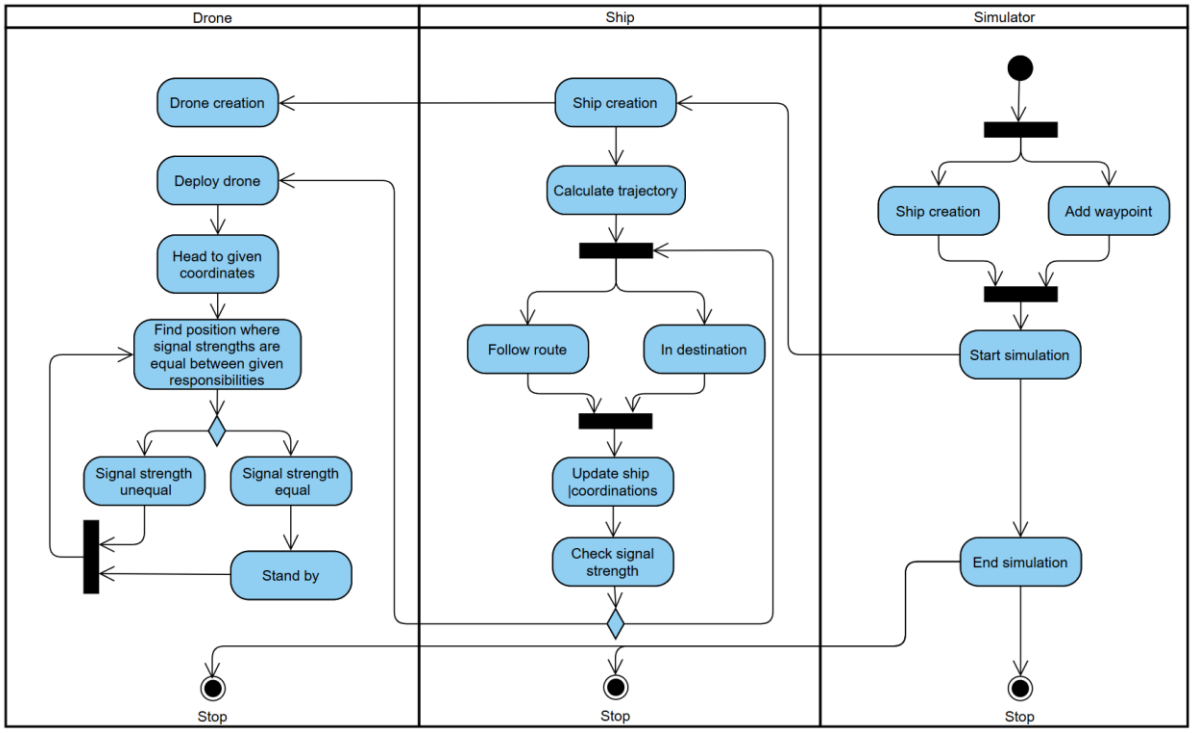

Figure 7 Simulator Design

To achieve this, the following modifications to the simulator must be made:

- The drone determines its direction, velocity, and rotation itself and sends this information to the simulator VM. Meanwhile, the ship follows the user-defined trajectory provided before the simulation.
- The simulator uses a publish-subscribe model to share required information like signal strength, positions, and orientations of nodes that are only available in the simulator control.

3.3. Algorithm Design

Before we get into how we decided to solve this problem, we need to talk about the various challenges we had to consider before being able to proceed with a concrete implementation plan. There were many potential solutions, but many of them we eliminated due to their drawbacks. One of the other factors we were trying to keep

in mind while designing this algorithm was that the solution should also be valid in real life, not uniquely on the simulator.

3.3.1. Considered approaches

We considered a graph approach, where ships are nodes and their connections are edges. We would have had to use GPS information and algorithms on relay point placement to connect fleets (specifically Voronoi diagrams for Steiner point placement), but there were numerous issues with such an approach. Namely:

1. There are different antennas for different ships, and they all have their own range, small, medium, and large. This means that methods using geometric calculations would have to be adjusted to handle this issue.
2. It is difficult to have a complete overview of the network, A geometric solution would often require knowing the position and rotation of each ship.
3. A solution using geometry will always have to use predictions and assumptions, rather than using observed data. We considered using a heatmap for this. A prediction of the signal strength in a particular spot could probably be made, but reality often deviates from simulation. This disconnection between predictions of the signal strength and the real signal strength might cause issues, such as unexpected disconnections, or unsatisfactory connection bandwidth.
4. Ships are moving. Graph problems are usually static, as the nodes don't move, however in our case they do. Even if a proper algorithm to provide a solution for the entire fleet at a given time point was found, the movement of ships might cause the algorithm to diverge into an entirely different solution in the next iteration. Another major issue involves repositioning all drones while maintaining connectivity among ships.

We came up with this idea when we were thinking of larger and more complex situations during the brainstorming period, like providing relaying in entire fleets. In such scenarios, it might be worth reconsidering and could be a solution to Smart Drone Collaboration in Future works. It is worth noting that the client explicitly stated that sharing an overview of the entire network was not allowed, and this was not a suitable choice.

Therefore, we had to approach this implementation from a dynamic perspective as well as keep the client's wish in mind. After understanding the architecture of the Maritime Manet system more, we figured the key may be somewhere in the networking side. This key was found within the SENSE packets, specifically, the

RSSI values. This would solve all of the issues described previously! It would account for the different sizes of the ships, as signal strength is relative to a ship's size. It would account for the movement of the ships, as the signal strength is relative to the ship's position. It also accounts for the client's wish to have a local algorithm on the drone, which shares minimal information between directly neighboring nodes. By using this information, we can create an algorithm that does not need to track previous states and makes decisions on observed information (rather than predictions). Following this discovery, we pitched it to the client, and we agreed to go forward with this for the implementation.

For the drone positioning algorithm, we chose to calculate coordinates based on weights determined by the signal strength of responsibilities. However, there was another possible solution where drones would be using purely the received signal strength indicator (RSSI) to re-position in the optimal spot. It would involve moving and rotating the drone, and observing what happens to the RSSI of the ships it is relaying for. This would require the nodes to keep a history of this RSSI and create predictions of where it should move to next. This approach is less customizable and could result in less precise positioning choices by the drone. We decided to not use this approach for 2 reasons:

1. The drone would have to keep a history of previously observed RSSI, adding another layer of complexity to the drone. Its decision-making would then not purely depend on its current state, but also on its previous state.
2. If the ships that we are relaying for are moving, the RSSI would constantly be changing, even if the drone was stationary. It might be difficult to determine whether movement or rotation of the drone caused a change in signal strength, and what change it caused. Essentially there would be a huge amount of noise that would somehow have to be filtered out.

The second point here is a major issue. It might be theoretically feasible to use curve-fitting and machine learning algorithms that manage to deal with the moving ships. Such an algorithm might even try to extrapolate the locations and movements of ships to be able to filter out the impacts these movements have on the RSSI, such that it can determine how the drones' own movements affected the signal strengths.

3.3.2. Responsibilities

A key feature of the drone system is the concept of "responsibilities". To put it simply, a drone's responsibility is a list of nodes which it should relay for. If a drone

were to strengthen the link in a scenario where there are two ships (referred to as "A" and "B"), its responsibilities would be "A" and "B".

3.3.3. Information sharing

Nodes share a reduced version of their state with their directly connected peers, as described in Section 5.3.1 (Info_storage.py). For ships this includes their current direct peers and the state of their drone, and for drones their current state and their responsibilities. This helps ships make smarter decisions about launching drones, and it helps drones to make smarter decisions about taking on new responsibilities or dropping existing ones.

3.3.4. Drone Algorithm

Because the drone has different behaviors needed at different points in time, we created three states which all entail their own behavior.
The drone can be in three states: initial deployment, optimizing, and returning to base. We show these in Fig. 8.

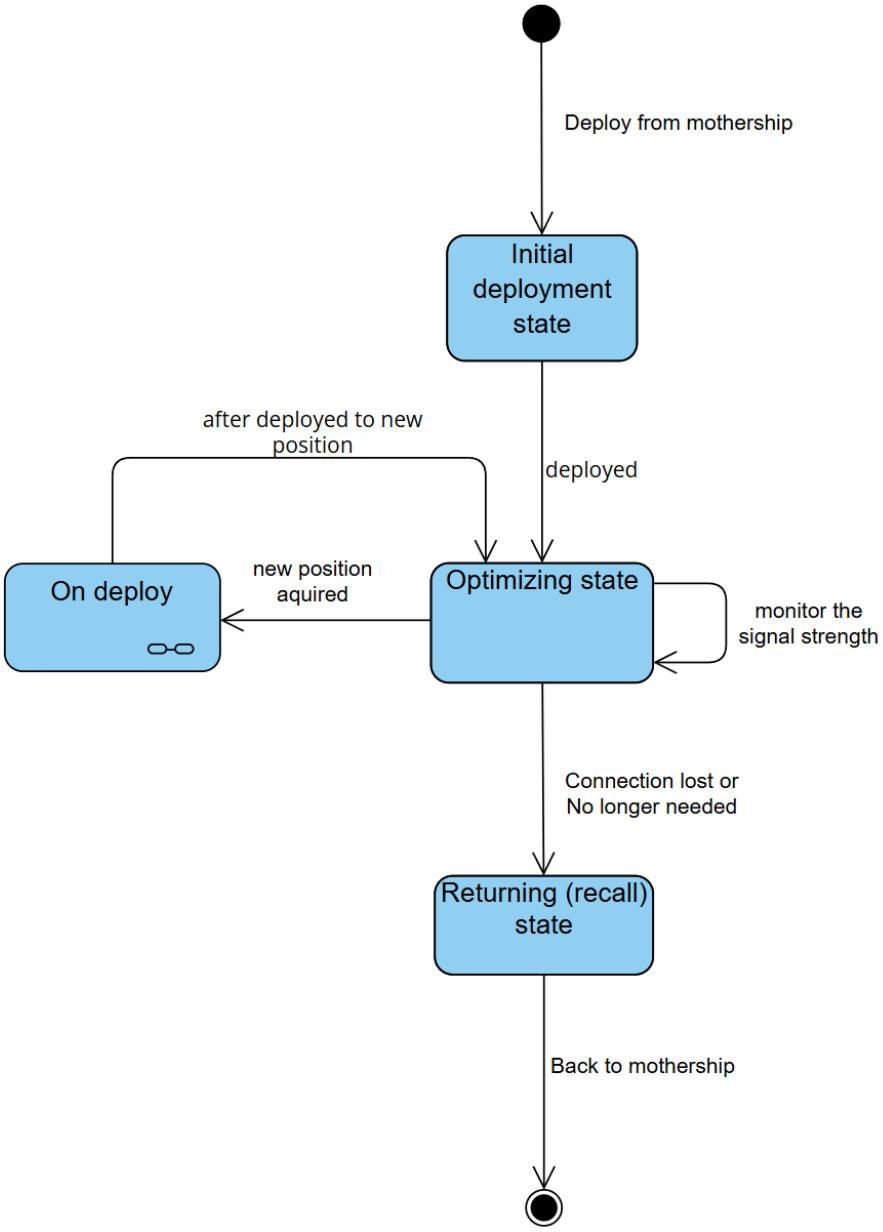

Figure 8 Drone States Diagram

Initial deployment

Once deployed, a drone moves to the coordinate given to it by its mothership. This coordinate is the point where signal strength between two ships is equal. Once it arrives at this coordinate, or is connected to all its responsibilities, the state becomes a "search optimal position" state.

During the initial deployment, the drone will not accept new responsibilities or drop existing ones.

Optimizing

In the optimizing state, the drone will attempt to find the best possible position and rotation to find a spot where the signal strengths of all ships it is relaying for are equal and maximal.

Currently, to find the optimal position, it takes the weighted average of the positions of all ships it is relaying for. Ships with a weaker RSSI will be given a more significant weight in this algorithm. By doing this, a drone position will converge to a position where signal strengths are at least equal, and most likely to be very near to the maximal possible signal strength.

To find the best possible alignment for the drone, the algorithm calculates all the angles to its responsibilities. To put it more clearly, we take the location of the drone, and the location of a node we are responsible for. The algorithm then calculates the angle of the vector from the location of the drone to the location of this node. The result is of course a list of angles.

The algorithm then simulates 100 linearly spaced possible rotations from 0 to 2pi/6, This can be seen more clearly in figure 9 below. The 100 here is some arbitrarily picked value; it could be changed to 10 or 1000, for example, to change the number of rotations we search through. The 6 constant represents that we are dealing 1/6th of a full circle. If a drone were to have a different number of antennas, this constant factor would change. For this reason, 6 is labelled as “ANTENNA_COUNT” in our code.

When simulating all these antennas, we calculate the “error” between each responsibility’s angle and the antenna which is pointing best at the ship. When the algorithm finds an angle such that the sum of all these errors is minimal, this is set to be the target angle for the drone to rotate to.

During the optimizing state, the drone will selectively decide whether to add newly connected nodes as responsibilities, and whether to drop existing responsibilities.

A drone will not add another drone to its responsibility list if its mothership is already connected to the other drone's mothership. This will avoid edge cases where different drones are sent out to strengthen different connections and end up neglecting their initial responsibilities for a connection that doesn't need to be strengthened.

Nodes that disconnect during the optimizing state are always dropped as responsibility, except for the mothership. When a node disconnects, the drone is

already minimizing the signal strength for its other responsibilities to maximize the connection for the disconnecting node.

If the disconnecting node is a ship, this allows the drone to easily swap the responsibility for the disconnecting node for its potentially launched drone, as soon as it connects.

Nodes that have a very strong connection to the drone are also dropped or not accepted. Since the drone stays always connected to the mothership, nodes with a very strong connection to the drone will likely have a decent connection with the mothership already. If that is not the case, and they are by coincidence close to the drone, they have low priority as they are already being relayed for with a strong connection. The drone can better focus on its other responsibilities. Once they move further away, they will automatically be added as a responsibility again.

Returning

Once all a drone's responsibilities have strong connections, they are dropped. This will effectively trigger a "recall" of the drone. If on the way back to the mothership it turns out one connection is getting too weak, it will automatically become a responsibility again, and the drone will be Optimizing.

If there is only one responsibility left, this will by default be the mothership, as this responsibility is never dropped while the drone is deployed. Since optimizing the connection for only one node by design means moving as close as possible, this will trigger the drone to automatically move to the mothership.

While the drone is active, its responsibilities will always contain its mothership in the first position of the array. If it loses connection to its mothership, it will move towards the last known position of the mothership to restore the connection.

Once the drone gets close to the mothership, it will transfer to the returning state, to avoid triggering ships close to the mothership to launch a drone, and to avoid adding ships or drones close to the mothership as responsibility. If the drone is practically on the mothership, the mothership will tell the simulator to deactivate the drone node in the network, set it as not deployed, and remove itself from the responsibilities list of the drone.

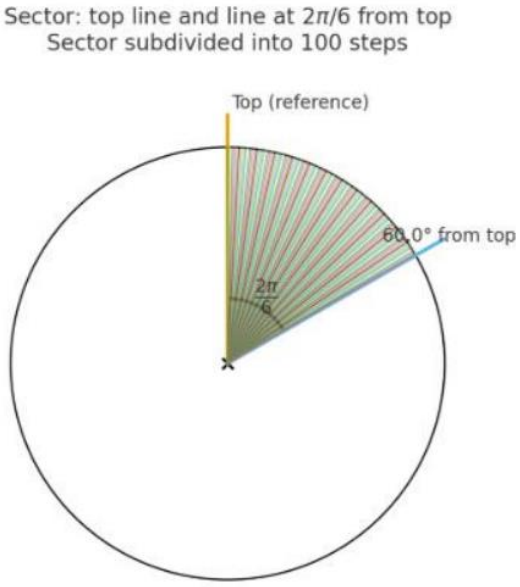

Figure 9 Illustration of optimal rotation search

Overall, this approach is a hybrid combination between using location data and information on signal strength. We have chosen this approach because it is robust, scalable, and simple to implement. We also believe that for further work on the drone algorithm, a hybrid approach is the simplest way to achieve satisfactory results

The implementation of the drone algorithm can be found in algorithm.py (drone_algorithm). The code is elaborated upon in the Components section.

3.3.5. Ship Algorithm

On the ship side of the algorithm, in order for ships to detect a weakening connection, they must continuously monitor the signal strength. Ships also keep track of previous signal strengths, to determine if a connection is weakening or improving. The goal is that the signal strength is equal between all detected responsibilities. The values that we found for the RSSI are: above –80 dBm for an excellent connection (allows transmission of around 1 Mb/s) and –90 dBm for no connection at all. Since it takes a little time for the drone to arrive at its destination, we want to avoid disconnection between the moment the drone is deployed and the moment the drone is en route, so we chose –87 for our deploy threshold. Once the bandwidth between a ship and another node reaches this threshold, the ship will enter the phase of code where a drone should be deployed with this ship as its responsibility.
To prevent two ships from sending two drones at the same time, there must be a universal protocol established. At first, we considered a formal handshake protocol (now part of the Future Improvements section), but it has been replaced by a simpler

version where the ship with the lower ID deploys its drone. In order to determine this, a ship's ID is compared with all of its peer's IDs. In this case, it will be the first to send out a drone.

We also handle the case where the other ships have larger IDs. Let's consider a case where a ship has a weakening connection with another node. If the ship detects the other node has a lower ID (therefore, it's also a ship), which already has deployed its drone more than three seconds ago, it will also deploy its drone. The three-second delay is in place to allow the drone from the other ship that is potentially deployed to strengthen this very connection, to first connect with us, before we can determine that another drone is needed. Once a ship determines it can deploy, the launch command includes the direction the drone must initially move in, and this is called "deployment_target" on the drone. This acts as deploying the drone in a certain direction. This process is shown in Fig. 10.

If the other node is a drone, it will check if the drone currently accepts responsibilities (not in the initial deployment state or in the returning state) and check if it isn't connected to the mothership of the drone already. This is done to avoid launching drones to strengthen a connection with a drone launched to or returning from a close by peer.

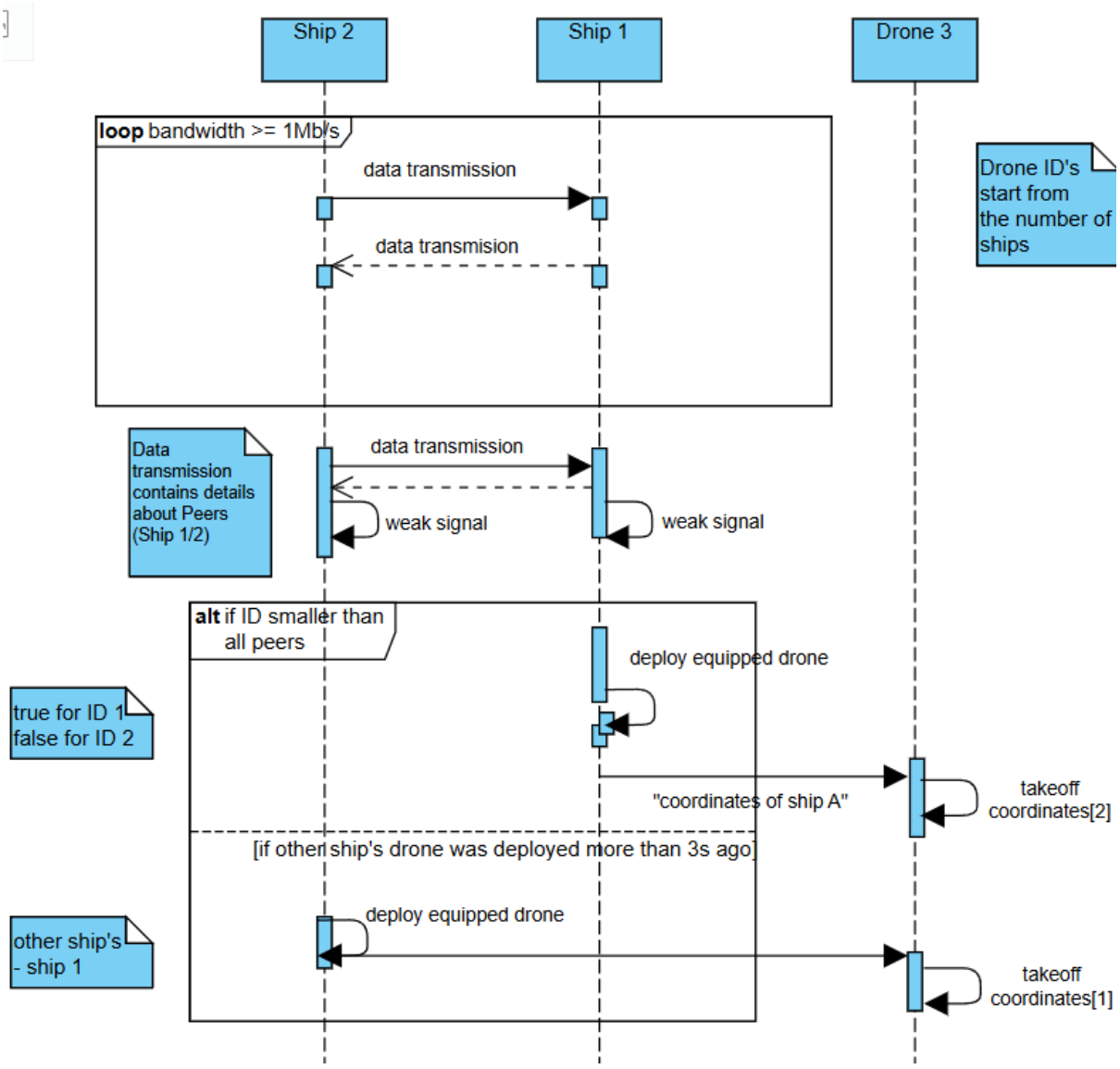

Figure 10 Deployment protocol sequence diagram

The details about the implementation of the ship algorithm can be found in algorithm.py (ship_algorithm). The code is elaborated upon in the Components section.

4. Risk and Challenges

4.1. Simulator dependency

Implementing and testing the distributed algorithm heavily depends on the correct functioning of autonomous drone deployment and positioning in the simulator. If implementing these modifications takes too long, the time we can spend on implementing and testing the algorithm that runs on the drones will be significantly reduced.

Mitigation: Working in two groups, where one will work on the simulator and the other will work on the design and preliminary testing of the algorithm, allows both simulator and algorithm development to progress in parallel. While the simulator team implemented the required changes, the algorithm team designed and refined the algorithm.

4.2. Implementation trajectory

After describing our approach based on the current simulator design, we present an action plan using the sprint methodology. We assigned story points to every task based on its complexity, using a scale from 0 to 15. The total workload amount equals 141 story points. This work has been planned across 10 sprints.

Week	Task	Story points	Total
1	Team formation and communication with supervisor and client	1	1
2	Setup the Proxmox machine remotely	3	10
	Begin to understand simulator's architecture	7	
3	Continue reading simulator's architecture	6	12
	Setup Git workflow, project proposal, team structure other platforms (ex: Trello)	3	
	Split into 2 teams: algorithm and simulator to mitigate potential risks	3	
4	*Simulator*: Implement drone creation (unified node) in the backend	9	14
	Algorithm: Analyze potential edge cases and design the simulator	5	
5	*Simulator*: Implement drone deployment in the backend and display in the front-end	7	15
	Algorithm: Start creating a sketch on algorithm implementation in the backend	8	

6	*Simulator*: Implement autonomous drone movement in the EE VM.	12	27
	Algorithm: Start implementing the algorithm independent of any component.	15	
7	*Algorithm:* Implement initial deploy and recall algorithm	7	33
	Algorithm: Implement weighted positioning algorithm	9	
	Algorithm: Implement rotation alignment algorithm	10	
	Start working on the final progress report	7	
8	Continue to work on the deliverables	7	20
	Optimize algorithm-simulator integration	6	
	Testing	7	
9	Work on the poster and client presentation	5	5
10	Final touches to the software and present deliverables	4	4

We compare this workload to an ideal linear burndown (Fig. 11). In the ideal scenario, the team should complete 15.67 story points/week. In the graph below, we illustrate an overlap between the ideal and proposed burndown. The next section

provides the design rationale and implementation of our planned approaches.

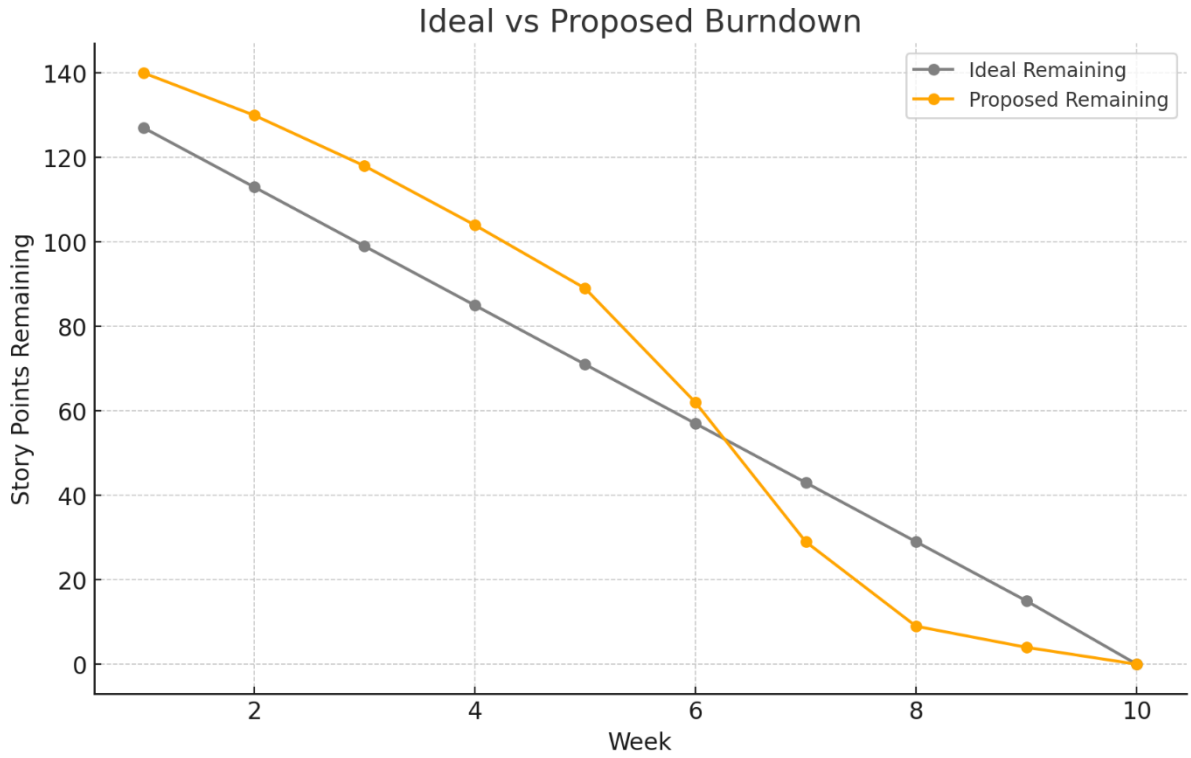

Figure 11 Ideal and proposed burndown

5. System Architecture

5.1. Introduction

This section describes the implementation of the approach discussed in chapter 3 and presents how the simulator was modified to achieve the project's main goal. Fig. 12 below shows the class diagram of the simulator's functionality prior to any changes.

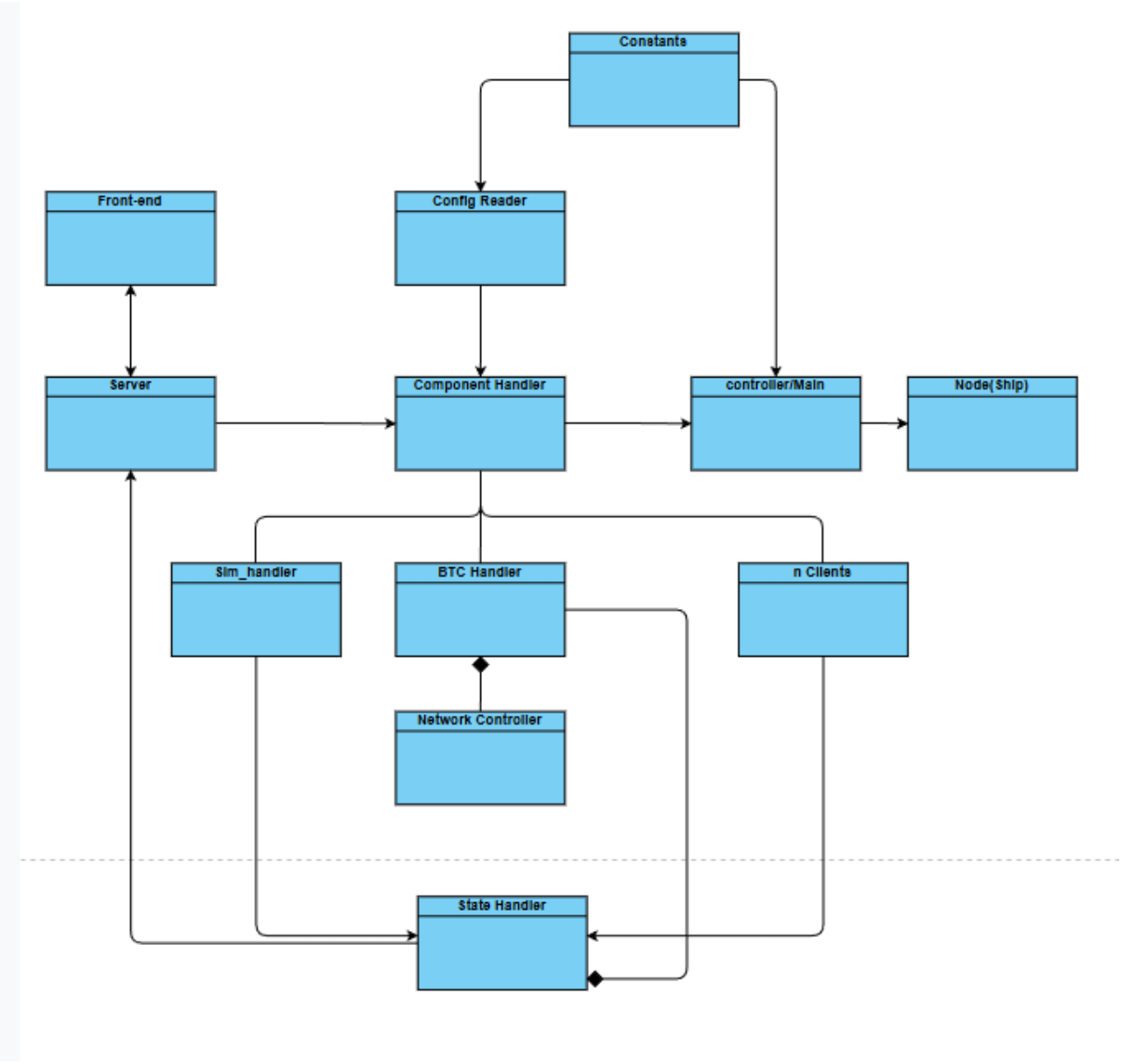

Figure 12 Current simulator's class diagram

5.2. Components – Simulator

5.2.1. Drone creation

Once the simulation is started, the manually added front-end ship nodes are saved to "config-nodes.json". This file is then sent to the "Server" component as illustrated in Fig. 13.

The different components of the simulator will read this file using the "config_reader". The config reader will add a drone node for every ship in the config-nodes.json.
Because the different components (such as the "btc_handler") of the simulator expect

a sequential list of nodes, locations, directions, velocities, etc., the drone nodes are added to the nodes list after the ships. The same goes for their entries in the locations, directions, velocities lists, etc. A simulation with *n* ships will therefore result in a list of 2n nodes, a list of 2n locations, a list of 2n directions, a list of 2n velocities, etc.

In this setup, one ship has exactly one drone, and since the drone nodes are created sequentially after the ship nodes, every ship and its drone are exactly n/2 indices apart in every list. If a ship has index 0, its drone has index 0 + n/2, etc.

The nodes list that the config reader produces previously only consisted of a two-tuple, the ID of the node and the number of sectors of the node. This has been changed to a NodeConfig instance for every node. This instance contains the following attributes:

id: *int*
sectors: *int*
is_drone: *bool (defines the movement behavior of the node)*
active: *bool (defines the status of the node in the MaritimeManet networking)*

The EE-VM is not put into use for current system architecture, but since it barely affects the system's performance, we decided to retain its initialization for future development. In the diagram below, we present the changes made to the original architecture to fulfill the drone creation task.

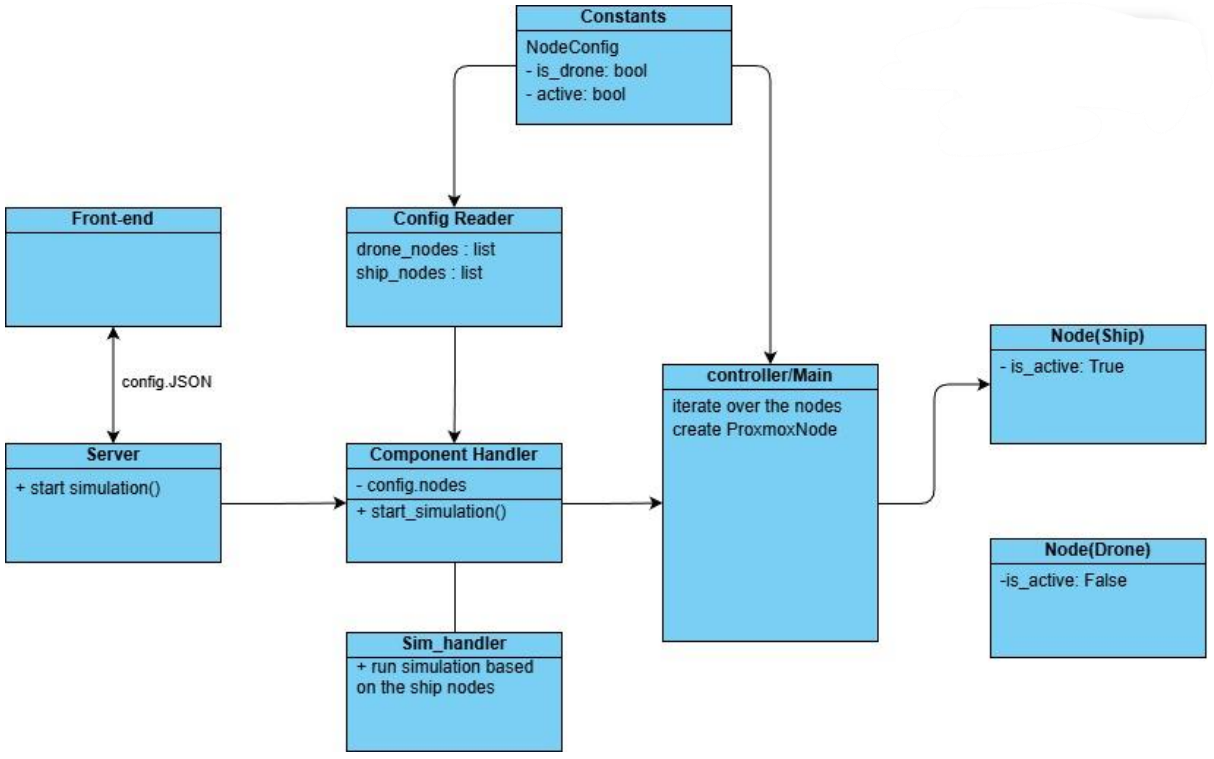

Figure 13 Simulator architecture

5.2.2. Drone deployment and movement

The “sim_handler” stores movement information like velocities and directions of each node in a one-dimensional list. On every simulation iteration, it calculates the new velocity and direction vector for every node based on the next waypoint set for the node. After that, it calculates the new position of every node based on the old position and the updated velocity and direction vectors. If there are no waypoints left for a node, it will set its direction vector and the velocity to zero.

Drone nodes are excluded from trajectory calculations and can manipulate their own position in the simulation through the physical device server. This means that drones can move independently. Unlike ships, where the “sim_handler” calculates the velocity, direction and rotation of nodes for them, the sim_handler instead uses the velocity and direction vectors as well as rotation speed given by the drone nodes to determine the next location of the drone nodes. These values are calculated in the algorithm logic. The nodes communicate these values to the “sim_handler” through Python sockets. This effectively allows drone nodes to move autonomously.

If a node has the ‘is_drone’ attribute set to true, and the ‘active’ attribute set to false, it is a drone that is not deployed and therefore remains on board of the mothership, until the active attribute is changed.

Since drone nodes are created sequentially and added to the nodes list after the ships (see section Drone Creation), and the “sim_handler” loops through the nodes in a sequential order when calculating their new positions in a simulation iteration, the positions of the ships will always be calculated before the drones.
When the “sim_handler” reaches a drone node that is not active, it simply sets its location to the location of the ship that has already been calculated.

5.3. Components – Algorithm

5.3.1. Info_storage.py

This module defines the data structures used to compute autonomous drone deployment and positioning.

Each node has a “NodeInfo” instance containing information about itself, while information about connected peers is stored in a list of “PeerInfo” instances.

After running its algorithm, a node shares a reduced version of its state as a “PeerInfo” instance to all its connected peers.

NodeInfo	
Name	*Value*
node_type	SHIP / DRONE
location	Coordinates (x, y, orientation)
peer_signal_strength	Dict of peer ID -> (sector, signal strength)
peer_information	Dict of peer ID -> PeerInfo

Table 6 NodeInfo

NodeInfo is extended into ShipNodeInfo and DroneNodeInfo to store ship-specific and drone-specific information. These NodeInfo classes contain the data necessary for the ships and drones to calculate their next move through their specific algorithms. They are split into different classes because they require different data.

A custom Coordinates class that stores coordinates in a 3-tuple of the X-coordinate, Y-coordinate and rotation coefficient.

ShipInfo (extends NodeInfo):	
Name	*Value*
drone_id	Node ID of assigned drone
drone_deployed	Boolean
deployment_timestamp	Timestamp of deployment
previous_signal_strengths	Dict of peer ID -> (sector, signal strength)

Table 7 ShipInfo

DroneInfo (extends NodeInfo):	
Name	*Value*
mothership_id	Node ID of mothership
responsibilities	List of node IDs the drone is responsible for
deployment_target	Coordinates
last_known_mothership_location	Coordinates
current_state	INITIAL_DEPLOYMENT / OPTIMIZING / RETURNING

Table 8 DroneInfo

PeerInfo

Name	*Value*
node_type	SHIP / DRONE
location	Coordinates (x, y, orientation)

Table 9 PeerInfo

ShipPeerInfo (extends PeerInfo):	
Name	*Value*
drone_deployed	Boolean
deployment_timestamp	Timestamp of deployment
connected_nodes	List of peer node IDs

Table 10 ShipPeerInfo

DronePeerInfo (extends PeerInfo)	
Name	*Value*
responsibilities	List of node IDs the drone is responsible for
current_state	INITIAL_DEPLOYMENT / OPTIMIZING / RETURNING

Table 11 DronePeerInfo

5.3.2. Algorithm.py

Contains the autonomous behavior algorithms for ships and drones. Both algorithms are executed on each iteration, and they are used to determine the next actions of the node based on the type of node, current state and peer information. Ships and drones have different algorithms. Algorithms are stateless in terms of networking, they output messages, which are then routed by the node's connector.

“ship_algorithm”: runs on each iteration for a ship node, monitoring peer signal strengths, deploying or recalling drones as necessary, and broadcasting updated peer information to the network. When a weakening connection is detected, the respective node IDs are stored as responsibilities. If the deployment condition is fulfilled, the equipped drone is deployed with these responsibilities, as well as an initial target position. This ensures the drone is launched in the correct direction.

“drone_algorithm”: runs on each iteration for a drone node, adjusting its movement based on current responsibilities, mothership location, and signal strengths, and

broadcasts updated peer information.
The mothership location is always stored as “last known mothership position”, since we want the drone to always know where its mothership is approximately, even if the connection was lost. This is because the drone is aware only of the locations of connected peers.

5.3.3. Physical_device_simulator.py

Provides a simulated environment for nodes to communicate with each other and the simulator.

PhysicalDeviceServer: Sends each node its own location and connections. Relays movement commands, drone deployment, and recall instructions to the simulation handler.

Each node manages its own connections and communication with other nodes; the server only interacts with nodes individually.

5.3.4. Server.py

Implements a generic TCP server.

5.3.5. Client.py

Implements a generic client that connects to a server. Represents the nodes.

5.3.6. Connector.py

The core of each node in the physical device simulator, managing all communication and coordination. It handles the connection to the simulator via the PhysicalDeviceConnector, runs the node’s own PeerToPeerServer, manages outgoing PeerToPeerClient connections to other nodes, and integrates the algorithm logic, ensuring movement commands, drone deployment, and responsibilities are processed and relayed correctly.

PhysicalDeviceConnector: extends Client and is used by a node to communicate with the physical device simulator.

PeerToPeerServer: extends Server and is used by a node to accept peer-to-peer connections from other nodes.

PeerToPeerClient: extends Client and connects a node to a peer server in the peer-to-peer network.

PeerToPeerService: manages the peer-to-peer network for a node. Each node runs one PeerToPeerServer representing itself, and has PeerToPeerClients for every node it's connected to.

Algorithm: manages a node's information, peer-to-peer connections and algorithm logic. Processes incoming messages and runs either the ship or drone algorithm. Relays messages to the simulator via the physicalDeviceConnector and relays messages from and to peers via the PeerToPeerService.

5.4. Inter-component communication

Drone creation

Component	Accessed component	Comment
Front-end	Server	The front-end sends the "config-ships.json" file to the server component when the simulation starts.
Config_reader	Constants	Config_reader iterates over the ships stored in the JSON file and configures them based on the NodeConfig class in the Constants component. It first creates the ship configuration, then creates the drones and sets the flag "is_drone" to true.
Component_handler	Config_reader	Component_handler accesses the new configuration created that includes ships and drones.
Component_handler	/controller/Main.py	Component_handler calls the "/start" API endpoint in the main file and passes the configured nodes.
Main.py	Constants	Main.py fetches the ProxmoxNodeConfig class from the Constants class. This configuration is responsible for the VM node creation in the physical layer.
Main.py	NodeManager	The main file creates the VMs based on ProxmoxNodeConfig. The nodes with the "is_drone" and "active" set to false will be ignored in the simulation unless their status is changed.

Table 12 Drone Creation

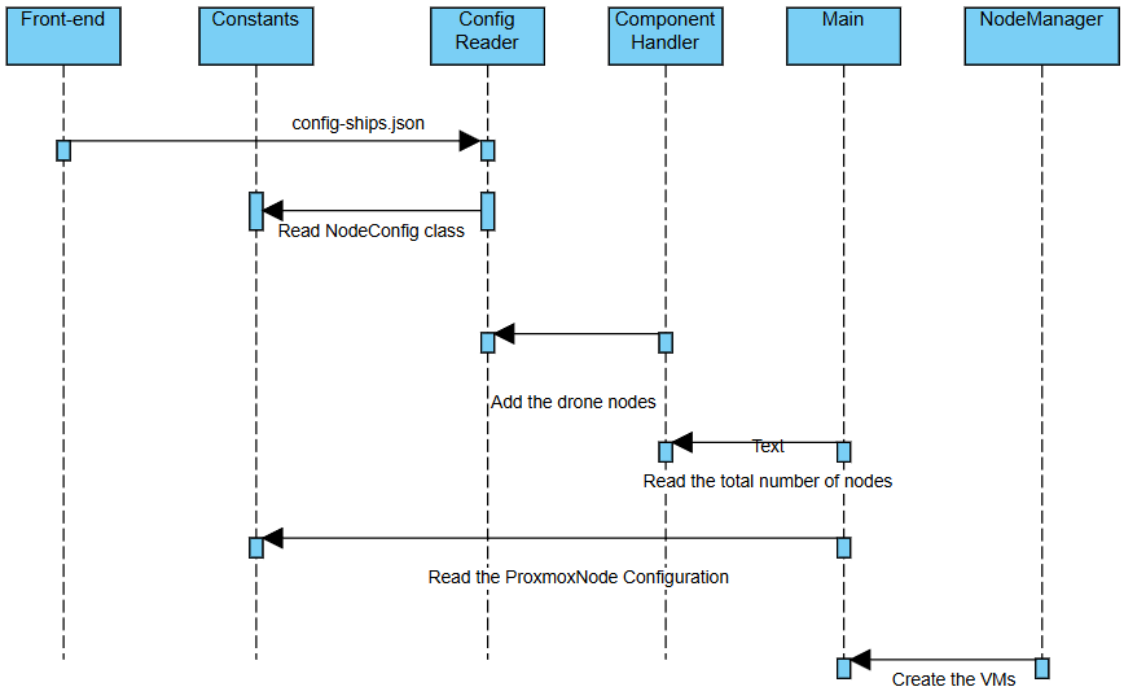

Figure 14 Inter-component communication

Drone deployment and routing

Physical device simulator	Sim Handler	The sim handler periodically polls the physical device simulator to check if any drones should be deployed. It also shares locations of nodes and checks for any movement updates of autonomous vehicles.
Sim_handler	State_handler	When the function “activate_node” is called, it sends the node_status to the State Handler.
State_handler	Front-end	The State Handler forwards a list of node_status entries to the front end. As long as a drone is not active, it will not be displayed.
Connector	Physical device simulator Connector Algorithm	The connector connects to the physical device simulator and to other nodes/connectors through a peer-to-peer-style connection. It bundles all information into an information object.

Algorithm	Back-end	The algorithm gets polled to execute its logic on each iteration; this is done on a fixed timestep. It will determine the best course of action based on an information object provided through it by the Connector. It can decide whether to deploy a drone or move in a direction based on the information provided.

Table 13 Drone Deployment and Routing

6. Result

A system with the previously specified design was successfully implemented (Fig. 15) and delivered to the client. The client was pleased with the results, as we were successfully able to fulfill almost all the proposed system requirements. This outcome was unexpected given the challenges described later in the Evaluation and the limited development time. The deliverables included the modified simulator with the new algorithm implementation atop it, as well as an updated user manual for the simulator, at the request of the client.

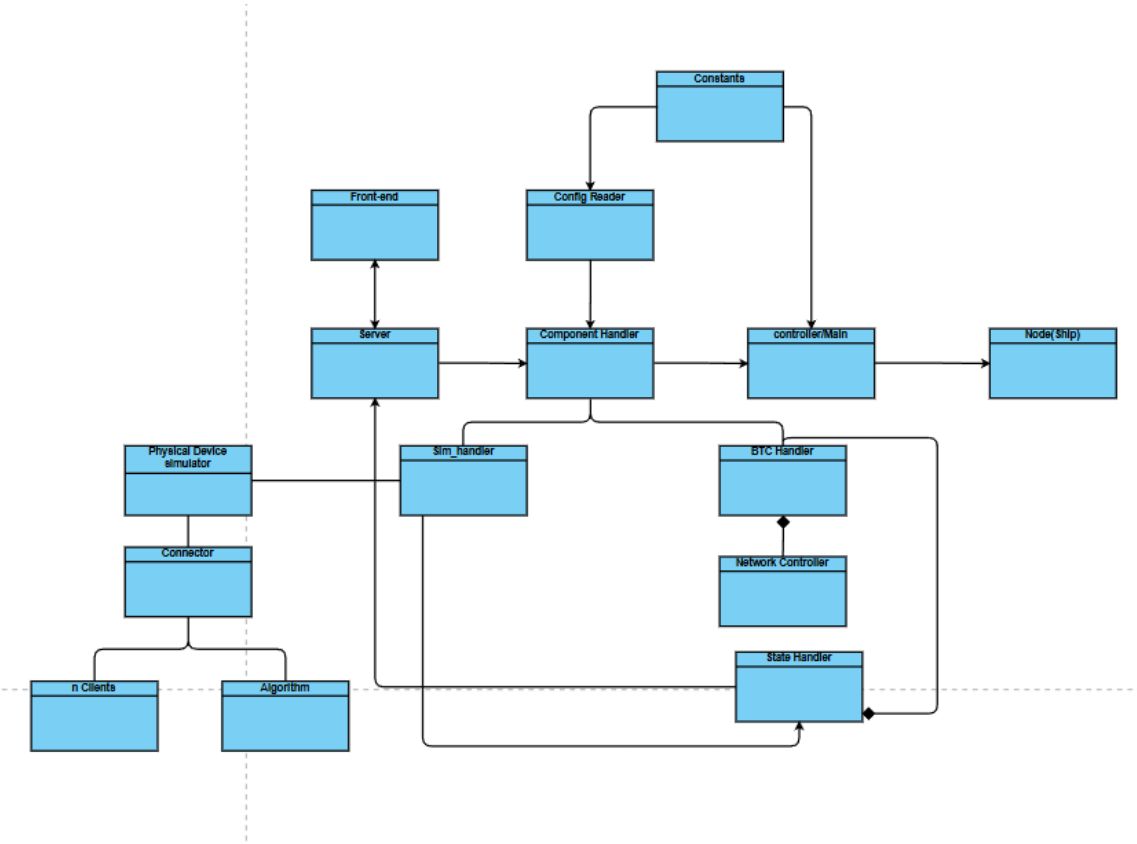

Figure 15 Final Class diagram of the simulator

7. Testing

After implementing the simulator changes and the algorithm, their functionality must be ensured to satisfy the client. This chapter will introduce the testing approach used so that the system meets all the requirements previously discussed.

7.1. Testing strategy

The current project runs across multiple VMs to simulate the connections between ships and drones. The nature of the distributed setup involves network communication and message passing, while the simulation environment is dynamic and real-time, making it challenging to isolate and test with predictable results such as unit tests. Instead, we decided to apply manual testing which allows us to observe the system as a whole, and how the components interact together.

7.2. Testing methodology

To ensure our changes led to the desired output within the preexisting framework, we decided to use regression testing. Due to the nature of the system, these tests for overall system functionality were conducted manually. The system is designed to act closely to how it would be in real life and is involved with physical device simulators. Nodes run on real-time VMs and simulate network conditions as they

would in real life. Many of these factors make automated testing difficult for our project. Debugging would also occur during these tests. One of the difficulties faced when debugging was the slow response time of the Proxmox machine. The following process was run for each test:

1. Write code to be tested on our own machine
2. Push code to GitLab remote
3. Fetch/pull code to the proxmox virtual machine
4. Run the updated code on the proxmox virtual machine
5. Observe any changes on the canvas - if behavior is not correct, fix code piece, check print statements and repeat 1-5 (debug)
6. If system behavior is compliant, finish

Some examples of specific tests include:

- Testing connections with no ship movement
    - During this test we found a bug, which was that ships would not make connections when the rotation was 0. This was subsequently fixed.
- Testing simple drone objects existing on the canvas
    - The ships are constrained by the pre-determined trajectories created by the user, but drones move independently. We initially created the drones as external objects from the ships and tested them with simple trajectories. Then we tested that they could move outside of a pre-determined trajectory by making them move in a circular pattern. Once we had this independence from the trajectory, we were able to implement the autonomous algorithm.
- Testing drones making visual connections with ships
- Testing communication between drones and the simulation handler
- Testing drones moving autonomously
    - For this, we first gave the drones a simple algorithm, specifically moving them in a circular pattern. This again was a manual test as this can only be determined visually on the simulator's canvas. This was an important test, because if we could successfully implement a simple trajectory, we could later adapt it into the final autonomous version of the algorithm.
- Testing automatic drone deployment

    - For this, we were testing specifically on deploy threshold values, such that the drone doesn't deploy too early or too late. For the purpose of debugging the Nodes (Drones and Ships) we created terminals displaying the node data. We were able to use print statements from within the nodes for more efficient debugging.

- Testing automatic drone repositioning
    - For this, we stress tested the system by setting up multiple pairs of ships, horizontally, vertically, and miscellaneous patterns. The purpose of these tests was to determine if the repositioning algorithm always maintains the connection between two ships, regardless of movement.

Debugging

As we needed information to debug what was happening internally on the drones while they were deployed, we enabled terminals which print the statuses of each node as shown in Fig. 16. This greatly helped us identify and fix bugs. An example of a crucial bug fixed with this was the delay problem. We noticed that the printed status of the drone was not in line with what was happening on the canvas. This helped us make some correct determinations about the drone algorithm as well, specifically checking if the values were correct for a given drone behavior.
These terminals were disabled before handing the code over to the client. They can be re-enabled in "DSM/component_handler.py" for future development as shown in Fig. 17.

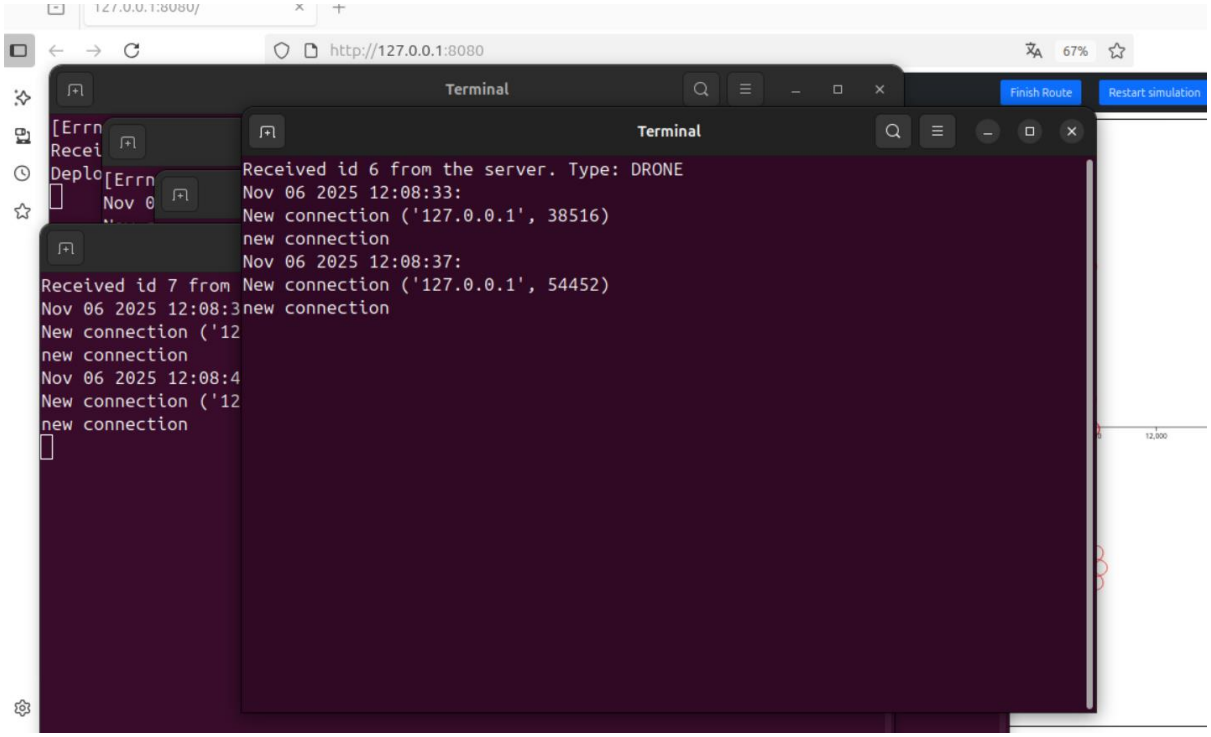

Figure 16 Debugging terminals

DSM/component_handler.py

```
...  ...  @@ -52,11 +52,7 @@ def start_components():
52   52       # Start all clients
53   53       for _ in range(n):
54   54           started_subprocesses.append(subprocess.Popen([sys.executable, "client.py", "&"]))
55       -       started_subprocesses.append(subprocess.Popen([
56       -           "gnome-terminal",
57       -           "--",
58       -           "bash", "-c", f"{sys.executable} relay_algorithm/connector.py; exec bash"
59       -       ]))
        55  +       started_subprocesses.append(subprocess.Popen([sys.executable, "relay_algorithm/connector.py", "&"]))
```

Figure 17 Terminal disable

7.3. Complexity of edge cases

In the design stage of this project, we brainstormed many possible situations before finalizing the implementation plan. We encountered various edge cases, such as cases where drones relay well for a group of ships, but it could perhaps be done in a more intelligent way. For example, Fig. 18 shows how our system currently handles three medium ships moving away from each other. To handle these edge cases, additional functionality is required for drones to coordinate in a smarter way, as seen in Fig. 19. These situations usually occur in cases with three or more ships. They require more functionality and are outside the scope of our project. We outline this as a possible future improvement in the Future Work section, specifically the “Smart Drone Collaboration”.

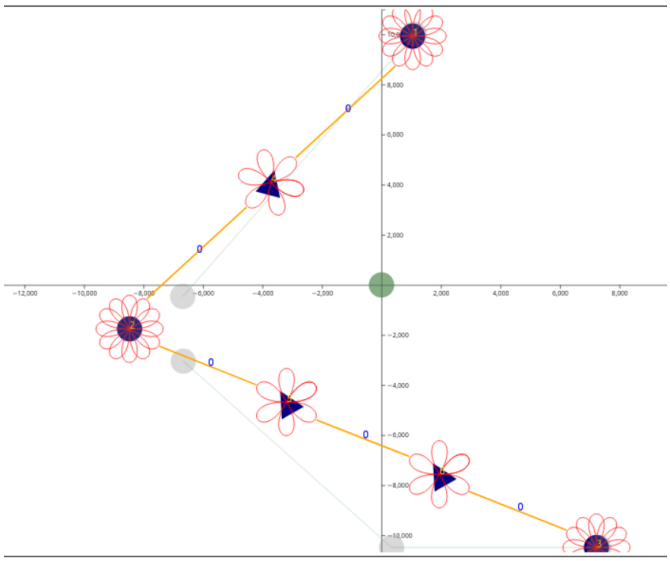

Figure 18 Complex case - currently

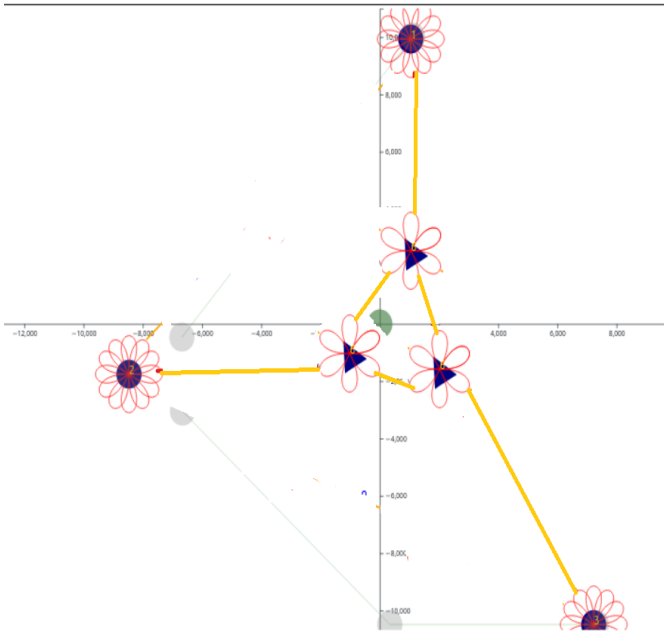

Figure 19 Complex case - better

8. Future work

8.1. Dynamic Drones per Ship

Allow each ship to have a user-defined number of drones, configurable in the front-end, while keeping node lists (directions, velocities, locations, etc.) sequential to maintain compatibility with the simulator's current architecture.

8.2. Front-End

- Add a toggle allowing the user to specify whether a ship can carry a drone.
- Make drones editable in the node configuration UI (e.g., speed, sectors, rotation, etc.).

8.3. Initialization / Config Reader

The current system architecture allows each ship to carry only one drone, regardless of the size of the ship. It can smoothly blend in with the current design, but it could be beneficial to let the user decide how many drones a ship could carry.

8.4. Ship–Drone Ownership

8.4.1. Option A — Embed IDs in NodeConfig

Instead of storing the drone and its mothership id relation in the NodeConfig. Mapping the drone ID into its mothership's ShipNodeConfig and mapping the mothership ID of such a drone in the DroneNodeConfig rather than having a single NodeConfig for all the nodes.

8.4.2. Option B — Fleet Structures

In current system architecture, the drone is created along with a ship, it would be more efficient to create a fleet (a mothership with at least one drone). With the new fleet class, the "state_handler" can broadcast the fleet tuple contained with both ship id and drone id as a dictionary. By using such a structure, the communication could be improved.

8.5. Disable EE-environments

In the current system architecture, the EE-VM is not put into use but initialized along with each node on the canvas. Thus, disabling the initialization of EE-environment at the beginning of simulation would slightly help speed up simulation duration. To do this, one can modify the EE_machine id in constants.py, then shut down the

initialization trigger in node.py. However, disabling the EE_VM might lead to errors due to code dependencies.

8.6. Drone-environment establishment

The existing system design initializes the drone with same VM as the ship. In the future, it would be a convenient way to swap it to separate VM template to have a better version control of drone implementation. This could be helpful to dispatch drones for a mission.

8.7. Sequencing Constraint

Keep the produced lists (nodes, directions, velocities, locations, etc.) sequential to preserve compatibility with the existing simulator architecture.

8.8. Handshake

During the project proposal phase of the project, we came up with the idea of implementing a drone deployment handshake (Fig. 20). This would be needed when two ships detect a weaker signal strength and must decide which ship would be the one to send a drone first. Due to the complexity of the algorithm at hand, we decided to use a simpler decision algorithm, by allowing the ship with the lower ID to deploy its drone first.
A proposal for future improvements would be to implement this formal handshake. Below is a concept diagram created earlier in the design stage.

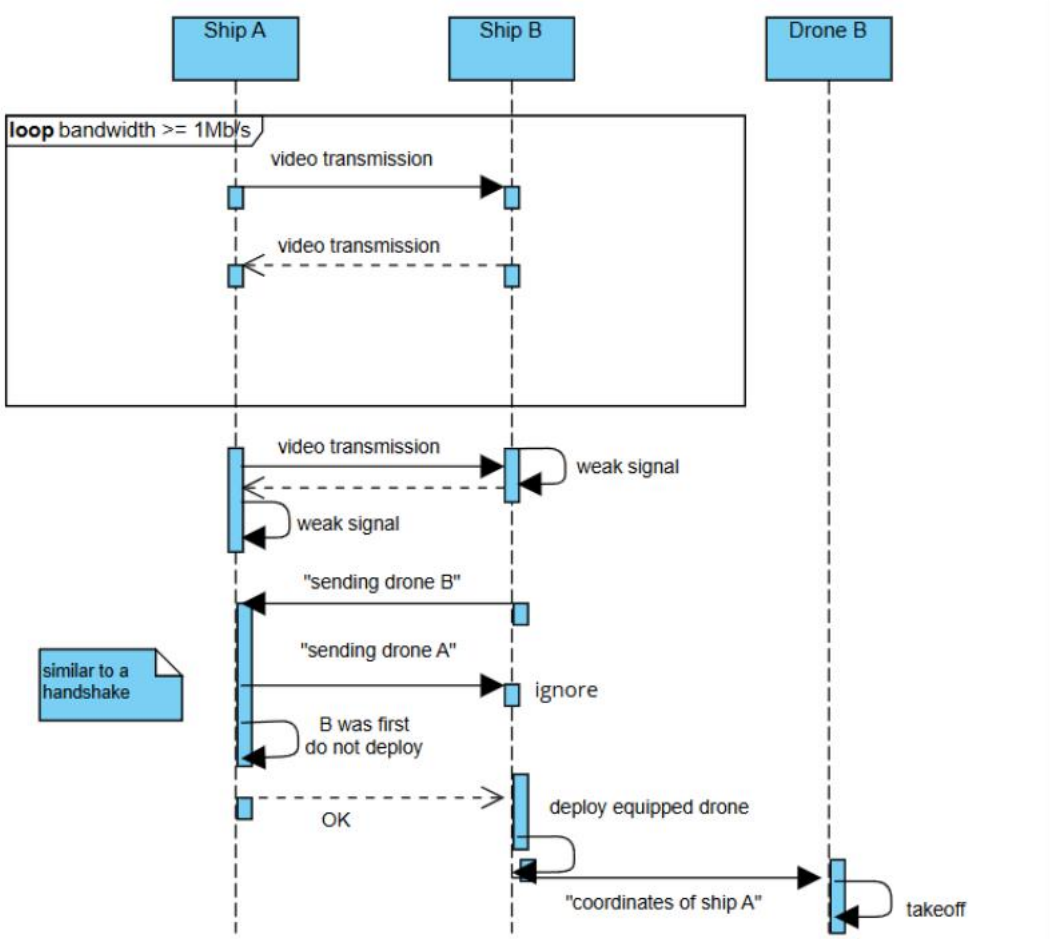

Figure 20 Handshake

8.9. Smart Drone Collaboration

The difficulty for drone deployment increases exponentially with the number of ships that we have on the map. In more complex cases, the drones require a rather advanced algorithm which would handle cases with more than three ships. Here, the drones should collaborate on finding an optimal solution, where optimal means maximizing the bandwidth and minimizing the number of drones needed. A potential idea for this is deploying drones in polygons and dividing this list of responsibilities in a smart way between the deployed drones (Fig. 21 and Fig. 22).

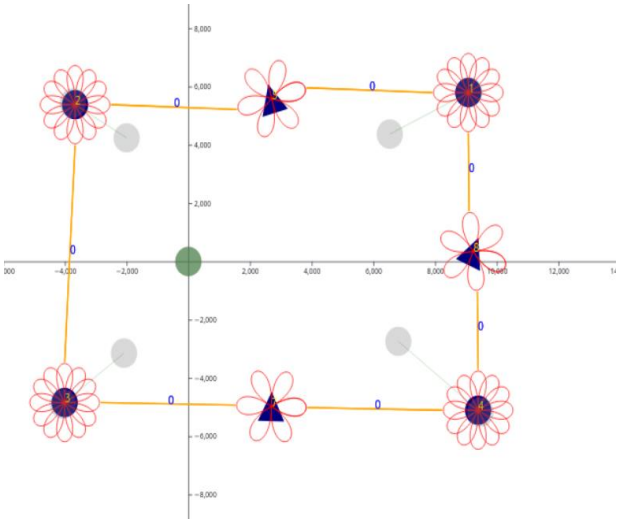

Figure 21 No Drone Collaboration

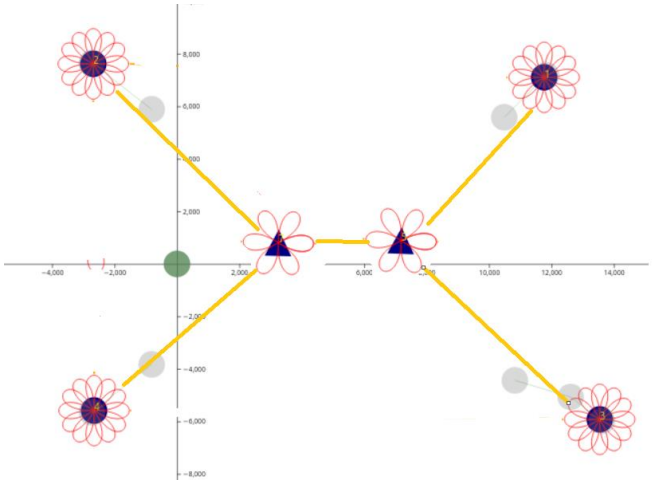

Figure 22 Smart Drone Collaboration

8.10.Redundant drone deployment

Our system sometimes experiences redundant drone deployment. Although the drones return to the mothership quickly, it would be better to prevent redundant deployments. We believe this may have to do with the existing MaritimeManet implementation.

8.11.Restart button

The system does not properly restart drones when using the ‘Restart simulation’ button during execution. Restarting the simulation manually (stopping and restarting) works as intended. When using the button, the drones will still appear on the canvas and attempt to make connections with other nodes. On the other hand, ships reset properly. A potential fix might be hard-resetting the drone states.

8.12.Other autonomous moving vehicles

To facilitate autonomous movement for the drones, a new layer was added to the simulator in the form of the physicalDeviceServer (on the simulator) and physicalDeviceConnector (on the node side). With this new code, allowing other vehicles to move autonomously in the simulator will be easier. Future groups could either choose to copy some of the existing code and modify it or connect to the physicalDeviceServer through other means. If they set up a TCP socket which connects to the physicalDeviceServer, this is possible. It is important to note that the physicalDeviceServer will send updates on a node's location for each simulator step; this is in the form of a specifically formatted JSON message. A node can also change its own direction as frequently as required, provided it sends a similarly formatted message to the physicalDeviceServer. For a clearer understanding of the messages and the different message types, we recommend examining the messagetypes.py and message.py files in our program.

9. Evaluation

9.1. Overall team collaboration

9.1.1. Team organization and communication

The team used Discord for meetings and a WhatsApp group for direct messages and quick communication. Trello was used to assign tasks, and GitLab was used to handle the coding aspect of the project. The physical computer that runs the simulation was hosted by one of our team members and was configured such that it can be accessed remotely by everyone. When debugging our code, we took turns as to who was able to use the machine at what point.

9.2. Planning

We decided to use the Agile method in development, specifically Scrum. This approach was chosen to maintain collaboration and ensure client satisfaction. The reason for this is to ensure that the progress made on the system between each sprint is always aligned with the client's wishes. We would submit progress reports by the end of each week, and the client would offer his feedback. We made sure to incorporate the client's feedback on every iteration of our report and code implementation.

9.2.1. Communication with the client

The team had sprint meetings with the client once a week, every Wednesday. Ahead of each meeting, a progress report was submitted detailing problems that occurred, possible solutions, and the next steps to be taken. During the session, we, together with the client, would clarify certain topics and requirements, and make sure that the product we were developing was still in line with the client's wishes. There have been several occasions when we had an idea of how we should implement something, but when discussed with the client there was a mismatch, and we had to rethink it. There was a Microsoft Teams channel set up by the client, where files would be shared in order to get prompt replies and feedback from him. We made sure to maintain a professional attitude throughout the whole development process.

In week 10, we had a final meeting with our client where we discussed the code handover, as well as our personal experiences with the development. The client and team agreed that there was no need for a formal deed. The client informed us that members at Thales were eager to see our results, so we were excited to present him with our final implementation. The client expressed satisfaction with the final version of our code.

9.3. Member contributions

Adela Simion

Implemented and designed the distributed drone algorithm in collaboration with Lars Gortemaker. Implemented algorithm side, specifically the autonomous drone positioning using signal strength, which was later updated to the current weighted approach. Team-wise, I was the organizer and contact person for the client and supervisor, as well as coordinating the team, keeping track of progress and deadlines. In the report, I wrote part of the algorithm design (justifications and algorithm), part of the Components description, and some of the reflection. Created some of the diagrams.

Lars Wijntjes

Worked on modifications to the simulator together with the rest of the simulator team. In particular, I helped with creating drones on the back end, displaying drones on the front-end, and communication between the back end and front end about the drones. Team-wise, I was responsible for some of our organizational aspects (hosting the Proxmox machine, setting up the shared drive, and updating the user manual), as well as being present at and participating in every scheduled meeting and every presentation. In this report, I wrote parts of the sections of System Requirements, System Architecture, and Testing.

Hayo van de Werfhorst

Worked with the simulator team on designing the modifications for the simulator. Implemented all modifications to the simulator. Heavily reworked and improved the algorithm and peer-to-peer connection to achieve the desired behavior. Documented the modified architecture for the simulator and algorithm components and documented how responsibilities and decision-making work for both ships and drones.

Lars Gortemaker

- Implemented the method to facilitate communication between nodes and the sim-handler (PhysicalDeviceServer and PhysicalDeviceClient)
- Implemented peer-to-peer communication to allow sharing of information between connected nodes. Later improved by Hayo van de Werfhorst
- Designed part of the drone positioning algorithm together with Adela Simion, specifically the weighted positioning idea.
- Created the drone rotation algorithm, seeking to find the best alignment of the drones' antennas with neighboring nodes.

Huanbo Meng

Worked on modifications to the simulator together with the rest of the simulator team, attempted to implement drone creation from backend and assign the drone to a separated virtual machine. Team-wise, contributed weekly progress reports for our client, the presentation slides, and the project proposal. In the report part, I wrote part of the introduction, the simulator design and justification, future works on simulator side, part of testing, the rest of team reflection, created some of the diagrams, and modified the report following Jan's feedback.

Luca Vlasceanu

Was part of the simulator team, helped forward the simulator's progress. Contributed to weekly progress reports, as well as structuring the peer review presentations. Report-wise, created the report's structure, fully completed the inter-component part, and managed/changed/reviewed any information about how the simulator works, rewrote and restructured a bit of the introduction, and ensured that the report has a natural flow.

9.3.1 Actual team performance

Based on the progress reports that we submitted and individual team contributions, we now present a table and a graph of the team's performance. Following our quantified evaluation, we will reflect on how this project shaped our ability to work collaboratively and make realistic technical decisions to increase our future performance throughput.

Firstly, based on the progress reports, we observe how the implementation of tasks actually happened, and how many story points have been deducted each week.

Week	Task	Team members	Story points	Total
1	Initial client clarification and team formation	All	1	1
2	Setup the Proxmox machine remotely and understanding simulator's architecture	Lars W., Luca, Huanbo	4	10
	Reading existing code structure and communication model	All	6	

3	Gitlab setup, workflow planning and team split	All	4	8
	Drafting project proposal and reviewing with supervisor/client	Adela(lead), the entire team	4	
4	Implementing drone object in the simulator's backend	Luca, Huanbo, Lars W.	9	15
	Early algorithm case modelling and decision triggers	Adela and Lars. G	6	
5	Drone deployment backend, visual rendering	Luca and Huanbo	9	17
	First algorithm sketch and messaging structure	Adela and Lars G.	8	
6	Full autonomous drone movement and drone routing	Luca, Huanbo, Lars W.	10	22
	Full independent algorithm implementation	Adela and Lars G.	12	
7	Drone weighted coordinate algorithm optimization	Hayo, Lars G., Adela	14	28
	Drone rotation algorithm implementation	Adela, Hayo, Lars. G	14	
8	Algorithm-Simulator integration and debugging	Adela, Lars G., Hayo	12	19
	Poster, report writing, final code documentation	All	7	
9	Algorithm-Simulator behavior tuning	Adela and Lars G.	7	12
	Manual testing	All	5	
10	Poster and supervisor presentation	All without Luca	5	9

	Final polish and hand-in deliverables		4	

Now, we will overlap the ideal, proposed, and actual trajectories. After the table that shows the real story point completion, Fig. 23 result will illustrate the comparison.

Week	Ideal remaining	Proposed remaining	Actual remaining
1	127	140	140
2	113	130	130
3	99	118	122
4	85	104	107
5	71	89	90
6	57	62	68
7	43	29	40
8	29	9	21
9	15	4	9
10	0	0	0

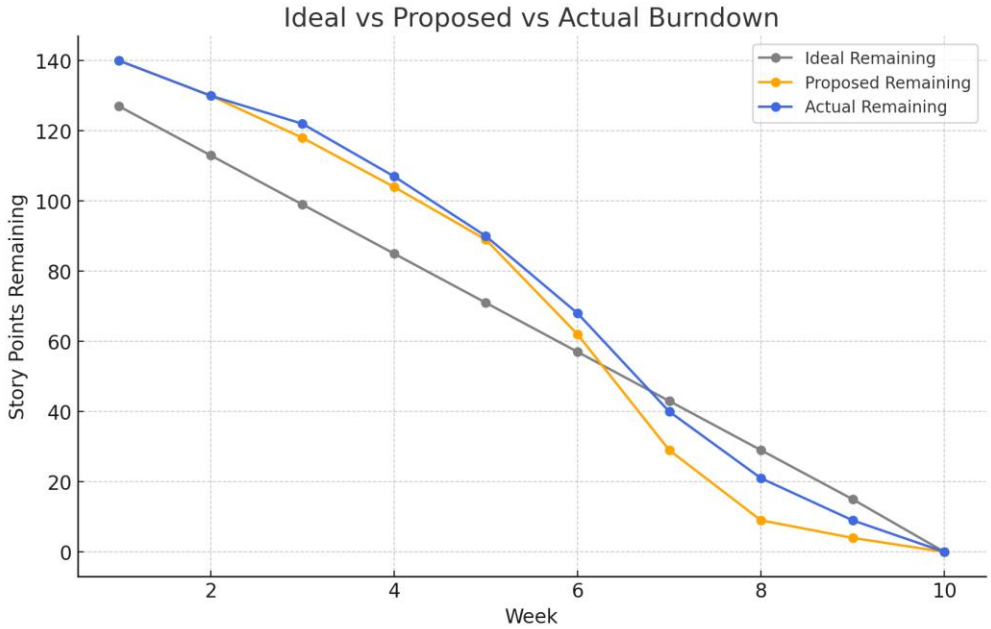

Figure 23 Ideal vs Proposed vs Actual performance

9.4. Team reflections

At the beginning of the project, we had some delays caused by the fact that the physical simulation system had to be handed down to us by a previous student. We did not have access to the Proxmox server until week 3, but after we acquired access to it, everything got back on track. During this time, we drafted a project proposal, which went under many changes with the supervision of the client. After analyzing the requirement of this project, we decided to divide our team into two sub-teams, with one team focusing on studying and expanding the simulator side (the front-end

and back-end creation of drones and the spawn logic of drones), while the other team focuses on the possible and accessible algorithm for the needs of the client. At the beginning, the simulator implementation was quite daunting, and we struggled a lot. Because this project was handled by multiple past student teams, the structure and coordination of the code was rather chaotic and complex, which led to the simulator team's progress progressing slowly in the early stage. Fortunately, we recognized the problem and raised it during our client meetings, to let him know that we are experiencing some difficulties. The client provided us with the previous student papers, which contained the necessary documentation for us to get accustomed to the existing codebase. However, as it was a student-made project, there were bugs present in the code, which we had to fix before proceeding. For example, ships would not communicate if the rotation was zero, which hindered us in the signal strength implementation.

We chose to transfer one team member from the algorithm team to the simulator team, since tackling the simulator code was more challenging. The additional teammate provided a clearer perspective that enhanced our understanding of the existing code architecture and increased our implementation speed. By dividing the team into two parts, two people focusing on the algorithm and four focusing on the more complex simulator part, we could tackle the two problems in parallel.

To conclude, during this module, we learnt that taking over existing projects is a different experience from other projects we had to do in other modules and is rather similar to how things are in the field in real life. Some time is required for understanding the code base before actually implementing anything, and there can always be unexpected moments, such as finding some critical, previously unknown bugs. We had to face other people’s design choices, with sometimes inconsistent code documentation, deal with the redundant code, etc.

Besides coding-related lessons, we also learnt about the importance of good team collaboration and working in parallel, analyzing the current situation and how to keep our project on schedule. We learnt how to plan things at our own pace in order to have a working product by the end of week 10. In the end, this project turned out to be much more challenging than we were expecting in the beginning.

We hope that we were able to make a good contribution to this project, as well as to Thales and our client. We hope that our code is maintainable, such that the future teams that may work on it will not face as many issues as we did. We left extensive comments, especially on the algorithm side, as the logic can get complex and confusing, to ensure that future teams can continue with the implementation of what we outlined in potential future work.

We express our sincere gratitude to our client and supervisor for their continued support during these 10 weeks.