AUTONOMOUS MULTI-AGENT AERIAL MAPPING USING STRUCTURE FROM MOTION AND WIRELESS MESH NETWORKS

by

Raunak Mukhia

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of Master in Engineering in Computer Science

Examination Committee:	Prof. Matthew N. Dailey (Chairperson) Dr. Adisorn Lertsinsrubtavee Dr. Mongkol Ekpanyapong
Nationality:	Indian
Previous Degree:	Bachelor of Technology in Computer Science and Engineering Sikkim Manipal Institute of Technology India
Scholarship Donor:	AIT Fellowship

Asian Institute of Technology
School of Engineering and Technology
Thailand
May 2021

AUTHOR'S DECELERATION

I, Raunak Mukhia, declare that the research work carried out for this thesis was in accordance with the regulations of the Asian Institute of Technology. The work presented in it are my own and has been generated by me as the result of my own original research, and if external sources were used, such sources have been cited. It is original and has not been submitted to any other institution to obtain another degree or qualification. This is a true copy of the thesis, including final revisions.

Date: 13 Jan 2021

Name (in printed letters): RAUNAK MUKHIA

Signature:

ACKNOWLEDGMENTS

I hold sincere gratitude towards my faculty advisor, Dr. Matthew N. Daily for his guidance, encouragement, and direction. I am extremely grateful to him for supporting my study, taking the time to review and correct my report, and for providing me with equipment even outside working hours. I would also like to extend my deepest gratitude to my examination committee members, Dr. Adisorn Lertsinsrubtavee and Dr. Mongkol Ekpanyapong for their suggestions and time throughout the study. I would also like to express my deepest appreciation to Dr. Lertsinsrubtavee for his constant support and guidance. My thesis would not have been possible without Dr. Daily and Dr. Lertsinsrubtavee.

I am deeply indebted to IntERLab for providing me with the infrastructure and networking equipment to complete my study. I would like to express my heartfelt appreciation to Nunthaphat Weshsuwannarugs, Kalana Jayaratne, and Nisarat Tansakul for their help towards making the study realize into a real drone with a mesh network towards the end of my study.

I would like to thank our CSIM secretary, Sireekant Thanwongpan, for scheduling and arranging presentation dates with my examination committee.

I also thank my parents, family, and friends who encouraged and supported me throughout the time of my study. I and extremely grateful to my mother for her continuous support and motivation to complete my study.

ABSTRACT

Drones have become inexpensive and can be used to survey an area of interest and send real time data to the ground control station. However, low-cost drones suffer from the constraints of limited flight time due to battery limitations. If the region is denied of network infrastructure (cellular network) such as in remote areas or disaster then the range of the drone is limited to the wifi network range. A system that uses multiple drones can be used to survey and acquire real-time data as it reduces the amount of flight time for each drone. The range of the drones can be extended using a wireless mesh network with each drone working as a node of the mesh.

For such systems, the operator can either set the flight path for each drone before the mission or many operators can fly the drones individually. However, it becomes the responsibility of the operators that the drones do not collide with each other, and they do not fly beyond the range of the mesh network. The mesh nodes are mobile, therefore the area of coverage of the network is dynamic throughout the mission.

In this study, I propose a centralized system called Pegasus for autonomous exploration and aerial map building with multiple drones using a wireless mesh network. The different phases in the pipeline used in this system to achieve autonomous flight and map building are described. The pipeline consists of the presentation layer, multi-agent coverage path planning layer, motion control layer, real time image acquisition layer, and map generation layer using structure from motion. To evaluate the system, a simulation framework is presented in this study. Finally, the system is evaluated in the real world with a single drone.

Keywords: Multi-agent CPP, Wireless mesh network, SfM, Autonomous UAV

CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1
INTRODUCTION

1.1 Background of the Study

The availability and financial accessibility of unmanned aerial vehicles (UAVs) have made them more widespread, with applications ranging over a wide range of civilian activities. Multi-rotors are used for recreational flying, research, cinematography, disaster observation, logistics, agriculture, public safety, construction, surveillance, and environmental protection, amongst other purposes.

A cluster of inexpensive autonomous multi-rotors connected through a wireless mesh network that can be deployed in a post-disaster situation could help avoid dangerous situations faced by ground-based human observers in such scenarios. The drones should be able to perform aerial mapping quickly, as they can coordinate so that one drone does not need to visit locations already visited by other drones in the cluster. The range of the cluster can also be large, since each drone would act as a wireless mesh access point. At the same time, each drone could act as an access point to provide connectivity in the disaster-stricken area.

Aerial mapping drones can be outfitted with an array of sensors such as ultrasonic sensors, infrared sensors, multi-array laser sensors, and RGB-D cameras. However, systems with many sensors will suffer from integration complexity, weight constraints, reduced battery lifetime, and high cost. Structure from motion (SfM) using monocular cameras attached to each drone provides a low cost, lightweight, simple alternative to sensor-intensive approaches.

This study proposes a system for command and control of a cluster of autonomous drones, each being a node in a wireless mesh network, controlled from a centralized control station that oversees coordinated exploratory path planning and aerial mapping using SfM.

1.2 Statement of the Problem

An inexpensive cluster of UAVs providing autonomous coordinated aerial mapping after a disaster can be a valuable asset for disaster observation and public safety that does not pose any risks for ground-based surveillance personnel. Multiple drones will reduce flight time and increase the spatial extent of the SfM result. A wireless mesh network can also be an inexpensive solution when communication infrastructure such as cellular data services are dysfunctional. Increasing the number of drones in the system can enable increased range of the mesh network, because each drone acts as a mesh node. Providing emergency network services using UAVs and mesh networks in disaster-stricken areas is an active research area, as discussed by Chand et al. (2018), but the best method to combine coverage for mapping

while maintaining mesh connectivity is an open problem. Sabino et al. (2018) discuss optimal placement of UAVs in a mesh network, and separately, a survey of the multi-agent coverage path planning problem (CPP) by Almadhoun et al. (2019) reviews many CPP methods proposed in the literature. However, the requirement for mesh network connectivity imposes severe constraint on the movement of agents, making the resulting ”constrained CPP” an open research area.

Simulation environments exist that can be used for multi-drone planning and wireless mesh networks. Pixhawk PX4's software in the loop (SITL) along with Gazebo can simulate a physical fleet of UAVs with cameras. Wireless mesh networks can be simulated by Network Simulator 3 (NS3). The Robot Operating System (ROS), with its range of tools and available packages, provides a strong platform to develop this system. FlySimNet by Baidya et al. (2018) is a UAV network simulator created by combining NS3 and Ardupilot's UAV SITL simulator with lightweight publish/subscribe (ZMQ) based middleware. It uses a custom graphical ground control system (GCS) application that communicates via the ZMQ messaging service, which does not support ROS, and cameras are not supported by Ardupilot SITL. The CORNET middleware framework by Acharya et al. (2020) combines Gazebo, NS3 and PX4, but it is available only as a published paper and implementation of their method is not quite clear. Neither of these frameworks implement wireless mesh network simulation on NS3. Gazebo, NS3, PX4, and ROS together provide an open source platform, but integration of these tools is required.

This thesis aims to design and implement a multi-agent CPP method that satisfies wireless mesh network constraints, runs multi-agent SfM. It also aims to develop a simulation framework to support the CPP method.

1.3 Objectives of the Study

The main objective of this thesis is to improve on the state of the art in mapping disaster-stricken areas by designing and implementing a method for autonomous control of a multi-agent exploration and coordinated aerial mapping team performing SfM. The focus of this study is to map a disaster-stricken area with the possible damage to communication infrastructure; therefore, the system will use a wireless mesh network for internal communication, with each agent working as a mesh node. The objectives can be decomposed into the following specific tasks:

- Design and implement a user-friendly pipeline utilizing the Robot Operating System (ROS) that allows the operator to select an area to survey.
- Design and implement strategies to find the transformations between the global map and the local map of each drone continuously such that the control station can

coordinate the position of each drone with respect to the global map.

- Design and implement a method for establishing a reliable communication channel between each drone and the control station over the mesh network.
- Design and implement an autonomous exploration and coordination system for the agents that operates within the restriction of the range of the wireless mesh network. The system should maximize the extent of the mappable area, by planning an appropriate arrangement of agents over time.
- Implement a multi-agent SfM method capable of generating an aerial map of the area under surveillance.
- Design and implement a simulation framework for the entire system using PixHawk PX4 as the drone flight controller, Gazebo and Network Simulator 3 as simulation tools, ROS for overall coordination, and useful robotics/vision algorithms.
- Design and implement a wireless mesh network with each drone as a mesh node. As the drones are fast-moving agents, this will require the study and implementation of solutions to maintain a reliable network with a usable quality of service, despite constant topology changes.
- Evaluate the system's ability to execute effectively in the real world.

1.4 Limitations and Scope

The study will not cover the following.

- *Obstacle detection and avoidance*: The control station will generate paths for the drones, such that they will not collide with each other. It is assumed that no unknown obstacle will be present at the drone's operational altitude. It is the operator's responsibility to indicate the minimum altitude during mapping.
- *Dynamic path reconfiguration*: The control station will generate complete paths for the drones before the start of the mission.
- *Recovery mechanism*: If communication is lost with any drone, it will return to the home position using the default behavior in PX4. The remaining drones will continue their operation.
- *Global Positioning System failure*: It is assumed that GPS will be continuously functional.
- *Number of drones in evaluation*: The simulation environment have been tested with a maximum of three drones, real world validation uses only one drone.

1.5 Organization of the Thesis

The thesis is organized as follows:

- Chapter 2 provides a literature review related to this study.
- Chapter 3 proposes the methodology for the system developed in this study.
- Chapter 4 presents experimental results using the system developed in this study.
- Finally, chapter 5 concludes the thesis and provides recommendation for future work.

CHAPTER 2
LITERATURE REVIEW

This chapter begins with the review of the functional architecture of autonomous driving vehicles. Then coordinate system is discussed to lay the ground work for the maps used in the system. After that the types of wireless mesh network and the protocols used for different scenarios are discussed. Following that computer network simulators and, robotic simulator and robotic software are briefly reviewed. Subsequently, Robot Operating System is described. The chapter ends with a review of different multi-agent coverage path planning in literature and a description of structure from motion.

2.1 Software Architecture of Autonomous Driving Vehicles

Behere and Törngren (2015) splits the major components of the motion control part of autonomous driving systems into three main categories as shown in Figure 2.1. These categories are:

- Perception of the external environment in which the vehicle operates
- Decision making and control of vehicle motion with respect to the external environment that is perceived
- Vehicle platform manipulation, which deals mostly with sensing, control, and actuation of the vehicle, with the intention of achieving the desired motion.

Each category can be further broken down into several components. I describe each component in more details in the following sections.

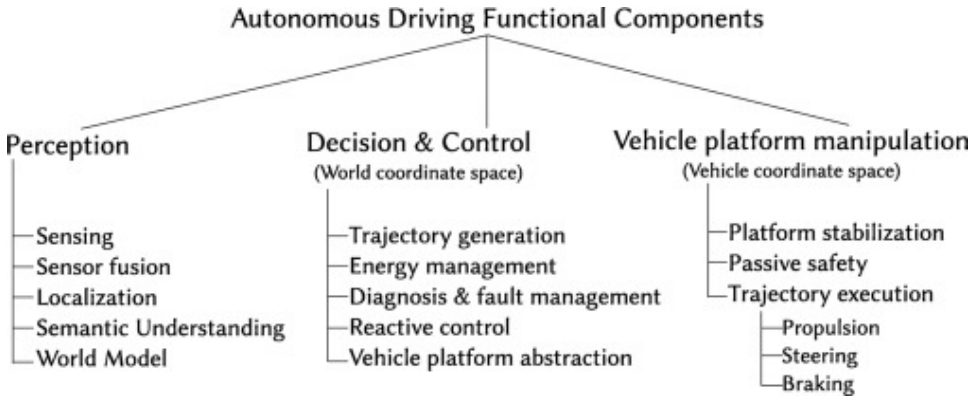
2.1.1 Perception

Sensing components sense the state of the vehicle and the state of the environment in which the vehicle operates. The sensor fusion component considers multiple sources of information to construct a hypothesis or belief about the state of the environment. The localization component determines the location of the vehicle with respect to a global map. The semantic understanding component processes the sensor input and derives meaningful information from it. The world model component holds the current estimate of the state of the external environment.

2.1.2 Decision and Control

The trajectory generation component repeatedly generates a set of obstacle-free trajectories in the world coordinate system and picks an optimal trajectory from the set. Energy management components deal with energy management of the vehicle. Diagnosis and fault management monitors the state of the overall system and its components. Reactive control components are used for immediate responses to unanticipated stimuli from the environ-

Figure 2.1
Functional architecture of an autonomous driving system. Reprinted from Behere (2016).



ment. The vehicle platform abstraction component refers to a minimal model of the vehicle platform.

2.1.3 Vehicle Platform Manipulation

The platform stabilization component's task is to keep the vehicle platform in a controllable state during operation. Trajectory execution components are responsible for executing the trajectory generated by the decision and control component.

2.2 Coordinate Systems

In this section I describe the different coordinate system conventions in use in robotic applications.

2.2.1 Earth Centric, Earth Fixed

The earth-centered earth-fixed (ECEF) coordinate system, rotates with the Earth and has its origin at the center of the Earth. Refer to Figure 2.2 (a) for a visualization. ECEF follows the right-handed coordinate system convention. Wikipedia (2019) describes ECEF as follows.

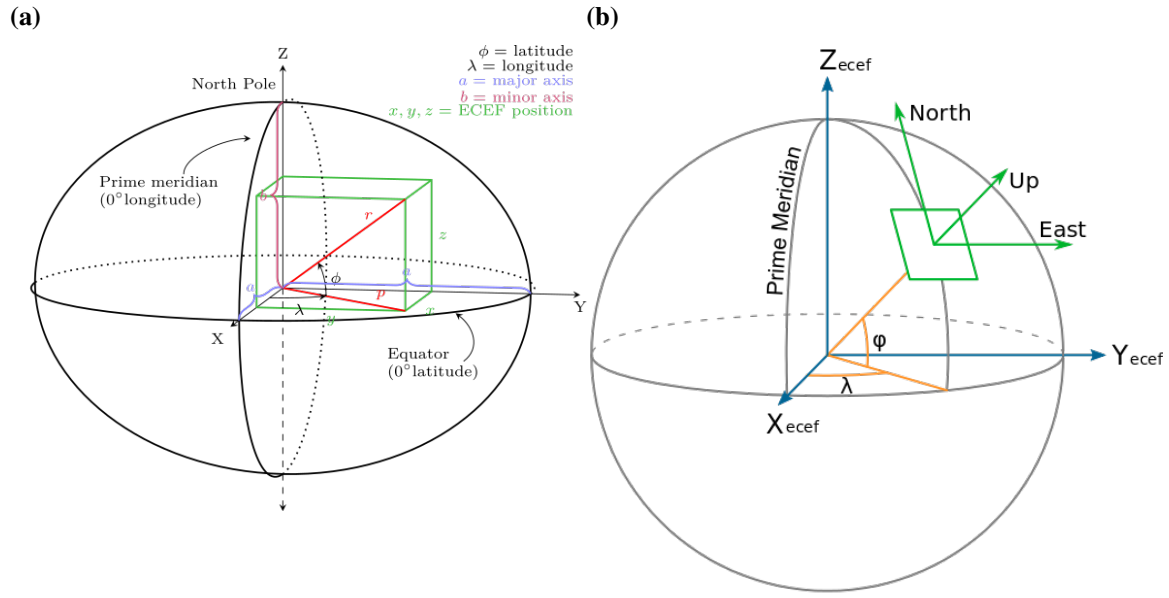
- The origin is at the center of mass of the Earth, a point close to the Earth's center.
- The Z axis is on the line between the magnetic north and south poles, with positive values increasing northward (but not exactly coinciding with the Earth's rotational axis).
- The X and Y axes lie in the plane of the equator.
- The X axis passes through the point on the equator from 180 degrees longitude (negative) to the prime meridian (0 degrees longitude, positive).
- The Y axis passes through the point at 90 degrees west longitude along the equator (negative) to 90 degrees east along the equator (positive).

2.2.2 World Geodetic System

The World Geodetic System is a standard system used in GPS, cartography and geodesy. The latest revision is WGS 1984 (WGS 84) that was established and is maintained by the

Figure 2.2

Coordinate systems. (a) Earth centric, earth fixed (ECEF) coordinate system. (b) Local tangent plane coordinate systems. Reprinted from Krishnavedala.



United States National Geospatial-Intelligence Agency. It was last revised in 2004. The World Geodetic System 1984 (WGS 84) is one of the best global geodetic reference system for the Earth available presently. The datum (ellipsoid) defined by WGS 84 is generally used to convert the GPS latitudes and longitude to the UTM coordinate system.

2.2.3 Universal Transverse Mercator Geopraphic Coordinate System

The Universal Transverse Mercator (UTM) is a coordinate system that divided the earth into 60 zones. Each zone is a plane with its own x and y coordinates in meters. It ignores altitude and treats the earth as a perfect ellipsoid. A UTM coordinate has:

- A zone number.
- A hemisphere (N/S).
- An easting (X coordinate).
- A northing (Y coordinate)

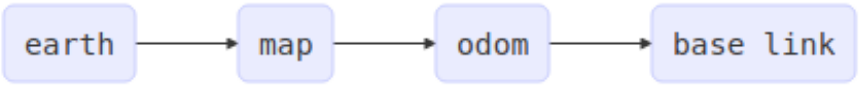
A coordinate 14.08057584 °N, 100.61284553 °E in latitude and longitude is represented as 47 N 674132 1557234 in UTM.

2.2.4 Local tangent plane

In the local tangent plane coordinate system, a position on the earth is fixed about the origin. There are two conventions as shown in Figure 2.2 (b).

- X=East, Y=North, Z=Up (ENU).
- X=North, Y=East, Z=Down (NED), which is commonly used in aviation, as the objects of interest usually lie before an aircraft (down or positive Z).

Figure 2.3
Relationship between ROS frames. Reprinted from Meeussen (2010).



2.2.5 PixHawk and ROS coordinate systems

PixHawk follows the NED convention, whereas ROS follows the ENU convention. The conversion between these different conventions is handled automatically by MAVROS. To translate air-frame related data, a rotation of 180° degrees is applied about the roll (X) axis. For local data, a 180° rotation is applied about the roll (X) and 90° rotation is applied about the yaw (Z) axes.

It is good practice in ROS to set up reference frames as described by Meeussen (2010), shown in Figure 2.3.

- The coordinate frame called base_link is rigidly attached to the mobile robot base. It can be attached in any arbitrary position or orientation.
- The coordinate frame called odom is a world-fixed frame. The pose of a robot is continuous in this frame, but it can drift over time.
- The coordinate frame called map is a world fixed frame, with Z-axis pointing upwards.
- The coordinate frame called earth is the origin of the ECEF frame.

2.3 Wireless Mesh Network

A wireless mesh network's (WMN) consists of two types of nodes, mesh routers and mesh clients. Mesh routers form the mesh backbone for mesh clients. Every node in the mesh has the capability to route packets forward for other nodes if the packet destination is not in direct wireless transmission range. Mesh routers provide functionality as bridge/gateway that enables the WMN to integrate with other networks such as cellular, ad-hoc and ethernet. In a wireless mesh network, all the participating nodes automatically establish and maintain mesh connectivity and are in effect dynamically self-organized and self-configured. WMN can be rapidly deployed where there is no network coverage and has the potential for application in disaster management.

2.3.1 WMN Network Architecture

A wireless mesh network has two types of nodes, mesh routers and mesh clients. Mesh routers can have multiple networking interfaces to provide bridge/gateway functionality to the mesh network and have additional routing functions to support mesh networking. Mesh

routers can be built on similar hardware platform as a conventional wireless router. Mesh routers can be build based upon embedded systems and general-purpose computer systems. Mesh clients have capability for mesh networking and can also work as a mesh router, nevertheless they lack bridge/gateway function and usually have only one wireless interface.
The architecture of WMNs can be classified into three types based upon the composition of mesh routers and clients.

- *Infrastructure/Backbone WMNs*:In this architecture mesh routers form an infrastructure for clients that connect to them. The mesh routers automatically establish and maintain links between themselves and provide gateway functionality to connect the mesh network to other networks and the internet.
- *Client WMNs*: In this architecture, mesh routers are not required for maintaining a mesh network. Mesh clients provide peer-to-peer networking, performing routing and configuration management. A packet may reach the destination node through multiples hops through client nodes.
- *Hybrid WMNs*: In this architecture, mesh routers provide infrastructure backbone with access to other networks. Client nodes can connect to the mesh network through meshing with other clients, as well as through mesh routers. This in effect, is a combination of infrastructure and client WMNs, and provides increased connectivity and coverage.

2.3.2 Features of WMNs

WMNs have redundant nodes and can provide high level of fault tolerance and robustness. If the case of link failure, the mesh network can adapt and reroute the data. Using multihop strategies, the coverage range can be increased, as intermediate router and clients can relay packets between hosts which do not have a direct line of sight. WMNs can be rapidly deployed with less maintenance because of self-forming, self-healing and self-organization capability through Mobile Ad hoc Network (MANET) routing protocols. They are compatible with ad-hoc wireless clients.

2.3.3 Routing Protocols for WMNs

WMNs has redundant and mobile clients, hence, the routing protocols should find and maintain routes between the source and destination nodes by detecting and responding to changes in network topology. Ideally routing protocols should maximize the capacity of the network and minimize the packet delivery delays by creating and selecting efficient routes between nodes.
Wireless mesh routing protocols can be classified into three categories:

- *Proactive Routing Protocol*: Proactive routing protocols build and maintain the routing table by sharing routing data among nodes at periodic intervals. It selects

the route from the routing table while forwarding packets from one node to the other. Optimized Link State Routing Protocol (OLSR) and Better Approach to Mobile Ad-hoc Networks (BATMAN) falls under this category.

- *Reactive Routing Protocol*: Reactive routing protocols searches the route from source to destination node on demand as it is required. Examples of this category are Ad Hoc On-Demand Distance Vector Protocol (AODV) and Dynamic Source Routing Protocol (DSR).
- *Hybrid Routing Protocol*: Hybrid Routing Protocol uses both reactive and proactive routing strategies depending upon the environment. As a reactive routing protocol, it will provide the path on demand, based on up-to-date information. Examples of hybrid routing protocols are ZRP (zone routing protocol), HSLS (hazy sighted link state routing protocol).

2.3.3.1 Comparision of Routing Protocols Mbarushimana and Shahrabi (2007) does a comparative study between the performance of reactive (AODV and DSR) and proactive (OSLR) protocols. They run simulations for 1000 seconds, considering four different scenerios, namely, workload, number of flows, number of nodes in the network, and node movement speed. The performance of the three routing protocols is measured based on throughput, good-put, routing load and end-to-end delay. They come to the conclusion that OSLR shows the best performance in terms of data delivery ratio and end-to-end delay, and provide insight that network layer tends to drop more packets while reactive protocol is computing the route to the destination. In their study OSLR still outperforms the reactive routing protocols at higher speeds even though it has the highest routing overhead due to its exchange of periodic routing updates.

Kulla et al. (2012) does a performance comparison of OSLR and BATMAN routing protocols in an indoor stairs environment of 5 floor building. They come to the conclusion that both the protocols perform well for one-hop and two-hop communication. The delay and packet loss increased after three hops for static node scenario and after two hops for moving node scenario. BATMAN performs better than OSLR for packet loss metrics because BATMAN buffers packets when routes are unstable. In general, OSLR protocol showed better performance than BATMAN protocol. OSLR shows better performance than BATMAN regarding the delay metrics, which is critical for real-time communication between a cluster of drones.

From these studies of performance comparisons OSLR seems to be the most suited to my use case, as:

- Proactive protocols (OSLR) performs better than reactive protocols in terms of data delivery ratio and end-to-end delay.

- Reactive protocols tends to drop packets while they are computing the route.
- OSLR shows better performance than BATMAN for delay metrics.
- Latest information about the drones in the mesh network is more critical than stale information, which means OSLR with less delay is more suited to the use case of my study.

2.3.4 Optimized Link State Routing Protocol

Optimized Link State Routing Protocol (OSLR) belongs to the class of link state routing protocol, with optimization for mobile ad hoc networks. It is a distributed protocol with no central agency that exchanges topology information with other node of the network periodically. Each node maintains a routing table, and is a proactive protocol. When data is to be sent, it refers to the routing table. The route to other nodes are maintained by each node proactively.

HELLO message is used for link sensing, neighbor detection, and multipoint relay (MPR) selection. HELLO messages are not forwarded by the nodes. The nodes in OSLR use only selected nodes called multipoint relay (MPR) to re-transmit control messages, reducing overhead from control messages flooding the network. Each node in the network selects a set of nodes in its symmetric 1-hop neighborhood called MPR set of that node. Neighbors which do not belong to MPR set of the node receive and process broadcast messages but do not re-transmit it. If an interface of a node has at least one verified bi-directional link with an interface of another node, it is considered as its symmetric 1-hop neighbor.

Topology Control (TC) message is used to disseminated topology information through the network, using optimized flooding through MPR mechanism. The TC messages distributed to each node is used to calculate routing table in each node.

If a node has multiple interfaces, it must announce, periodically, information describing its interface configuration to other nodes in the network. Multiple Interface Declaration (MID) message is flooded by such nodes using MPR mechanism in the network, and is used for routing table calculations.

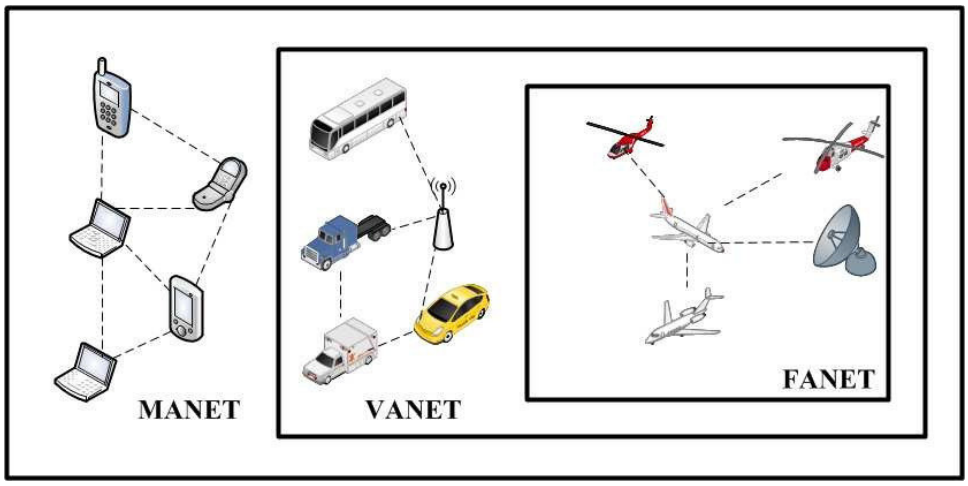
Gateway/bridge nodes have non OSLR interfaces connected to external network such as the ethernet. Host and Network Association (HNA) message are periodically broadcasted by such nodes to register such routing information in the routing table.

The role of OSLR is not to forward data packets. It maintains the routing table in each node which is used by the system to perform network layer forwarding function.

2.3.5 FANET

Flying ad hoc network (FANET) is a specialized subset of mobile ad hoc network(MANET). FANET takes into consideration the challenges faced by MANET with flying UAVs as the mesh nodes. FANET can also be classified as a subset of Vehicular ad hoc network

Figure 2.4
MANET, VANET,and FANET Reprinted from Tareque et al. (2015).



(VANET), where the mesh nodes are vehicles on the ground. The relation between MANET, VANET and FANET is given in Figure 2.4.

Surveys of FANET has been carried out by Chriki et al. (2019) and Tareque et al. (2015). The challenges of FANET described can be summarized as follows:

- *Network topology change*: In FANET, rapidly moving UAVs changes the network topology more frequently.
- *Node mobility*: UAVs have a higher degree of mobility than vehicles or people on ground. Node mobility issues can be considered as the most significant difference between FANET and other ad hoc networks.
- *Node density*: Node density in FANET is lower than other MANETs. FANET nodes are spread across the sky. There are less number of average nodes in a unit area in FANET.
- *Radio propagation model*: MANET and VANET nodes are close to the ground and because of the environment, they may not have direct line of sight. FANET nodes are in the sky and for most cases have direct line of sight between the nodes.
- *Power Consumption*: For large UAVs power consumption is not a significant factor when designing the FANET. However for small UAVs the power consumed by FANET must be taken into consideration.
- *Localization*: UAVs because of its higher mobility degree may have inaccurate GPS position information as compared to other MANETs.

Routing protocols for FANET follows similar classification as WMNs and are as follows:

- *Static protocols*: These protocols have fixed routing tables.
- *Proactive protocols*: Updates the routing tables periodically.
- *Reactive protocols*: Finds path to the destination node when data needs to be sent.

- *Hybrid protocols*: These use both reactive and proactive strategies when required.
- *Geographic based protocols*: These protocols use position information.

There are multiple optimization and extensions for OSLR specifically done to solve challenges for FANET. Some of them are:

- *DOLSR*: Alshabtat and Dong (2011) proposes Directional Optimized Link State Routing Protocol (DOLSR) which uses directional antenna and heuristic to minimize the number of MPRs. They show reduced end-to-end delay and enhanced overall throughout using DOLSR for FANET.
- *P-OSLR*: Rosati et al. (2013) proposes Predictive OSLR (P-OSLR) that uses GPS information to help the routing protocol. It takes the relative speed between the nodes in account to calculate expected transmission count (ETX) metric. This metric is used by OSLR for link quality sensing. The show increased multi-hop reliability with minimum outage time as compared to OSLR in FANET environment.
- *ML-OSLR*: Zheng et al. (2014) proposes Mobility and Load aware OSLR (ML-OSLR) that introduces mobility and load aware algorithm in the routing protocol. ML-OLSR does not select high speed nodes as the MPR and avoid routing through high load, congested and high speed nodes to increase network stability. They show increased performance in packet delivery ratio and average end-to-end delay compared to OSLR with FANET based simulation.

Singh and Verma (2015) presents the application of OSLR to FANET under different mobility model of the nodes through simulation. He shows that the increase in speed of nodes decreases throughout, increases end-to-end delay and decreases packet delivery ratio for most mobility model. However, for my study, the speed of the node are relatively low, and nodes spend most of their time waiting for other nodes to arrive to their way-point and take picture, therefore I shall purse the use of generic OSLR for this thesis.

2.4 Simulators

An approximate imitation of a process or system's operation over time is known as simulation. A real-life or hypothetical situation modeled on a computer that captures the behavior of how the system works is a computer simulation. Computer simulation can be a powerful tool to develop, investigate the behavior and save cost, by providing a means to virtually access the system under study.

Some of the classification of simulations are:

- *Continuous simulation*: Simulation is based on continuous time and uses numerical integration of differential equations.

- *Discrete-event simulation*: Simulation is based on discrete time intervals. The state of the components of the simulation change their value only at discrete time.
- *Stochastic simulation*: Simulation with same input will produce different results within some confidence interval. During simulation some variables or process is subject to random variations.
- *Deterministic simulation*: Simulation with same input will always produce the same deterministic results. The variables and process are regulated by deterministic algorithms.

2.4.1 Computer Network Simulators

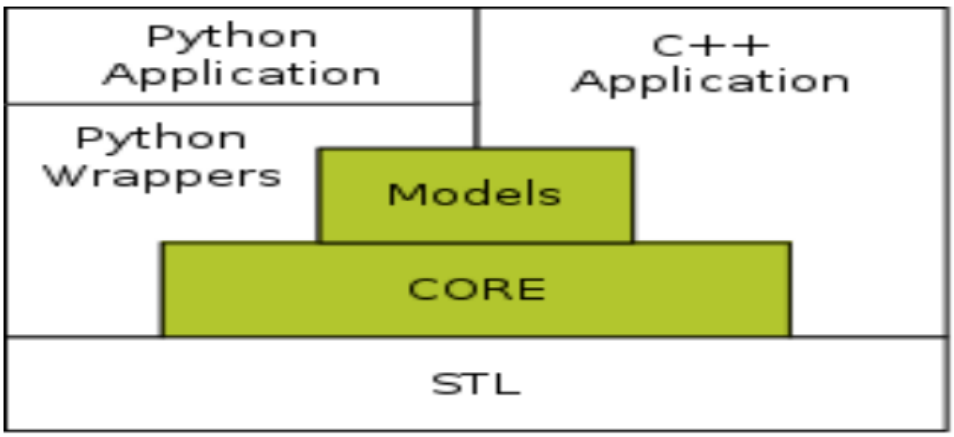
Network simulators are those types of software that are capable of performing predictions on how a computer network will behave. The output of these simulations can be analyzed to derived metrics of the network under consideration. The use of network simulators provide a cost-effective method to validate and optimize the performance of the network before deployment. Network simulators may have the capability for emulation, that is, to pass real packets through the network simulation, and examine the effects of the network on the packets.

Siraj et al. (2012) has done survey on network simulators (ns-2, ns-3, OPNET, NetSim, OMNet++, REAL, J-Sim and QualNet), their availability, programming languages used and their architecture. Toor and Jain (2017) has done survey on network simulator which can support wireless infrastructure (ns-2, Ns-3, J-Sim, OMNeT++, OPNET, QUALNET and MATLAB) and listed their key features and limitations. Patel et al. (2018) has done survey on network simulator (ns-2, ns-3, OMNeT++, NetSim, REAL, OPNET and QualNet) and listed their features, advantages, disadvantages.

Network Simulator 3 (ns-3) is a discrete-event network simulator, and it's intended uses are for research and education. It is written from scratch using C++ and is not backward compatible with ns-2. NS-3 is a collection of core libraries, modules, and applications which are build using a python based build tool called waf. It can be linked with a user C++ or python application to create a simulation. It is possible to integrate ns-3 with other software through the user application. The architecture of ns-3 is illustrated in Figure 2.5. In this paper I use ns-3 because:

- It is intended for scientific research.
- It is more flexible than other simulators.
- It tries to simulate protocol genuinely.
- It is open source and free to use.
- It can be integrated with other software.
- It is actively developed and maintained. ns-2 development stopped around 2010.

Figure 2.5
Ns-3 architecture. Reprinted from Siraj et al. (2012)



2.4.2 Robotics Simulator and Robotic Software

Applications for physical robots can be created using robotics simulator without depending on the real hardware. This saves time and cost in the development of the application. Certain robotics simulator can render 3D model of robots and its environment, and can emulate robotic models, sensors, and control in virtual environment. They use physics engine to make the simulation more realistic.

Staranowicz and Mariottini (2011) has conducted a survey presenting comparison between the popular commercial and open-source robotic software for simulation and interfacing with real robots. Ivaldi et al. (2014) has conducted an online survey where participants present their feedback about the tools and use of dynamic simulation in robotics.

The most used robotics simulator are:

- *Gazebo.* Gazebo can simulate multiple robots in outdoor environments. It supports multiple physics engine. It can integrate with ROS using a set of ROS packages named gazebo_ros_pkgs. Gazebo is open source and free to use.
- *ODE.* Open Dynamics Engine is a open-source physics library for simulating rigid body dynamics. It is used in computer games and simulation tools. It is open source.
- *Bullet.* Bullet is an open-source physics library and is used in 3D animation software and game engines. It is open source.
- *V-Rep.* V-Rep is a robot simulator software developed by Coppelia Robotics. It is free for academic usage. Commercial license is not free.
- *Webots.* Webots is an open-source robot simulator developed by Cyberbotics. It was open sourced in 2018. Robots can be modeled, programmed and simulated with the provided development environment.

In this study, I intend to use Gazebo as the robotics simulator because:

- The UAV firmware PixHawk provides SITL simulation for Gazebo and is the recommended option.
- Gazebo supports PixHawk provided quad-copter (Iris).
- Gazebo supports multiple quad-copters.
- External world can be built in Gazebo using real world height maps.
- Gazebo supports sensors such as GPS and cameras.
- Gazebo can be integrated with ROS.
- Gazebo server provides API to integrate with other software. In my use case, I update the FANET nodes in ns-3 using the position provided by the robotics simulator.

2.5 Robot Operating System

Quigley et al. (2009) presents an overview of Robot Operating System (ROS). ROS is a framework for robotics software. It provides set of tools and communication layer on top of a host operating system, which helps in developing software for robotics. Availability of plethora of open source ROS packages avoids reinventing the wheel while developing the application.

ROS provides peer-to-peer connection to different processes of the robotics system. Each process is called a node and handles a specific aspect of the robotic system. The peer-to-peer topology makes use of a master node which enables the running nodes to find each other at run-time. Nodes exchange information with each other using messages. A message is a predefined strictly typed data structure. Nodes can exchange messages using either topic or service.

Nodes can publish message on a topic. Another node which uses the data generated by it can subscribe to the topic to receive messages. Nodes and topics have many-to-many association with each other. A single node can publish and subscribe to many topics and a topic may be published and subscribed by many nodes at once.

For synchronous communication, a node may call a service registered by another node. A node sends a request message to call a service and the node providing the service will reply with a response message. Unlike topics, service of a particular name can only be advertised by one node.

Nodes, topics and services can be grouped into namespaces. Topics and services follow URI reference naming convention. For example a topic that publishes GPS information for UAV 1 in a multi-UAV system may be named as */uav0/global_position/gps*.

ROS has transformation system called tf/tf2. Tf2 is the new iteration of the transformation library. The transformation system helps the user keep track of multiple coordinate frames

over time by using a tree structure to maintain relationship between coordinate frames. It allows for easy transformation of data between any two coordinate frames.
ROS has a wide array of tools for creating, debugging, inspecting and visualizing the data exchanged between the different nodes which speeds up the development.

2.6 Multi-Agent Coverage Path Planning

Coverage path planning (CPP) is the process of finding a suitable path that passes through a set of way-points in order to completely explore the area or volume of interest. The CCP problem is closely related to the lawnmower problem described by Arkin et al. (2000), and is proven to be NP-hard even for simple polygonal regions. Lawnmower problem is defined as finding a path/tour π such that every point of the region R is covered by some placement of the agent along π.
Galceran and Carreras (2013), on their survey of CPP methods mentions other closely related problems:

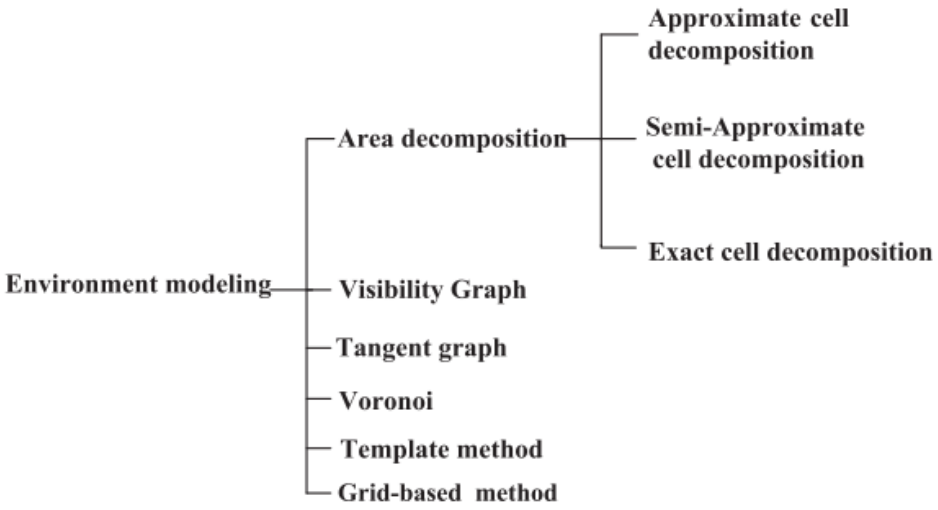
- *Covering salesman problem*: A generalization of the traveling salesman problem.
- *Art gallery problem*: The minimum number of guards needed to station in a polygon area such that each point is visible to atleast one guard.
- *Watchman route problem*: The shortest route in a polygon area such that each point is visible from the route.

All the above mentions problems are proven in general cases, to be NP-hard. Therefore, the solution to CPP problems are approximations to the problem.
Multiple agents can be helpful for the CPP task in many ways. Multiple agents can decrease the time taken to cover the area of interest and can be more robust as the exploration task can still be carried out if case of an agent failure. Multi-agent CPP is the process of finding a set of suitable paths through a set of way-points where the union of the found paths completely explore the area or volume of interest.
Almadhoun et al. (2019) has carried out an extensive survey for multi-robot coverage path planning for model reconstruction and mapping. They classified the topics involved into:

- Viewpoint generation,
- Coverage path planning approaches,
- Communication and task allocation, and
- Mapping.

Viewpoint generation can be generalized to generating potential way-points that can be used in coverage planning. Visiting all the potential way-points will result in complete coverage of the area under scrutiny.
In another survey by Xu et al. (2019), which focuses on multi-agent coverage search, the

Figure 2.6
Classic environment modeling methods. Reprinted from Xu et al. (2019).



topics have been classified as:

- Environment modeling,
- Agent deployment, and
- Search Path Planning.

Environment modeling is a special case of viewpoint generation when the obstacles in the area to search is known beforehand. For unknown and dynamic environments, non-model based viewpoint generation is applied.

2.6.1 Viewpoint Generation

Viewpoint generation is the process of finding points or sub-regions in the area of interest. If all the viewpoints are visited by the participating agents, then complete coverage of the area is obtained. There are two methods in which viewpoints are generated, model based, and non-model based.

In model based viewpoint generation, prior information about the environment to survey is known. In non-model based viewpoint generation, the next viewpoint is generated on the fly using real time information about the environment. The classic methods for environment modeling are shown in Figure 2.6. Frontier based multi-robot approach for coverage of unknown environment proposed Dileep Muddu et al. (2015) is an example of non-model based viewpoint generation, where the robots expands the explored region in a coordinated manner.

2.6.2 Coverage Path Planning Approaches

Coverage path planning for multiple agents can be classified into methods for static and dynamic environment. In static environment the obstacles in the area of interest do not change during the course of the mission. In dynamic environment, obstacles move and new

obstacles are introduced during the course of the mission.

The static environment path planning approaches can be categorized as:

- *Grid based*: The area of interest is decomposed into cells and are allocated to different agents. Each of these cells are covered using coverage approaches such as lawnmower pattern, boustrophedon algorithm, etc. to obtain complete coverage.
- *Geometry based*: Visibility graph is used in generating the paths. In visibilty graph, the nodes represent the viewpoints and edges are line segments that do not pass through any obstacle. The most used geometric based method in multi-robot CPP is the Voronoi diagrams.
- *Sampling based*: Search path through random sampling in the search space. Probabilistic roadmap algorithm (PRM) and rapidly exploring random tree algorithm (RRT) are examples of sampling based search algorithm.
- *Heuristic search*: Search path using heuristics. A* algorithm, BA* algorithm purposed by Viet et al. (2013) that combines boustrophedon motion and A* search agorithm, and ant colony algorithm are examples of heuristic search based algorithm.

Dynamic environment path planning for mobile agents using distributed reinforcement learning is purposed in their study by Xiao et al. (2020). Ganganath et al. (2016) utilizes distributed antiflocking algorithm to navigate mobile sensor networks in a dynamic environment.

2.7 Structure from Motion

Multiview structure from motion (SfM) problem attempts to reconstruct 3D structure using many images of the stationary scene. SfM estimates the camera pose and the 3D coordinates of features in the scene using overlapping images and attempts to build a sparse point cloud of the scene. During the SfM process, the pose of the camera are estimated which is used later to build denser point cloud using Multi-View Stereo (MVS).

The SfM problem is solved by optimizing the cost function known as the total reprojection error. This optimization is called bundle adjustment, and the objective is to simultaneously compute the calibration parameters for each of the n cameras and the structure of the scene from n images, while minimizing the cost function.

The sparse features are detected using feature descriptors such as scale-invariant feature transform (SIFT) or speeded up robust features (SURF). Matching algorithm such as Lucas-Kanade tracker are used to find pair wise feature correspondence between images. Random sampling and consensus (RANSAC) is used estimate essential matrix between the pair of images and discard out the outlier corresponding features matches. OpenSfM uses itera-

tive bundle adjustment technique, that is images are added and bundle adjustment is solved repeatedly. However Özyeşil et al. (2017) in their paper mentions that global approaches, considering all images simultaneously may provide improved solutions. After reprojection error is minimized for all images, the camera poses are recovered and the 3D coordinates of the features are used to build the sparse point cloud of the scene.

2.8 Chapter Summary

That chapter covers all the different aspects of literature and the tools that will allow me to develop the proposed system in this study.

CHAPTER 3
METHODOLOGY

The different components of the proposed system and how they work together are described in this chapter. This chapter describes in detail the functional components of the system; the presentation layer, planning layer, motion control layer, image acquisition layer and the map building layer.

3.1 System Overview

The system uses multiple drones with PixHawk flight controllers, each paired with an individual companion computer and a ground control station (GCS). The operator sets the origin of the global map and select a region of interest on the GCS using Mapviz, a ROS package. The system creates a grid over the region of interest and calculate paths for the drones. After the expected paths are calculated, the system calibrates the drones by finding an accurate transformation between each drone's local frame of reference and the global frame. To accomplish this, the drones move along a predefined flight path while the transformations between each drone's local map and the global map is refined. After calibration is complete, the system coordinates the flights of the drones as they move along the paths the system calculated, ensuring coverage of the region of interest. While flying their paths, the drones stream images from their cameras to the GCS. On the GCS, the SfM module receives the images and constructs an aerial map (a textured 3D mesh) with respect to the global map frame for the region of interest initially selected by the operator.

All the components runs on a single computer while simulating the system. In simulation mode, PX4 will be run in software in the loop (SITL) mode. Gazebo will be used to visualize the physical world and render simulated video feeds. Network Simulator 3 (ns-3) will be used to simulate the networking functions of the wireless mesh network.

3.2 System Design

The different components comprising the system are described here.

3.2.1 Components

The system can be divided into two domains, one comprising the components in the drones, and the other comprising the components in the GCS. Components running on drones do not share information with other drones; they only share with the GCS. These components will be involved with vehicle platform manipulation and forwarding the images captured. The components on the GCS will be involved with getting the initial region of interest from the operator, planning the path of each drone, coordinating the movement of each drone during

plan execution, requesting and receiving images from the drones, and aerial mapping from the images received.

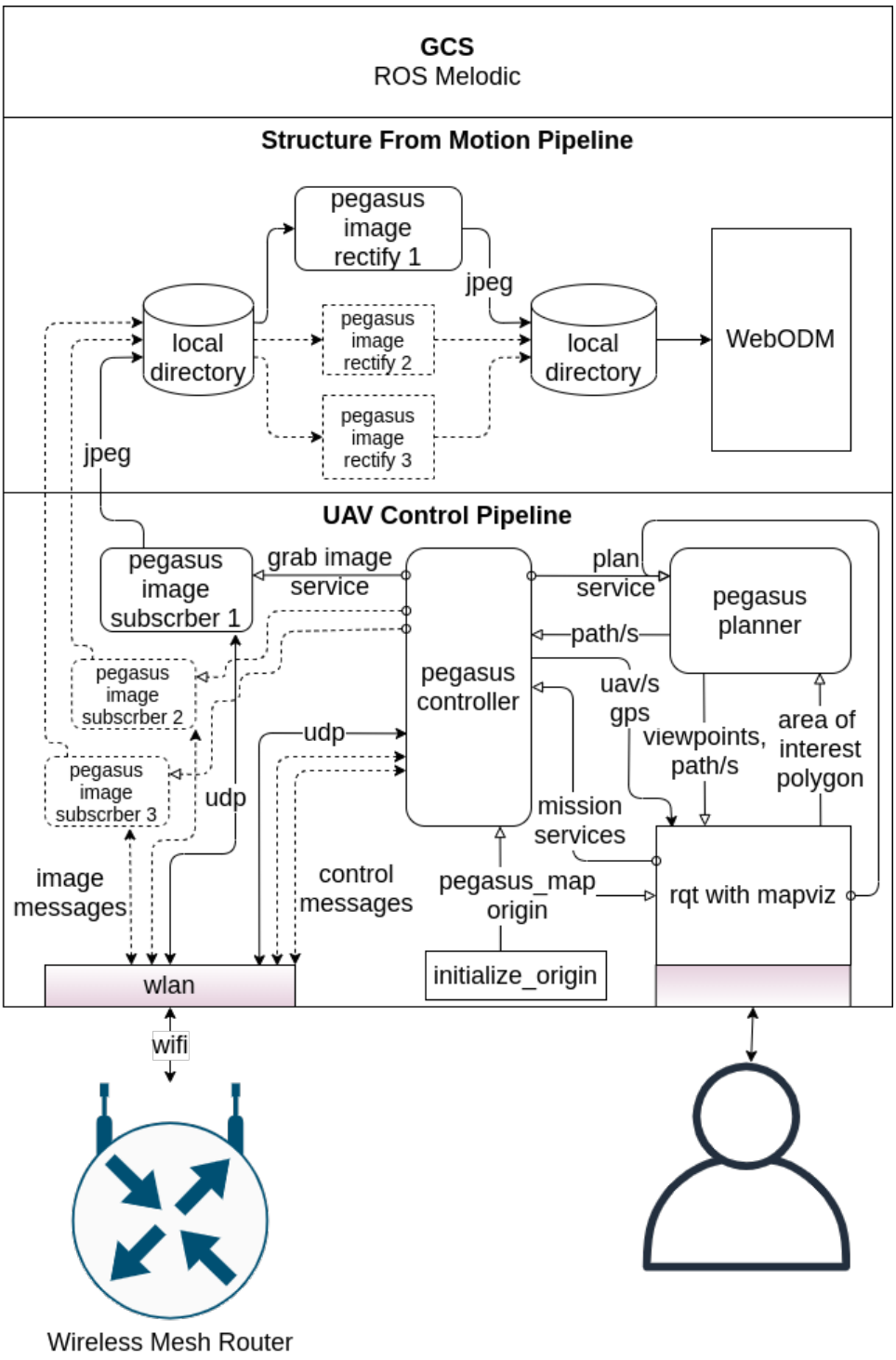
The flight controller is installed with PX4 firmware. The companion computers and the GCS runs Raspbian Linux and Ubuntu respectively with the Robot Operating System (ROS) installed and is connected to a wireless mesh network through wireless mesh routers. The mesh routers form a backbone mesh network and the devices are connected via the gateway network of the routers.

The components on the GCS are illustrated in Figure 3.1 and as follows:

- *Computer*: An i7 laptop with 16 GB of RAM running Ubuntu 18 is the processing unit of the GCS.
- *Wirless mesh router*: Wireless mesh router provides mesh network to the laptop, as well as maintain mesh network with the drones.
- *Robot Operating System*: All the components is designed and implemented with ROS as the base infrastructure. This gives access to the plethora of ROS tools and packages.
- *mapviz*: Mapviz running as a plugin inside rqt is the GUI interface for the system. Mapviz is a ROS-based visualization tool with a plug-in system focused on visualization of 2D data. It shows tiled maps using the Microsoft Bing Maps API and has plugins to select polygons on the map, show markers, show NavSat data, among other features that fit the requirements of this system. Mapviz also allows us to fix the origin of the global map.
- *image_pipeline*: ROS image pipeline is used to rectify the image obtained from the cameras attached to the drones.
- *pegasus_planner*: A new multi-agent path planning ROS node is designed for the project.
- *pegasus_controller*: A new ROS node that calculates and maintains translations between different map frames, monitor links with the drones, transmit path planning information to the drones, and coordinate the drones during flight.
- *pegasus_image_subscriber*: A new ROS node that requests and receives images from the drones.
- *pegasus_image_rectify*: A new ROS node that rectifies the received images for each individual drones' cameras using image_pipeline.
- *Structure from Motion software*: An aerial map is constructed using the rectified images acquired from drones by this component. ODM (Open Drone Map) is used to build the map. ODM uses OpenSfM, an open source SfM software framework.

The components in the drones are illustrated in Figure 3.2 and as follows:

Figure 3.1
Architecture diagram of ground control station.



- *Flight Controller*: The drones uses Pixhawk flight controllers with PX4 as their firmware.
- *Companion Computer*: The companion computer controls the flight controller in off-board mode through a serial link.
- *Mobile wireless mesh router*: The mobile wireless mesh router maintains mesh network using OSLR, and provides network to the companion computer through ethernet link.
- *USB camera*: The USB camera attached to the computer enables the drone to capture images.
- *Robot Operating System*: The software components are designed and implemented as ROS nodes.
- *mavros*: The companion computer uses mavros, a ROS package for MavLink communication between companion computers, and flight controllers, to control the flight controller in off-board.
- *gscam*: Gscam is used to acquire video stream from the camera attached to the drone. Gscam is a ROS package which is based upon gstreamer.
- *pegasus_commander*: A new ROS node designed for the project that receives paths and control messages from the GCS. It publishes and subscribe to mavros for controlling, and receiving status of the drone. It also sends to the GCS, the drone's local poses, GPS positions, flight controller status updates, and GCS link monitoring information.
- *pegasus_image_publisher*: The video feed from the onboard camera is captured, geo-tagged with GPS information and sent to the GCS by this component.

The operator selects the global map origin and region of interest through mapviz. The pegasus_planner creates a grid-based representation of the region of interest and plan the paths for the drones in the global map reference frame. The path planned by pegasus_planner is sent to pegasus_controller. Pegasus_controller runs a calibration routine on each of the drones to find the transformation between the local origin of the drone and the global map origin. These transformations allow the system to maintain the global map (pegasus_map), where planning is carried out and the local map (uav0_map, uav1_map, etc.), where the control of the drones are achieved as shown in Figure 3.3. After calibration completes, pegasus_controller transforms the paths for the drones from global map to the local map for each drone and send it to the respective drone. Pegasus_controller monitors the link with each drone by sending a periodic heart-beat packet. If a drone does not receive a heart-beat packet for some interval, it changes to Return To Home (RTL) mode and abort its current path. To avoid collision, pegasus_controller also maintains a path index counter. Paths are made up of a list of poses.

Figure 3.2
Architecture diagram of drone.

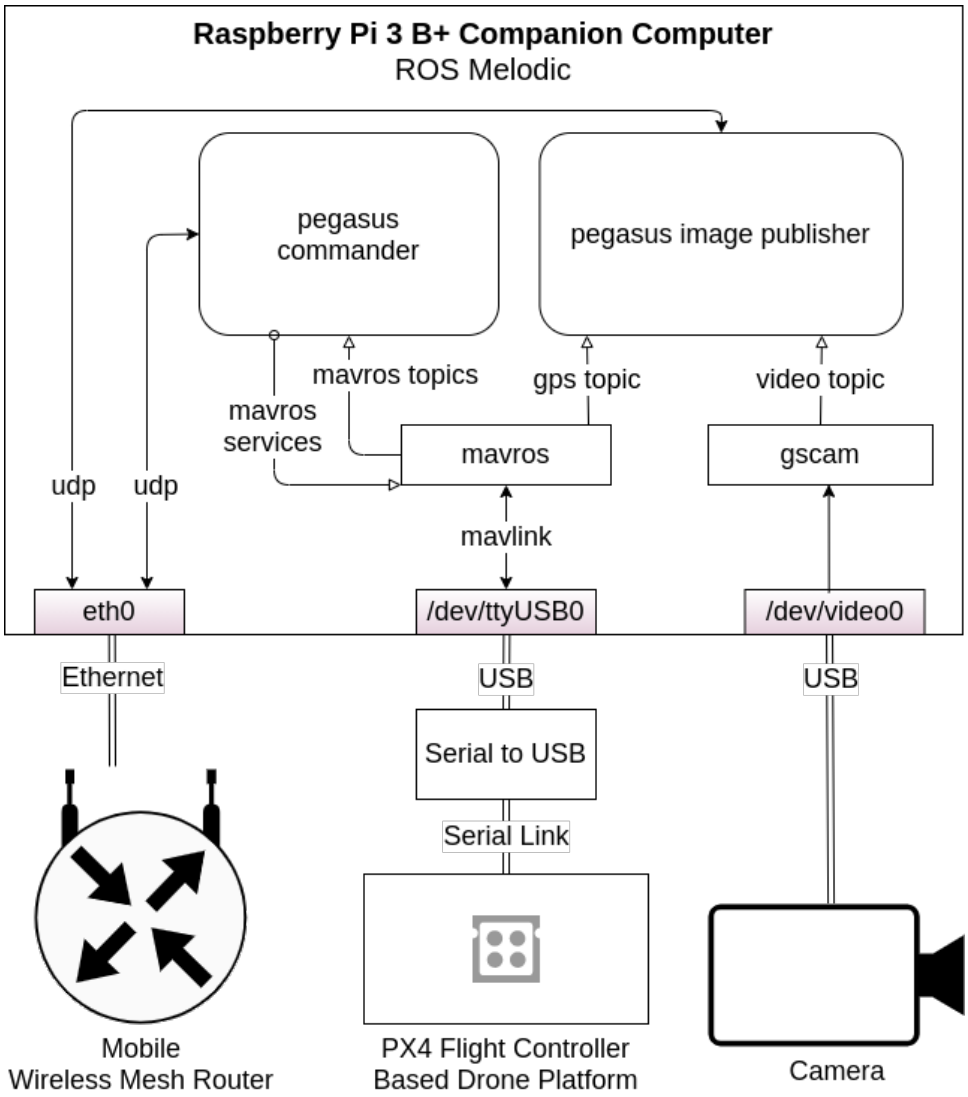
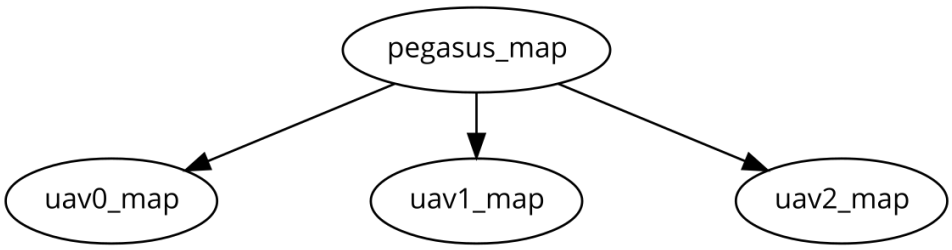


Figure 3.3
Different maps in pegasus system.



Pegasus_controller tells each drone to move to the next pose in its path after all the drones have reached their current goal pose.

The drones uses mavros, a ROS package that exposes MavLink protocol parameters and services as ROS topics and services. Pegasus_commander receives commands and path information from the GCS and executes its plan while sending local poses, global GPS positions, and flight controller state information back to the GCS. Gscam publishes video stream from the USB camera as a ROS camera topic. The pegasus_image_publisher captures images from the published camera topic, apply GPS EXIF tag and sends them to the GCS, when it receives an image request from pegasus_image_subscriber. Pegasus_image_subscriber saves the received images to a local directory on the GCS.

Pegasus_rectify_image rectifies the images received using the camera calibration parameters, prepares the images for map building and saves them to a directory in GCS . The SfM module constructs an aerial map for the region of interest selected by the operator from the rectified images.

3.2.2 Simulation

Most of the components remains the same regardless of whether running in simulation or in the real world. The simulation runs in a single computer. Four more components will be utilized to simulate the system:

- *Gazebo*: Gazebo simulator is used because it supports fleet of drones and camera feed.
- *PX4 Software in the loop*: PX4 firmware is run as software in the loop, to simulate the drone's firmware.
- *pegasus-net-sim*: A custom network simulator 3 module simulates the mobility of the drones, and simulate the network behavior of actual control messages and video feed between the drones and the GCS. It will also publish the SNR status of each device in the network.
- *pegasus_network_status_util_node*: A custom ROS node receives noise and signal strength from pegasus-net-sim and publishes the signal-to-noise ratio (SNR) value

of each drone as ROS topic.

PX4 software in the loop (SITL) is used to simulate the drone. PX4 SITL uses Gazebo to simulate the world, the drone mechanics and the video feed. Pegasus-net-sim, a Network Simulator 3 custom module is used to simulate the wireless mesh network. Pegasus-net-sim receives the model info from Gazebo and update the position of its nodes. Each node in pegasus-net-sim have NS3PegasusApp, a custom ns-3 application installed, that sends and receives packets in the simulated network. Pegasus-net-sim functions as a proxy application that receives data in one udp port, simulate it in ns-3 and transmit it through another udp socket. Using the trace feature in ns-3, pegasus-net-sim publishes the noise and signal value for each node of the wireless mesh network.

Figure 3.4 and Figure 3.5 illustrate the components used in simulation.

3.2.2.1 pegasus-net-sim Network Simulator 3 is used to simulate wireless mesh network infrastructure. Ns-3 provides a framework to write simulations that can be integrated with other software. For this study ns-3 has to:

- Simulate a wireless mesh network with OLSR as the underlying mesh protocol to build the routing table for each node.
- Update the position of the nodes using the pose information from Gazebo. Gazebo library will be used to subscribe to gazebo pose topic.
- Inject and eject real world traffic between GCS and drones to the ns-3 simulation. Pegasus-net-sim will be designed as a proxy application between the components of the pegasus system to intercept traffic among them.
- Trace the signal noise value different nodes and publish it using the trace feature of ns-3.

The architecture diagram of pegasus-net-sim is given in Figure 3.6. NS3PegasusDroneApp is a custom application for ns-3 that will inject and eject the udp packets to and from the simulation. Each ns-3 node will have a NS3PegasusDroneApp installed. A NS3PegasusDroneApp will have a set of PegassUDPSocket encapsulating real udp sockets and its virtual ns-3 socket counterpart. Each PegasusUDPSocket class will have transmit and receive threads. These threads will continuously poll for any packet to send or receive in the real udp socket. PegasusUDPSocket will have a tx and rx lockless queue. Receive thread of a socket will write to the rx queue when it receives a packet. Transmit thread will poll the tx queue and transmit the packet to socket. NS3PegasusDroneApp running in ns-3 simulation will poll the rx/tx queues of the PegasusUDPSocket in its set and inject/eject the packets to its virtual udp socket inside the simulation.

To intercept and route a packet within pegasus-net-sim, it has:

- *port*: The real udp port that intercepts real world traffic.

Figure 3.4
Architecture diagram of simulated ground control station.

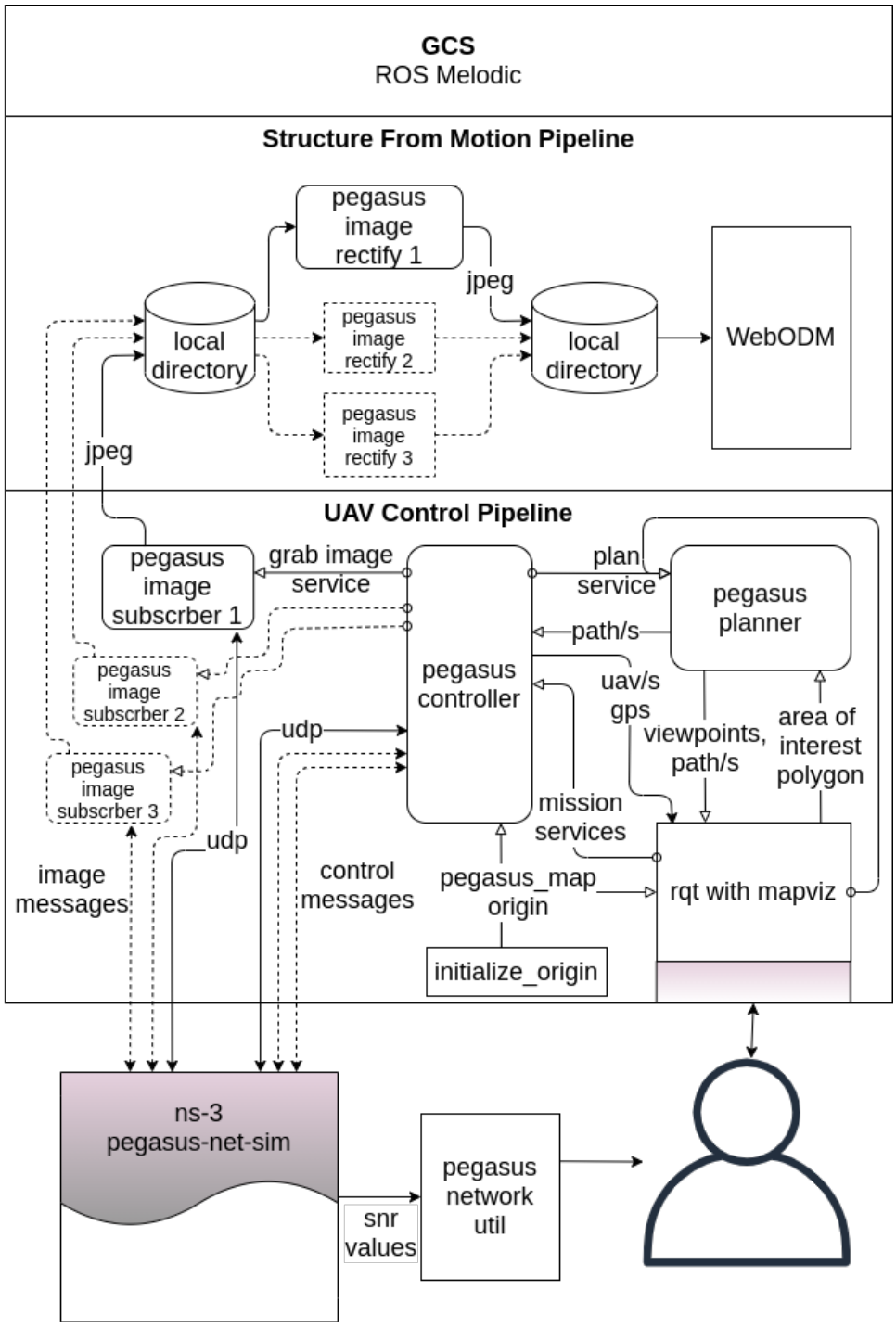


Figure 3.5
Architecture diagram of simulated drone.

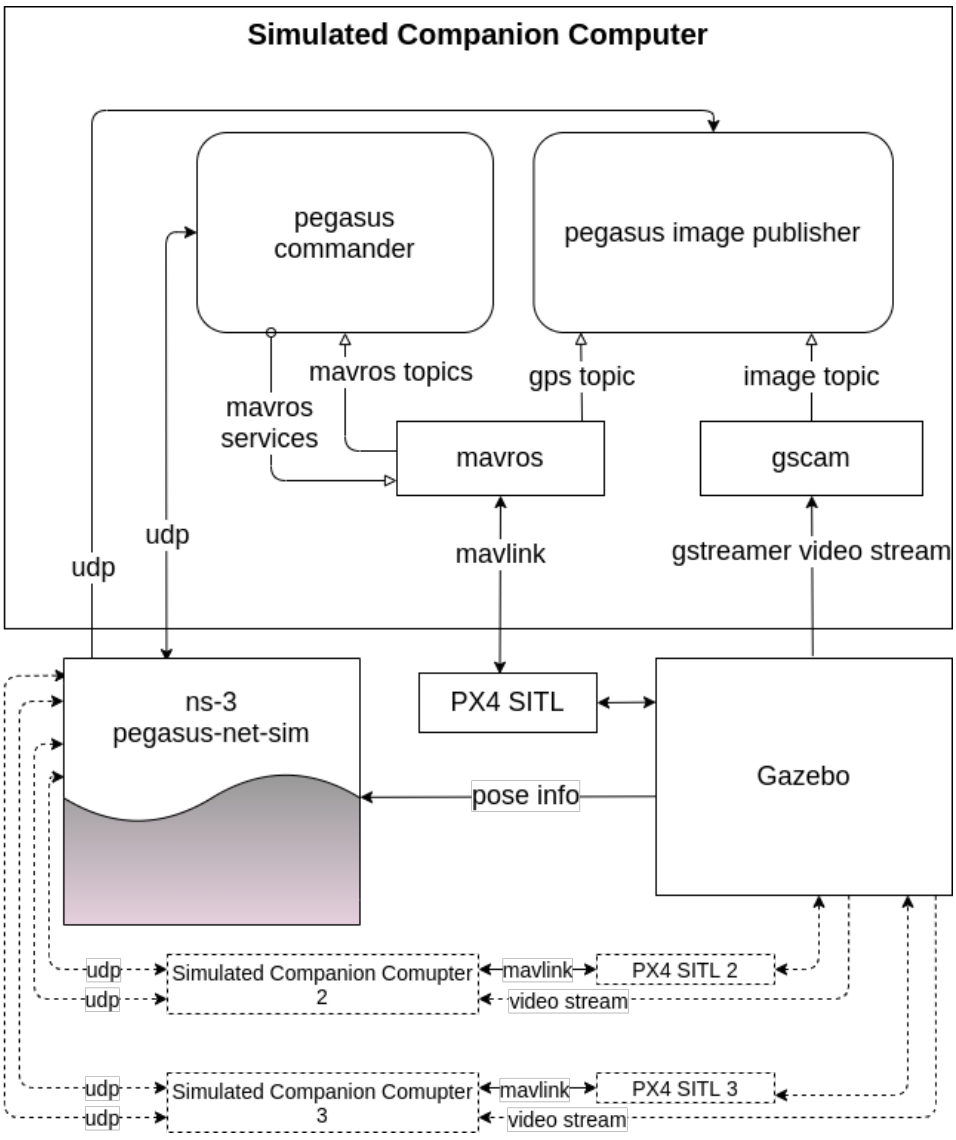


Figure 3.6
Architecture diagram of `pegasus-net-sim`.

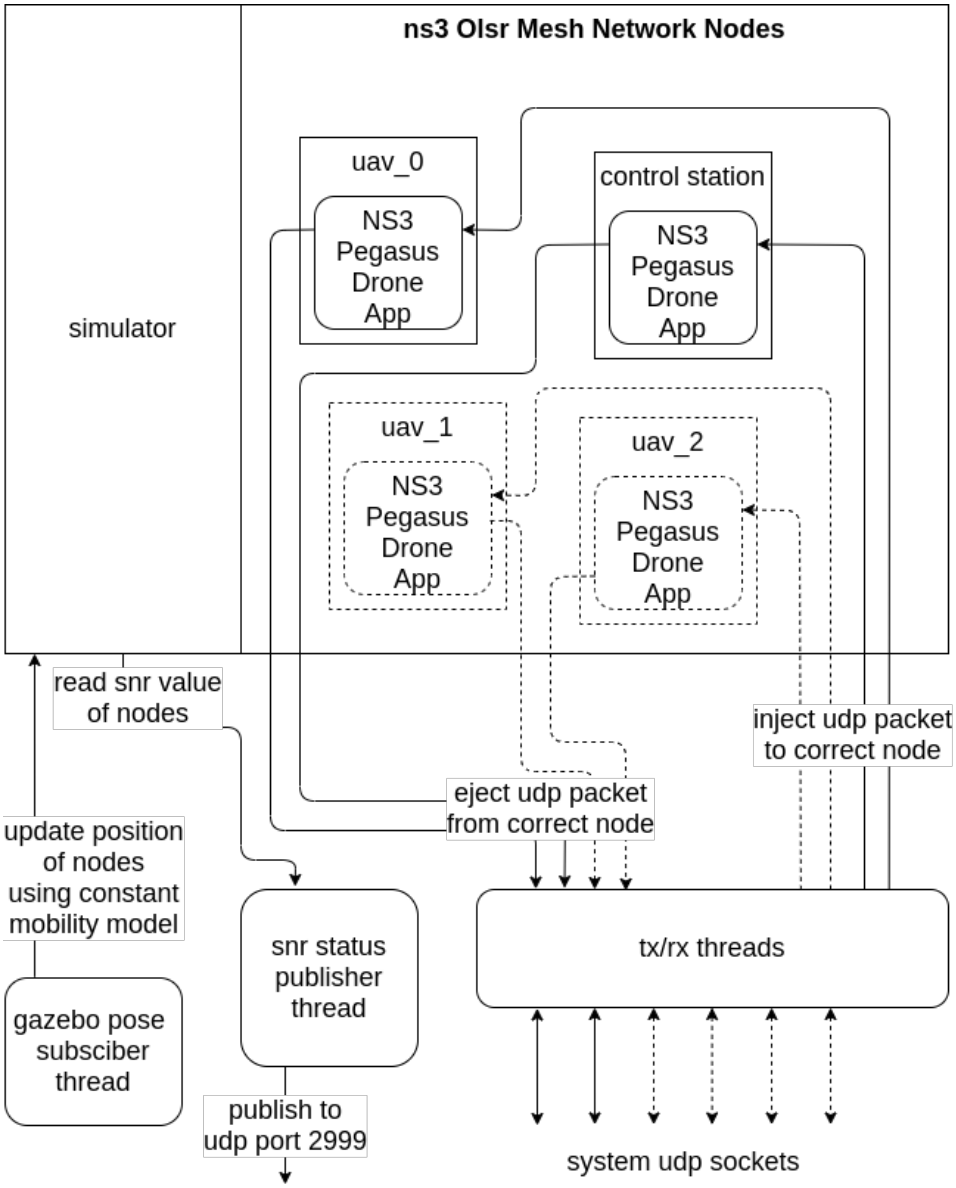
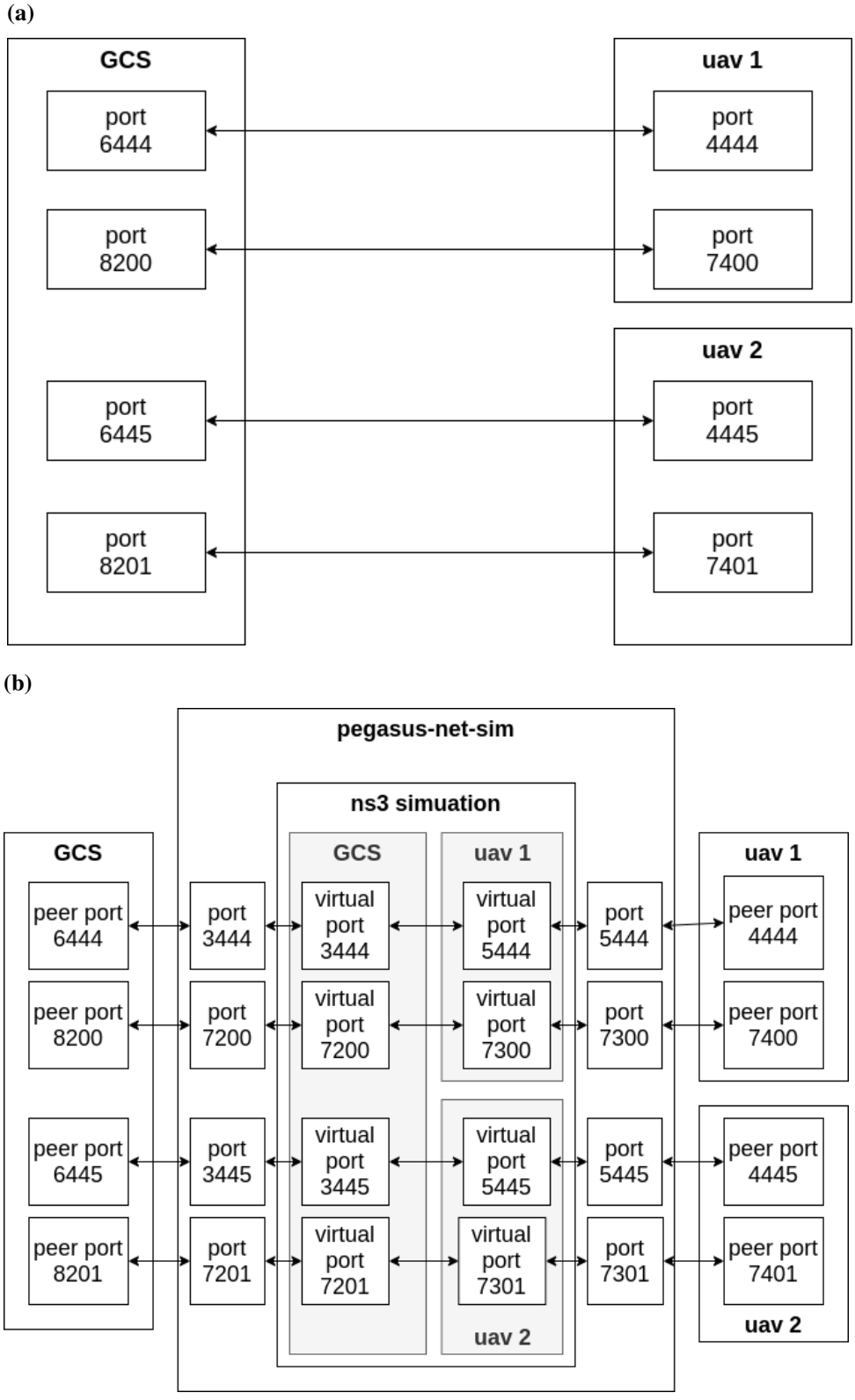


Figure 3.7
A scenario with one GCS and two drone (a) Communication without network simulation (b) Communication packets being routed through `pegasus-net-sim`.



- *virtual port*: The ns-3 virtual port that simulates port.
- *peer port*: The real udp port which the port transmits to.
- *virtual peer port*: The ns-3 virtual udp port that the traffic gets transmitted to.

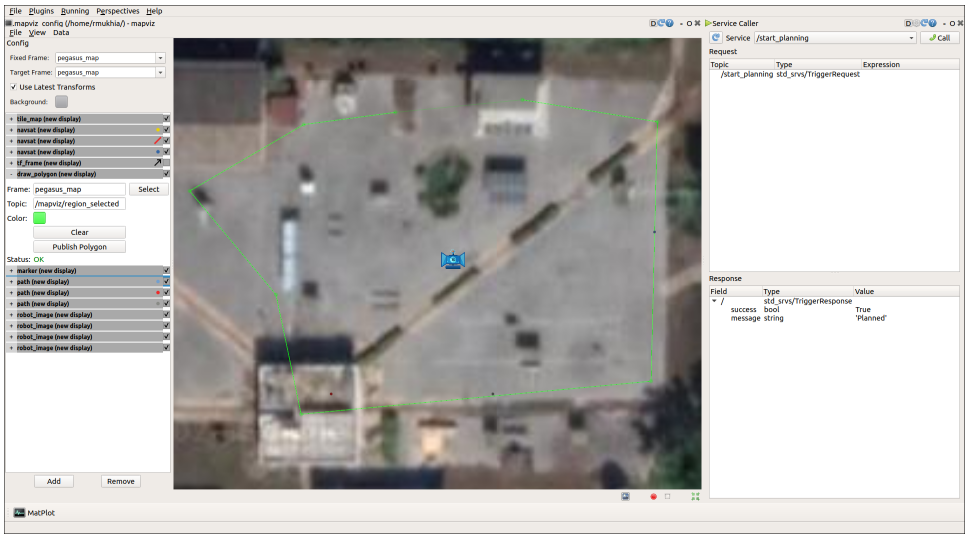
The port and virtual port setting is implicitly set to be the same port number. For the configuration, three elements are defined in the tuple {port, peer port, virtual peer port}. For the scenario given in Figure 3.7, the corresponding configuration in PegasusConfig.cc is:

```
std::map<std::string, std::vector<PegasusPortConfig>>
PegasusConfig::m_config {
  {
    "iris_0", {
      {5444, 4444, 3444},
      {7300, 7400, 7200},
    }
  },
  {
    "iris_1", {
      {5445, 4445, 3445},
      {7301, 7401, 7201},
    }
  },
  {
    CONTROL_STATION_STR , {
      {3444, 6444, 5444},
      {7200, 8200, 7300},
      {3445, 6445, 5445},
      {7201, 8201, 7301},
   }
  },
};
```

3.2.3 pegasus_network_status_util_node

The ns-3 module pegasus-net-sim traces the signal and noise decibel of each packet of each node, smooths and averages it out per 100 millisecond per node and advertises it to udp port 2999. The pegasus_network_status_util_node reads the information on udp port 2999, and publishes signal to noise ratio (SNR) as ROS topics for each node for the user to analyses.

Figure 3.8
ROS rqt with mapviz visualizer. Origin for pegasus_map is set near CSIM.



3.3 Functional Components

The functional software components of the pegasus system can be categorized as:

- *Presentation*: This layer defines how the operator interacts with the system. Solely in the GCS.
- *Planning*: This layer will calculate a list of grid based viewpoints and provide path through those viewpoints for the drones in the system, to cover the region of interest selected in the presentation layer. This layer will be in the GCS.
- *Motion control*: This layer will calculate the drones' local and global map transformations and control the motion of the drones according to path calculated in the planning layer. It will also align the drones in a particular direction at the viewpoints where it needs to capture an image and will request image acquisition service to acquire the image. This layer will be in the GCS as well as the drone.
- *Image acquisition*: This layer will be present in the GCS and the drone. It will provide service to capture image from the drone, apply GPS tags and deliver it to the GCS over the network. The images acquired will be saved in a local directory of the GCS.
- *Map building*: After the drones have covered their paths, the map building layer will process the images in the local directory of the GCS.

3.3.1 Presentation

This layer presents how an operator interacts with the system. The front-end of the system utilizes a ROS package called mapviz. Mapviz is used as a plugin inside ROS rqt. The ROS rqt provides a dashboard for the operator to interact with. The GUI that the operator is presented with is illustrated in Figure 3.8. The presentation layer runs in the GCS.

3.3.1.1 Mapviz Mapviz requires the operator to select the global map origin as a ROS parameter. ROS parameters can be set through the rosparams command-line tool, programmatically or in the launch file. In our case, we set the global map origin though the launch file using the ROS package swri_transform_util.

```
<?xml version="1.0"?>
  <launch>
    <node pkg="swri_transform_util" type="initialize_origin.py"
      name="initialize_origin" >
     <param name="local_xy_frame" value="/pegasus_map"/>
    <param name="local_xy_origin" value="control_station"/>
    <rosparam param="local_xy_origins">
      [{ name: control_station,
         latitude: 14.081104,
         longitude: 100.612743,
         altitude: 7,
         heading: 0.0}]
    </rosparam>
  </node>
  <node name = "pegasus_dashboard" pkg = "rqt_gui" type = "rqt_gui"
    respawn = "false" output = "screen" args =
    "--perspective-file $(find pegasus_ros)/pegasus_ros.perspective"/>
</launch>
```

It publishes two ROS topics:

- */local_xy_origin*: Origin of the global map (pegasus_map).
- */mapviz/region_selected*: The points of the polygon selected by the operator.

The mapviz module can use online map APIs such as google maps and bing maps. For offline use, mapproxy can be used to cache data locally and serve it to mapviz.
`https://github.com/danielsnider/MapViz-Tile-Map-Google-Maps-Satellite` provides a docker image for mapproxy that integrates easily with mapviz.
The flow that the operator has to follow while using this GUI is presented in Figure 3.9.

3.3.2 Planning

This layer computes the path for the drones that covers the region of interest selected by the operator. It has one component, the pegasus_planner.

3.3.2.1 Pegasus_planner Pegasus_planner handles the following tasks:

- *Viewpoint generation*: Generate grid cells based viewpoint in the region of interest.

Figure 3.9
A flowchart of the operator's interaction with the presentation layer.

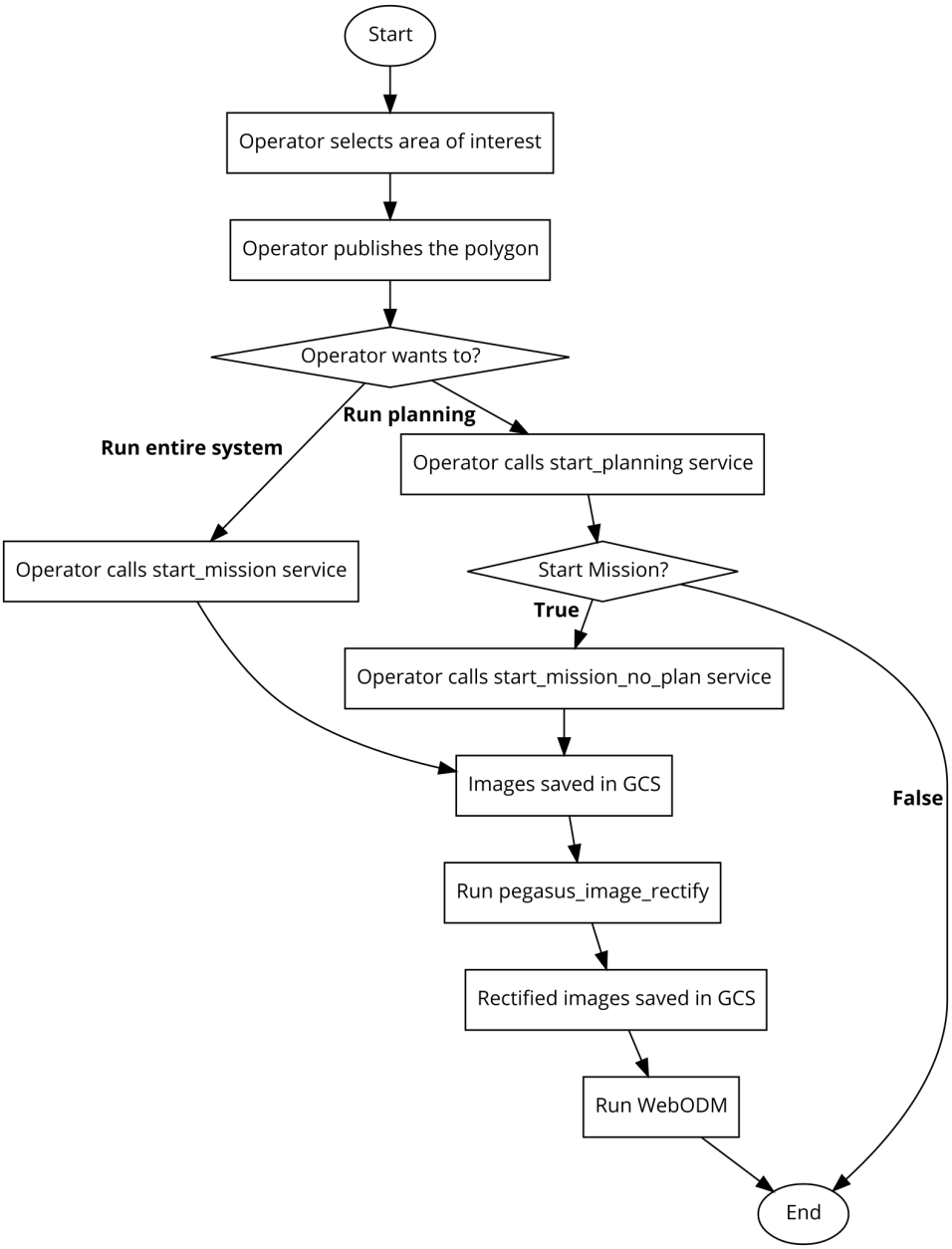
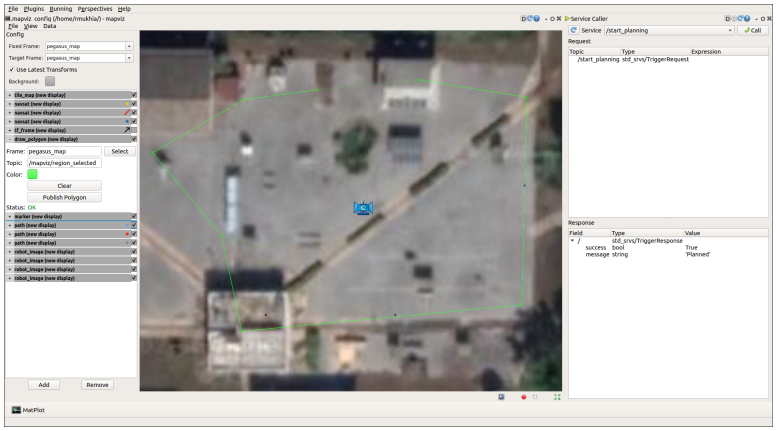


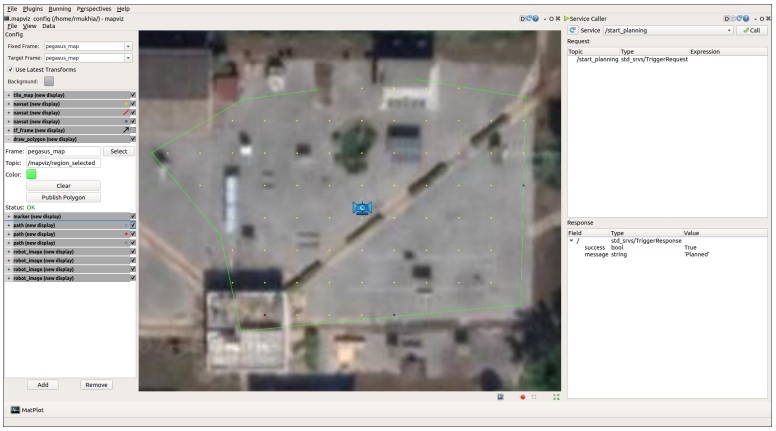
Figure 3.10

A scenario where an operator (a) Selects an area of interest (b) Publishes the polygon (c) Calls start_planning service.

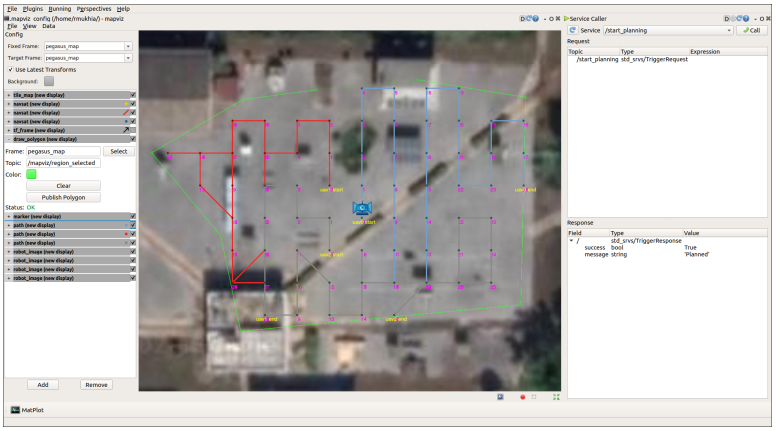
(a)



(b)



(c)



- *Multi-agent coverage path planning*: A* search to calculate the optimal paths for each drone, such that they avoid collisions and stay within the constraints of the wireless mesh network in the global map frame.

Viewpoint Generation

To generate a valid grid inside the region of interest, consider a polygon with n vertices:

$$\text{polygon} = \begin{bmatrix} x^0 & y^0 \\ x^1 & y^1 \\ x^3 & y^2 \\ \vdots & \vdots \\ x^{n-1} & y^{n-1} \end{bmatrix}.$$

The vertices are arranged counter-clockwise/clockwise. To find the bounding box of the polygon, we need to find the vertices.

$$\text{boundingBox} = \begin{bmatrix} x_{min} & y_{min} \\ x_{max} & y_{min} \\ x_{max} & y_{max} \\ x_{min} & y_{max} \end{bmatrix},$$

where each row defines a vertex. Consider a grid with n cells either horizontally or vertically. Let c be the grid cell size.

$$c = \text{BoundingBox}_{width|heigh} \div n$$

Alternatively, we can set a fixed size for each cell, or calculate the cell size from camera parameters and the altitude of the drone.

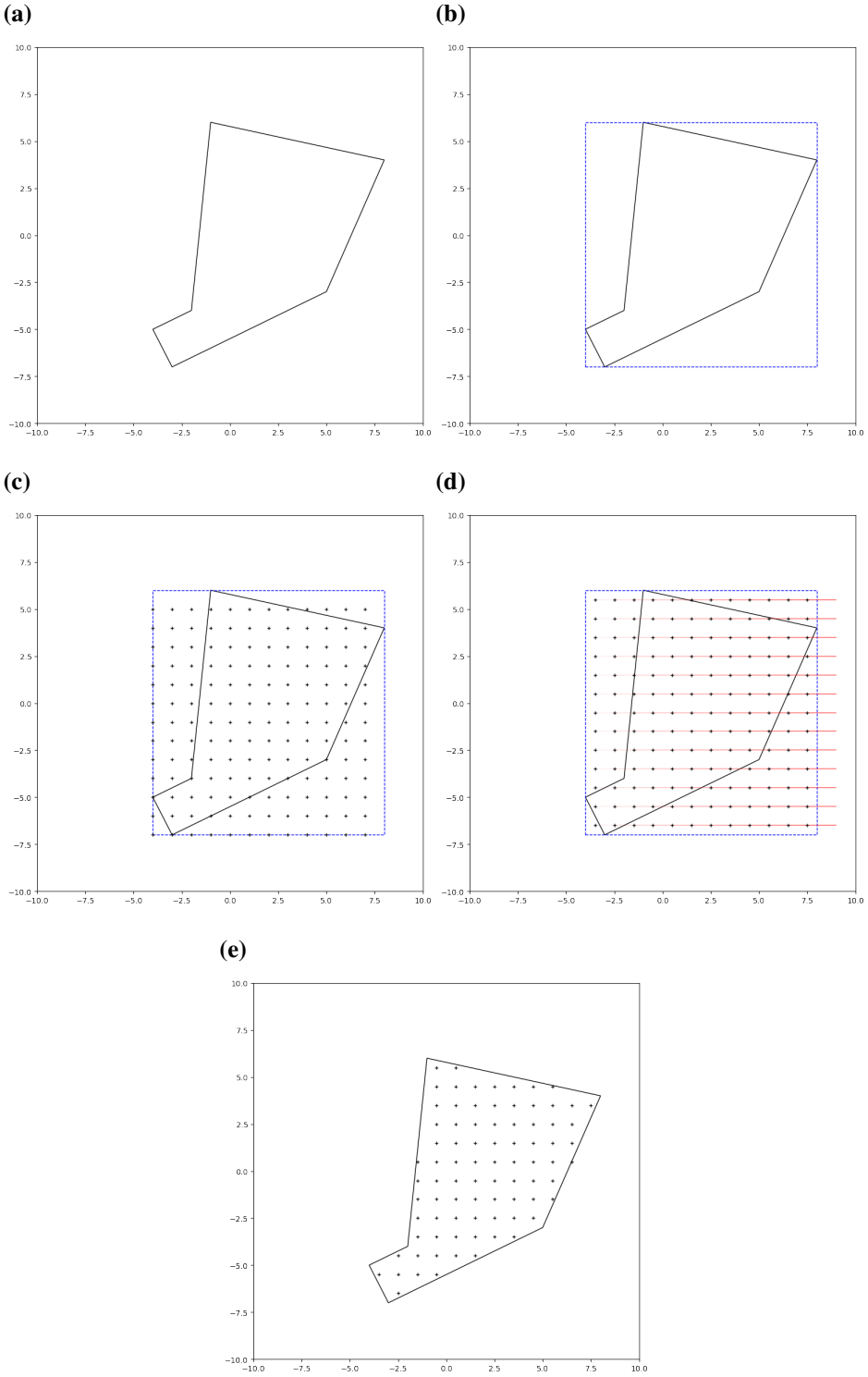
We can then calculate the coordinates $(b_x^{i,j}, b_y^{i,j})$ of the bottom left vertex of each cell.

$$\text{cells} = \begin{bmatrix} b_x^{0,0} & b_y^{0,0} \\ b_x^{1,0} & b_y^{1,0} \\ b_x^{2,0} & b_y^{2,0} \\ \vdots & \vdots \\ b_x^{i_{max}-1,0} & b_y^{i_{max}-1,0} \\ b_x^{0,1} & b_y^{0,1} \\ b_x^{1,1} & b_y^{1,1} \\ b_x^{2,1} & b_y^{2,1} \\ \vdots & \vdots \\ b_x^{i_{max}-1,0} & b_y^{i_{max}-1,0} \\ \vdots & \vdots \\ b_x^{i_{max}-1,j_{max}-1} & b_y^{i_{max}-1,j_{max}-1} \end{bmatrix}$$

$$b_x^{i,j} = x_{min} + i * c$$

Figure 3.11

The steps in viewpoint generation (a) Polygon showing the area of interest. (b) Bounding box on the Polygon showing the area of interest. (c) Bottom left position of the cells in the bounding box. (d) Rays projecting from cell centre to $x_{max} + 1$. (e) Grid cells inside the region of interest.



$$b_y^{i,j} = y_{min} + j * c$$

i, j are the index in x and y axis respectively.
We can get the range of i and j as follows.

$$i_{max} = \left\lceil \frac{x_{max} - x_{min}}{c} \right\rceil$$

$$j_{max} = \left\lceil \frac{y_{max} - y_{min}}{c} \right\rceil$$

Let the number of vertices in $gridCells$ be k:

$$k = i_{max} \times j_{max}.$$

We can then calculate grid index (i, j) from $[0, k]$:

$$j = \left\lfloor \frac{k}{i_{max}} \right\rfloor$$
$$i = k - j \times i_{max}.$$

Since the vertex v of the polygon is arranged counter-clockwise/clockwise, we can easily get the vertices of line segment for each side of the polygon.

$$v_i = (x_i, y_i)$$

$$s < n$$

$$\text{polygonLineSegments} = \begin{bmatrix} v_0 & v_1 \\ v_1 & v_2 \\ \vdots & \vdots \\ v_s & v_{s+1} \\ \vdots & \vdots \\ v_{s_{max}-1} & s_0 \end{bmatrix}$$

$$= \begin{bmatrix} x_0 & y_0 & x_1 & y_1 \\ x_1 & y_1 & x_2 & y_2 \\ \vdots & & \vdots & \\ x_s & y_s & x_{s+1} & y_{s+1} \\ \vdots & & \vdots & \\ x_{n-1} & y_{n-1} & x_0 & y_0 \end{bmatrix}$$

We can imagine that the center of each grid cell projects a line parallel to the x axis till $x_{max} + 1$.

$$\text{cellCenter} = [\text{cells}] + \frac{c}{2}$$

$$\text{rays} = \begin{bmatrix} \text{cellCenter}_x^{0,0} & \text{cellCenter}_y^{0,1} & x_{max}+1 & \text{cellCenter}_y^{0,1} \\ \text{cellCenter}_x^{1,0} & \text{cellCenter}_y^{1,1} & x_{max}+1 & \text{cellCenter}_y^{1,1} \\ \text{cellCenter}_x^{2,0} & \text{cellCenter}_y^{2,1} & x_{max}+1 & \text{cellCenter}_y^{2,1} \\ \vdots & & \vdots & \\ \text{cellCenter}_x^{k-1,0} & \text{cellCenter}_y^{k-1,1} & x_{max}+1 & \text{cellCenter}_y^{k-1,1} \end{bmatrix}$$

Now we apply ray tracing to check if $cell^{i,j}$ is inside our polygon or not. We count how many intersections each $rays^{i,j}$ makes with all the lines in $polygonLineSegments$. If the number of intersections are even, then the $cell$ is outside the polygon. If the number of intersections are odd, then the $cell$ is inside the polygon. We now have the valid cells that can the drones can traverse.

Multi-agent Coverage Path Planning

Considering a set of cells C for an area of interest with n agents. Multi-agent coverage path planning should compute coverage paths p_i for each ith agent where $p_i \subset C$ and $\bigcup_{i=1}^{n} p_i \subseteq C$. If $\bigcup_{i=1}^{n} p_i = C$, then complete coverage is obtained.

We use A* search to find p_i in C for each agent. Let the set of search space in the A* search space be denoted as S, which is a tree with root node s_0. For A* search the grid index is used to represent the position of the agents and cells, not the real world position.

A* search uses the cost function

$$f = g + h$$

Where g_k is the total cost till the current state s_k and h_k is the heuristic cost till the goal state s_{goal} from s_k. k is the id of nodes in the search tree.

There are nine directions an agent can move to:

1. Right,
2. Left,
3. Up,
4. Down,
5. Right up,
6. Left up,
7. Left down,
8. Right down,
9. Stay.

In our case in each state progression a single agent moves in one direction. For example, if there are three agents, in state s_l agent 1 may move left, in state s_{l+1} agent 2 may move right, in state s_{l+2} agent 3 may move down, in state s_{l+3} agent 1 may move top, and so on.

Let us define g_k and h_k for our planner. m_k^c is the movement cost accumulated in cell $c \in C$ at state s_k.

For right, left, up and down movement:

$$m_k^c = 1, \text{if agent reaches an unvisited cell.}$$

$$m_k^c = m_{k-1}^c \times 5, \text{if agent reaches a visited cell.}$$

For diagonal movements and stay:

$$m_k^c = 1.44, \text{if agent reaches an unvisited cell.}$$

$$m_k^c = 1.44 \cdot (5 \cdot m_{k-1}^c), \text{if agent reaches a visited cell.}$$

Therefore, the total movement cost at state s_k is $m_k = \sum^{C} m_k^c$.

$$g_k = m_k$$

$$h_k = h_k^{free} + h_k^{dist}$$

h_k^{free} is the number of unvisited cells at state s_k, which can be defined as $|\{c \in C | m_k^c = 0\}|$. h_k^{dist} is the minimum euclidean distance between any agents' position and the closest unvisited cell at state s_k.

Therefore, a state s_k will have:

- m_k^c: The immediate movements to reach s_k.
- g_k: All movement cost for all agents to reach state s_k.
- h_k: Number of unvisited cell and minimum euclidean distance to the closest unvisited cell at s_k .
- O_k: Set of cells that the agents are occupying in the state s_k.
- K_k: Cells with recorded movement costs. To compute h_k for the state.

$$s_k = \{m_k^c, g_k, h_k, O_k, K_k\}$$

$E_k \subset K_k$, are the cells which have not been visited at s_k. We must define equality operation for the state for A* search to work.

$$s_\sigma = s_\theta, \iff O_\sigma = O_\theta \wedge E_{k_\sigma} = E_{k_\theta}$$

The other parameters we need to set for our A* search algorithm are:

- *MAX_DISTANCE*: The real world maximum range of mesh client.
- *CS_POSITION*: The real world position of control station in global map.
- *Z_HEIGHT*: The real world operational height of agents.

The constraints for A* search are:

- At least one agent must move.
- The euclidean distance between the real world positions of any two pair of agents should not be greater than MAX_DISTANCE.
- At least one agent should have euclidean distance with the real world position of CS_POSITION less than or equal to MAX_DISTANCE.
- Any two agents cannot swap position.

$$\begin{matrix} A & B \\ 0 & 0 \end{matrix} \quad \not\rightarrow \quad \begin{matrix} B & A \\ 0 & 0 \end{matrix}$$

- Any two agents path line equation must not intersect between the previous cell and present cell.

$$\begin{matrix} A & 0 \\ B & 0 \end{matrix} \quad \not\rightarrow \quad \begin{matrix} 0 & B \\ 0 & A \end{matrix}$$

Running A* using these parameters computes paths p_i, with $\bigcup_{i=1}^{n} p_i = C$. However, it is not efficient as A* is not an approximation algorithm but an exhaustive search algorithm, and the problem we are trying to solve is NP-hard.

Branching factor for each node of our search tree is nine. However, the depth of the search tree increases as we add more agents. Therefore, the number of cells and agents drastically increases our running time. To decrease this, we will not compute the optimal steps to the end step, but we will look ahead certain number of steps, and progressively step ahead. This way we may not get the optimal solution, but the computing time will be saved.

We will progressively reach our goal state through iterations of A* search algorithm. For each A* iteration, we will stop the search by:

- *Depth exit*: The depth exit mechanism will return d number of states, that is it will travel $d+1$ depth in the search space and return the optimum D_q path.
- *Early exit*: If we do not minimize the cost f for ε number of nodes, we will do early exit and return D_q, the path with the best heuristics encountered.
- *Increasing depth exit*: In the beginning of the search, none of the cells are visited. Therefore small depth exit values (≥ 2) is used in the beginning to cover the cells, and as more cells are visited, increase the depth exit value proportionally ($\leq d$). This makes the algorithm greedy at the beginning of the search and as the states before more complex, it starts taking more intelligent decisions. This is done by using the curve,

$$y = x^c$$
$$x = \frac{|K_k - E_k|}{|K_k|}.$$

$c < 1$ such that the the depth exit value rises fast to d.

In this algorithm the agents will calculated q number of D paths iteratively. If the optimal path is the search space is o, the path returned by the planner will be sub-optimal.

$$cost(o) \leq \sum_{i=0}^{q} cost(D_i)$$

Also, if the heuristics does not improve for σ A* iterations, then the search will stop. In this case, all the cells will not be covered.

$$if, h(D_i) = h(D_{i+1}), \forall 0 \leq i < \sigma - 1; \text{stop}$$

Then the paths D_q are concatenated to form a single path. The resulting path may have trailing states which have the same h. This is inconvenient and does not contribute to the solution, so the trailing states with the same h are pruned.

A single agent needs to explore all the cells hence it does not utilize the heuristics given for multiple agents. For single agent path planning, the cost function is modified to $f_k = g_k$. This cost function changes A* search into Dijkstra algorithm and explores all states.

After the A* search is complete and the paths are concatenated and pruned, the movement of each agent is separated into different lists, from the final path. Then the paths for the agents are built by referring to movement list of the agents and the position of the cells in the global map. The yaw of the agents in the paths are calculated with the formula $\alpha = \tan^{-1}\left(\frac{y_2 - y_1}{x_2 - x_1}\right)$. The height of the poses in the path is set to Z_HEIGHT. The paths are published to the ROS topics `/pegasus/[agent]/path`.

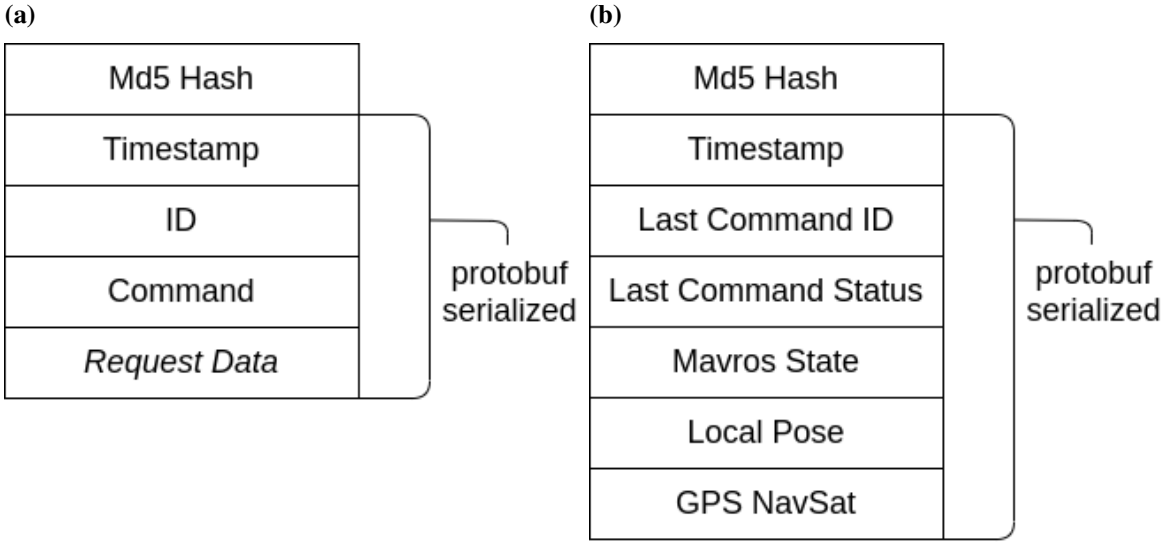
3.3.3 Motion Control

This layer handles the control of drones through the ground control station. It has the components, pegasus_controller in the GCS, and pegasus_commander, mavros and PX4 in the drone. A single pegasus_controller sends requests to each drone's pegasus_commander. The structure of response and reply messages are illustrated in Figure 3.12. The different control messages that is received by pegasus_commander from pegasus_controller are:

- *HEART_BEAT*: Heart beat in regular intervals. This message is the only type to which pegasus_commander responds.
- *SET_OFFBOARD*: Put the flight controller in off-board mode.
- *SET_RETURN_TO_HOME*: Put the flight controller in return mode.
- *SET_ARM*: Arm the flight controller.
- *GOTO(pose)*: Move to drone to pose position.

HEART_BEAT has command id 0. SET_OFFBOARD, SET_RETURN_TO_HOME, SET_ARM, and GOTO have incremental id. That is for every new command, the id is incremented by one. HEART_BEAT reply has the last command id received by pegasus_commander and the status of its execution. Last command status is set to true when the command has been ex-

Figure 3.12
Structure of control messages between `pegasus_controller` *and* `pegasus_commander` *(a) Request Message (b) Response message.*



ecuted. Figure 3.13 illustrates the communication between one pegasus_controller and two pegasus_commander.

3.3.3.1 Pegasus_controller Pegasus_controller is responsible for:

- Calculating and maintaining the transformations between the local map frames of each drone and the global map frame.
- Coordinating and sending control messages to the drones.
- Monitoring the link with the drones by sending heart beat control message.
- Publishing local pose, global GPS coordinates and state of each drone, received in the heart beat reply as ROS topics.
- Calling image acquisition service to acquire images from the drones.

The architecture of pegasus_controller is given in Figure 3.14.

The state diagram in Figure 3.15 can be further elaborated as follows:

- *IDLE*: Do nothing.
- *PLAN*: Call services provided by pegasus_planner.
- *PREP*: Prepare the drone to enter off-board mode.
- *OFFBOARD_MODE*: Send SET_OFFBOARD control message to the drones and wait for confirmation.
- *ARMING*: Send SET_ARM control message to the drones and wait for confirmation.
- *TAKE_OFF*: Takeoff to operational height.
- *CALIBRATE*: Send the calibration path to the drones and coordinate the flight of the drone along the calibration path and capture the corresponding local poses and

Figure 3.13
GOTO command communication diagram between `pegasus_controller` *and two* `pegasus_commanders`.

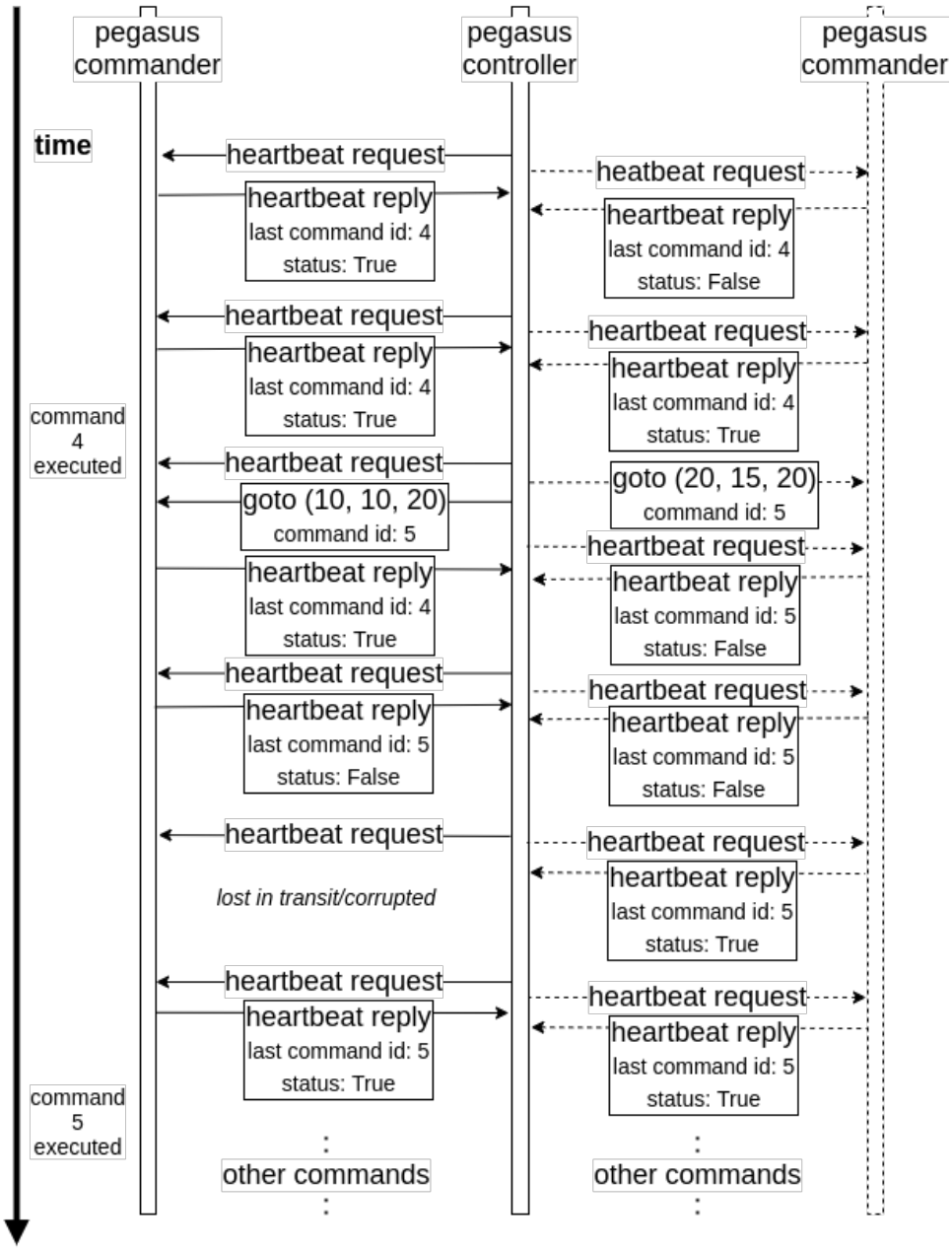


Figure 3.14
Architecture of `pegasus_controller`.

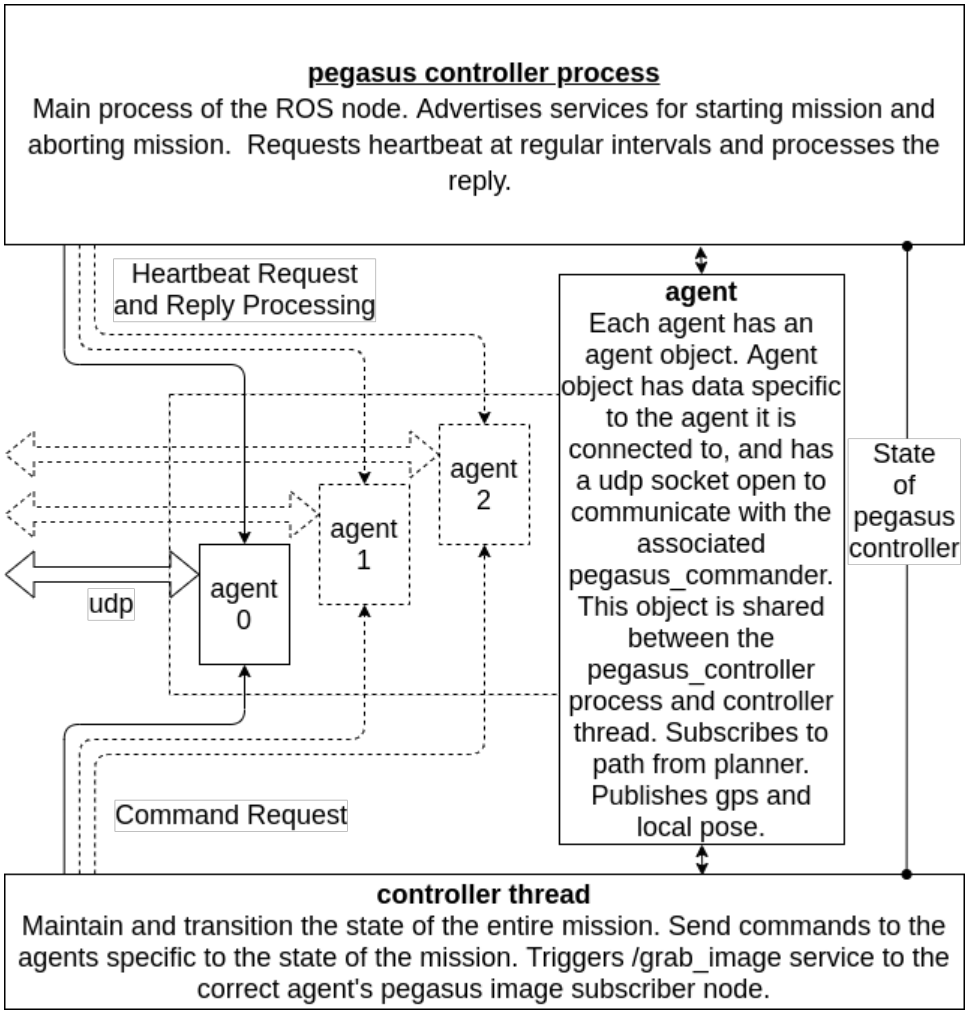


Figure 3.15
State diagram of `pegasus_controller`.

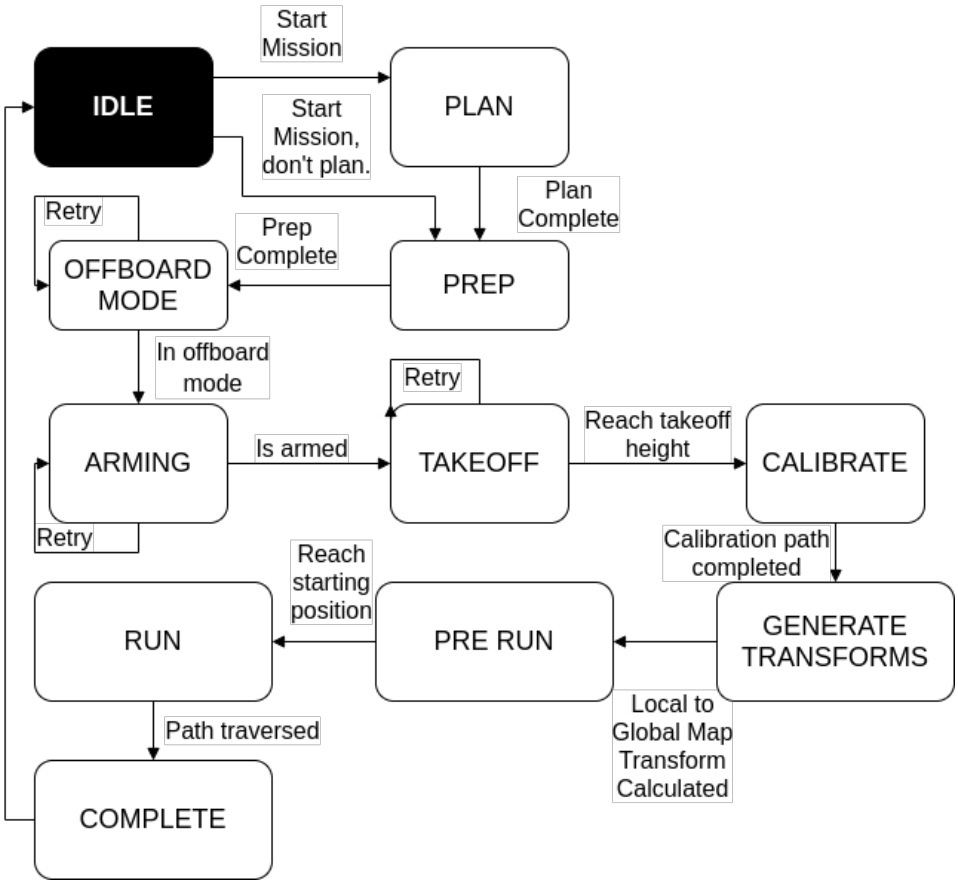
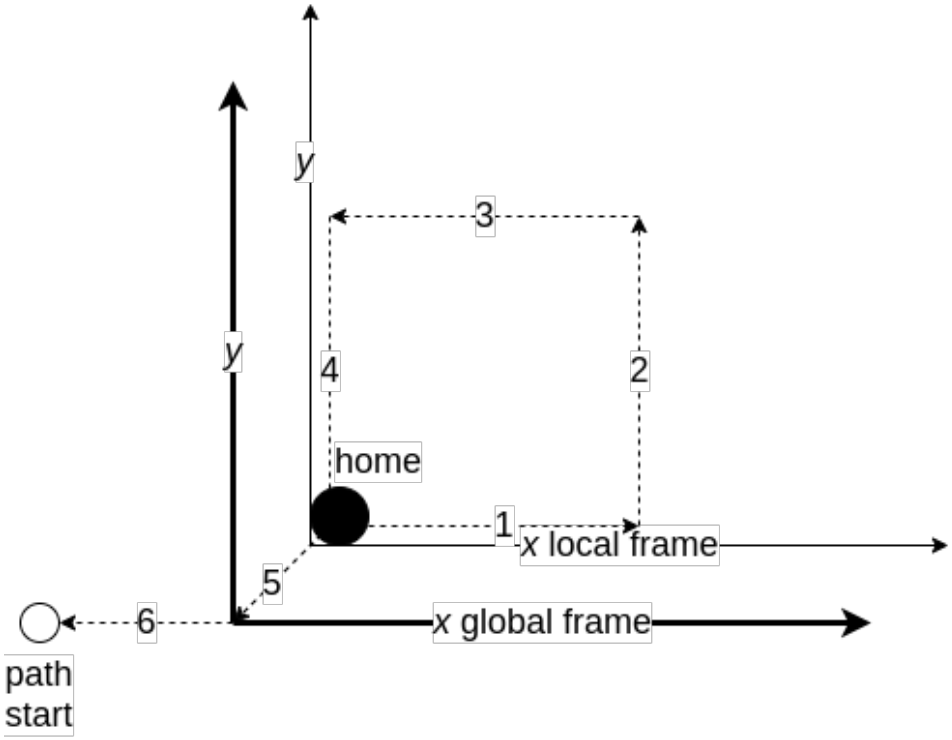


Figure 3.16
Predefined routine before the start of the coordinated path traversal. Steps 1-4 will be executed in CALIBRATE state, step 5 will be executed in PRE_RUN state, and in step 6 the drone will move to the first pose in the planned path.



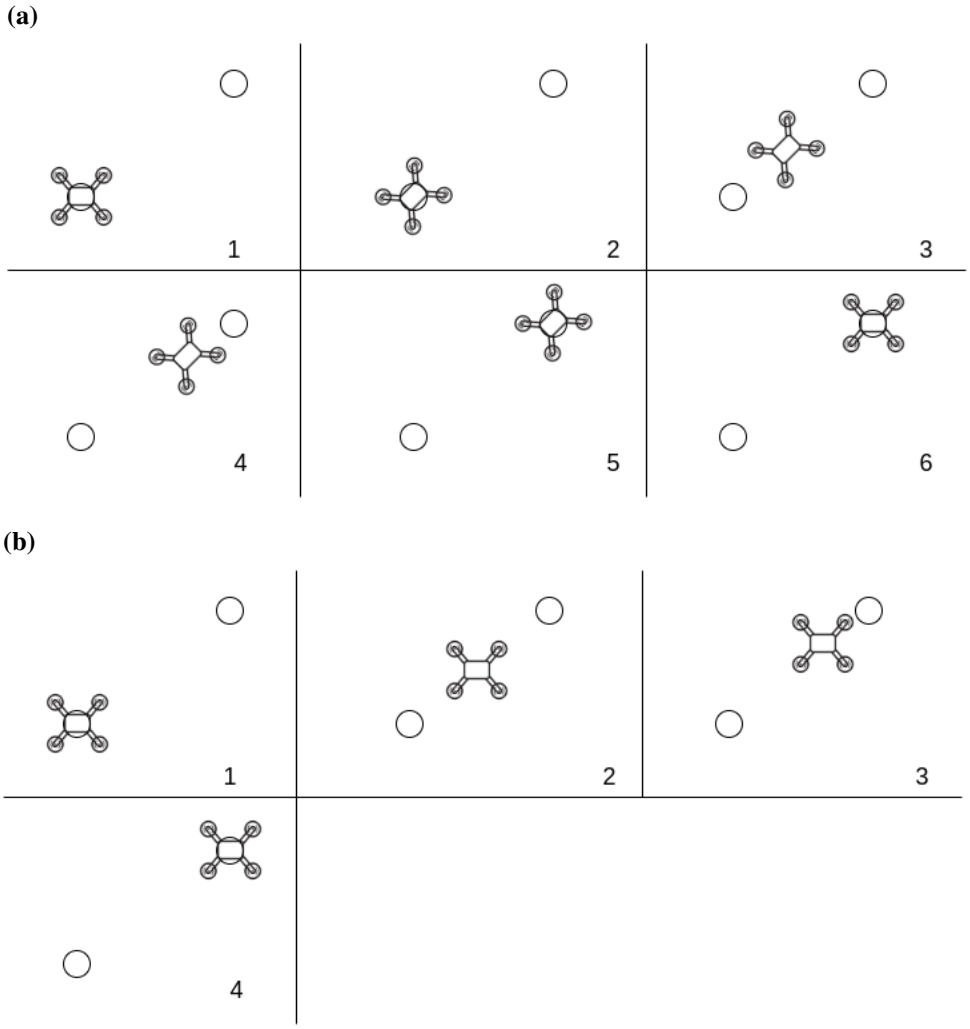
global GPS positions for calculation the map transformations.

- *GENERATE_TRANSFORMS*: Generate transforms between the local map frame of each drone and the global map frame using the corresponding points collected in CALIBRATE state.
- *PRE_RUN*: Move the drones to their starting positions.
- *RUN*: Execute path of the drones in a coordinated manner.
- *COMPLETE*: It will reach this state when the system's execution is complete. Set the drone to Return mode by sending SET_RETURN_TO_HOME control message.

The drone takes off in its home position. After it reaches its operational height, it follows a predefined routine from CALIBRATE state to PRE_RUN state as shown in Figure 3.16. All the drones are positioned at the origin of the global map (pegasus_map) at different altitudes in PRE_RUN state. For a mission with operational height z_o and n drones, the i^{th} drone, were $0 <= i < n$ has $z = z_o + 2 \cdot i$ meters altitude in PRE_RUN state.

In GENERATE_TRANSFORM state , algorithm 1 is used to find transforms between local map of each drone and the global map. The problem of finding transforms between all the local and global maps can be done with planar homography because the z axis of all the maps are parallel and pointing upwards.

Figure 3.17
Drones can be configured to follow either (a) Towards movement, or (b) Strafe movement. Towards movement takes more time than strafe movement because it has to know whether it has reached the destination pose in step 5, and if it has properly aligned to take image in step 6. In strafe movement the drone is already aligned to take image and it just needs to know wheather it has reached the destination pose in step 4.



In RUN state,pegasus_commander is configurable to allow two types of movement:

- *Towards*: The drones will face toward the direction they are moving. During image acquisition the drones orients themselves by setting their yaw to 0°, as shown in Figure 3.17a.
- *Strafe*: The drones will always have their yaw at 0°. They move sideways to the next viewpoint. This movement requires less time when the viewpoints are separated by small distances. Figure 3.17b shows this movement.

Image acquisition is skipped if the drone has already visited the viewpoint to avoid duplicate images.

In COMPLETE state, the drones returns to their home position following the behavior de-

Algorithm 1 Find Local To Global Transformation.

Input: L: set of all corresponding local poses
Input: G: set of all corresponding global GPS positions
Input: O: global map origin GPS coordinate
Output: T: transformation between local and global map

$\widetilde{O} \leftarrow$ CONVERT-TO-UTM(O)
$\widetilde{G} \leftarrow$ CONVERT-TO-UTM(G)
$M \leftarrow \widetilde{G} - \widetilde{O}$
$\widehat{M} \leftarrow$ REMOVE-Z-AXIS$(\widetilde{M})$
$\widehat{L} \leftarrow$ REMOVE-Z-AXIS(L)
$H \leftarrow$ FIND-HOMOGRAPHY$(\widehat{L}, \widehat{M}, \text{RANSAC})$

$$T \leftarrow \begin{bmatrix} H_{1,1} & H_{2,1} & 0 & H_{3,1} \\ H_{1,2} & H_{2,2} & 0 & H_{3,2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

fined in PX4 firmware. PX4 firmware ascends the drone to a safe altitude before returning them to home. To avoid collision, the drones' safe altitude should be set different from each other.

3.3.3.2 Pegasus_commander Pegasus_commander is a new ROS node customized for this study that uses the topics and services provided by mavros to control the drone in off-board mode. It receives control messages from the pegasus_controller. It also monitors the link between the GCS and the drone, and if a HEART_BEAT control message is not received for a particular duration, then the flight controller will be set to return mode. The architecture of this module is shown in Figure 3.18. It responds to HEART_BEAT control messages with:

- The last command id it received and the status of the command,
- State of mavros,
- Local Pose of the drone in the local map,
- GPS coordinates of the drone.

For SET_ARM, SET_OFFBOARD and SET_RETURN_TO_HOME it calls the relevant mavros services. For GOTO, it publishes the pose it received to mavros until it reaches the desired pose. Algorithm 2 is used to determine if the drone has reached the desired pose.

3.3.3.3 PX4 The drone runs PX4 Firmware on the flight controller. The flight controller's movement and altitude can be controlled in off-board mode through the MAVlink protocol. The MAVLink commands supported in this mode are.

- SET_POSITION_TARGET_LOCAL_NED: Control vehicle position.
- SET_ATTITUDE_TARGET: Control vehicle altitude and orientation.

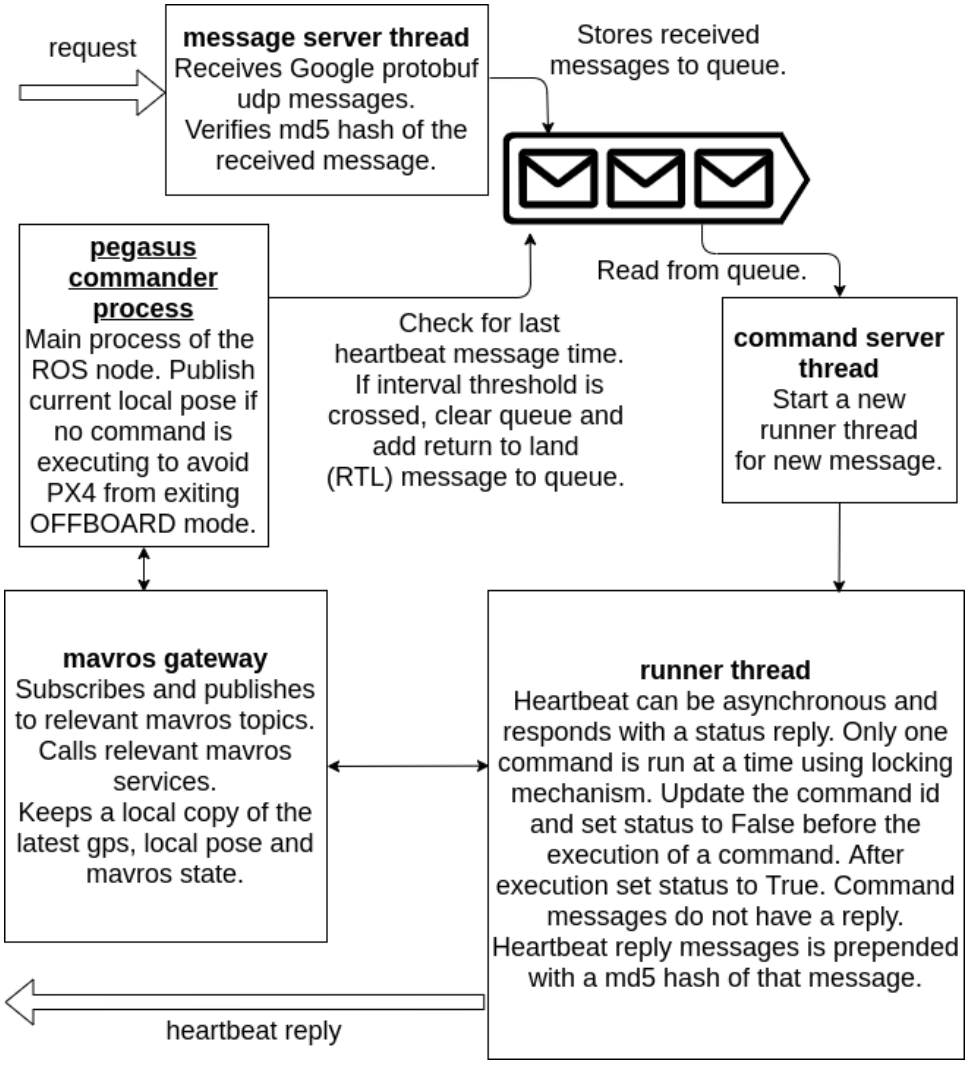
PX4 supports the coordinate frames:

Algorithm 2 Is at position.

Input: P_e: expected pose
Input: P_a: actual pose
Input: d: offset distance
Input: y: offset yaw
Output: A: Boolean True or False

$D \leftarrow |\text{EUCLIDEAN\_DISTANCE}(P_e.position, P_a.position)|$
$Y_e \leftarrow \text{GET\_YAW\_FROM\_QUATERNION}(P_e.orientation)$
$Y_a \leftarrow \text{GET\_YAW\_FROM\_QUATERNION}(P_a.orientation)$
$Y \leftarrow |Y_e - Y_a|$
$A \leftarrow (D < d) \wedge (Y < y)$

Figure 3.18
Architecture diagram of `pegasus_commander`.



- MAV_FRAME_BODY_NED, and
- MAV_FRAME_LOCAL_NED.

Before engaging the offboard mode a stream of pose must be receive by the flight controller. To keep the flight controller in off-board mode, poses must be publish at the rate of 2Hz or more otherwise it will exit from off-board mode. If it loses communication with an companion computer or GCS in off-board mode, after a timeout (COM_OF_LOSS_T) the vehicle will attempt to land or perform some other failsafe action. The action is defined in the parameters COM_OBL_ACT and COM_OBL_RC_ACT. These parameters can be set through mavros' mavparam.

3.3.3.4 Mavros Mavros is a ROS node that provides ROS interfaces for communication with various autopilots with MAVLink communication protocol. The ROS services provided by Mavros, of interest for this system are:

- *mavros/set_mode*: Required to set mode of the flight controller to off-board mode and return to launch mode.
- *mavros/cmd/arming*: Required to arm the flight controller.

The ROS topics published by Mavros that are useful for this system are:

- *mavros/state*: Publishes the state of the flight controller.
- *mavros/local_position/pose*: Publishes pose, i.e., position and orientation of the flight controller in the local frame. Mavros handles the translation between NED and ENU conventions.
- *mavros/global_position/global*: Publishes the GPS coordinates, i.e., latitude, longitude and altitude of the flight controller.

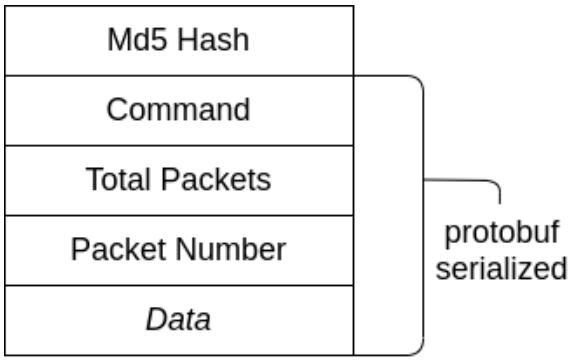
The ROS topics subscribed by Mavros, that this system will use are:

- *mavros/setpoint_position/local.* Set the pose, i.e. position and orientation that the flight controller is desired to achieve in local frame.

3.3.4 Image Acquisition

This layer provides image acquisition service. The GCS is able to request geo-tagged image from the drone in flight, using image acquisition service. The software components in this layer are gscam and pegasus_image_publisher in the drones and pegasus_image_subscriber in the GCS. The gscam ROS node is used to publish camera video feed as ROS camera topic using gstreamer. Mavros is used to acquire GPS location of the image captured. The pegasus_image_subscriber provides a ROS service in the GCS which is called by pegasus_controller to acquire images at the viewpoints. The pegasus_image_subscriber is a udp client that requests image from the pegasus_image_publisher udp server in the drone. For multiple drones there are multiple pegasus_image_subscriber running under different ROS namespace in the GCS. The subscriber and the publisher transfers data with each other using

Figure 3.19
Packet structure for image acquisition.



the packet as shown in Figure 3.19.

The same packet structure is used by both the subscriber and publisher, however the commands are different:

- *REQUEST*: REQUEST is sent from subscriber to the publisher. The publisher drops the previous buffer, captures a new image, adds GPS tag to it, segments the image into multiple packets and stores the data in buffer.
- *REPLY*: The publisher responds to the subscriber's REQUEST packet. REPLY has the total packets in the buffer for the image requested.
- *PKT*: The subscriber sends PKT request for image data from the publisher using the packet number it wants. It requests the same packet number until it receives the packet and then increments the packet number. The publisher fills the data section with the image data and replies with the PKT message. The request is made till the packet number reaches the total packets received in REPLY.
- *ERR*: If the publisher cannot find the packet number the subscriber is requesting for, it responds with ERR message.

The communication diagram is illustrated in Figure 3.20.

3.3.4.1 Pegasus_image_publisher The gscam ROS node provides the USB camera video feed as a ROS topic. The pegasus_image_publisher node subscribes to the ROS camera topic and GPS topic published by mavros. When it receives a REQUEST message from the pegasus_image_subscriber, it gets the current video frame and gps data, and stores GPS data using Exchangeable Image File (EXIF) format. It stores the geo-tagged image in buffer till a new REQUEST message is received. It provides image data to the subscriber from the current buffer.

3.3.4.2 Pegasus_image_subscriber The pegasus_image_subscriber provides image acquisition service to the GCS. It exposes ROS service `/[drone_namespace]/grab_image`. For multiple drones there should be multiple pegasus_image_subscriber running under different ROS namespace in the GCS. It receives the image and store it in the local directory

Figure 3.20
Communication diagram for image acquisition.

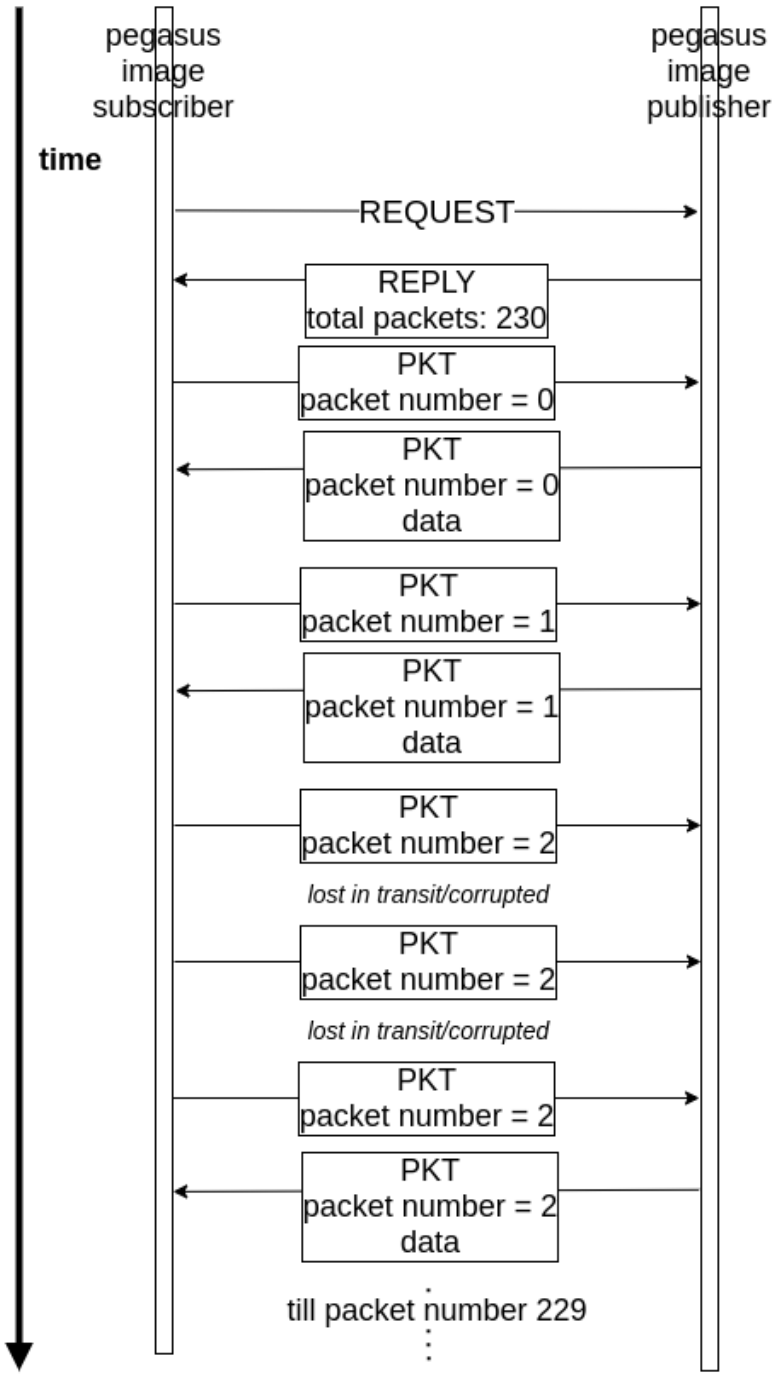
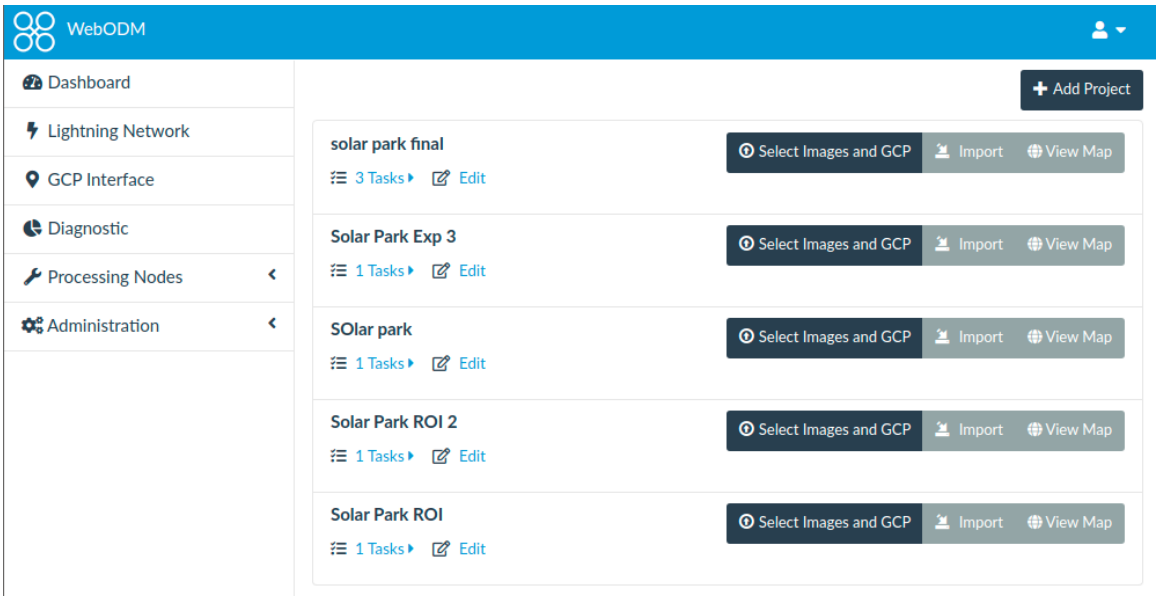


Figure 3.21
WebODM application for generating map from images.



under the name `[drone-namespace]-i.jpeg`, where `i` is the image number.

3.3.5 Map Building

This layer builds the map from the images captured during the mission. This is run after the drones have finished their mission. This layer has two components, pegasus_image_rectify and WebODM.

3.3.5.1 Pegasus_image_rectify Pegasus_image_rectify uses the camera calibration file generated by camera_calibration node available in image_pipeline ROS package to rectify the images received from the drones. The images capture from the drones follows the naming convention of `[drone-namespace]-i.jpeg`. The pegasus_image_rectify processes the images for the associated camera calibration file by identifying the images using the drone-namespace. It also provides a feature to select region of interest to crop the images received from the drones. The rectified images are saved with the same filename in another output directory in the GCS.

3.3.5.2 WebODM WebODM provides a web based GUI to use Open Drone Map (ODM). The operator uploads the rectified images to WebODM and generate orthophoto, 3D point cloud, and textured mesh among other features provides by ODM.

3.3.6 Pegasus_fov

Pegasus_fov is a helper node to assist the operator in selecting a good grid size for the viewpoints. It calculates the area covered by an image given the operational height of the drone and the camera calibration file. The camera calibration file has the following fields:

- w: Image width.
- h: Image height.

- K: Camera matrix, where

$$K = \begin{bmatrix} f_x & s & x_0 \\ 0 & f_y & y_0 \\ 0 & 0 & 1 \end{bmatrix}.$$

f_x and f_y are x and y focal length. (x_0, y_0) is the principal point offset. s is the axis skew.
To find the field of view angles,

$$\theta_x = 2 \times \tan^{-1}(\frac{w}{2 \times f_x})$$

$$\theta_y = 2 \times \tan^{-1}(\frac{h}{2 \times f_y}).$$

From θ_x and θ_y, the area of coverage per viewpoint from altitute z can be calculated.

$$x = 2 \times \tan(\frac{\theta_x}{2}) \times z$$
$$y = 2 \times \tan(\frac{\theta_y}{2}) \times z$$

(x, y) is the area captured by the camera at altitude z. The operator can use this information to choose the size of the grids in path planning.
This node is accessible using rosrun with parameters _camera_file and _height as follows:

```
$ rosrun pegasus_uav_ros pegasus_fov.py
    _camera_file:=$(pwd)/src/pegasus_uav_ros/calibration/mobius.yaml
    _height:=20
```

3.4 Chapter Summary

This chapter documents the design of the proposed system. I have described multiple components working together in this system. I also descibed the components on the drone and the GCS.

CHAPTER 4
EXPERIMENTAL RESULTS

In this chapter, two experiments comprising simulations with three drones and a real world experiment with one drone are described.

4.1 Real World Experiment

To test the automated system, an experiment was run in the Energy Park behind CSIM building. The site was chosen as it has structures which will help SURF descriptors recognize features.

4.1.1 Mesh Network

The mesh network consists of two wireless mesh routers running OpenWrt. Both of the routers are dual channel 2.5/5 GHz's routers with two radios. The GCS wireless router is a Netgear router (Figure 4.3a). The GCS laptop is connected to the router via 2.5GHz WiFi network with SSID UAV01_01. The TP-Link mobile wireless router (Figure 4.3b) mounted on the drone is connected to the companion Raspberry Pi via Ethernet. The SSID of the mesh network is UAV01_MESH5G and is running on 5 GHz channel. Figure 4.2 shows the network infrastructure used.

The GCS laptop and companion are not mesh clients but are connected to different networks. Route between the GCS laptop and companion Raspberry Pi is achieved by using OSLR Host and network association (HNA) message to allow connection to these networks. Since GCS laptop is connected to different subnet than the Raspberry Pi, a route `$ sudo route add -net 192.168.41.0/24 gw 192.168.40.1 wlp0s20f3` has to be added manually in the GCS laptop to divert the packet for the Raspberry Pi's subnet via the wireless LAN interface of the GCS to the the mesh network.

The IP assignment for different nodes in the network is given in Table 4.1. The OSLR messages' interval and validity for the GCS wireless mesh router and the mobile wireless mesh router is given in Table 4.2.

4.1.2 Automated Planning

For planning the path, the operational altitude was set at 20 meters, the grid size was set at 10 meters and the mesh range was set to 40 meters. The approximate area of coverage was 35 by 40 meters. The results of planning is given in Table 4.3. The viewpoints and the path generated is shown is Figure 4.4. The optimum length of the path for the viewpoints is sixteen, but the planner provided a path of eighteen viewpoints which resulted in the drone visiting two viewpoints more than once. The length of the path was 178.2 meters.

Figure 4.1
The drone used in real world experiment. (a) Top view. (b) Side View. The weight of the drone is 1.8 kilogram.

(a) **(b)**

Figure 4.2
Network infrastructure for real world experiment.

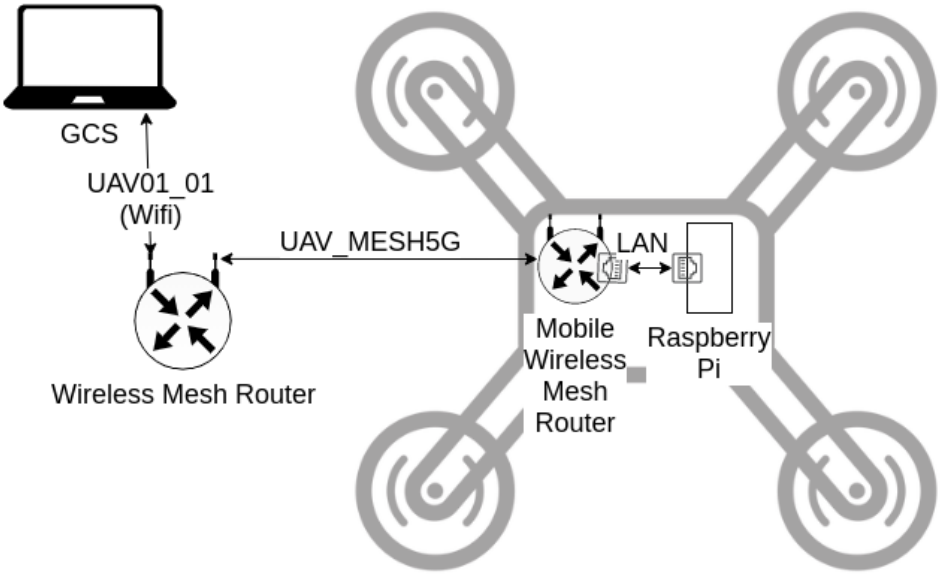


Figure 4.3
Routers used for the experiment. (a) Netgear wireless router for the GCS. (b) TP-Link mobile wireless router mounted on the drone.

(a) **(b)**

Figure 4.4
Path generated for real world experiment.

Table 4.1
IP assignment for real world experiment. Laptop and Raspberry Pi have local network addresses assigned by the respective routers they are connected to, but all communication is through the mesh network with the routers as gateways.

Node	IP Address	Subnet	OSLR Interface IP
GCS Wireless Router	192.168.40.1	192.168.40.1/24	192.168.30.1
Mobile Wireless Router	192.168.41.1	192.168.41.1/24	192.168.30.2
GCS Laptop	192.168.40.216	192.168.40.1/24	-
Companion Raspberry Pi	192.168.41.200	192.168.41.1/24	-

Table 4.2
OLSR messages' interval and validity parameters for real world experiment.

Message	Interval	Validity Time
Hello	5 sec	40 sec
TC	2 sec	256 sec
MID	18 sec	324 sec
HNA	18 sec	108 sec

4.1.3 Motion Control

The pegasus_controller was able to successfully communicate with the pegasus_com-mander in the drone and control the motion of the drone in off-board mode throughout the experiment. The movement type was set to strafe to reduce the time taken for the mission as the drone would not need to align itself at yaw 0°after reaching every viewpoint. The pegasus_controller was successfully able to calculate the transform between the local map of the drone and the global map after the calibration routine. To ensure the safety of the drone, the system was started with the drone in HOLD mode at 20 meters altitude. There was a slight drift in the position of the drone when it was being set to off-board mode through the companion computer. The actual path of the drone during the mission as reported by GPS data received through the heart beat reply is shown in Figure 4.5. The figure does not show drone returning to its home position after the end of the mission because heart beat message is stopped after the drone has reached and collected image in the last pose of its path. Figure 4.6 shows the complete GPS path in QGroundControl in a mobile smartphone connected through ESP32 wifi telemetry module attached to the flight controller. The difference between Figure 4.5 and 4.6 is that first figure is being reported through the Pegasus heart beat mechanism with an interval of one seconds, while the latter is being published directly by the flight controller.

Figure 4.5
GPS position reported through Pegasus system.

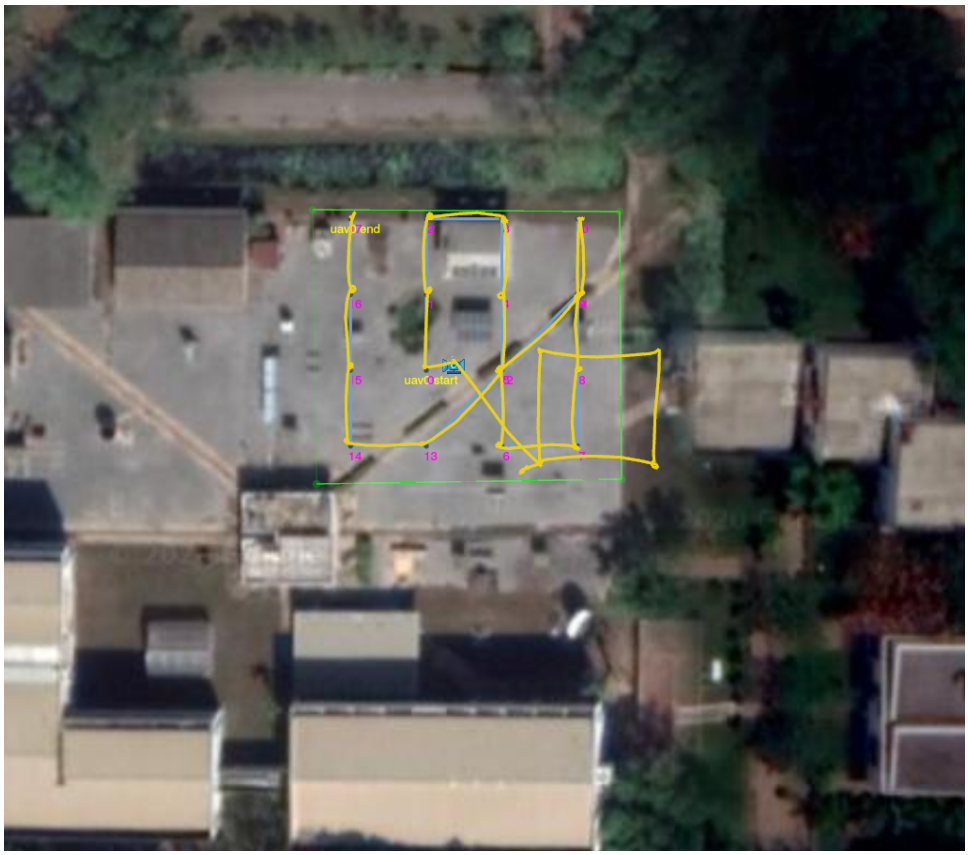


Figure 4.6
GPS position reported through telemetry WiFi in QGroundControl.

Table 4.3
Result of `pegasus_planner` in real world experiment.

Type	Number of Drones	Viewpoints Generated	Path Size
Real Experiment	1	16	18

Figure 4.7
Images captured by Pegasus in real-time.

4.1.4 Result

The pegasus_image_subscriber and pegasus_image_publisher successfully acquired 16 images from 16 viewpoints of the mission in real time as shown in Figure 4.7. Using an application called photini, the location of the images can be seen in Figure 4.8. The GPS EXIF tag from the images show correct viewpoint position.

The images captured is rectified using pegasus_image_rectify using the camera calibration file. Then the rectified images are processed using WebODM.

- Figure 4.9 shows the generated orthophoto of the region of interest. The orthophoto is skewed from the actual satellite image of the area of interest. It can be seen that the area covered is consistent with the GPS location of the image, but the position of the landmarks are skewed and the image appears smaller than the actual area. Due to the wide field of view of the camera used in the experiment, the image has

Figure 4.8
Location of images captured in real world experiment.

fish eye distortion. The rectification procedure is not enough to make the images undistorted. It can be seen that in the orthophoto, straight lines are not conserved.

- Figure 4.10 shows the point cloud for the region of interest. The point cloud is sparse and does not have good definition.
- Figure 4.11 shows the generated textured mesh of the region of interest. The outside edges are tapering outwards, because of fish eye distortion of the camera used to capture the image and less less overlapping images in the edges of the area of interest.
- Figure 4.12 shows the camera positions determined by ODM over the region of interest. The placement of a few cameras are different from the actual position. The SfM pipeline is not able to properly localize the camera poses due to distortion of the camera in use.

4.1.5 Summary of Real World Experiment

The system executes the presentation layer, planning layer, motion control layer and image acquisition layer as expected using wireless mesh network. The map building layer did not provide good quality orthophoto, 3D point cloud and textured mesh. The good quality results of the simulation where the camera provides images without distortion may point to the fish eye camera mounted on the drone being inadequate for our experiment.

4.2 Simulation

Simulation is run on the Gazebo world shown in Figure 4.13. The simulation experiment uses three drones.

4.2.1 Simulated Network Configuration

All the network packets was routed though ns-3. The OLSR messages' interval for the simulated network is given in Table 4.4. Ns-3 was configured to have one GCS node and three drone nodes. Therefore the GCS had a total of six udp sockets to control the system of three drones. The pegasus-net-sim ns-3 simulator was configured as follows:

```
std::map<std::string, std::vector<PegasusPortConfig>>
PegasusConfig::m_config {
  {
    "iris_0", {
      {5444, 4444, 3444},
      {7300, 7400, 7200},
     }
  },
  {
```

Figure 4.9
Orthophoto of Energy Park overlay on top of Google map.

Figure 4.10
Point cloud of Energy Park.

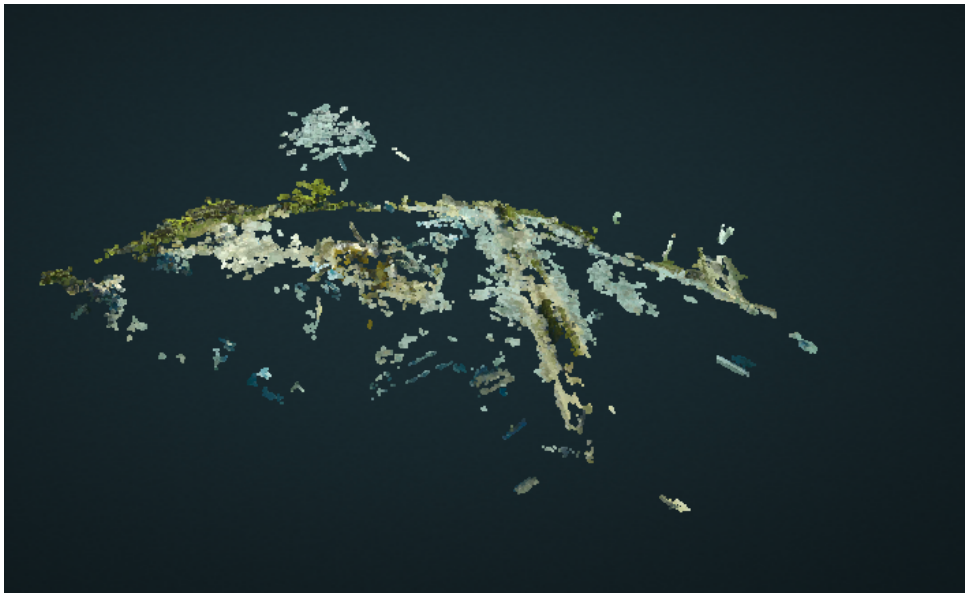


Figure 4.11
Textured mesh of Energy Park.

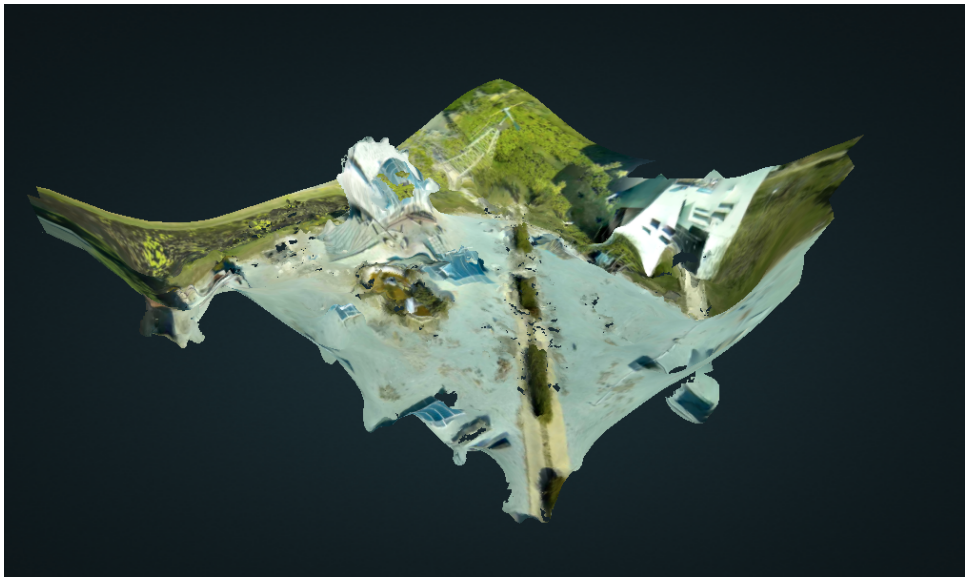


Figure 4.12
Camera Position calculated by ODM.


```
    "iris_1", {
      {5445, 4445, 3445},
      {7301, 7401, 7201},
     }
  },
  {
    "iris_2", {
      {5446, 4446, 3446},
      {7302, 7402, 7202},
     }
  },
  {
    CONTROL_STATION_STR , {
      {3444, 6444, 5444},
      {7200, 8200, 7300},
      {3445, 6445, 5445},
      {7201, 8201, 7301},
      {3446, 6446, 5446},
      {7202, 8202, 7302},
    }
  },
};
```

Figure 4.13
Gazebo simulation world (a) Top View. (b) Side View.

(a)

(b)

Table 4.4
OLSR messages' interval parameters for simulated experiment.

Message	Interval
Hello	2 sec
TC	5 sec
MID	5 sec
HNA	5 sec

Table 4.5
Result of `pegasus_planner` in simulated experiment with three drones.

Type	Number of Drones	Viewpoints Generated	Path Size
Simulation	3	25	11, 11, 10

4.2.2 Automated Planning

The area of interest was larger than the real world experiment and utilized three drones instead of one, and had more viewpoints. The operational height for the simulation was set at 30 meters. The grid size was set at 10 meters. The mesh range was set to 40 meters. The approximate area to cover was 50 by 55 meters. Planning required 3.74 seconds. Results of planning are given in Table 4.5. The path planned by the planner is shown in Figure 4.14. The total number of viewpoints is 25. The three drones cumulatively covered 32 viewpoints. Therefore, seven viewpoints were visited more than once.

4.2.3 Motion Control

The pegasus_controller was able to control three drones at once though the pegasus_commander process running on each drone. The transformations between the local maps of each drones and the global map were successfully calculated. Pegasus_controller was able to successfully carry out the coordinated mission without the drones colliding with each other. The paths taken by the drones are shown in Figure 4.15. The system never lost mesh connectivity during the mission.

4.2.4 Results

Thirty images were acquired from the three drones in real time. The images were captured with the gst (gstreamer) camera plugin for Gazebo without any distortion. The captured images are shown in Figure 4.16. The position of the camera for each image as described by the EXIF tag is shown in Figure 4.17. This corresponds with the positions of the viewpoints.

The network traffic and the ODM outputs are analyzed for the simulation.

Figure 4.14
Paths generated for three drones in simulation experiment.

Figure 4.15
The paths followed by the drones as reported through the simulated GPS.

Figure 4.16
The images captured in real time during simulation.

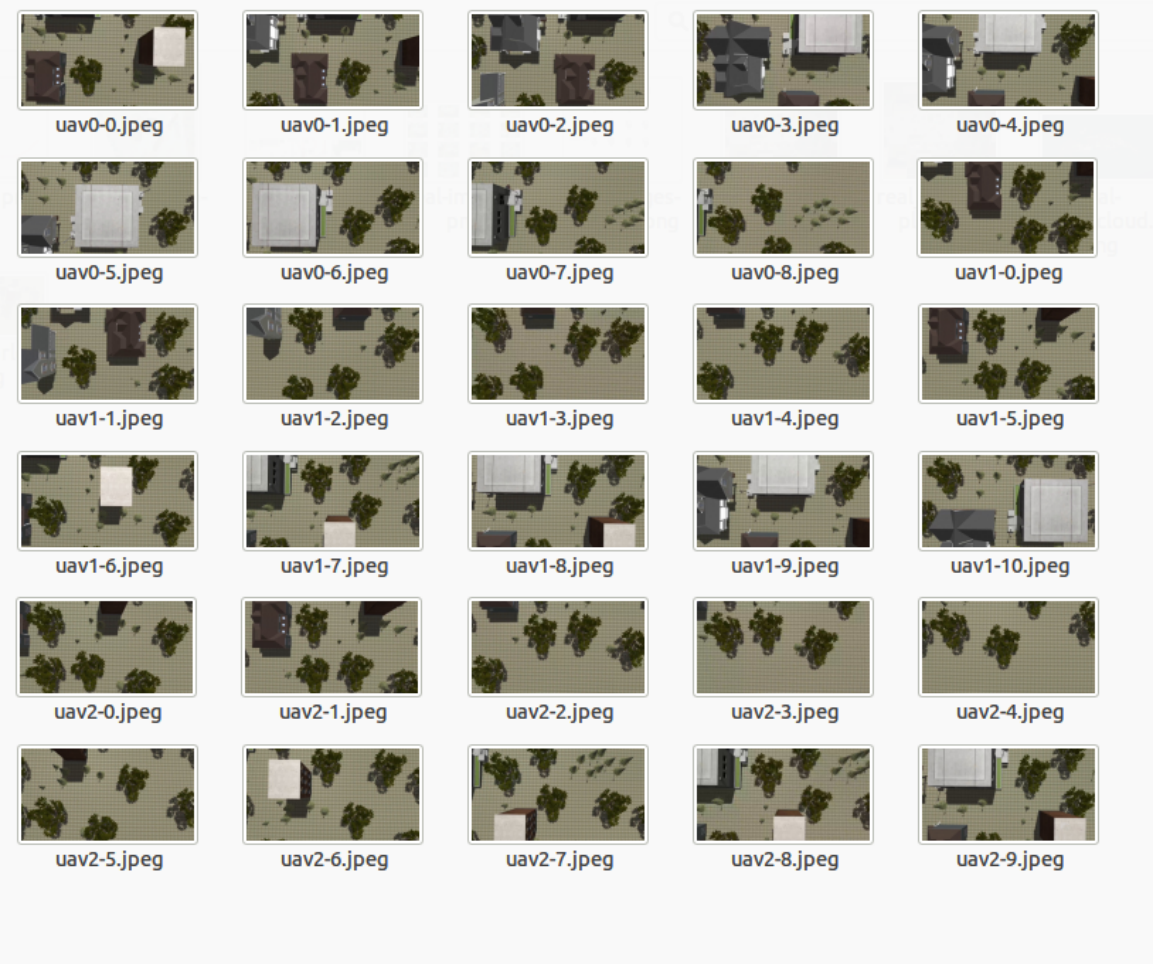
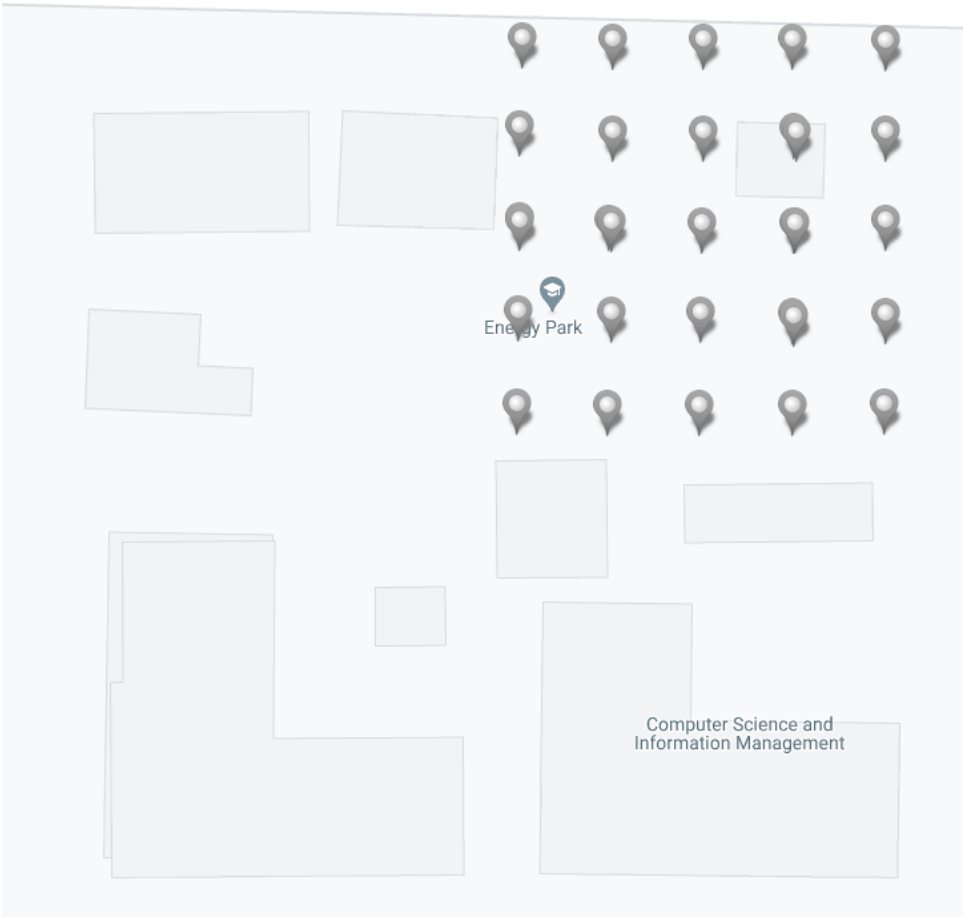


Figure 4.17
The GPS location of the images captured during simulation.



4.2.4.1 Network Properties The traffic in the simulated network was captured in pcap format, then the Wireshark application was used to analyze the traffic. The distribution of packet size is given in Figure 4.18a. There were a total of 217,475 packets. 114,551 (52.7%) of the packets were either IEEE 802.11 frame control or management packets. 59,120 (27.2%) of the packets were OLSR packets. 43,716 (20.1%) of the packets were generated by the system. Frame control packets are 14 bytes, and management packets have a size of 65 to 75 bytes. OLSR packets have sizes between 98 and 122 bytes. The Pegasus system generated packets of minimum size 99 to a maximum of 1046 bytes. Figure 4.18b shows that the Pegasus system was responsible for the least amount of packets; most of of the packets in the network are OLSR and IEEE 802.11 overhead. The Pegasus system generates two types of packets, motion control and image acquisition packets. Figures 4.19a and 4.19b show the packet count and megabytes transferred between the GCS and the drones during the simulation. The motion control packets are in the range of kilobytes and did not put any load on the system. The image acquisition layer generated a total of 21.8 megabytes to transfer thirty images. The image acquisition layer data depends upon the resolution of the cameras used, hence higher resolution cameras will generate more packets. The cumulative size of the thirty images stored at the GCS was 18.6 megabytes. The image acquisition layer generated a total of 3.2 megabytes as overhead data.

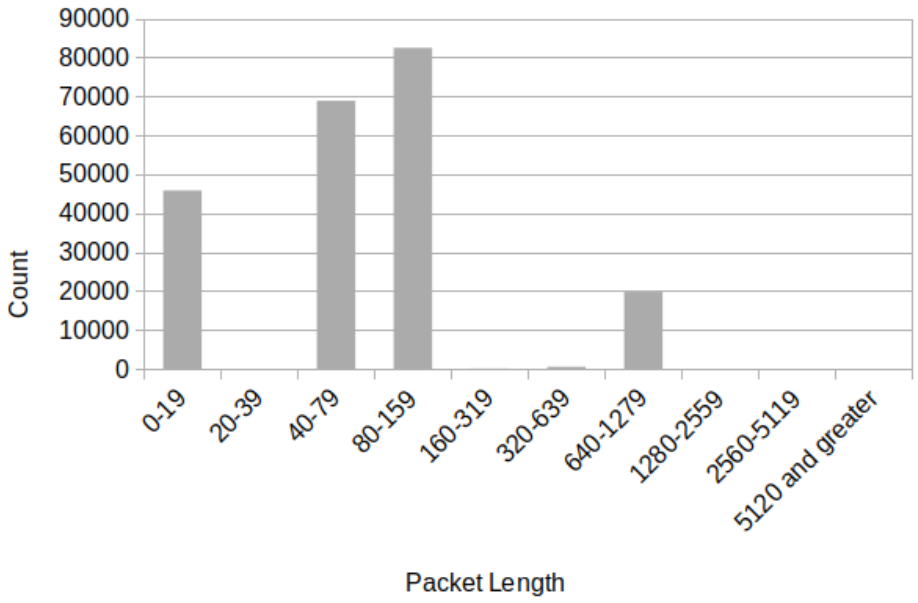
4.2.4.2 Map building The images did not have distortion, hence they were directly processed using WebODM.

- Figure 4.20 shows the generated orthophoto of the region of interest. The resulting orthophoto is not skewed, and straight lines in the simulated world appear as straight lines in the orthophoto. It covers the entire area of interest selected for the experiment. This result gives us the hint that better cameras with less distortion will result in better orthophotos in real world experiments.
- Figure 4.21 shows the point cloud for the region of interest. The point cloud is sparse but shows better definition than in the real world experiment. This result also provides the insight that less distorted cameras should provide better point clouds.
- Figure 4.22 shows the generated textured mesh of the region of interest. The walls of the buildings are not present because the images are taken from a top-view perspective.
- Figure 4.23 shows the camera positions determined by ODM over the region of interest, which corresponds to the simulated GPS positions. The SfM pipeline was able to correctly determine the pose of the cameras when the image did not have distortion.

Figure 4.18

Packet size and type distribution in simulated network. (a) The distribution of packet size. (b) The distribution of packet type.

(a)



(b)

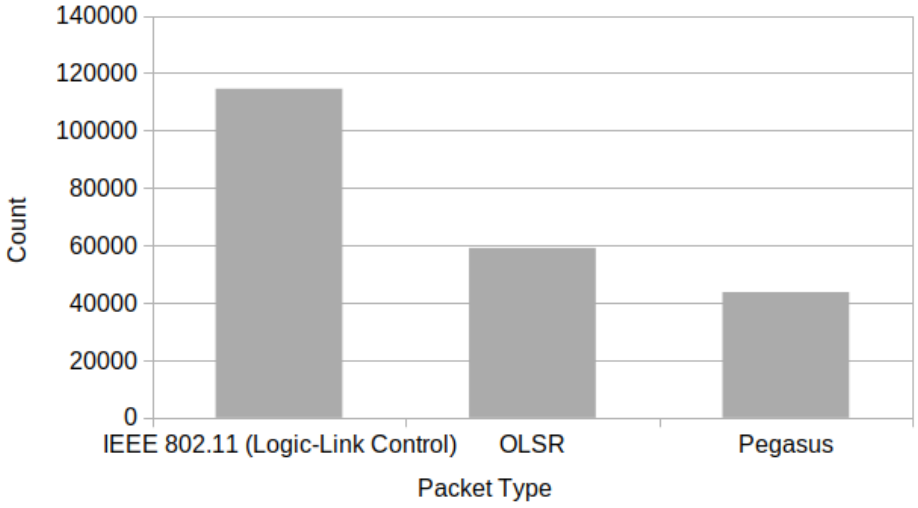
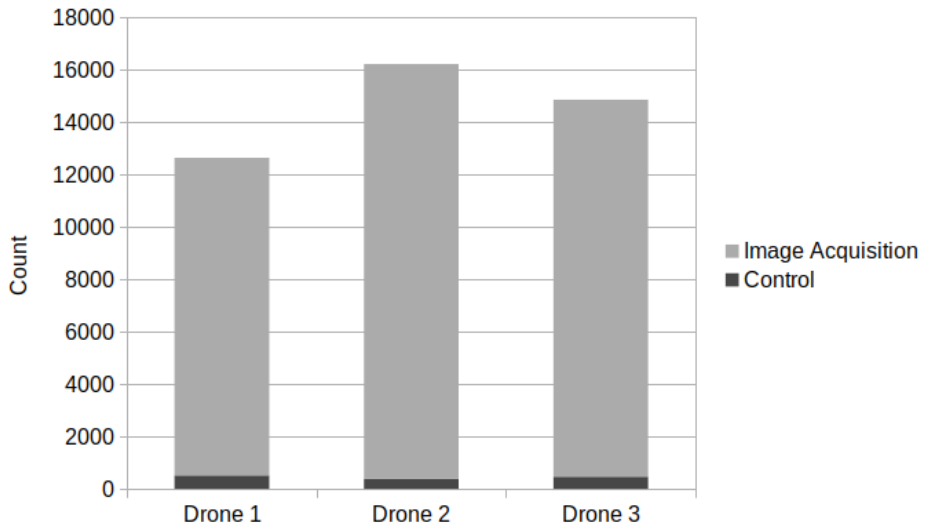


Figure 4.19
Packets generated by `Pegasus` system between the GCS and the simulated drones. (a) Based on packet count. (b) Based on megabytes transferred.

(a)



(b)

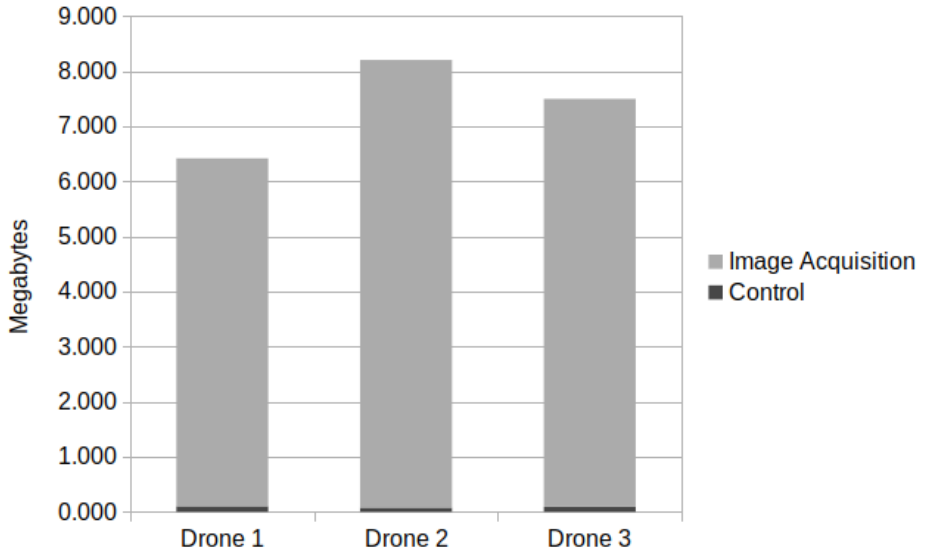


Figure 4.20
Orthophoto of simulated region of interest.

Figure 4.21
Point cloud of simulated region of interest.

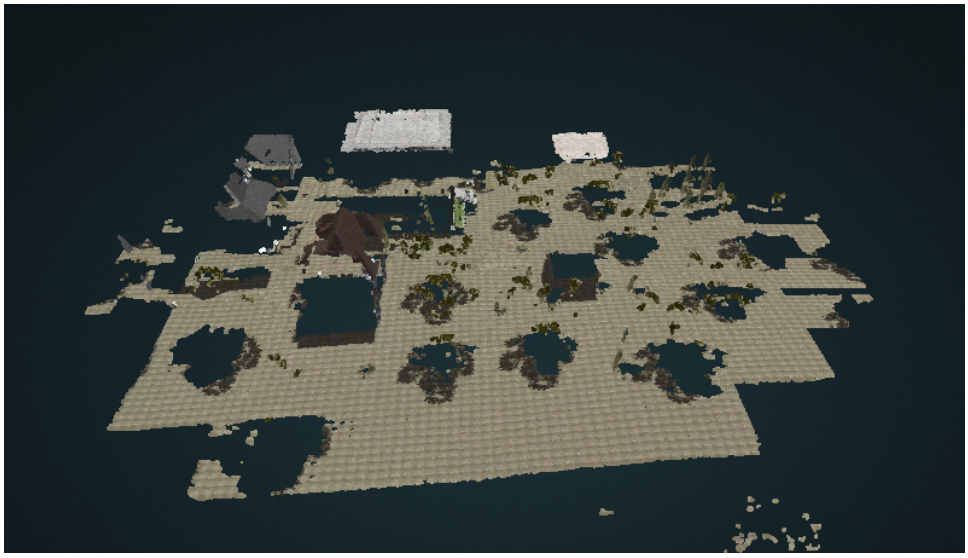


Figure 4.22
Textured mesh of simulated region of interest.

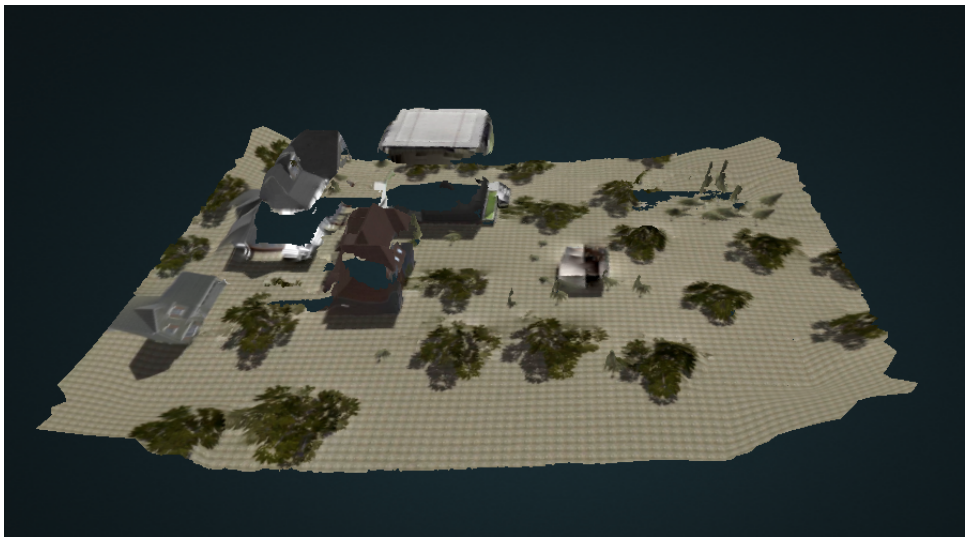


Figure 4.23
Position of simulated camera calculated by ODM.

4.2.5 Summary of Simulation Experiment

The system executed the presentation layer, planning layer, motion control layer, image acquisition layer, and map building layer using three drones. Analysis of the simulated network showed that the IEEE 802.11 logic link layer and OLSR added a great deal of overhead to the system. The networking overhead was actually larger than the actual data generated by the system. The motion control layer generated a small amount of traffic. The simulated cameras provided images without any distortion. The quality of the orthophoto, 3D point cloud and the textured mesh was good in the simulation experiment.

4.3 Chapter Summary

In this chapter, the path planned, the real path taken by the drone, the images acquired by the GCS in real time, and the output of ODM (orthophoto, point cloud and textured mesh) for both the real world and simulation experiments are presented. The distribution of packet size and traffic in the simulated mesh network for the simulation experiment is also described. Overall, the system performs well.

CHAPTER 5
CONCLUSION

The thesis study is concluded in this chapter and recommendations for future work is provided. The system is able to run in simulation with multiple drones and it has been evaluated in the real world with one drone.

5.1 Conclusion

I propose a system to autonomously survey a region of interest using multiple drones connected to a wireless mesh network using a ground control station to coordinate their movements in my thesis study. Furthermore, the system is capable of acquiring geo-tagged images from the drones in real time, which can be used after the mission is complete to build a 3D model and orthophoto image of the region of interest. One of the use cases of the Pegasus system is to survey an area after a disaster using multiple drones autonomously. Multiple drones can survey a region of interest in a shorter amount of time than a single drone. Thus, cost effective drones can be used despite the constraints of short battery life.

The system provides the operator with an interface to select the area of interest to survey. Viewpoints are generated in the area of interest. An iterative A* search algorithm is used find an efficient path for the drones to cover the viewpoints without losing mesh network coverage or colliding with each other. Since the problem of finding the optimal paths for the drones to cover all the viewpoints is NP-hard, the A* algorithm is tweaked to provide reasonable solutions in a reasonable amount of time. This forms the planning facet of the system.

The drones are controlled in real time by a centralized ground control station connected to the mesh network. Each of the drones and the GCS operate in different local frames of reference. The system finds the transformation between all local frames of reference and the global frame of reference through a predefined calibration routine that is run before the drones start following their planned paths. Then each drone operates within its local map while following planned paths calculated in the global map. The GCS instructs each drone to go to a desired location with a desired orientation. If the drone loses communication with the GCS, the drone returns to its home position autonomously. This forms the motion control aspect of the system.

The GCS acquires geo-tagged images from the drones in real time using the mesh network. This forms the image acquisition part of the system. Finally, the images acquired during the mission from the drone are rectified and processed with WebODM. This is the map-building part of the system.

The system was tested through simulation with popular simulation tools. PX4 SITL, Gazebo and ns-3 were integrated and used together to simulate the working of the system with three drones. Simulation showed that the system will work with multiple drones. It also showed that the traffic generated by the system between the drones and the GCS is only a few megabytes, mostly for image acquisition.

The system was also tested in the real world using one real drone. The system was able to control the motion of the drone in real time, plan a path for the area of interest, and acquire images from the planned viewpoints during flight. The acquired images were used to build a 3D model, a textured mesh, and an orthophoto of the area the system surveyed autonomously.

The source code for the developed system is available at the Github repository `https://github.com/rmukhia/pegasus`.

5.2 Recommendations

The system is unsafe, that is, it does not have obstacle avoidance and dynamic path adjustment. It also does not have fail-safe mechanisms for fatal conditions such as low battery in the drone. If the battery becomes low enough, the drone will simply fall to the ground from whatever altitude it is at. The system can be made more safe by integrating sensors in the drones to avoid surrounding obstacles. The system can also be extended to include real-time monitoring of the battery level of the drones to execute fail-safe behavior if the battery level falls below a particular threshold.

Currently, the system is complicated to set up. There are many ROS nodes and launch files required to execute the system. Further work can be done to make the system easier to access.

Using a higher quality camera with less radial distortion and higher resolution on the drone will provide better results, but the companion computer may need to be upgraded to handle a higher resolution video feed.

5.3 Chapter Summary

This chapter provides the conclusion and result of the study. Recommendations to further improve the system are mentioned in the chapter.

REFERENCES

Acharya, S., Bharadwaj, A., Simmhan, Y., Gopalan, A., Parag, P., & Tyagi, H. (2020). CORNET: A Co-Simulation Middleware for Robot Networks. In *2020 International Conference on COMmunication Systems and NETworkS, COMSNETS 2020* (pp. 245–251). doi: 10.1109/COMSNETS48256.2020.9027459

Almadhoun, R., Taha, T., Seneviratne, L., & Zweiri, Y. (2019, aug). A survey on multi-robot coverage path planning for model reconstruction and mapping. *SN Applied Sciences*, *1*(8). doi: 10.1007/s42452-019-0872-y

Alshabtat, A. I., & Dong, L. (2011). Low latency routing algorithm for unmanned aerial vehicles ad-hoc networks. *World Academy of Science, Engineering and Technology*, *80*, 705–711.

Arkin, E. M., Fekete, S. P., & Mitchell, J. S. (2000). Approximation algorithms for lawn mowing and milling.A preliminary version of this paper was entitled "The lawnmower problem" and appears in the Proc. 5th Canad. Conf. Comput. Geom., Waterloo, Canada, 1993, pp. 461–466. *Computational Geometry*, *17*(1-2), 25–50. doi: 10.1016/s0925-7721(00)00015-8

Baidya, S., Shaikh, Z., & Levorato, M. (2018). FlynetSim: An open source synchronized UAV network simulator based on ns-3 and ardupilot. In *MSWiM 2018 - Proceedings of the 21st ACM International Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems* (pp. 37–45). Retrieved from `https://doi.org/10.1145/3242102.3242118` doi: 10.1145/3242102.3242118

Behere, S., & Törngren, M. (2015). A functional architecture for autonomous driving. In *WASA 2015 - Proceedings of the 2015 ACM Workshop on Automotive Software Architecture, Part of CompArch 2015* (pp. 3–10). Retrieved from `http://dx.doi.org/10.1145/2752489.2752491.` doi: 10.1145/2752489.2752491

Chand, G., Lee, M., & Shin, S. (2018). Drone Based Wireless Mesh Network for Disaster/Military Environment. *Journal of Computer and Communications*, *06*, 44–52. doi: 10.4236/jcc.2018.64004

Chriki, A., Touati, H., Snoussi, H., & Kamoun, F. (2019, nov). FANET: Communication, mobility models and security issues. *Computer Networks*, *163*, 106877. doi: 10.1016/j.comnet.2019.106877

Dileep Muddu, R. S., Wu, D., & Wu, L. (2015). A frontier based multi-robot approach for coverage of unknown environments. In *2015 IEEE International Conference on Robotics and Biomimetics, IEEE-ROBIO 2015* (pp. 72–77). Institute of Electrical and Electronics Engineers Inc. doi: 10.1109/ROBIO.2015.7414626

Galceran, E., & Carreras, M. (2013). A survey on coverage path planning for robotics. *Robotics and Autonomous Systems*, *61*(12), 1258–1276. Retrieved from `http://dx.doi.org/10.1016/j.robot.2013.09.004` doi: 10.1016/j.robot.2013.09.004

Ganganath, N., Cheng, C. T., & Tse, C. K. (2016, oct). Distributed Antiflocking Algorithms for Dynamic Coverage of Mobile Sensor Networks. *IEEE Transactions on Industrial Informatics*, *12*(5), 1795–1805. doi: 10.1109/TII.2016.2519913

Ivaldi, S., Padois, V., & Nori, F. (2014, feb). Tools for dynamics simulation of robots: a survey based on user feedback. Retrieved from `http://arxiv.org/abs/1402.7050`

Kulla, E., Hiyama, M., Ikeda, M., & Barolli, L. (2012, jan). Performance comparison of OLSR and BATMAN routing protocols by a MANET testbed in stairs environment. In *Computers and Mathematics with Applications* (Vol. 63, pp. 339–349). Pergamon. doi: 10.1016/j.camwa.2011.07.035

Mbarushimana, C., & Shahrabi, A. (2007). Comparative study of reactive and proactive routing protocols performance in mobile ad hoc networks. In *Proceedings - 21st International Conference on Advanced Information Networking and Applications Workshops/Symposia, AINAW'07* (Vol. 1, pp. 679–684). doi: 10.1109/AINAW.2007.123

Meeussen, W. (2010). *REP 105 – Coordinate Frames for Mobile Platforms (ROS.org).* Retrieved 2020-05-15, from `https://www.ros.org/reps/rep-0105.html#coordinate-frames`

Meza, J., Marrugo, A. G., Ospina, G., Guerrero, M., & Romero, L. A. (2019). A Structure-from-Motion Pipeline for Generating Digital Elevation Models for Surface-Runoff Analysis. In *Journal of Physics: Conference Series* (Vol. 1247). doi: 10.1088/1742-6596/1247/1/012039

Özyeşil, O., Voroninski, V., Basri, R., & Singer, A. (2017). A survey of structure from motion. *Acta Numerica*, *26*, 305–364. doi: 10.1017/S096249291700006X

Patel, R., Pathak, M., & Nayak, A. (2018). Survey on Network Simulators. *International Journal of Computer Applications*, *182*(21), 23–30. doi: 10.5120/ijca2018917974

Quigley, M., Conley, K., Gerkey, B. P., Faust, J., Foote, T., Leibs, J., ... Ng, A. Y. (2009). {ROS}: An open-source robot operating system. In *Workshops at the {IEEE} International Conference on Robotics and Automation.*

Rosati, S., Kruzelecki, K., Traynard, L., & Rimoldi, B. (2013). Speed-aware routing for UAV ad-hoc networks. In *2013 IEEE Globecom Workshops, GC Wkshps 2013* (pp. 1367–1373). IEEE Computer Society. doi: 10.1109/GLOCOMW.2013.6825185

Sabino, S., Horta, N., & Grilo, A. (2018). Centralized Unmanned Aerial Vehicle Mesh Network Placement Scheme: A Multi-Objective Evolutionary Algorithm Approach. *Sensors*, *18*(12), 4387. Retrieved from `http://dx.doi.org/10.3390/s18124387` doi: 10.3390/s18124387

Singh, K., & Verma, A. K. (2015, jan). Applying OLSR routing in FANETs. In *Proceedings of 2014 IEEE International Conference on Advanced Communication, Control and Computing Technologies, ICACCCT 2014* (pp. 1212–1215). Institute of Electrical and Electronics Engineers Inc. doi: 10.1109/ICACCCT.2014.7019290

Siraj, S., Gupta, A. K., & Badgujar-Rinku. (2012). Network Simulation Tools Survey. *International Journal of Advanced Research in Computer and Communication Engineering Vol. 1, Issue 4, June 2012*, *1*(4), 201–210.

Staranowicz, A., & Mariottini, G. L. (2011). A survey and comparison of commercial and open-source robotic simulator software. In *ACM International Conference Proceeding Series.* Association for Computing Machinery. doi: 10.1145/2141622.2141689

Tareque, M. H., Hossain, M. S., & Atiquzzaman, M. (2015). On the routing in flying ad hoc networks. *Proceedings of the 2015 Federated Conference on Computer Science and Information Systems, FedCSIS 2015*(October), 1–9. doi: 10.15439/2015F002

Toor, A. S., & Jain, A. K. (2017). A survey on wireless network simulators. *Bulletin of Electrical Engineering and Informatics*, *6*(1), 62–69. doi: 10.11591/eei.v6i1.568

Viet, H. H., Dang, V. H., Laskar, M. N. U., & Chung, T. (2013). BA: An online complete coverage algorithm for cleaning robots. *Applied Intelligence*, *39*(2), 217–235. doi: 10.1007/s10489-012-0406-4

Wikipedia. (2019). *Geographic coordinate system.* Retrieved 09-12-2020, from `https://en.wikipedia.org/wiki/Geographic{_}coordinate{_}system`

Xiao, J., Wang, G., Zhang, Y., & Cheng, L. (2020). A Distributed Multi-Agent Dynamic Area Coverage Algorithm Based on Reinforcement Learning. *IEEE Access*, *8*, 33511–33521. doi: 10.1109/ACCESS.2020.2967225

Xu, X., Yang, L., Meng, W., Cai, Q., & Fu, M. (2019, jul). Multi-agent coverage search in unknown environments with obstacles: A survey. In *Chinese Control Conference, CCC* (Vol. 2019-July, pp. 2317–2322). IEEE Computer Society. doi: 10.23919/ChiCC.2019.8865126

Zheng, Y., Wang, Y., Li, Z., Dong, L., Jiang, Y., & Zhang, H. (2014). A mobility and load aware OLSR routing protocol for UAV mobile AD-HOC networks. *2014 International Conference on Information and Communications Technologies, ICT 2014*.