COLLABORATIVE COORDINATION AND CONTROL FOR AN IMPLEMENTED HETEROGENEOUS SWARM OF UAVS AND UGVS

By

JOSHUA N. WEAVER

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

2014

To my awe-inspiring wife Toni and my amazing children Zoe and Kaiden

ACKNOWLEDGMENTS

First, I would like to start by thanking my extraordinary wife for her continued love, patience, and support during the pursuit of my degrees. She not only believed in my potential, but motivated and cheered me on during the times that I did not trust in myself. Without her by my side, I would have been unable to accomplished my goals. Next I want to thank my wonderful children for providing hope and reason to continue my work.

I would also like to thank my advisors Dr. A. Antonio Arroyo and Dr. Eric M. Schwartz for guidance, support, and understanding over the years. It was with their assistance that provided me the opportunity to continue my education in the environment of robotics that I enjoy. They provided me with the freedom to follow my goals in research, while also instructing me in my many other interests of education. Further, I would like to thank the rest of the members of the Machine Intelligence Lab, with a special thanks to my team, CongreGators. Your friendship and collaboration during my time at the University of Florida made it much more enjoyable.

I am also thankful for the support I received from the Department of Defense S.M.A.R.T. Fellowship Program. It was through this program that I received the funding for my graduate degree and the opportunity to further my education.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ALVAR	A Library for Virtual and Augmented Reality
APM	ArduPilot-Mega
AR	Augmented Reality
COTS	Commercial Off-the-Shelf
GPS	Global Position System
GRASP	General Robotics, Automation, Sensing, & Perception
GUI	Graphical User Interface
IMU	Inertial Measurement Unit
LIDAR	Light Detection and Ranging
MANET	Mobile Ad-Hoc NETWorks
MAVLink	Micro Air Vehicle Link
MIL	Machine Intelligence Lab
MSG	Message
OpenCV	Open Source Computer Vision
PID	Proportional Integral Derivative
RC	Remote Control
ROI	Region of Interest
ROS	Robot Operating System
RPC	Remote Procedure Call
RQT	ROS Qt
SLAM	Simultaneous Localization And Mapping

SRV Service

UAV Unmanned Aerial Vehicle

UGV Unmanned Ground Vehicle

USV Unmanned Surface Vehicle

UTM Universal Transverse Mercator

UUV Unmanned Underwater Vehicle

Abstract of Dissertation Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy

COLLABORATIVE COORDINATION AND CONTROL FOR AN IMPLEMENTED HETEROGENEOUS SWARM OF UAVS AND UGVS

By

Joshua N. Weaver

May 2014

Chair: A. Antonio Arroyo
Major: Electrical and Computer Engineering

Over the last few years, cooperative autonomous systems have become a popular solution for accomplishing tasks that are otherwise performed by human operators. Several strides have been made with homogeneous systems of vehicles in areas of localization, formation behaviors, path planning, task allocation, and vehicle controls. This literature will describe work performed in the development of a heterogeneous system and architecture of platforms consisting of unmanned ground vehicles (UGVs) and unmanned aerial vehicles (UAVs) that cooperatively work together to accomplish tasks such as search and rescue in an outdoor environment. The Robot Operating System (ROS) is used between and within each vehicle to handle cooperative high level planning and task allocation, as well as control of each vehicle's individual capabilities specializing in identification or classification. Mission control between vehicles includes heterogeneous path planning, task allocation, and vehicle control through a custom swarm architecture. A description of the system as well as developed vehicle capabilities is also described. Finally, real environment testing results are given to characterize the benefits of the heterogeneous architecture.

CHAPTER 1
INTRODUCTION

Recently, unmanned systems have come to the forefront of technology being developed for military, civilian, and research purposes. This is in part due to their growing economic feasibility, ease of use, and the ability to travel to locations humans are unable to reach. These systems are also able to safely and effectively handle difficult situations without putting humans lives at risk. This ability to create a safer environment in dangerous situations has created a push in both the private and public sector to make these tools more accessible. One main focus is to replace manned aerial and ground capabilities with unmanned aerial (UAV), ground (UGV), underwater (UUV), and surface (USV).

A focus has also been placed on the increased autonomy of these systems to make decisions and accomplish tasks with minimal user input, freeing up manpower for other uses. This is highly useful in a system where more than a single vehicle is used to search and gather information. For instance, in a search and rescue scenario, where safely covering as much ground as possible in a very short amount of time is key, vehicles that can perform their task without taking up manpower is ideal. Beyond simply building autonomous unmanned vehicles, research goals have looked into the benefits of using homogeneous, i.e. similar nature and type vehicles, or heterogeneous, which uses varying types of vehicles, possibly in different planes of operation.

1.1 Motivation

This research describes a swarm architecture used within a system of multiple autonomous unmanned ground and aerial vehicles forming a heterogeneous cooperative swarm to accomplish goals and missions such as search and rescue in an outdoor environment. Though some examples of fully heterogeneous systems exist, this is the first time, as far as we know, that a system on this level has been created. A coalition of UGVs and UAVs have been created to work independently of each other;

with each vehicle having the capacity of being completely autonomous using either waypoint navigation or a mission arbitrator that exists within the Robot Operating System (ROS) environment. Though each vehicle is independent they have the capability to cooperatively plan their desired tasks to complete the mission.

Vehicles are initially unknown to each other as they join the swarm architecture. During the processes of connecting to the swarm, each vehicle goes through a process of roll call handshaking to acknowledge the base controller of the swarm while allowing all other vehicles already in the swarm to be updated of it's presence. The architecture allows a defined mission to be processed and sent to vehicles in the swarm, while also updating new vehicles as they come on line. The architecture allows multiple plugin mission types as well as behaviors for vehicle controls per mission type. One such mission may be defined as a search and rescue mission where vehicles split the field of operation into sections based on the number of vehicles in the group. UAVs may be used to either search the field ahead of UGVs to mark low-confidence targets of interest as well as obstacles that should be avoided, or to search with UGVs, staying above them as they travel through the environment. Whether with or following behind UAVs, UGVs are typically used to better identify targets with a higher degree of confidence. When targets of interests are determined, a classification phase is started in which a specialized group of UGVs will travel to objects of interest to attempt to classify the target for any human operators.

1.2 Organization

This dissertation is organized as follows: In Chapter 2, previous work from external research in homogeneous and heterogeneous cooperative systems as well as their developments into advancements that help these systems are detailed. Cooperative systems that have been fully developed by university and small project research is first described followed by the small advancements in systems that are not fully tested but still may be used to improve cooperative systems.

In Chapter 3, relevant work contributing to the primary research is highlighted. A homogeneous swarm of UGVs that interact through a cloud environment is presented, detailing the system and what has been accomplished. An autonomous UAV that is controlled through the Robot Operating System (ROS) environment and is capable of taking pre-planned missions through an Android Tablet is described in terms of vehicle control, sensor control, and Android application detail. Further details of expanding the autonomous UAV system in ROS to multiple vehicles are given. Finally a vision system made for allowing an UAV to launch and land from an autonomous boat is described.

In Chapter 4, the primary research of the heterogeneous collaborative system are detailed. Information in regards to unmanned ground and aerial vehicles used in the cooperative system is given, following a breakdown of the system information sharing process. The swarm architecture used to allow the various vehicles to communicate and work together is described. The high level goal planner and low level vehicle controller systems are explained as well as how they may be modified for custom designs. Finally, the LIDAR and vision based obstacle detection system is detailed as well as how information is shared between agents.

In Chapter 5, the method for testing and general assumptions are outlined. Results from the various stages of testing are given. Experimentation results with methods of centralized and decentralized control are provided. Conclusions and contributions of this research are examined. Furthermore, plans and ideas for future work within this research are described.

CHAPTER 2
BACKGROUND

There are various focuses currently being researched in the area of cooperative homogeneous and heterogeneous systems. Some of these systems focus on the concept of homogeneous vehicles swarming and forming a collective behavior, while others focus more on developing additional capabilities of using a heterogeneous system in either a single environment or multiple environments, such as sea, ground, or air.

2.1 Fully Developed Cooperative Systems

Over the past few years the state of the art for homogeneous and heterogeneous groups of vehicles have been researched. Duan and Liu found various projects that have advanced techniques in flocking, formation, and network control, which were used in various mission types for demonstration [7]. Specifically they found groups, such as the University of Pennsylvania, looked into systems of UGVs and UAVs that work cooperatively to accomplish tasks by creating formations. They also found a strong push to create systems utilizing the Mobile Ad-Hoc Networks (MANET) feature which turns UAVs into network repeaters for UGVs. They also discuss the mission types these systems are being built to work within, such as Searching and Localization, or Tracking and Pursuit-Evasion systems.

A more in-depth review of the these findings as well as well as other projects and their developments has been given below.

2.1.1 University of Pennsylvania

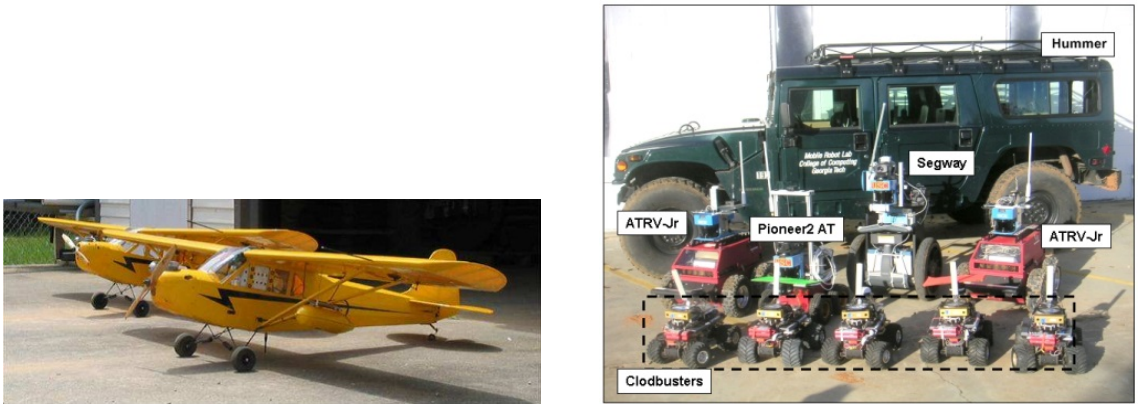
The University of Pennsylvania General Robotics, Automation, Sensing, and Perception (GRASP) Lab has been working on the development of collaborative systems over the past decade. During early development, the GRASP Lab focused on formation control of ground vehicles using a system of UGVs that coalescing into formations around a shepherding UAV, or blimp as shown in Figure 2-1 [2, 3, 22]. Focus

was then placed on using Expected Maximization statistical and pattern recognition methods for generating Gaussian distributions of vehicles on the ground, then using these distributions in order to split the groups into formations per UAV. The overall goal of this system was to show how multiple UAVs could be used to collect information on a group of UGVs, then using this information they command groups of UGVs to first form a formation, and then guiding the group from waypoint to waypoint.

Figure 2-1. UAV Blimp and UGV vehicles used in earlier UPenn research

UPenn also developed a system for performing reconnaissance and tracking in an urban environment using UAVs and UGVs [11, 15, 23]. In this system, shown in Figure 2-2, UAV planes outfitted with an Inertial Measurement Unit (IMU), control package, and camera package are tasked with flying into an urban area and planning their own path to search and collect ground feature data. Once coverage is complete, the UAVs travel back to a base where the data is off-loaded, processed, and used to create an a priori map of features for UGVs to use. Once generated, this map is sent to a group of UGVs, which are then commanded to enter the field of operation and perform search and tracking behaviors according to targets of interest on the map.

One such demonstration shows the UGVs locating a building that holds a target of interest and then tracking/following that target outside the field of operation. A major focus for this system was how to use a priori information to localize targets in the area so that ground vehicles are able to efficiently collect data.



A B

Figure 2-2. Collection of vehicles used in Multi-Robot team in GRASP Lab: A) UAV Planes used by Grasp Lab B) Collection of UGVs used by Grasp Lab

More recently, UPenn has become famous for their various videos of cooperation between UAV quadrotors performing formation flights, aerial dances, grasping and moving objects, building structures, and various other cooperative tasks. Kumar and Michael described how micro UAVs, specifically quadrotors, offer a new area of opportunity in research and application for systems such as UPenn's previous work [16]. One of the first projects applied to this theory was the use of a group of AscTec Hummingbird Quadrotors UAVs collectively moving from point to point in a tight formation (shown in Figure 2-3). Given precise pose information, Turpin *et al.* described how a formation is created using parameters given through a shape matrix [29] . Each UAV plans its own trajectory, using both the given parameters and a motion sensing camera to estimate the state of other UAVs and plan accordingly. Mellinger *et al.* developed a system of quadrotors that cooperatively grab and move wood blocks of varying size throughout a three-dimensional environment [21]. Most of the focus of these systems has been placed on the non-linear control and trajectory planning for each vehicle in a controlled environment. All of the above work is performed in a controlled environment with multiple motion capture cameras by Vicon. Although vehicles within

these systems know their exact pose in the environment, a limitation due to the use of these cameras will not allow operation outside of a strictly controlled environment.

Figure 2-3. GRASP Lab UPenn AscTec quadrotor swarm

2.1.2 Eidgenössische Technische Hochschule Zürich

ETH Zurich is another university gaining notoriety for their use of UAV multirotor vehicles. Through a research partnership with AscTec, they use Hummingbird, Pelican and Firefly UAVs to cooperatively accomplish tasks. One such project incorporates a Swarm of micro FLYing robots (sFLY) for use in search and rescue mission where UAVs attempt to find victims in a GPS-Denied environment [1].

ETH Zurich details a system where multiple UAVs are used to collect imagery via a single camera, which is then fused with vehicle state information to form a global map with highlighted areas of interest [5, 35]. The focus of this work is to perform localization by creating a map that is used to extract locations of objects in the area to develop a pose, and then locating objects of interest in relation to the vehicles own understanding of the environment. Though initial testing was performed in a controlled environment, they have since moved to outside environments and shown that their method is still effective.

As an example of their efficient control of quadrotors, ETH Zurich's Flying Machine Arena lab developed algorithms that allow quadrotors to cooperate in playing acrobatic games. One such algorithm, as defined by Ritz *et al.*, allows a fleet of quadrotors to play

catch with a ball [25]. This work has advanced into a system where quadrotors balance poles on the top of their frame and then have the ability to ”toss” the pole between vehicles. This work follows closely to the University of Pennsylvania with the focus on controls and trajectory tracking of UAV solely operates by use of a motion capture camera system.

2.1.3 Stanford University

In 2001, Stanford University created a project known as the ”Stanford Testbed of Autonomous Rotorcraft for Multi Agent Control” or STARMAC [12]. Using both centralized and decentralized control, their research focused on the use of small UAV quadrotors using waypoint navigation, with a goal of developing a testbed system in outdoor environments in order to show the validity of using multi-agent algorithms in a real-world situation. Later projects developed a trajectory tracking control system that create a path from given waypoints and velocities allowing a UAV quadrotor to navigate through a cluttered environment [13]. The system was built for a singular UAV and tested in an indoor environment, with eventual goals of placing the system technique into a group of UAVs for improved control.

2.1.4 Smaller Collaborative Projects

Several other groups have also created systems for cooperative vehicles, and though on a smaller scale than projects previous mentioned, significant advancements have been made. For instance, Vandapel *et al.* at Carnegie Mellon University worked on the development of a system using a single UAV and multiple UGVs [30]. Though the system did not cooperatively work with each other in real time, a system was created in which a UAV could be used to travel into a field of operation and collect imagery localization data. Once processed this data would be used by the UGVs to understand an environment before driving into it.

Another example of work focusing on quadrotor controls and trajectory tracking in formation has been in development at the Massachusetts Institute of Technology. Called

RAVEN (Real-time indoor Autonomous test ENvironment), this system, which used Vicon motion capture camera systems, performed robust coordination algorithms for a group of quadrotors to further develop vision-based sensing systems that both detected and guided the vehicles. How *et al.* describes the systems purpose as a way to simplify and handle multi-vehicle coordination and control at a lower level, therefore allowing high-level behaviors to be implemented quickly [14].

The University of Texas in Arlington has also worked with collaborative systems at different levels. Zengin and Dogan focused on how a group of UAVs following a cooperative technique can track a target through an adversarial environment [37]. They developed cost functions base off of information given such as zones that are declared unsafe in terms of threat level exposure, obstacles within the environment, the desired formation that the UAVs should keep while tracking, and finally the distance to the target. This cost function is then used to generate probability density functions that are used to steer the UAVs towards or away from objects within the environment while still tracking the primary target.

A more controlled, but larger heterogeneous swarm project was developed between vehicles called eye-bots and foot-bots. Ducatelle *et al.* and Mathews *et al.* both describe a system made of small track/wheel ground vehicles (foot-bots) and five thruster air vehicles (eye-bots) that solve tasks within a controlled indoor environment. Eye-bots are setup to fly through the environment and attach themselves magnetically to a ceiling. Though eye-bots are described as capable of flight, all tests explain that each is manually placed in a grid pattern to assist foot-bots in navigating between two targets in a foraging task system that closely mimics ant colony movement [8, 9]. Other focuses of the research show how a eye-bots assist foot-bots in traversing a gap clearing environment [19], using a single eye-bot to assist foot-bots in traveling over a hill in terms of angle steepness and calculating group formation in order to traverse an angle too steep for a single vehicle to climb [20].

Another example of focus in localization is given by De Silva *et al.* who describes a heterogeneous system, in which UAVs use a combination of visual and acoustic data to localize position with a ground vehicle within a closed environment [27]. A bearing sensor uses imagery data from a pinhole camera on the UAV to lock onto a ground vehicle and calculate the generated heading. A ranging sensor, or Time of Flight (TOF) sensor, uses an acoustic source and receiver between the UAV and UGV. The amount of time for the source to travel through air gives the distance between the two targets. A combination of the two sensors gives a statistically accurate relative location between the two vehicles.

More recently Garzón *et al.* developed a heterogeneous cooperative system between a single UAV and UGV (Figure 2-4) [10]. A UAV was manually flown above an autonomous UGV with the requirement that the UGV stay in the field of view during the entire flight. Aerial imagery was collected and processed to supply the UGV with obstacle data in the forward direction of the UGV. Using this information, a UGV could then traverse a path in an outside environment safely given that the UAV accurately portrayed the environment.

2.2 Systems Developed for Groups of UAVs or UGVs

There are a few research projects that place focus on individual components of a cooperative systems in a controlled or synthesized environment. One such project, explains Tanner and Christodoulakis of University of New Mexico, focuses on the coordination between two heterogeneous groups of UAVs and UGVs [28]. In their system, ground vehicles interact on a nearest neighbor system, synchronizing velocities to maintain formation during travel. Ground vehicles, meanwhile, estimate the centroid of the formation and transmit the information to an aerial vehicle which travels along with the formation.

Other systems focus on how to handle leader and follower development between robot swarms. Lee *et al.* uses a group consensus algorithm to assist in the selection

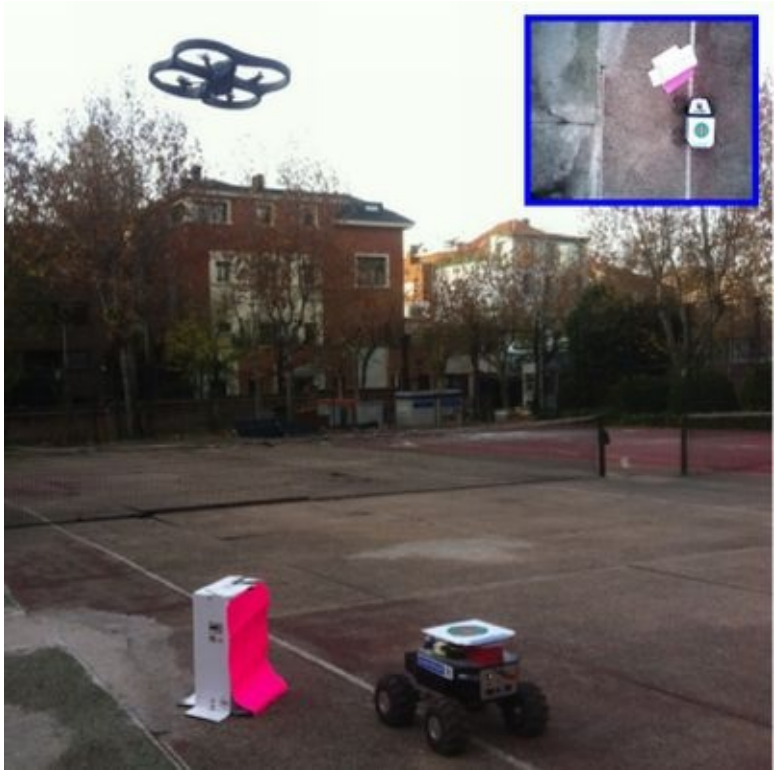


Figure 2-4. Object Avoidance using UAV and UGV

of a leader for a group of undefined robot swarms [17]. The algorithm allows for self-adjustment, giving a leader the ability to either recruit more members as the swarm grows, or as a task requires more agents. This system of vehicles communicates back and forth using local probabilistic methods to define which vehicle is best to lead and which vehicles will attempt to accomplish a given task, therefore, inherently splitting the group.

Vision is always a major component of cooperative systems, and in some cases, is used for docking of a UAV on a UGV. In order to combat the relatively limited run-time of a UAV, one such system utilizing both marker-based and optical-flow based computer vision techniques was attempted by Li *et al.*, giving an AscTec Hummingbird the ability to track position and velocity of a Pioneer 3-DX UGV vehicle and then to align to the UGV for docking [18]. Marker-based vision techniques are used to track two leds on the UGV to assist in the Hummingbirds tracking of the ground vehicle. Once the Hummingbird

is too close for marker-based vision to work, it switches to an optical-flow technique to finish landing the vehicle.

2.3 Conclusions

Through all of the papers reviewed, it can be seen that a major trend towards cooperative systems utilizing UAVs has been established. Most of the systems focus on control and formation behaviors. Various systems have the benefit of giving a more accurate pose for each vehicle given the use of a myriad of motion capture cameras, allowing research to focus more on the high-level control behaviors and less on how to get accurate information on vehicle location. As discussed, these platforms allow for higher functioning, however they cannot function in a real world test environment due to their dependency on cameras that require a controlled environment.

Other systems, such as ETH Zurichs sFly project, focus on how to use systems in an outdoor environment. These systems have created new techniques in Localization, Collective Networking, Formations and Flocking, Obstacle Avoidance, and Path Planning to allow a cooperative system to be utilized in a real world test environment. Though these systems are advancing the capabilities of a cooperative homogeneous group of vehicles, very few focus on the benefits of using heterogeneous groups.

CHAPTER 3
EXPLORATORY HOMOGENEOUS SWARM DEVELOPMENT

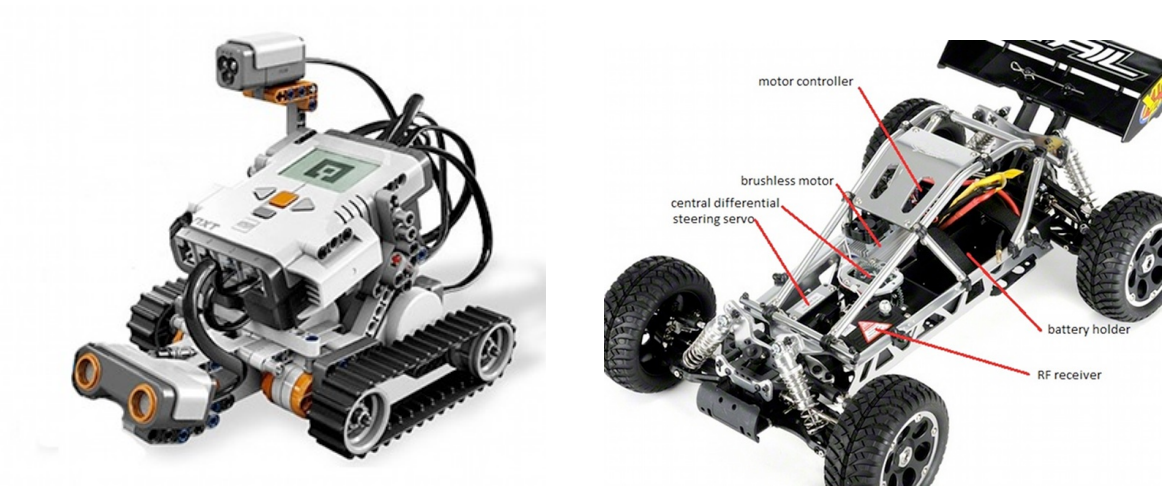
Initially, a better understanding in how to work with a collaborative system and to develop a base for the proposed heterogeneous system was pursued, using an external project with a homogeneous swarm of UGVs that was inherited and further developed. Afterward, a project was initiated to first develop a single UAV being controlled by an Android tablet within the Robot Operating System (ROS) environment was established. A follow up project was created to expand the capabilities of the UAV using vision to locate and land at a specified target.

3.1 Homogeneous Swarm of UGVs in Google Cloud

3.1.1 Development of Inherited System

Initially, an external research group first set out to develop an autonomous group of UGV robots with the hopes of forming an emergent collective behavior between all vehicles while completing a task. Wright *et al.* describes the goal of the work to be a system that uses simple and cost efficient vehicles with basic sensing capabilities to form a more capable group of vehicles with the ability to accomplish complex tasks [36]. Software development for the swarm of vehicles first began on Lego Minestorms NXT platforms (Figure 3-1A) to allow for flexible, but applied, testing of their system. The project was then moved to a more capable RC ground vehicle, the XTM Rail (Figure 3-1B), which was modified with a small ARM based computer, communication components, control electronics, and sonar sensing. The XTM Rail was used for experimental testing and exercises of missions such as Area Survey and Area Setup in an outdoor testing environment.

The swarm architecture consisted of three main levels: a cloud environment, a base station, and individual UGV swarm agents. The default setup of the system was for each individual UGV to have two-way communication with a base station that was located in the field of operation. The base station had the ability to send mission parameters



A B

Figure 3-1. Vehicles used in developing a homogeneous swarm of autonomous UGV robots: A) Lego Minestorm NXT Platform B) XTM Rail RC Vehicle

to each individual agent and, upon confirmation the mission was received, the base station could then send the command to execute the mission. While the mission was underway, the base station then entered a 10Hz command loop where it received desired information for each agent executing the mission.

The individual agents had their own execution loop that would receive a mission and once all parameters were received, would then send an acknowledgment command. During travel, these agents had a low level control loop and self preservation behaviors to handle movement in the field of operation, while also having a communication loop that would continuously transmit position data to the base station. As objects of interest were sensed in the environment, the recognition of the object was also sent to the base station.

Along with the vehicles, a cloud environment existing within Google Cloud was used with the ability to control or view through a Google Cloud App. The cloud environment existed for two purposes. Firstly to share information that the swarm had collected with an external system running a Google Cloud App program displaying the information onto a Google Maps representation, thus allowing human observers field situational

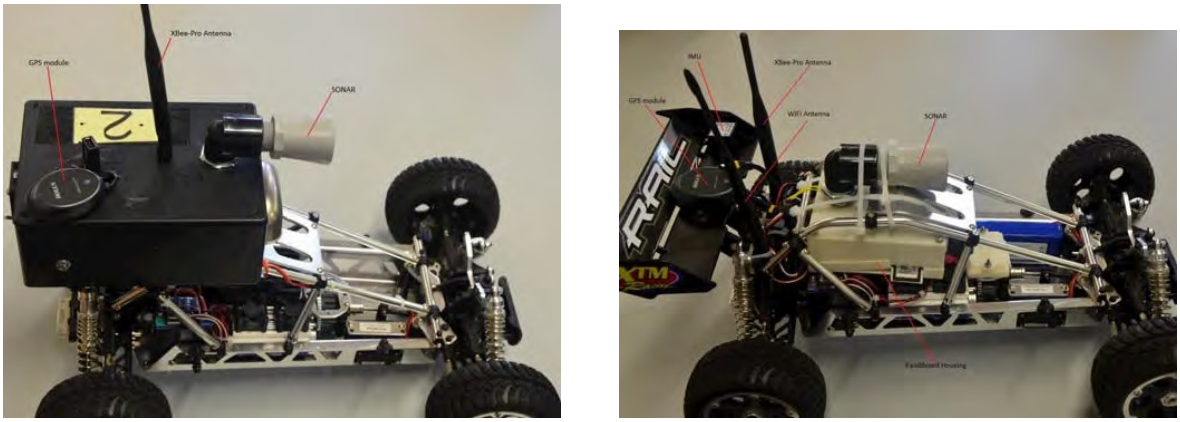
awareness. Secondly this system allowed mission parameters to be decided upon and then sent to the swarm. All data collected by agents in the swarm was transmitted to the base station and subsequently relayed to the cloud environment. All cloud data was then displayed in real time through a human interface using a 2D model of the field of operation. The human interface allowed users to control the swarm as desired by changing the individual parameters, authorizing, or altogether canceling the missions.

Along with creating the physical swarm of vehicles, work was performed in developing a mesh network topology in order to handle communications with individual UGVs and the base station. Scenarios were reviewed in terms of how low or lost network signal situations could affect the swarm as a whole. One mesh healing technique used to repair the strength of communication involved halting a vehicle's current mission progress and returning to the last known centroid of the swarm. In situations where the vehicle lost signal completely, the vehicle would follow the previous described mesh healing method while the entire swarm would stop their mission progress and travel to the last known position of the vehicle with the lost signal. Given the inefficiency of this method, a final method, ”Leave No Man Behind”, was explored. In this method, the vehicle with the lost signal would remain in place and a small group of vehicles were dispatched from the swarm to find the lost vehicle. Though each method is discussed in detail, the least efficient method of handling lost connections in which the agent with the lost signal returns to the last known position of the base station was used.

During experimentation, a mission was sent to the cloud environment and the base station would receive the details. Once the agents of the swarm received mission parameters and began the mission, they would attempt to randomly travel around the environment to complete the mission. They self-formed a grid pattern while ”bouncing” away from each other according to avoidance algorithms.

3.1.2 Further Advancements for Homogeneous Swarm of UGVs

The system was developed further once it was fully inherited. Time was initially spent modifying the code base and the first prototype, as seen in Figure 3-2A, of the agents was slightly overhauled to clean up some of the issues found during previous testing, while plans were made as to what the new phase of the system would be [4, 32]. Vehicle control was changed from a simple GPS heading system which resulted in a serpentine driving pattern to an Inertial Measurement Unit (IMU) based system with Proportional Integral Derivative (PID) control to better estimate position and orientation between GPS readings and resulting in better controlled steering.



A B

Figure 3-2. XTM Rail Prototypes used in Swarm System: A) XTM Rail Prototype 1 B) XTM Rail Prototype 2

Due to capability limitations, some hardware modifications were made to the vehicles. A faster CPU capable of handling sensor and vehicle state more efficiently was added. All equipment on the vehicle was moved from a top heavy container to within the roll cage of the vehicle. Wi-Fi was also added to the vehicles to allow for a Mobile AdHoc Network (MANET) when in range of other agents or the base station, thus allowing for a better bandwidth versus the previous serial connections made via XBee RF Transmitters.

Tests were been performed with both the new prototype (Figure 3-2B) and old prototype forming the swarm. The same resulting behavior from previous tests was observed, however, the control and capability of the newer prototype was proven to be more efficient at accomplishing tasks and vehicles would accomplish desired trajectories more seamlessly.

3.2 Autonomous UAV in ROS Environment

Upon completion of the UGV based homogeneous swarm project, another project was started with the focus of developing a system where a UAV could be controlled via commands given by an Android tablet in a 3G Mobile network. There were two main focuses in the development of this system: Firstly to develop a better understanding of ROS in terms of communication, control, and computer vision. Secondly to develop a UAV that could be given a mission in terms of waypoints, then travel to those points while receiving imagery data. The project was split into four major developments: Preparation of the UAV system, developing the ROS control system for the UAV, the ROS components controlling vision, and the Android Application for sending and controlling missions.

3.2.1 Background in ROS Environment

ROS is an open-source framework for copious robot platforms containing various tools, libraries, and conventions to assist in rapid development of various complex behaviors while providing an operating system-like environment. Originally developed by Stanford Artificial Intelligence Laboratory and currently under Willow Garage, a large community of companies and hobbyists now pull together to create open-source software within ROS to assist in communicating with devices, controlling vehicles, vision and mapping, and many other areas of interest.

First and foremost, ROS provides a message passing framework to allow multiple processes within a system to communicate. By forcing users to implement clear interfaces defined in the message Interface Description Language employed within

ROS, specific parameters and types are used to define a simple *Message*data structure. To handle communication between processes, *Topics* are defined as anonymous buses which pass message data structures between processes running under a primary ROS Master database. Being anonymous, topics are independent of who creates and who consumes the information transmitted. A process may publish a topic using the *publisher* interface, while another process may subscribe to the topic using a *subscriber* interface.

Another method of communication between processes follows a request and response paradigm. Opposite to message's many-to-many one-way transport, ROS allows a Remote Procedure Call (RPC) structure through the distributed process environment with the use of *Services*, which are split into two messages that define the request and reply components. Running within ROS, a "Service" process will present a service under a specified string name using the reply message type, while another "Client" process calls the service by calling the service name using a request message. The Service-Client communication is two-way, so that once the service has completed the desired task, a response is sent back to the calling client.

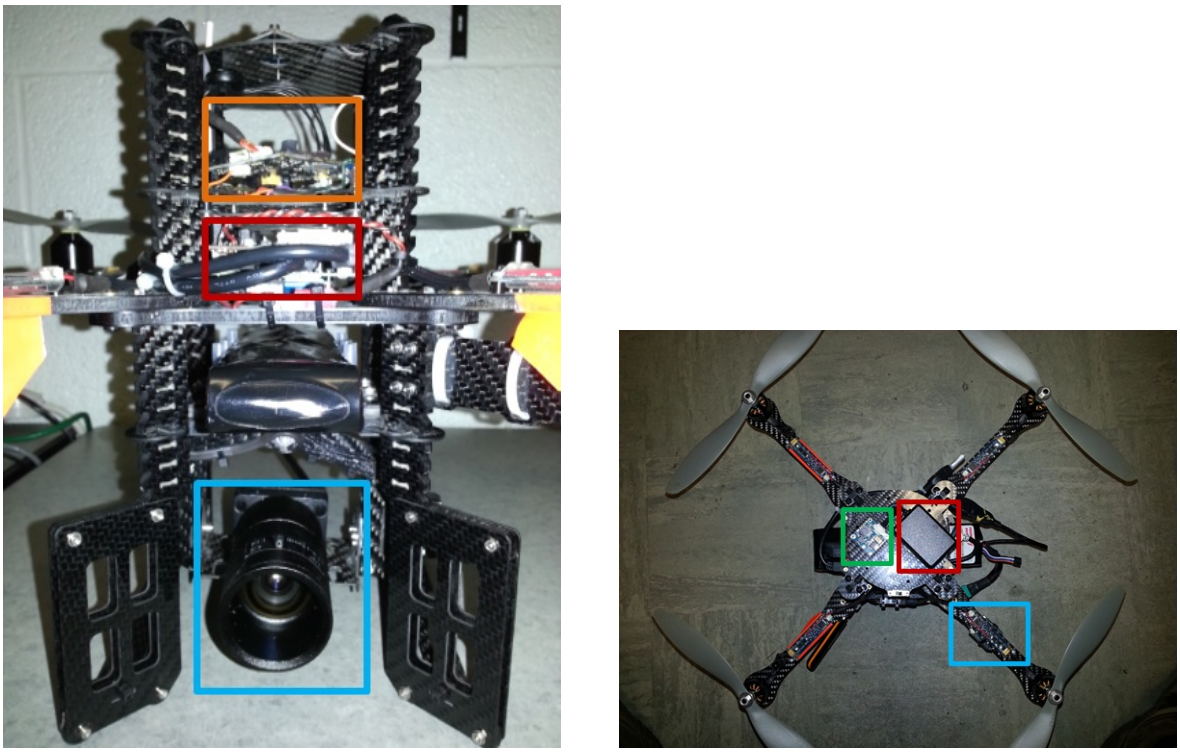
Processes in ROS are typically defined as *Nodes*. Each node, when started, connects to the ROS Master which provides naming and registration using a *nodehandler*. ROS Master combines nodes into a graph structure, assisting nodes with finding each other as well as streaming topics, RPC services, and an internal parameter server. Nodes are intended to control a specific process, for example, a driver for a laser range finder, a behavior to generate an obstacle avoidance path, or a driver to control motors dependent on a specific input.

All software in ROS is organized into packages. Packages are typically categorized to handle a specific component or behavior while being lightweight and easy to reuse under various projects. A Package may contain nodes, messages, services, libraries,

configuration files, or anything else that may be needed to create a specific component of a design. ROS also provides the ability to group

Under ROS, all nodes must connect to a single ROS Master. In the majority of multiple vehicle systems, a ROS Master/Slave configuration is used. In this case, a single ROS Master is created and all other systems are treated as slaves that are configured to connect to the single ROS Master. In the situation where the ROS Master is lost, the nodes running on the ROS Slave configuration cease to function.

3.2.2 UAV System



A B

Figure 3-3. Top and Side Views of the AscTec Pelican Used For Project: A) AscTec Pelican Side View B) AscTec Pelican Top View

The UAV used for the project, an AscTec Pelican, can be seen in Figure 3-3. Figure 3-3A shows a side profile of the AscTec Pelican. The top section designated by an orange box shows the in-house AscTec Autopilot system created by AscTec that handles low-level control such as attitude, altitude, and loiter control. Command protocols are

sent to the AscTec Pelican via a serial link between the autopilot system and AscTec Atomboard housed within the UAV body, designated by a red box in Figure 3-3A. Ubuntu and ROS were installed onto the Atomboard to handle high-level behaviors. The final component of the side view, shown in the blue box in Figure 3-3A, is the FireFly MV Mission Vision Camera system by PointGrey.

Figure 3-3B shows a top profile of the AscTec Pelican. All three devices designated by the green, red, and blue boxes (GPS, magnetometer, and Electronic Speed Controller respectively), are directly connected to the AscTec Autopilot. The GPS and magnetometer supply the Autopilot with sensor readings for Loiter and Attitude control, while the Electronic Speed Controllers are used to control each motor.

3.2.3 ROS UAV Control System

Communication, control, and data collection on the high-level Atomboard is handled within the ROS environment through multiple nodes. The main node for the UAV, ”AscTec_Drivers”, was initially created by Community College of New York for their research in 3D indoor mapping with micro-UAVs [6, 24]. The node originally provided manual control and basic status messages, however, it was later modified to include waypoint control and various status messages needed for the project. The update implemented allowed a single waypoint to be received through a ROS message that is then sent to the Pelican as the current desired waypoint. The node also received launch, land, and go to waypoint commands that sent the Pelican on its way to each desired waypoint during the mission.

The ”Vehicle_Control” node was created as a bridge between an external control program such as, but not limited to, an Android Device and the AscTec Pelican. The node had a component that received a mission in the form of a waypoint list through ROS from the external control program. Commands in terms of starting, pausing, modifying, or stopping the mission can also be sent to this node. The other components included in the node were an information loop linked to an external control program

and a high-level arbitrator that was used to decide whether the current waypoint had been reached and a new one established. A final component of the node received and transmitted sensor and vehicle state information from the UAV to the external control program for situational awareness of the vehicle and mission progress.

The ”Vehicle_Control” node was made specifically to be as neutral as possible, thus allowing it to work with any UAV Driver node that would receive basic commands and a waypoint for navigation. It is accurate to note that the stability of the vehicle when flying autonomously is owed mostly to the capability of the AscTec Autopilot while the efficiency of the mission control is owed to the ”Vehicle_Control” node. Stability of other vehicles that use the software will rely directly on that vehicle’s Autopilot.

3.2.4 ROS Vision Control System

Vision control was a separate component of the UAV and was tasked to collect mission imagery from a FireFly MV Mission Vision Camera system using a firewire based 1394 driver package within ROS to drive the camera. The ”Mission_Logger” node was then created to log flight data and store imagery. Flight data was logged by receiving status information of the vehicle at a configurable rate. Status information such as vehicle state was logged into a CSV file for playback and mission review. The second purpose of the system was to time and gps stamp images while saving them, also at a configurable rate.

Both of these components stored information to a ”black box” that could be removed and replaced at the end of the mission. All data could then be placed into the external control program and be used to playback the mission and review all collected information, or be processed into computer vision algorithms to extract data such as 3D information on the field of operation.

3.2.5 Android Application for Mission Control

The interactive component of the system was contained within the external control program. An Android based application was made by combining Google

Maps, ROS-Java, and a database and control system to hold mission plans. The application, shown in Figure 3-4, allowed a user to plan a mission via a waypoint list and desired movements or settings to make at those waypoints. The left side of the application provided the user a Google Maps interface to enter a default waypoint by simply touching that point on the interface. By manipulating the waypoints as they were created, a complete reconnaissance mission with directional control could be created. The user could then use the Android application to provide mission control over the connected vehicle, in this case the Pelican UAV.

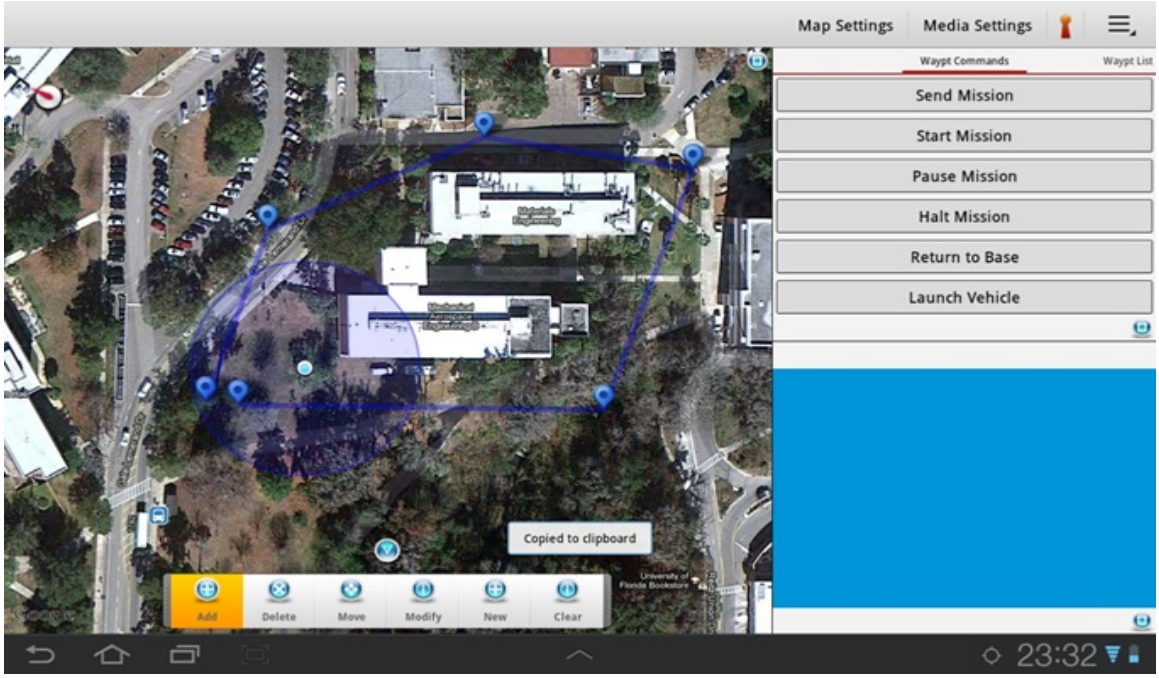


Figure 3-4. Android Application for Mission Control

The ROS-JAVA environment on the Android tablet and the ROS environment on the Atomboard initialized communication protocols running in background threads, while the ”Vehicle_Control” node handled mission control and receipt of commands. Once a mission was underway, the top right window of the application could be scrolled in order to view more accurate information in terms of mission status. The bottom right window of the application streamed either captured images when connected in 3G mode, or video when on a higher bandwidth Wi-Fi connection.

3.2.6 Testing

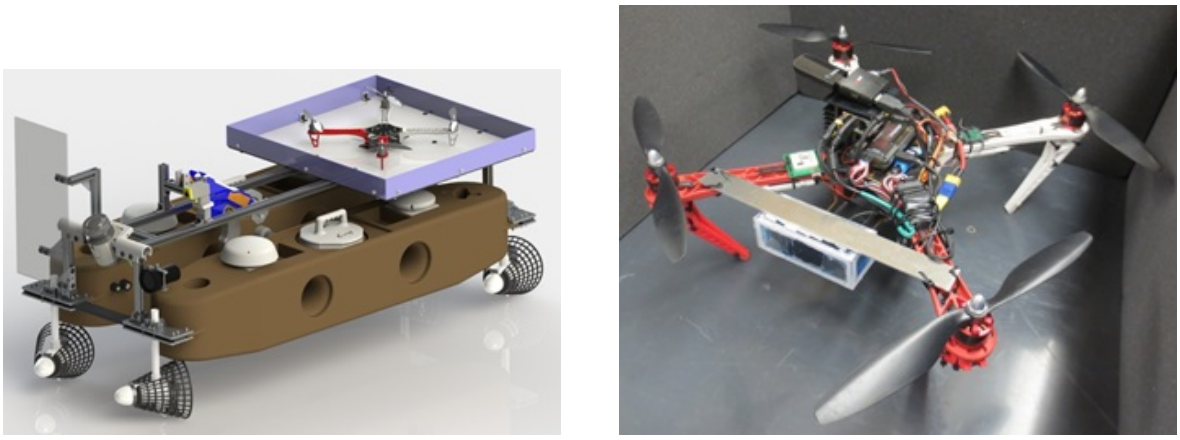
Multiple tests of the system were performed to measure the ability of the system when handling a complete mission. Mission commands such as loiter, user-defined lawnmower search patterns, building reconnaissance, and random patterns were generated to demonstrate the vehicles capability to follow a pre-planned mission. Vehicle data and imagery was collected throughout all missions, and once a mission was completed, the vehicle would land, the ”black box” was switched out for another, and a new mission was started. While the new mission was underway, the previously used ”black box” was then used to play the previous mission on the Android Application while allowing the ability to switch between mission playback and current mission view.

3.3 UAV Performing Autonomous Landing on USV

A follow up project and a joint exercise with the autonomous USV within MIL, PropaGator, was to use the autonomous UAV to assist the USV in the RoboBoat competition [34]. In preparing for the RoboBoat competition in which various tasks must be completed by the USV, one such task required the use of an external platform to leave the USV and retrieve a puck. The project involved the development of a cheaper UAV with respect to the AscTec Pelican as well as further advancements in vision to assist in completing the task.

3.3.1 USV and UAV system

A model of the PropaGator USV used for the RoboBoat competition is shown in Figure 3-5A. The UAV used for the project was a quadrotor using a commercially available DJI Flame Wheel F450 frame. Off-the-shelf parts were selected for the UAV to minimize the cost of the overall system, as well as to promote opens-source collaboration. The ABS landing gear of the UAV was designed so that in the event the UAV crashes over land, the landing gear will absorb most of the impact and break off, leaving the rest of the frame with minimum damage. Figure 3-5B shows the physical UAV used in this study.



A B

Figure 3-5. USV Cad Model and Flamewheel F450 Quadrotor for RoboBoat Competition: A) USV Cad Model B) Flamewheel F450

Basic stabilization and navigation was handled with an ArduPilot Mega (APM) 2.5+ autopilot control board. Control and sensor processing abilities were provided to the UAV via a quad-core ARM ODROID-U2 installed with Linux-Ubuntu and ROS. The ODROID retrieved information such as current orientation and GPS location from the APM. The ODROID was also capable of sending the APM information such as a desired attitude or GPS waypoint, similar to the earlier project with the AscTec Pelican. The main sensor of the UAV was a Logitech HD Pro Webcam C920 camera which was used for computer vision as well as obstacle and target identification. Communication between the UAV and USV was handled by XBee RF devices which have a range of approximately of 300 feet.

3.3.2 ROS USV and UAV Control System

Both the USV and UAV make use of the ROS environment for control. Since all nodes created are started with and must have access to a ROS Master Core, a loss of connection to such ROS Master would result in all nodes and controls shutting down, effectively stopping the vehicle from functioning. Each vehicle is therefore set up as a ROS Master to avoid this failure. Though each vehicle is independent and is the master

of its own instance of ROS, vehicles are able to communicate via RF so that they can cooperatively plan their desired tasks to complete the mission.

Communication, control, and data is all handled within the ROS environment through multiple nodes. The *ROS_Bridge* node runs separately on each ROS Master, handling communication between the USV and UAV within the overall ROS environment (Figure 3-6A). *ROS_Bridge* is configured using specific parameters which define desired topics and services to be shared between multiple ROS Masters. Once a topic or service is modified with new data within a ROS Master, *ROS_Bridge* acknowledges the change, and broadcasts the resulting data via RF communications. Other RF devices then receive this data, and subsequently update their own ROS Masters. Within this research, *ROS_Bridge* primarily handles mission objects in the form of desired GPS points as well as mission status information.

Direct UAV control and stabilization is left to the APM autopilot. However, mission navigation, as well as direct control for visual servoing, is handled via the *APM_Driver* node. The APM allows external devices to communicate with the onboard autopilot software via a standardized Micro Air Vehicle Communication Protocol called MAVLink. Specifically, the *APM_Driver* uses MAVLink to receive attitude and status information from the APM which is then published to ROS. Other nodes may specify mission objectives in the form of GPS waypoints which are sent to the APM for waypoint navigation. Commands may also be sent that result in the APM arming, disarming, launching, landing, or modifying stabilization modes. During visual servoing, the *APM_Driver* node is used to send roll, pitch, yaw, and altitude adjustments to the controller.

Mission planning, control, and communication with a device is managed through the *Agent_Control* node, the successor to *Vehicle_Control* created for the AscTec Pelican. *Agent_Control* acts as an interpreter and mission controller for vehicle driver nodes (Figure 3-6B). Sensor data such as IMU, battery voltage, and GPS information

are retrieved from a driver, such as *APM_Driver*. The data may be modified and then published back to ROS as desired. Subsequently, the node performs as the mission planner and control system.

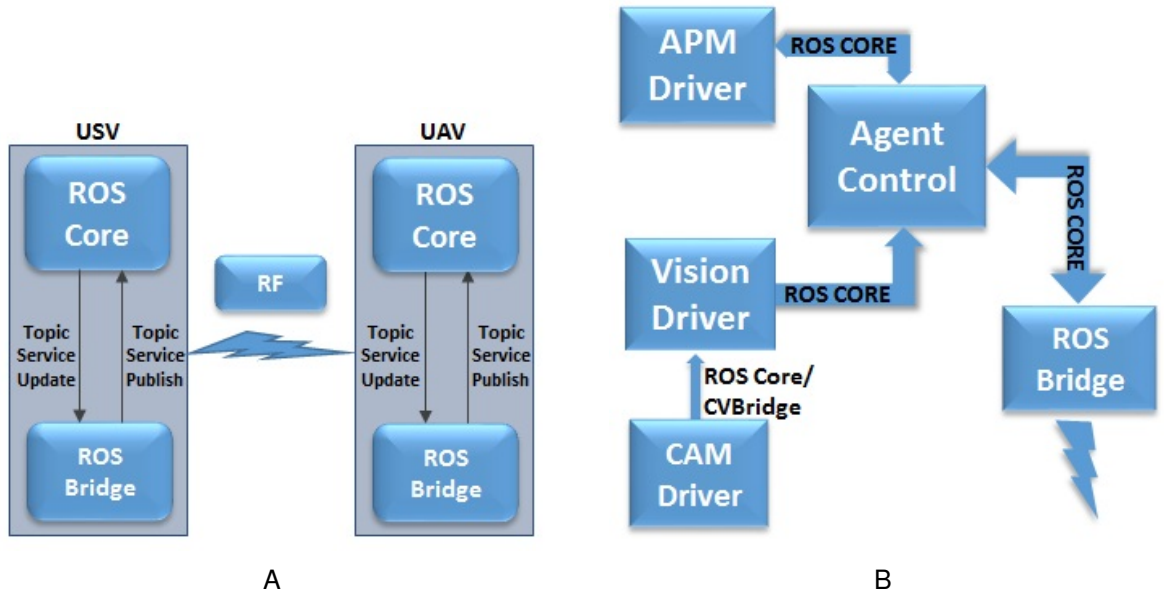


Figure 3-6. *ROS_Bridge* and *Agent_Control* Data Flow: A) *ROS_Bridge* communication between vehicles B) Data flow with *Agent_Control*

The mission state system uses data retrieved from the APM, external sources such as the USV, waypoint lists, and tracking data from vision. Based on the state of the mission planner, waypoints and commands may be sent to the *APM_Driver*. Through a combination of the *APM_Driver*, *Agent_Control*, and ROS, many types of multirotor aerial vehicles may be controlled.

Visual processing is handled within the *Vision_Driver* node. Images are collected from the camera through another node, *CAM_Driver*, and are then published to ROS using standard image messages. Images are transferred to a format usable in OpenCV by *CVBridge*, a ROS library which acts as an interface between ROS and OpenCV. The *Vision_Driver* node processes the incoming image, extracts the required data, and then publishes target data of the item being tracked to ROS. There are various other smaller nodes that are used to assist in the control or sharing of information as needed.

As previously mentioned, the USV also uses ROS for control, however, for the purposes of this research, only two components were in primary use when communicating with the UAV: *Mission_Planner* and *ROS_Bridge*. During the launch and recovery of the UAV, the USV loitered in a general position, using the *Mission_Planner* node to start the UAV mission as well as send and receive status updates during the mission. When the mission was complete and the UAV was returning for recovery, GPS positions of desired targets were sent to the UAV as well as the USVs location. *Mission_Planner* sends this information from the ROS Master on the USV to the ROS Master on the UAV through *ROS_Bridge*.

3.3.3 Visual Processing and Control

All of the vision processing performed by the UAV is computed on-board by the ODROID. The UAV is able to tell when it has found desired targets, such as a landing pad, by running an application that uses adaptive thresholding and blob detection algorithms. The application is written using OpenCV, an open-source library of programming functions for real time computer vision.

As each image is received, an initial Gaussian blur filter is performed to reduce image noise. The resulting image is split into individual R, G, and B channels as well as converted to HSV and Lab color spaces. Adaptive thresholding and modification of each color space channels is used to create various color filters tasked to detect colors of the desired targets to be tracked. Dilation and erosion morphological functions are used to further reduce noise and fill in gaps of the thresholded image.

The resulting filtered image is processed further to detect blobs and contours in the image. Each blob found results in a calculated area, perimeter, centroid, and shape declaration. Blobs are reviewed to meet certain criteria in terms of minimum and maximum sizes as well as closeness to square or circular objects. Depending on the desired target, certain aspects of the image are extracted.

To allow for robustness, a Kalman Filter is used on the final image to allow for accurate tracking in case of obstructions or short loss of the desired target. The filter allowed the UAV to return to an expected area if it potentially overshot the desired target.

Three red squares forming an isosceles triangle are used to designate the desired landing place on the USV. The centroid of each square is located using vision. If only one or two squares were located, the centroid of either an individual square or both squares combined were used as the desired target for the UAV. Once all three squares were located within the image, the length between each square is calculated to understand the shape of the triangle. The centroid of all three squares combined is used as the desired target location, while the shape of the triangle is used to designate the desired heading for the UAV to land. Figure 3-7 shows an example of this procedure.

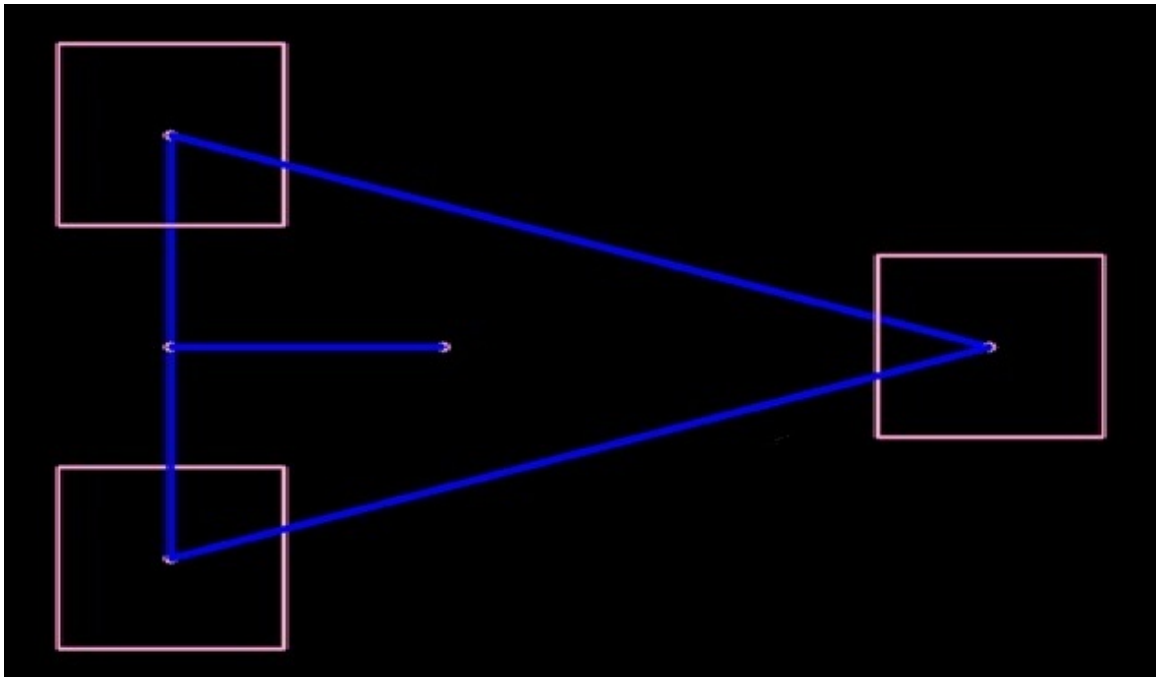
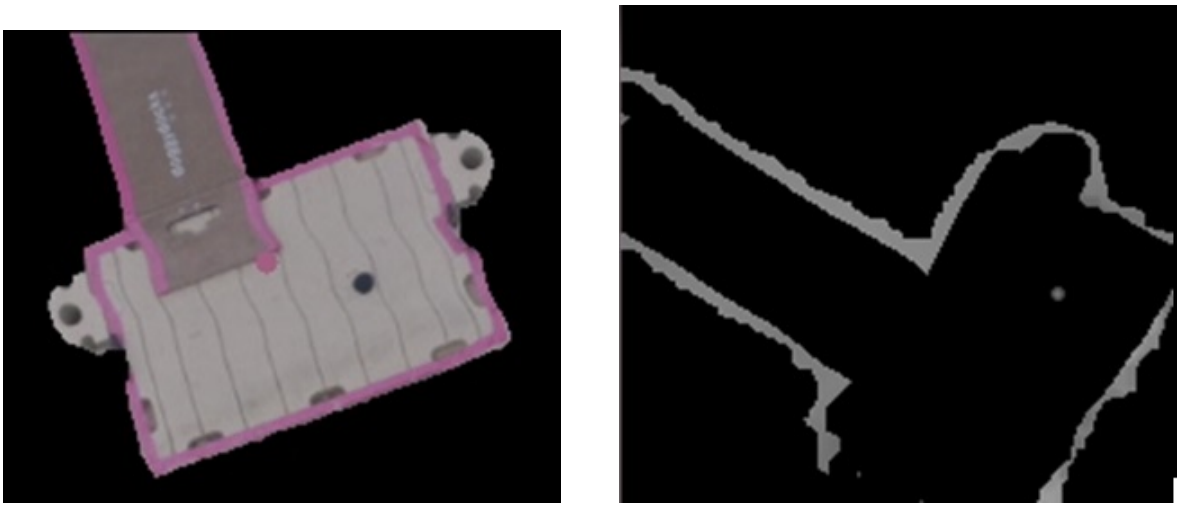


Figure 3-7. Processed Image of Landing Pad

Another type of target, in the form of a landing dock, holds a small circular object, or hockey puck. The landing dock is also designated by a colored pink border around the dock. When attempting to locate the landing dock, the colored border is first recognized and used to extract a region of interest (ROI) from the image, as shown in Figure 3-8A.

It is also possible to ignore the colored border and extract the shape of the dock against the foreground of the water. The new ROI is processed to look for the small object, shown in Figure 3-8B, in which the centroid is used as the desired target location. Once the centroid of the object is located, a nearest neighbor between the centroid and the edge of the dock is found, resulting in a desired heading for the UAV to land and avoid falling into the water.



A B

Figure 3-8. *ROS_Bridge* and *Agent_Control* Data Flow: A) Filtered and Extracted Image of Landing Dock and Puck B) Processed Image Showing Landing Dock Border and Puck

Once a target is located through vision processing and the resulting position of the target is published in ROS, the *Agent_Control* node handles visual servoing to align and land on the target. This is handled by a two stage PID controller that takes in target data and attitude information and results in a desired roll, pitch, and yaw adjustments.

One stage of the PID controller handles alignment of heading. Once a desired heading, depending on the type of target, is published to ROS, the yaw controller handles aligning the vehicle to the desired target. This is to handle offsets in camera placement or aligning the vehicle to a desired landing location on the landing pad.

If a desired heading is not published to ROS or if the yaw controller has accomplished its current goal, stage two of the PID controller handles roll and pitch control. A directional vector is chosen between the current vehicles position and heading in reference to the desired targets position. The vector is broken down into roll and pitch errors which are used within the stage two PID controller to adjust the UAV position. Once the UAV is above the desired target it begins to decrease in altitude. Adjustments may be made in roll, pitch, and yaw during the descent.

3.3.4 Testing

The mission planner on the USV initiated the launch and recovery mission for the UAV. The USV sent a command to the UAV to launch via *ROS_Bridge*. Once airborne, the UAV began its own mission by following a series of waypoints until it located the landing dock with its camera. The UAV then began visual servoing so that it was properly aligned over the circular Velcro-covered object to be recovered. The UAV landed on the object and picked it up using a Velcro net.

The UAV then launched with the object from the landing dock and began making its way back to the USV using GPS waypoints. Once it was close enough for the camera to locate the landing pad on the boat, the UAVs vision processing began visual servoing to orient and position itself before it began its descent. Once the boat successfully landed, the UAV sends confirmation of mission accomplished to the USV. The USV is then able to continue the rest of its own planned missions.

CHAPTER 4
HETEROGENEOUS SWARM ARCHITECTURE

In this chapter, an implementation of a heterogeneous collaborative architecture of UAVs and UGVs working together in unison in order to complete a task in a real world out-door environment is given.

4.1 Cooperative System Overview

Descriptions of each agent that compose the ground and air components of the system as well as a basic description of the base station are given below.

4.1.1 Unmanned Ground Vehicle Design

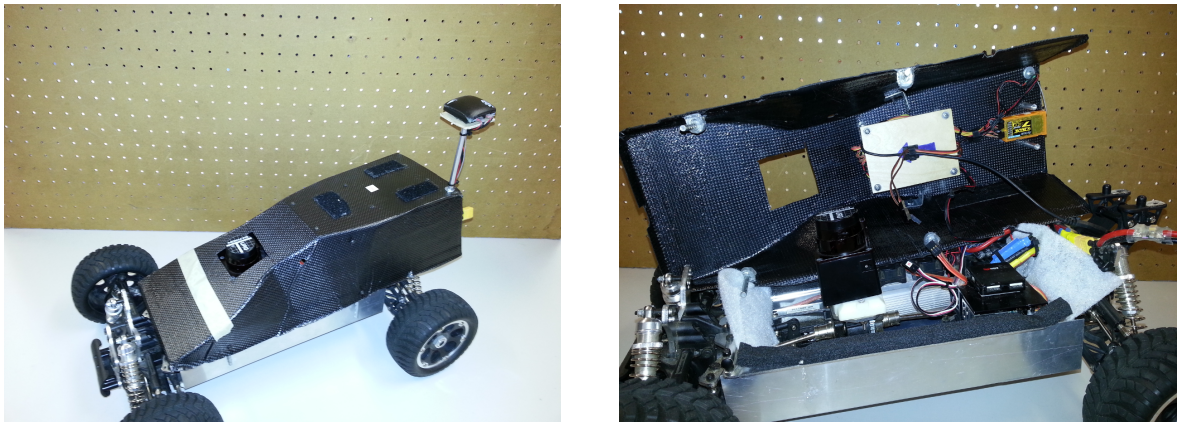
The Unmanned Ground Vehicles (UGV) that constitute ground agents in the cooperative system are a revised version of those described in Chapter 3. Each agent is a modified RC racing vehicle with a custom carbon fiber frame designed and manufactured to help protect the internally-housed electronic components. Each agent houses either an ODROID-U2 quad-core 1.7GHz Exynos ARM single board computer, or, due to updates during development, some agents are outfitted with an updated ODROID-U3, which is a functionally equivalent board to the ODROID-U2. All high level functions for the agent are performed on the ODROID, including, but not limited to, planning, direct PID steering and control, and sensor information processing.

Agent navigation and control is partially handled by an ArduPilot-Mega (APM) 2.6. The APM supports various open source applications committed to the control of fixed wing, multi-rotor, and rover type agents. It houses an Inertial Measurement Unit (IMU) with accelerometer and gyroscope sensors for positional information, and a barometer for altitude information. An off-board GPS and magnetometer are used to improve positional and orientation information. The software suite loaded on the APM is a modified version of the rover software intended to command an Ackermann based steering vehicle such as the XTM Rail. The ODROID directly interfaces with the

APM to allow sensor information to be acquired, as well as provide single waypoint or steering/throttle control commands.

In addition to the APM, the sensor suite consists of sonar for obstacle avoidance, a Hokuyo URG-04LX-UG01 LIDAR for obstacle and target identification as well as avoidance, and a Logitech C920 web camera for computer vision classification. Only some of the ground agents are equipped with cameras to allow for a local heterogeneous group of agents with different capabilities.

Communication is handled by both XBee 900 HP DigiMesh enabled RF modules as well as Wireless-N Wi-Fi. Wi-Fi is used when in range of the base station to transmit high bandwidth imagery and other data that may be useful to the base station for control decisions or human observation. Wi-Fi is also used between agents through a shared AD-HOC network that allows distribution of localization and environment data within the ROS environment. RF communication is used to send missions and commands to agents as well allow sharing of status and obstacle information.



A B

Figure 4-1. XTM Rail Final Design used in Heterogeneous Swarm Architecture: A) XTM Rail Final Design (external) B) XTM Rail Final Design (internal)

Figure 4-1 shows an external and internal view of the UGV agent.

Table 4-1 describes the three possible roles that a UGV will serve. UGVs with the **Search** role calculate a list of goals to cover an ROI in conjunction with other search

Table 4-1. Roles for UGVs Performing Missions

Task Type	Description
Search	Using a determined plan, search an area for obstacles and targets with other agents.
Identify	Using found obstacle information, navigate to each target and attempt to identify.
Path Following	Using a simple predetermined path, navigate and collect obstacle and target data.

agents. Once a list of goals is chosen, the agent then navigates to each goal while performing an obstacle and target search. A global lists of obstacles and targets is updated for other agents to use. **Identify** based UGVs use located obstacle and target information to plan a path through the environment to each potential target. Once the **Identify** UGV has traveled to the potential target, it attempts identification of the target according to set parameters. Similar to the **Search** role, **Path Following** UGVs travel along a path and attempt to locate obstacles and targets, however, the path is predetermined and provided by a operator.

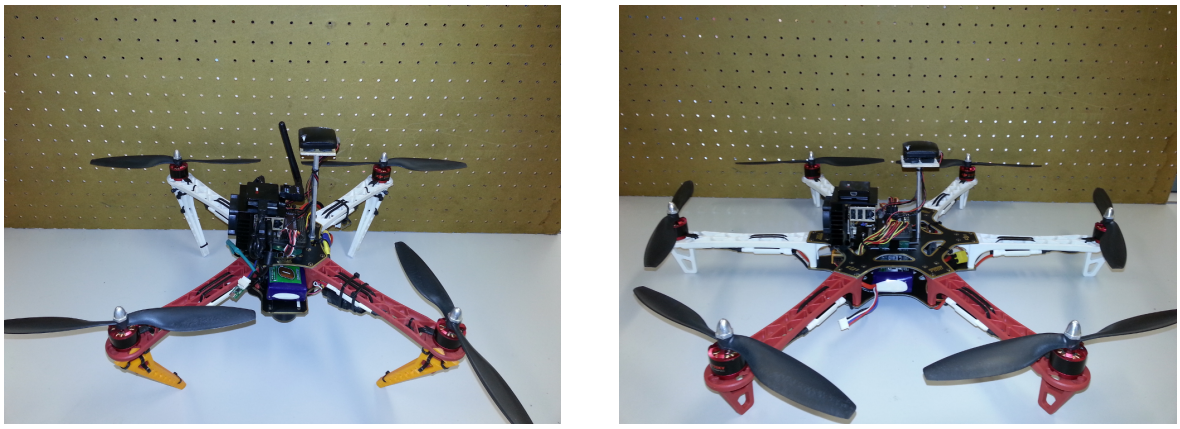
4.1.2 Unmanned Aerial Vehicle Design

The Unmanned Aerial Vehicles (UAV) that compose the air agents within the swarm use a variety of Commercial of-the-shelf (COTS) frames including the DJI F450 quadrotor and DJI F550 hexarotor Flamewheel models. The AscTec pelican described in the UAV project from Chapter 3 is also utilized within the swarm. For more added aerial diversity, two collapsible hexarotor and octorotor models were custom designed and built.

All UAVs use brushless motors powered by three or four cell LiPo batteries. Similar to the UGVs, an APM is used to handle basic stabilization of each agent. As UAVs are used as an ”eye in the sky”, the primary sensor for each is a Logitech C920 web camera, which is used for computer vision performing obstacle and target identification. All high level function, decision making, vision processing, and PID agent control is handled on board using the same quad-core ARM computer as the UGVs, the

ODROID-U2. Also similar to the UGVs, communications for UAVs is implemented with the XBee 900 HP DigiMesh RF module and Wireless-N Wi-Fi.

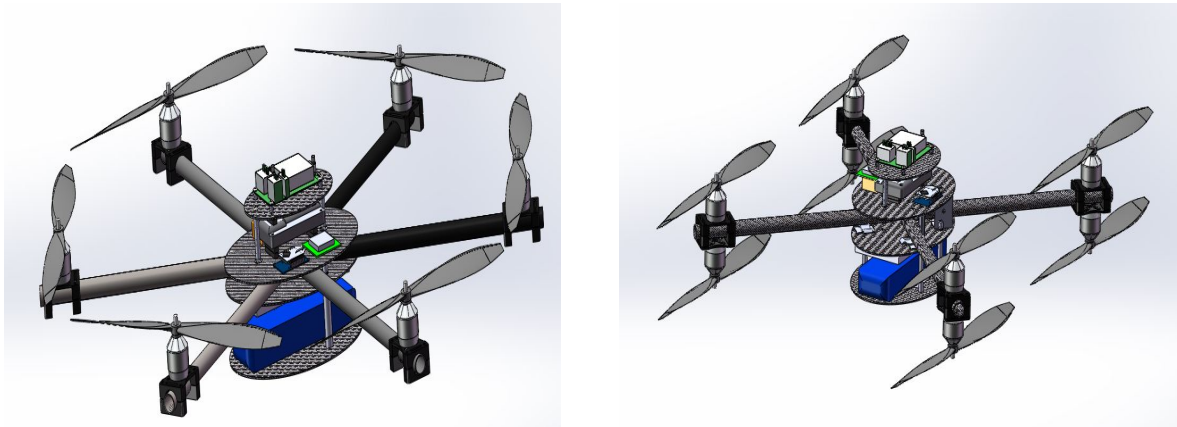
Figure 4-2 shows each Flamewheel model.



A B

Figure 4-2. Flamewheel Models used in Heterogeneous Swarm Architecture: A) F450 Flamewheel Quadrotor B) F550 Flamewheel Hexarotor

Figure 4-3 shows each custom model.



A B

Figure 4-3. Custom Models used in Heterogeneous Swarm Architecture: A) Custom Hexarotor Design B) Custom Octorotor X-8 Design

Table 4-2 describes the three possible roles that a UAV will serve. For UAV agents, the **Search** and **Path Following** roles are similar to the method employed on UGV agents, except search paths are only calculated using other UAV agents. UAVs

Table 4-2. Roles for UAVs Performing Missions

Task Type	Description
Search	Using a determined plan, search an area for obstacles and targets with other agents.
Track	Using position information for an agent, navigate to agent, located tracking target, then follow agent.
Path Following	Using a simple predetermined path, navigate and collect obstacle and target data.

employing the **Track** role are used to follow another agent, typically a UGV, and act as an extra sensor. Initially, the UAV will navigate to a desired agent's position and then locate a target on the agent using vision. The **Track** UAV will then follow the agent throughout the environment while feeding it information that may be used any way necessary.

4.1.3 Base Station Layout

A base station is used to provide a central hub for all the information on agents within the cooperative system as well as a method for control of the system. The base station is comprised of a computer, such as a desktop or tablet, with communication capabilities that mimic the UGVs and UAVs. If the collaborative system is set up in a centralized structure, missions and commands are generated on the base station and are sent to each agent. Positional data, mission status, and target or obstacle lists may be represented on the base station. Otherwise, if the system is in a decentralized structure, the base station is used to simply relay information about the mission to a operator.

4.2 Graphical User Interfaces

Graphical User Interfaces (GUIs) are used within the swarm architecture to better provide operators with information and control of the agents performing a mission. The base station provides the primary GUI, however, satellite tablet base stations running on Android tablets may be used to direct or command the swarm, as described in [33].

4.2.1 Primary ROS Qt Graphical User Interface

Within ROS, there is a package called ROS Qt (RQT). This package allows various QT Framework based plugins to be created and implemented within a single graphical interface. Each plugin exists within the ROS environment, allowing them to use ROS based functions as needed. Many of the tools within ROS may also be docked in the graphical framework of RQT, allowing for a GUI that benefits from ROS visualization tools.

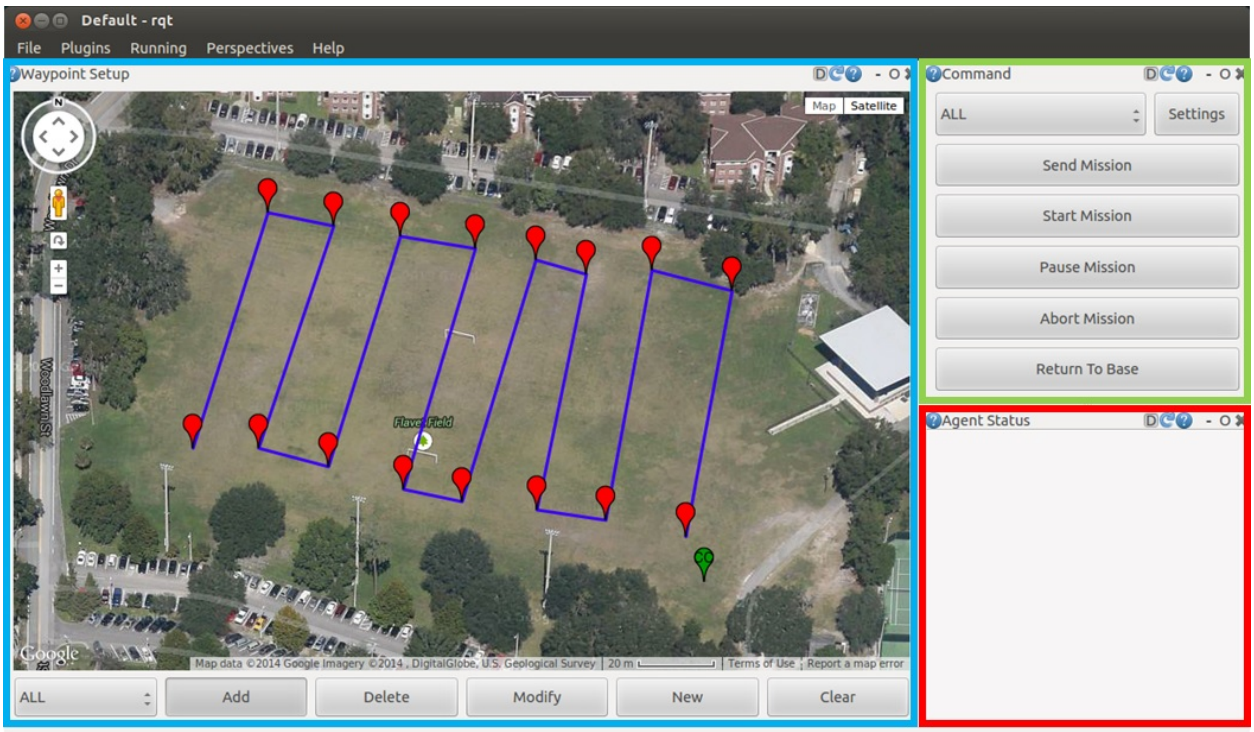


Figure 4-4. The CongreGators GUI

The CongreGators GUI is the primary interface between a operator at the base station and the agents used in the swarm. The GUI is used to prepare, manage, and command missions while monitoring agent and mission status information. Specifically made to work within a ROS environment using RQT, the GUI package is called congregators_rqt, following the framework defined by RQT, is split into various smaller plugins that may be swapped in or out as desired. Each plugin is shown in

Figure 4-4, where the waypoint, command, and status plugins are represented by the blue, green, and red boxes respectively.

4.2.1.1 Waypoint Plugin

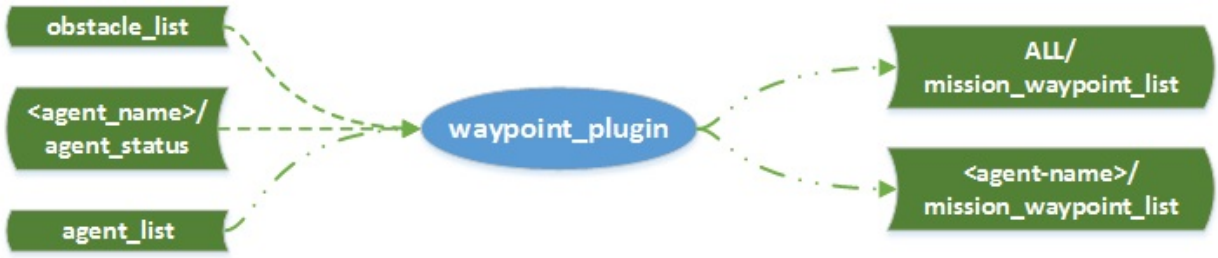


Figure 4-5. ROS Message and Service flow for waypoint_plugin

The ***waypoint_plugin*** consists of a QT webview to hold the Google Maps JavaScript API as well as a few buttons for operator input. Initially, the plugin is used to define direct waypoint-to-waypoint paths or to define a Region of Interest (ROI) that will be used in mission planning. Figure 4-5 describes the data flow in and out of the ***waypoint_plugin*** of congregators_rqt. Dashed Lines represent data that is transmitted to or from an external ROS Master, while dashed-dotted lines represent data that is transmitted to or from either an internal or external ROS Master.

Waypoints are defined by operator interaction with the plugin interface. To modify the waypoints, various options are given to the operator through command buttons: Add, Delete, Modify, New, and Clear. Add, Delete, and Modify are toggle buttons and must be activated before any change may be made through the ***waypoint_plugin***. The New and Clear buttons will allow either a new waypoint to be created using a pop-up dialog, or for all current waypoints to be cleared. Waypoint lists may be intended for individual agents or all agents, therefore, a drop down box is provided to assist in choosing what the intended target is. As the intended target is changed, all waypoints are refreshed with those from the new intended targets waypoint list. Any time a waypoint is modified, a ROS message in the form of $<$*intended_operative*$>$*waypoint_list* is generated, where $<$*intended_operative*$>$ is replaced with the agent name or the *ALL* reference.

The ***waypoint_plugin*** is also used to track and report positional status information of any agent or obstacle information which has been reported to the base station. Agent status information is received through the <*agent_name*>*agent_status* message while obstacle information is collected through the *obstacle_list* message.

Given that a priori knowledge of agents is not known, the *agent_list* message is used to provide names of all agents that have reported in for the swarm.

4.2.1.2 Command Plugin

Figure 4-6. ROS Message and Service flow for command_plugin

Figure 4-6 describes the data flow in and out of the ***waypoint_plugin*** of congregators_rqt. Data may be transmitted to or from an internal or external ROS Master. The ***command_plugin*** provides the operator with direct control over the mission. Similar to the ***waypoint_plugin***, a drop down box is provided to decide who the intended operative to receive commands is. The **Settings** button opens up a pop-up dialog for the operator to change and set specific settings pertaining to a mission type. Each time the settings are updated, the *mission_settings* message is transmitted. The **Send Mission** button triggers a service, *send_mission*, that may be used by a lower level package (such as mission_control) to trigger a combination of all current relevant mission data. The **Start**, **Pause**, **Abort**, and **Return to Base** buttons all trigger the *send_command* service that may be used by a lower level package to modify the state of the mission.

4.2.1.3 Status Plugin

The ***status_plugin*** provides a very simple interface to show an agents information once it has been received. Multiple status tabbed windows will be populated as agents come online and begin sharing information. Figure 4-7 describes the data flow in and

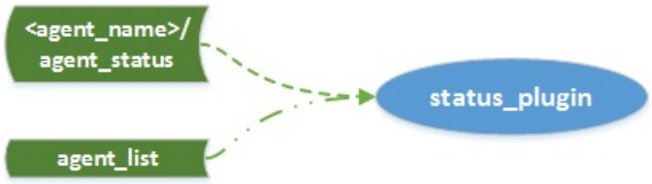


Figure 4-7. ROS Message and Service flow for status_plugin

out of the ***waypoint_plugin*** of congregators_rqt. Data may be transmitted to or from an internal or external ROS Master.

4.2.2 Satellite Android Tablet Interface

To expand controllability of the swarm, Android tablets may be used as satellite control stations within the field of operation. Android tablets provide an operator with the same capability that the CongreGators GUI provides in terms of field control and observation. While in the field, an operator may change mission plans or take control of a specific vehicle.

The Android tablets run a modified version of the application introduced for the UAV swarm project described in Chapter 3. The application's waypoint fragment has been fully rewritten using Google Maps Android API v2, which allows for a more seamless use of maps within fragment activities. Additionally, instead of connecting to the ROS network as a ROS slave device, communication is handled through RF communications to allow for the same range and network that each agent runs on, effectively turning the tablet into an agent of the swarm.

RF communication is provided by a small physical package that is attached to the tablet. The package houses an IOIO-OTG development board, an XBee 900 HP DigiMesh enabled RF module, and battery. The new android application's communication structure has been rewritten to communicate with the communications package with the capability to send and parse data between agents in the swarm. The method employed is similar to the same method that RF communications are handled on each UGV or UAV.

The input and output type data is equivalent to what is transmitted to and from the CongreGators GUI.

4.3 XBee Bridge Communications

A major focus in the swarm architecture design is communication. Within a ROS master, data is transmitted between nodes using the ROS infrastructure of messages and services, however, disseminating this type of data between multiple ROS based devices can become difficult. Nodes must have access to a ROS master to handle naming and registration, as well as tracking and administering topics of data.

It is possible to make one device in the environment the master, and every other device the slave, as was performed in homogeneous swarm testing discussed in Chapter 3. However, if the master is ever lost, all slave devices will cease to function. Consequently, in the swarm environment, each vehicle is setup as a master. Sharing information between multiple masters is a developing situation within ROS. Two solutions employed for multi-master communication within the swarm architecture include the ROS provided multi-master package and the most commonly used custom XBee Bridge package created for this research.

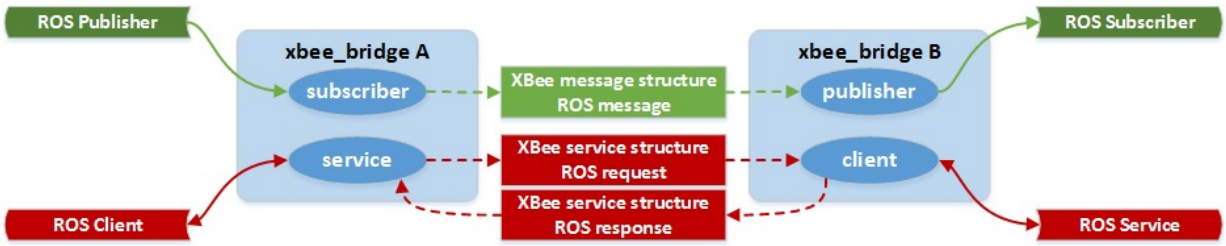


Figure 4-8. Data flow between two XBee Bridge configured devices

The XBee Bridge package is a multi-master solution used within the defined swarm architecture and serves as a significantly more advanced version of the *ROS_Bridge* node described in Chapter 3. Moving away from the limitations of Wi-Fi, long range RF XBee devices are used to share information between agents. The XBee Bridge package is structured to listen to specific messages or services on a ROS Master and

then transmit them externally to other agents, also running a ROS Master. XBee Bridge supplies the ***xbee_bridge*** ROS Python node.

Figure 4-8 summarizes the data pathways for ROS messages and services. If an agent is publishing a ROS message that is configured within XBee Bridge to be transmitted over RF to another agent, it is treated as a single direction message. XBee Bridge is configured to create a subscriber to listen for the desired ROS message. Whenever the ROS message is published on the agent, XBee Bridge packages the message, transmits it over an XBee device, and then a desired receiving XBee device is tasked with repackaging the message into a ROS message to be published on the new ROS database.

If an agent is starting a ROS client request that is configured within XBee Bridge, it is treated as a send and respond message. XBee Bridge is designed to create a matching service in place of a constructed client. When that client sends a request, it is packaged and transmitted over an XBee device. A desired receiving XBee attempts to repackage the message into a client, contact the service on the new ROS database, processes a response, and then transmits the response back to the original calling agent. At any point throughout the process a failure occurs, the original ROS client receives a failed response.

To	From	Packet #	Packet Size	Tag	Start String	Data	Checksum
4	4	2	2	3	Variable Size	Variable Size	2

Figure 4-9. Message structure for messages in XBee Bridge

To	From	Packet #	Packet Size	Tag	T/R Flag	Start String	Data	Checksum
4	4	2	2	3	1	Variable Size	Variable Size	2

Figure 4-10. Message structure for services in XBee Bridge

Standard ROS messages are transmitted in the format shown in Figure 4-9, while Figure 4-10 demonstrates how services are transmitted. When sending data between

Table 4-3. Possible IDs for who will receive the message

To ID	Description
ALLD	All devices
ALLA	All agents
BASE	Base station
TABx	Tablet with x number
AGxx	Ground agent with xx number
AAxx	Air agent with xx number

Table 4-4. Possible IDs for who may send the message

From ID	Description
BASE	Base station
TABx	Tablet with x number
AGxx	Ground agent with xx number
AAxx	Air agent with xx number

systems in the swarm, an ID is given for the receiver and the transmitter. Examples of possible IDs to be used for the receiver, or the *To* field, are given in Table 4-3, while examples of possible IDs for the transmitter, or *From* field, are given in Table 4-4. The *Packet #* and *Packet Size* fields are used to specify whether the XBee message is part of a packet with a given size, and which number in the packet the current message represents. The *Tag* field is a three character string that represents the type of data that the message holds. This component is operator defined per message type as defined later during configuration of the XBee Bridge package. For services, the *T/R Flag* field represents whether the message is a ROS service being transmitted (requested) or a ROS client responding. Finally, the *Data* field is a comma delimited list of all variables that make up the XBee message and is to be parsed into the desired ROS message or service. Each field is summarized in Table 4-5.

4.3.1 Node Initialization

XBee Bridge is setup as a dynamic package to expedite setup and remove the need for reprogramming each time new messages or services are to be implemented. Through the use of a configuration file, the communication is setup to listen to any topic on ROS as well as to define which variables within the data topic will be transmitted.

Table 4-5. XBee message fields

Field	Description
To	Four character string representing who message is transmitted to.
From	Four character string representing who message is transmitted from.
Packet #	Number representing which packet this message is in a single or group of packets.
Packet Size	Maximum number of packets in the related message.
Tag	Three character string representing the type of data the message belongs to.
T/R Flag #	One character flag representing whether message is a request or response (Only for ROS services).
Data	A comma delimited, variably sized, character string representing data of message of Tag type.
Checksum	Two character checksum result of the entire message (Only in XBee AT mode).

This allows a level of expandability that current multi-master packages do not allow.

Topics may include ROS messages or services and are defined in the setup file as:

```
topics:
    topic1:
        tag_id: <TAG>
        default_name: <NAME>
        type: msg
        package: <PACKAGE>
        module: <MODULE>
        var:
          - <VAR>
          - <CLASS>
          - <[LIST]>
    topic2:
        tag_id: <TAG>
        default_name: <NAME>
        type: srv
        package: <PACKAGE>
        module: <MODULE>
        req:
```

```
    - <VAR>
    - <CLASS>
    - <[LIST]>
  res:
    - <VAR>
    - <CLASS>
    - <[LIST]>
```

Topics in the configuration file hold various parameters. The *tag_id* parameter provides a three character string that is tied to all messages of this topic type. This *tag_id* is added to each message to control how the data is packaged for transmission or how it is parsed into a ROS topic. The *default_name* parameter holds a simple key tied to the topic that is used for the internal data structures of XBee Bridge. The *type* parameter specifies the type of package being defined by the topic, whether **msg** or **srv**. The *package* and *module* parameters are used to specify the name of the ROS package as well as the name of the ROS message or ROS service that a topic belongs to. The *var*, *req*, and *res* parameters are used to specify the which of the exact variables belonging to the topic will be transmitted. Variables and Classes are simply defined by a dash, while lists are defined by a dash with a comma delimited list surrounded by brackets.

The ***xbee_bridge*** node provides the desired functionality of the XBee Bridge package. Before starting, ***xbee_bridge*** expects that the configuration file described previously has been loaded through the use of the ROS parameter server. Figure 4-11 shows the node initialization of ***xbee_bridge***. At first, the XBee device is configured according to which mode it should communicate in. A parameter of ***xbee_bridge*** specifies whether the XBee is in AT or API mode, allowing for either serial or special API packet communication between XBee devices. If in AT mode, a simple serial connection is made for the XBee port, however, if in API mode, an imported library is used to create a connection and variable for use with the XBee.

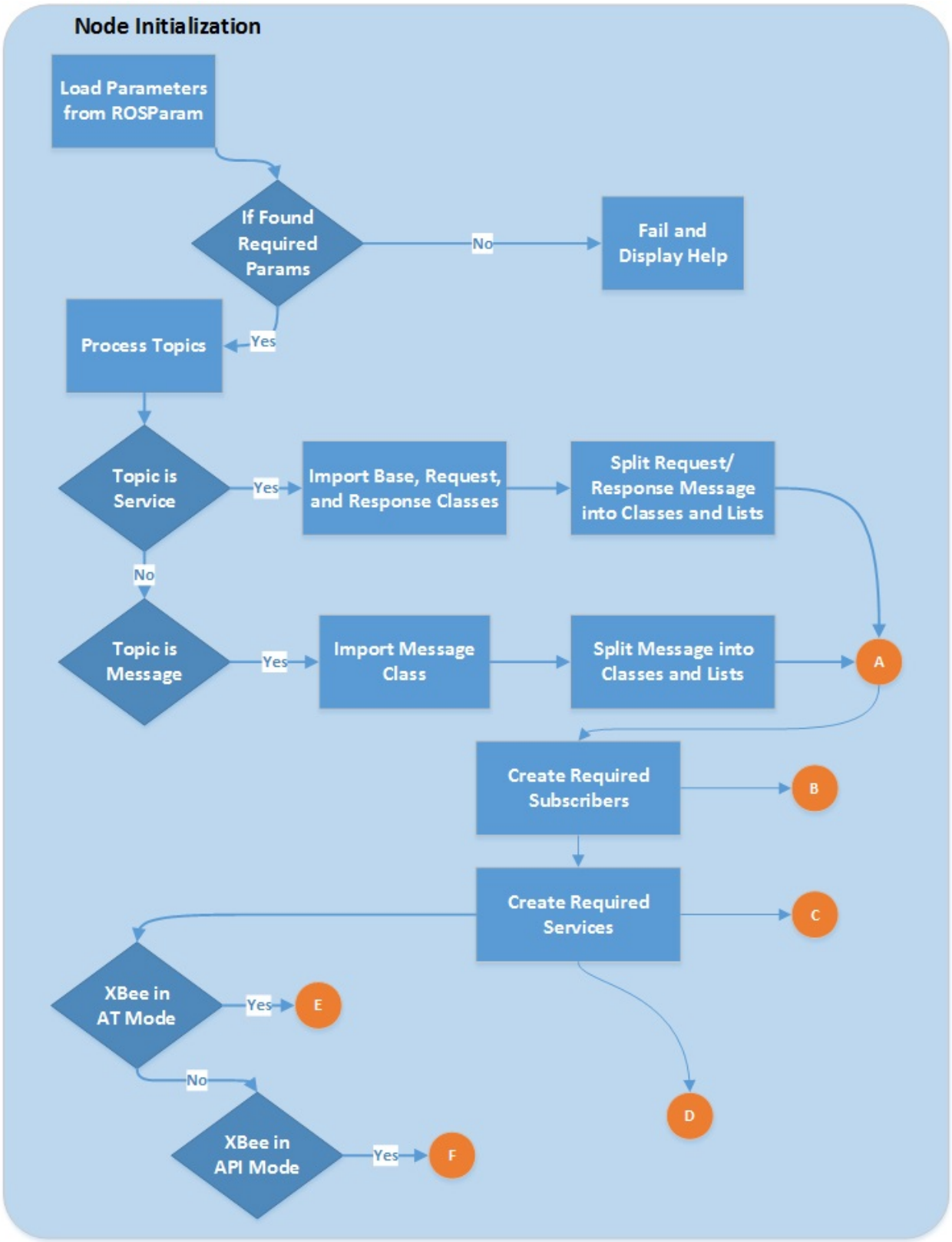


Figure 4-11. Node Initialization for xbee_bridge.

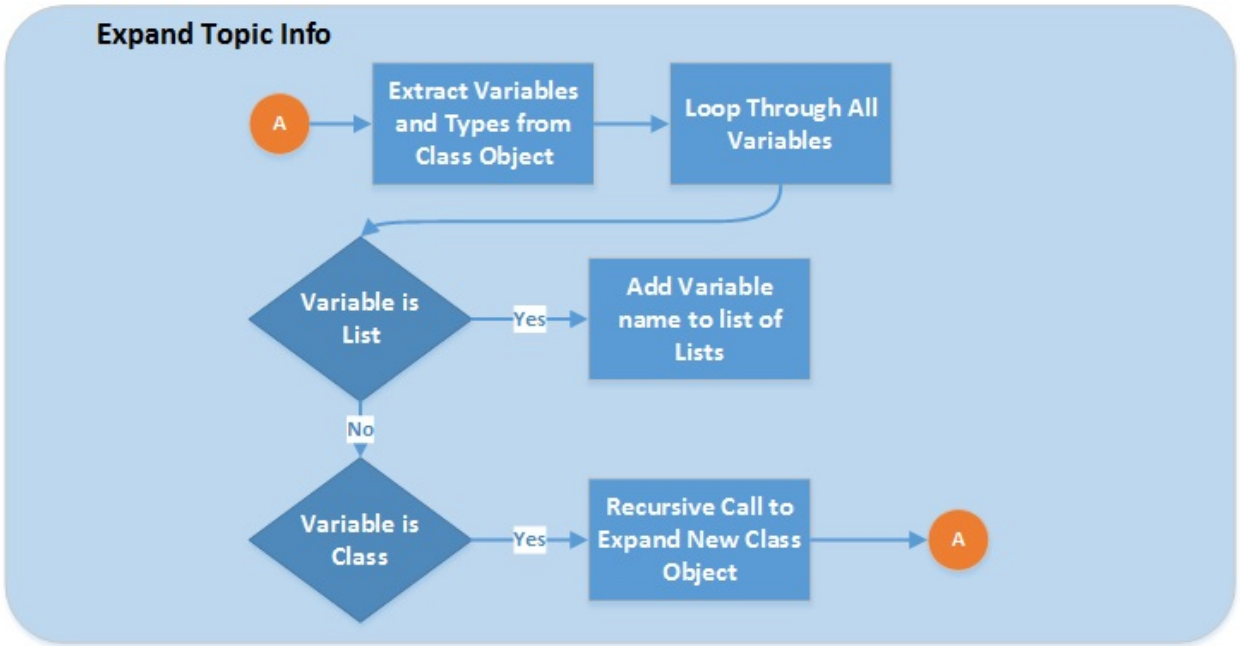


Figure 4-12. Expand topic parameters for xbee_bridge.

Next, the *topics*, *subscribers*, and *services* parameters are loaded from the ROS parameter server into python data structures. Each key in the list of topics is checked as to whether it is a service or a message topic. If the topic is a service, the base, request, and response modules are each loaded. If the topic is a message, the message module for the topic is loaded. Next each module is parsed, as shown in Figure 4-12 to find information on each variable. The *expand_topic_info* function loops through all variables that make up a module. If the variable is of type list, the name of the variable is added to a list lists. However, if the variable is of type class, the name of the variable is added to a list of classes, and then the variable is treated as a nested module and recursively called by *expand_topic_info* to be further parsed. *expand_topic_info* returns the collection of lists and classes which are then stored in the topics data structure for later use.

Once the required information on a topic has been extracted, subscribers and services are created for each variable specified in the *subscribers* and *services* parameters. With initialization complete, the node creates a callback function specified by the type of mode the XBee employs for processing the incoming data.

If in API mode, the imported python-xbee library handles receipt of a message and confirmation that it is accurate. A callback function within the library is then utilized to call the *parse_string* function.

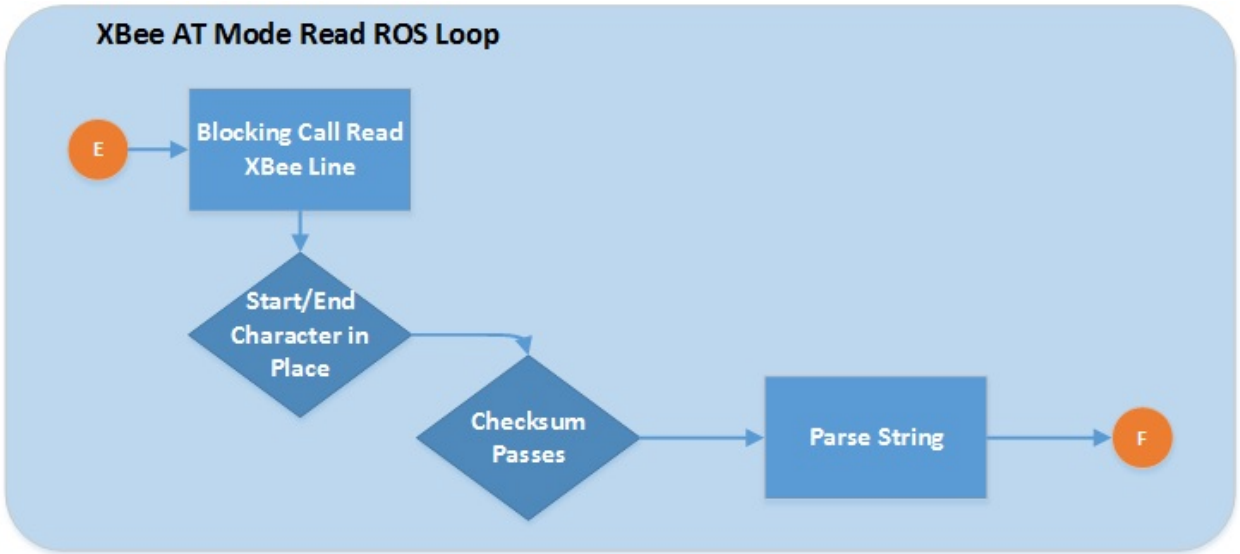


Figure 4-13. Read loop of XBee in AT mode for xbee_bridge.

If in AT mode, a ROS spin loop is created that handles the serial data received from the XBee. As shown in Figure 4-13, a blocking call to the serial port is started. In AT mode, there is no built in confirmation that the serial message received is accurate. To establish the received message as accurate, it is first checked for left and right brackets to represent a complete message. Then, a checksum is then calculated and compared to the last two characters of the serial string to confirm that all fields of the serial string was received correctly. After confirmation that the message is accurate, *parse_string* function is called.

After the read callback has been configured, the main function enters a configurable loop that monitors the state of subscribers and services on the ROS Master, as shown in Figure 4-14. As new subscribers and services are created, their names are compared to those loaded into the parameter list for ***xbee_bridge***. If a name has a partial match, it is added to the list of subscribers or services that ***xbee_bridge*** is configured to listen

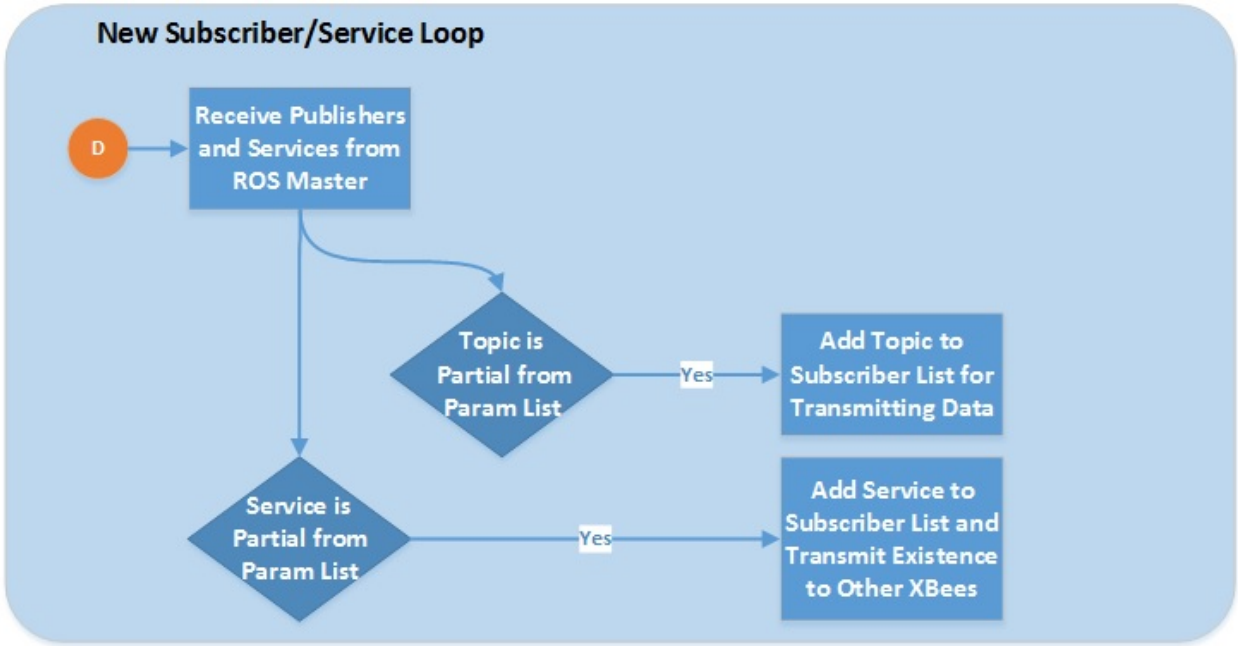


Figure 4-14. Find New Subscriber/Service Loop for xbee_bridge.

to. Any component of the topic or service name that is not registered is stored to be transmitted with the message.

4.3.2 Management of ROS Messages

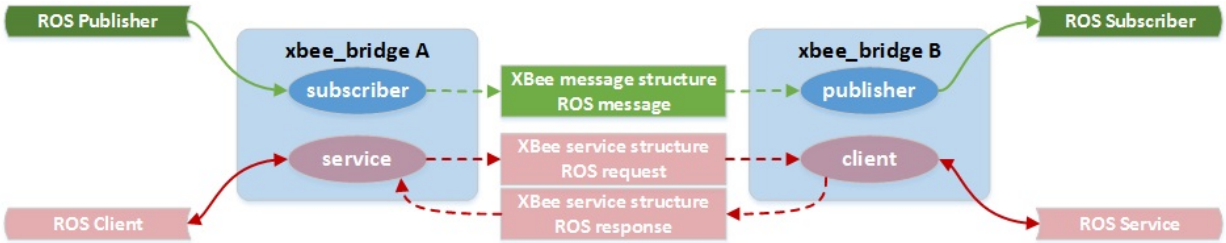


Figure 4-15. ROS message data flow between two XBee Bridge configured devices

Figure 4-16. Standard subscriber callback for xbee_bridge.

Once initialized, ***xbee_bridge*** is tasked with handling ROS messages and services. Figure 4-15 shows the primary path for data flow of ROS Messages between two ***xbee_bridge*** configured devices. During initialization, subscribers are created for desired topics. Whenever the subscribed message is published to the ROS database, ***xbee_bridge*** triggers the *standard_subscriber_callback*, shown in Figure 4-16. In *standard_subscriber_callback*, the type of ROS message and the desired target are used to form the beginning parts of the message to be sent. Next, the ROS message data structure is sent to the *transmit_data* function for formatting.

Figure 4-17 shows the *transmit_data* function, which is tasked with looping over the ROS message data structure to form a XBee message. The XBee message is formed using the *build_message_string* function, shown in Figure 4-18.

As individual components of the ROS message are sent to *build_message_string*, the type of component is reviewed. For each component reviewed, a resulting string representing the value of the component is returned. If the component is a simple variable, the value is converted into a string and returned. If the component is a list, it is assumed that there are multiple variables inside the list, therefore, they are recursively passed into the *build_message_string* function until the list is empty. All returned values of the recursive calls are combined into a single string and then returned. Finally, if the component is a class, all components of the class are recursively passed into the *build_message_string* similarly to how lists are handled.

All returned values from the *build_message_string* function call within *transmit_data* are combined into a single string. Now that a message string is collected, it is formatted according to the mode for communication that the XBee is configured for. If the XBee is in AT mode, a checksum is calculated over the entire message and is added to the end of the message string. The message is checked according to size and is possibly split into multiple smaller packets if needed. Finally, each message is surrounded by left and right brackets before being sent serially to the XBee device for transmission.

If the XBee is in API mode, the message is structured using the API XBee variable setup during initialization. The message is checked according to size and is possibly split into multiple smaller packets if needed. Each message is configured in the API packet structure for the XBee and then transmitted.

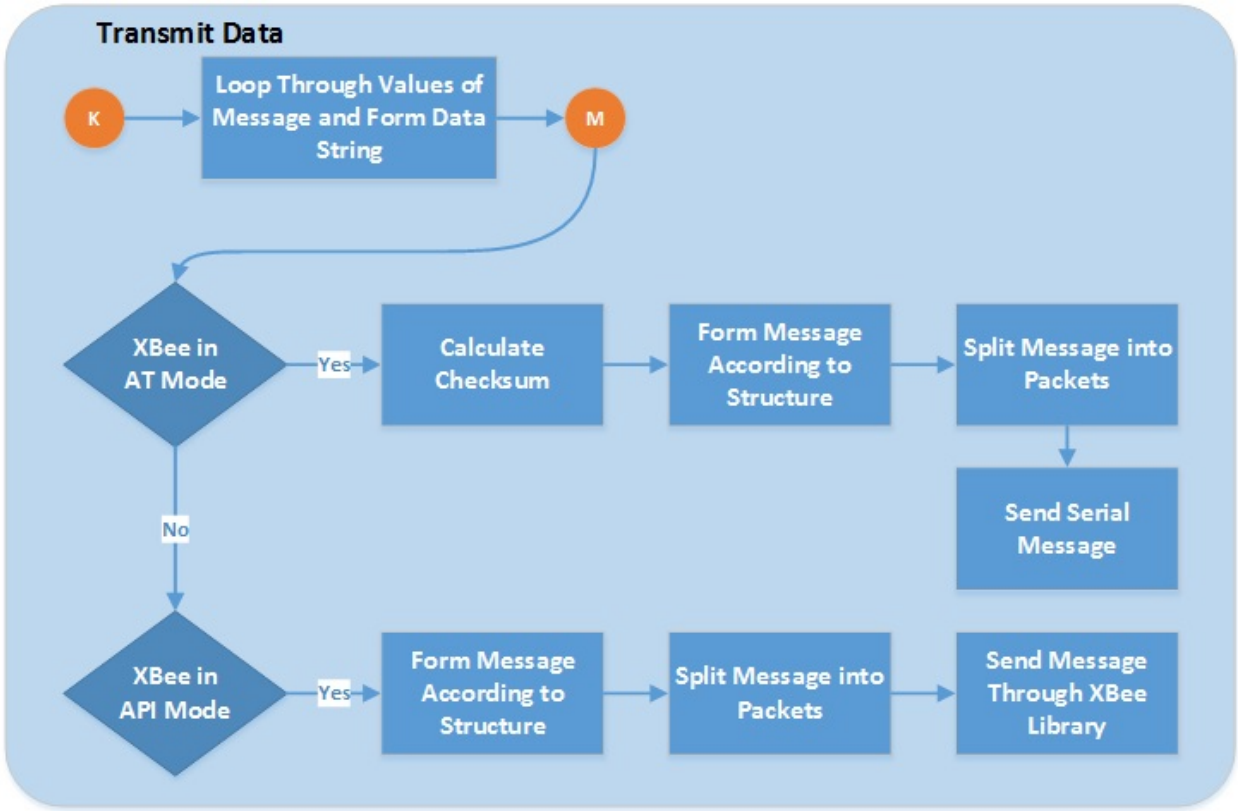


Figure 4-17. Transmit data for xbee_bridge.

On the receiving end, ***xbee_bridge*** has already been configured to confirm a message as accurate, and then to pass the data to the *parse_string* function. As shown in Figure 4-19, the *parse_string* function executes parsing serial messages into ROS messages or services. First, a message is split into components: *toStr*, *fromStr*, *currPacketNum*, *packetSize*, and data. The data component is further split into the *tagStr* and the *dataStr*. If data is part of a packet, then it is stored in a mapped variable tied specifically to a key that consists of who the message was from and the given tag

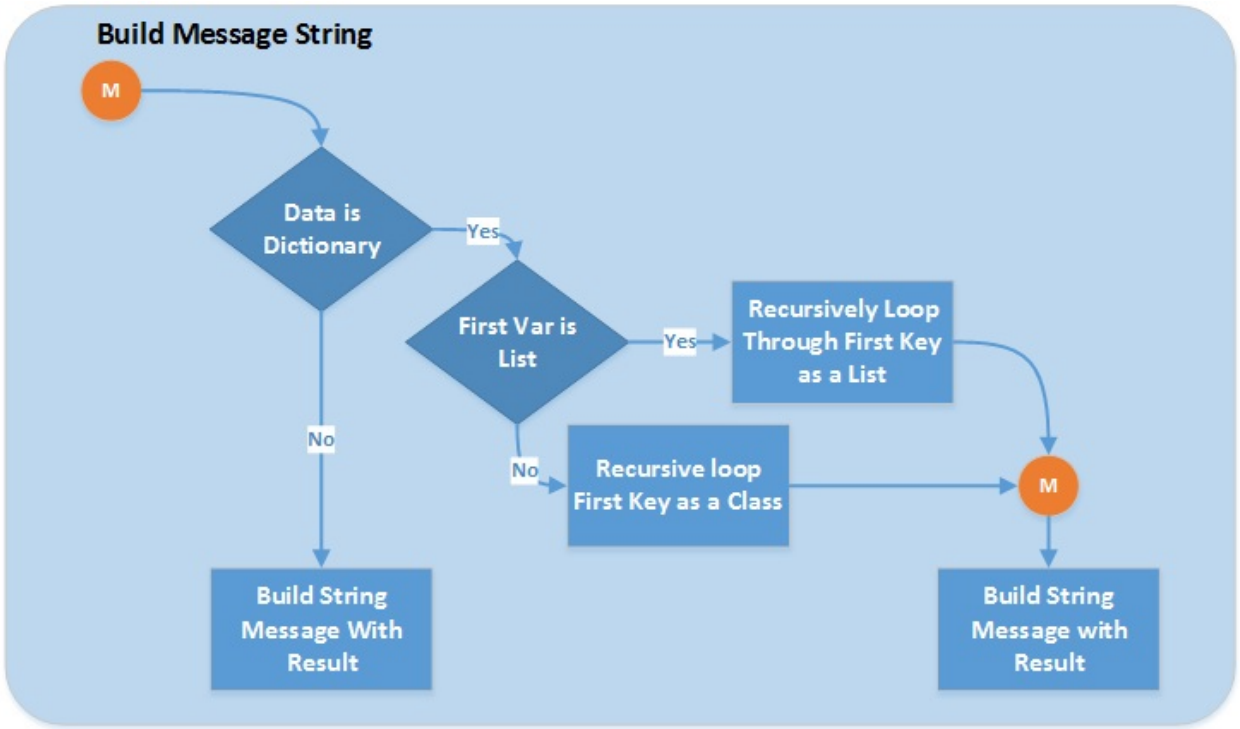


Figure 4-18. Build message string for xbee_bridge.

for the message. If the data is the last part of a packet or is a single packet, it is parsed according to the type of message.

If the message is a ROS message, an initial check is performed to see if a publisher for the topic has been created yet, and then the message data and ROS message variable data structure is sent to the *standard_publisher* function.

As shown in Figure 4-20, the *standard_publisher* function simply splits the received message string into separate components. Next, the *tagStr* component is used to extract a data structure for the ROS message from the topic data structure that was loaded during initialization. This data structure, known as *var_list*, includes a list of all variables that the parsed message string should have, as well as the order that they should be in. Along with *var_list*, relevant classes and lists are loaded from those found in the *expand_topic_info* function. Finally, a ROS message variable, overhead message, is created for the desired ROS message type. Each of these components are sent to the

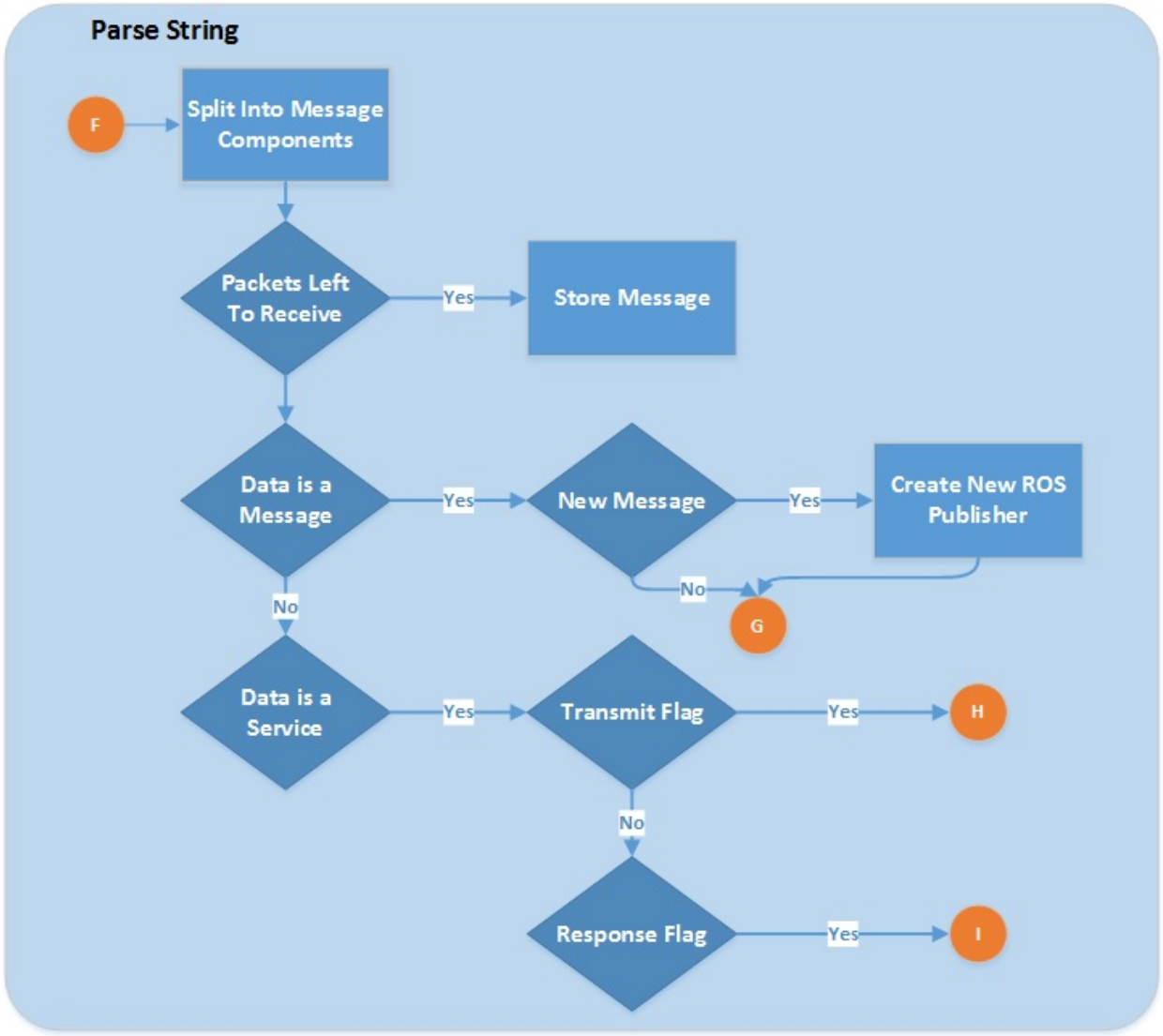


Figure 4-19. Parse String for xbee_bridge.

build_ros_message function. The *build_ros_message* function, shown in Figure 4-21, is similar in operation to the *build_message_string* function.

Initially, the ROS *build_ros_message*, checks the first component of the *var_list* data structure as a *key*. If the *key* is a simple variable, the first value of the message string is typecast into the correct type designated by the *key*, and then set as the attribute of the overhead message specified by *key*. If the component is a list, it is assumed that there are multiple items inside the list that may be a variable, nested list, or nested class. Therefore, they are recursively passed into the *build_ros_message* function until the

list is empty. A temporary class defined by the type of the list is created to hold values found from the recursive return of *build_ros_message*. As each value is returned, they are appended to the temporary class, forming a list. Once complete, the temporary class is set as the attributed of the overhead message, just like individual variables are. Finally, if the *key* is a class, all components of the class are recursively passed into the *build_message_string* similarly to how lists are handled.

Once the newly formed ROS message is returned, it is published to the ROS database.

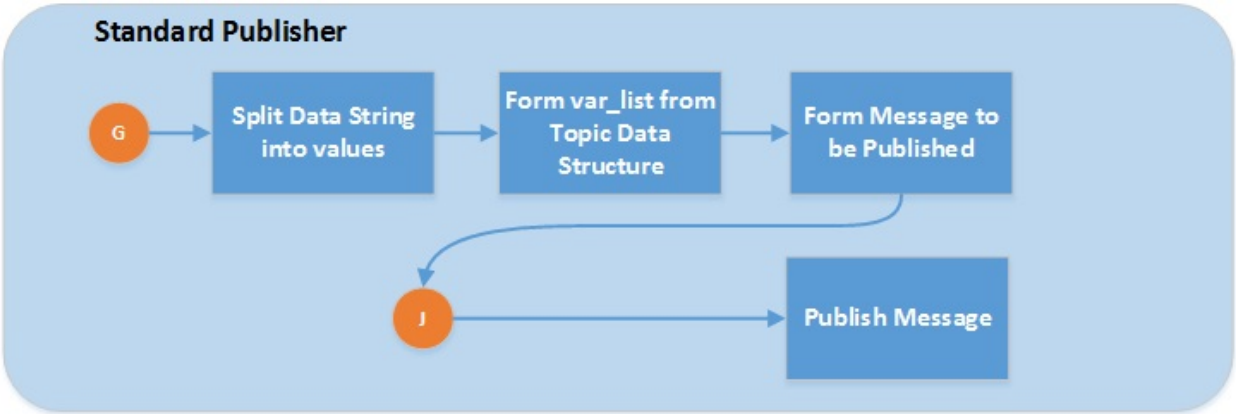


Figure 4-20. Standard publisher for xbee_bridge.

4.3.3 Management of ROS Services

xbee_bridge is also tasked with handling ROS services. Shown in Figure 4-22, ROS service transmission is similar to ROS messages, however, there is an added response path that occurs. During initialization, services are created for desired topics. Whenever a ROS client triggers a request to a registered XBee service, ***xbee_bridge*** triggers the *standard_service_callback*, shown in Figure 4-23. In *standard_service_callback*, the type of ROS service request and the desired target are used to form the beginning portions of the message to be sent. Next, the ROS service request data structure is sent to the *transmit_data* function for formatting. Similar to how ROS messages are developed, *transmit_data* structures the ROS service

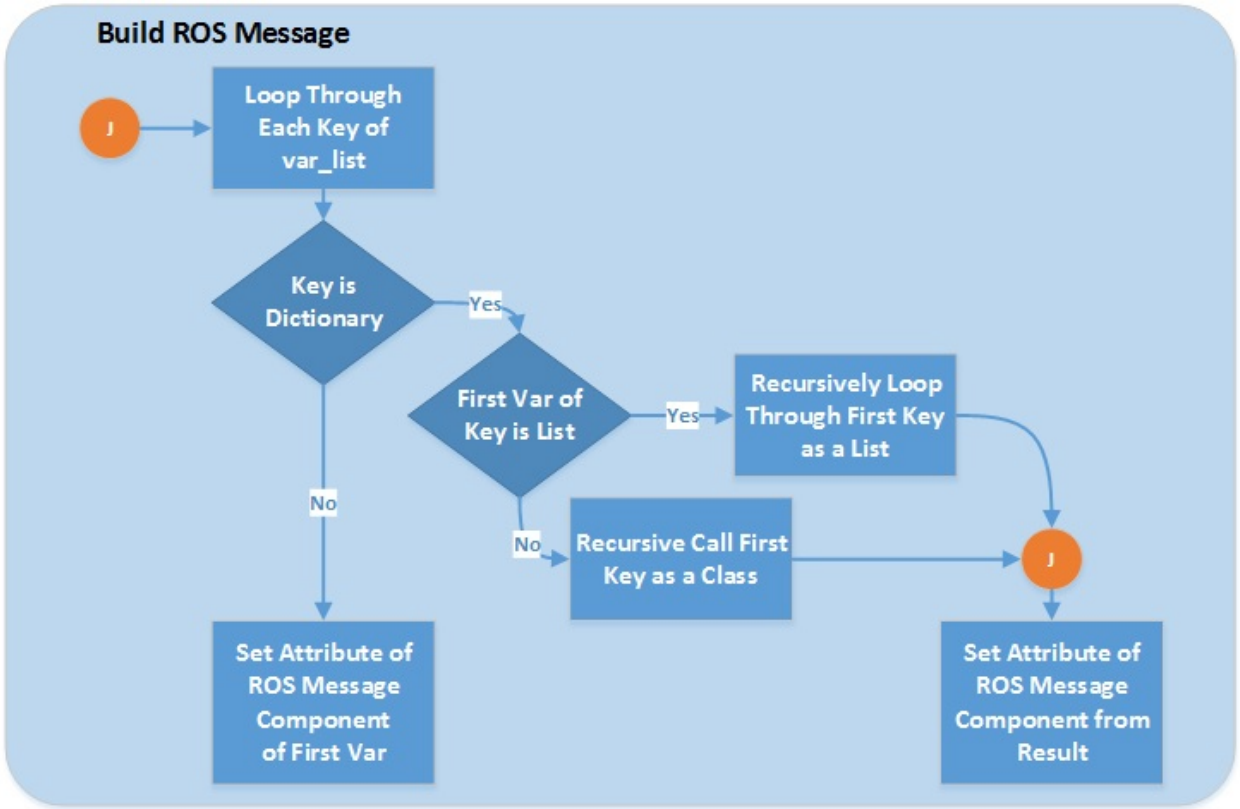


Figure 4-21. Build ROS message for xbee_bridge.

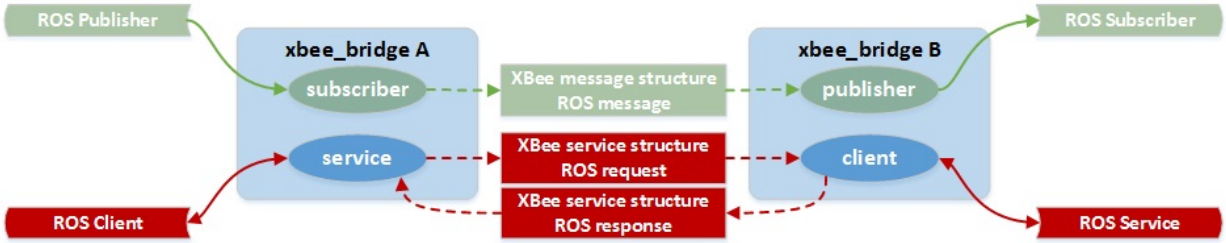


Figure 4-22. ROS service data flow between two XBee Bridge configured devices

request, except with an additional character, the *trFlag*, that specifies the message as a transmitted request.

After transmitting the data, the *standard_service_callback* holds for a response. The callback adds itself to a data structure of other callbacks that are awaiting responses from client requests described later. If a request is not received in a certain timeout period, a failed response is returned.

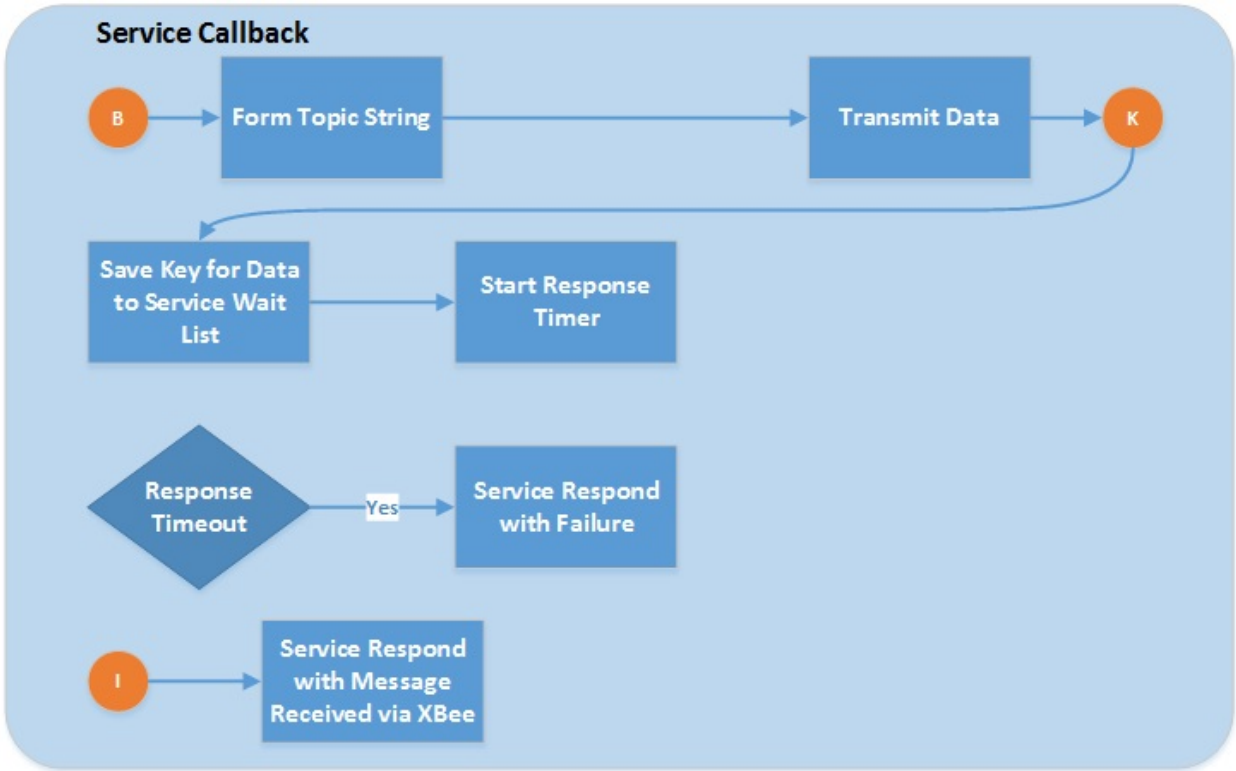


Figure 4-23. Standard service callback for xbee_bridge.

On the receiving end, ***xbee_bridge*** again behaves similar to how ROS messages are received. The *parse_string* function is called to split the message and decide what type of message was received. When the message is a ROS service specified as a request message by the *trFlag*, the *standard_client_transmit* function is called.

Figure 4-24 shows the *standard_client_transmit* function begins in the same way as *standard_publisher*. A message is first structured into a ROS message using the *build_ros_message* function, but is not published. The purpose of *standard_client_transmit* is to manage transmitting a ROS client request message to a ROS service. Once the received client request has been structured, a blocking hold checks to see if the desired service has been started. If the service has not been started, a timeout occurs and a failed response is sent back to the original calling ROS client. Otherwise, the data is sent to the ROS service, processed, and a response is received.

This response , in turn, is sent to the original requesting XBee device through the *transmit_data* function. In the case of a ROS service response, *transmit_data* structures the ROS service response message with the *trFlag* set for a transmitted response.

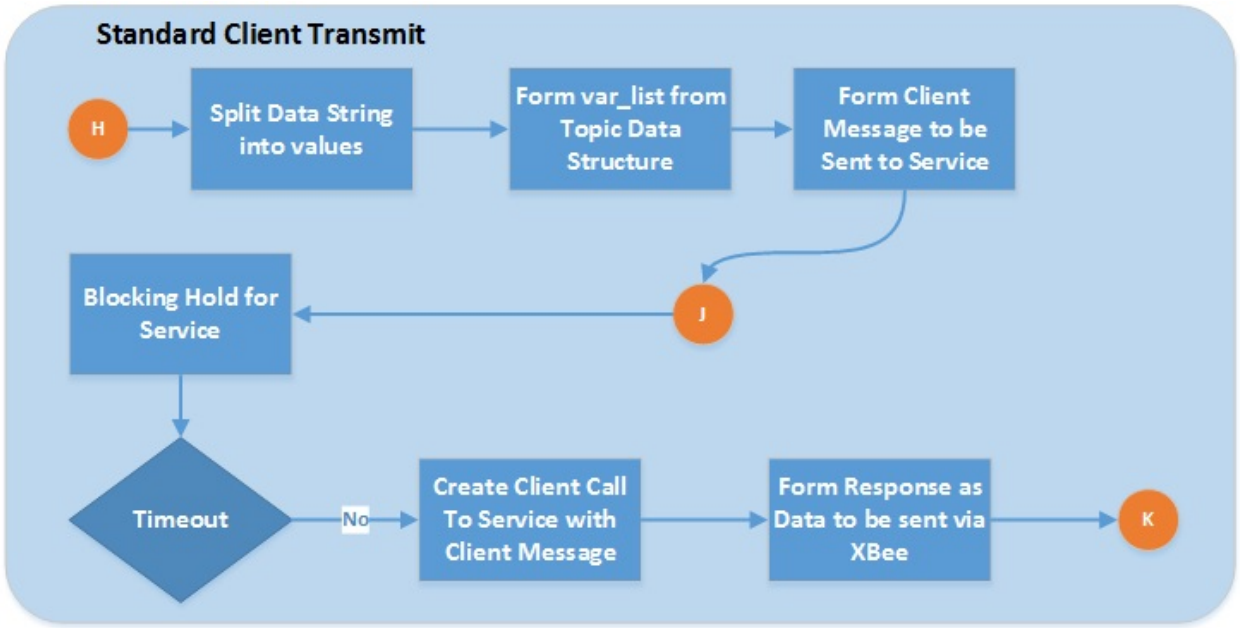


Figure 4-24. Standard client transmit for xbee_bridge.

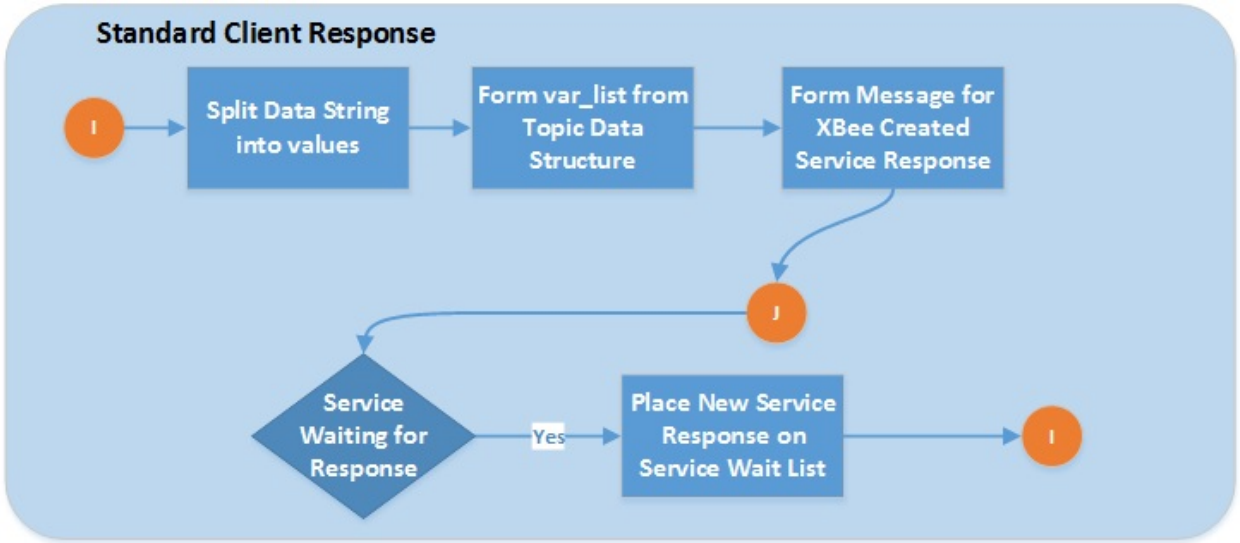


Figure 4-25. Standard client response for xbee_bridge.

During the final leg for transmission, the ROS service response is received by the original transmitting XBee. The message is again passed to the *parse_string* function which uses the *trFlag* to recognize the messages as a ROS service response and then calls the *standard_client_response* function, which is shown in Figure 4-25. The message is structured into a ROS message using the *build_ros_message* function. The type of message and source is then compared to a data structure holding a list of services that have been sent out and are awaiting responses. If the received message is part of the data structure, it then triggers a response in the *standard_service_callback* function shown in Figure 4-23. The response is passed to the original ROS client and is removed from ***xbee_bridge***.

4.4 Swarm Core

The Swarm Core (***swarm_core***) meta-package serves as the core component in the architecture for directing missions among the various agents that make up the swarm. The meta-package is made of multiple packages that direct several stages of a mission. Swarm Core is made to be customizable, allowing a diverse selection of mission types, planners, or vehicle control applications to be implemented. To provide this customization, the meta-package takes advantage of ROSs communication infrastructure and plugin library system to give a user modularity of which methods are utilized to complete a mission.

Swarm Core is setup using a hierarchical structure. The Roll Call package assists in registration of new vehicles to the swarm. The Mission Control and Agent Control packages are used to administer mission communication between the base station and agents of the swarm. Two plugin libraries exist and are used within Agent Control to assist in goal planning according to mission settings and low level vehicle control per mission settings and goals. It is presumed that the Swarm Core framework is configured throughout all agents and at least a single base station. The purpose of the base station

is to either assist a user in choosing and controlling the mission with the aid of Mission Control, or to simply give a user information on the progression of the mission.

4.4.1 Roll Call

Since a function of Swarm Core is expandability, knowing all agents and their purpose a priori to mission start is not desired. As new agents come online, it is preferred that they report their presence and role to all vehicles in the swarm. Roll Call is a simple package with two nodes, ***agent_role_server*** and ***agent_list_publisher***, tasked with achieving this goal. ***agent_role_server*** is intended to run on the agent, while ***agent_list_publisher*** runs on the base station. As agents are activated, they initiate the *agent_role_call* service message, resulting in ***agent_role_server*** sending the *agent_role* message to the base station containing the agent's name, role, and parameters used to define its' capabilities. This information is received on the base station by the ***agent_list_publisher***, which is tasked with responding to ***agent_role_server*** via the *role_acknowledge* message, and then to package the agents information into a list, *agent_list*, which is published for all agents. This list allows new agents to get information on agents that have already been incorporated into the swarm, while also allowing other agents to get information of the newly introduced agent. Figure 4-26 shows how messages and transmitted between nodes of the Roll Call package.

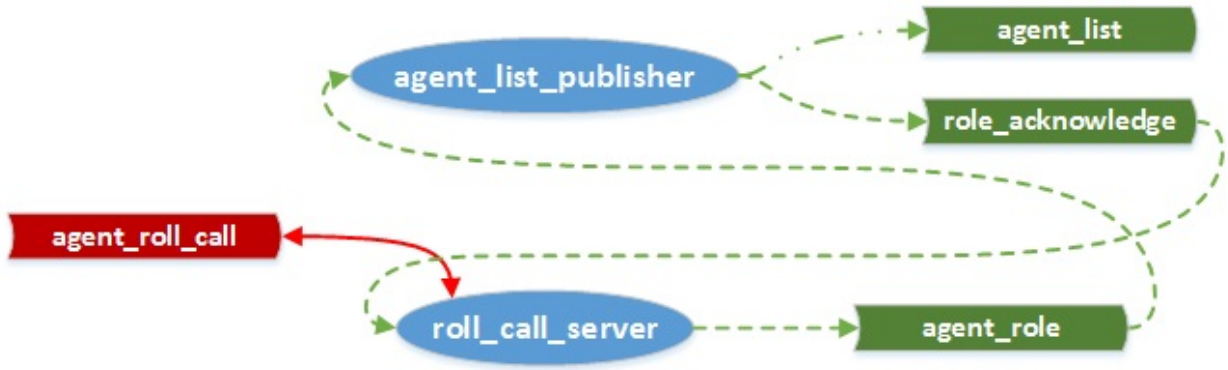


Figure 4-26. ROS data flow between agent_role_server and agent_list_publisher.

4.4.2 Mission Control

The Mission Control package manages dissemination of missions and parameters to agents, while handling overhead control of the progress of a mission for a single agent or all agents at any given time. All processing for Mission Control is handled through the ***mission_control*** node.

A mission is prepared externally either through the base station GUI or through a simple package reading a configuration file with mission details. Missions are defined as a set of mission settings, including the type of mission, as well as a given waypoint list or region of interest in which to work. Missions may be defined for a single agent or for the entire group. Once a mission has been received and is packaged, ***mission_control*** attempts to send it to the intended target, waiting a set period for feedback from each target. If the target for the mission is the entire group of agents, the mission is stored for later use. Missions include a minimum number of agents that must respond before acknowledging successful transmission of the mission.

Once the minimum number of agents have received the mission, ***mission_control*** prepares to receive either new mission updates or commands from an external user in the form of ”Start”, ”Abort”, ”Pause”, or ”Return to Base.” Similar to how missions are transmitted, once a command is received, the command is packaged and sent to each intended target and then awaits a response. If sent to the group, the command is stored.

A major focus of ***mission_control*** is the effort for handling new agents added to the swarm after a group mission or command is sent. When any mission or command is sent to a single agent, it is assumed that the information was never intended for the group, so only that vehicle will be affected. When group missions or commands are sent, they are stored. As a new agent is added to the group, ***mission_control*** is tasked with sending the new agent the latest stored mission and command so they can join the ongoing mission.

An important factor of ***mission_control*** is modularity. Though described as existing on the base station, ***mission_control*** can run on any agent in the swarm, further decentralizing the system. It is intended that only a single ***mission_control*** node is running in the entire swarm architecture, so a leader explorer package is used to define the identity of the leader of the swarm. Once the leader is discovered, it will become the central point of mission control unless it fails and another leader must be chosen.

The data flow of ROS messages used in ***mission_control*** are shown in Figure 4-27. The following sections describe the functional flow of the ***mission_control*** node.

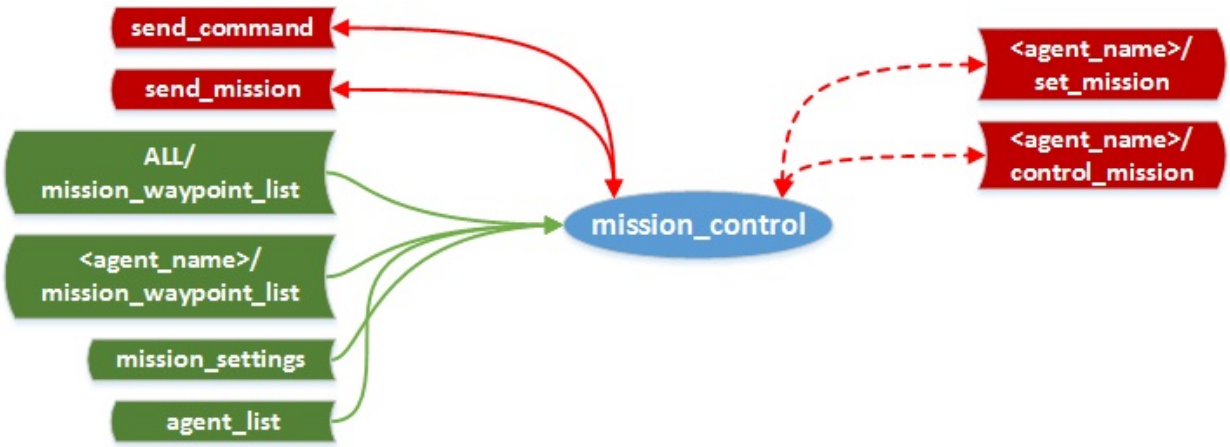


Figure 4-27. ROS data flow for mission_control

4.4.2.1 Node Initialization

The initialization of ***mission_control***, shown in Figure 4-28, is tasked with loading parameters and setting up ROS communications. First, all communications for mission parameters, the *mission_settings* and *ALL/mission_waypoint_list* ROS messages, are configured with the *updateMissionCallback* and *updateWaypointsCallback* functions. Next, ROS services for *send_mission* and *send_command* are given the *sendMission-Handler* and the *sendCommandHandler* functions respectively. The *send_mission* service triggers preparing missions to be sent to agents, while the *send_command* triggers the ”Start”, ”Abort”, ”Pause”, or ”Return to Base” commands to be sent to

agents. Finally, the *agent_list* message from the Roll Call package is subscribed with the *groupAgentListCallback*.

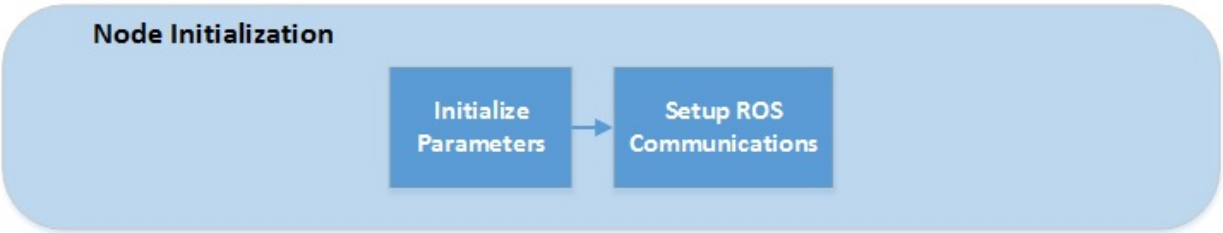


Figure 4-28. Node initialization for mission_control

4.4.2.2 Building List of Known Agents

As new agents are added to the swarm, they register through the Roll Call package and the *agent_list* message is updated with the agent's capabilities. The ***mission_control*** node listens for this message and triggers the *groupAgentListCallback* function, shown in Figure 4-29. Initially, this function handles updating a local list of all agents. The *agent_list* message is stored for use throughout the node and is checked for any changes in available agents. If an agent is new, a new subscriber is setup to listen for mission waypoint lists that pertain to the new agent using the message *<agent_name >/mission_waypoint_list*.

In situations where a mission is already in progress, the *groupAgentListCallback* function is tasked with sending any relevant mission and command information to the new agent. This is performed in the same manner that the *sendMissionHandler* and the *sendCommandHandler* functions handle mission and command transmission.

4.4.2.3 Mission Preparation and Transmission

A primary focus of ***mission_control*** is to form missions that will be used by the swarm. This is performed through the assistance of an external package. Missions are first transmitted through the *mission_settings* message triggering the *updateMissionCallback* function which simply stores the settings for later use. Missions also require mission waypoint list that defines either a path or ROI to be used for the mission.

Figure 4-29. Agent list callback for mission_control

Mission waypoint lists are received for either the entire group of agents or for individual agents through the *ALL/mission_waypoint_list* or *<agent_name >/mission_waypoint_list* respectively. When each list is published by the external package, the *updateWaypointsCallback* function is called to simply store the waypoint list in a data structure.

Missions are triggered for packaging and transmission when the *send_mission* service triggers the *sendMissionHandler* function. Shown in Figure 4-30, *sendMissionHandler* starts by confirming that at least one agent has joined the swarm. Next, the desired operative of the *send_mission* service is checked against being *ALL* agents or being a specific agent.

If the mission is for a specific agent, it is confirmed that a mission waypoint list for the agent has been stored before a mission can be sent. If the list exists, a mission is packaged according to the parameters set for the mission and is then sent to the agent through the *<agent_name >/set_mission* service. If the agent returns an acknowledge, then mission send is a success and is stored for future use, otherwise it is reported as a failure.

If the mission is for *ALL* agents, it is confirmed that the mission waypoint list given by *ALL/mission_waypoint_list* has been stored. If the list exists, a mission is

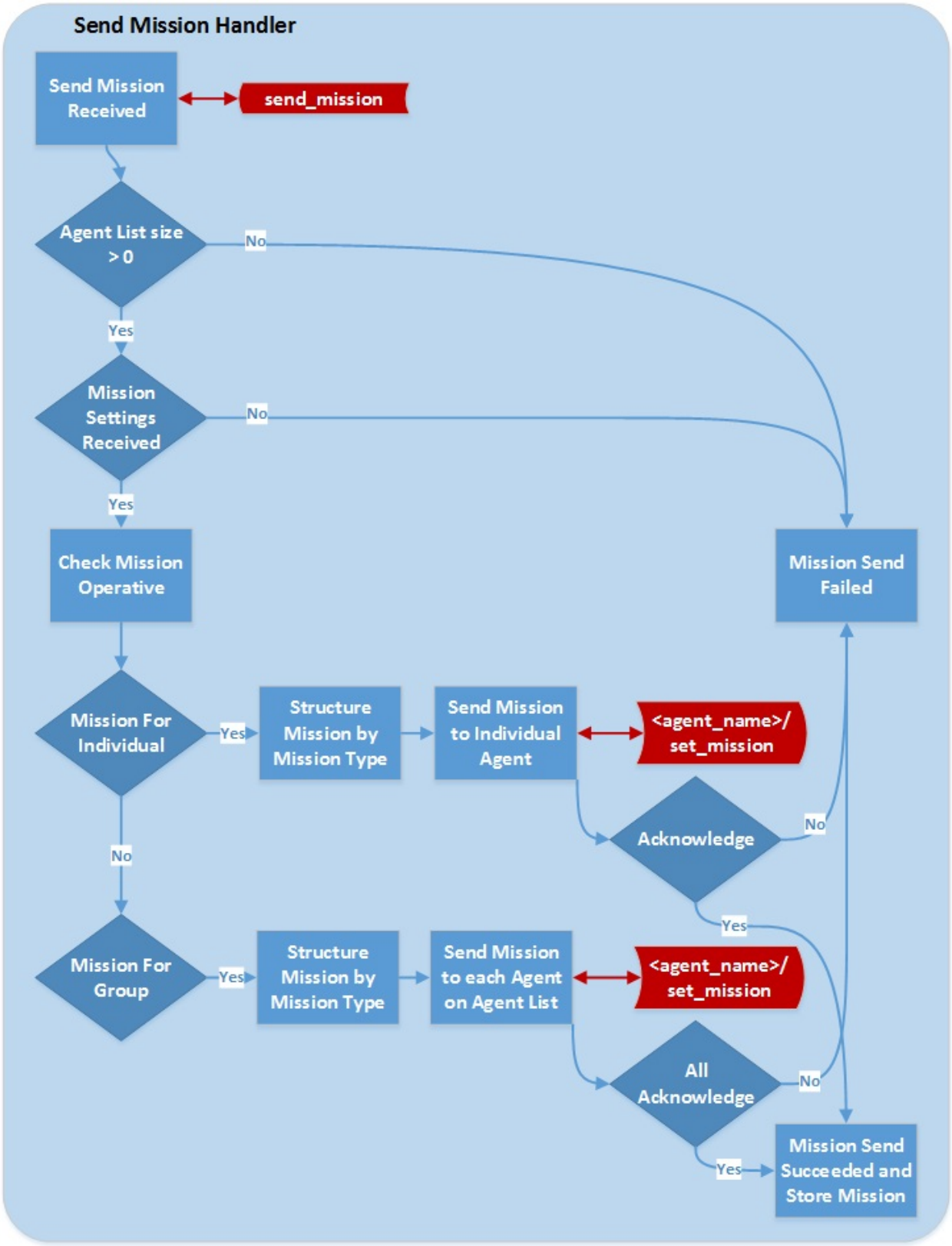


Figure 4-30. Send mission handler for mission_control

packaged according to the parameters set for the mission. Next, the mission is sent to all agents who have checked in through Roll Call. Missions are transmitted through the *<agent_name >/set_mission* service, making note of which agents acknowledge receipt or not. Once all agents acknowledge receipt, mission send is considered a success and is stored for future use.

4.4.2.4 Mission Commands

Another focus of ***mission_control*** is managing the command state of the swarm. Commands for the mission are send to ***mission_control*** from an external package through the *send_command* service which triggers the *sendCommandHandler* function. Shown in Figure 4-31, *sendCommandHandler* functions similarly to *sendMissionHandler*. First, it is confirmed that at least one agent has joined the swarm. Next the desired operative of the *send_command* service is checked against being *ALL* agents or being a specific agent.

If the command is for a specific agent, it is first confirmed that a mission has been sent and received by the agent. If the agent has acknowledged a mission, a command is sent to the agent through the *<agent_name >/control_mission* service. If the agent returns an acknowledge for the command transmission, then command send is a success, otherwise it is reported as a failure.

If the command is for *ALL* agents, it is first confirmed that a mission has been sent to *ALL* agents. If *ALL* agents have acknowledged receipt of a mission, a command is transmitted to each agent through the *<agent_name >/control_mission* service. If each agent acknowledges that the command was received, the command send is a success and the current command is stored for future use.

4.4.3 Agent Control

Contrasting to the Mission Control package, the Agent Control package has the purpose of receiving missions and commands that are transmitted from Mission Control. The package manages parsing a mission into specifics required to develop a plan for

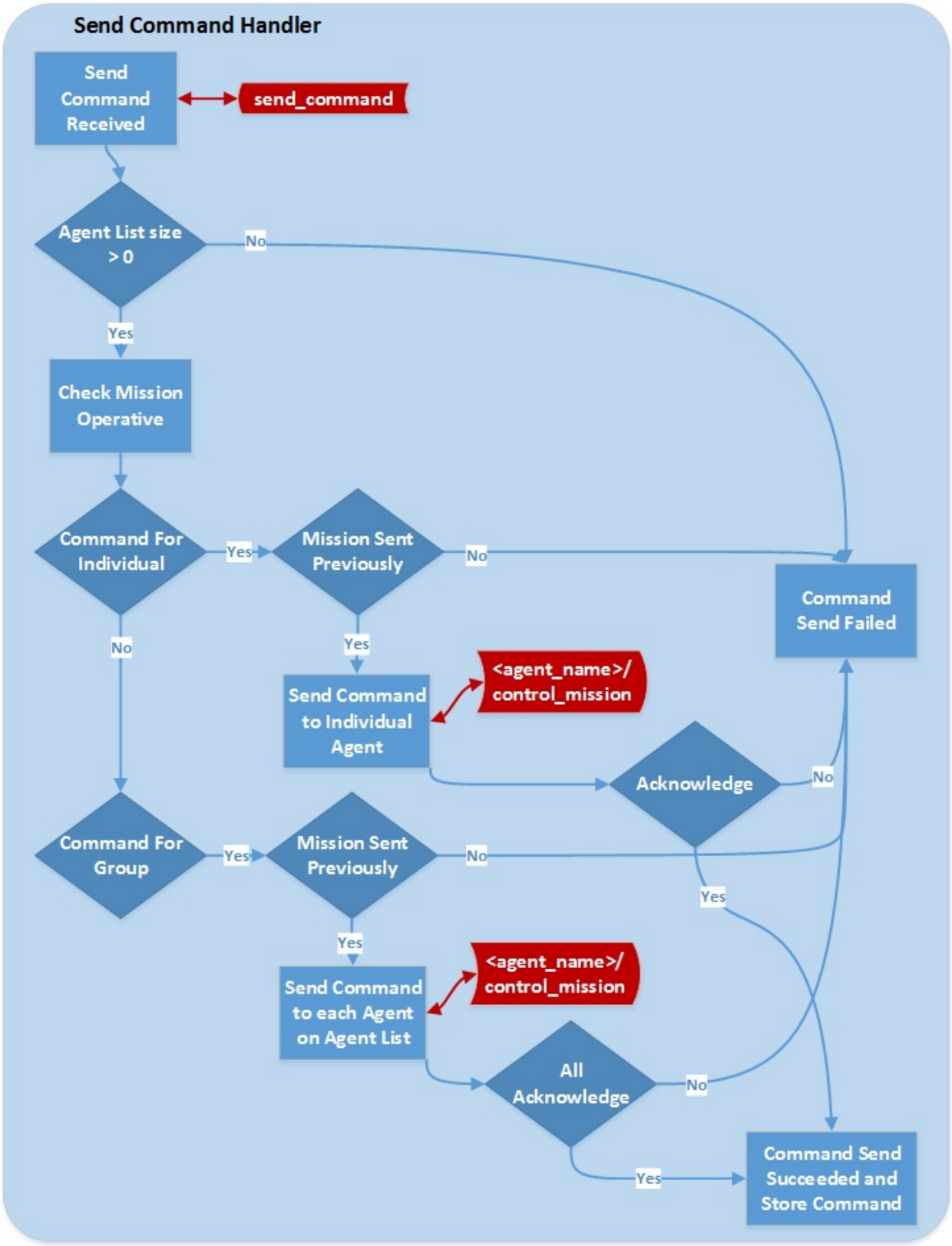


Figure 4-31. Send command handler for mission_control

the agent, while categorizing commands to modify the mission state of the vehicle for when a plan has been found. Agent Control also handles high level mission related communication with agents that join the group and are described on agent list. All processing for the Agent Control package is handled within the ***agent_control*** node.

As ***agent_control*** is tasked to handle mission goal planning as well as direct vehicle control, separate libraries are created to handle each component. It is intended that ***agent_control*** be customizable, therefore, instead of using libraries directly, the ROS pluginlib package is utilized to dynamically load custom defined libraries that handle the required tasks. The Base Goal Planner and Base Vehicle Control packages included within the Swarm Core meta-package provide standard libraries for use with ***agent_control***. C++ interfaces that define what the *base_goal_planner* library and the *base_vehicle_control* library must do are defined in the Swarm Core package. Custom libraries may be made as long as these interfaces are followed. Also defined in the Swarm Core package is the Agent class, which is used to hold parameters and define information about a specific Agent. This class is used throughout the custom libraries to hold the required information about all agents.

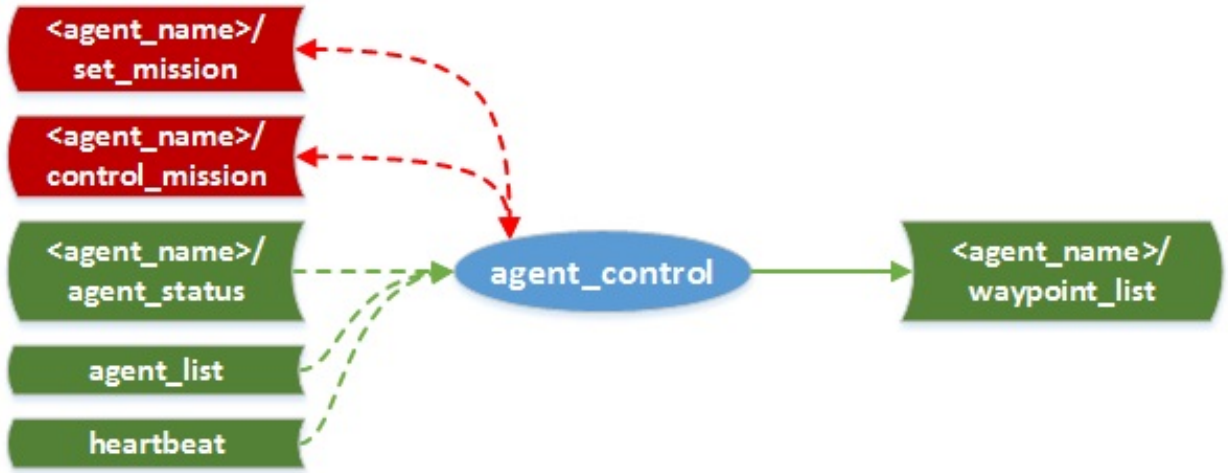


Figure 4-32. ROS data flow for agent_control

Figure 4-32 shows the ROS data flow for the ***agent_control*** node. The following paragraphs discuss the functional progression of the ***agent_control*** node for mission management.

4.4.3.1 Node Initialization

At startup, the ***agent_control*** node uses the ROS parameter system to load various parameters needed for direct vehicle control and planning. ROS Subscribers are initialized to communicate with local agent_status, agent_list, and the global heartbeat messages. Two services are started to listen for the set_mission and control_mission service commands.

As ***agent_control*** is tasked to handle mission goal planning as well as direct vehicle control, separate libraries are created to handle each component. It is intended that ***agent_control*** be customizable, therefore, instead of using libraries directly, the ROS Library Plugin package is utilized to load custom defined libraries that handle the required tasks. To define which library to use, the *base_goal_planner* and *base_vehicle_control* string parameters are loaded with the libraries name on startup. The name is compared to all shared libraries within the namespace of the Swarm Core package. If the name results in an active class, it is loaded to a class variable for later use.

4.4.3.2 Flow of Agent State Machine

After the initial start-up of the package, ***agent_control*** enters the mission state machine.

In the Initialization state, ***agent_control*** is tasked to transmit its role and capabilities to the swarm. To perform this assignment, the agent_roll_call service command is sent to trigger the ***agent_role_server*** node. If ***agent_role_server*** responds with an acknowledge response, ***agent_control*** moves to the Mission Wait state, however, a failed transmission will result in multiple repeat attempts. One of the preloaded parameters defines a timeout that will be used to end repeat attempts. If ***agent_control***

is unable to transmit the required information, the state machine is set to the Mission Failed state since initial communication with the swarm is very important.

While in the Mission Wait state, ***agent_control*** holds until the set_mission service command is received. During receipt of the message, the mission data is stored and the state is changed to Plan Mission. Before the Plan Mission state will progress, the agent_list message must have been received and parsed at least once. This message should be received from the subscriber setup during startup. Each time a new agent is added to the group, this message is called to parse the agent_list and split agents into mission groups structured in the Agent class. A mission group is defined as a group of agents tasked with the same role (e.g., search, identify, etc.).

Once data has been received about the swarm, the Plan Mission state is further tasked to handle discovery of a goal plan for the agent. Goals are only defined when a new mission is received or a new agent has been added to the mission group. Calculating the desired plan for an agent is performed according to the mission chosen, the mission parameters given, and the role of the agent (which mission group it belongs to). There are three types of defined missions as well as three types of mission groups for ***agent_control***. Search missions assume that the outcome for vehicles is to perform a search over a region of interest while maximizing coverage. Tasks for mission groups performing missions are defined in Table 4-6.

Path missions are much simpler than Search missions. A Path mission is typically sent to a single vehicle and is treated as a method of overriding any current mission and giving a vehicle a direct path to travel. This may be performed during testing or during missions if a operator desires a direct search path. Random missions are similar to Search missions in regards to the desire for search an area. The primary difference is search vehicles in a Random mission are not given a goal list as it is assumed they will travel the environment randomly. Track and Identify vehicles follow the same planning behavior performed in Search missions. If planning for the agent is unsuccessful, the

Table 4-6. Task Types for Agents Performing Missions

Task Type	Description
Search	Every agents information and the region of interest for the search are sent to the base_goal_planner class as defined in the interface. The regions of interest is assumed to be defined as an area of GPS locations given in the mission settings. It is expected that the base_goal_planner will return a series of GPS points to be used as the goal list to travel the environment and complete coverage. If for any reason that the goal planner could not generate a list, it is assumed that the mission fails and the state is set to Mission Fail. If the Goal list is found, send it to all agents in the group to reach a consensus.
Track	A simple method of detecting which agent should be tracked is used to get the agents GPS location. The single location is sent as the goal list.
Identify	An empty goal list is sent. Identify vehicles are given goals dynamically as the mission progresses.

state is set to Mission Fail. If planning is successful, the *base_vehicle_control* class is initialized with the plan, mission details, and agent details before moving to the Command Wait state.

While in the Command Wait state, ***agent_control*** holds until the control_mission service command is received. The only command accepted at this moment is the ”Run” command, which will result in the current mission plan set to the *base_vehicle_control* class being started. Once the mission has begun, ***agent_control*** will enter the Mission Running state.

While the mission is running, ***agent_control*** is set to listen to the results of the mission in progress in the *base_vehicle_control* class. If the mission is complete, ***agent_control*** moves to the Mission Complete state, however, if there are any errors or failures, ***agent_control*** moves to the Mission Fail state. During the current state, other commands sent through the control_mission service command are also watched. If the ”Pause”, ”Abort”, or ”Return to Base” commands are ever received, then ***agent_control*** will move to the respective states.

When a mission is finished and ***agent_control*** enters the Mission Complete state, the system first checks to see if any follow-up missions are required. This specifically happens if a new plan is required for the given role of the Agent to complete the overall mission. If a plan is required, all settings are collected and the state is set to the Plan Mission state. If no plan is required, the state is set to Return to Base, where the *base_vehicle_control* class is sent the Return to Base command and is assumed to be traveling back to the base station.

Previously, the Mission Pause and Mission Abort states were mentioned. While in the Mission Pause state, the *base_vehicle_control* class is sent the Pause command. Typically, pause should result in the vehicle taking precautions to find a stationary position and wait for new orders. While in the Mission Abort state, *base_vehicle_control* class is sent the Abort command, which is assumed to force the vehicle to stop immediate and safely. Abort is seen as a hard failsafe to end the mission in case a operator sees an issue.

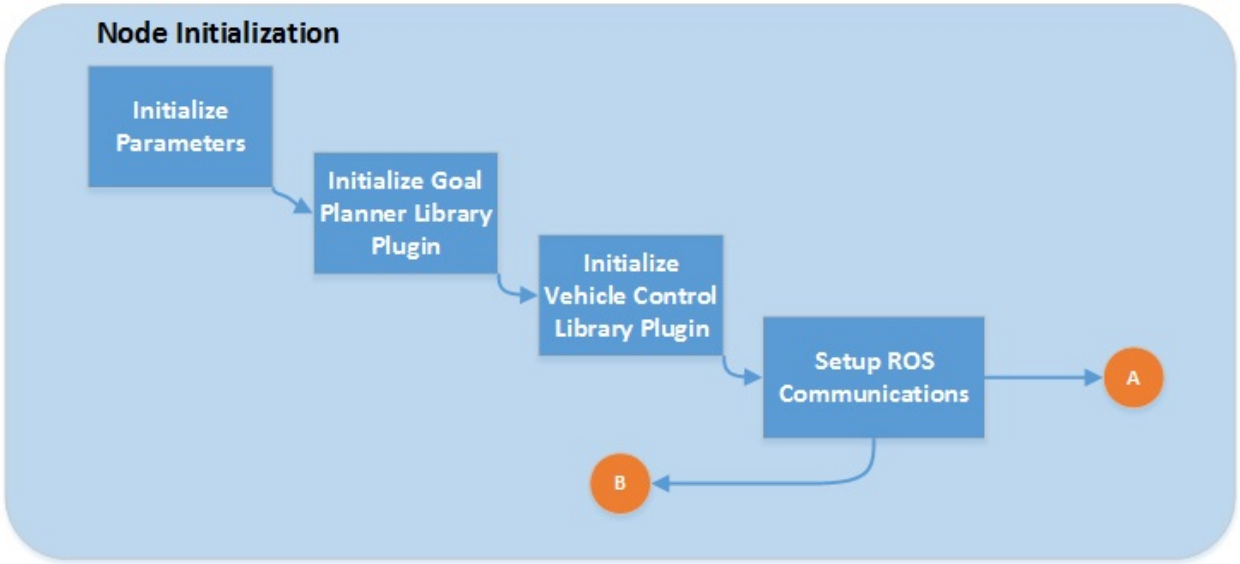


Figure 4-33. Node Initialization for agent_control

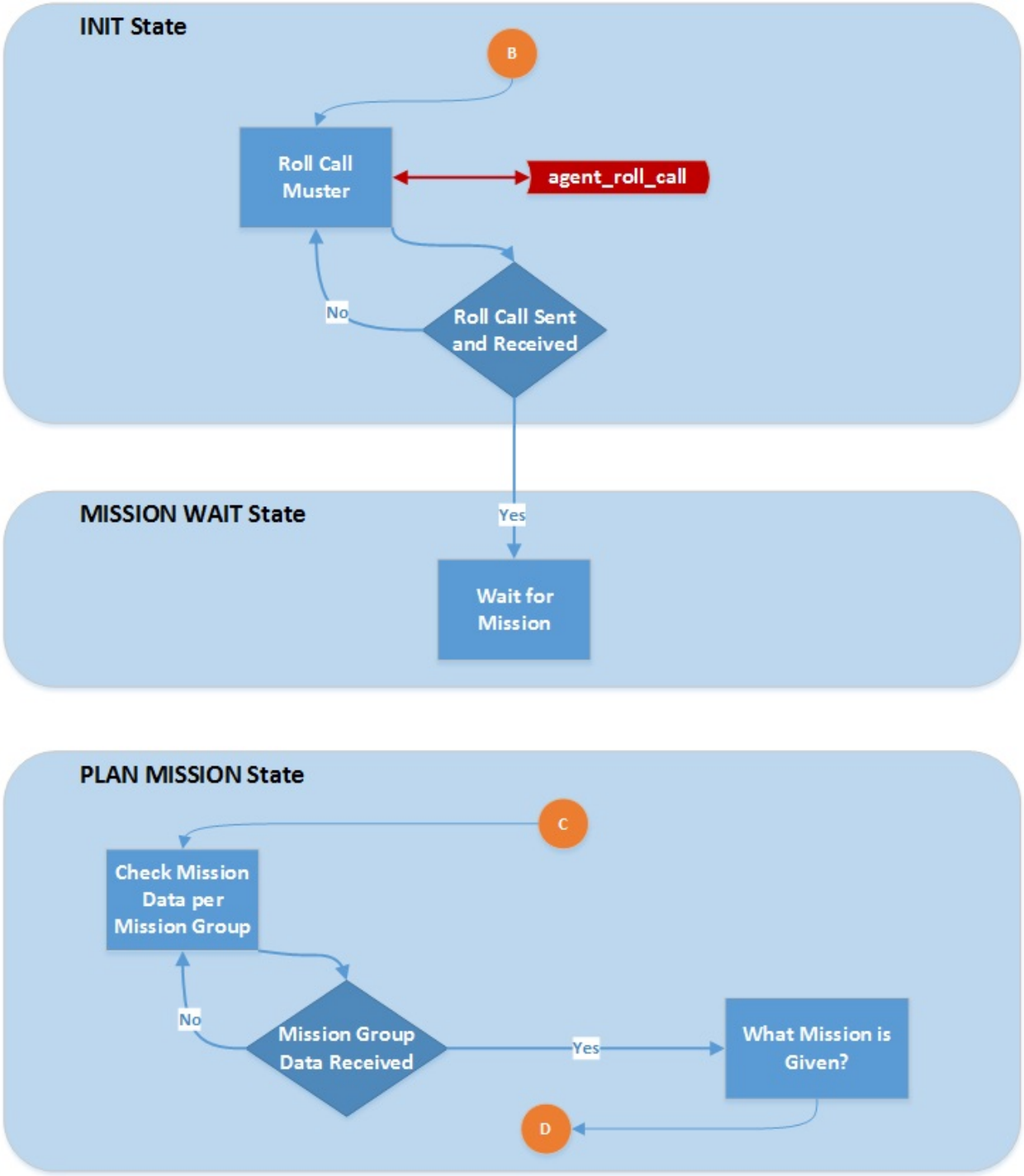


Figure 4-34. Flow of the state timer for agent_control

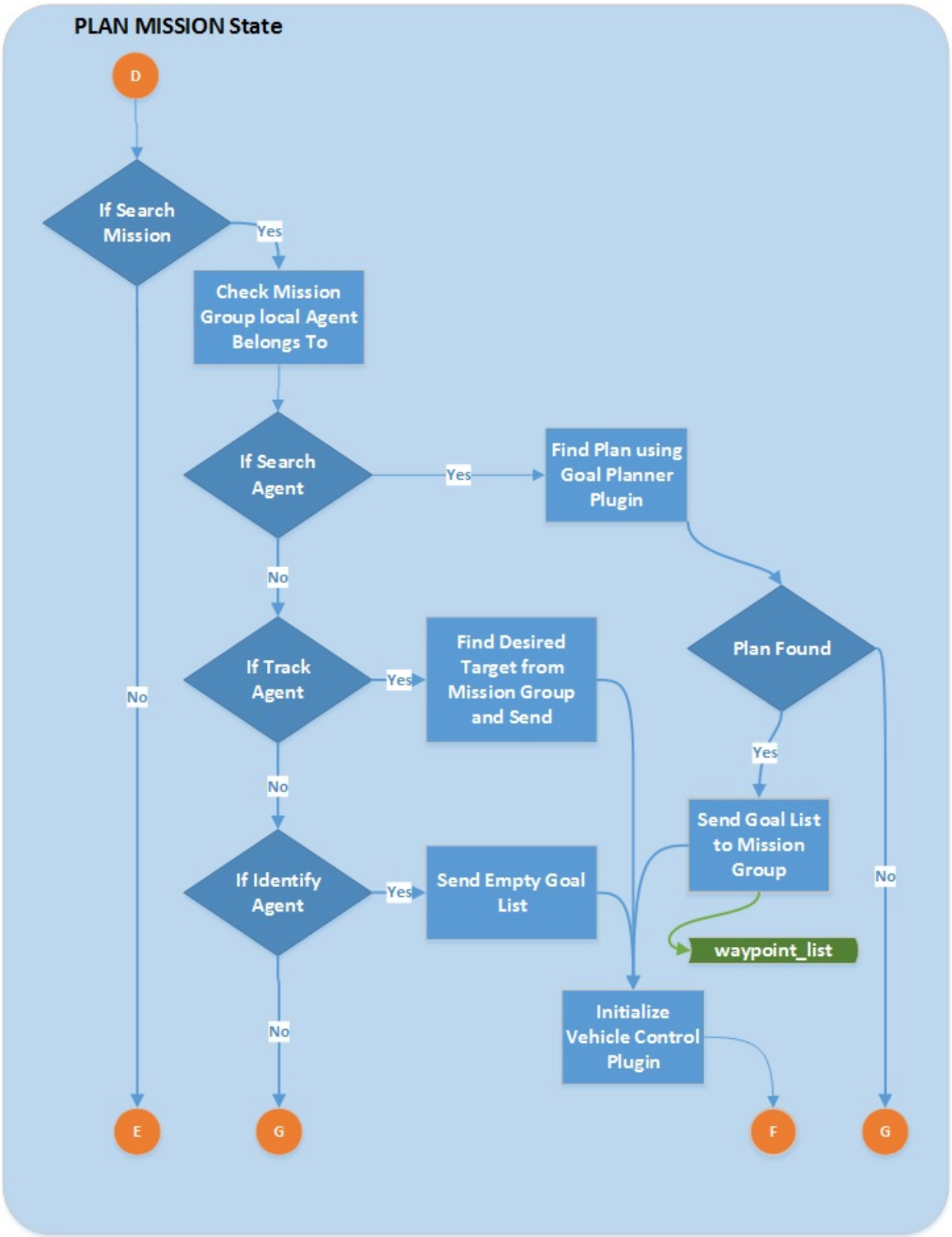


Figure 4-34. Flow of the state timer for agent_control (continued)

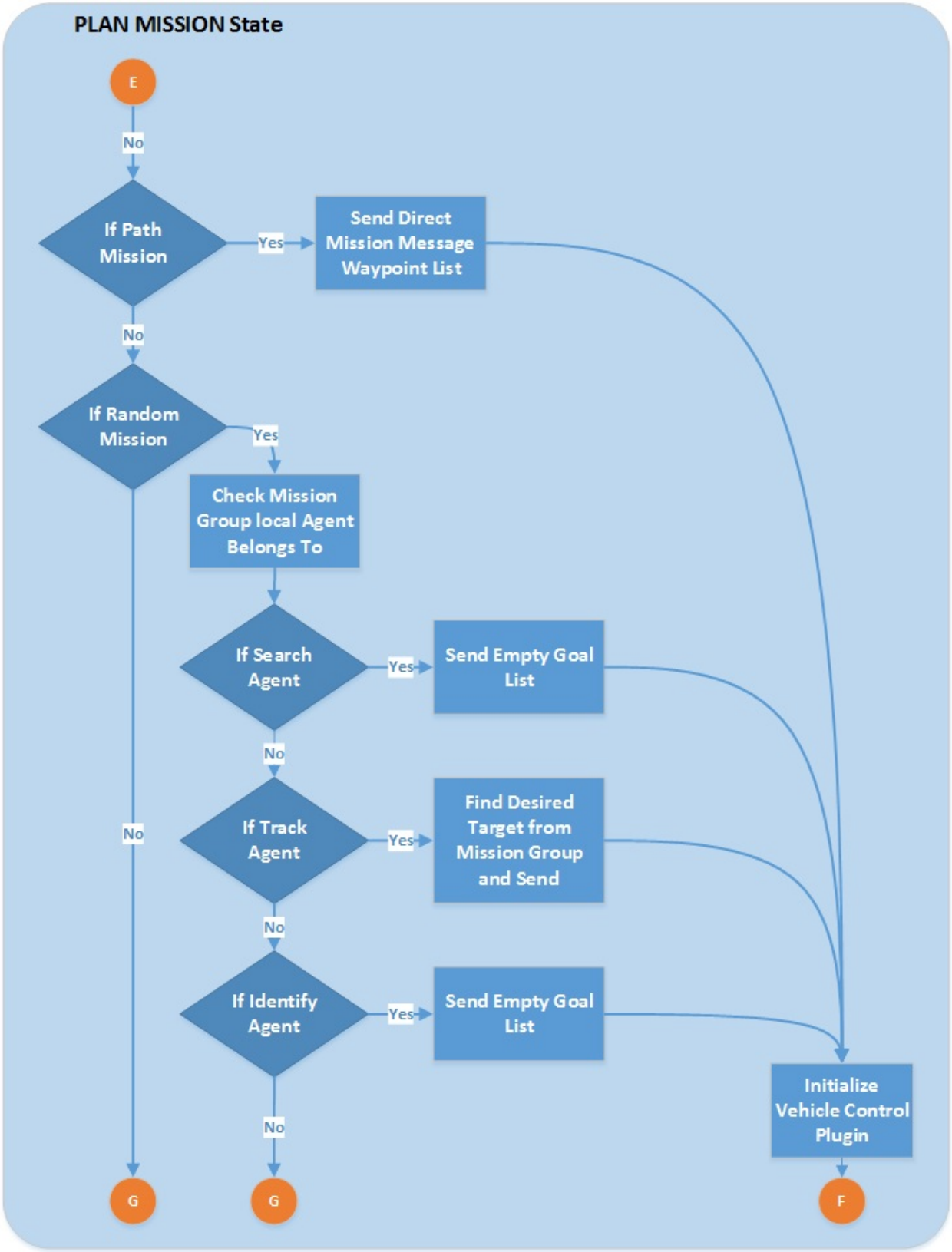


Figure 4-34. Flow of the state timer for agent_control (continued)

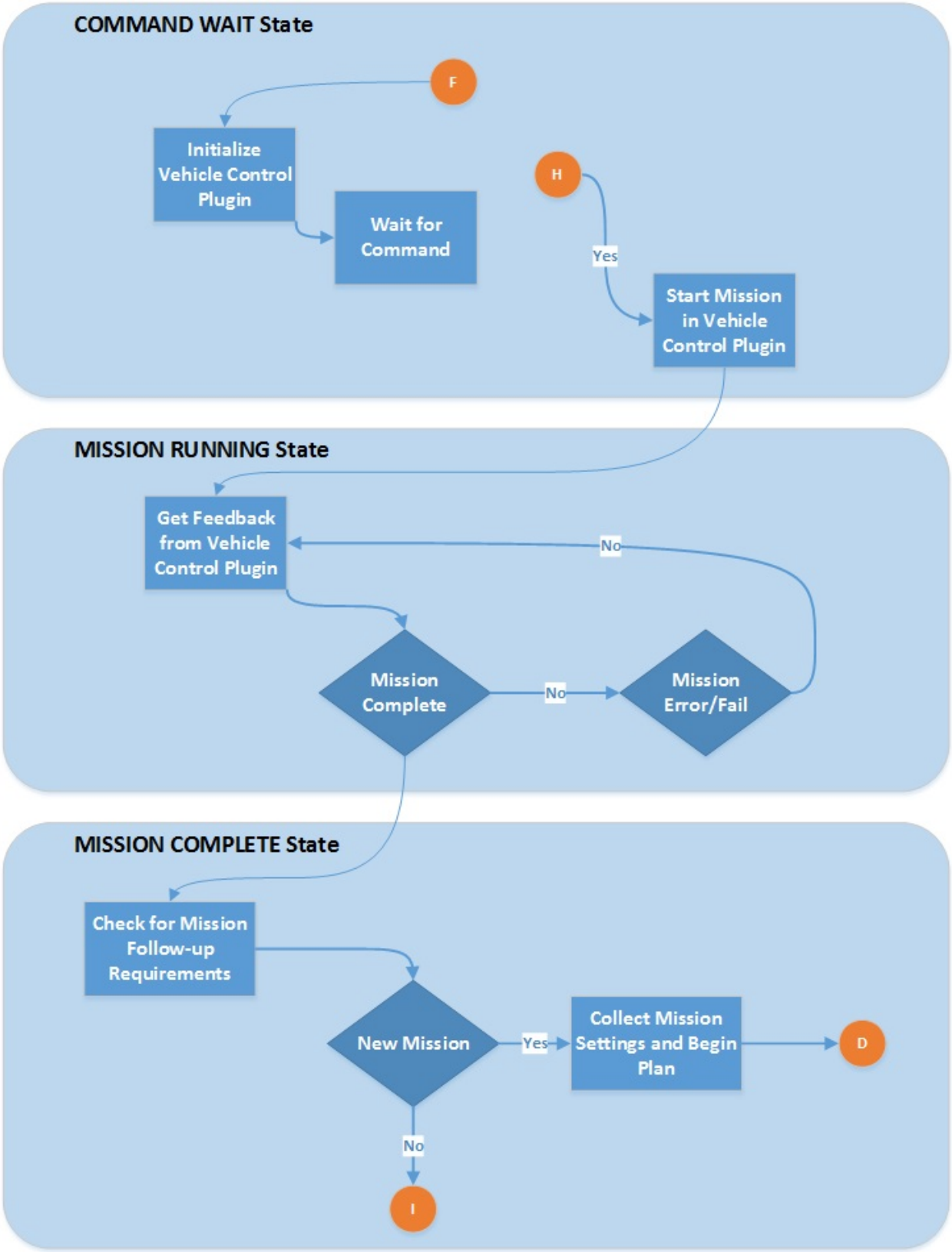


Figure 4-34. Flow of the state timer for agent_control (continued)

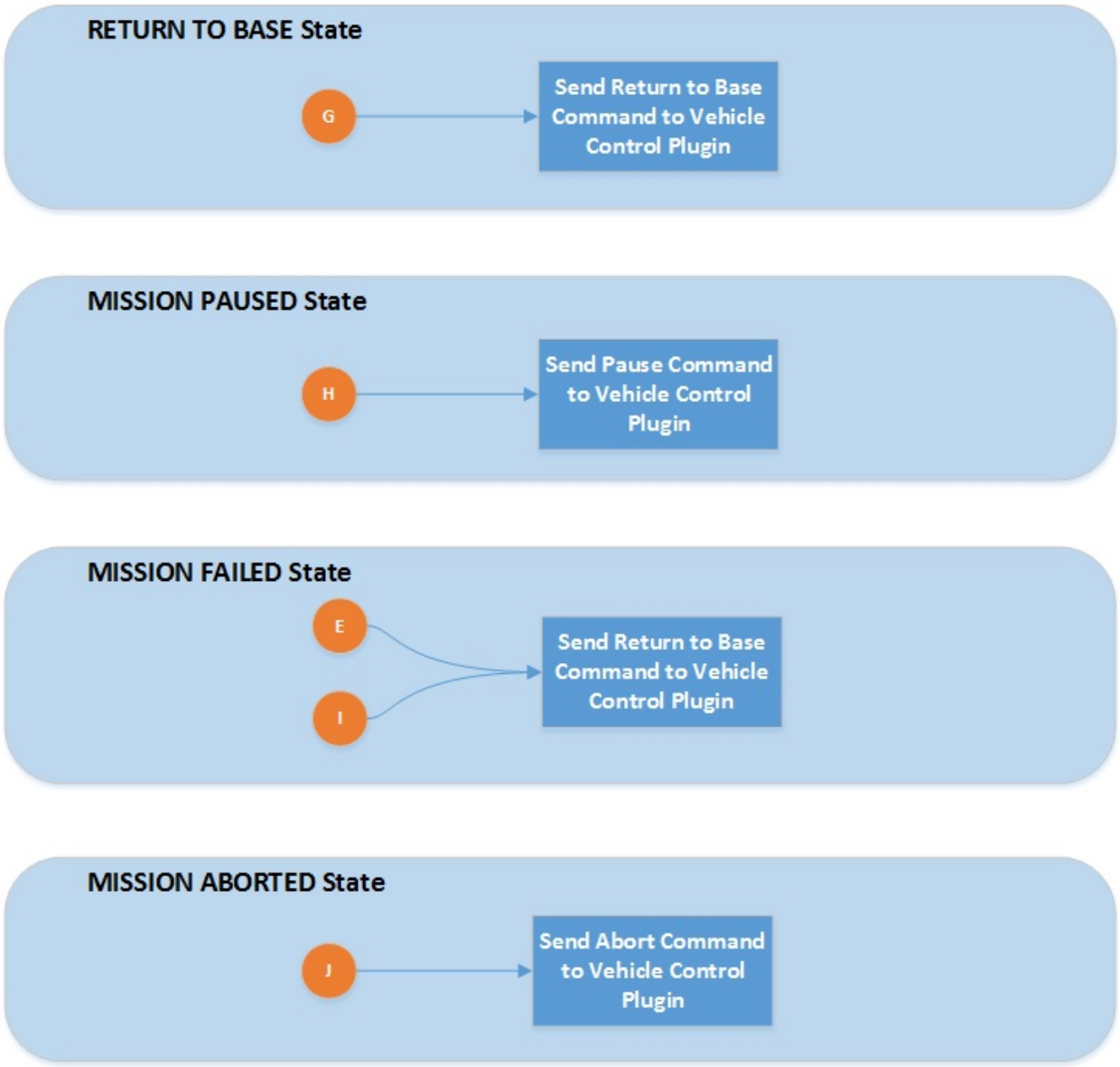


Figure 4-34. Flow of the state timer for agent_control (continued)

4.4.4 Swarm Base

Within the Swarm Base package, two C++ interfaces will be found that define two plugin libraries that should be used within the Agent Control Package. The first interface, *base_goal_planner*, defines functions needed for the plugin library tasked with calculating the goals needed for an agent to complete a mission. The interface describes a function, makePlan, that takes arguments for the known list of agents, the

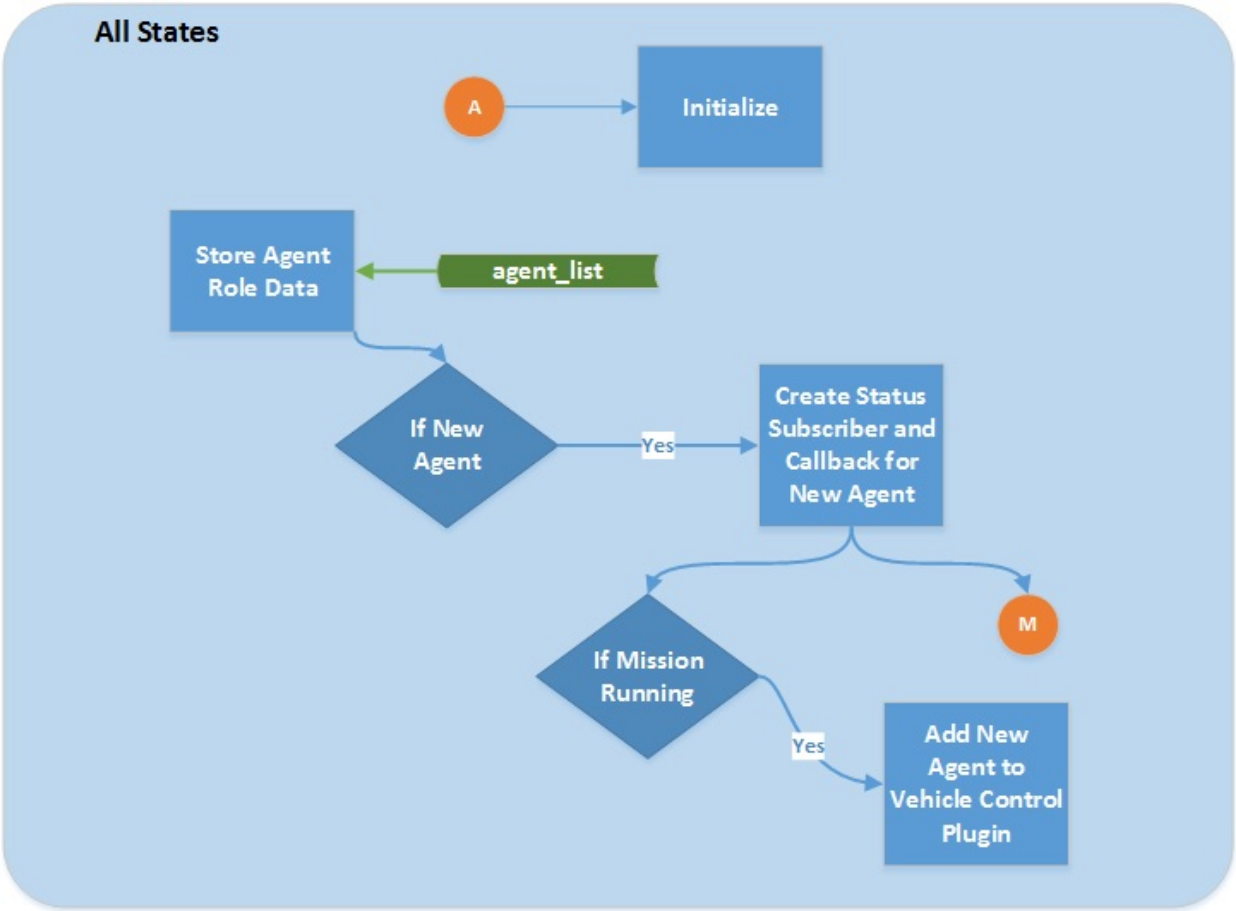


Figure 4-35. Agent list subscriber for agent_control

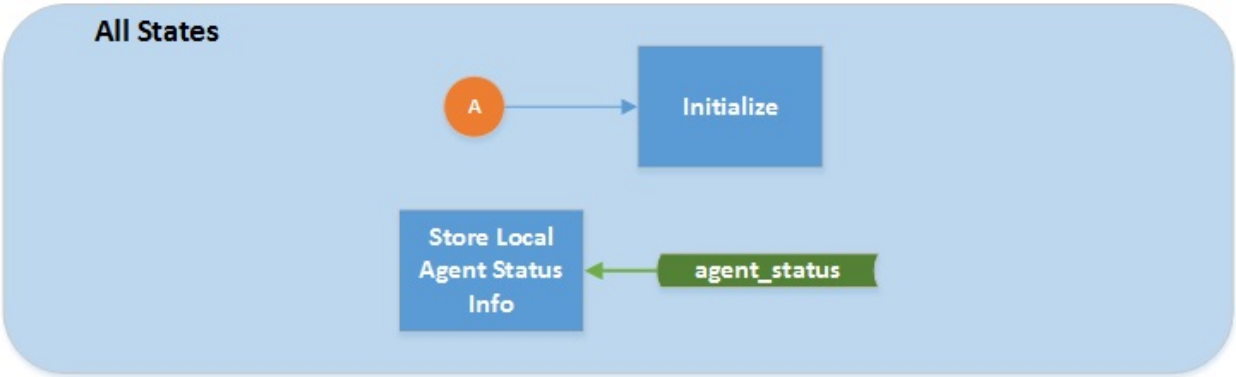


Figure 4-36. Local agent status subscriber for agent_control

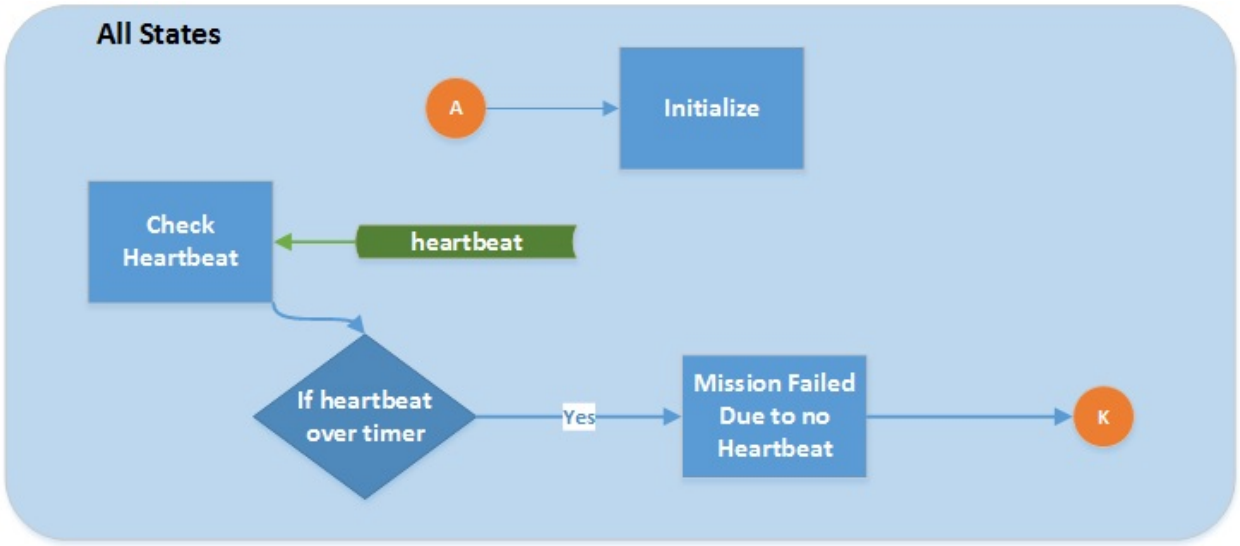


Figure 4-37. Heartbeat subscriber for agent_control

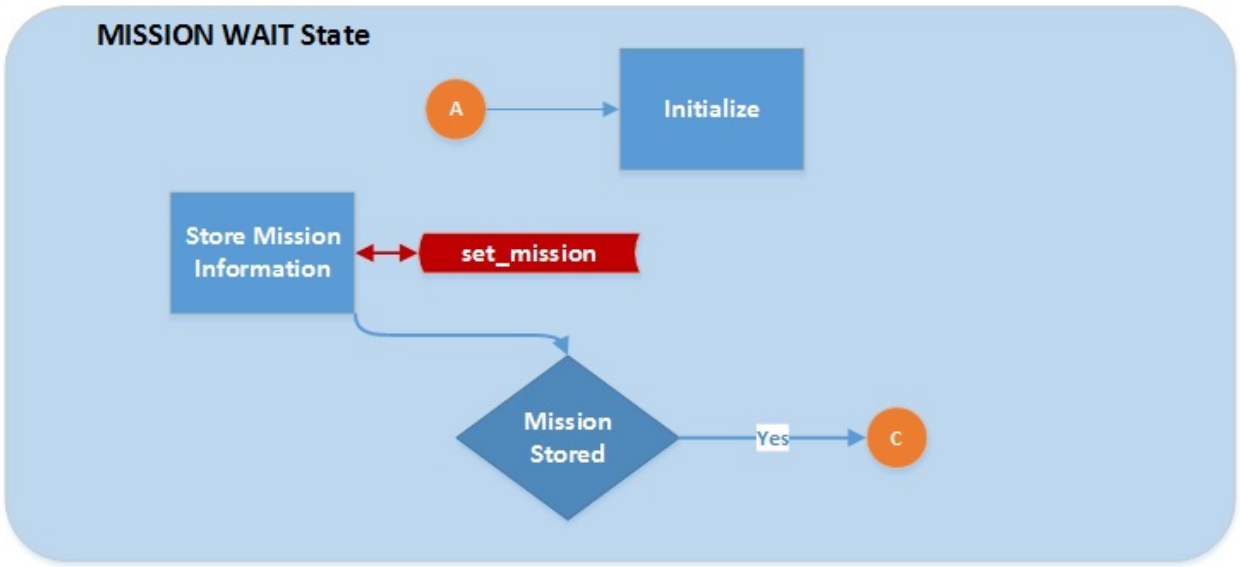


Figure 4-38. Set Mission service for agent_control

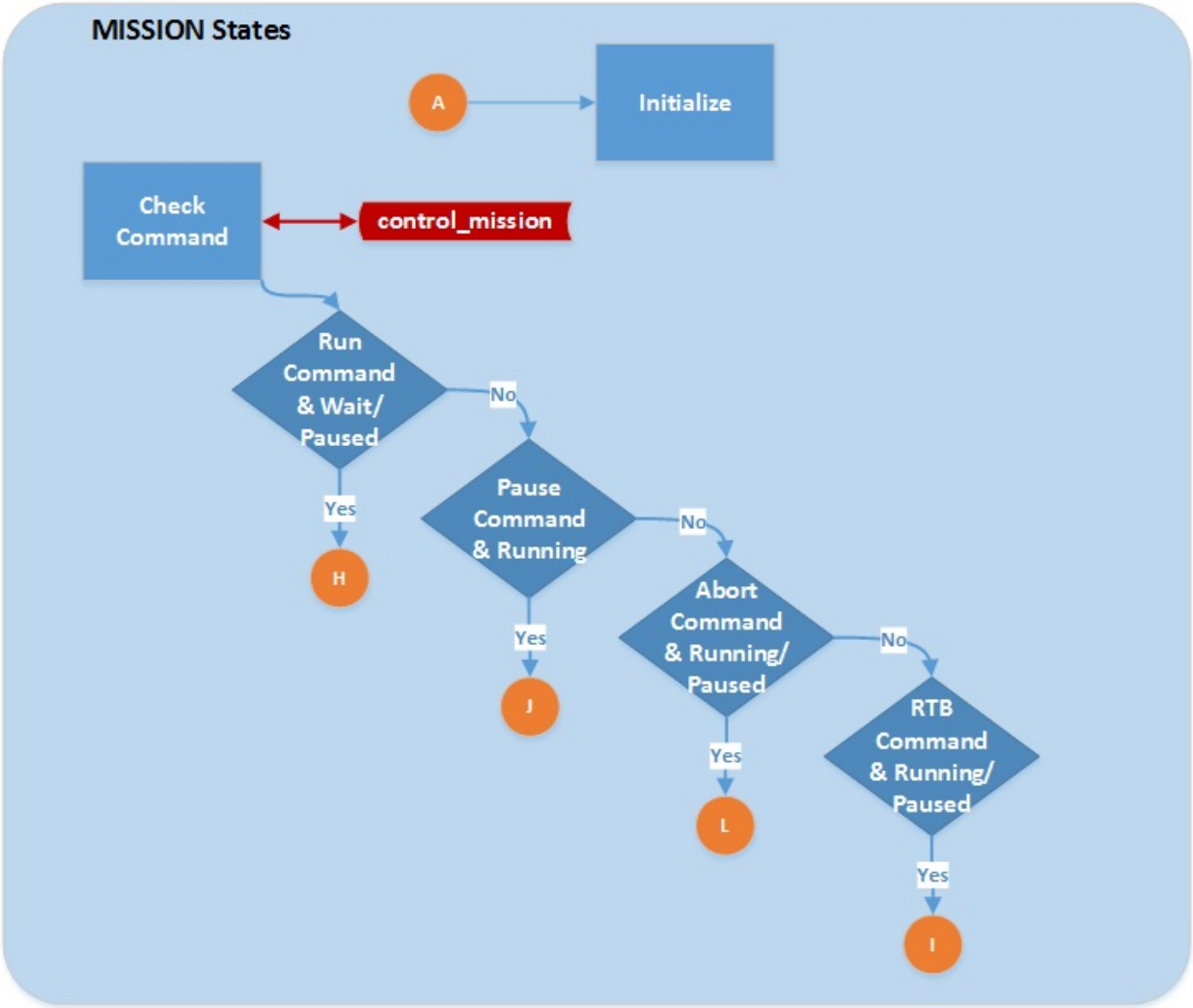


Figure 4-39. Control Mission service for agent_control

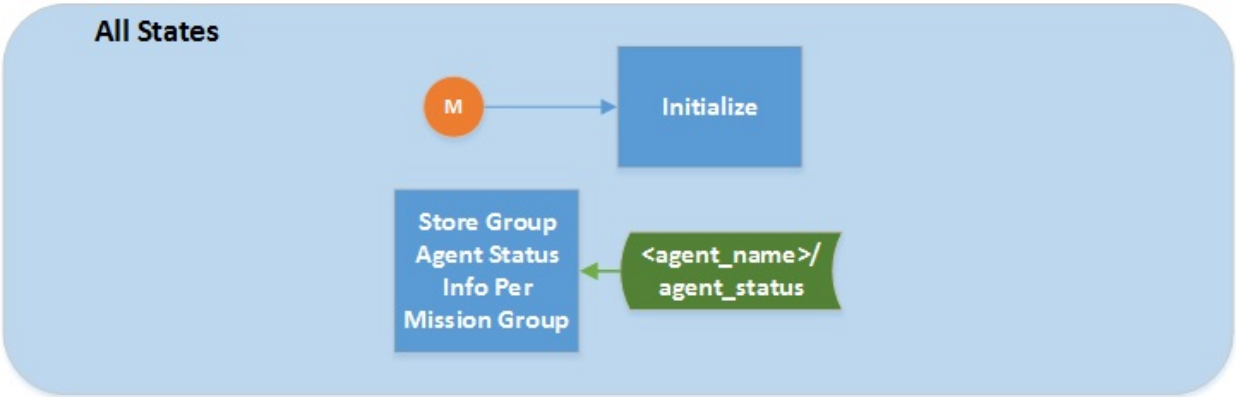


Figure 4-40. Group agent status subscriber for agent_control

region of interest polygon that define the outside boundary of a mission according to mission settings, and finally a reference to the list of points to be returned as the goal. Another function described in the function, replan, uses the previous defined boundary, but takes arguments for the new current location of agents and a reference to a new list of points to be returned as the goal.

The second interface, *base_vehicle_control*, defines functions needed for the plugin library tasked with control over a specific agent. This library is specialized, requiring direct knowledge of the type of vehicle, the controller that is employed, and the type of status messages it sends. The first function described in the interface is the setPlan function, which takes a list of points defining goals for the vehicles mission as well as a list of known agents in the swarm. Another function, isMissionComplete, is a request to see if all goals of the current agents mission have been met, resulting in a Boolean response. The final series of functions simply pass expected command types to the plugin library in the form of ”Start”, ”Pause”, ”Abort”, and ”Return to Base”. It is up to the implementer of the class to decide whether this plugin should control the vehicle directly, or just use the interface to instigate a lower level controller such as that found in the Navigation package.

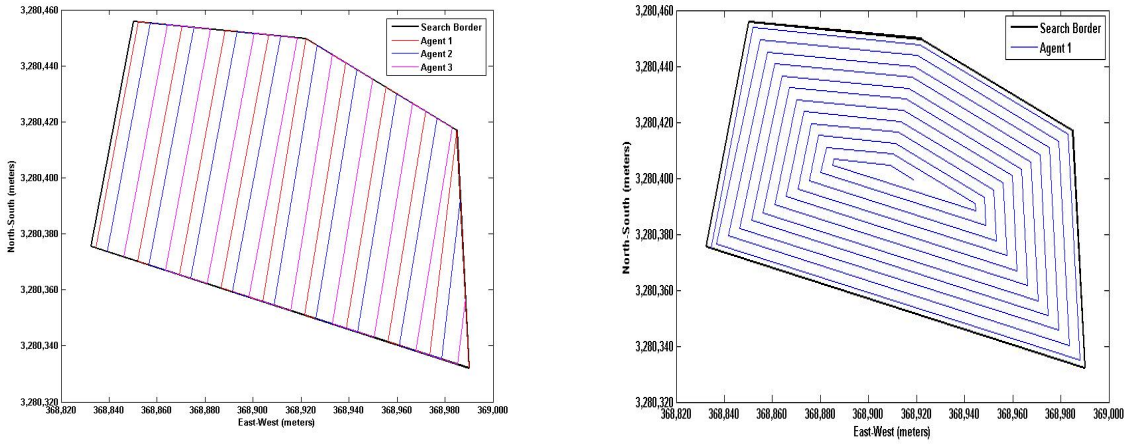
A final component of the Swarm Base package is the Agent class. This is a simple, central definition of what agents are. The class defines characteristics of agents and serves as a location to store relevant data about agents that may be needed throughout the Swarm Navigation meta-package. Example of some data include coverage rate, fuel status, and fuel cost, which are used for planning goals for the agent.

4.4.5 Base Goal Planer

As discussed in the Agent Control package section, the ***agent_control*** node expects that two plugin libraries are created using the interfaces defined in the Swarm Base package. The Base Goal Planner defines one such plugin library that is used within the ***agent_control*** node. This plugin library is tasked with the creation of a

search plan according to a given set of agents and a known region of interest (ROI) that defines an outer search boundary. The Base Goal Planner package generates the *base_goal_planner* library.

A list of agents is defined using the Agent class described in the Swarm Base package section. Information such as coverage rate, fuel status, fuel cost, and start position are sent to the *base_goal_planner* along with the search boundary. Using these arguments, a search pattern is found for each agent, including the local agent in which the ***agent_control*** node is currently running on. The desired search pattern type is defined using ROS parameters. The *base_goal_planner* is capable of generating a crossing lawnmower (agents overlap), separate lawnmower paths per agent (no overlap), or a spiral search pattern per agent. Examples may be found in Figure 4-41.



A B

Figure 4-41. Goal paths found from the base_goal_planner: A) Crossing lawnmower path for multiple agents B) Spiral path for single agent

It is assumed that *base_goal_planner* desires a path which results in the agent making it back to the start position. During calculation of the paths, fuel used is monitored so a point of no return may be found and used for ending the mission. It is also possible to force a re-plan of the mission if desired from ***agent_control***.

It is important to note that each agent will plan their own path and an expected path for each agent. The following section details the methodology an agent uses to calculate the desired paths.

4.4.5.1 Configuration of Goal Planner

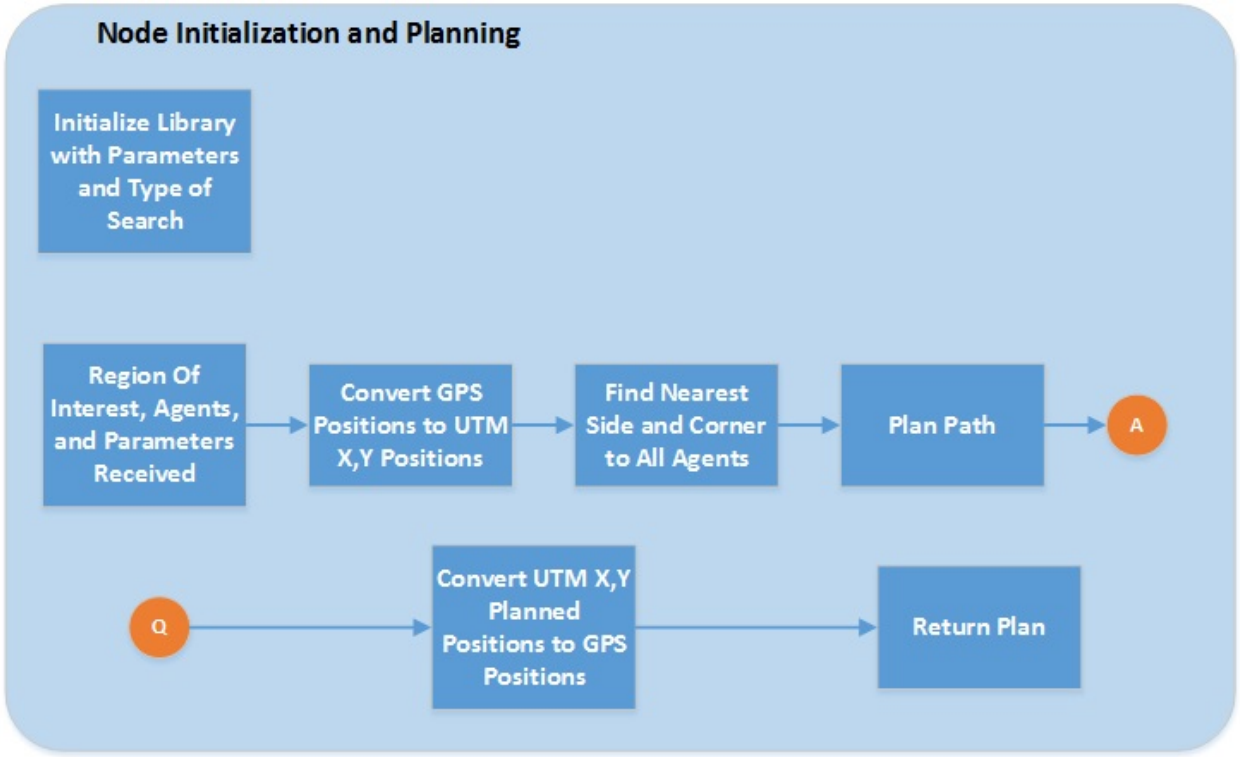


Figure 4-42. Initialization and Planning for plugin library goal_planner

The *base_goal_planner* is first initialized during its creation in the ***agent_control*** node. As shown in Figure 4-42, initialization first configures desired settings as well as the type of path generation to be used for the mission. After initialization, the goal planner is started when planning missions that require a list of goals for preparing a search. A series of GPS points in the shape of a convex polygon is sent as a ROI, as well as a list of Agents, their capabilities, and a list of mission parameters. All GPS points, including the initial positions of each agent is converted from GPS Latitude and Longitude values to Universal Transverse Mercator (UTM) x and y points. After the

conversion, the nearest side and corner of the ROI polygon to all agents is chosen as the starting points for calculating the search path.

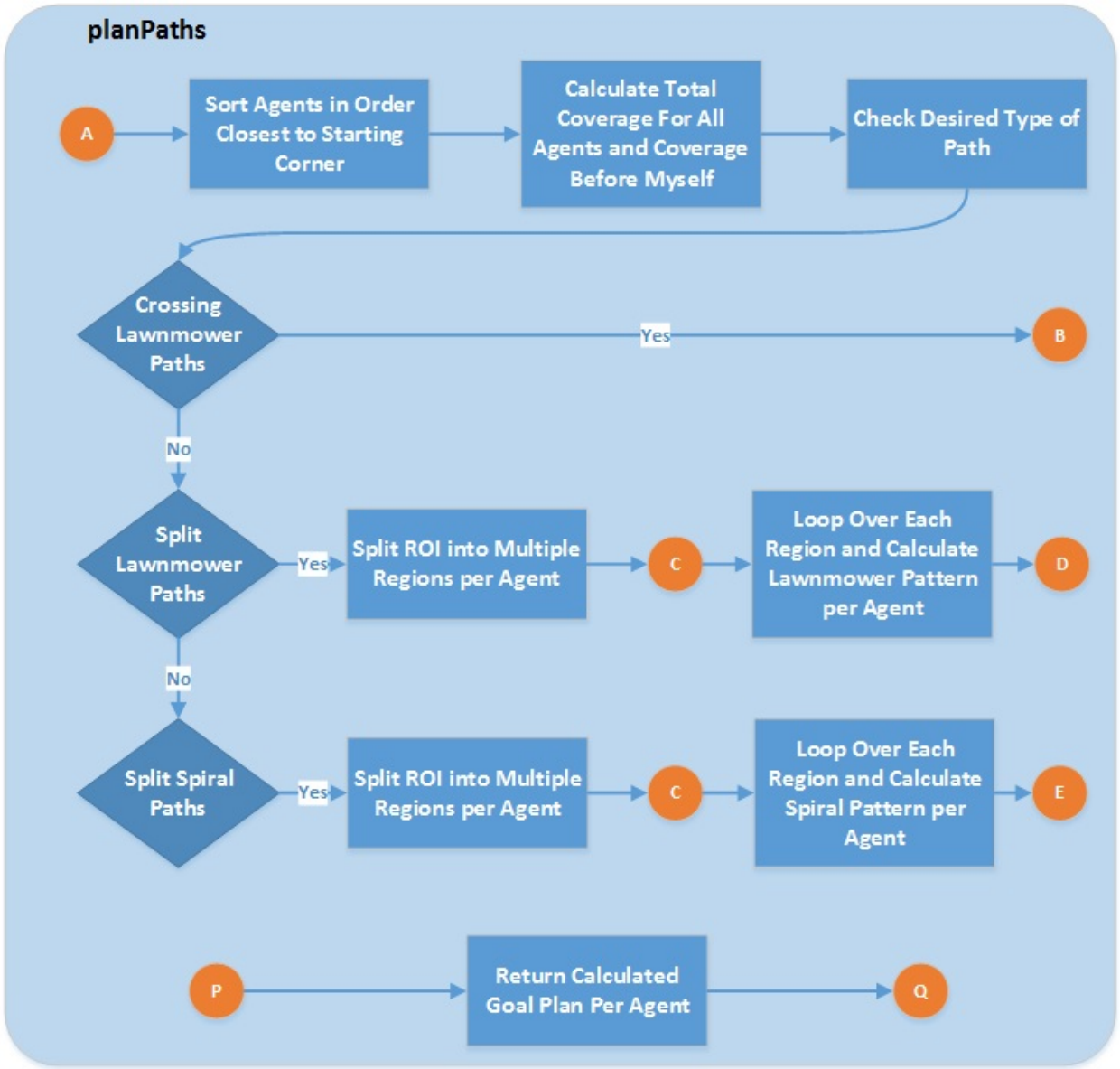


Figure 4-43. Plan Paths function for goal_planner

With each agents information, the converted ROI polygon, and the starting side of the polygon, the planPaths function is called. Shown in Figure 4-43, initially the list of agents is sorted in order of which agent is closes to the starting corner of the polygon. Next, the collective coverage area for each agent is calculated, as well as the area of

coverage before the agent that is planning the path. Finally, the type of plan to be found is checked and performed. If a Crossing Lawnmower pattern is chosen, all information regarding the mission and agents is passed direction to the method for generating the pattern. If a Split Lawnmower or Spiral pattern is chosen, the ROI polygon is first split into sections per vehicle based on their fuel and search capabilities. After being split, the method loops over each section while planning the desired pattern per agent in order of closest to the starting corner point. Once a complete goal plan has been calculated, it is returned to the calling function in ***agent_control***.

4.4.5.2 Calculating Different Goal Path Types

The first possible pattern is the crossing lawnmower path, described in Figure 4-44. This pattern focuses on creating a path for each agent, assuming that they each start near each other and will take the next path in order from closest to the starting corner of the ROI polygon.

Initially, a path is desired to start from the nearest corner of all agents and for each agent to enter the ROI through the nearest side that each agent is along. From here, the desired path will move along the next closest side of the ROI. This orientation is defined as moving from the bottom segment, along the side segment, and then to the top segment. The bottom segment will always be the ROI side that the path is moving away from, while the top segment will always be the ROI side the path is moving towards. The side segment is defined initially as the ROI side the path moves along, but is later shown as paths that have been generated and are now used as references when creating new paths.

Calculating a path is split into two types of points: the crossover and upward/downward point. The crossover point is used to generate a path as it crosses from the end of one path to the beginning of another path. The upward/downward point defines the end of a path that began at a crossover point. To begin the process of calculating the desired points, the first step is to calculate the direction of the starting indices which is used to

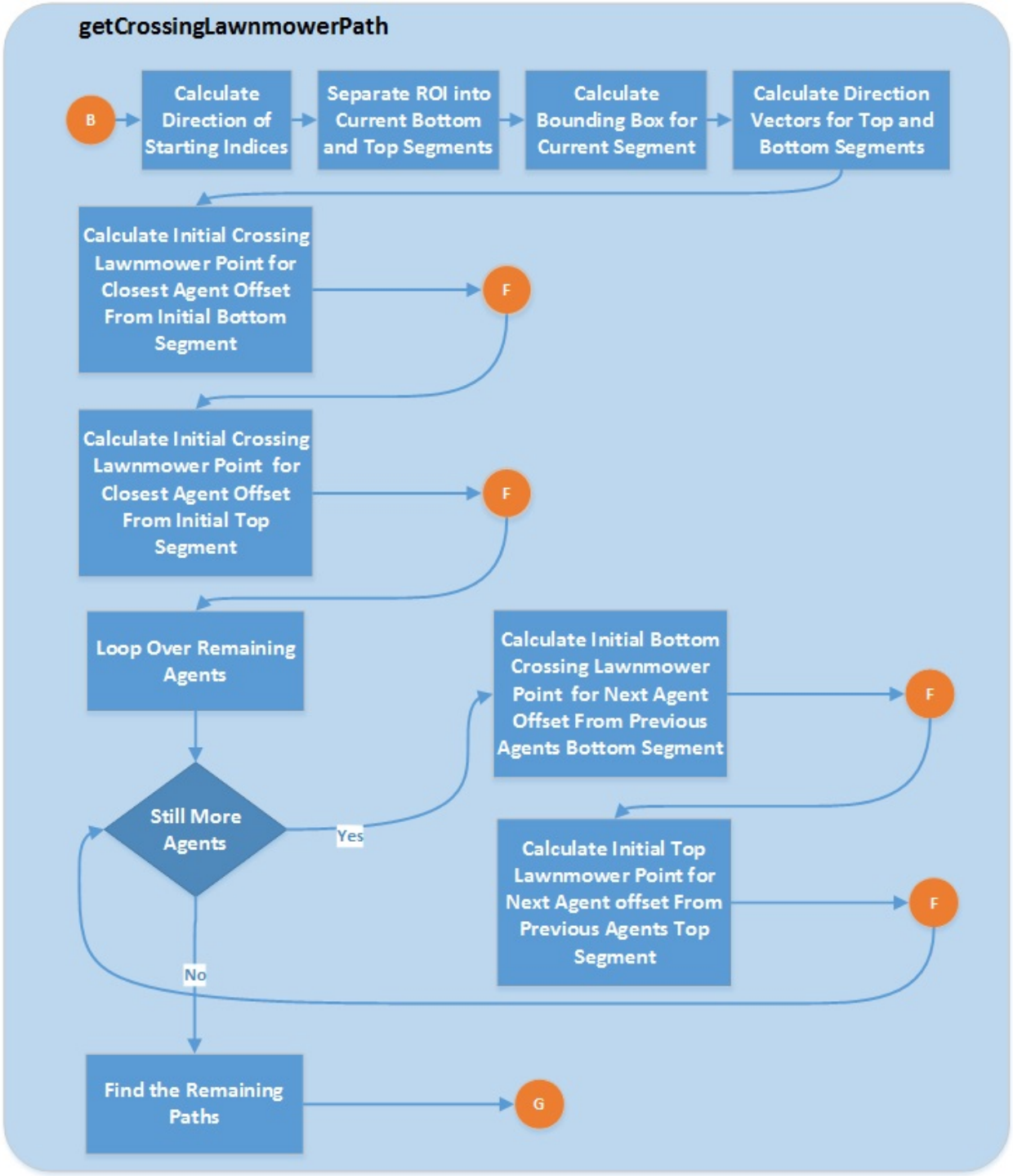


Figure 4-44. Container to collect points of a crossing lawnmower path in goal_planner

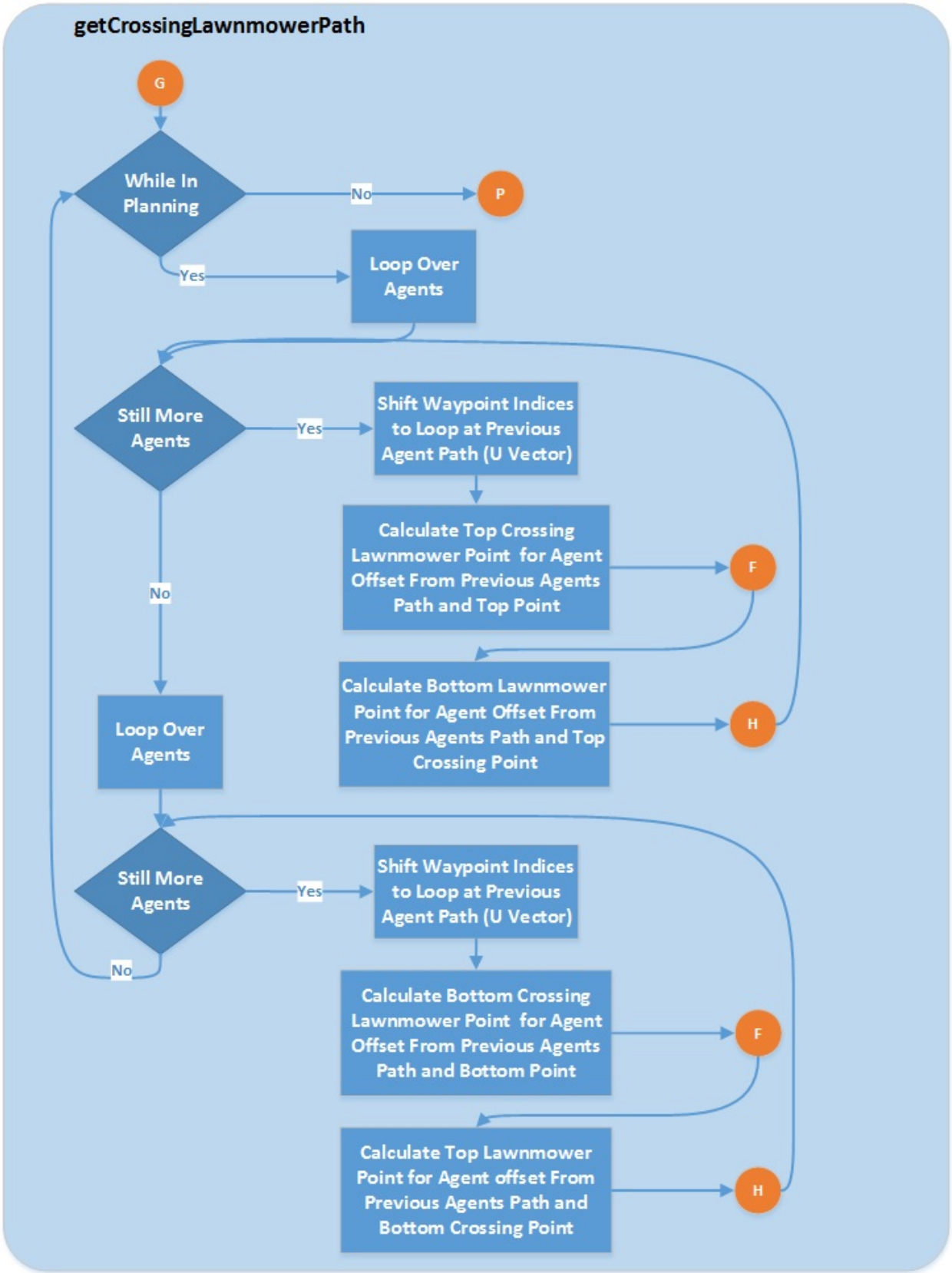


Figure 4-44. Container to collect points of a crossing lawnmower path in goal_planner (continued)

know what direction within the ROI the path will travel. This is used to calculate the initial bottom, top, and side segments.

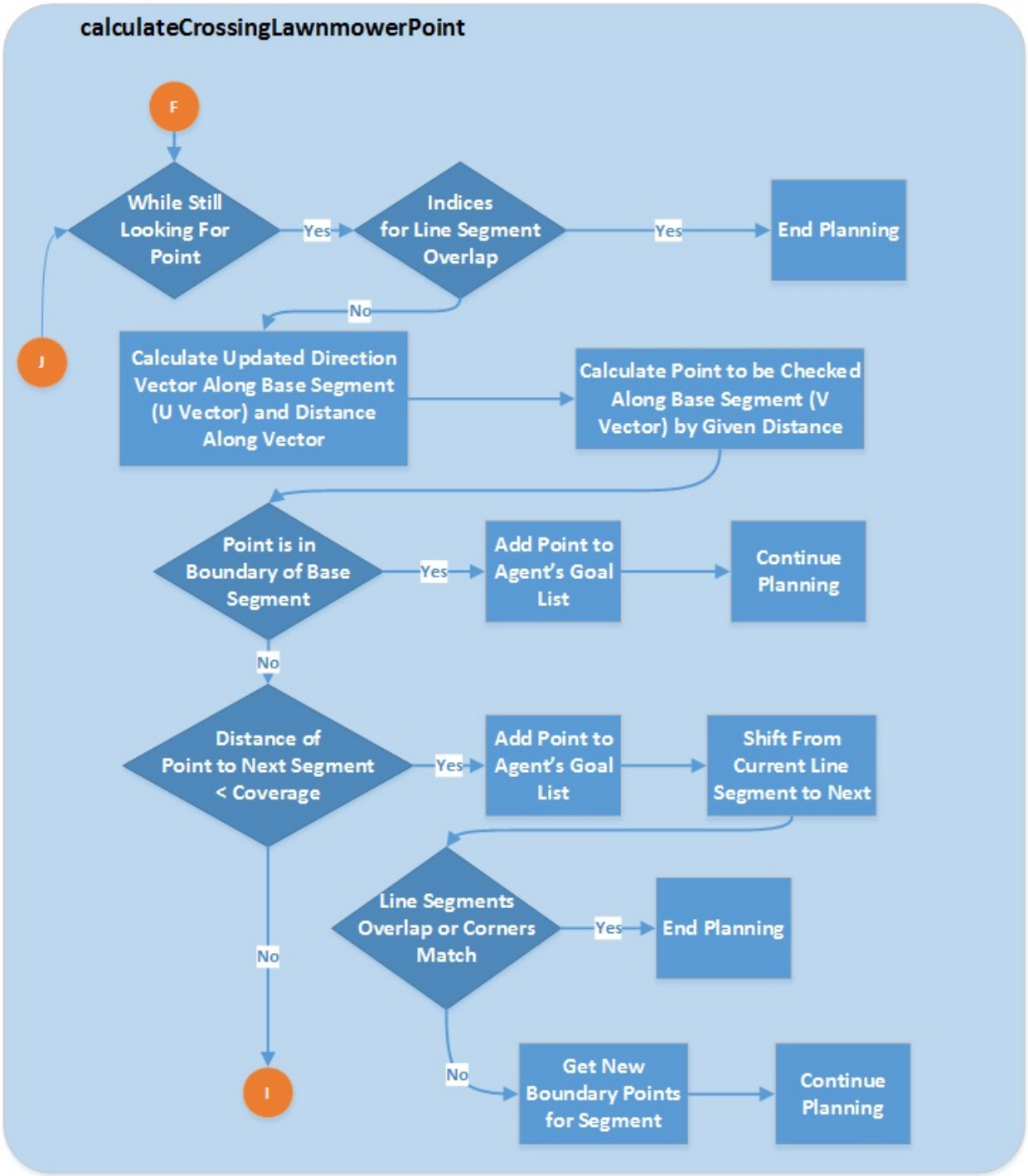


Figure 4-45. Method for calculating a crossing lawnmower point in goal_planner

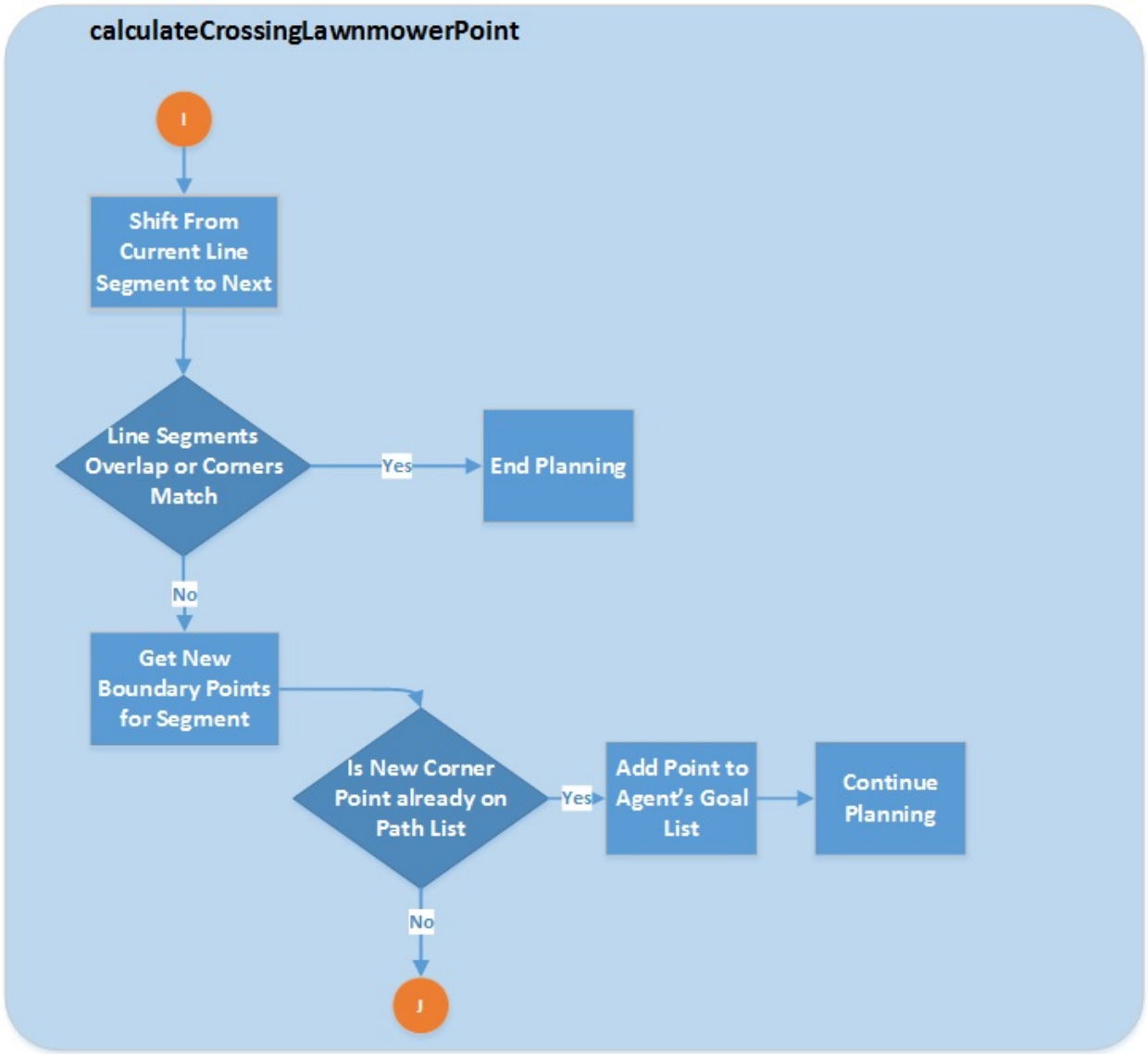


Figure 4-45. Method for calculating a crossing lawnmower point in goal_planner (continued)

Once all needed data is collected, the initial path goal is calculated given a set distance away from the starting corner along the bottom segment. Since this is the first point, it is treated as a crossover point from the side segment and is found using the method described in Figure 4-45. The method is used for calculating any crossover points needed to generate the path.

This method first confirms that the indices do not overlap or generate a path that would go no where. Next a point is calculated along the bottom segment away

from the side segment. This crossover point, as well as all other crossover points and upward/downward points found along a top or bottom segment, is found through the sue of analytic-geometry.

First, the side segment is defined as vector **U** while the top or bottom segment is defined as vector **V**. Vector **U** is given as the equation of a line

$$ax + by + c = 0 \tag{4–1}$$

$$(y_1 - y_2)x + (x_2 - x_1)y + (x_1 y_2 - x_2 y_1) = 0 \tag{4–2}$$

where (x_1, y_1) and (x_2, y_2) are defined by the end points of the side segment. **V** is defined in parametric terms

$$x(t) = mt + x_0 \tag{4–3}$$

$$y(t) = nt + y_0 \tag{4–4}$$

where point $\mathbf{P}_0$ is defined at the intersection of vectors **U** and **V**, or (x_0, y_0), and $< m, n >$ is defined as a vector parallel to it. $< m, n >$ is the normalized result of the end point and start point of **V** subtracted. This creates the desired distance vector that will later be used.

Given a desired distance, or coverage capability, between a point **P**=(x, y) and **U**, Equation 4–5 is defined, where **t** represents the distance along **V** to achieve the desired distance **d**.

$$\frac{|ax(t) + by(t) + c|}{\sqrt{a^2 + b^2}} = d \tag{4–5}$$

With the given assumptions, the equation is simplified to Equation 4–6

$$\frac{|(am + bn)t|}{\sqrt{a^2 + b^2}} = d \tag{4–6}$$

and solved for **t** in Equation 4–7

$$t = \pm \frac{d\sqrt{a^2 + b^2}}{am + bn} \tag{4–7}$$

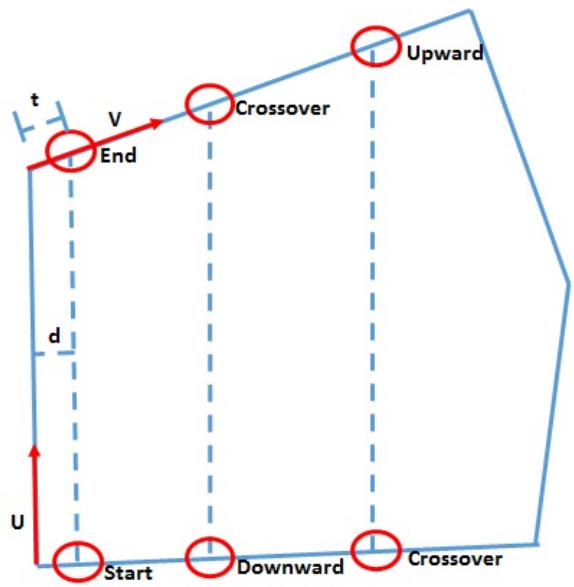


Figure 4-46. Analytic Geometry model used for planning paths

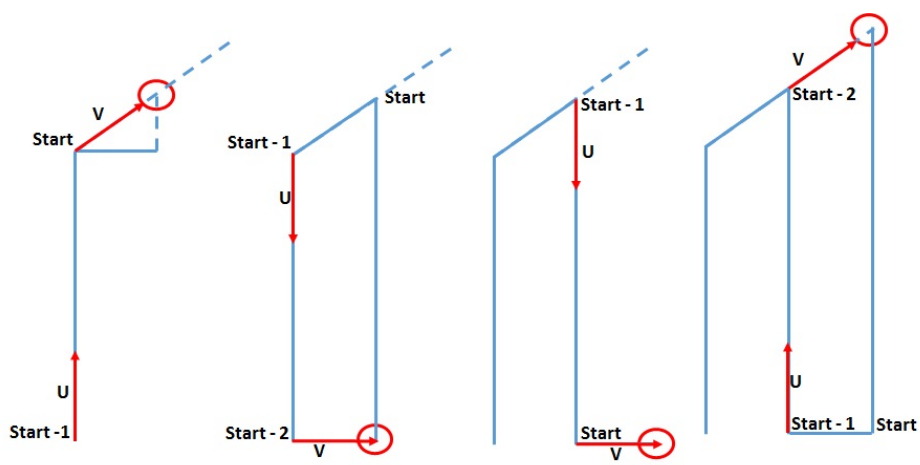


Figure 4-47. Series of stages that lead to find a full sequence for a lawnmower path

Only the positive result of **t** is needed. Once **t** is found, it is multiplied by the distance vector $< m, n >$ to find the point **P**=(x, y) along **V**. In the method being described, this point is defined as the crossover point. A reference for the above equations is given in Figure 4-46. Figure 4-47 provides an example of the first few stages when finding a lawnmower path, while Table 4-7 provides information on how **U** and $\mathbf{P}_0$ is defined.

Table 4-7. Description of stages for a lawnmower path from Figure 4-47

Stage	Vector **U**	Point $\mathbf{P}_0$
I	(start) - (start-1)	start
II	(start-1) - (start-2)	start-2
III	(start) - (start-1)	start
IV	(start-1) - (start-2)	start-2

The next step in the method is to confirm that the point lies along the segment defined by **V** (top or bottom segment). If the point is along the segment, it is pushed to the path list and planning is continues to look for the next point. If the point is not on the segment, the distance between the side segment and the next segment along the ROI is calculated. If the distance is greater than the agents coverage range, the corner point is added to the goal list. The segment being tested is shifted to the next segment in order along the ROI and is confirmed to be a new segment. If it is, new boundary points are collected and planning continues, otherwise, planning is finished.

When checking the distance between the side segment and the next segment along the ROI, if the distance is less than the coverage area, the line segment should be shifted to the next segment. Again, the new segment should be confirmed to be a new segment and the new boundary points calculated. Finally, if the new corner is not already on the list, add it and continue planning. If the new corner is on the list, run the method again and attempt to find a crossover point along the new segment.

With the first point found, the method returns to find the rest of the crossing lawnmower points, again shown in Figure 4-44. The second point is similar to the first as it is treated as an initial crossover point from the side segment. Due to this, it is found

using the same crossover lawnmower point function. This method is repeated over the number of agents included in the search mission, each using the previous agents path as the new side segment. This process generates each agents initial upward path through the ROI. The small change is now that a beginning path has been found, each new agent path uses the method for calculating an upward/downward lawnmower point.

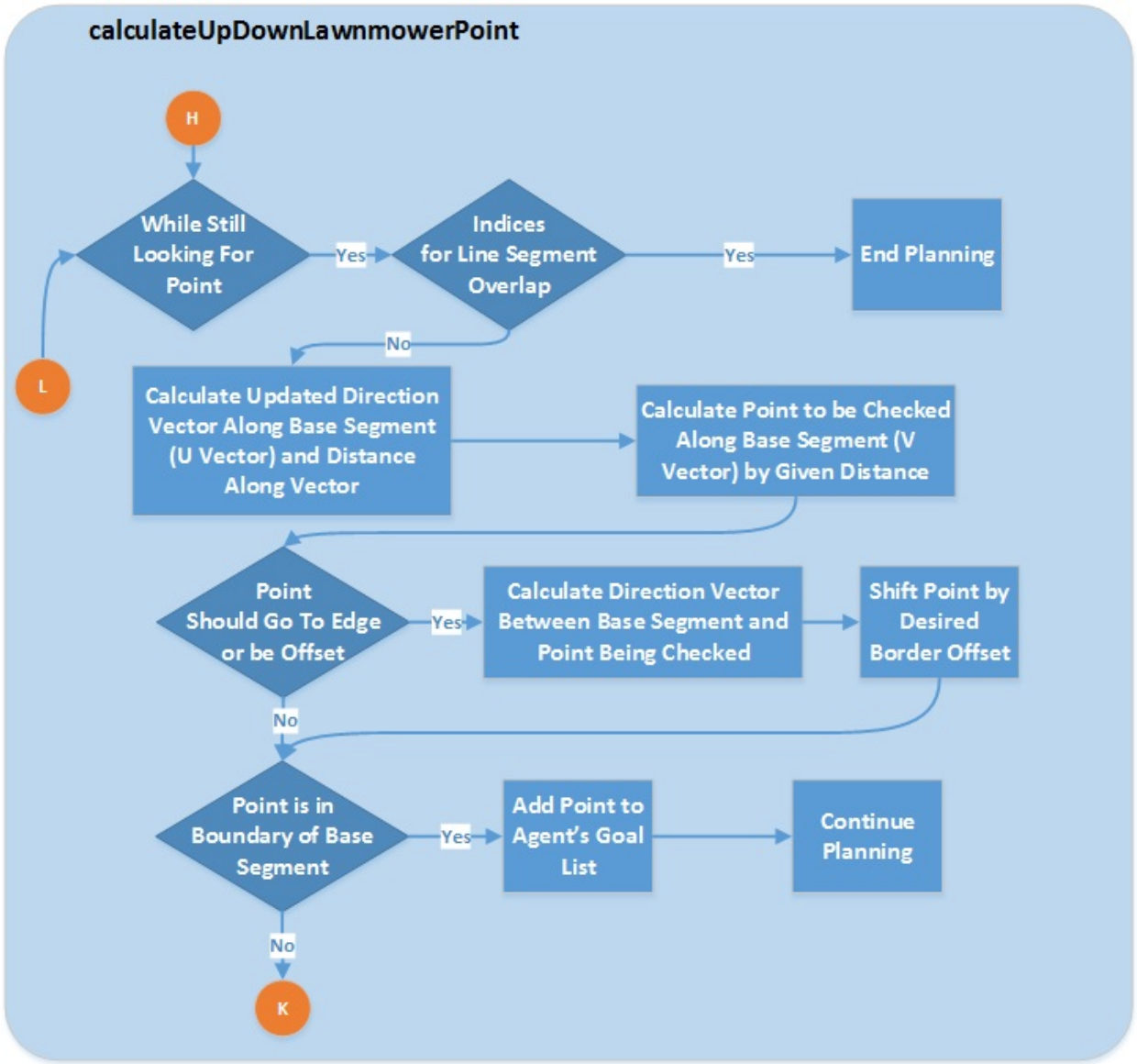


Figure 4-48. Method for calculating an up/down lawnmower point in goal_planner

There are two primary differences between the crossover lawnmower point method and the upward/downward lawnmower point method shown in Figure 4-48. Firstly,

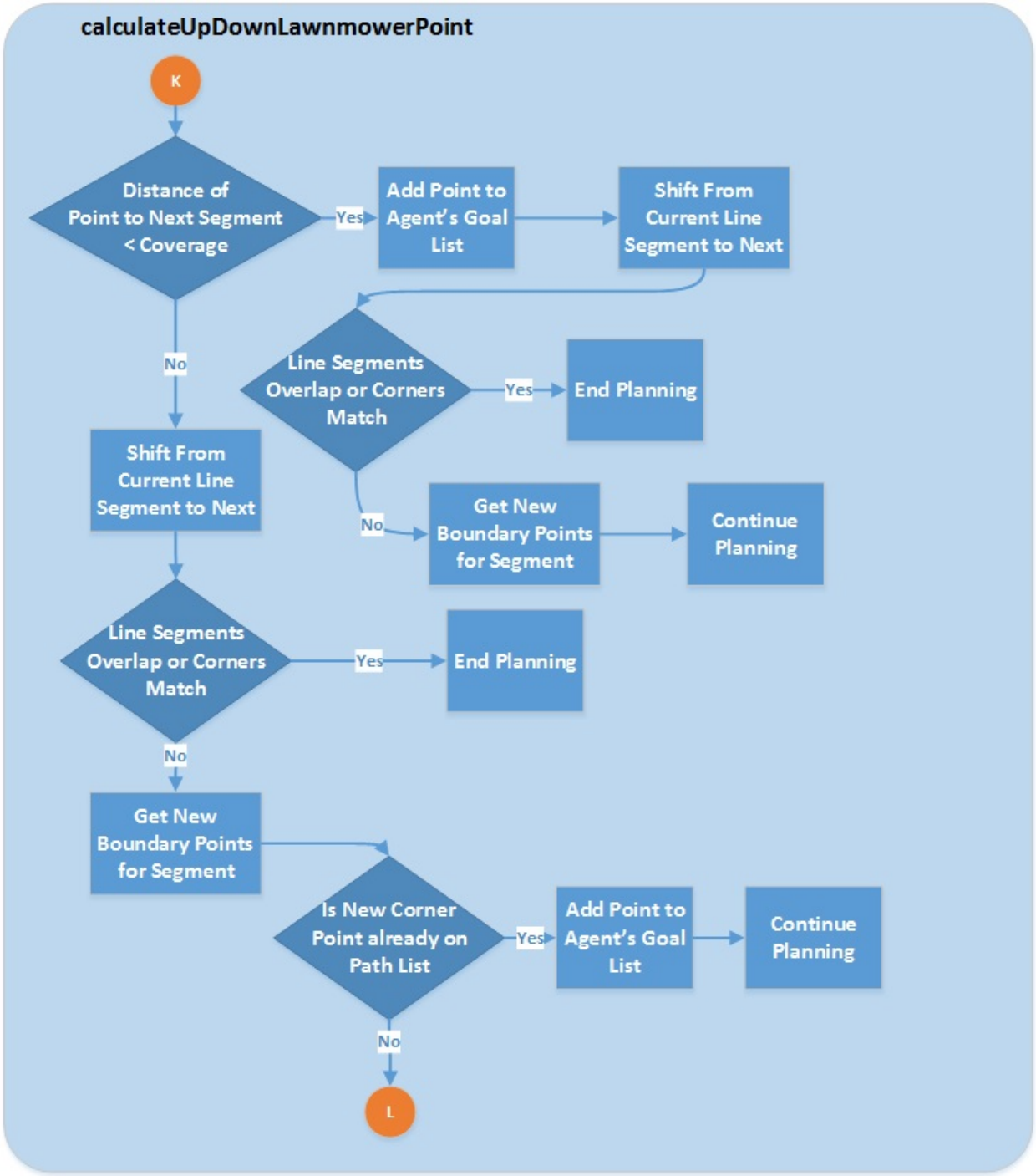


Figure 4-48. Method for calculating an up/down lawnmower point in goal_planner (continued)

when calculating the new point, the side segment chosen is the same as that used for calculating the previous crossover point, however, the direction of the side segment is reversed.

The next difference is whether an option for the point to be calculated at the edge or be offset. If offset, a temporary direction vector along the path being tested is calculated. The new point is shifted backwards along the direction vector at the given offset distance. This point or the point without the offset if not requested is checked in the same manner as the crossover lawnmower point.

After an initial upward path has been found for each agent, a loop is performed, looking for the crossover and downward path for each agent, followed by the crossover and upward path for each agent. This process is continued until the last possible path for the ROI is found, which then triggers all paths to be returned.

Along with the crossing lawnmower pattern, the split lawnmower path attempts to find a lawnmower pattern for each agent. The primary difference is that the polygon is split into smaller polygons before hand and the lawnmower pattern being found is for a single agent. This singular approach is shown in Figure 4-49.

The final pattern that may be found is the spiral path. This method only generates spiral paths per agent using smaller polygon regions. The method for finding a spiral path is described in Figure 4-50. Similar to the lawnmower pattern, the first step is to calculate the direction of the starting indices which is used to know what direction within the ROI the path will travel. This is used to calculate the initial bottom, top, and side segments. Next, the ROI polygon is checked to see if the points are defined in a clockwise or counter-clockwise fashion. Afterward, indices for the ROI are ordered in the direction that the spiral will be formed.

Now that all required information is found, the initial spiral point is found along the first side segment using the method shown in Figure 4-51. First, the boundary points for the side and top segments is found. Next, calculating the spiral point uses the same

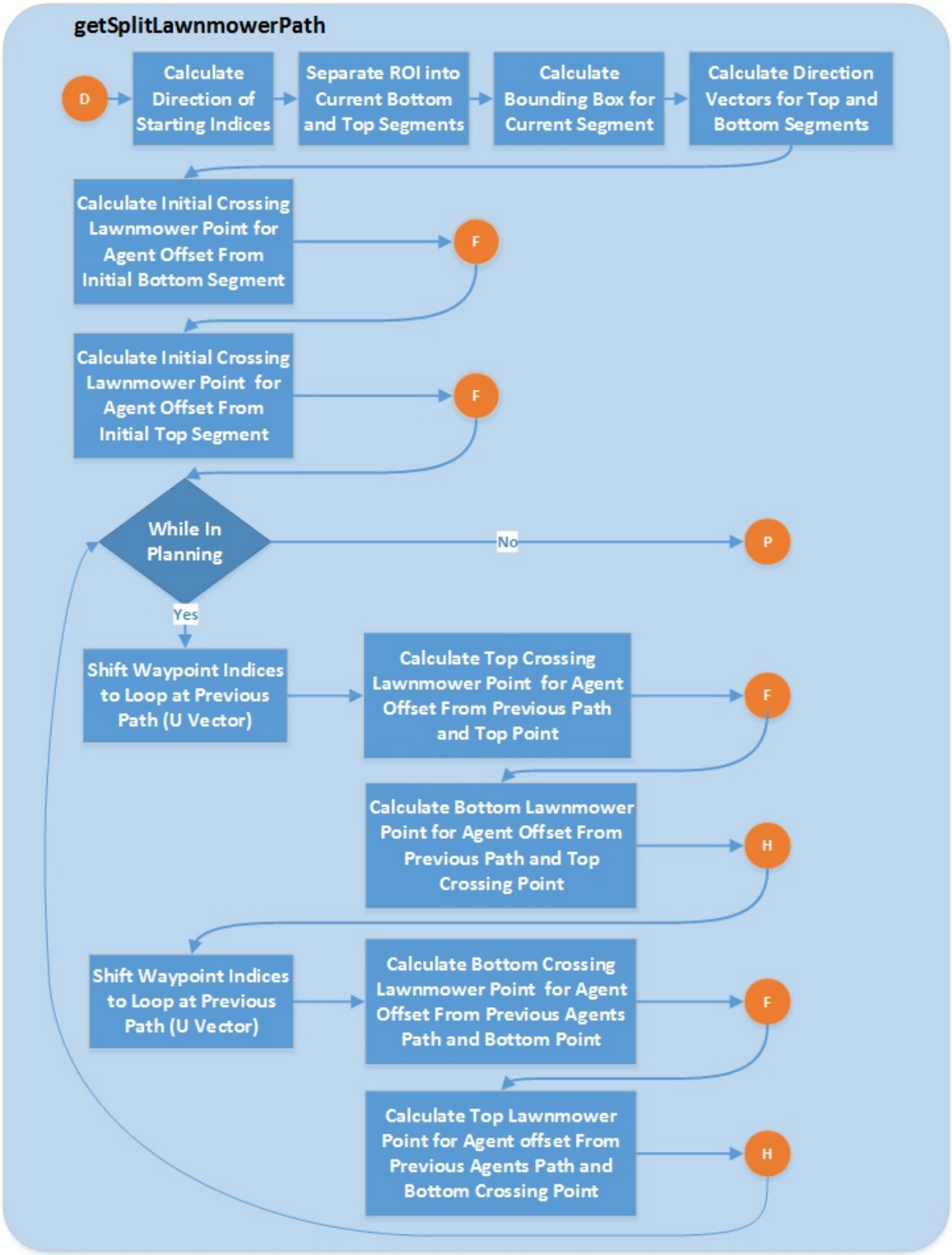


Figure 4-49. Container to collect points of a split lawnmower path in goal_planner

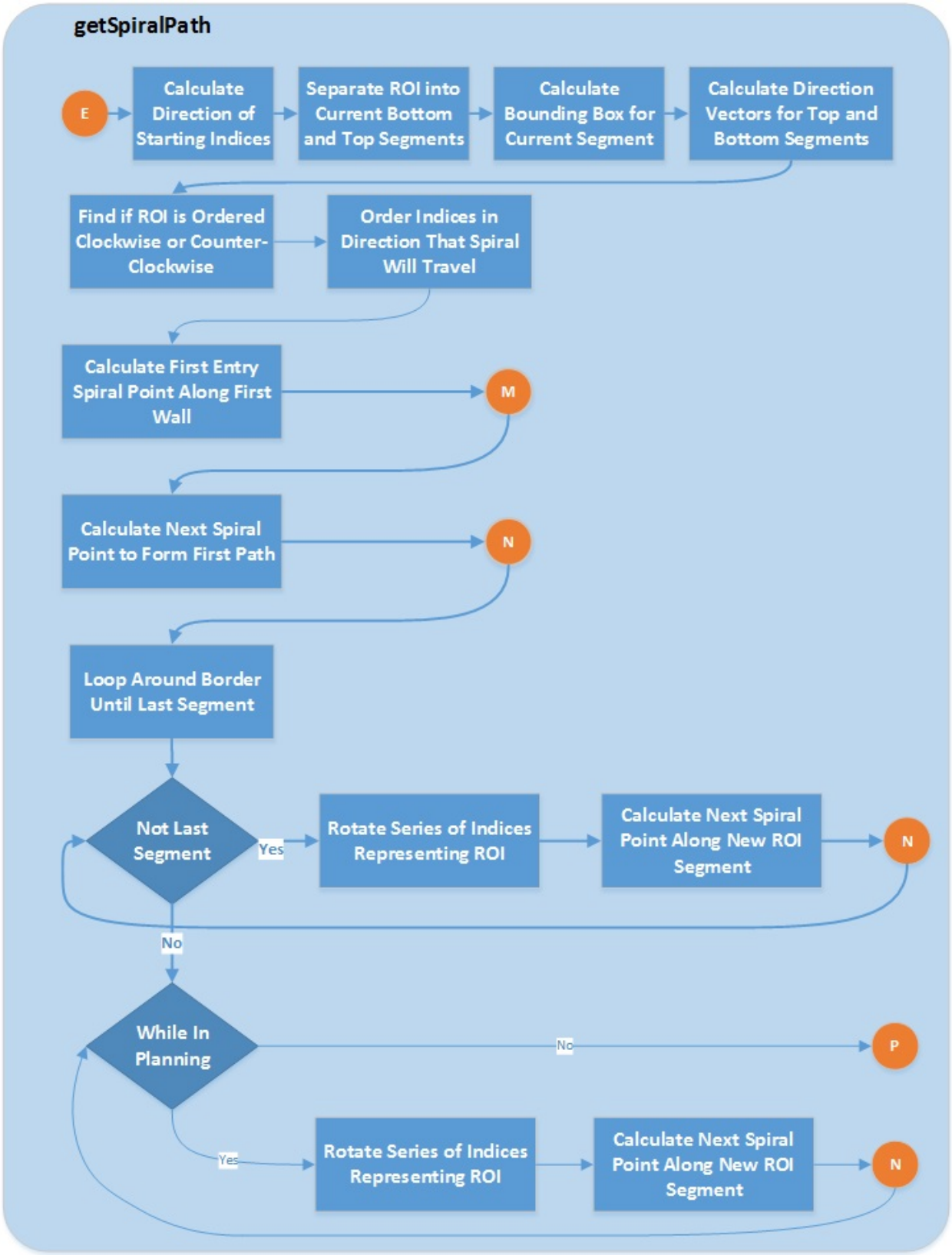


Figure 4-50. Container to collect points of a spiral path in goal_planner

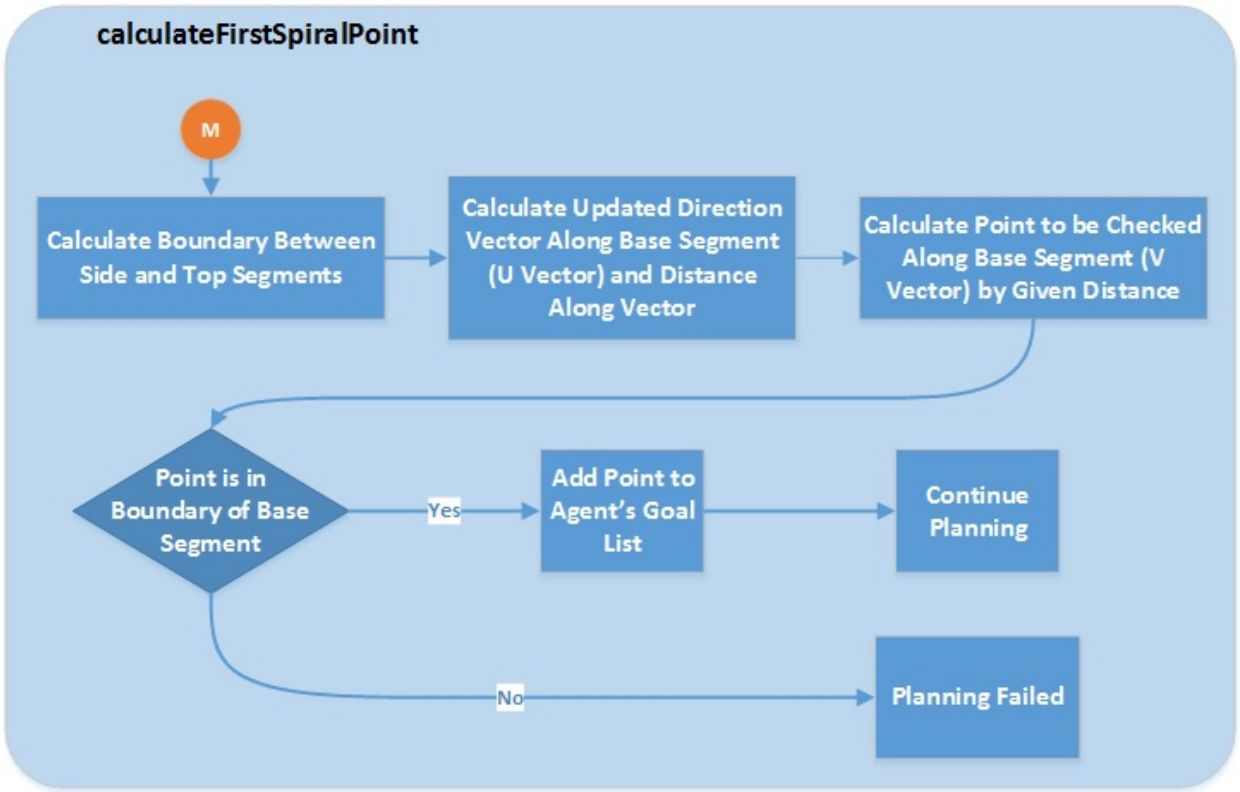


Figure 4-51. Method for calculating a first spiral point in goal_planner

analytic geometry method described for the lawnmower path in Equations 4–1 through 4–7. The point is confirmed be in the desired boundaries and is added to the path.

After the initial spiral point is found, another point is found to finish out the first path. This point uses the method described in Figure 4-52. Initially, it is confirmed that the overall length of the top segment is greater than the coverage area, otherwise planning continues with the next segment. A new point is calculated and confirmed to be in the boundary of the top segment. If within the boundary, a direction vector is calculated for the current path segment being processed. Then, a new point is found by traveling the distance of the agents coverage rate backwards along the path away from the segment. If the shifted point is on the correct side of the bottom segment, it is added to the path and planning continues.

If the point being checked was not originally within the boundary space, the distance between the side segment and top segment is calculated. If there is room, a new point

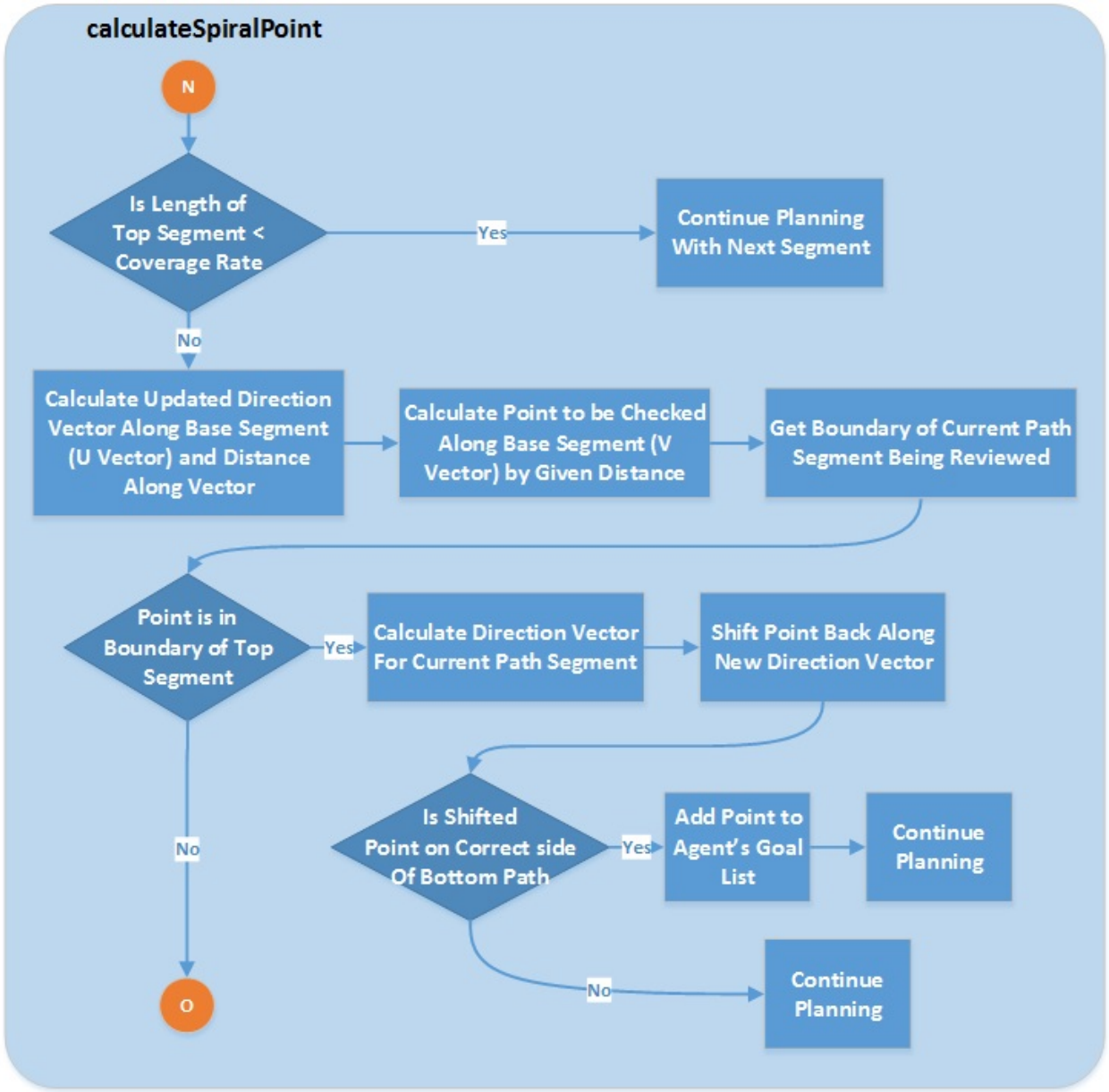


Figure 4-52. Method for calculating a spiral point in goal_planner

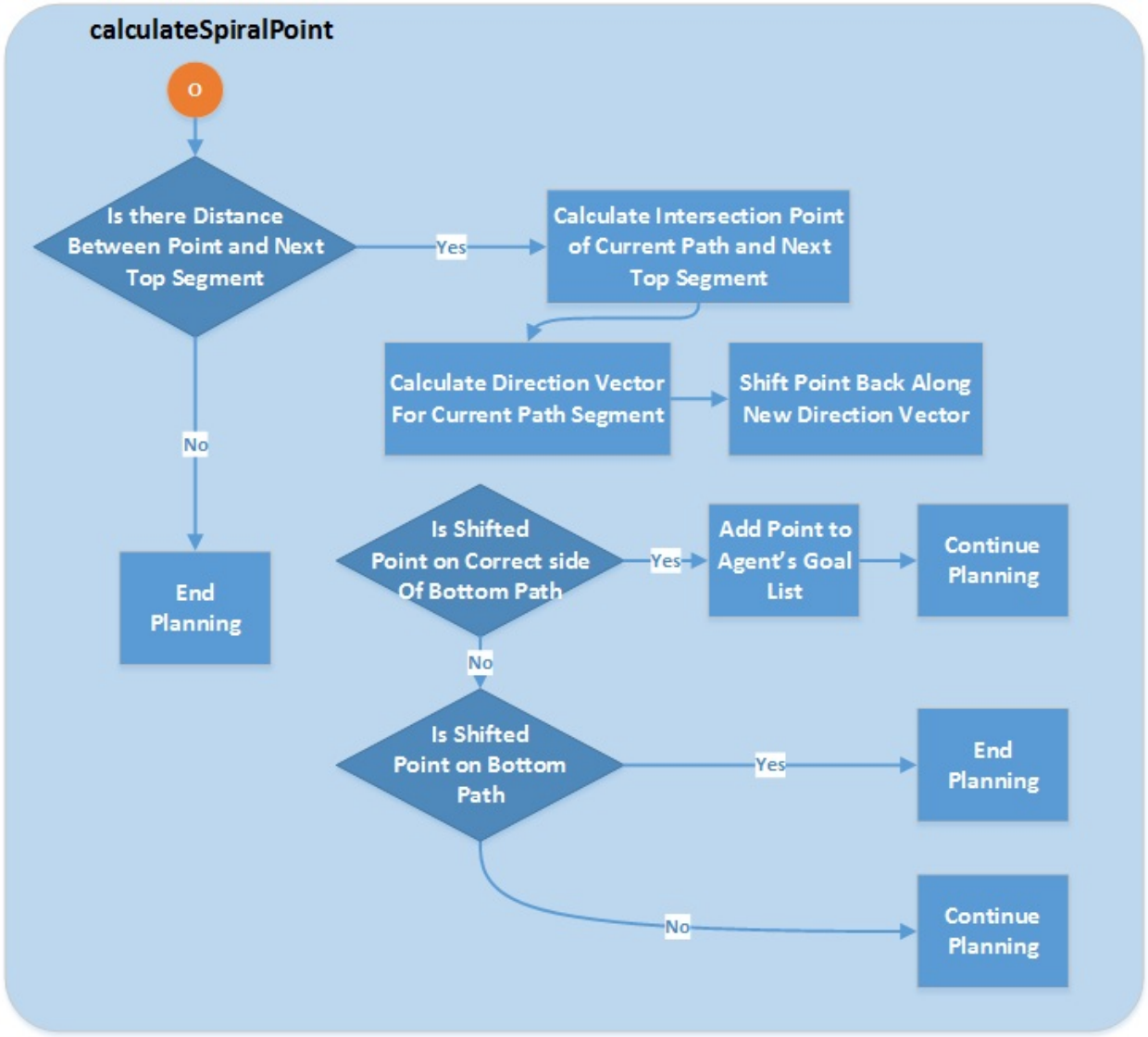


Figure 4-52. Method for calculating a spiral point in goal_planner (continued)

based on the corner of the ROI is chosen and confirmed to be acceptable. If the point is not acceptable, planning is finished. If there is no distance between the side segment and the top segment, planning is also finished.

After the first path for the spiral pattern is found, the series of indices is rotated to look at the next sequence of segments and calculate a new path beside the ROI. This continues until the last segment. From this point, the series of indices is rotated, however, instead of looking at the ROI, the new indices point to the inner most path used

to generate the spiral. In this manner, the spiral continuously closes in on the centroid of the ROI. See Figure 4-53 for an example of the first stages in creating a spiral pattern.

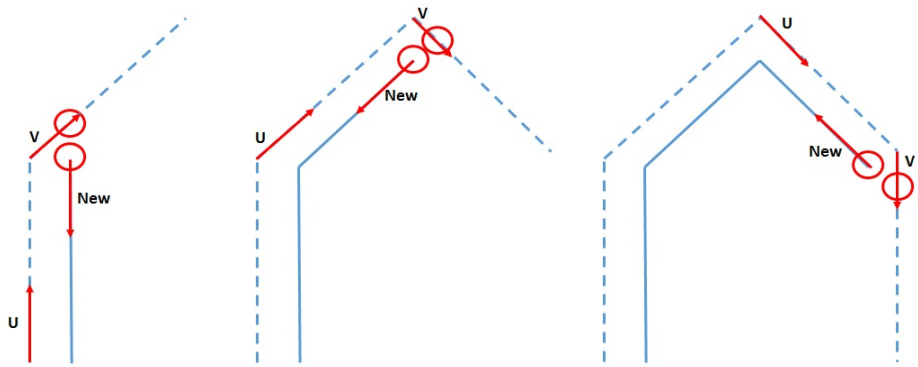


Figure 4-53. Series of stages that lead to find a full sequence for a spiral path

4.4.6 Base Vehicle Control

As discussed in the Agent Control package section, the ***agent_control*** node expects that two plugin libraries are created using the interfaces defined in the Swarm Base package. The Vehicle Control package defines one such plugin library that is used within the ***agent_control*** node. This plugin library is tasked with the control of the agent during a mission using direct vehicle interaction. The primary vehicle type used in the defined swarm are vehicles that use the APM autopilot controller, therefore, the described plugin library is dedicated to nodes meant specifically for the APM.

The Vehicle Control Package creates multiple ROS nodes that are used to control both air and ground vehicles tasked with various mission types. These nodes will either define commands to be directly processed by the Vehicle Driver package, or will define goals to be used and performed within another vehicle planning package such as the Navigation package.

4.5 Agent Capabilities

To allow the agents a series of capabilities throughout the swarm architecture, various behaviors were written using functions involving LIDAR and Vision.

4.5.1 LIDAR

Ground agents use the Hokuyo URG-04LX-UG01 LIDAR for detecting obstacles within the environment. Obstacle data is collected either to be transmitted into the swarm for other vehicles to use, or for obstacle avoidance while traveling through the environment.

4.5.1.1 Obstacle Detection

Obstacle detection is performed in multiple stages. First, data is received from the LIDAR and transformed to a Point Cloud. The transformation takes into account the orientation transform between the vehicle and the angle that the LIDAR is looking out into the environment. Next the data is searched to extract blocks of points that represent an obstacle. To reduce the amount of information required to represent an object, each collection of points representing a block is passed through a looping filter that uses Equation 4–8 to compare points and see if they are collinear. A given threshold may be modified to allow better detection in a nosier environment. Obstacles are reduced to endpoints that represent the lines that make up the object.

$$|(y_1 - y_2)(x_1 - x_3) - (y_1 - y_3)(x_1 - x_2)| <= threshold \tag{4–8}$$

After detecting objects, the current orientation of all endpoints is transformed in relation to the vehicles heading relative to the cardinal coordinate system. The GPS position of the vehicle is converted to UTM and the vehicle's position is added to each endpoint to create an accurate UTM position for each endpoint. Endpoints are then stored to be reviewed on subsequent scans. When a new scan is received, all points are converted again, and then compared to previous collections of points. Segments are compared to previous segments using four combinations to see if they are collinear. The combinations are defined for two segments (p_1, q_1) and (p_2, q_2) in terms of checking collinearity between (p_1, q_1, p_2), (p_1, q_1, q_2), (p_2, q_2, p_1), and (p_2, q_2, q_1).

As obstacles are defined and no longer seen, their endpoints are converted from UTM to GPS positions and transmitted to vehicles to use.

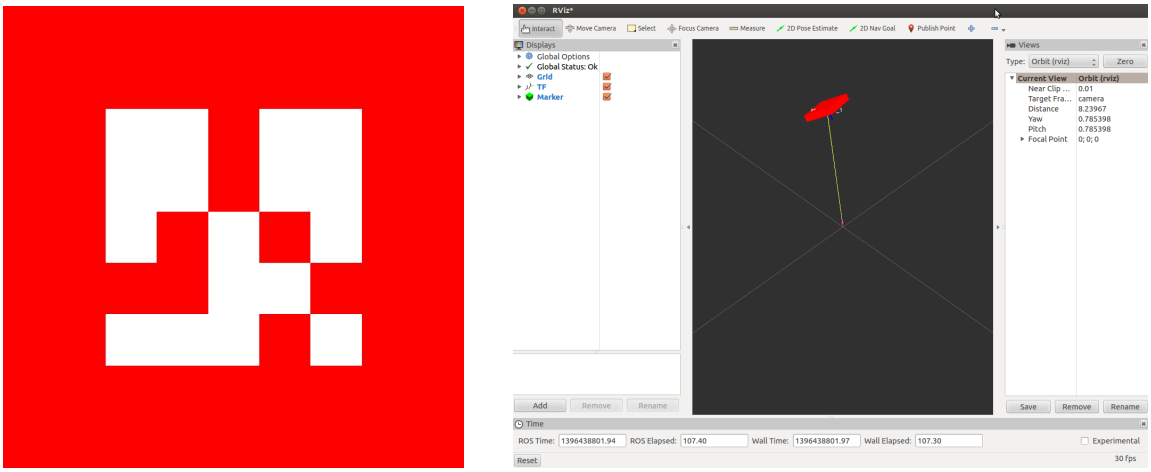
4.5.1.2 Obstacle Avoidance

During the process of detecting obstacles, a LIDAR scan is used to also check for objects within a threshold. As objects are detected, a PID controller is used to navigate around the obstacle.

4.5.2 Vision

Vision is used on both ground and air vehicles to locate possible targets and identify them. A number of UAVs use vision to identify a UGV and attempt to follow it using fiducial marker tracking. During this process, a UAV may attempt to locate possible targets to be avoided or identified. UGVs use vision when classifying an object as something that must be rescued or reviewed by a user.

4.5.2.1 Fiducial Marker Tracking



A B

Figure 4-54. Fiducial Marker Image and Visualization Within the ROS Environment: A) Fiducial AR Marker B) ROS Visualization of Fiducial Marker Position and Orientation

UAVs track UGVs in the environment with the aid of a fiducial marker tracking system. With the assistance of ***ALVAR***, an open source augmented reality (AR) tag

tracking library, a fiducial marker, as shown in Figure 4-54A, is created with a specific number representation. Each UGV is then given the numbered AR tag.

Before processing may be completed on-board an UAV, cameras are calibrated using a classical black-white checkerboard pattern to detect the distortion matrix and camera matrix. The distortion matrix is used for the rectification process of taking the camera ‘ and outputting a clearer image. The camera matrix is used to hold the focal lengths and optical center of the camera that may be used in calculating position of objects in an image.

When an UAV is started, it will link up and travel to a known position of a specific UGV. Once the link is made, not other vehicle will link up between the two that made the original link. When the UAV has reached the desired location, the camera image, along with the distortion and camera matrices, are passed to the AR tag library to locate and extract the AR tag. 3D position and orientation for the tag is located and transmitted back the the UAV. Using this information, the UAV then attempts to follow the vehicle in the environment using a simple PID controller that controls roll and pitch. For AR tag tracking, coupling heading to the UGV is not required as the camera is downward-facing. The known width and calculated location of the object is used as a reference during obstacle detection.

4.5.2.2 Obstacle Detection

While UAVs are traveling through the environment, they are constantly attempting to locate obstacles or targets within the environment. As these targets are located, their estimated position is calculated with respect to the AR tag’s location and current size on the screen. Since the AR tag’s physical size is a known parameter during the mission, the number of pixels that make up the tag are used to estimate the distance between objects given the number of pixels and angle to the object.

CHAPTER 5
FIELD TESTING RESULTS, CONCLUSIONS, AND FUTURE WORK

5.1 Testing and Results

The following section details various assumptions made for testing as well as the results of testing.

5.1.1 Assumptions

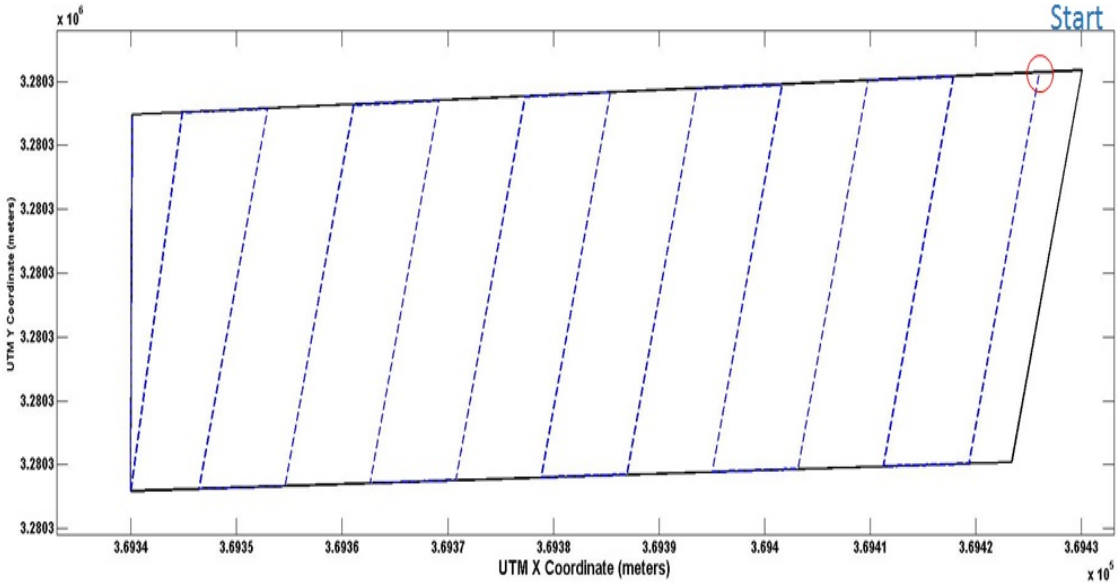
There were a few assumptions made for the cooperative system to function as intended. First, the outdoor environment is assumed to be an open field with no high structures that will cause collision issues with the UAVs. Tests are performed in a completely open field as to focus more on group behaviors versus UAV obstacle avoidance, which is an issue being solved by other works and is not a focus of this research. Further testing is desired in a more cluttered environment, in which trees and other obstacles may obstruct a UAVs view of the ground, but will be performed in follow up works.

Another assumption is that obstacles in the field of operation will be controlled. The vision used in testing does not follow computer vision or pattern recognition techniques to perceive complex objects, but instead is used to detect objects of specific shapes and colors. These different objects will pertain to either obstacles or targets of interest depending on the technique used. Along with the obstacles in the field, special patterns and colors known as AR tags are used for tracking the agents to assist in localization.

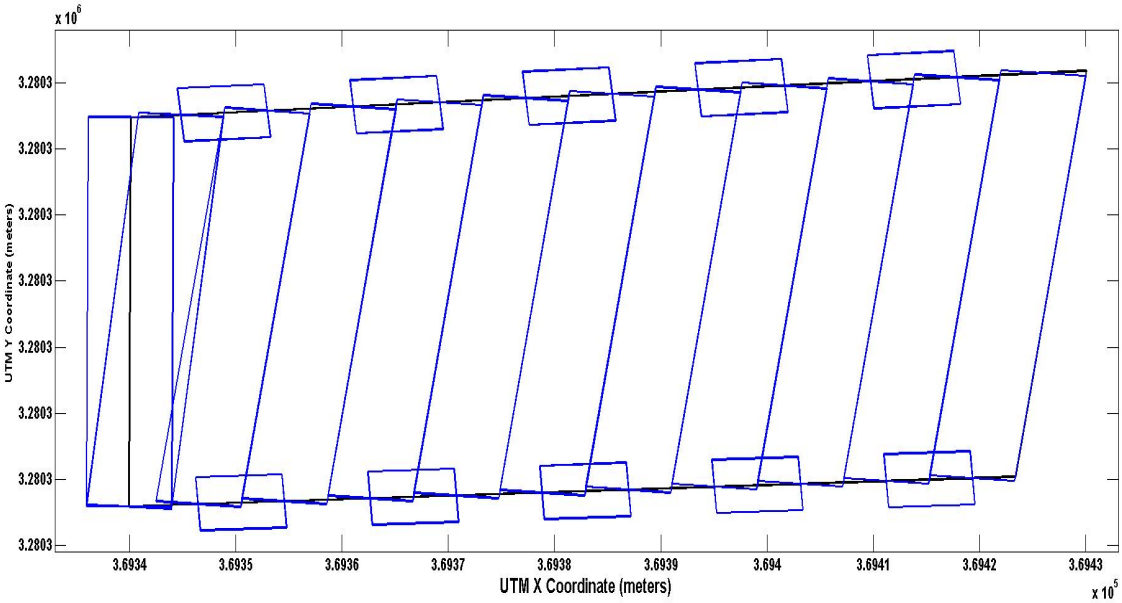
5.1.2 Systems Testing

Before environment testing, various capability tests were performed on the components of the architecture. Communications involved the ability of the ***xbee_bridge*** to handle any type of topic or service that ROS could generate within the architecture. Testing involved generating multiple messages internally and externally to be transmitted in a single direction between vehicles while reviewing loss of data. Next, various services were also created to test the systems ability to send a service request from one

vehicle to another vehicle that would handle the request, process it, and transmit the response back to the vehicle.



A



B

Figure 5-1. Single Vehicle Lawnmower Plan and Coverage Range: A) Single Vehicle Lawnmower Plan B) Coverage Range of a Single Vehicle Lawnmower Plan

As stated in Chapter 4, the ***xbee_bridge*** handles configuring communications through the use of a configuration file. Testing originally focused on whether these messages or services were generated automatically and correctly if the configuration file was properly configured. Tests also confirmed the robustness and speed at which messages were configured and transmitted, recognizing the speed at which they are generated within ROS and then transmitted and received correctly. Each of these tests showed that the ***xbee_bridge*** behaved as intended and handled all communications as required.

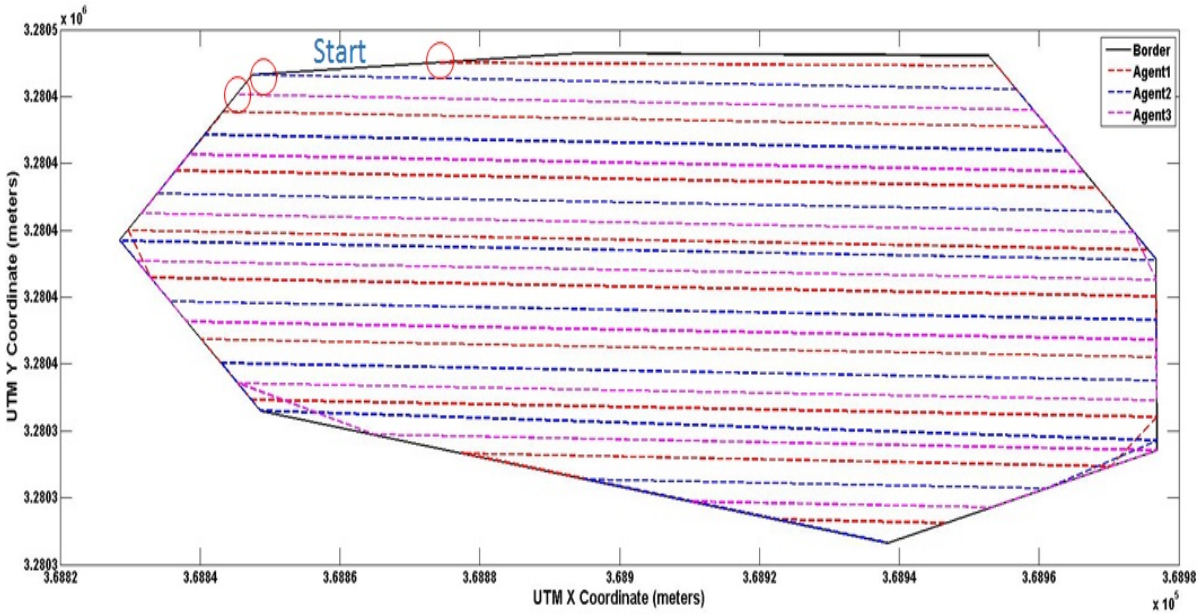


Figure 5-2. Planned Crossing Lawnmower Search Pattern for Three Agents in a Multi-sided Polygon

System testing also confirmed that the Base Goal Planner package created the various lawnmower and spiral patterns as desired. Various vehicles were added to the planner with varying search ranges and capabilities to test how a plan was generated between single and multiple vehicles. Initially, the ability to generate a plan and have complete or near complete coverage was confirmed. Figure 5-1A demonstrates a simple desired lawnmower plan for a single vehicle, while Figure 5-1B provides the expected coverage rate of the vehicle while performing the mission.

Multiple vehicles were given simulated positions from a real environment with actual search range and capabilities to generate the possible types of paths that may be generated. Figure 5-2 shows three vehicles that start near the top left corner of a multi-sided polygon and prepare to perform a crossing lawnmower search. Each vehicle has a different search capability; from shortest to largest range, the order agent 1, agent 3, and agent 2. It can be seen as goals are generated that various paths between vehicles may be very close together or further apart due to the capability of the agent. Goals crossover each other as the vehicle is expected to traverse the environment, while paying special attention to occasionally double cover a corner to attempt one hundred percent coverage.

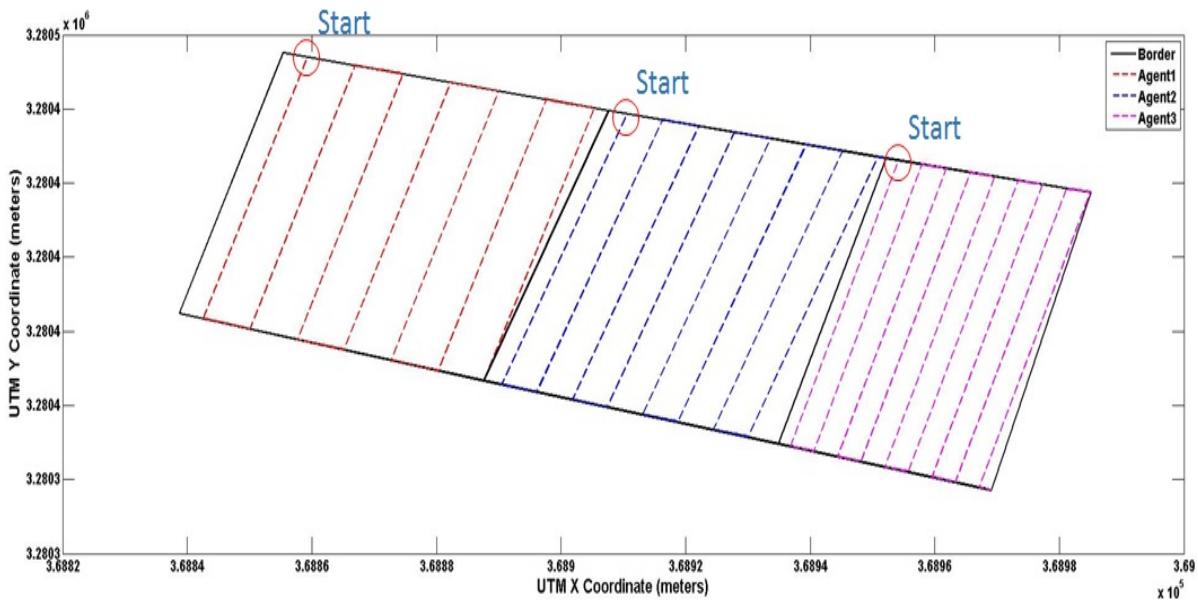


Figure 5-3. Planned Split Lawnmower Search Pattern for Three Agents in a Simple Polygon

Two other search paths demonstrate the split lawnmower and spiral patterns, as shown in Figure 5-3 and Figure 5-4 respectively. Each pattern is extracted from a single polygon that is split into three smaller polygons based on a ratio defined by each agents capability. Agents are shown to have varying ranges of search, in order from highest to lowest, agent 1, agent 2, and agent 3. All three vehicles start at the top left of the search

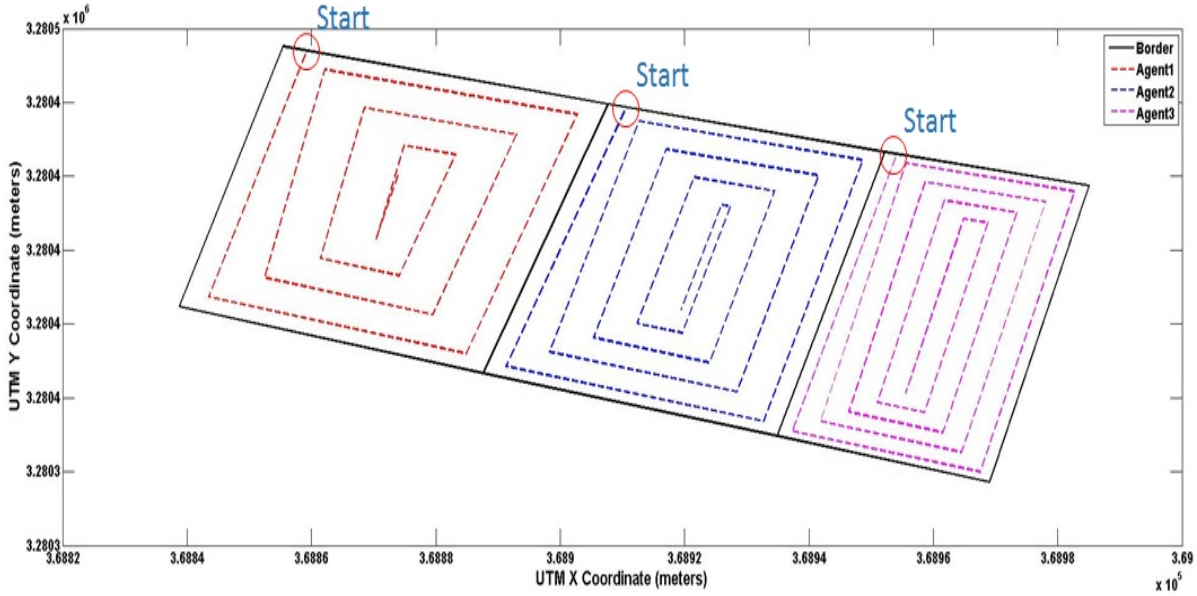


Figure 5-4. Planned Split Spiral Search Pattern for Three Agents in a Simple Polygon

area, and, therefore, begin there planned search at the top left of their designated sub-region.

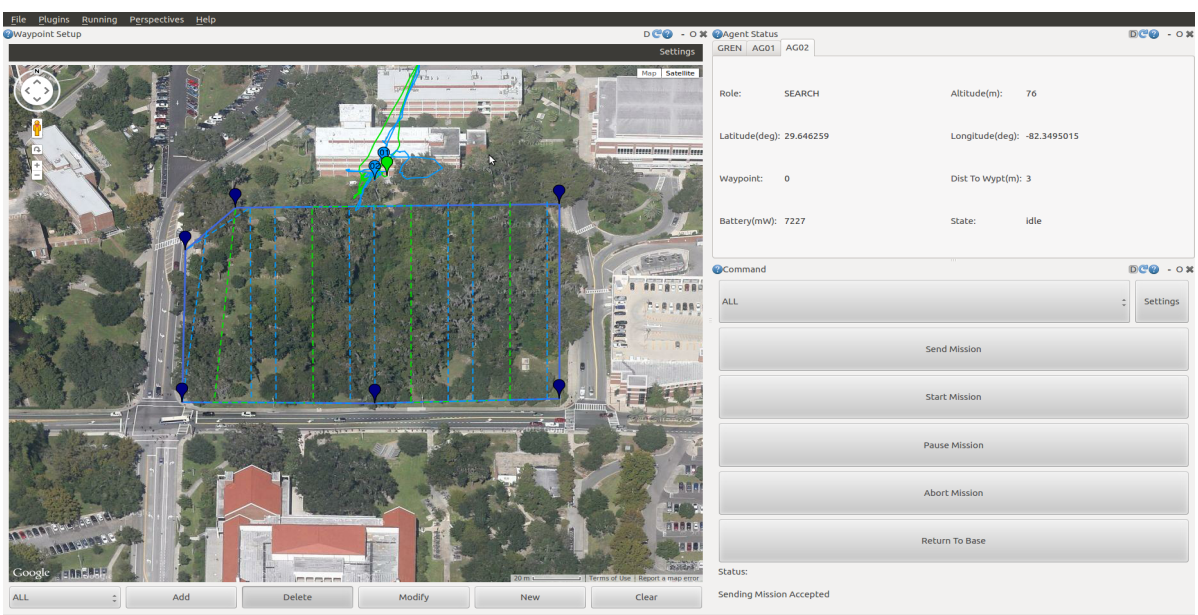


Figure 5-5. Planned Crossing Lawnmower Pattern for Two UGVs and One UAV within the Swarm Architecture GUI

An example of a mission plan created with the swarm architecture GUI is shown in Figure 5-5. The image shows a planned mission using two ground vehicles (blue dashed lines) and one air vehicle (green line) preparing to perform a crossing lawnmower pattern.

The final systems test included multiple iterations of testing LIDAR and vision components in varying situations with changing structures in controlled and uncontrolled lighting situations. Occasional testing was performed within a multi-vehicle simulation within a custom configured Player-Stage simulation and a master slave ROS setup. Each test confirmed the same results as discussed in Chapter 4 in which each system was able to detect the desired results as needed.

Beyond simply locating objects with vision, the vision tracking system was tested between an UAV tracking an UGV. The UAV vehicle was setup with the track behavior and the UGV was setup with the search behavior. At mission start, the UGV would be given an ROI and plan a search mission. When the mission was started, the UGV would hold position until the linked UAV sends a command for it to begin the desired search. The UAV would hold position until the GPS location of the UGV was received, triggering the UAV to launch and travel to the UGV's location. When at the desired location, the UAV would use vision to locate the AR tag on top of the UGV while telling the UGV to start it's mission. The two vehicles navigated through the environment, the UAV following the UGV.

5.1.3 Field Testing

After the primary components of the architecture were fully tested in a controlled manner, multiple field tests were performed to view each behavior in a real environment. Early testing involved various issues in regards to controllability of vehicles when performing real behaviors. For UGVs, the worst case failures involved a ground vehicle attempting to leave the search area at high speeds, for UAVs, however, worst case failures typically included increasing altitude drastically until taken under remote control

(RC) by a safety flyer, or decreasing altitude quickly until the vehicle crashed. After changing some configurations with the control of the vehicles, as well as enabling and turning the thresholds of fail-safes that were created at the beginning of the design for the architecture, safety of the vehicles was increased.

Testing involved a single laptop acting as the base station. The swarm architecture GUI was running on the laptop which also ran the ***mission_control***, ***roll_call*** , ***xbee_bridge*** nodes. Communications were handled via XBee device and antenna. Both UAVs and UGVs ran the various vehicle controllers, ***agent_control***, ***roll_call***, ***xbee_bridge***, ***vision***, and ***lidar*** nodes. Within agent control , the required mission planner and behavior plugins were implemented to handle the vehicles various roles based on the desired mission. Earlier testing did occur on Wi-Fi networks where the base station was the master and the vehicles were the slave, however, for primary testing, the XBee's and multi-master system were used. All testing occurred at the University of Florida in an open field named Flavet Field, GPS position latitude:29.6464592, longitude: -82.3543872.

5.1.4 Initial Architecture Testing

During the initial tests UGVs and UAVs separately to get a baseline for control. Three UGVs were given search patterns to navigate through a region of interest. The various fail-safes and commands of the GUI were tested to confirm that the UGVs responded as expected while the mission was underway, including re-planning when required. A similar mission was performed with two UAVs that employed 3D navigation and a larger capable search area.

A follow up test combined both ground and air vehicles. Two UGVs and one UAV were combined to create the swarm, each set with search roles. UGVs were configured with search ranges of five meters while UAVs were configured with search ranges of ten meters; each range dictated by the abilities of the LIDAR and vision camera systems respectively. Figure 5-6 shows a picture of two UGVs and one UAV before the test

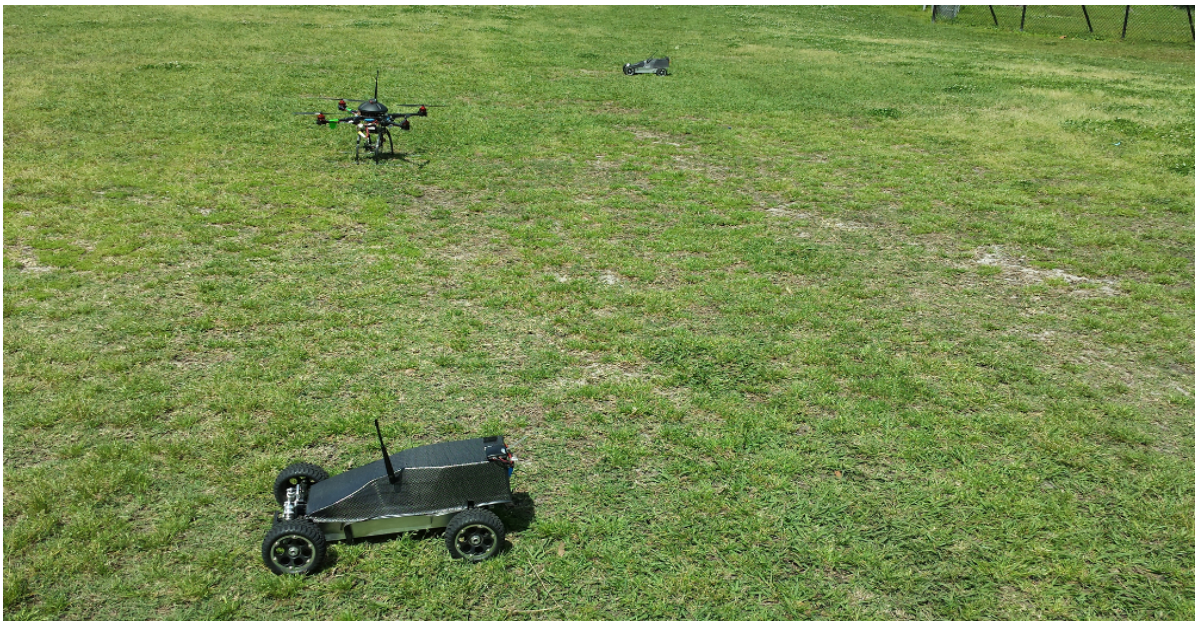


Figure 5-6. One UAV and Two UGVs Being Used in Field Testing

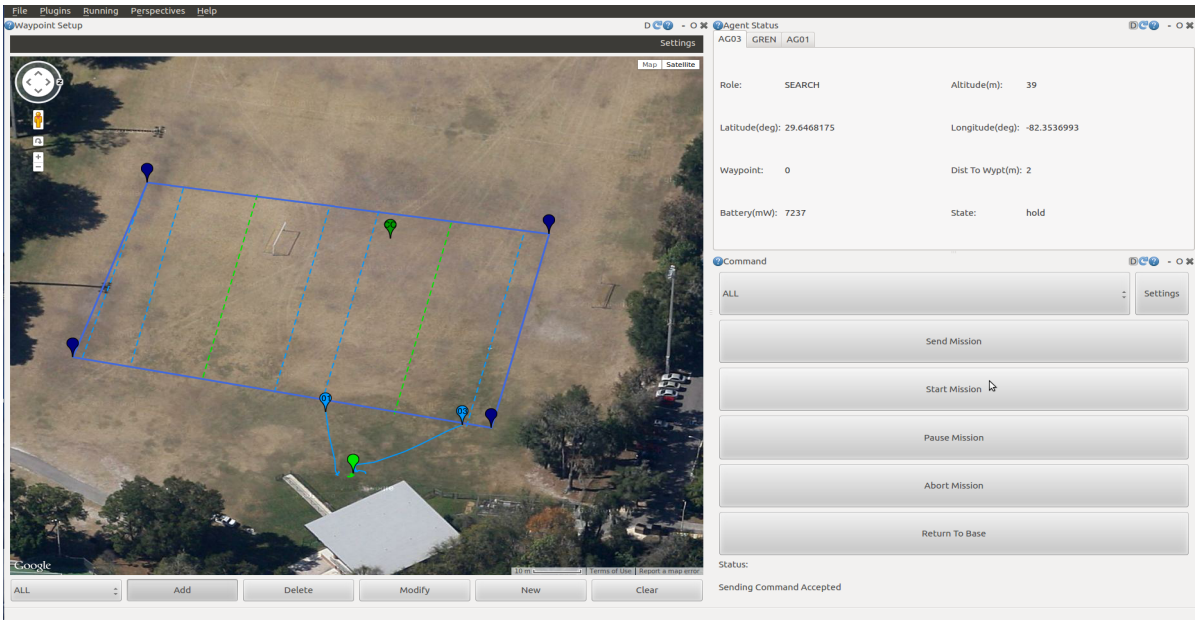


Figure 5-7. Mission Start Showing ROI and Planned Search Paths for Each Agent

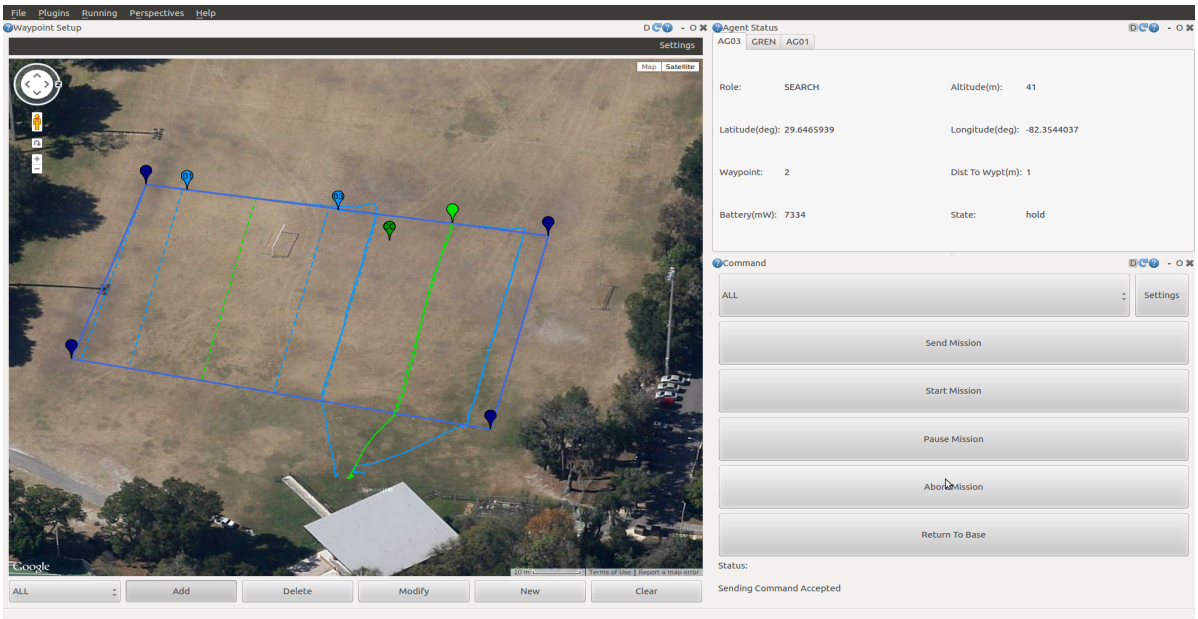


Figure 5-8. Mission After Completing One Leg of Planned Search Paths

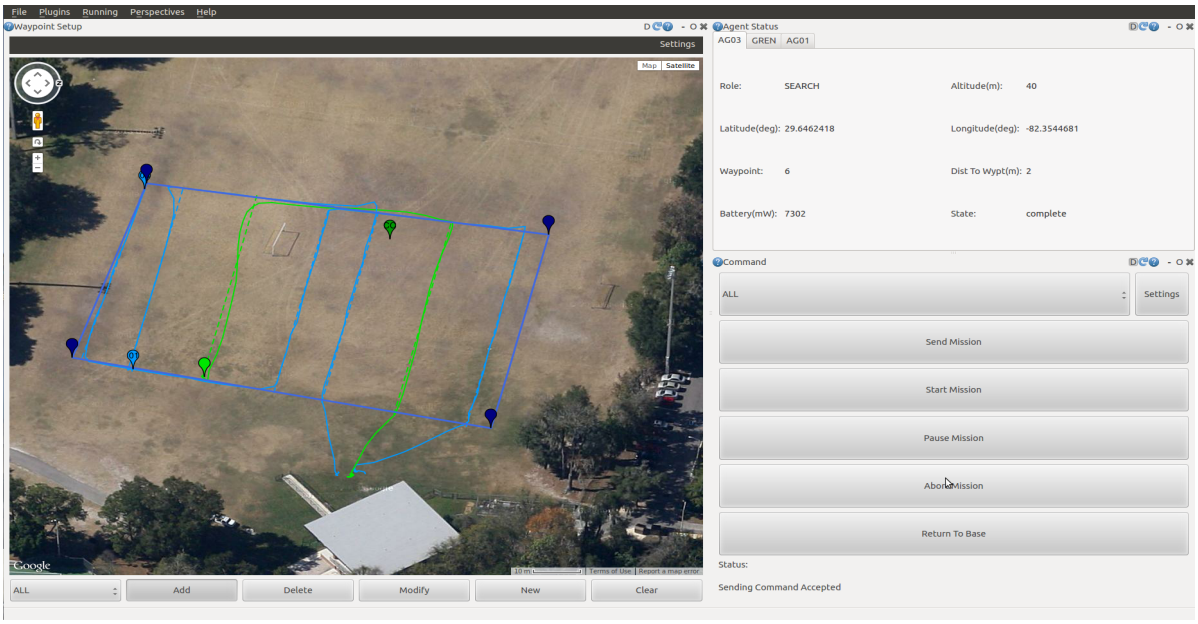


Figure 5-9. Mission Complete Showing Agents at Final Goal Point in Search Pattern

begins. Figures 5-7, 5-8, and 5-9 show three separate phases of the mission. The first figure focuses on the displaying the ROI and the planned goal path for the two UGVs (blue dashed line) and UAV (green dashed line). The second figure is taken of mission progress on the base station after each vehicle has completed the first leg of the search and are traveling to the next leg. The final figure displayed that results of the vehicles following the planned path and reaching the final goal on their list.

After this test was run multiple times to confirm consistency, the centralized structure was traded for a decentralized setup. As described in Chapter 4, the code was written in a method where mission control and the base station is irrelevant of where it is run. The main mission control and base station code was moved to an UAV, while the original laptop acting as a base station was only used as a satellite station where the mission ROI was still chosen and results were displayed. This mission was tested and concluded with the same results as all previous tests. The primary differences noticed were a slightly slower run time and a small loss of data around 10%; each expected due to all XBee communications running through a single vehicle computer which is less capable than the laptop computer.

5.1.5 Complete System Test

The complete swarm architecture test involved a combination of three UGVs and two UAVs. A ROI was chosen at the testing grounds (Flavet Field) using the swarm architecture GUI. Two UGVs and a single UAV performed search behaviors, dividing the ROI into multiple paths based on each vehicles search capabilities. The second UAV was given a track behavior and configured to follow and assist the closest UGV. The final UGV was given the identify behavior, which was tasked with traveling to targets located by the vehicles during their search missions. Obstacles were defined as orange circular buckets in the ROI, while targets were defined as pink boxes. Figure 5-10 shows the initial setup of the field and the vehicles.

A

B

Figure 5-10. Initial Setup for Field and Vehicles During Complete System Tests: A) Field Setup B) Vehicle Setup

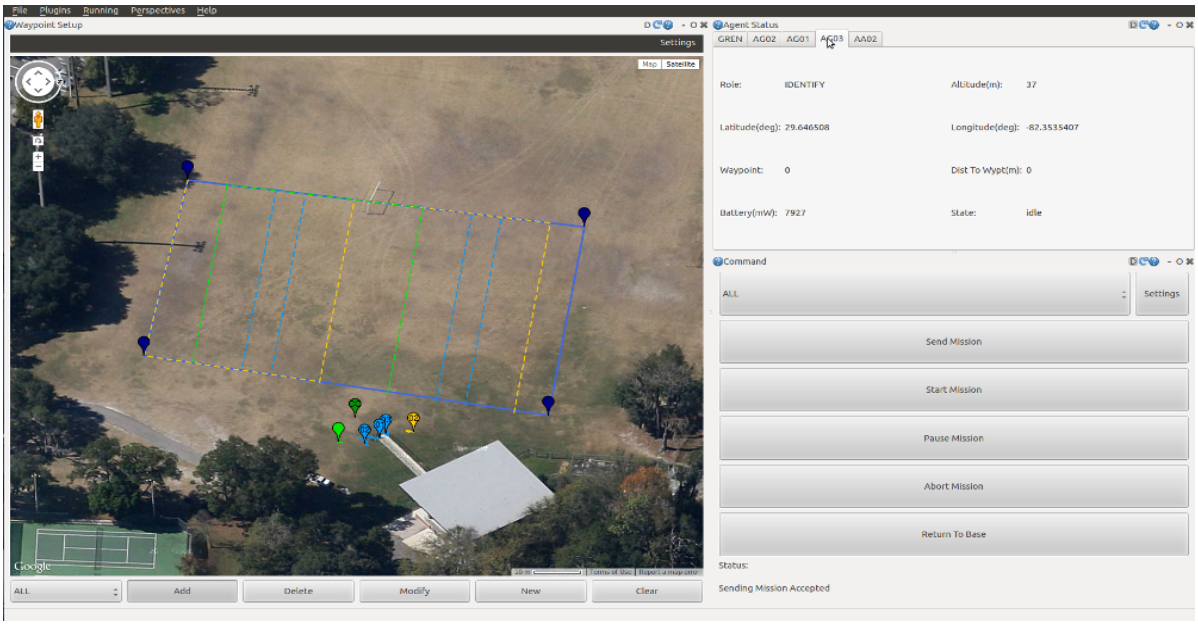


Figure 5-11. Initial Mission Setup for Complete System Tests

Figure 5-12. Mission Result for Complete System Tests Detecting All objects

As shown in Figure 5-11, the mission was configured with a specific ROI for the field. The planned paths for the search vehicles were found and displayed within the GUI. The mission was tested various times with different settings. During missions, each search vehicle began traveling through the environment. Both the UGVs and UAVs followed the calculated path as closely as possible, however, due to environmental issues including high winds and heavy cloud cover causing low GPS signals, the UAV occasionally strayed from the search path.

During the first experiments, UGVs and UAVs were tasked with locating pink boxes. UGVs used LIDAR to locate boxes by extracting the lines that make up the box. UAVs used vision to locate the pink tops of the boxes while flying over them. As the pink targets were located, the UGV with the identify behavior was tasked to travel to the target and attempt to identify it with vision.

After these experiments were completed, detecting the orange obstacles was added to the requirements to distinguish detection of the two types of objects. UGVs did have issues occasionally detecting the circular buckets, while the UAVs detected the objects and provided all vehicles with the obstacle data. UGVs were able to use the obstacle information to navigate around the obstacles in the environment. The final results of the search mission for all obstacles is shown in Figure 5-12.

5.1.6 Separate Group Testing

Due to a related project, another group at the University of Florida took advantage of the work described in this research. The project called for two quadrotors to work together to complete a task. The first quadrotor, acting as a search vehicle, would travel through an environment looking for a marker representing a target, in this case a large pink ball. The test was initiated by choosing a ROI for the search area within the swarm architecture GUI. From here, a spiral search pattern was chosen using the Base Goal Planner package and given to the quadrotor.

During the search, as soon as the marker was located, the GPS location of the ball was calculated and transmitted to another quadrotor acting as an identify vehicle. With the location of the marker found, the second quadrotor launched and traveled towards the marker as the first quadrotor returned to its launch position. The second quadrotor located the marker, dropped a payload at the location, and then returned back to its launch position.

Though the project did not specifically demonstrate the heterogeneous capability of the swarm architecture, it does show the overall capability. For their project, all that was simply needed was a vision function, two slightly modified behaviors for their vehicles, and an extra configuration line for the ***xbee_bridge***. The project could have also easily taken advantage of an UGV to travel to the marker position instead of an UAV, or for both to work together in tandem.

5.2 Conclusions

The work introduced in this dissertation is the first of its kind in terms of development for an architecture for a cooperative heterogeneous system of vehicles. The system described defines a heterogeneous set of platforms to be used in further research for the Machine Intelligence Lab (MIL). Various contributions have been made in the area of control and navigation of multiple ground and air vehicles using centralized and decentralized control schemes through the use of a central architecture. Other contributions include a ROS based multi-master communication system using the custom RF communication system between agents of the swarm, as well as advancements in task allocation and path planning for multiple vehicles in a convex defined polygon search area. Finally, a method for allowing air vehicles to follow and track ground vehicles using fiducial image methods as well as locating objects in the environment is provided. Each of these advancements has been described within the given architecture and shown to function through various field tests performed on multiple platform vehicles.

Each of the above contributions have been produced with a standard Berkeley Software Distribution (BSD) license and are provided online with the open-source ROS community for further use and developments.

5.3 Future Work

The work presented in this paper demonstrates the ground work for developing a fully capable swarm environment between varying types of ground and air vehicles operating in a real environment. The work also provides the beginnings of various behaviors that will be very useful in adding further capabilities to each vehicle, however, there are various ideas that were conceived before the research or during.

First, further advancements may be made in the capabilities of the swarm in regards to navigation and control of the vehicles. During the research defined in this paper, some work was started on introducing squad based control along with local Wi-Fi multi-master networks while using RF for long distance. If the swarm started with greater numbers it would initially be split into smaller groups with a leader per group. Planning for the swarm would then split the field into separate parts per squad. Internal communications would be handled between each other within a Mobile AdHoc Network (MANET). Communications to other squads or a base station would be collated and ferried between squad leaders. Much of this infrastructure was developed between the smaller homogeneous swarm work but was not included in the heterogeneous swarm architecture.

Next, work was performed on graph theory based navigation and Simultaneous Localization and Mapping (SLAM) for ground vehicles based on Ackermann steering. Starting with an unknown environment, UGVs used LIDAR to map obstacles or targets as they travel through the environment. As other vehicles may travel through this environment, they may use a global planner with gained knowledge of the environment to plan a path using graph theory based navigation. Then a local planner would be

used to assist in navigating the environment through the aid of LIDAR based obstacle avoidance.

One of the main components that was not able to be added involved planning for vehicles with the Base Goal Planner package and could be a focus for future work. Planning currently is only possible within a convex defined polygon, however, it would be very useful to allow the user to define a plan as a concave polygon.

Another focus could be placed on the type of missions to be performed. Time could be spent on further implementing the Area Survey mission that was used on the original homogeneous swarm work described in Chapter 3. A perimeter maintenance mission from previous work involving multiple UGVs and UAVs could be used to form an equidistant perimeter around a desired centroid, providing the entire outer perimeter with coverage. As the centroid of the perimeter traveled, the vehicles would follow. These ideas can easily be implemented through added behaviors that work within the swarm architecture.

A final focus of future work could focus on the individual vehicle capabilities used in this work. Vision was used on the UAVs to follow the UGVs throughout the environment while also attempting to detect obstacles in the environment. More advanced methods may be used to more accurately estimate the position of objects in the environment. Instead of identifying targets that are simple shapes and colors, better methods in pattern recognition may be used to detect human shaped objects for a better search and rescue system.

APPENDIX A
DYNAMICS OF A QUADROTOR

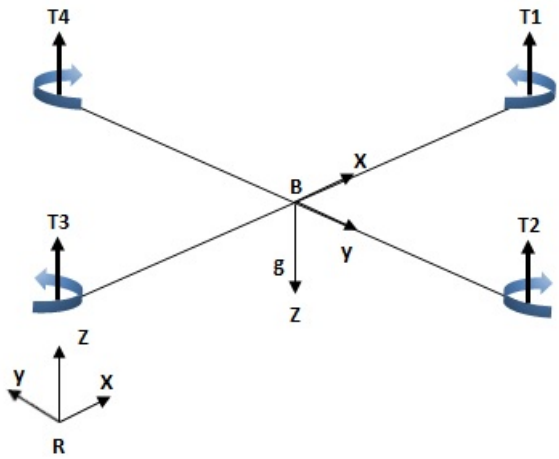


Figure A-1. Dynamic system model and reference frame.

Though the multirotors used in this research already include an autopilot that handled the controls for stabilization, the dynamics of a quadrotor was explored for an improved understanding before developing PID controllers used for high level navigation. The dynamic model for a quadrotor may be defined in two reference frames. The first frame is in terms of the Earth Fixed Frame (inertial frame) with ENU (East-North-Up) coordinate system. The second frame is a Body Fixed Frame (body frame) with NED (North-East-Down) coordinate system. The body frame matches the fixed frame's origin on the body of the quadcopter at the same point as the center of mass for the vehicle. An example of the frame setup can also be seen in Figure A-1.

To describe the dynamics of the 6 degree of freedom vehicle, various states of the system may be described. For the ability to track and know the location of the vehicle in relationship to some origin point, the position of the center of mass of the body frame in the inertial frame r_i, or the (x, y, z) point, and the linear velocity components in the body frame v_b, or (u, v, w), must be defined. For the ability to realize the vehicles attitude and orientation about all points of rotation at a given point, the orientation of the body in the inertial frame Φ, or Euler angles roll (ϕ), pitch (θ), and yaw (ψ), along with the angular

rotations in the body frame w_b, or (p, q, r) must be defined. The components and states are presented as

$$r_i = \begin{Bmatrix} x_i \\ y_i \\ z_i \end{Bmatrix} \quad \text{and} \quad v_b = \begin{Bmatrix} u \\ v \\ w \end{Bmatrix} = \begin{Bmatrix} \dot{x}_b \\ \dot{y}_b \\ \dot{z}_b \end{Bmatrix}$$

$$\Phi = \begin{Bmatrix} \phi \\ \theta \\ \psi \end{Bmatrix} \quad \text{and} \quad w_b = \begin{Bmatrix} p \\ q \\ r \end{Bmatrix} = \begin{Bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{Bmatrix}$$

$$x = \begin{bmatrix} r_i & \Phi & v_b & w_b \end{bmatrix}$$

$$x = \begin{bmatrix} x & y & z & \phi & \theta & \psi & u & v & w & p & q & r \end{bmatrix}^T \tag{A–1}$$

The controls for the system may be defined by four separate inputs (T_1, T_2, T_3, T_4) which may be described in four different combinations.

- Collective thrust for the system.

$$u_1 = (T_1 + T_2 + T_3 + T_4) \tag{A–2}$$

- Roll or the moment about the x axis of the system dependent on the speeds of the second and fourth prop (d is distance from prop to center of mass).

$$u_2 = d(T_4 - T_2) \tag{A–3}$$

- Pitch or the moment about the y axis of the system dependent on the speeds of the first and third prop (d is distance from prop to center of mass).

$$u_3 = d(T_1 - T_3) \tag{A–4}$$

- Yaw or the moment about the y axis of the system dependent on the speeds of all props.

$$u_4 = k_{tm}(T_1 - T_2 + T_3 - T_4) \tag{A–5}$$

The following equations were created with reference to [26] and [31].

Kinematics

Three rotations based on the geometrical locations for Φ shown in Figure A-1 can be used to move components from the inertial frame to the body frame by generating a coordinate transformation matrix.

$$R(\phi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & c\phi & s\phi \\ 0 & -s\phi & c\phi \end{bmatrix} \quad R(\theta) = \begin{bmatrix} c\theta & 0 & -s\theta \\ 0 & 1 & 0 \\ s\theta & 0 & c\theta \end{bmatrix}$$

$$R(\psi) = \begin{bmatrix} c\psi & s\psi & 0 \\ -s\psi & c\psi & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$S = R(\phi)R(\theta)R(\psi)$$

$$S = \begin{bmatrix} c\theta c\psi & c\theta s\psi & -s\theta \\ s\phi s\theta c\psi - c\phi s\psi & c\phi c\psi + s\phi s\theta s\psi & s\phi c\theta \\ c\phi s\theta c\psi + s\phi s\psi & c\phi s\theta s\psi - s\phi c\psi & c\phi c\theta \end{bmatrix} \tag{A–6}$$

The components of the body frame may be characterized in the inertial frame using Equation A–6, or vice versa using a transpose of Equation A–6.

$$\begin{Bmatrix} x_b \\ y_b \\ z_b \end{Bmatrix} = S \begin{Bmatrix} x_i \\ y_i \\ z_i \end{Bmatrix}$$

$$r_i = \begin{Bmatrix} x_i \\ y_i \\ z_i \end{Bmatrix} = S^T \begin{Bmatrix} x_b \\ y_b \\ z_b \end{Bmatrix} \tag{A–7}$$

Equation A–7 also demonstrates that if velocity components in the body frame V_b are known, then velocity components in the inertial frame may also be found.

$$\begin{Bmatrix} \dot{x}_i \\ \dot{y}_i \\ \dot{z}_i \end{Bmatrix} = S^T \begin{Bmatrix} u \\ v \\ w \end{Bmatrix}$$

$$\begin{Bmatrix} \dot{x}_i \\ \dot{y}_i \\ \dot{z}_i \end{Bmatrix} = \begin{bmatrix} c\theta c\psi & (s\phi s\theta c\psi - c\phi s\psi) & (c\phi s\theta c\psi + s\phi s\psi) \\ c\theta s\psi & (s\phi s\theta s\psi + c\phi c\psi) & (c\phi s\theta s\psi - s\phi c\psi) \\ -s\theta & s\phi c\theta & c\phi c\theta \end{bmatrix} \begin{bmatrix} u \\ v \\ w \end{bmatrix} \tag{A–8}$$

Euler Angles

Equation A–8 may be integrated to find the position of the body frame in inertial space, however, values for ϕ, θ, and ψ must be known. These values change over time and define the Euler rates $\dot{\phi}$, $\dot{\theta}$, and $\dot{\psi}$ which depend on the angular body rates p, q, and r. The different rates are related through rotations shown as

$$\begin{Bmatrix} p \\ q \\ r \end{Bmatrix} = R(\phi) \begin{bmatrix} \dot{\phi} \\ 0 \\ 0 \end{bmatrix} + R(\phi)R(\theta) \begin{bmatrix} 0 \\ \dot{\theta} \\ 0 \end{bmatrix} + R(\phi)R(\theta)R(\psi) \begin{bmatrix} 0 \\ 0 \\ \dot{\psi} \end{bmatrix}$$

$$\begin{Bmatrix} p \\ q \\ r \end{Bmatrix} = \begin{bmatrix} 1 & 0 & -s\theta \\ 0 & c\phi & c\theta s\phi \\ 0 & -s\phi & c\theta c\phi \end{bmatrix} \begin{Bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{Bmatrix}$$

$$\begin{Bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{Bmatrix} = \begin{bmatrix} 1 & s\phi tan\theta & c\phi tan\theta \\ 0 & c\phi & -s\phi \\ 0 & s\phi sec\theta & c\phi sec\theta \end{bmatrix} \begin{Bmatrix} p \\ q \\ r \end{Bmatrix} \tag{A–9}$$

Linear and Angular Momentum

Force and moments applied to the vehicle reveal the following

$$\sum F = m\frac{\delta v_b}{dt} + m(w_b \times v_b) \tag{A–10}$$

$$\sum M = I\frac{\delta w_b}{dt} + (w_b \times Iv_b) \tag{A–11}$$

Noting that $\sum F = F_g + F_{prop}$, F_g and F_{prop} must be found. Given the orientation of the vehicle, gravity is defined as being in the z direction, therefore gravitational forces (F_g) may be translated as

$$F_g = mS\begin{bmatrix}0 & 0 & -g\end{bmatrix}^T$$

$$F_g = mg\begin{bmatrix}-s\theta & s\phi c\theta & c\phi c\theta\end{bmatrix}^T \tag{A–12}$$

For F_{prop} the vehicle is assumed to be in a hovering state and therefore results in forces only acting in the z direction of the vehicle. This force is calculated from the thrust forces and results in

$$F_{prop} = \begin{bmatrix}0 & 0 & -u_1\end{bmatrix}^T \tag{A–13}$$

Given a combination of Equations A–12 and A–13 into Equation A–10, along with a reordering of terms, the resulting vector of linear accelerations may be found as

$$\frac{\delta v_b}{dt} = \frac{1}{m}\sum F - (w_b \times v_b)$$

$$\begin{Bmatrix}\dot{u}\\ \dot{v}\\ \dot{w}\end{Bmatrix} = g\begin{bmatrix}-s\theta\\ s\phi c\theta\\ c\phi c\theta\end{bmatrix} + \frac{1}{m}\begin{bmatrix}0\\ 0\\ -u_1\end{bmatrix} - \begin{bmatrix}qw - rv\\ ru - pw\\ pv - qu\end{bmatrix} \tag{A–14}$$

where the variables T_1, T_2, T_3, and T_4 represent the thrust forces generated from the props of the vehicle, and the $\dot{u}$, $\dot{v}$, $\dot{w}$ represent the acceleration components for the body.

Looking at the moment of inertia for a symmetric, rigid body which is aligned with the center of mass, the following Inertia matrix is found

$$I = \begin{bmatrix} I_{11} & 0 & 0 \\ 0 & I_{22} & 0 \\ 0 & 0 & I_{33} \end{bmatrix} \tag{A–15}$$

Revisiting Equation A–11 with the inertia matrix from Equation A–15 shows

$$\begin{aligned} \sum M &= I\frac{\delta w_b}{dt} + (w_b \times Iv_b) \\ \sum M &= \begin{bmatrix} I_{11}\dot{p} \\ I_{22}\dot{q} \\ I_{33}\dot{r} \end{bmatrix} + \begin{bmatrix} p \\ q \\ r \end{bmatrix} \times \begin{bmatrix} I_{11}p \\ I_{22}q \\ I_{33}r \end{bmatrix} \\ \sum M &= \begin{bmatrix} I_{11}\dot{p} \\ I_{22}\dot{q} \\ I_{33}\dot{r} \end{bmatrix} + \begin{bmatrix} qI_{33}r - rI_{22}q \\ rI_{11}p - pI_{33}r \\ pI_{22}q - qI_{11}p \end{bmatrix} \end{aligned} \tag{A–16}$$

Solving Equation A–16 for $\dot{p}$, $\dot{q}$, and $\dot{r}$ results in

$$\begin{bmatrix} \dot{p} \\ \dot{q} \\ \dot{r} \end{bmatrix} = \begin{bmatrix} \frac{M_x}{I_{11}} \\ \frac{M_y}{I_{22}} \\ \frac{M_z}{I_{33}} \end{bmatrix} - \begin{bmatrix} \frac{qr(I_{33}-I_{22})}{I_{11}} \\ \frac{rp(I_{11}-I_{33})}{I_{22}} \\ \frac{pq(I_{22}-I_{11})}{I_{33}} \end{bmatrix} \tag{A–17}$$

To find the moments of the body frame defined in Figure A-1, the sum of all moments about roll, pitch, and yaw must determined. These moments were defined as the inputs to our system and are described in Equations A–3, A–4, and A–5.

$$M_x = u_2 = d(T_4 - T_2) \tag{A–18}$$

$$M_y = u_3 = d(T_3 - T_1) \tag{A–19}$$

$$M_z = u_4 = k_{tm}(T_1 - T_2 + T_3 - T_4) \tag{A–20}$$

Revisiting Equation A–17 with the various moment equations, the values for $\dot{p}$, $\dot{q}$, and $\dot{r}$ may be found.

Calculation of Inertial Equations

The moment of inertia equations [26] may be used to calculate the terms from Equation A–15.

$$
\begin{aligned}
I_{12} &= \frac{1}{12} m_m (l_y^2 + l_z^2) \\
I_{13} &= \frac{1}{12} m_m (l_y^2 + l_z^2) + m_m d^2 \\
I_{11} &= 2I_{12} + 2I_{13}
\end{aligned} \tag{A–21}
$$

$$
\begin{aligned}
I_{21} &= \frac{1}{12} m_m (l_x^2 + l_z^2) \\
I_{23} &= \frac{1}{12} m_m (l_x^2 + l_z^2) + m_m d^2 \\
I_{22} &= 2I_{21} + 2I_{23}
\end{aligned} \tag{A–22}
$$

$$
\begin{aligned}
I_{31} &= \frac{1}{12} m_m (l_x^2 + l_y^2) + m_m d^2 \\
I_{32} &= \frac{1}{12} m_m (l_x^2 + l_y^2) + m_m d^2 \\
I_{33} &= 2I_{31} + 2I_{32}
\end{aligned} \tag{A–23}
$$

State Space Model using Equations of Motion

Given the state vector described in Equation A–1 and the input vector

$$u = \begin{bmatrix} T_1 & T_2 & T_3 & T_4 \end{bmatrix}^T$$

the following linear system may be determined

$$
\begin{aligned}
\dot{\mathbf{x}}(t) &= \mathbf{A}\mathbf{x}(t) + \mathbf{B}\mathbf{u}(t) \\
\mathbf{y}(t) &= \mathbf{C}\mathbf{x}(t) + \mathbf{D}\mathbf{u}(t)
\end{aligned}
$$

Where

$$\dot{\mathbf{x}} = \begin{bmatrix} \dot{x} & \dot{y} & \dot{z} & \dot{\phi} & \dot{\theta} & \dot{\psi} & \dot{u} & \dot{v} & \dot{w} & \dot{p} & \dot{q} & \dot{r} \end{bmatrix}^T$$

$$\dot{\mathbf{x}} = \begin{bmatrix} u & v & w & p & q & r & \dot{u} & \dot{v} & \dot{w} & \dot{p} & \dot{q} & \dot{r} \end{bmatrix}^T \tag{A–24}$$

(A–25)

The equations of motion defined from Equations A–8, A–9, A–14, A–17 provide the following equations for the system (where c and s represent cos and sin)

$$f(x,u) = \begin{cases} \dot{x} & = (c\theta c\psi)u + (s\phi s\theta c\psi - c\phi s\psi)v + (c\phi s\theta c\psi + s\phi s\psi)w \\ \dot{y} & = (c\theta s\psi)u + (s\phi s\theta s\psi + c\phi c\psi)v + (c\phi s\theta s\psi - s\phi c\psi)w \\ \dot{z} & = (-s\theta)u + (s\phi c\theta)v + (c\phi c\theta)w \\ \dot{\phi} & = p + (s\phi tan\theta)q + (c\phi tan\theta)r \\ \dot{\theta} & = (c\phi)q - (s\phi)r \\ \dot{\psi} & = (s\phi sec\theta)q + (c\phi sec\theta)r \\ \dot{u} & = -gs\theta - (qw - rv) \\ \dot{v} & = g(s\phi c\theta) - (ru - pw) \\ \dot{w} & = g(c\phi c\theta) - \frac{1}{m}(u_1) - (pv - qu) \\ \dot{p} & = -\dfrac{qr(I_{33} - I_{22})}{I_{11}} + \dfrac{u_2}{I_{11}} \\ \dot{q} & = -\dfrac{rp(I_{11} - I_{33})}{I_{22}} + \dfrac{u_3}{I_{22}} \\ \dot{r} & = -\dfrac{pq(I_{22} - I_{11})}{I_{33}} + \dfrac{u_4}{I_{33}} \end{cases} \tag{A–26}$$

APPENDIX B
CODE SAMPLES

Python Code Sample from xbee_bridge node.

```
def expand_msg(msg, classes, lists):
    slots = msg.__slots__
    types = msg._slot_types

    for slot, typ in itertools.izip(slots, types):
        value=getattr(msg,slot)
        split = typ.split("/")

        if isinstance(value, list):
            lists.append(slot)

        if len(split) > 1:
            # See if class is a list
            if split[1].find("[]") != -1:
                split[1] = split[1][:-2]
                lists.append(slot)

            module = getattr(import_module(split[0]+"."+"msg"), split[1])
            classes[slot] = module
            tempclasses, templists = expand_msg(module, dict(), [])

            classes = dict(classes, ** tempclasses)
            lists += templists

    return (classes, lists)
```

C++ Code Sample from goal_planner

```
int calculateVerticalLawnmowerPoint(double myCoverage,
                        double otherCoverage,
                        const Eigen::Vector2d & sidePoint1,
```

```
                    const Eigen::Vector2d & sidePoint2,
                    const Point2DVector & coordinates,
                    Eigen::Vector4d & boundary,
                    Eigen::Vector2i & currentIndex,
                    Eigen::Vector2i otherIndex,
                    Eigen::Vector2d & point,
                    Point2DVector & pointList,
                    int direction, ) {
Eigen::Vector2d testPoint;
Eigen::Vector2d nextPoint;
Eigen::Vector2d directionVector(0,0);
Eigen::Vector2d currentDirectionVector(0,0);
double directionDistance = 0;

//*********************************************************
// Attempt to calculate next point for lawnmower path, loop if
// needed
//*********************************************************
while(true) {
  // If indices overlap, end plan
  if ((sidePoint1 == coordinates[currentIndex[0]]) &&
     (sidePoint2 == coordinates[currentIndex[1]])) {
                      return false;
  }

  calculateDirectionVectorAndDistance(sidePoint1,
    sidePoint2, coordinates[currentIndex[START]],
    coordinates[currentIndex[END]], directionVector,
    myCoverage + otherCoverage, directionDistance);

  //*****************************************************************
  // Calculate next point to be tested
  // x(t) = m*t + x0
```

```
// y(t) = n*t + y0
//*****************************************************************
testPoint.x() = directionVector[START]*directionDistance +
                sidePoint1.x();
testPoint.y() = directionVector[END]*directionDistance +
                sidePoint1.y();

// Offset from border
if (!edgeSearch) {
  currentDirectionVector = nextPoint - pointList[pointList.size()-1];
  currentDirectionVector = currentDirectionVector /
                                          currentDirectionVector.norm()
                                              ;

  nextPoint.x() = currentDirectionVector[START] * -borderOffset +
                  nextPoint.x();
  nextPoint.y() = currentDirectionVector[START] * -borderOffset +
                  nextPoint.y();
} else {
  nextPoint = testPoint;
}

ROS_INFO("Point to check (%f, %f)", nextPoint.x(), nextPoint.y());

//*****************************************************************
// Check point to see if it is valid:
// If the calculated point is within the final point on the line, push
// to list
// else if the small chance that the final point == the last point on
// line, just end
//*****************************************************************
if( ((almostEqualUlps(boundary[LEFT],boundary[RIGHT],maxUlps) &&
      almostEqualUlps(boundary[LEFT],nextPoint.x(),maxUlps)) ||
```

```
      (nextPoint.x() >= boundary[LEFT]    &&
      nextPoint.x() <= boundary[RIGHT])) &&
     ((almostEqualUlps(boundary[BOTTOM],boundary[TOP],maxUlps) &&
       almostEqualUlps(boundary[TOP],nextPoint.y(),maxUlps)) ||
      (nextPoint.y() >= boundary[BOTTOM] &&
       nextPoint.y() <= boundary[TOP])) )
{
  //*****************************************************************
  // Push point to list and return
  //*****************************************************************
  ROS_INFO("Point is within line segment");

  point = nextPoint;
  pointList.push_back(nextPoint);

  return true;
} else if((nextPoint.x() < boundary[LEFT] ||
           nextPoint.x() > boundary[RIGHT]) ||
          (nextPoint.y() < boundary[BOTTOM] ||
           nextPoint.y() > boundary[TOP] ) ) {
  // TODO Review and see if it is best to only add the final point if
  // coverage has not happened
  // hint: this should be done when taking angle of entry into account.
  ROS_INFO("Point is outside line segment");
  ROS_INFO("Start Point (%f, %f) End Point (%f, %f)",
           sidePoint1.x(), sidePoint1.y(),
           coordinates[currentIndex[END]].x(),
           coordinates[currentIndex[END]].y());
  ROS_INFO("Distance between is %f", (sidePoint1 -
           coordinates[currentIndex[END]]).norm());
  ROS_INFO("My Range is %f", myCoverage);
  ROS_INFO("Previous Range is %f", otherCoverage);
```

```
//*************************************************************
// Since point is not on segment, check distance to next segment.
// If less than horizontal Coverage - agent coverage, end point
//     should be added for this agent to cover final hole
// else begin checking next line segment
//*************************************************************
if (((sidePoint1 - coordinates[currentIndex[END]]).norm()) >
  (otherCoverage)) {
  ROS_INFO("Agent should still handle searching this area");

  //*********************************************************
  // Push end point to list
  //*********************************************************
  point = coordinates[currentIndex[END]];
  pointList.push_back(coordinates[currentIndex[END]]);

  //*********************************************************
  // Shift to next line segment
  //*********************************************************
  // Shift to next indices
  ROS_INFO("Find New Indices");
  ROS_INFO("Before Top (%d, %d) Bottom (%d, %d)", currentIndex[START],
            currentIndex[END], otherIndex[START], otherIndex[END]);
  calculateNextIndices(currentIndex, direction, coordinates.size() -1);
  ROS_INFO("Before Top (%d, %d) Bottom (%d, %d)", currentIndex[START],
            currentIndex[END], otherIndex[START], otherIndex[END]);

  //*********************************************************
  // Check if new points segments are valid for planning
  // If line segments overlap, end planning
  // Else if Corner End point endices match and either corner point
  //     of new line segment and distance to the parallel point
  //     already on the list is less than my desired coverage,
```

```
    //      end planning (Distance long entire segment is worth
    //      searching)
    // Else, calculate a new direction vector and check if the corner
    //      point is already on the list.
    //*************************************************************
    if ((currentIndex[0] == otherIndex[1]) &&
        (currentIndex[1] == otherIndex[0])) {
      ROS_INFO("Line Segments Overlap");
      return false;
    } else if ((currentIndex[1] == otherIndex[1]) &&
        ((sidePoint1 - coordinates[currentIndex[0]]).norm() <=
         (myCoverage + otherCoverage)) &&
        ((sidePoint2 - coordinates[currentIndex[1]]).norm() <=
         (myCoverage + otherCoverage))) {
      ROS_INFO("New Point Matches previous corner point");
      return false;
    }

    // Redefine Bounding Box for new line segment
    getBoundaryPoints(coordinates[currentIndex[START]],
                      coordinates[currentIndex[END]], boundary);

    return true;
  } else {
    ROS_INFO("No Point, shift to next segment");
    //*************************************************************
    // Shift to next line segment
    //*************************************************************
    // Shift to next indices
    ROS_INFO("Find New Indices");
    ROS_INFO("Before Top (%d, %d) Bottom (%d, %d)", currentIndex[START],
             currentIndex[END], otherIndex[START], otherIndex[END]);
    calculateNextIndices(currentIndex, direction, coordinates.size() -1);
```

```
ROS_INFO("Before Top (%d, %d) Bottom (%d, %d)", currentIndex[START],
          currentIndex[END], otherIndex[START], otherIndex[END]);

//*************************************************************
// Check if new points segments are valid for planning
// If line segments overlap, end planning
// Else if Corner End point endices match and either corner point
//     of new line segment and distance to the parallel point
//     already on the list is less than my desired coverage,
//     end planning (Distance long entire segment is worth
//     searching)
// Else, calculate a new direction vector and check if the corner
//     point is already on the list.
//*************************************************************
if ((currentIndex[0] == otherIndex[1]) &&
    (currentIndex[1] == otherIndex[0])) {
  ROS_INFO("Line Segments Overlap");
  return false;
} else if ((currentIndex[1] == otherIndex[1]) &&
    ((sidePoint1 - coordinates[currentIndex[0]]).norm() <=
     (myCoverage + otherCoverage)) &&
    ((sidePoint2 - coordinates[currentIndex[1]]).norm() <=
     (myCoverage + otherCoverage))) {
  ROS_INFO("New Point Matches previous corner point");
  return false;
}

// Redefine Bounding Box for new line segment
getBoundaryPoints(coordinates[currentIndex[START]],
                  coordinates[currentIndex[END]], boundary);

// If the corner point is already on the list continue current
// search and find next point
```

```
        // else push the corner point to list
        if (coordinates[currentIndex[START]] == sidePoint1) {
          ROS_INFO("Continue and find point along next segment");
          continue;
        } else {
          ROS_INFO("Put First coordinate of next line on list");
          point = coordinates[currentIndex[START]];
          pointList.push_back(coordinates[currentIndex[START]]);
          return true;
        }
      }
    }
  }
}
```

REFERENCES

[1] Achtelik, Markus, Achtelik, Michael, Brunet, Yorick, Chli, Margarita, Chatzichristofis, Savvas, Decotignie, Jead-Dominique, Doth, Klaus-Michael, Fraundorfer, Friedrich, Kneip, Laurent, Gurdan, Daniel, Heng, Lionel, Kosmatopoulos, Elias, Doitsidis, Lefteris, Lee, Gim Hee, Lynen, Simon, Martinelli, Agostino, Meier, Lorenz, Pollefeys, Marc, Piguet, Damien, Renzaglia, Alessandro, Scaramuzza, Davide, Siegwart, Roland, Stumpf, Jan, Tanskanen, Petri, Troiani, Chiara, and Weiss, Stephan. "SFly: Swarm of micro flying robots." *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*. 2012, 2649–2650.

[2] Chaimowicz, Luiz, Grocholsky, Ben, Keller, James F., Kumar, R. Vijay, and Taylor, Camillo J. "Experiments in multirobot air-ground coordination." *Robotics and Automation, 2004. Proceedings. ICRA '04. 2004 IEEE International Conference on*. vol. 4. 2004, 4053–4058.

[3] Chaimowicz, Luiz and Kumar, Vijay. "Aerial shepherds: Coordination among uavs and swarms of robots." *In Proc. of DARS04*. 2004, 243–252.

[4] Dash, Gautam, Weaver, Joshua, Arroyo, Dr. A. Antonio, Wright, Sean, and Schwartz, Dr. Eric M. "Orchestration of Intelligent Ground Vehicles into a Homogeneous Swarm via the Google Cloud." *Florida Conference on Recent Advances in Robotics*. Boca Raton, FL, 2012.

[5] Doitsidis, Lefteris, Weiss, Stephan, Renzaglia, Alessandro, Achtelik, MarkusW., Kosmatopoulos, Elias, Siegwart, Roland, and Scaramuzza, Davide. "Optimal surveillance coverage for teams of micro aerial vehicles in GPS-denied environments using onboard vision." *Autonomous Robots* 33 (2012): 173–188.

[6] Dryanovski, Ivan, Morris, William, and Xiao, Jizhong. "Multi-Volume Occupancy Grids: an Efficient Probabilistic 3D Mapping Model for Micro Aerial Vehicles." *IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS2010)*. 2010, 1553 – 1559.

[7] Duan, HaiBin and Liu, SenQi. "Unmanned air/ground vehicles heterogeneous cooperative techniques: Current status and prospects." *Science China Technological Sciences* 53 (2010): 1349–1355.

[8] Ducatelle, Frederick, Caro, GianniA., and Gambardella, LucaM. "Cooperative Stigmergic Navigation in a Heterogeneous Robotic Swarm." *From Animals to Animats 11*. eds. Stphane Doncieux, Benot Girard, Agns Guillot, John Hallam, Jean-Arcady Meyer, and Jean-Baptiste Mouret, vol. 6226 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2010. 607–617.

[9] Ducatelle, Frederick, Di Caro, Gianni A., and Gambardella, Luca M. "Cooperative self-organization in a heterogeneous swarm robotic system." *Proceedings of the 12th annual conference on Genetic and evolutionary computation*. GECCO '10. New York, NY, USA: ACM, 2010, 87–94.

[10] Garzn, Mario, Valente, Joo, Zapata, David, and Barrientos, Antonio. “An Aerial-Ground Robotic System for Navigation and Obstacle Mapping in Large Outdoor Areas.” *Sensors* 13 (2013).1: 1247–1267.

[11] Grocholsky, Ben, Keller, James F., Kumar, Vijay, and Pappas, George. “Cooperative air and ground surveillance.” *Robotics Automation Magazine, IEEE* 13 (2006).3: 16 –25.

[12] Hoffmann, Gabe, Rajnarayan, Dev Gorur, Waslander, Steven L., Dostal, David, Jang, Jung Soon, and Tomlin, Claire J. “The Stanford Testbed of Autonomous Rotorcraft for Multi Agent Control (STARMAC).” *in Proceedings of the 23rd Digital Avionics Systems Conference*. Salt Lake City, Utah, 2004, 12.E.4–1 – 10.

[13] Hoffmann, Gabriel M., Waslander, Steven L., , and Tomlin, Claire J. “Quadrotor Helicopter Trajectory Tracking Control.” *In the Proceedings of the AIAA Guidance, Navigation, and Control Conference*. Honolulu, HI, 2008, 251–255.

[14] How, Jonathan P., Bethke, Brett, Frank, Adrian, Dale, Daniel, and Vian, John. “Real-time indoor autonomous vehicle test environment.” *Control Systems, IEEE* 28 (2008).2: 51–64.

[15] Hsieh, M. Ani, Cowley, Anthony, Keller, James F., Chaimowicz, Luiz, Grocholsky, Ben, Kumar, Vijay, Taylor, Camillo J., Endo, Yoichiro, Arkin, Ronald C., Jung, Boyoon, Wolf, Denis F., Sukhatme, Gaurav S., and MacKenzie, Douglas C. “Adaptive teams of autonomous aerial and ground robots for situational awareness: Field Reports.” *Journal of Field Robotics* 24 (2007).11-12: 991–1014.

[16] Kumar, Vijay and Michael, Nathan. “Opportunities and challenges with autonomous micro aerial vehicles.” *Int. J. Rob. Res.* 31 (2012).11: 1279–1291.

[17] Lee, Geunho, Sato, Y., and Chong, Nak Young. “Decentralized task-oriented local group generation for robot swarms.” *Ubiquitous Robots and Ambient Intelligence (URAI), 2011 8th International Conference on*. 2011, 174 –179.

[18] Li, Wei, Zhang, Tianguang, and Kuhnlenz, K. “A vision-guided autonomous quadrotor in an air-ground multi-robot system.” *Robotics and Automation (ICRA), 2011 IEEE International Conference on*. 2011, 2980–2985.

[19] Mathews, Nithin, Christensen, AndersLyhne, OGrady, Rehan, and Dorigo, Marco. “Cooperation in a Heterogeneous Robot Swarm through Spatially Targeted Communication.” *Swarm Intelligence*. eds. Marco Dorigo, Mauro Birattari, GianniA. Caro, Ren Doursat, AndriesP. Engelbrecht, Dario Floreano, LucaMaria Gambardella, Roderich Gro, Erol ?ahin, Hiroki Sayama, and Thomas Sttzle, vol. 6234 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2010. 400–407.

[20] Mathews, Nithin, Stranieri, Alessandro, Scheidler, Alexander, and Dorigo, Marco. “Supervised morphogenesis: morphology control of ground-based self-assembling

robots by aerial robots." *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems - Volume 1*. AAMAS '12. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems, 2012, 97–104.

[21] Mellinger, Daniel, Shomin, Michael, Michael, Nathan, and Kumar, Vijay. "Cooperative Grasping and Transport Using Multiple Quadrotors." *Distributed Autonomous Robotic Systems*. eds. Alcherio Martinoli, Francesco Mondada, Nikolaus Correll, Grgory Mermoud, Magnus Egerstedt, M. Ani Hsieh, Lynne E. Parker, and Kasper Sty, vol. 83 of *Springer Tracts in Advanced Robotics*. Springer Berlin Heidelberg, 2013. 545–558.

[22] Michael, Nathan, Belta, Calin, and KumarKumar, Vijay. "Controlling three dimensional swarms of robots." *Robotics and Automation, 2006. ICRA 2006. Proceedings 2006 IEEE International Conference on*. 2006, 964 –969.

[23] Michael, Nathan, Fink, Jonathan, and Kumar, Vijay. "Controlling a team of ground robots via an aerial robot." *Intelligent Robots and Systems, 2007. IROS 2007. IEEE/RSJ International Conference on*. 2007, 965 –970.

[24] Morris, William, Dryanovski, Ivan, and Xiao, Jizhong. "3D Indoor Mapping for Micro-UAVs Using Hybrid Range Finders and Multi-Volume Occupancy Grids." *the workshop on RGB-D: Advanced Reasoning with Depth Cameras, on June 27, 2010, in conjunction with 2010 Robotics: Science and Systems Conference (RSS2010)*. Universidad de Zaragoza, Spain, 2010.

[25] Ritz, Robin, Muller, Mark W., Hehn, Markus, and D'Andrea, Raffaello. "Cooperative quadrocopter ball throwing and catching." *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*. 2012, 4972–4978.

[26] Schmidt, Louis V. *Introduction to aircraft flight dynamics*. Amer Inst of Aeronautics, 1998.

[27] Silva, Oscar De, Mann, George K. I., and Gosine, Raymond G. "Development of a relative localization scheme for ground-aerial multi-robot systems." *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*. 2012, 870–875.

[28] Tanner, Herbert G. and Christodoulakis, Dimitrios K. "Decentralized cooperative control of heterogeneous vehicle groups." *Robotics and Autonomous Systems* 55 (2007).11: 811 – 823.

[29] Turpin, Matthew, Michael, Nathan, and Kumar, Vijay. "Trajectory design and control for aggressive formation flight with quadrotors." *Autonomous Robots* 33 (2012): 143–156.

[30] Vandapel, Nicolas, Donamukkala, Raghavendra Rao, and Hebert, Martial. "Unmanned Ground Vehicle Navigation Using Aerial Ladar Data." *The International Journal of Robotics Research* 25 (2006).1: 31–51.

[31] Waslander, Steven L., Hoffmann, Gabriel M., Jang, Jung Soon, and Tomlin, Claire J. "Multi-Agent Quadrotor Testbed Control Design: Integral Sliding Mode vs. Reinforcement Learning." *IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2005, 3712–3717.

[32] Weaver, Joshua, Dash, Gautam, Arroyo, Dr. A. Antonio, and Schwartz, Dr. Eric M. "Heterogeneous Collaborative Swarms of Autonomous Agents with Varying Capabilities." *Florida Conference on Recent Advances in Robotics*. Boca Raton, FL, 2012.

[33] Weaver, Joshua, Dash, Gautam, Thompson, Matthew, Arroyo, Dr. A. Antonio, and Schwartz, Dr. Eric M. "Control of UAV Through the Robot Operating System and Android." *Florida Conference on Recent Advances in Robotics*. Tallahassee, FL, 2013.

[34] Weaver, Joshua, Frank, Daniel, Arroyo, Dr. A. Antonio, and Schwartz, Dr. Eric M. "UAV Performing Autonomous Landing on USV Utilizing the Robot Operating System." *ASME Early Career Technical Conference 2013*. vol. 12. Birmingham, AL, 2013, 119–124.

[35] Weiss, Stephan, Scaramuzza, Davide, and Siegwart, Roland. "Monocular-SLAMbased navigation for autonomous micro helicopters in GPS-denied environments." *Journal of Field Robotics* 28 (2011).6: 854–874.

[36] Wright, Sean, Arroyo, Dr. A. Antonio, Arroyo, Ian A., Simpkins, Joseph W., Wright, Brock, and Zamstein, Dr. Lavi M. "Emergent Collective Behavior in Autonomous Synergistic Swarm Robots." *Florida Conference on Recent Advances in Robotics*. Gainesville, FL, 2011.

[37] Zengin, Ugur and Dogan, Atilla. "Cooperative target pursuit by multiple UAVs in an adversarial environment." *Robotics and Autonomous Systems* 59 (2011).12: 1049 – 1059.

BIOGRAPHICAL SKETCH

Joshua Nathaniel Weaver was born in Tacoma, Washington, November 29, 1983. As he grew up he had always been interested in science and technology, taking everything in his way apart to learn how it worked, and later attempting to put it back together. Through middle school and high school he always found himself very interested in the concepts that were taught and the advancements that were made every day, while dreaming that he would be a leader in his field some day. After graduating in the Top Ten of his class at Mosley High School he attended Gulf Coast Community College where he continue his education while also research and learning on his own. While at Gulf Coast Community College he was a part of the honors program and the president of Phi Theta Kappa.

Mr. Weaver later attended Florida State University where he continued his focus on electrical and computer engineering while always looking further into robotics. After graduating at the top of his class with his Bachelors of Science in electrical engineering in 2007. Next he took time working for a private company before moving to work for the US Navy where he was able to continue his interest in robotics. After being accepted into the SMART Fellowship program, he attended the University of Florida where he worked in the Machine Intelligence Laboratory (MIL) under the advisement and mentorship of Dr A. Antonio Arroyo and Dr. Eric Schwartz. With this new focus, Mr. Weaver was able to further his understanding in software development for autonomous robotics for ground and aerial vehicles. He first completed his Masters of Science in late 2011 while later completing his Doctor of Philosophy in electrical and computer engineering in 2014. His PhD research focused on the advancement and development of a swarm architecture for multiple unmanned ground and aerial vehicles.