MODULAR HETEROGENOUS MULTI-AGENT CONTROL FRAMEWORK WITH INTEGRATED PAYLOADS

A Thesis

by

CAMERON TYLER ROGERS

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Chair of Committee,	John Valasek
Committee Members,	Gregory H. Huff
	Moble Benedict
	P.R. Kumar
Head of Department,	Rodney D. W. Bowersox

August 2017

Major Subject: Aerospace Engineering

ABSTRACT

Small unmanned aircraft are being used in an increasing number of applications ranging from emergency response to parcel delivery. Many of these applications are benefited when employed as a multiple-vehicle operation. Such operations often require tight cooperation between heterogeneous vehicles and often depend on integration with sensors and payloads. Multi-agent control algorithms can be implemented to control such systems but often require the development of an underlying vehicle communications framework in addition to a sensors and payloads communications framework. This thesis presents a single unified modular framework, named Clark, and supports heterogeneous multi-agent control and sensor/payload integration. Clark provides a wireless network between agents without relying on pre-existing communications infrastructure, and provides software interfaces for connecting to a variety of payloads. This thesis first reviews small unmanned aircraft systems (SUAS), multi-agent control, multi-agent control testbeds, and wireless networking technologies used on SUAS. Systems engineering is then employed to develop an Identified Need, Concept of Operations (ConOps), and requirements. All Defined, Derived, and Design Requirements are explained and justified. Some requirements are highlighted to demonstrate key features of the Clark framework. The software architecture is explained in detail in a top-down approach. Hardware is selected for prototyping and shown to meet the requirements. Bench tests, ground tests, and flight tests are conducted to verify the framework's ability to communicate between agents and affect control. Ground testing includes a multi-agent cooperative mission while flight testing features two and three agent missions. Test results are presented and demonstrate the candidacy of Clark as a modular heterogeneous multi-agent control framework with integrated payloads.

DEDICATION

To my loving and incredible wife Elisabeth who has helped and supported me the entire way.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a thesis committee consisting of Dr. John Valasek and Dr. Moble Benedict of the Department of Aerospace Engineering and Dr. Gregory Huff and Dr. P.R. Kumar of the Department of Electrical & Computer Engineering. The author also recognizes the contribution of student collaborators Han-Hsun Lu, Charles Noren, Zeke Bowden, James Henrickson, and Vinicius Goecks who gave feedback on the system design and supported flight testing.

Funding Sources

Graduate study was supported by fellowship from Texas A&M University.

NOMENCLATURE

SUAS	Small Unmanned Aircraft System
UAS	Unmanned Aircraft System
FAA	Federal Aviation Administration
SA	Situational Awareness
GCS	Ground Control Station
MEMS	Microelectromechanical Systems
INS	Inertial Navigation System
ROS	Robotic Operating System
MANET	Mobile Ad-hoc Network
ConOps	Concept of Operations
VSCL	Vehicle Systems & Control Laboratory
MAVLink	Micro Aerial Vehicle Link
GUI	Graphical User Interface
MCS	Mission Control Station
API	Application Programming Interfaces
TCP/IP	Transmission Control Protocol/Internet Protocol
UDP/IP	User Datagram Protocol/Internet Protocol

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION AND LITERATURE REVIEW

This thesis investigates a design for a system capable of testing multi-agent control algorithms. The system should support the algorithms such that heterogeneous vehicle can be controlled autonomously. The main class of vehicles of interest are small unmanned aircraft systems (SUAS), but this thesis will also include ground-based vehicles. This thesis does not seek to develop a novel multi-agent control law but rather to develop a framework upon which multi-agent control laws can be implemented. The investigation includes integrating designs of a multi-agent communications network, an in situ payload network, and a control law implementation layer into a single system. This design is driven by Needs and requirements developed through Systems Engineering. The system is designed to be operated in a testing environment and does not attempt to address several issues pertinent to full-scale commercial use and is therefore not intended for commercial use. In all, this thesis aims to utilize Systems Engineering to develop and test a candidate multi-agent control framework named Clark.

In order to understand the importance of a multi-agent SUAS framework, it is necessary to understand SUAS, how they have been used, and their current uses today. Unmanned aircraft systems (UAS) are defined by the Federal Aviation Administration (FAA) as “an aircraft without a human pilot onboard — instead, the UAS is controlled from an operator on the ground” [1]. The FAA further specifies that an SUAS differs from a UAS by weighing less than 55 pounds. The operator can control the aircraft through a radio transmitter as with remote control operation or with a ground control station (GCS). It is important to note that by definition, SUAS are multi-agent systems, because the ground operator and vehicle are considered agents, but they are not explicitly multi-vehicle systems. SUAS have historically been used for a variety of mission types including missions designated as dull, dirty, or dangerous by both military and civil operators [2]. Recently, SUAS have been seen in a different light: as a tool to be used by businesses and other civil organizations. These organizations are attracted to the flexibility and low operational costs of SUAS [3]. Such civil applications may include parcel delivery, infrastructure assessment [4], traffic monitoring [5], catastrophe response [6], search and rescue [7], precision agriculture [8], and natural resource management [9]. Many of these applications can utilize multiple SUAS. The possibility that a large number of SUAS would operate in close proximity leads to concerns of congested airspace and lack of adequate and predictable control in general [10]. This concern is overshadowed by the potential benefits of using multiple SUAS in teams, but must still be resolved. A resolution is available by enabling communication between SUAS and controlling them using shared information. Therefore, a key element for enabling large-scale use of SUAS is to have an autonomous multi-vehicle system capable of keeping teams of SUAS connected and controlled.

1.1 Literature Review: Multi-Agent UAS Controllers

There is a great deal of complexity in controlling multiple SUAS due to the fact that there are multiple levels of control. The low-level control loop maintains stability of the vehicle and can maintain the altitude, velocity, and heading of the vehicle [11]. These low-level controllers are dependent only on reference values, the vehicle states such as attitude and velocity, and the vehicle dynamics. The next level is generally the navigation loop. Navigation loops are present in many single-vehicle systems that allow for GPS guided navigation such as waypoint following. For multi-agent control, the navigation loop often receives inputs from multi-agent algorithms. The multi-agent control algorithm takes vehicle states and mission objectives as an input and sets the reference values for the lower-level loops. Together the lower-level loops and high level multi-agent control algorithms form a multi-agent controller. An important input to a multi-agent control algorithm is situational awareness (SA). SA is the information synthesized from inputs from multiple agents. An example of SA could be the size of a contaminated area in an application where SUAS are used to map a chemical spill. Thus it can be seen that multi-agent controllers are highly dependent on mission objectives, SA, and vehicle states. Therefore, multi-agent controllers vary greatly and are complex, requiring extensive underlying communications framework for support.

While many controllers have been built for a variety of applications [12] [13] [14] [15], one of particular interest was put forward by Beard and McLain [16]. They proposed a control technique that could be used to cooperatively plan trajectories of SUAS surveying a selected area assuming a limited sensing range. The controller takes into account realistic assumptions that communication range constrains are present and that a commanded heading, velocity, and altitude can be maintained by the low-level loop of the autopilot. The proposed controller first generates a trajectory given the current state and SA. It then smooths the trajectory and compares it against other trajectories by a cooperative waypoint path planner to avoid collisions. Figure 1.1 shows this process. Through simulation they showed that collision avoidance can be achieved while also accomplishing the mission objective. This example controller demonstrates the importance of high-level computation that must be made to develop the trajectories and make comparisons to avoid collisions. This computation goes beyond that of the low-level micro-controller typically found in autopilots. Additionally, it highlights the need for inter-agent communication.

Another example algorithm was put forward by Kurdi et al. [17] in which a Locust-Inspired Algorithm for task allocation in Multiple UAS (LIAM) search and rescue missions is proposed. This algorithm is distributed amongst search UASs and rescue UASs where these two types of UASs are assumed to have different resources available. These UASs collaborate and report back to a GCS which then adapts the UASs behavior to best complete the search and rescue objectives. After simulating LIAM with the MASPLANES

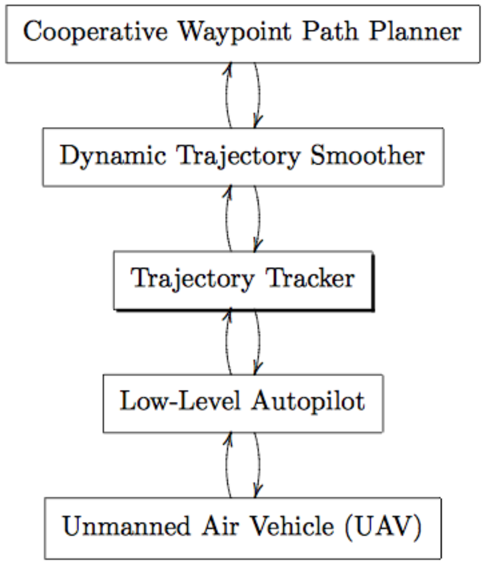

Cooperative Waypoint Path Planner
Dynamic Trajectory Smoother
Trajectory Tracker
Low-Level Autopilot
Unmanned Air Vehicle (UAV)

Figure 1.1: Path planning architecture proposed by Beard and McLain; reprinted from [16]

[18] environment, it is shown to have superior performance to comparable algorithms. This algorithm highlights the need for a multi-agent system to support heterogeneous vehicles and function outdoors where an algorithm such as LIAM would be realistically implemented.

When operating multiple SUAS it is generally more simple to implement control from a centralized planner. However, such systems are susceptible to a single point of failure. In response to this, several schemes have been proposed from initiating the central planner on several vehicles [19] [20] to a complete distribution of the planning [21]. One such distributed approach was suggested by Choi et al. [22]. In this work, Choi presented two algorithms which utilize consensus between the agents that guarantees a conflict-free solutions for the SUAS. In other words, the SA sensed by each agent is then reconciled by one of the two algorithms to create a single SA independent of inconsistencies in the sensed data. These algorithms demonstrate a clear advantage of implementing multi-agent control on a completely distributed framework.

1.2 Literature Review: Multi-Agent Control Testbeds

Currently, there are several multi-agent control testbeds in use, including: the Massachusets Intstitute of Technology's (MIT) Real-time Autonomous Vehicle indoor test ENvironment (RAVEN) [23], the Stanford Testbed of Autonomous Rotorcraft for Multi-Agent Control (STARMAC) [24], the Flying Machine Arena (FMA) [25], the General Robotics, Automation, Sensing and Perception (GRASP) Laboratory Multiple Micro UAV Testbed [26], and Etherware [27]. These testbeds have allowed many control algorithms to be tested and all have advantages and limitations.

The MIT RAVEN is an indoor testbed that utilizes ground-based computers and a Vicon MX camera system for calculating control and measuring the attitude and position of the vehicles. The testbed can be used with quadrotors, helicopters, and even fixed-wing aircraft in a nose-up hover. The RAVEN testbed has been used in implementing a variety of algorithms including combining vehicle health monitoring with task allocation [28]. The main advantages of the MIT RAVEN system is the high 120 Hz rate at which it can run, accurate attitude information from the Vicon camera system, and ground-based computation. The Vicon camera gives position and attitude estimates that can be much more accurate than a GPS- and microelectromechanical system (MEMS)-based inertial navigation system (INS) system. However, the MIT RAVEN is an indoor testbed that is dependent on resources external to the flight vehicles.

The STARMAC testbed features the X4 flyer as the only supported vehicle and is designed for testing collision avoidance and task allocation. It is built with Bluetooth communication hardware but can also be run using the open source Robotic Operating System (ROS) [29]. It functions outdoors with a range of up to 100 meters but depends on a central computation cluster of one or multiple machines for computation. STARMAC can be extended beyond multi-agent control, as Huang et al. demonstrated on autonomous quadrotors that perform highly aggressive maneuvering [30]. In all, the STARMAC system is capable because it functions outdoors, however it still has a limited communications range and only supports the X4 flyer.

The Flying Machine Arena, Multiple Micro UAV Testbeds, and Etherware are all indoor testbeds. The FMA and Multiple Micro UAV Testbeds make use of the Vicon camera system for positioning and all three testbeds utilize 2.4 GHz radio communication. In general, all systems have a similar architecture as shown by Figure 1.2 [31] and depend on a central computing resource. An important contribution made by Etherware was the addition of the Control Time Protocol (CTP). This protocol allows for prioritization of control information over other information on the network. While useful, all three testbeds are designed for indoor use and have centralized computational nodes. Etherware, however demonstrates the reliability of using a protocol designed explicitly for control of multi-agent systems.

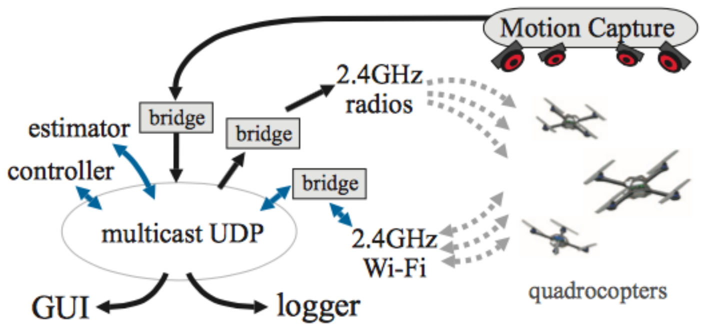


Figure 1.2: Overview of the FMA architecture; reprinted from [25]

1.3 Literature Review: Vehicular Ad-Hoc Networks

An important aspect of controlling multiple vehicles is the network that is chosen for the communication. There are several topologies and protocols for connecting several SUAS. These topologies can be centralized or decentralized. A centralized topology typically has a single router through which all communication is handled. In the case of multi-agent control, this is limiting because distribution of the control is not possible and consequentially must be centralized. It also presents a risk of system failure if the router loses connection or has an error. An alternative is a mobile ad-hoc network (MANET). Much research has been conducted in using and implementing MANETs on SUAS. MANETs allow for agents to join and leave the network as needed without reconfiguration and exhibit self healing if a part of the systems stops functioning. Christmann [32] investigated leveraging a vehicle's characteristics, such as positions, to facilitate inter-vehicle communication between SUAS on an ad-hoc network. The resulting MANET distributed the location services, data routing, and node databases amongst agents resulting in an efficient communications network. Vehicular ad-hoc networks provide clear advantages in allowing information to be completely distributed, and provide a method for distributing information in a multi-vehicle SUAS system.

1.4 Approach Summary

This thesis will utilize Systems Engineering in conjunction with Spiral Development to create a new multi-agent control framework to best fit the needs of Texas A&M University's Vehicle Systems & Control Laboratory (VSCL) for testing multi-agent control algorithms. Hardware will then be selected to test this system in an outdoor environment. The system will include a software package and procedures for using the system. The software package will include the software for real time operation and software for analysis and troubleshooting. Hardware will be selected based upon the requirements and integrated with the software to form a prototype testbed. The scope of the software will be designated through requirements, which will be continually revisited during software development. The system will then be evaluated by analyzing the ability to send and receive state information of agents within the system and affect control. The developed software will also be analyzed to certify that all requirements have been met. Through these steps, the system put forward in this thesis will be verified as a heterogeneous multi-agent modular control framework.

This document is structured in the following manner: Chapter 1 introduces multi-agent control frameworks and provides insights into multi-agent controllers. Chapter 2 sets forward the Systems Engineering process, requirements, and implications of the requirement on the design. Chapter 3 explores the development of the software and design changes that came about during software development. Selected hardware for prototyping and testing are given in Chapter 4. Chapter 5 contains the testing setup and results. This work then finishes with Chapter 6-7 where conclusions are drawn and recommendations for future work are given.

2. SYSTEM DESIGN

The Clark system was designed using Systems Engineering [33] and Spiral Development [34] to ensure functionality and simplicity of the system. Systems Engineering is commonly used for the development of systems that contain many subsystems that must interact to accomplish a larger objective than any of the subsystems could achieve individually. It also facilitates efficiency of development by delineating the functions and features that must be included versus those which are desired but not necessary. The Systems Engineering process begins with an identified Need. This Need gives a high-level explanation of *what* is to be done without specifying *how* it is to be accomplished. This distinction allows the design process to refine the method through which the Need will be met because it does not prematurely lock in the design. The second step is to develop a ConOps from this Need. The ConOps builds a theoretical operation around a system that acts as a solution to the need. The ConOps permits operability and usability to enter the design. It is important to note that the ConOps does not provide specifics on *how* the final design will look but continues to expand on *what* the system will do. Defined Requirements are then extracted from the ConOps. Derived and Design Requirement are extracted to detail the system specifications and then reviewed successively throughout the design process. Figure 2.1 shows the "V" diagram which outlines the Systems Engineering process [35].

Spiral Development is a process that provides a designer more flexibility to change the requirements when writing software. During the course of software development, code is often rewritten when unexpected limitations or advantages are found. Spiral Development allows the requirements to be modified as these limitations or advantages are discovered while maintaining a system outlook so not to negatively affect another aspect of the design. Moreover, it promotes development, integration, and testing of core components before all

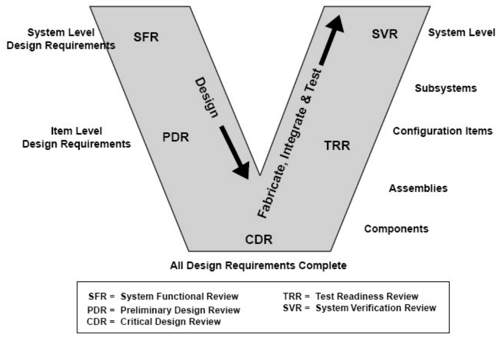

System Level Design Requirements
SFR
Design
Item Level Design Requirements
PDR
CDR
All Design Requirements Complete
Fabricate, Integrate & Test
SVR
System Level
Subsystems
Configuration Items
TRR
Assemblies
Components
SFR = System Functional Review
PDR = Preliminary Design Review
CDR = Critical Design Review
TRR = Test Readiness Review
SVR = System Verification Review

Figure 2.1: Systems Engineering "V" diagram; reprinted from [35]

requirements are solidified to allow for changes to key requirements. Core software components are determined by assessing the risk posed by that component in system. Figure 2.2 [34] shows the Spiral Development model as presented by Boehm. Essentially, spiral development allows a system designer to work horizontally between the left and right side of the Systems Engineering "V" (see Figure 2.1) in addition to working along it. The

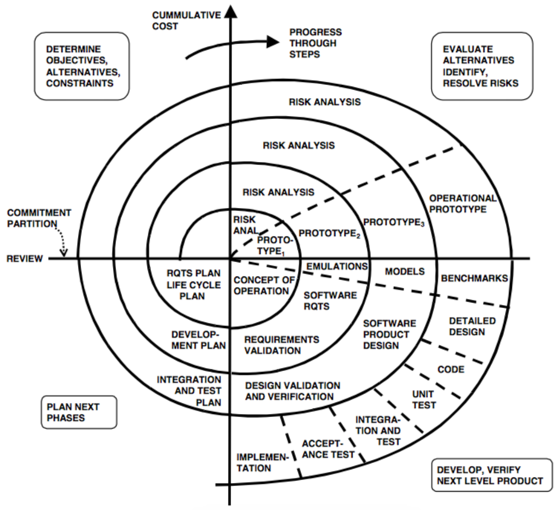


Figure 2.2: Spiral Development diagram; reprinted from [34]

Needs statement, ConOps, and requirements contained in this document are a result of eight iterations through the Systems Engineering "V" diagram. Although many requirements from previous iterations shaped the final requirements, they will not be included the the following discussion.

2.1 Needs Statement

The following needs statement was developed with the intent to provide a solution to allowing multi-agent controllers to be quickly tested by the VSCL.

> The VSCL desires a system capable of connecting multiple UAS and other vehicles together and providing them with control inputs for testing control algorithms for multiple SUAS. In addition to connecting multiple vehicles, the system needs to integrate sensors, avionics, and payloads that will be used for controlling the vehicles. The system would need to operate outdoors when conditions are ideal for testing. The system needs to be easy to use and quick to change for a variety of laboratory testing needs. It needs to provide a human operator access for supervision and intervention.

This needs statement highlights that the system must be able to function outdoors, be reconfigurable, connect multiple vehicles, and have the ability to connect a variety of sensors, payloads, and avionics. A hypothetical example of a series of tests that the system would support could be as follows:

> The first test is to implement an reinforcement learning cooperation algorithm to image an area using aircraft A with a multi-spectral camera and direct aircraft B to areas of interest. A follow on test is to then implement a formation-flying algorithm using the same vehicles. The algorithm requires

sonar readings for control in a tight formation to simulate a refueling operation. The desire is to have a system that can support both flight tests without months of development and independent testing on subsystems.

2.2 Concept of Operations

A ConOps is a description of how the characteristics of a system from the viewpoint of the user. It does not have to give specifics such as that found in a users manual but will highlight how one would start, stop, interact with, and change a system. During the preliminary design process the system references in the needs statement (see Section 2.1) was given the name of Clark. The ConOps given below is based on that needs statement.

> Clark will to be an open architecture where hardware changes can be made easily and quickly with little to no software development. The open architecture will be beneficial when projects shift and new hardware is needed. Clark should be accessible to VSCL students current and future to be used on a variety of projects. Documentation and tutorial videos will show students how it is to be used provide a basis from which systems can be built. Students should be able to understand the system architecture quickly to the extent that they can modify it. Clark will be easy to modify or swap the controller and/or number of agents used in testing. As a research tool, Clark shall providing data logs, debugging information, test supervision, and fail-safes to the mission control station operator. Clark will be capable of operating four vehicles with up to five payloads by a single mission control operator. For safety, pilots will oversee each vehicle and if necessary retake control.

The concept of operation highlights usability and flexibility. The entire idea of this system is to allow users to quickly learn, use, and modify the framework to fit individual projects. Ultimately, Clark will be a software package with many support documents suggesting hardware and defining its use as mentioned previously.

2.3 Requirements

In general requirement present *what* is to be accomplished by a system. The word “shall” will be used for requirements that must be met for the system to be considered complete and adequate. The word “should” will be used for requirements where a specified characteristic is desirable but may not be necessary for the system to be fully functional and meet the expectation of the needs statement and ConOps.

For the purposed of this work, the requirements will be broken into three levels of requirements. The highest level are the Defined Requirements, followed by the Defined Requirements, completed by the Design Requirements. The Defined Requirements seek to capture all aspects of the ConOps and transcribe Needs in to requirement statements. The Defined Requirements should not begin to address specifics on how a Need will be resolved. The Derived Requirements are intended to give more structure to the Defined Requirements. An example of this would be an aircraft system with a Defined Requirement that it shall be safe. This can be broken into more concrete requirements such as the system shall provide autonomous return to launch and landing capability. The Defined Requirement can then be broken into Design Requirements. Design Requirements implicitly begin to answer the question of *how* that need will be met. The Design Requirements are the most coupled of the requirements and often have a cascading effect on one anther. This can be a source of conflict for the design that should can be avoided by constructing the Defined Requirements as a structure to Design Requirements. However, even with well posed defined and Derived Requirement the Design Requirements are highly dependent on one another and must be reevaluated after each modification or addition of requirements. Going back to the previous example, a Design Requirement could be that the system shall have an autopilot capable of autonomous landings or the system shall have a dual-band fault-tolerant wireless connection with the ground for tele-operated landings. These requirements specify one of the ways that the aircraft system will fit the safety need.

A full list of the requirement can be found in the Appendix A. In the following sections, requirements will be justified, explained, and evaluated. The explanations will tie the requirements directly back to the needs statement, ConOps, and other requirements. The justifications will explain *how* the explanation solves the need presented. The evaluation will detail and expand on the system impact such as the limitations, assumptions, consequences, and overall effects of a requirement.

2.4 Defined Requirements

Eight Defined Requirements transfer the need statement and concept of operations into high-level requirements. The following subsections contain all of the Defined Requirements with their associated explanations.

2.4.1 Defined Requirement 1

1. Clark shall be used in an outdoor environment excluding extreme weather

Requirement one was made to ensure that the system would function outdoors with environmental conditions such as wind and head, but with the caveat that that it would not be required to function in extreme weather. Testing outdoors can be more difficult due to larger variations in the environment but ultimately most SUAS operate outdoors. Therefore, the highest fidelity or truest to reality testing can be accomplished in an outdoor setting. The main consequence of this requirement is that the communications must work at longer distances that an indoor arena would require.

2.4.2 Defined Requirement 2

2. Clark shall be easy to use by both developers and operators

Requirement two that states that the system “be easy to use by both developers and operators.” This requirement may appear vague but it is important and necessary. The purpose of leaving it open ended was due to the uncertainty of what exactly would make Clark easy to use. In the following sections more detailed requirements will expand upon how the system should be easy to use. This requirement is important because a primary purposes of this work is to develop a system capable of integrating multiple vehicles and payloads easily and in a short amount of time. One of the key assumptions that will be made is that what is currently easy to be used by members of the VSCL will be easy to use for future members. No attempt will be make in this work to seek user feedback and make resulting design changes. The largest overall effect of this requirement is that it necessitates more requirements to determine the balance between modularity and ease of integration (see Subsections 2.4.6-2.4.7).

2.4.3 Defined Requirement 3

3. Clark shall connect multiple vehicles (agents) over a wireless network

Requirement number three can clearly be pulled from the needs statement and ConOps. Clark is to work with SUAS which fly and are generally not tethered to the ground. Therefore, it was determined that the connection be over a wireless network was made based off of a lack of any alternatives to connecting moving vehicles and the prevalence of wireless communication hardware.

2.4.4 Defined Requirement 4

4. Clark shall connect multiple sensors, payloads, and/or avionics aboard a single agent over one or several wired network

Requirement four was generated because many multi-control algorithms are dependent on the sensors completing the feedback loop. Clark must be capable of connecting multiple sensors to allow for changes in hardware. This requirement naturally opposes requirement two because typically a very modular system is not easy to integrate or work with. However, this requirement ensures that the ease of use does not infringe upon the systems modularity but rather extends requirement two in that the modular aspect of Clark must be easy to use.

2.4.5 Defined Requirement 5

5. Clark shall provide a manned mission control station with tools to observe and control all agents and payloads on the same network

Requirement five was defined as a result of the need to be used in a laboratory setting where safety and insight from testing are desired. Prototype controllers are often times tested in an environment where many failsafes are in place and the testing can be terminated if the prototype does not function as intended. This requirement is necessary as it begins to address the Need of allowing a human into the loop. This requirement does not specify the functionality and tools given to the human supervisor, and they will have to be specified as Derived Requirements.

2.4.6 Defined Requirement 6

6. Clark shall be an open architecture

Requirement six extends requirement four. While both requirement could be consolidated into one, it was decided to keep them separate in order to set apart the open architecture from payload modularity. Similar justifications for this requirement exist as previously stated in requirement four but extended to the wireless network hardware and other interfaces.

2.4.7 Defined Requirement 7

7. Clark shall provide a modular multi-agent control architecture

Requirement seven defines one of the key features of the Clark system by bringing information from several components and networks to a single layer for control law implementation. This requirement means that the system should be able to handle changing between various controllers without changing hardware or other parts of the system just but swapping the abstracting layer for another one. There are obvious limitations to this modularity such as hardware dependent control designs but for most applications this requirement holds. An example of modularity in the control architecture may be just in the algorithm such as swapping an linear quadratic controller for an adaptive controller. A more complex situation may be where a formation flight controller can be swapped with a cooperative searching controller without making any other software changes.

2.4.8 Defined Requirement 8

8. Clark shall utilize hardware with specifications conducive to use on a specified agent

Requirement eight was developed after Design Requirements led to the a selection of hardware too large to fit onto an SUAS. It is simply intended to ensure that all hardware selected for the system is compatible with the host vehicle. It is important to note that this requirement does not effect the software design but only hardware selection.

2.5 Derived Requirements

The following subsections contain the twenty one Derived Requirements for the Clark system. The associated Defined Requirement is also included for reference.

2.5.1 Derived Requirements 1.1-1.3

1. Clark shall be used in an outdoor environment excluding extreme weather

    1.1. Clark shall to be used in temperature ranges outside of [0, 70] deg Celsius

    1.2. Clark shall not be used in wet weather such as rain or snow

1.3. The wireless network shall extend to the limit of line-of-sight flight

Requirements 1.1-1.3 set limits of the environmental conditions in which the Clark system should function. These requirements were derived in part from MIL-STD-810G [36] where standards for laboratory testing are set forward. This standard was not used in whole to avoid over constraining the design to meet potentially unnecessary requirements. The requirement on wireless communication range up to one mile stems from the FAA Rules Part 107 [37] that SUAS operations must be conducted within line-of-sight of the pilot. This work assumes that a one mile range is beyond line-of-sight operation for SUAS. The first two requirements imply that commercial off the shelf parts can be used and do not require weatherproofing, and requirement 1.3 gives a specific operating range for the wireless communications hardware.

2.5.2 Derived Requirement 2.1-2.4

2. Clark shall be easy to use by both developers and operators

    2.1. Clark shall be programmed in a programming language which can be quickly learned by students without previous programming experience

    2.2. Clark shall be accompanied by extensive software and operations documentation

    2.3. Clark should provide testing tools for individual modules to verify functionality before integration with the system.

    2.4. Clark shall provide data logs for use in debugging, controller analysis, and general system performance.

Requirements 2.1-2.4 are intended to give more context to the Defined Requirement that the system must be easy to use. This clarifies that the programming language must be easy to learn. Often times aerospace students do not have prior programming experience so this requirement helps to ensure that using Clark will be as easy to learn. Requirements 2.2-2.4 give specifications that allow for better utilization of the system. For example, when testing inevitably there will be an error. It is most helpful to have logs files to determine the cause of the error and correct it.

2.5.3 Derived Requirement 3.1-3.2

3. Clark shall connect multiple vehicles (agents) over a wireless network

   3.1. The wireless network shall be distributed

   3.2. Clark shall be able to communicate with a radio transceiver both sending commands and retrieving network data

Requirements 3.1-3.2 state the wireless network should be distributed and duplex. The distributed network allows for the system to communicate without having single points of failure as can be present in other networking topologies. There could also be a situation where a controller must have a distributed network to function properly. Additionally, the system must be able to send and receive on the network for all agents. This requirement was created to purposely exclude any wireless communication system where an agent can only receive or send. The implications of these requirements is that the system must employ hardware that supports mesh or ad-hoc network and provide software capable of interfacing with such hardware.

2.5.4 Derived Requirements 4.1-4.4

4. Clark shall connect multiple sensors, payloads, and/or avionics aboard a single agent over one or several wired network

   4.1. The wired network shall use a consistent protocol for all payloads connected

   4.2. The wired network shall use MAVLink over serial as the protocol to connect to the autopilot

4.3. Clark shall connect to the autopilot through a wired connection

4.4. Clark shall connect to the payload or payload network through a wired connection

Requirements 4.1-4.4 address in what way Clark must connect to multiple payloads. The first requirement is to ensure that only one payload interface needs to be employed at any given instance because all payloads will utilize the same protocol. Specifically, 4.2 selects MAVLink to be the protocol of choice for communicating with the autopilot. MAVLink or micro aerial vehicle link is a common standard for commercial autopilots used on SUAS such as the Pixhawk, IntellinAir, Pixfalcon, SmartAP, and Autoquad autopilots. This requirement does limit the system for being able to integrate with other autopilots which do no utilize MAVLink. However, this does not mean that Clark will not work with another autopilot but simply that it was not designed for use with different protocols. 4.3-4.4 all the connections to the autopilot and payloads to be independent.

2.5.5 Derived Requirements 5.1-5.2

5. Clark shall provide a manned mission control station with tools to observe and control all agents and payloads on the same network

   5.1. The second human operator shall be for backup

   5.2. Clark shall provide a basic graphical user interface for mission supervision

Requirements 5.1-5.2 define what the human operated mission control station will provide to supervise and control the system. The end result of these requirements is that a graphical user interface (GUI) must be used to support the use of the Clark system. This requirement poses some operational constraints but does not affect other requirements.

2.5.6 Derived Requirements 6.1-6.2

6. Clark shall be an open architecture

6.1. Clark shall consist of multiple interchangeable modules that provide an API set (specified by requirements of this document) to a central module

6.2. Clark shall not use machine specific libraries or function sets

Requirements 6.1-6.2 detail in what tools Clark will use to be an open architecture. The use of application programming interfaces (API) is common in website and application development because it provides modularity and accessibility. API's are what make downloading applications onto a smart phone possible without having to recode the program for each phone model. In a similar fashion this requirement stipulates that a common API set be used to communicate with a central module. Requirement 6.2 restricts libraries that can be used but ensures that the software package will run on different hardware.

2.5.7 Derived Requirements 7.1-7.3

7. Clark shall provide a modular multi-agent control architecture

    7.1. Clark shall be able to communicate with the autopilot both sending commands and retrieving state data

    7.2. Clark shall be able to store information of several agents on each agent

    7.3. Clark shall provide an abstraction layer for defining mission communications, computation, and control

Requirements 7.1-7.3 add structure to how information will be stored and commands executed for the mission abstraction layer. These requirements are key to the functionality of the Clark as a modular control framework. 7.2 implicitly distributes the storage of data onto each agent. This mean that each agent will have a database of all states of all agents in the system. However, it importantly does not require that database to be fully populated and constantly updated. 7.3 is a direct descendant of its Defined Requirement and ensures that Design Requirements be made for the abstraction layer.

2.6 Design Requirements

Systems engineering requirement development concludes with the formation of Design Requirements. Rather than enumerate each Design Requirement as in the previous sections this section will address groups of requirements. Design Requirements form the foundation of the overall system design, the software outline presented in Chapter 3, and the hardware selected in Chapter 4. To facilitate in the software development and hardware selection it is helpful to group related Design Requirements together. The final list of requirement contained ninety seven requirements and was cumbersome to navigate. For this reason all Design Requirements were either grouped as system requirements, software requirements, or hardware requirements. Figure 2.3 demonstrates the process of how requirements are categorized. It should be noted that that some requirements are placed into multiple categories.

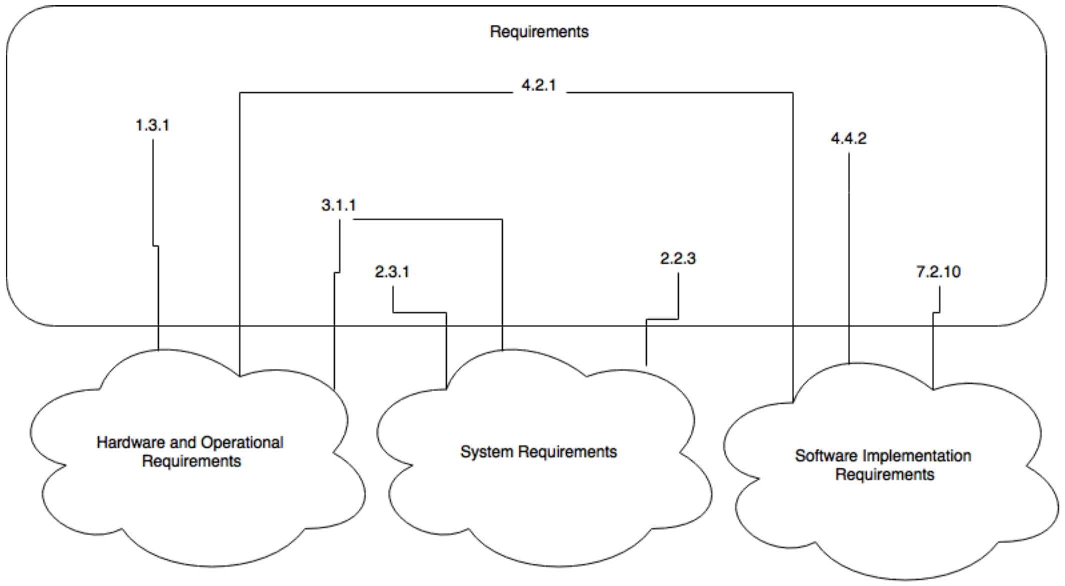

Requirements
4.2.1
1.3.1
4.4.2
3.1.1
2.3.1
2.2.3
7.2.10
Hardware and Operational Requirements
System Requirements
Software Implementation Requirements

Figure 2.3: Categorization of Design Requirements

2.6.1 System Requirements

The system requirements developed during the Systems Engineering process drive the overall system design. These requirements include: 1.3.1, 2.1.1-2.1.2, 2.2.1-2.2.4, 2.3.1-2.3.3, 2.4.3-2.4.4, 3.1.1, 4.3.1, and 4.4.1.

2.6.2 Hardware and Procedural Requirements

Although the Clark system is primarily a software system it is important for specifications on the hardware. Without requirements to define hardware limitations the software would have to function on every conceivable hardware solution; an impractical design. These hardware requirements have clear connections to the ConOps and ensure a feasible system. The hardware and procedural requirements include: 1.3.1, 3.1.1, 4.2.1, and 6.2.1-6.2.2

2.6.3 Software Requirements

Most of the Design Requirements can be categorized as software implementation requirements. These requirement were further categorized by the piece of software that the requirement would take effect. This final categorization was modified continually Throughout the software development process as design choices were made to include more classes and modules than initially considered. With the addition of these classes and modules the final categorization of the software Design Requirements had to be continually updated. Although this process was tedious it proved to be valuable in maintaining the software well organized and readable to future users. Tables 2.1, 2.2, and 2.3 contain the categorized list of Design Requirements. Chapter 3 explains the implementation of these requirements and the resulting software architecture.

Table 2.1: Mission abstraction layer and Clark layer Design Requirements.

Software Component	Design Requirement
Mission Abstraction Layer	
	2.4.2. There shall be a high level log consisting of mission level information
	7.3.1. The mission module shall have access to all functions within Clark
	7.3.2. The mission module shall be able to have a defined sample period
	7.3.3. The mission module shall be able to modify the order and frequency of communication within a sample period
	7.3.4. The mission module shall be able to modify the order and frequency of commands within a sample period
	7.3.5. The mission module shall be able to terminate all connections and shutdown
	7.3.6. The mission module shall export updated states to a CSV file
	7.3.7. The mission module shall have a function for checking for heartbeat timeouts
	7.3.8. The mission module shall have a function for adding data to a buffer
Clark Layer	
	2.4.1. There shall be a lower level log consisting of interface information
	4.3.1. Clark shall connect to the autopilot through an autopilot interface
	4.4.1. Clark shall connect to the autopilot through an payload interface
	6.1.1. Clark shall be the central module to which all interfaces connect
	6.1.2. Clark shall communicate with the user interface through a UDP connection
	7.2.6. Clark shall provide a function to send data to the autopilot, radio, or payload from a buffer.
	7.2.7. Clark shall provide a function to send data to the autopilot, radio, or payload directly.
	7.2.8. Clark shall provide a function to read data to the autopilot, radio, or payload.

Table 2.2: Clark layer, radio interface, and autopilot interface Design Requirements.

Software Component	Design Requirement
Clark Layer	
	7.2.9. Clark shall be capable of being run in a testing mode where any interface can be enabled/disabled.
	7.2.10. Clark shall dynamically add agent as their heartbeat messages are received
	7.2.11. Clark shall be able to sync GPS time with system time
	7.2.12. Clark shall provide an outbound data buffer
Radio Interface	
	3.2.1. The radio interface shall have a function to check for data in waiting on the network
	3.2.2. The radio interface shall have a function to retrieve data in waiting
	3.2.3. The radio interface shall have a function to send data from a buffer to a specific destination(s)
	3.2.4. The radio interface shall have a function to retrieve the connected transceiver?s network address
	3.2.5. The radio interface shall have a function to change parameters of the radio transceiver
	3.2.6. The radio interface shall have a function to verify that the connection is functioning properly
	3.2.7. The radio interface shall have a function to close its connection with the radio transceiver
Autopilot Interface	
	4.2.1. DroneKit and Pymavlink should be used for interfacing with the autopilot
	4.3.2. The autopilot interface shall have a function for retrieving state information from the autopilot
	4.3.3. The autopilot interface shall have a function for commanding the autopilot
	4.3.4. The autopilot interface shall have a function for verifying the connection with the autopilot and that it is functioning properly
	4.3.5. The autopilot interface shall provide Clark with GPS time
	4.3.6. The autopilot interface shall have a function for closing the connection with the autopilot

Table 2.3: Payload interface, human interface, and agent interface Design Requirements.

Software Component	Design Requirement
Payload Interface	
	4.4.2. The payload interface shall have a function for receiving data from the payload(s)
	4.4.3. The payload interface shall have a function for sending data to the payload(s)
	4.4.4. The payload interface shall have a function for closing the connection with the payload(s)
Human Interface	
	5.2.1. The GUI shall provide each agents state data (lat, lon, alt, vel, mode)
	5.2.2. The GUI should provide time since last heartbeat
	5.2.3. The GUI shall provide the ability to activate a multi-agent controller
	5.2.4. The GUI should provide the ability to set individual agents to different modes
	5.2.5. The GUI shall be able to terminate the mission
Agent Interface	
	7.2.1. The agent interface shall be able to unpack all clarklink messages
	7.2.2. The agent interface shall be able to raise new data flags
	7.2.3. The agent interface shall be able to pack all clarklink messages
	7.2.4. The agent interface should be able to store all data for all agents on the network
	7.2.5. The agent interface shall use clarklink

3. SOFTWARE DEVELOPMENT

The software for the Clark framework was developed in stages in accordance with Spiral Development. The stages of development were considered complete when all the requirements put forward in Chapter 2 were met and the developed code was tested. First, the basic Python code for the radio interface, Clark, mission, and autopilot classes was developed. After testing and ensuring proper function, the Clark agent and payload interface classes were developed. All software was then thoroughly tested. Finally, additional functionality was added to meet all the software requirements. The use of Spiral Development in software development is necessary due to the disconnect between the requirements, which define the software's function, and the programmatic manner of how it is coded. Additionally, there can be cases where implementation of one requirement interferes with that of another requirement. In these cases both the requirements and implementation are revisited. The resulting software application fulfills all software Design Requirements and is operationally alike the ConOps.

This Chapter will give a top down explanation of the final Clark software application beginning with an overview of the software architecture in Section 3.1. The overview is followed by Sections 3.2-3.5 which explain the purpose and design of the the software configuration, mission abstraction layer, Clark layer, and peripheral classes. Finally, Sections 3.6-3.9 explains additional software components that were developed after the first iteration of the Clark framework.

3.1 Clark Software Overview

The Clark software package is written in the Python programming language (Python 2.7) and is intended to be operated as an application. The Python programming language was chosen due to its readability and ability to be learned [38]. Version 2.7 was selected due to several dependencies only being supported in Python 2.7. DroneKit [39] and Google's Protocol Buffers [40] were the primary dependencies that were not fully supported in Python 3.

The Clark application is executed as a single process without multi-threading. It is executed from the command line but could easily be converted into a daemon. Multi-processing would allow for faster performance but at the cost of readability and ease of use for a novice Python programmer. However, future investigations could lead to versions of Clark with multi-processing schemes that separate control computation, peripheral communication, and data handling.

3.1.1 Software Architecture

Clark is framed in a three-tiered software architecture consisting of several Python classes. The top tier is a single class that allows for a control designer to implement multi-agent control. This top tier is completely dependent on the lower tier for access to information and methods for commanding control. The second tier is comprised of the Clark class. The third tier consists of the radio, autopilot, and payload classes. This three-tiered architecture shown in Figure 3.1 can be expanded to include the Clark agent class that provides data storage and handling. The Clark agent class is explained in detail in Section 3.7. It is important to note that architecture depicted in Figure 3.1 represents the software architecture on a single agent.

3.2 Software Configuration

The Clark application is configured at runtime by a configuration file. The configuration file allows a user to define a specific mission and agent at runtime making it easy to configure a multi-agent mission. This means that a different configuration file would be created for each agent and for each mission. Many of the parameters would be constant between agents but some such as the selected agent identification number should be

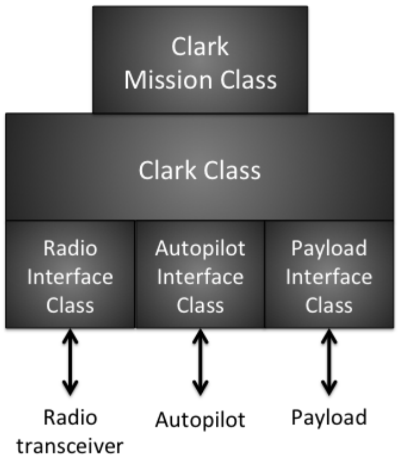


Figure 3.1: The three-tiered architecture of the Clark software

different. The configuration file consists of three sections.

3.2.1 Mission Configuration

The first section contains all of the parameters utilized by the mission abstraction layer. These parameters include: selected agent identification number (ID), mission ID, mission level logging file name, logging level, desired comma-separated value output file name, heartbeat timeout limit, and loop frequency. The selected agent ID designated a ID by which the Clark system will recognize and agent. The mission ID is mean to be a unique integer designated to each mission. When initialized warnings will be received when a mission IDs do not match between agents. The loop frequency is second in importance only to selected agent ID because it determines the sample period of an agent. This section is also intended to configure any parameters used by the mission layer to avoid numerical entries in the code.

3.2.2 Clark Configuration

The second section gives the necessary parameters for configuring the central class of the bottom tier, the Clark class. These parameters include: testing mode boolean, radio enable boolean, payload enable boolean, autopilot enable boolean, software in the loop enable boolean, GUI enable boolean, GUI IP address, GUI port, and GUI timeout. The testing mode boolean parameter allows for troubleshooting by enabling/disabling the next five parameters. If testing mode is disabled, then the enable/disable parameters are ignored and the framework will operate in "Normal Mode". In this mode the radio, autopilot, and payload are enabled for an agent configured to type "vehicle" (see Subsection 3.2.3) while an agent configured to type "mcs" will only have the radio and GUI enabled. Hence the GUI parameter is only used if an agent is configured as a mission control station (MCS) type.

3.2.3 Agent Configuration

The third section contains all agent specific parameters. This configuration section is also read by the Clark class. Many of these parameters are passed from the Clark class to the peripheral classes to configure connection ports and addresses, but for simplicity they are all read by the Clark class. The parameters in this section may include: agent ID, agent type, radio port, radio baud rate, autopilot port, autopilot baud rate, autopilot update rate, autopilot wait until ready boolean, and several payload parameters. The agent ID integer must match the selected agent ID given in the first section or an error will be given. The purpose of this repetition is to ensure that each agent is configured correctly. The agent type may be either "mcs" or "vehicle." The "mcs" type is reserved for the agent connected to the GUI where a human supervisor is located. The Clark framework does not limit the number of MCS to one and more could be connected. All other agents are of the type "vehicle." All other parameter in this section are agent specific and will vary with changes in hardware.

3.3 Mission Layer

The mission abstraction layer provides a control algorithm all of the information and methods to enact multi-agent control. It consists of a single class, the Clark mission class, with multiple methods used for interacting with the other classes to transmit and receive information. This layer cannot connect to or control any peripherals without the Clark class. For this reason it is dependent on the Clark class and peripheral classes. Ultimately, this class is utilized by calling the execute mission method. The execute mission method enters an infinite loop which enacts communications and control. The operations performed within this loop can be modified or shifted, but generally follow the process shown in Figure 3.2. It is important to note that executing the execute mission method will only initiate a single process on a single agent. Therefore it must be executed on each mission computer onboard each agent within the Clark system. The execute mission method loops at a frequency specified in the configuration parameters (see Subsection 3.2.1).

3.3.1 Important Mission Layer Functions

The mission abstraction layer performs many vital functions to enabling multi-agent control including instantiating the Clark class, verifying mission IDs between agents, checking for agent timeouts, recording mission data, and performing control calculations. During execution, the mission layer verifies the mission ID of other agents as they try to connect. This is intended to mitigate the dangerous situation of one agent running a different mission that the others. It also checks to see if other agents have lost connection or left the network by checking for a heartbeat message timeout. The timeout parameter is given in seconds in the mission configuration section explained in Subsection 3.2.1. The most important functions of the mission layer are to record data and calculate control so that a given mission can be accomplished. These two functions are accomplished through

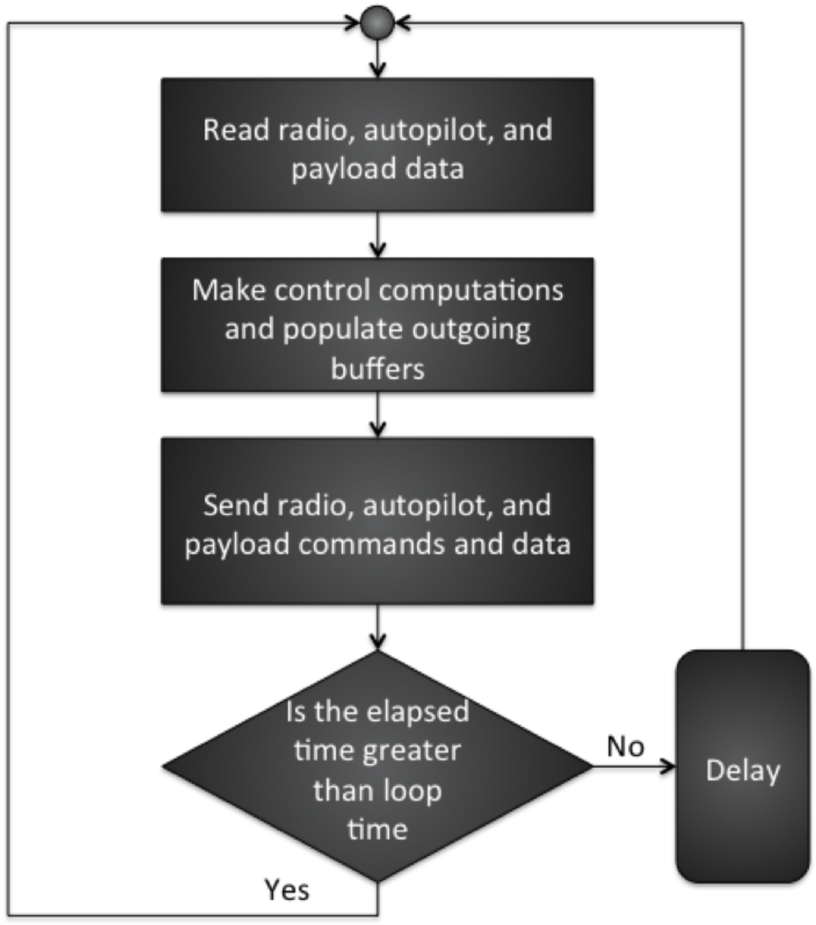

Read radio, autopilot, and payload data
Make control computations and populate outgoing buffers
Send radio, autopilot, and payload commands and data
Is the elapsed time greater than loop time
No
Delay
Yes

Figure 3.2: Generalized procedure of the mission loop

the “mission_compute” and “export_data” methods.

3.4 Clark Layer

The purpose of the Clark layer is to maintain communication between the hardware interface layer and mission layer in addition to storing outbound and inbound data. Figure 3.3 shows how the Clark class maintains communication between the mission layer, Clark agents, and hardware interface layer. Clark can be thought of as the middle man between the mission abstraction layer and all other parts of the software. In the case that the agent

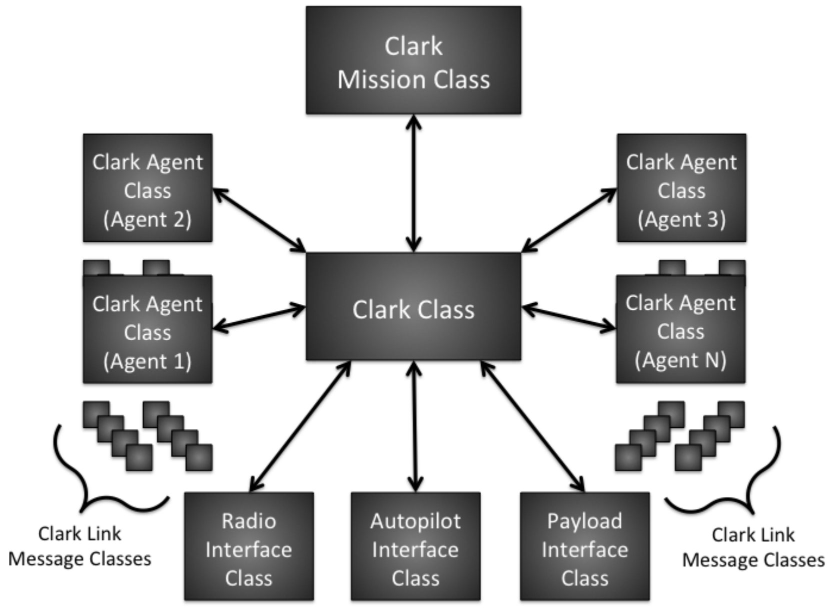

Clark Mission Class
Clark Agent Class (Agent 2)
Clark Agent Class (Agent 3)
Clark Agent Class (Agent 1)
Clark Class
Clark Agent Class (Agent N)
Clark Link Message Classes
Radio Interface Class
Autopilot Interface Class
Payload Interface Class
Clark Link Message Classes

Figure 3.3: The class structure—the Clark class maintains communication between all other classes

is configured as the MCS, this layer also handles communication directly with the GUI. In addition to the communication the Clark layer has a responsibility of initializing all interfaces and ensuring proper function prior to entering the mission loop. Another function of the Clark layer is to allow a user to enter "testing" mode where hardware interfaces can be enabled as disable. This allows a developer to isolate and correct bugs more quickly than if all the interfaces had to be running at once. Next, Clark also provides the mission layer an outbound buffer for each of the interface classes. Finally, one of the most important functions of the Clark layer is to dynamically expand the "Clark Agents" list when new agent connect to the network. The detail of the Clark Agent class will be explained in Section 3.7. Initially Clark only assumes that one vehicle exists on the network until a message is received from another agent. When messages are received from other agents then Clark expands its "database" to store that agent's information as well.

3.5 Hardware Interface Layer

As seen in Figure 3.3 the interfaces communicate only with the Clark layer and hardware components. The entire purpose of the hardware interfaces are to handle hardware dependent variables so that the system can communicate. The hardware interfaces make Clark an open architecture because they can be swapped in and out for varying hardware. Clark can operate with up to three hardware interfaces: the radio interface, the autopilot interface, and the payload interface. For normal operation, it is required that the radio and autopilot interfaces be used. The payload interface is optional due to the fact that some application may not require any added sensors or payloads.

3.5.1 Radio Interface

The radio interface class manages sending and receiving data from a radio transceiver on the mesh network. When initialized the interface establishes a connection with the transceiver and verifies the connection. The verification method is executed by the radio interface and not the Clark class because the method used to verify proper function of the transceiver can vary with selected hardware. Only the result of the verification test is returned to the Clark class. The radio interface then provides Clark a standard API set with some additional functions. The standard API set for all peripherals consists of three methods: talk, listen, and close. This API set is a result of requirements 3.2.1 - 3.2.7. The full radio interface API set can be found in Table 3.1. Essentially, the radio interface provides a tool set to Clark to then use for interacting with the radio mesh network as seen in Figure 3.4.

3.5.2 Autopilot Interface

The autopilot interface class handles communication with the autopilot which requires translation of commands into MAVLink [41]. MAVLink is required as specified in require-

Table 3.1: The Radio Interface API Set

Method Name	Arguments	Function
Talk	Destination & Data	Send data to specified destination
Listen	None	Returns data in the input buffer
Data_in_waiting	None	Returns the number of bytes in the input buffer
Get_address	None	Returns the hardware address of the transceiver
Verify_connection	None	Verifies communication with the transceiver
Close	None	Terminate hardware connection

ment 4.2.1. The autopilot interface makes use of the DroneKit module [39] to interfacing with the autopilot. DroneKit has been used in creating a research platform because of its user-friendly Python API [42]. DroneKit is built upon PyMAVLink which supports communication with all autopilots that use MAVLink. Similar to the radio interface, the autopilot interface initializes by establishing a connection with the autopilot and verifying that connection. It then provides an API set to Clark for interacting with the autopilot. In compliance with requirements 4.3.1 - 4.3.6, six methods were developed for the autopilot API set and are found in Table 3.2. The "talk" method is more complex with the autopilot interface as a command must be given as an input. Currently, three commands are made available by the autopilot interface: "waypoint", "do_set_servo", and "rcoverride". The "waypoint" command will autonomously direct a vehicle to a location by designating a latitude, longitude, and altitude and then changing the autopilot mode to "Guided". The altitude parameter only takes effect on air vehicles. The "do_set_servo" and "rcoverride" commands cause the autopilot to output a pulse width modulated (PWM) signal to a specified control servo on the vehicle. Using these commands the mission layer can alter control surface deflections or motor current. The difference between the two is that

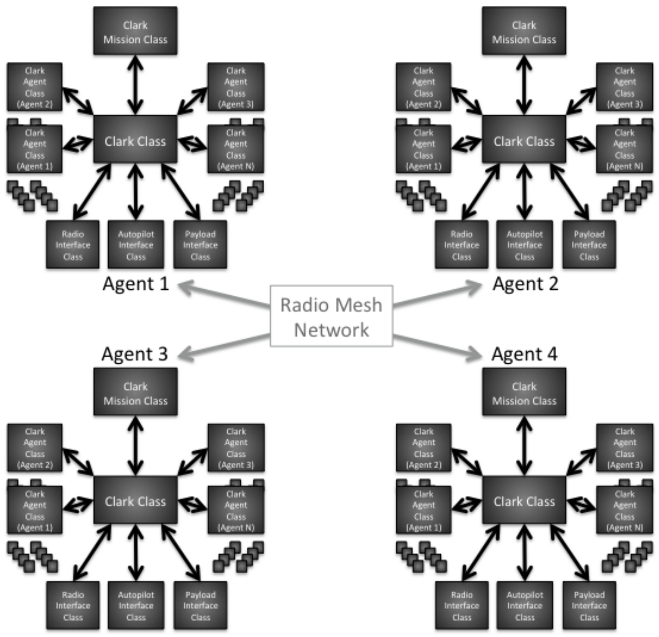

Clark Mission Class
Clark Agent Class (Agent 2)
Clark Agent Class (Agent 3)
Clark Agent Class (Agent 1)
Clark Class
Clark Agent Class (Agent N)
Radio Interface Class
Autopilot Interface Class
Payload Interface Class
Agent 1
Agent 2
Radio Mesh Network
Agent 3
Agent 4

Figure 3.4: An instance of the Clark software must be executed on each agent during multi-agent operations

"do_set_servo" is used on an unspecified channel while "rcoverride" is used on a channel configured to an remote control (RC) channel. All output channels on an autopilot are generally configured as a RC channel or unspecified channel. These channel configurations are made in the autopilot.

Table 3.2: The Autopilot Interface API Set

Method Name	Arguments	Function
Talk	Command & Data	Send data to specified destination
Listen	None	Returns data in the input buffer
Verify_connection	None	Verifies communication with the transceiver
Close	None	Terminate hardware connection

3.5.3 Payload Interface

The payload interface class is designed to handle communication with a single payload or with an entire payload network. The payload can include sensors or other non-flight critical devices such as a computer. One difference from the other interfaces' is that the payload interface does not provide a method to verify the connection upon initialization. The verification method is not provided to the Clark class due to the variability that verification method would have with different sensors. Therefore, in order to simplify the interaction between the payload interface class and Clark class this method was omitted. However, when a payload interface is developed, it is encouraged that it internally verify the connection before completing its initialization. This would allow Clark to verify the connection indirectly because the Clark class waits for the peripheral classes to complete initialization before continuing. Also, to communicate with a payload it may be necessary for the payload interface to translate data into a different protocol. Ultimately, the payload interface functions much like the radio and autopilot interfaces. It provides an API set to Clark as shown in Table 3.3.

Table 3.3: The Payload Interface API Set

Method Name	Arguments	Function
Talk	Command, Channel, & Data	Send data to specified destination
Listen	Channel	Returns data in the input buffer
Close	None	Terminate hardware connection

3.6 Additional Software

In addition to the peripherals, Clark utilizes three other classes to manage information and present the system status to a human supervisor. The Clark Agent class is utilized to unpack, pack, store, and manage all data within Clark. The Clark Agent class utilizes Clark Link classes to store most information. The Clark Link classes make up the Clark Link protocol, which manages data between several vehicles and several payloads. The Clark View class is a GUI application that connects with a MCS agent to allow a user to interact with the system.

3.7 Clark Agent

The Clark agent class holds all of the data for Clark and can be accessed through the Clark class. The Clark agent class instantiates an object for each Clark Link message in the protocol. The Clark Link objects are created using Protocol Buffers. Data is stored as an attribute to these Clark Link objects. The Clark Agent class has three methods for manipulating the data and Clark Link objects. The "unpack" method parses inbound Clark Link messages. The "pack" method stores an array of input data into a Clark Link object and the "pack2bin" method returns a binary sting serialized by Protocol Buffers. The Clark Agent class contains all possible states for a given agent. As explained in Section 3.4 multiple instances of the Clark Agent class are created, and each one of these instances contain the other agents states.

3.7.1 Inbound Clark Link Messages

When a Clark Link message is received by any of the hardware interfaces it is passed directly to this class. Clark Agent then parses and stores the data through the "unpack" method. This is made possible through the use of the Clark Link protocol which maintains three fields for all messages used for automatic parsing and storage. All Clark Link messages begin with message type, agent ID, and GPS time.

3.7.2 Accessing Data In Clark Link Messages

After unpacking a message, Clark Agent raises flags for messages where new data is received to alert both the Clark class and the mission layer of new data. The new data can then be accessed through the Clark agents list maintained by the Clark class (see Section 3.4 for details on the Clark agents list). Data can also be written directly to the Clark Link objects by the mission class through the Clark agents list.

3.7.3 Sending Clark Link Messages

Clark Link messages can be sent by serializing Clark Link object attributes. This can be done by using the “pack2bin” method and placing the returned binary string into an outbound buffer. Clark Agent serializes all outbound messages into binary strings to maximize network throughput. As previously stated, a GPS time stamp is automatically written into the serialization.

3.8 Clark Link

The Clark Link class houses the Clark Link protocol classes. These classes are based off of Google’s Protocol Buffer serialization scheme. Protocol buffers provides fast and efficient serialization of data structure to a binary format. Additionally, messages can be easily added to Clark Link using Protocol Buffer’s C++ to Python compiler. In the end, Clark Link consists of a message set and basic serialization method for storing and serializing data. Table 3.4 shows the messages contained in the fifth version of Clark Link.

As observed in Table 3.4 all messages begin with the same three fields. The agent ID informs Clark which agent the message belongs to so that the message can be forwarded onto the correct Clark Agent object. The message ID allows Clark Agent to determine how to parse the message. The GPS time stamp can be compared with the current GPS

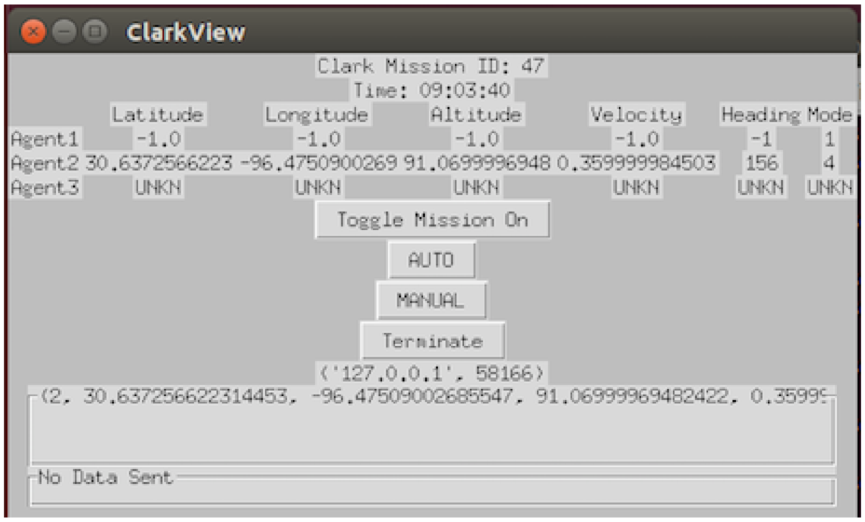
Table 3.4: Clark Link Message Set

Message Name	Fields
Message ID	Message ID, Agent ID, & GPS Time stamp
Heartbeat	Message ID, Agent ID, GPS Time stamp, Mission ID, Agent Type, Clark Link version, Condition, & Number of payloads connected
Agent Status	Message ID, Agent ID, GPS Time stamp, GPS Latitude, GPS Longitude, Altitude, Velocity, Heading, & Autopilot mode
Picture Data	Message ID, Agent ID, GPS Time stamp, GPS Latitude, GPS Longitude, & Altitude
Mode Change Request	Message ID, Agent ID, GPS Time stamp, & Mode
Activate Controller	Message ID, Agent ID, GPS Time stamp, Controller on boolean, Start time, & Optional field
Agent Attitude	Message ID, Agent ID, GPS Time stamp, Pitch, Yaw, & Roll

time to determine how old a message is. The Massage ID class is used exclusively for message type, agent ID, and GPS time stamp.

3.9 Clark View

The Clark View class can be executed as an application to provide a GUI to an operator. It depends on being connected to a agent with the designated type of MCS. It connects to this agent over a UDP/IP connection sending and receiving Clark Link messages. It is important to note that while the GUI uses Clark Link messages it does not utilize the Clark agent class therefore the “unpack”, “pack”, and “pack2bin” methods are not available. The Clark View GUI allows a human operator to view the latitude, longitude, altitude, velocity, heading, and autopilot mode as stated in requirement 5.2.1. It also provides a mean to activate a controller and terminate missions in accordance with requirement 5.2.3 and 5.2.4. Figure 3.5 shows an image of the Clark View GUI.



Clark Mission ID: 47

Time: 09:03:40

	Latitude	Longitude	Altitude	Velocity	Heading	Mode
Agent1	-1.0	-1.0	-1.0	-1.0	-1	1
Agent2	30.6372566223	-96.4750900269	91.0699996948	0.359999984503	156	4
Agent3	UNKN	UNKN	UNKN	UNKN	UNKN	UNKN

```
('127.0.0.1', 58166)
(2, 30.6372566223144453, -96.47509002685547, 91.06999969482422, 0.3599
No Data Sent
```

Figure 3.5: The Clark View GUI prototype

4. SELECTED HARDWARE AND INTEGRATION

The Clark system was prototyped and integrated onto four vehicles for testing. The electronic components were selected in accordance with requirements set forward in Chapter 2. In this chapter, electronic hardware and test vehicles will be presented.

4.1 Electronic Components

This thesis presents a system intended for use on SUAS, and therefore the electronics components selected for testing must function on SUAS without compromising the airworthiness of SUAS. SUAS are by definition small and for this reason weight, power consumption, and size are all important factors for selecting electronic components. The following components were selected with these constraints in mind, in addition to other considerations which will be explained in the following subsections.

4.1.1 Mission Computer

The Clark system was designed to include a mission computer so that the Clark software could be executed. Requirement 6.2.1 states that the mission computer must operate a UNIX-based operating system and be capable of connecting to a radio, payload, and autopilot. Additionally, the mission computer is required to have a total weight less than 500 grams including its power source. For the reason cited above, a Raspberry Pi 2 Model B Version 3 was selected. Figure 4.1 shows a Raspberry Pi 2. A Raspberry Pi 2 features seventeen general purpose input/output (GPIO) pins, four USB ports, and Ethernet port. This large number of GPIO pins meets requirement 6.2.2. It has a high performance 900 MHz quad-core ARM Cortex A7 processor and small 85.6 mm x 56.5 mm footprint. It weighs 45 grams and requires less than 1 W of power. It is commercially available for $35 and can be configured with a number of UNIX-based OS. The Raspberry Pis used

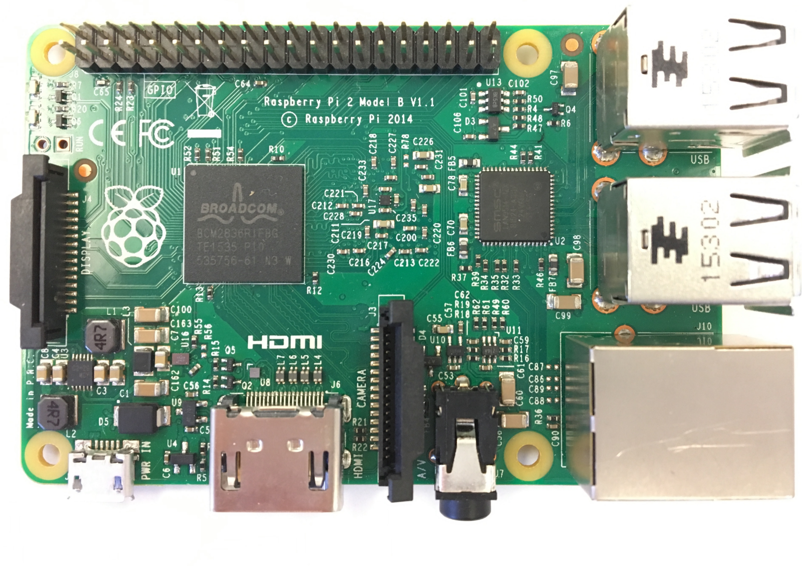


Figure 4.1: The Raspberry Pi 2

in testing utilized the Raspian Lite OS [43]. Raspbian is based off of Debian Linux and comes pre-installed with Python 2.7.

4.1.2 Radio Transceiver

The radio transceiver must be capable of functioning at ranges up to one mile as stated in requirement 1.3.1 and must be capable of forming a mesh network (Req. 3.1.1). With these requirements and the previously mentioned constrains of size, weight, and power consumption, the Digi XBee Pro 900HP radio transceivers were selected. Figure 4.2 shows the XBee Pro 900HP. It operates with the DigiMesh protocol, has a small footprint of 22mm x 33mm, and consumes less than 1W of power. It features a range of up to four miles with a relatively high bandwidth of 200 Kbps. The DigiMesh protocol, built on the IEEE 802.15.4 standard [44], treats all connected radio modules as routers, making for simple configuration and easy expansion of the network. Messages can be sent to specific

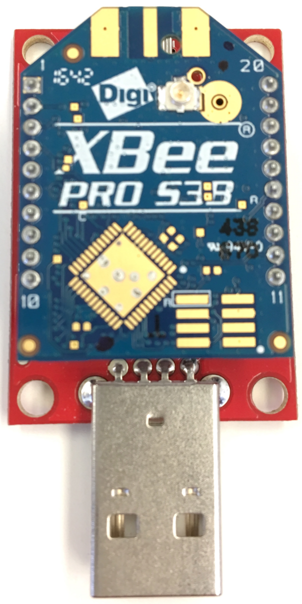


Figure 4.2: The XBee Pro 900HP radio transceiver

radios on the network or broadcast throughout. The XBee Pro 900HP is commercially available for $39.

The XBee transceivers are connected to the mission computer through USB by using an XBee Explorer Dongle. The radio interface class is able to utilize the XBee Python module which provides a user friendly API for send and receiving data. The XBee module can be configured using Digi International's X-CTU program. For this thesis the XBee modules were configured to operate in "API mode" which allows for messages to be addressed or broadcast. This mode confirms packet delivery but has more packet overhead than "Transparent mode." The radios could be reconfigured to operate in transparent mode without modification of the code but then all messages would be broadcast throughout the mesh network.

4.1.3 Autopilot

Requirement 4.2.1 requires that the autopilot utilize the MAVLink protocol to enable communication with the autopilot interface class. Many commercially available MAVLink based autopilots are available that support SUAS operations. However, to enable the testing of heterogeneous vehicles, the Pixhawk [45] autopilot was selected. The Pixhawk, shown in Figure 4.3, is a powerful microcontroller that can be configured with several version of firmware. The ArduPilot firmware stack [46] supports fixed-winged air vehicles (ArduPlane), rotor-winged air vehicles (ArduCopter), ground vehicles (ArduRover), and autonomous submersible vehicles (ArduSub). Furthermore, the VSCL commonly uses this autopilot. The Pixhawk features a 32-bit ARM Cortex M4 processor with a 32-bit

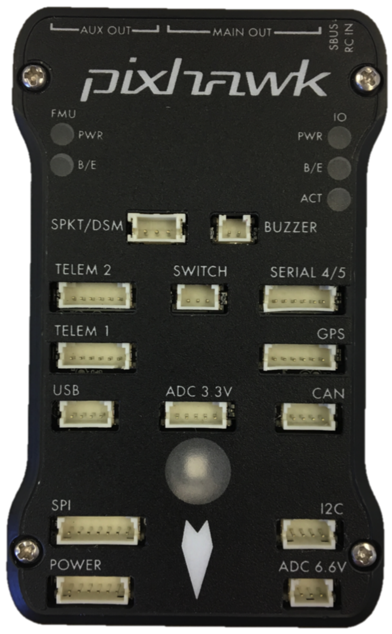


Figure 4.3: The Pixhawk autopilot

failsafe co-processor. A MPU6000 accelerometer, ST Micro 16-bit gyroscope, ST Micro 14-bit magnetometer, and barometer are build in. It can interface directly with an externally mounted GPS module and can be connected to the mission computer utilizing one of the five UART ports. It weighs 45 grams and has a footprint of 50mm x 81.5mm. The Pixhawk is commercially available for $300.

The autopilot is connected to the mission computer by USB. The autopilot interface class utilizes DroneKit which supports the Pixhawk autopilot. The speed at which DroneKit reads messages from the Pixhawk can be configured in software but is limited to the Pixhawk MAVLink message output rate of 50 Hz.

4.1.4 Payload

Although no requirements were set for payloads other than connection type, the selected payload had to comply with the general constraints of a SUAS. A Raspberry Pi 2 with an integrated Raspberry Pi camera [47] was selected as the payload. The camera is a 5.0MP sensor with a fixed-focus. It connects to the Raspberry Pi directly through a ribbon cable and the Raspbian OS contains pre-installed software for communicating with it. It is commercially available for approximately $25. Figure 4.4 shows the small size of the Raspberry Pi camera with a ribbon cable attached to a Raspberry Pi.

The payload was connected to the mission computer using TCP/IP over Ethernet. The MC acted as the TCP server while the payload acted as the client. Rather than send Clark Link messages, the payload sends ASCII strings to test the ability of the payload interface to translate data into a Clark Link message.

4.1.5 Power

AmazonBasics 1200 mAh power banks were used to power each mission computer and the payload. The mission computers powered the XBee transceivers through USB. The Pixhawk is powered using a lithium polymer (LiPo) 1500 mAh battery.

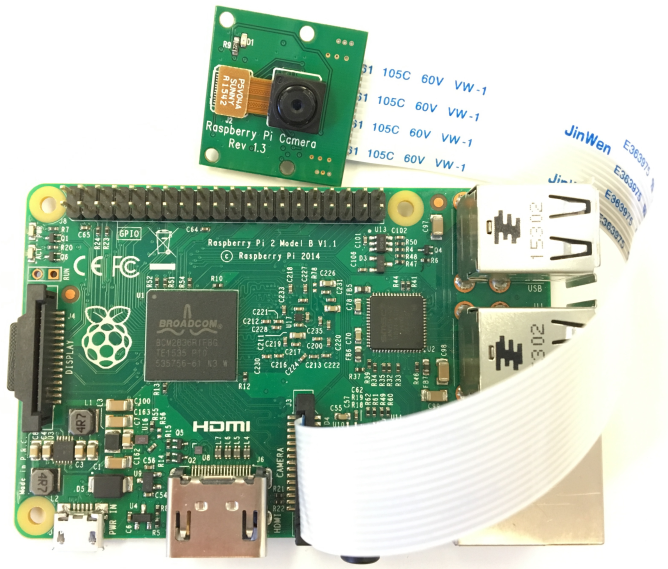


Figure 4.4: The Raspberry Pi camera

4.1.6 MCS Laptops

The MSC is designed to be capable of running on any OS as long as the dependencies are supported. However, only Linux machines have been tested and shown to work. Ubuntu 14.04.3 and 16.04 have been tested and shown to function.

4.1.7 Failsafe Multiplexer

A failsafe multiplexer was used to provide safety to testing. The 3DR failsafe multiplexer is a 8x4 PWM pass through multiplexer with a single select signal. The select signal was wired to an auxiliary channel on the remote control receiver so that at any time the remote control pilot could take full command of the vehicle. The failsafe multiplexer is commercially available for $19.

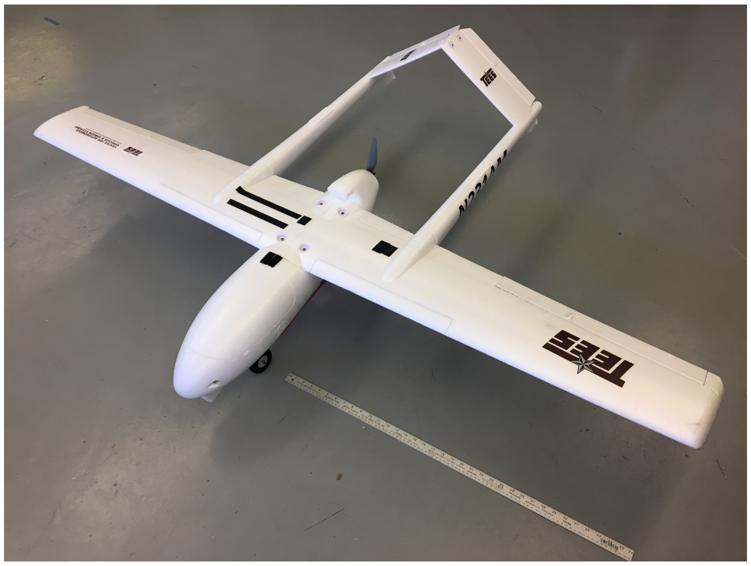


Figure 4.5: The Ready Made RC Anaconda

4.2 Fixed-Winged Unmanned Aerial Vehicles

Two fixed winged UAVs were selected for testing. The Ready Made Anaconda was selected as the primary platform for fixed wing air testing given the experience the VSCL has in its operation and its large payload capacity when compared to vehicles in its class. The Super Cub was also selected to test Clark's ability to affect control. The Super Cub has been used by the VSCL as a platform for developing an online system identification system (SYSID) where precise inputs and near real-time feedback to a supervisor are needed.

The Ready Made RC Anaconda features a airframe with a 2 meter wing span and total length of 1.3 meters. It has a twin boom empennage with an inverted "V" tail configuration and can carry up to 4 pounds of payload. Figure 4.5 shows the Anaconda airframe. It is commercially available for $200.

The anaconda was outfitted with payload mounting system to allow for easy access to payloads. The MC, Raspberry Pi camera, and power banks were attached to a mounting plate as seen in Figure 4.6.

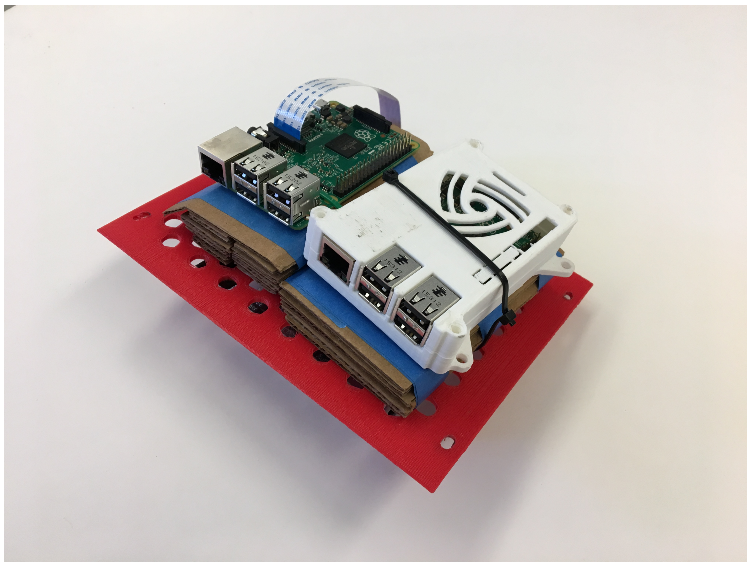


Figure 4.6: Anaconda payload mounting plate with attached MC and payload

The 1/4 scale Super Cub feature a traditional "tail-dragger" airframe with large defection surfaces on the ailerons, elevator, and rudder, making it a good candidate for SYSID. It has an 8.8 feet wingspan and can carry up to 12 pounds of payload. It is commercially available for $700. Figure 4.7 shows the Super Cub airframe.

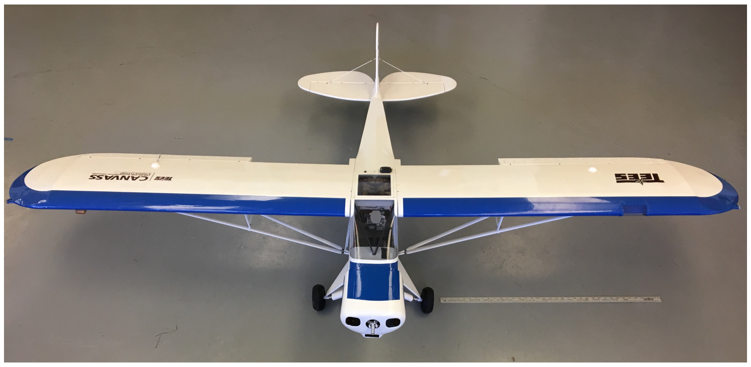


Figure 4.7: The 1/4 Scale Super Cub

The MC computer was mounted into an existing data acquisition stack.

4.2.1 Rotorcraft Unmanned Aerial Vehicle

The DJI F450 is a small quadcopter and was selected to demonstrate Clark support of rotorcraft and the compact size of the Clark system hardware. The F450 has a wheelbase of 450mm and a maximum payload of 1200 g including flight batteries. It is commercially available for $240. Figure 4.8 shows the F450 vehicle.

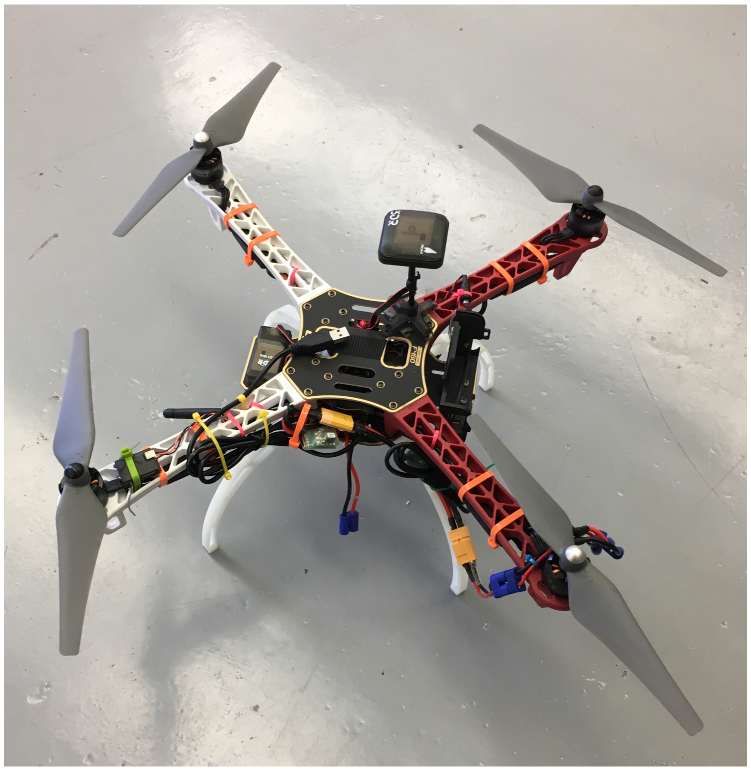


Figure 4.8: The DJI F450

4.2.2 Unmanned Ground Vehicle

The Traxxas Slash was selected to demonstrate compatibility with ground vehicles in addition to air vehicles. The Traxxas Slash is a 1/16 scale vehicle measuring 14 inches long and features a high torque brushed motor. Figure 4.9 shows the Slash vehicle.

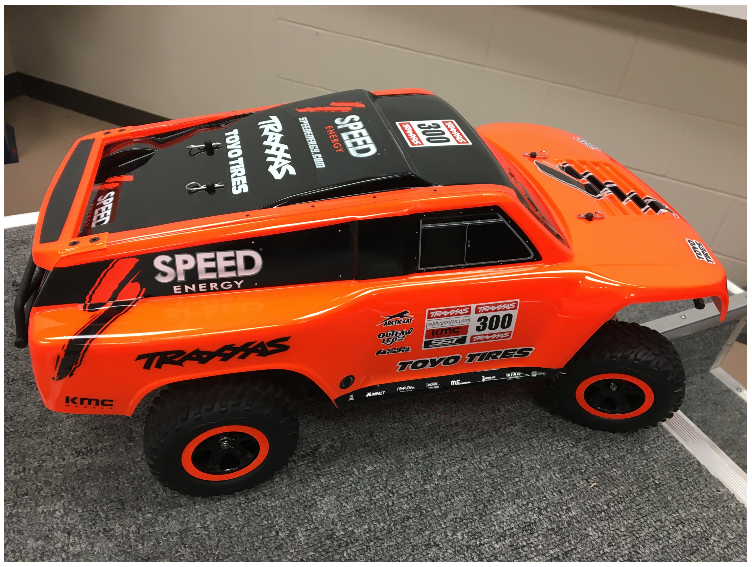


Figure 4.9: The Traxxas Slash

5. TESTING AND RESULTS

As the Clark software was developed and integrated onto hardware and vehicles, it became necessary to prove its functionality as a modular multi-agent control framework. A usable multi-agent control framework must be able to communicate information between all agents within the framework. Additionally, this thesis requires integration of payloads into the framework and this too must be shown. To facilitate usability of the framework it is also important to quantify the limitations of the system such as maximum sampling frequency and throughput. This chapter presents tests and results designed to prove Clark as a multi-agent control framework. The testing includes bench tests specific to determining the limitations of the software, and flight tests as a final demonstration of Clark as a modular multi-agent control framework with integrated payloads.

The results shown in this chapter were extracted from test data gathered by the author. All analyses are performed using MATLAB, Google Earth, Python, and X-CTU.

5.1 Bench Tests

Bench testing was conducted using the electronic hardware listed in Chapter 4 but not onboard the flight vehicles. This was done to ensure the system was safe prior to flight testing. Additionally, bench testing provided an opportunity for the software to be tested class by class throughout the Spiral Development process so that the necessary design and requirement changes could be made as needed.

5.1.1 Maximum Loop Frequency

As explained in Chapter 3, the "execute_mission" method enters into an infinite loop that can have its frequency configured by the user. In Chapter 4, the Raspberry Pi 2 was selected as the MC and has a finite computational limit limiting the maximum sampling frequency. Therefore, it was necessary to quantify the maximum sampling frequency that could be achieved on a Raspberry Pi. A test consisting of two agents was conducted to experimentally determine the fastest loop frequency. A two-agent system was selected so that computational time dedicated to the transmission and reception of messages would be included. The loop frequency is maintained by the "loop_delay" method of the Clark mission class. A logging statement was added to this method to output the calculated amount of delay time needed to maintain a constant loop frequency. A negative delay time indicated that the loop was no longer maintaining the specified sampling frequency and the limit was reached. The testing began with an initial sampling frequency of 1 Hz and was incremented by 1 Hz if the loop frequency was maintained for two minutes. The maximum loop frequency was found to be 13 Hz. It should be noted that this was for a two-agent system and the maximum loop frequency may decrease with an increasing number of agents because of the increased number of messages being transmitted and received.

The same test was also conducted on a Ubuntu machine with an i5 processor, and the maximum loop frequency was determined to be 98 Hz. This second test was conducted simply to show that the maximum loop frequency is also a function of the selected processor and if a specific application requires a faster loop frequency, then a more capable processor can be selected.

5.1.2 Message Latency

For the purposes of this thesis, message latency is defined as the amount of time between the packing of a message into an input buffer and extracting the information within the message into the Clark agent to be used by the mission layer. This implies that packing, unpacking, and retrieval from storage are all included in the message latency. This definition of message latency captures the time delay in context of being able to use information for control as captured in the following scenario: GPS coordinates are packed into a message on agent "A" and then transmitted to agent "B". The message is unpacked by agent "B" into a Clark agent field, and then called by the mission layer for control calculations on agent "B". In simpler terms, it is the amount of time it takes for a piece of information to be sent between two vehicles and be made available to a control algorithm.

The message latency testing was conducted in five stages, all using a single Raspberry Pi.

- Stage 1: Message latency as a function of testing time
- Stage 2: Message latency as a function of sample frequency
- Stage 3: Message latency of a two-agent system
- Stage 4: Message latency of a three-agent system
- Stage 5: Message latency of a four-agent system

A single Raspberry Pi was used to ensure that the same clock was being used for calculating the latency. The Raspberry Pi executed multiple instances of the Clark software during testing which may have led to conservative results. The software was configured to testing mode (see Subsection 3.2.2) so the autopilot interface and payload interface could be disabled. Each Clark instance outputs a comma-separated value file that contained the time a message was sent as well as the time that same message was received. The difference of these two times is the message latency. The message latency was calculated for the entire test creating a message latency set. A simple Python script was developed to parse and perform statistical analysis on this set.

After testing began, it was noted that the mean of a message latency set varied depending on when the two instances of the software were executed relative to one another. In other words, with the same testing conditions, the mean message latency of messages sent by agent "A" and received by agent "B" would vary from a small value to one close to the loop period. This was found to be a result of the asynchronous nature of the communications. However, the mean of the means does not vary. Or in the previous example, the mean latency of both messages from agent "A" to agent "B" and from agent "B" to agent "A" does not vary. Additionally, the sum of means is consistent. For this reason the mean of means and the sum of means was used in measuring the message latency. The deviation from ideal is simply the sample period subtracted from the sum of means. This deviation would be zero if the transmission of data was instantaneous and for this reason gives good insight into the true message latency.

Stage 1 was conducted to note any relationship between the length of time of testing and the observed message latency. Results of this test are found in Table 5.1. Little

Table 5.1: Two Agent Message Latency With Variable Test Duration

Approximate Length of Test (sec)	Mean of Means (sec)	Sum of Means (sec)	Deviation from Ideal (sec)
57	0.529	1.059	0.059
343	0.556	1.113	0.113
700	0.522	1.045	0.045

variation in the mean of means or in the deviation from ideal is noted. For this reason the following four stages were conducted using a test length of sixty seconds.

Stage 2 investigated the relationship between the message latency and sampling frequency. All tests lasted for 60 seconds. Table 5.2 shows little variation in the deviation from ideal between the 1 Hz and 5 Hz tests but a decrease in mean of means. Then when the sample frequency is increased to 10 Hz, which is close to the maximum frequency of

Table 5.2: Two Agent Message Latency With Variable Sampling Frequency

Clark sample frequency (Hz)	Mean of Means (sec)	Sum of Means (sec)	Deviation from Ideal (sec)
1	0.529	1.059	0.059
2	0.279	0.558	0.058
5	0.123	0.246	0.046
10	0.149	0.298	0.198

13 Hz, more latency is observed. Therefore, this test shows that message latency decreases with an increasing sampling frequency. This intuitively makes sense because data is being sent and received more frequently. However, it was noted that deviation of message latency from ideal increases as a function of increasing sample frequency but only when the sample frequency approaches the maximum sample frequency of the hardware.

Stages 3-5 were conducted to determine if message latency is a function of the number of agents. Tables 5.3-5.5 show the results of these tests. All tests were conducted at a loop speed of 1 Hz and for approximately 60 seconds. It can be noted from these tests that all

Table 5.3: Two Agent Message Latency

Two agent test	Messages Sent	Mean (sec)	Standard Deviation (sec)	Median (sec)	Min (sec)	Max (sec)
Agent1 -> Agent2	343	0.982	0.006	0.980	1.000	0.969
Agent2 -> Agent1	346	0.131	0.060	0.293	3.049	0.049

data had a small standard deviation leading to the conclusion that the recorded message latencies were consistent and not changing with time. Table 5.6 shows the mean of the means, sum of the means, and deviation from ideal of the previous data shown in Tables 5.3-5.5. This data shows that there is no significant correlation between the number of

Table 5.4: Three Agent Message Latency

Three agent test	Messages Sent	Mean (sec)	Standard Deviation (sec)	Median (sec)	Min (sec)	Max (sec)
Agent1 -> Agent2	67	0.824	0.007	0.820	0.840	0.810
Agent1 -> Agent3	66	0.118	0.008	0.120	0.140	0.110
Agent2 -> Agent1	68	0.264	0.119	0.250	1.240	0.240
Agent2 -> Agent3	67	0.333	0.003	0.330	0.340	0.320
Agent3 -> Agent1	68	0.997	0.264	0.950	2.930	0.950
Agent3 -> Agent2	68	0.752	0.117	0.740	1.710	0.720

Table 5.5: Four Agent Message Latency

Three agent test	Messages Sent	Mean (sec)	Standard Deviation (sec)	Median (sec)	Min (sec)	Max (sec)
Agent1 -> Agent2	77	0.726	0.006	0.730	0.740	0.710
Agent1 -> Agent3	75	0.910	0.005	0.910	0.920	0.900
Agent1 -> Agent4	74	0.806	0.005	0.810	0.820	0.800
Agent2 -> Agent1	77	0.368	0.007	0.370	0.380	0.340
Agent2 -> Agent3	76	0.228	0.007	0.230	0.240	0.210
Agent2 -> Agent4	75	0.232	0.306	0.130	1.120	0.120
Agent3 -> Agent1	76	0.197	0.115	0.180	1.190	0.180
Agent3 -> Agent2	76	0.862	0.004	0.860	0.870	0.860
Agent3 -> Agent4	75	0.942	0.004	0.940	0.960	0.940
Agent4 -> Agent1	77	0.368	0.415	0.290	3.260	0.280
Agent4 -> Agent2	77	1.006	0.246	0.970	2.930	0.960
Agent4 -> Agent3	77	0.189	0.247	0.150	2.120	0.140

agents and message latency.

5.1.3 Experimentally Determined Throughput

Many have researched the throughput of XBee Pro modules and other wireless communications transceivers utilizing the IEEE 802.15.4 standard in aerial networks [48] [49]. This thesis will not attempt such characterization, but rather establish an experimental

Table 5.6: Message Latency With Variable Number of Agents

Number of Agents	Mean of Means (sec)	Sum of Means (sec)	Deviation from Ideal (sec)
2	0.556	1.113	0.113
3	0.547	1.094	0.094
4	0.569	1.138	0.138

baseline of data throughput using XBee Pro HP900 modules. Maximum data throughput was determined by using the throughput tool in Digi International's X-CTU software. The XBee radios were set to API mode without escapes with varying baud rates. The full XBee configuration XML file can be found in Appendix B. Each test was run in loop-back mode for 10 seconds. Each test was then repeated five times. Figure 5.1 shows that the best case throughput for the XBee radios in the given configuration is 33 Kbps at the highest baud rate of 115200.

5.1.4 Payload Modularity

A basic bench test was performed to verify the functionality of the payload interface classes. In all, three interface classes were developed and tested: a TCP/IP interface, a UDP/IP interface, and a ROS interface. The TCP/IP interface was used in the flight tests described in Subsections 5.3.2 and 5.3.3. The UDP/IP interface could be used in high throughput applications between the MC and payload. The ROS interface module acts as a ROS node essentially tying Clark to a ROS network. All three interfaces were tested by sending a formatted string of data between a testing payload script and the payload interface. The payload protocol could be changed by simply changing the import statement in the Clark class. Through this testing it was determined that all three payload interfaces function properly.

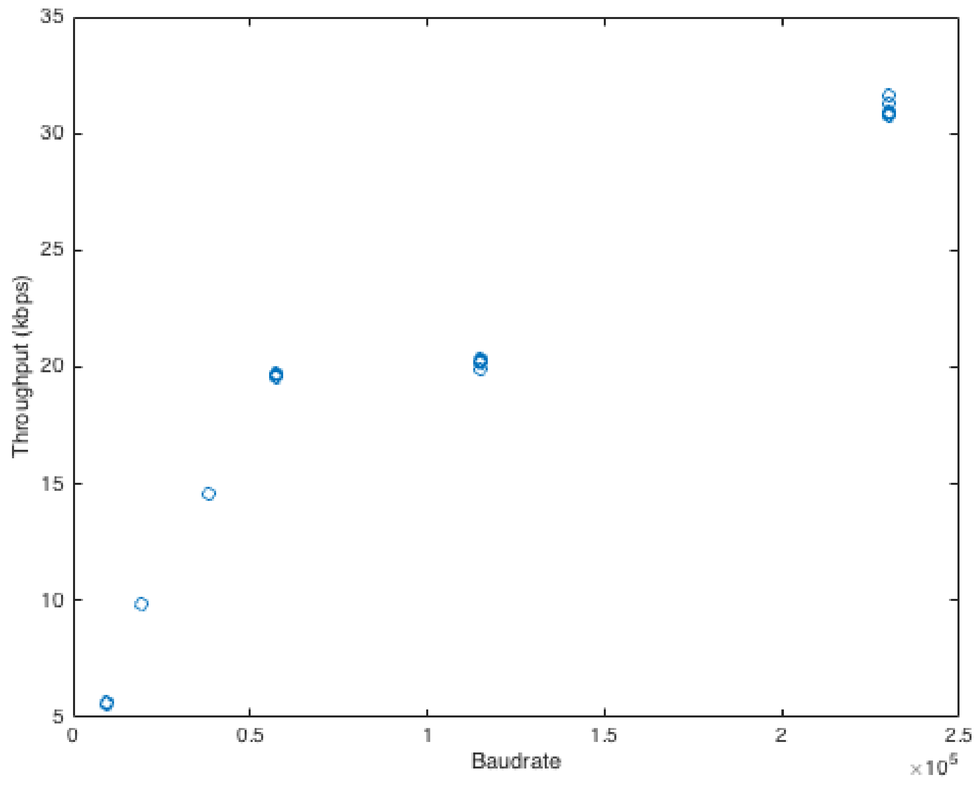


Figure 5.1: Experimentally determined throughput of XBee Pro 900HP Modules

5.1.5 Static GPS Error

For ground and flight testing the Pixhawk was integrated with a u-blox LEA-6h GPS module for measuring position. The LEA-6h GPS module features a horizontal accuracy of 2.5 meters when operated under 500 meters per second and below an absolute altitude of 50 kilometers. The update rate is 2 Hz. A static test of the unit demonstrated the accuracy of the LEA-6h GPS module. Figure 5.2 shows a plot of the recorded GPS position over the 7 minute and 30 seconds test. The average error was 0.62 meters with a standard deviation of 0.33 meters. The minimum and maximum errors were 0.00 and 1.62 meters respectively. This testing shows that the GPS has superior performance in a static situation

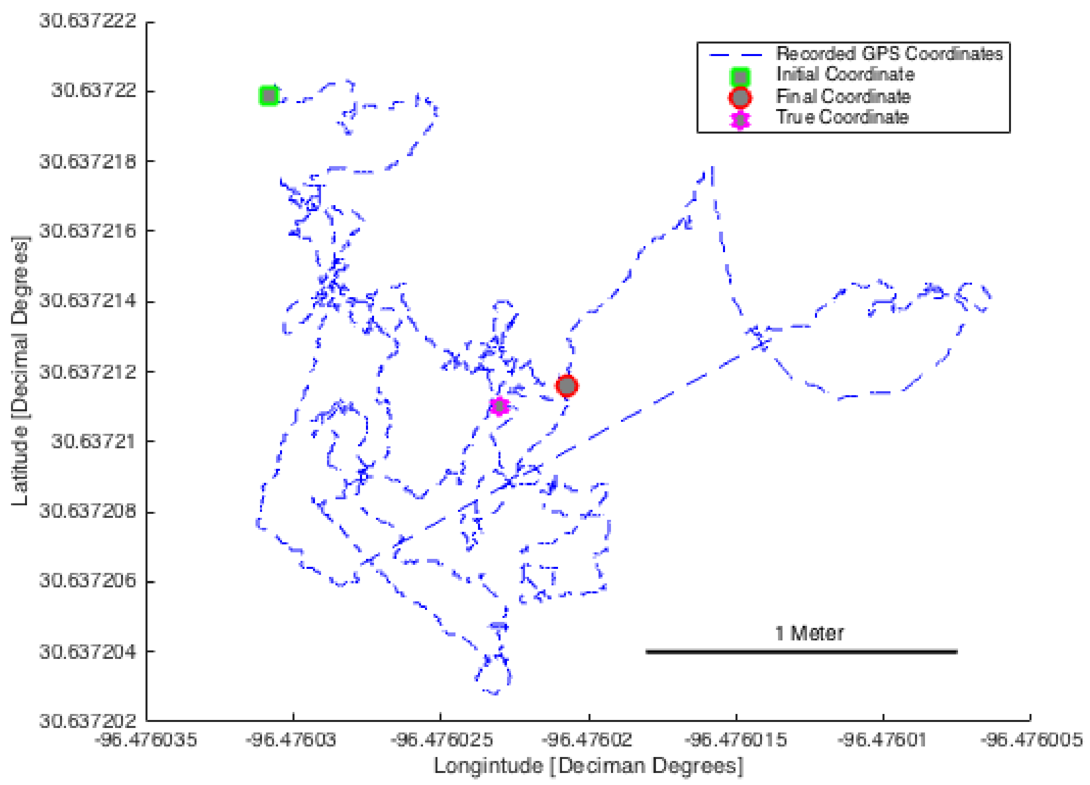


Figure 5.2: Experimentally determined static GPS error as recorded by the Pixhawk autopilot

and greater error will occur in any dynamic testing. The maximum GPS error is still within the bounds of many GPS waypoint driven applications, but may be too inaccurate for applications that require position measurements that have better than one meter accuracy.

5.2 Ground Test: Single Vehicle Mode Change

Ground tests were conducted using the Traxxas Slash vehicle to verify proper function of the software and controllability of the vehicle while Clark was running. An important test included verification of command and control of the vehicle through the autopilot interface. The test included setting a single waypoint on the autopilot and then switching the autopilot mode from manual to automatic. After the waypoint was reached, the vehicle would stop and the mission control station operator changed the mode back to manual. Once in manual mode, the vehicle was piloted to its starting position and the test was repeated. The mission layer was set to a sampling frequency of 0.5 Hz. Figure 5.3 shows

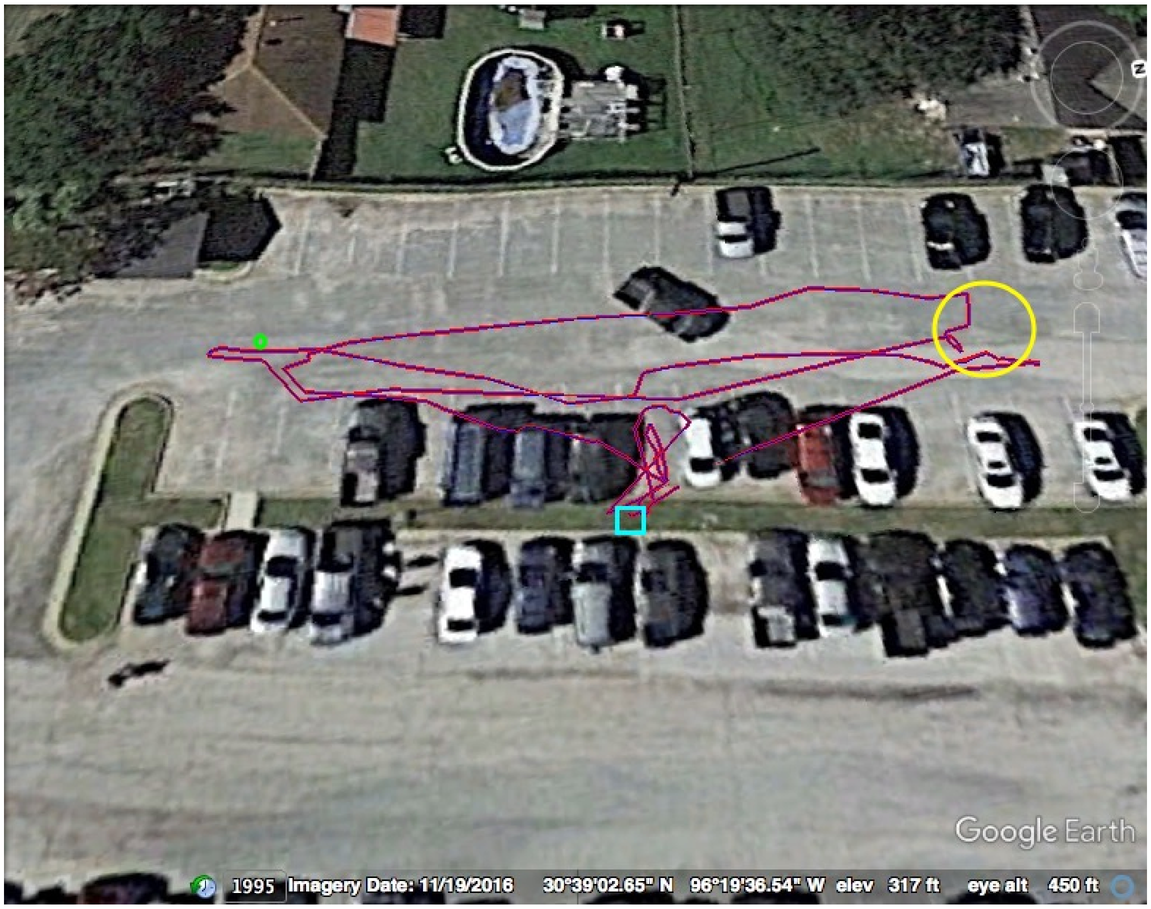

Google Earth
1995 Imagery Date: 11/19/2016 30°39'02.65" N 96°19'36.54" W elev 317 ft eye alt 450 ft

Figure 5.3: Vehicle path as recorded by the Slash vehicle (red) and MCS (blue)—the green, yellow, and cyan markers indicate the waypoint, starting position, and MCS location respectively

the path of the vehicle as recorded by both the vehicle (red) and the MCS (blue). The green marker shows the location of the the waypoint, while the yellow marker is the starting position. The cyan marker shows where the MCS was located during testing. It should be noted that 100% of Clark Link messages were received by the MCS and that 100% of the Clark Link messages commanding a change in mode were received by the Slash vehicle. This test demonstrated that GPS coordinates, altitude, velocity, heading, and autopilot mode can be transmitted through the framework from one agent to another. It also demonstrated that an agent could command the autopilot mode of another agent. These are two fundamental capabilities to being able to control multiple vehicles.

5.2.1 Cooperation Between Two Vehicles

A final ground test was conducted to demonstrate a cooperative mission between two vehicles. Two Slash vehicles were used for testing. The first vehicle (Slash A) was commanded on an eight-waypoint autonomous mission. The second vehicle (Slash B) was then commanded by a human operator to go to the position of the first Slash. It is important to note that the only action performed by the human operator was triggering Slash B to go to the location of Slash A. Many autonomous triggers could be devised to replace the human operator for this mission. The framework then autonomously relayed the GPS coordinates, created a waypoint, and commanded the Slash B into "Guided Mode." "Guided Mode" is an existing mode on the Pixhawk autopilot that will track a given waypoint or set of waypoints. In this case, only a single waypoint was commanded. Slash B then autonomously drove to the location of Slash A. The autopilots were configured with a waypoint tolerance of six meters. Figure 5.4 shows the paths of the two vehicles. This test highlights the ability of Clark to autonomously modify the state of a vehicle given input from another vehicle. The eight red markers indicate the waypoints for the autonomous mission for Slash A, and the green path is the path recorded by Slash A. The cyan marker was the autonomously generated waypoint, and the yellow path is the path of Slash B going to the waypoint.

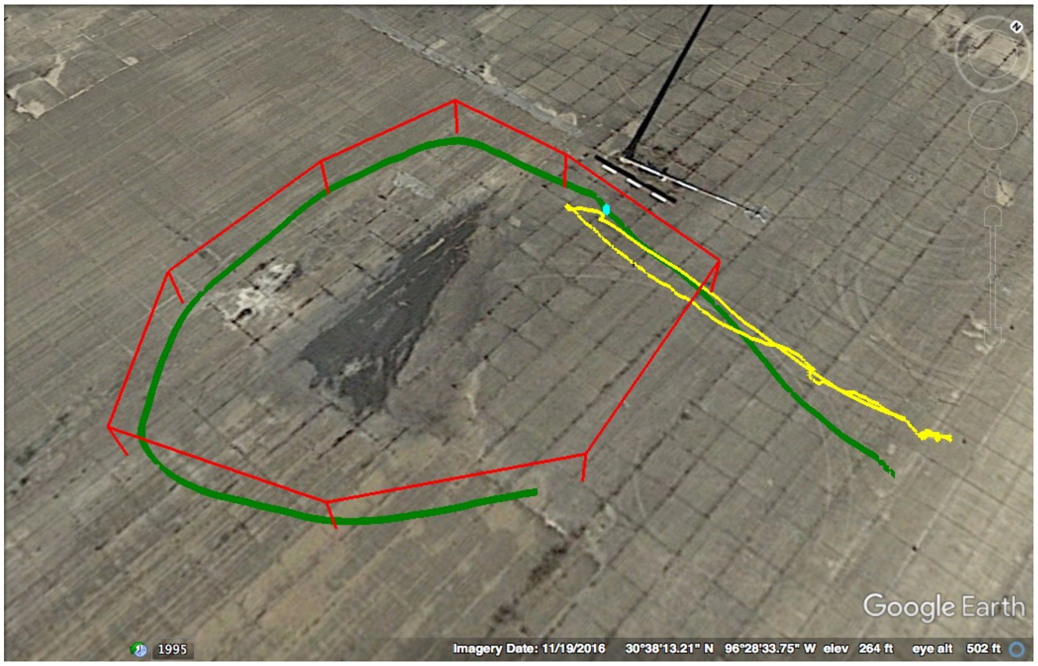

Google Earth
1995
Imagery Date: 11/19/2016 30°38'13.21" N 96°28'33.75" W elev 264 ft eye alt 502 ft

Figure 5.4: A cooperative mission showing the paths of Slash A (green) and Slash B (Yellow)—The eight mission waypoints (red) and the automatically generated waypoint (cyan) are shown

5.3 Flight Tests

All flight testing was conducted at the Texas A&M RELLIS campus located at site 83TX. For all tests listed in this section, data was transmitted using the heartbeat, agent status, and picture data Clark Link messages. The contents of these messages can be found in Table 3.4. Additionally, comparisons will be made with data recorded by the Pixhawk autopilot.

5.3.1 Two Vehicle Flight Test

With the completion of ground testing a single vehicle, a multi-agent system was then tested to verify the communication between multiple vehicles. In this test, the F450 quadrotor and the Slash rover were manually piloted. Both transmitted agent status mes-

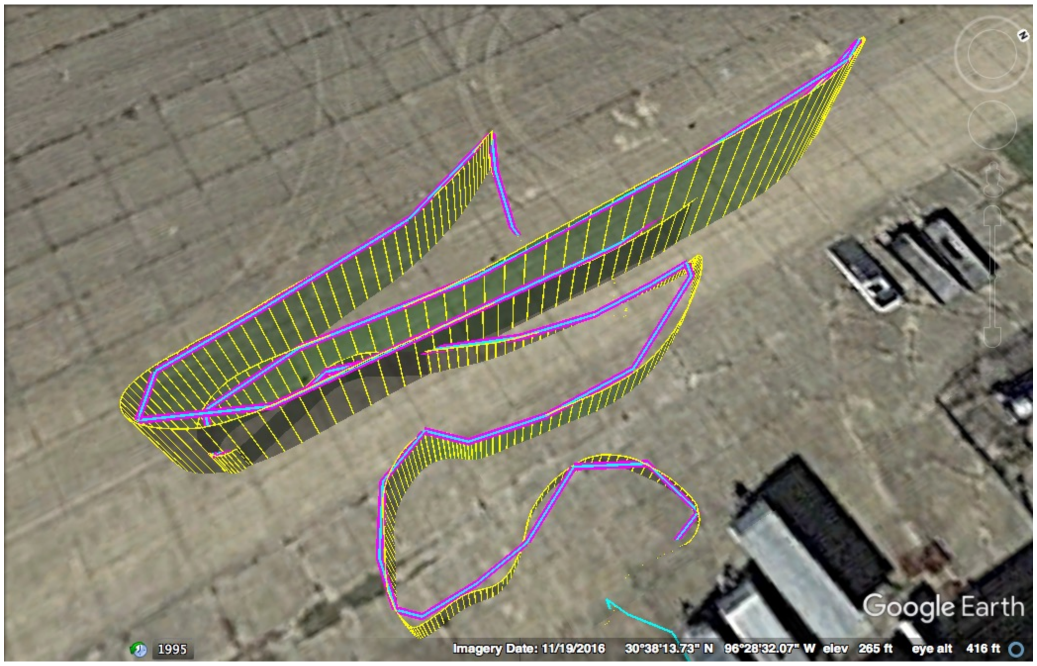

Google Earth
1995
Imagery Date: 11/19/2016 30°38'13.73" N 96°28'32.07" W elev 265 ft eye alt 416 ft

Figure 5.5: Flight path of the F450 as recorded by the autopilot (yellow), Slash (cyan), and F450 (magenta)

sages and recorded all received agent status messages. Clark mission was configured to operate with a sample frequency of 0.5 Hz. Figure 5.5 shows the flight path of the F450. The magenta line is the flight path of the F450 as recorded by the F450. The cyan line is the flight path of the F450 as recorded by the Slash. The yellow line is the flight path logged on the Pixhawk. Figure 5.6 displays the path of the Traxxas Slash. The same color scheme for Figure 5.5 applies to Figure 5.6.

The F450 flew for approximately 3 minutes and 52 seconds and the Slash drove for 7 minutes and 56 seconds. From Figure 5.5 it is observed the flight path of the F450 as recorded by the F450 matches the path as recorded by the Slash. This indicates that 100% of the agent status packets were successfully transmitted. Analysis of the data logs confirmed this finding. The successful transmission of data was expected in this test as the vehicles and MCS were never more than 150 feet apart. However, in Figure 5.6 it is

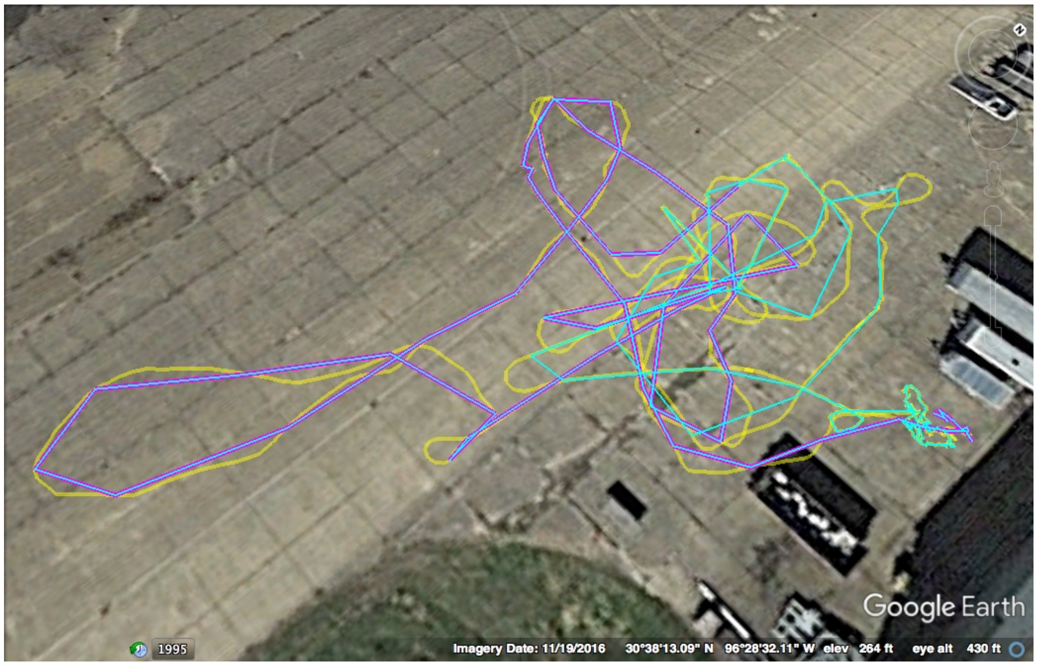

Google Earth
1995
Imagery Date: 11/19/2016 30°38'13.09" N 96°28'32.11" W elev 264 ft eye alt 430 ft

Figure 5.6: Vehicle path of the Slash as recorded by the autopilot (yellow), Slash (cyan), and F450 (magenta)

observed that the F450 stops tracking the Slash. An analysis of the logs showed that when the F450 landed, the Raspberry Pi reset. This in turn stopped the Clark software from tracking the position of the Slash. The MC aboard the F450 recorded 100% of the data transmitted by the Slash prior to being reset.

It should also be noted from these tests that the sampling frequency of 0.5 Hz is too slow and aliases the true position of the vehicles. This is shown in both figures by the mismatch between the Pixhawk's records (yellow line) and the Clark records (cyan and magenta lines). However, this test demonstrates that agent status messages can be transmitted between two vehicles in addition to being transmitted to the MCS.

5.3.2 Three Vehicle Flight Test With Payload

After confirming the function of the radio interface for communication and autopilot interface for commanding flight modes, it was necessary to verify the functionality of the payload interface. To do this, a test was conducted using the F450, Anaconda, and Slash vehicles. This test was an extension to the work presented by Rogers et al. [50]. The Anaconda was equipped with the Raspberry Pi with an attached camera connected over Ethernet to the MC. All three vehicles broadcast agent status messages. Additionally, the Anaconda transmitted a picture data message when an image was captured by the Raspberry Pi camera. The Clark mission class on the Anaconda was programmed so that every five seconds it would request an image capture from the payload. After the payload captured and saved an image it would report this back to the MC which would transmit a picture data message. All three vehicles were operated manually during this test. Figure 5.7 shows the flight paths of all three vehicles. The flight path shown of the Anaconda is only a portion of the full flight path and was truncated as to not obstruct the paths of the other vehicles. After analyzing the flight logs it was determined that 83 images were taken during the 6 minute and 57 second flight of the Anaconda. Due to high winds the F450 landed after only 49 seconds of flight. The Slash was operated for 7 minutes and 4 seconds. Table 5.7 shows the percent of agent status messages received by each agent including the MCS. In this test the MCS received 13% less agent status messages from the Slash as it did in the previous test (see Subsection 5.3.1). Additionally, the MCS only received 88% of the agent status messages from the Anaconda vehicle. This high message loss was ultimately attributed to poor placement of the MCS antenna and was rectified in future tests. It should be noted that the inter-agent communications performed much better with a maximum of 4% of agent status messages being dropped.

Analysis of the data logs showed that the MCS received 73 picture data messages,

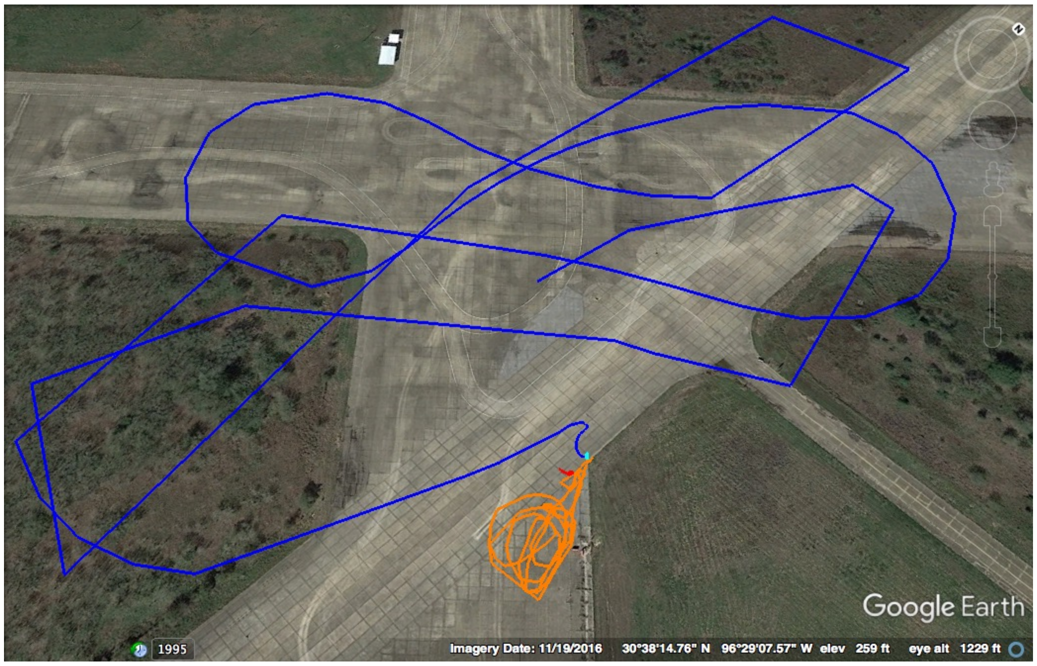

Google Earth
1995
Imagery Date: 11/19/2016 30°38'14.76" N 96°29'07.57" W elev 259 ft eye alt 1229 ft

Figure 5.7: Vehicle paths of the Anaconda (blue), F450 (red), and Slash (orange) with reference to the MCS (cyan)

or 88% of the total number of messages transmitted. This agrees with the percentage of Anaconda's agent status packets received by the MCS, as previously shown. Figure 5.8 shows the location of each picture data message. The yellow and green markers indicate messages that were received by the MCS. The red markers indicate messages that were dropped. The dropped picture data messages can also be attributed to the poor placement of the MCS's antenna. The image take at the green marker is shown in Figure 5.9.

The testing with three vehicles verified that the framework can communicate with a connected payload through the payload interface and then communicate that data with other agents within the system.

Table 5.7: Agent Status Message Communication

	MCS	F450	Anaconda	Slash
Total agent status packets transmitted	0	49	417	420
Agent status packets received from F450	49 (100%)	N/A	49 (100%)	49 (100%)
Agent status packets received from Anaconda	369 (88%)	48 (96%)	N/A	390 (94%)
Agent status packets received from Slash	366 (87%)	48 (96%)	408 (97%)	N/A

5.3.3 Auto-excitation for System Identification Flight Test

Final flight tests were conducted in collaboration with an effort to perform online system identification of SUAS. These tests provided an opportunity to implement a control algorithm in the Clark mission abstraction layer that sent control signals directly to the ailerons, elevator, and rudder. The Super Cub (Cub) vehicle was selected by those performing the system identification testing.

The first flight test was simply to verify the integration of the framework on the Cub. Only eight hours were spent developing the control algorithm, integrating the hardware and avionics, and bench testing the system. Figure 5.10 shows the flight paths of two flights. The yellow path indicates the flight path as recorded on the Pixhawk. The blue and cyan paths are the flight paths as recorded by the MCS on the first and second flights respectively. Analysis showed that 100% of the agent status messages were successfully transmitted except for 19 messages that were dropped at the end of the second flight when the MCS antenna was laid down while the pilot landed the aircraft. One key result of this test is that it was a result of only eight hours of work and then performed excellently in its maiden flight test. This demonstrates the ease of use of the system.

For the second and third flight tests, the control algorithm responsible for command-

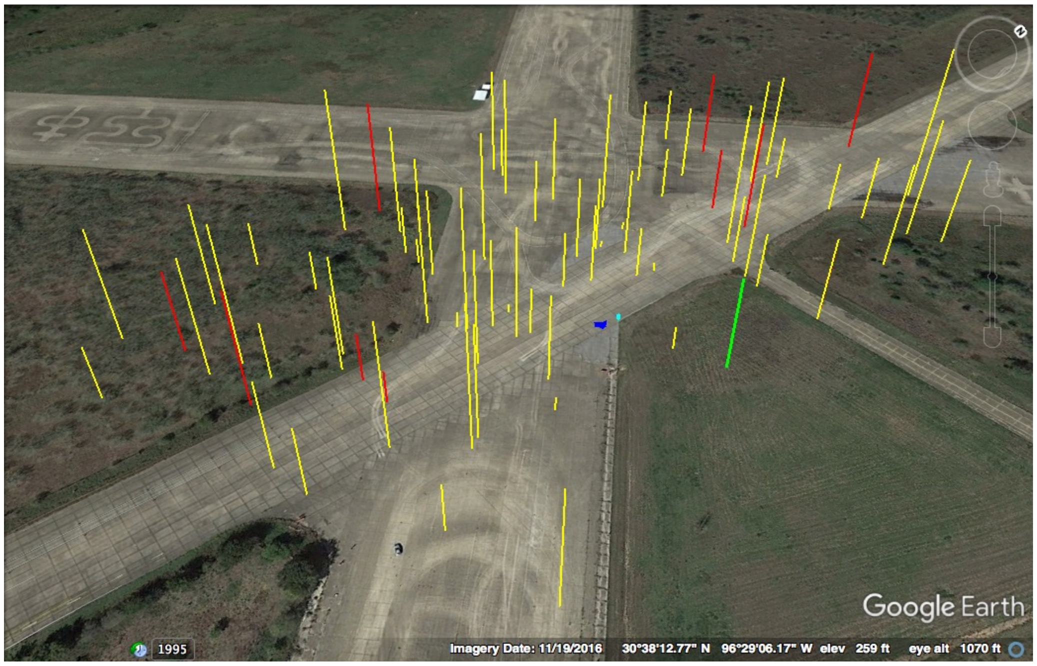

Google Earth
1995
Imagery Date: 11/19/2016 30°38'12.77" N 96°29'06.17" W elev 259 ft eye alt 1070 ft

Figure 5.8: Image capture locations are represented by markers indicating successful communication (yellow) and failed communication (red) of picture data messages—the green marker indicates the location of image seen in Figure 5.9; reprinted from [50]

ing precise inputs was used. The control algorithm commanded doublets, a pair of control inputs equal in amplitude and width but opposite in direction, to the ailerons and rudder. First an aileron doublet was commanded, followed by a rudder doublet. For the first test the algorithm waited to receive an activate controller Clark Link message from the MCS to initiate the doublet. This message was sent using the GUI. For the later test, the algorithm monitored the failsafe multiplexer RC channel and used it as the trigger for the doublet input. Figures 5.11-5.12 show the flight paths of the Cub. The yellow path was recorded by the autopilot. The green paths indicate when the control algorithm is active and controlling the vehicle. The blue path in Figure 5.11 shows the first flight path recorded by the MCS and the red path in Figure 5.12 shows the second flight path recorded by the MCS. With proper placement of the antenna, 100% of agent status messages were transmitted suc-

Figure 5.9: Sample image taken during the three vehicle flight test; reprinted from [50]

cessfully. Figure 5.13 shows a recorded doublet sequence and the accuracy with which it can be recorded. Pilots can be trained to provide adequate inputs for system identification, however it should be noted that the automatically generated inputs provide repeatability. Figure 5.14 demonstrates a frequency-varying sinusoidal input by both a human pilot and the framework. It can been seen that a human pilot is not able to maintain a constantly changing frequency in the sinusoid whereas the framework can provide a very precise frequency-varying sinusoid.

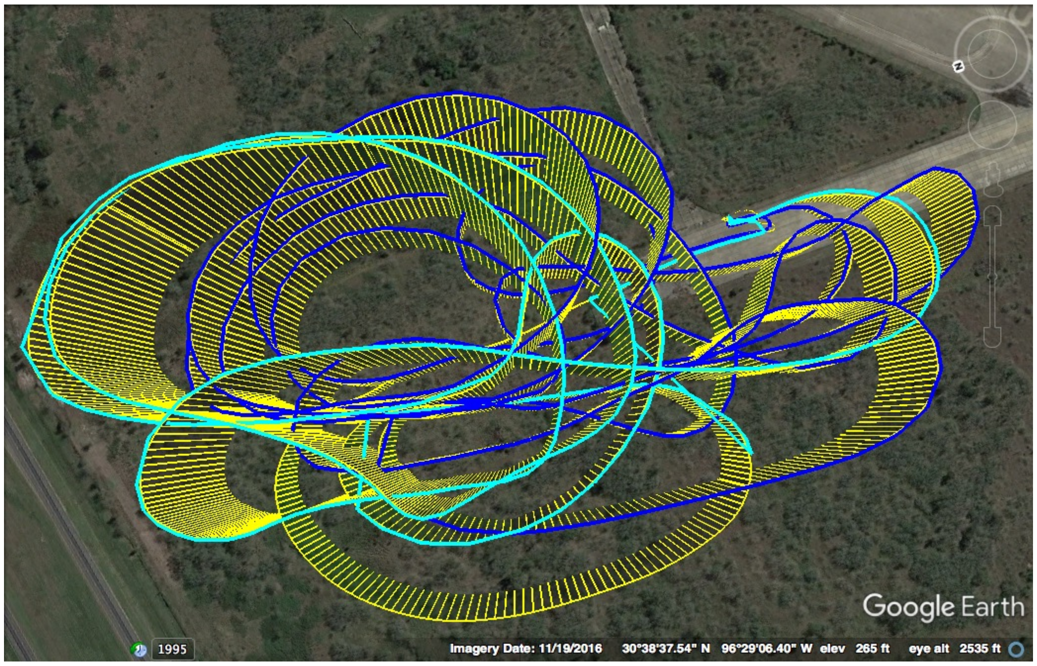

Google Earth
1995
Imagery Date: 11/19/2016 30°38'37.54" N 96°29'06.40" W elev 265 ft eye alt 2535 ft

Figure 5.10: System identification test flight with flight paths recorded by the autopilot (yellow) and MC (blue is first flight and cyan is second flight)

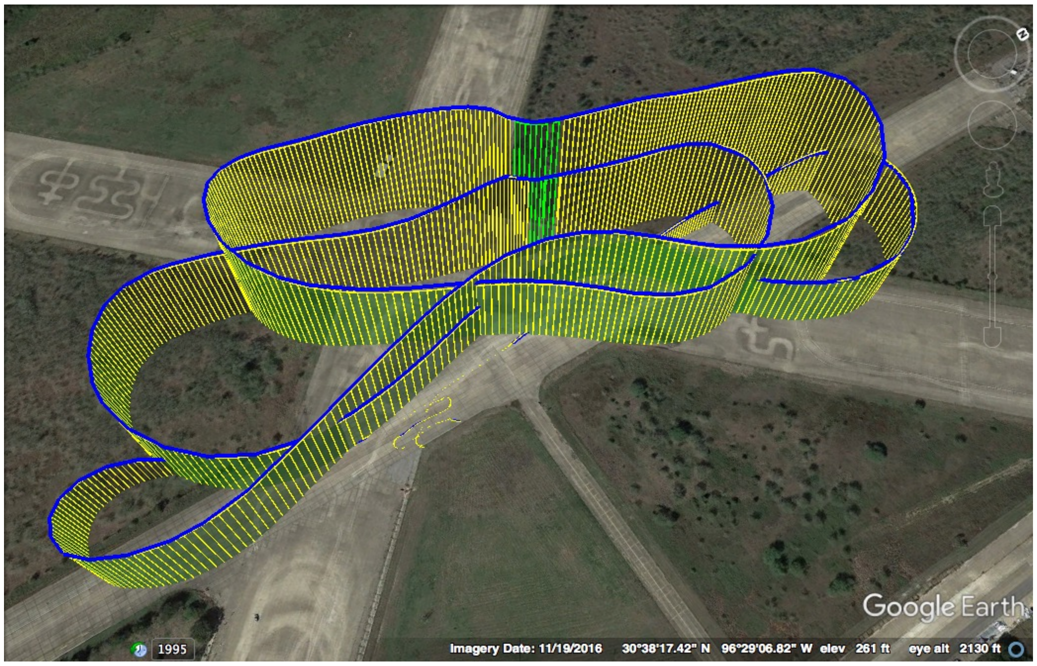

Google Earth
1995
Imagery Date: 11/19/2016 30°38'17.42" N 96°29'06.82" W elev 261 ft eye alt 2130 ft

Figure 5.11: System identification test flight with flight paths recorded by the autopilot (yellow) and MC (blue) with auto-excitation (green)

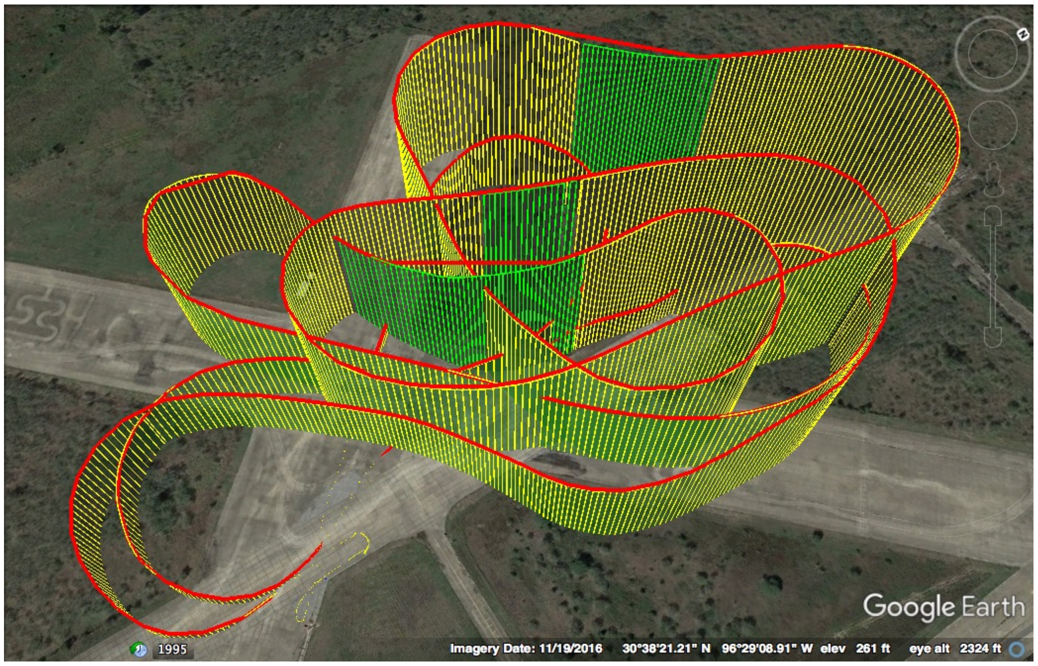

Google Earth
1995
Imagery Date: 11/19/2016 30°38'21.21" N 96°29'08.91" W elev 261 ft eye alt 2324 ft

Figure 5.12: System identification test flight with flight paths recorded by the autopilot (yellow) and MC (red) with auto-excitation (green)

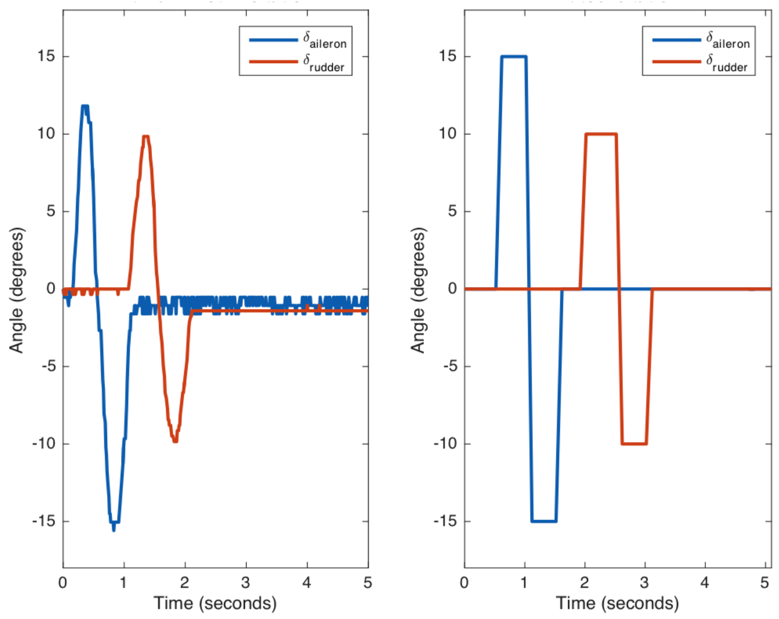


Figure 5.13: Comparison of doublets commanded by a pilot (left) and by Clark (right)

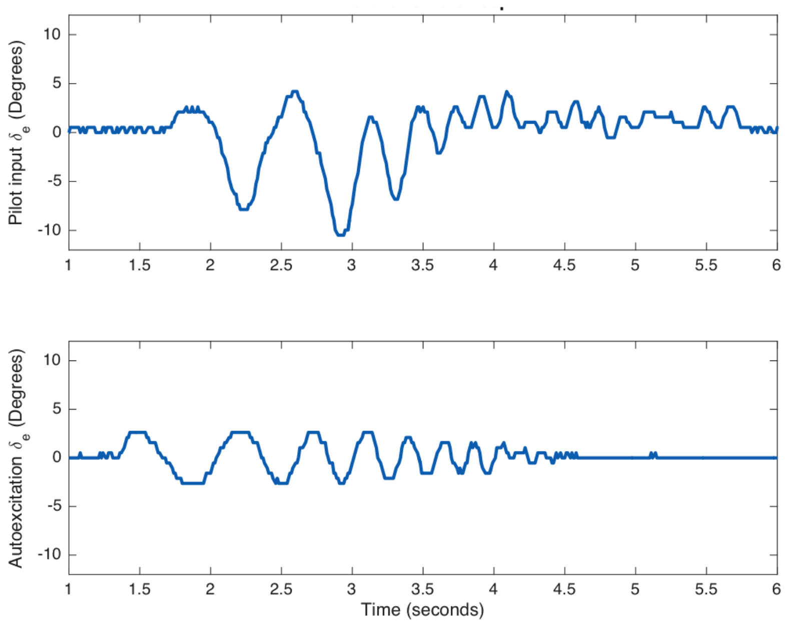


Figure 5.14: Comparison of sine sweeps commanded by a pilot (top) and by Clark (bottom)

6. CONCLUSIONS

Based on the development and results presented in the previous chapters, the following conclusions are drawn:

- The Clark framework was shown to provide a user with the ability to access state information from an autopilot or payload and transmit this information to any agent on the network. The framework's design supports many protocols for connecting payloads and communicating information. This information can then be used by control algorithms to affect control. With this ability to communicate and control, the Clark framework is a multi-agent control framework with integrated payloads.

- The design of the mission abstraction layer allows users to easily integrate various control algorithms into the Clark framework. These algorithms can then be swapped, as shown with the system identification test, without changing the underlying communications, and continue to function. Therefore, Clark provides modular control implementation.

- The Clark framework is able to communicate data between four agents and can be operated with a sampling frequency of up to 13 Hz when executed on a Raspberry Pi 2. Message latency can be as short as 150 milliseconds but is dependent on the sampling frequency.

- Through the use of Systems Engineering and Spiral Development a set of requirements was developed that meets the needs of the Vehicle Systems & Control Laboratory for quickly testing multi-agent control algorithms on an easy to use platform. The Clark framework developed meets all requirements generated by this process.

7. RECOMMENDATIONS

Based on the testing and results presented in Chapter 5, the author makes the following recommendations:

- The Clark framework was heavily influenced by the second defined requirement that imposed a constraint of ease of use. Design choices such as utilizing a single process could be altered to enhance the performance of the Clark framework. Therefore, Clark could be extended by developing hardware interface classes that are executed asynchronously in a multi-processing scheme. This would enable transmission and reception of messages independent of the “execute_mission” loop and could increase sampling frequency performance.

- The Clark framework currently transmits every Clark Link message in a separate packet, which is not optimal. Messages transmitted to the same address could be bundled into a single packet to decrease the number of packets transmitted each loop cycle. The Clark agent class could be modified with a method to parse messages from incoming packets before parsing the message itself. This should decrease the total number of packets sent and increase communication performance.

- The radio interface class requires the transceiver be configured in API mode, which transmits messages with API frames that add overhead, but with the added benefit of addressing. Instead, the addressing could be handled with the Clark agent class by adding a destination field to all Clark Link messages. The XBee radios could be reconfigured to transparent mode to increase the bandwidth.

- The Clark Framework supports software-in-the-loop testing as provided by DroneKit. It allows only for a single vehicle to be simulated per instance. While multiple instances can be run in parallel they are independent and do not share information, making simulation of multi-agent controllers difficult. Integration with multi-agent simulation software could allow for researchers to utilize Clark in both simulation and flight testing. MASPLANES [18] could be a possible open source software package for simulating multi-agent controllers while utilizing the Clark framework.

- The GUI developed for this thesis was designed to verify the functionality of the framework and does not include modular segments for specifying displays or data that may be desirable during testing of a particular multi-agent controller. Therefore, a GUI specifically designed for the testing of multi-agent control algorithms could be developed that gives researchers real-time feedback beyond simple state measurements.

- An investigation can be made into multiple radio links including long-range, short-range, and LTE connections. As stated, this thesis does not attempt to answer some questions necessary for commercial use, but this framework could support such investigations. High-bandwidth communications and the economics of such links would be an area that could be further explored.

- The Clark framework is designed with multi-agent control at the core of the requirements, however payload-directed flight algorithms can be tested through the hardware interfaces for payloads and the autopilot. Additionally, this framework can provide a telemetry link for monitoring such algorithms.

- A multiple tier computational intelligence based decision support tool could be investigated using the framework put forward in this thesis. The Clark mission class provides a clear location where such algorithms can be implemented.

REFERENCES

[1] F. A. Administration, “Unmanned aircraft systems,” 2016.

[2] P. Van Blyenburgh, “Uavs: an overview,” *Air & Space Europe*, vol. 1, no. 5, pp. 43–47, 1999.

[3] U. Ozdemir, Y. O. Aktas, A. Vuruskan, Y. Dereli, A. F. Tarhan, K. Demirbag, A. Erdem, G. D. Kalaycioglu, I. Ozkol, and G. Inalhan, “Design of a commercial hybrid vtol uav system,” *Journal of Intelligent & Robotic Systems*, vol. 74, no. 1-2, pp. 371–393, 2014.

[4] J. V. Henrickson, C. Rogers, H.-H. Lu, J. Valasek, and Y. Shi, “Infrastructure assessment with small unmanned aircraft systems,” in *Unmanned Aircraft Systems (ICUAS), 2016 International Conference on*, pp. 933–942, IEEE, 2016.

[5] K. Kanistras, G. Martins, M. J. Rutherford, and K. P. Valavanis, “Survey of unmanned aerial vehicles (uavs) for traffic monitoring,” in *Handbook of Unmanned Aerial Vehicles*, pp. 2643–2666, Springer, 2015.

[6] R. Murphy, J. Dufek, T. Sarmiento, G. Wilde, X. Xiao, J. Braun, L. Mullen, R. Smith, S. Allred, J. Adams, *et al.*, “Two case studies and gaps analysis of flood assessment for emergency management with small unmanned aerial systems,” in *Safety, Security, and Rescue Robotics (SSRR), 2016 IEEE International Symposium on*, pp. 54–61, IEEE, 2016.

[7] A. N. Chaves, P. S. Cugnasca, and J. Jose, “Adaptive search control applied to search and rescue operations using unmanned aerial vehicles (uavs),” *IEEE Latin America Transactions*, vol. 12, no. 7, pp. 1278–1283, 2014.

[8] Y. Shi, J. A. Thomasson, S. C. Murray, N. A. Pugh, W. L. Rooney, S. Shafian, N. Rajan, G. Rouze, C. L. Morgan, H. L. Neely, *et al.*, "Unmanned aerial vehicles for high-throughput phenotyping and agronomic research," *PloS one*, vol. 11, no. 7, p. e0159781, 2016.

[9] A. Rango, A. Laliberte, J. E. Herrick, C. Winters, K. Havstad, C. Steele, D. Browning, *et al.*, "Unmanned aerial vehicle-based remote sensing for rangeland assessment, monitoring, and management," *Journal of Applied Remote Sensing*, vol. 3, no. 1, p. 033542, 2009.

[10] J. S. McCarley and C. D. Wickens, *Human factors implications of UAVs in the national airspace*. University of Illinois at Urbana-Champaign, Aviation Human Factors Division, 2005.

[11] R. C. Nelson, *Flight stability and automatic control*, vol. 2. WCB/McGraw Hill New York, 1998.

[12] E. Pereira, P. M. Da Silva, C. Krainer, C. M. Kirsch, J. Morgado, and R. Sengupta, "A networked robotic system and its use in an oil spill monitoring exercise," 2013.

[13] M. A. Kovacina, D. Palmer, G. Yang, and R. Vaidyanathan, "Multi-agent control algorithms for chemical cloud detection and mapping using unmanned air vehicles," in *Intelligent Robots and Systems, 2002. IEEE/RSJ International Conference on*, vol. 3, pp. 2782–2788, IEEE, 2002.

[14] M. Quaritsch, R. Kuschnig, H. Hellwagner, B. Rinner, A. Adria, and U. Klagenfurt, "Fast aerial image acquisition and mosaicking for emergency response operations by collaborative uavs," in *Proceedings for the International ISCRAM Conference*, pp. 1–5, 2011.

[15] H. Ergezer and K. Leblebiciolu, “3d path planning for multiple uavs for maximum information collection,” *Journal of Intelligent & Robotic Systems*, vol. 73, no. 1-4, p. 737, 2014.

[16] R. W. Beard and T. W. McLain, “Multiple uav cooperative search under collision avoidance and limited range communication constraints,” in *Decision and Control, 2003. Proceedings. 42nd IEEE Conference on*, vol. 1, pp. 25–30, IEEE, 2003.

[17] H. Kurdi, J. How, and G. Bautista, “Bio-inspired algorithm for task allocation in multi-uav search and rescue missions,” in *AIAA Guidance, Navigation, and Control Conference*, p. 1377, 2016.

[18] M. Pujol-Gonzalez, “Masplanes simulator,” 2016.

[19] J. Curtis and R. Murphey, “Simultaneous area search and task assignment for a team of cooperative agents,” in *AIAA Guidance, Navigation, and Control Conference and Exhibit*, p. 5584, 2003.

[20] T. Shima, S. J. Rasmussen, and P. Chandler, “Uav team decision and control using efficient collaborative estimation,” *Journal of Dynamic Systems, Measurement, and Control*, vol. 129, no. 5, pp. 609–619, 2007.

[21] R. Olfati-Saber and R. M. Murray, “Consensus problems in networks of agents with switching topology and time-delays,” *IEEE Transactions on automatic control*, vol. 49, no. 9, pp. 1520–1533, 2004.

[22] H.-L. Choi, L. Brunet, and J. P. How, “Consensus-based decentralized auctions for robust task allocation,” *IEEE transactions on robotics*, vol. 25, no. 4, pp. 912–926, 2009.

[23] H. Jonathan, “Raven: Testbed for autonomous uavs,” 2007.

[24] G. Hoffmann, D. G. Rajnarayan, S. L. Waslander, D. Dostal, J. S. Jang, and C. J. Tomlin, “The stanford testbed of autonomous rotorcraft for multi agent control (starmac),” in *Digital Avionics Systems Conference, 2004. DASC 04. The 23rd*, vol. 2, pp. 12–E, IEEE, 2004.

[25] S. Lupashin, M. Hehn, M. W. Mueller, A. P. Schoellig, M. Sherback, and R. DâĂŹAndrea, “A platform for aerial robotics research and demonstration: The flying machine arena,” *Mechatronics*, vol. 24, no. 1, pp. 41–54, 2014.

[26] N. Michael, D. Mellinger, Q. Lindsey, and V. Kumar, “The grasp multiple micro-uav testbed,” *IEEE Robotics & Automation Magazine*, vol. 17, no. 3, pp. 56–65, 2010.

[27] G. Baliga, S. Graham, L. Sha, and P. Kumar, “Etherware: Domainware for wireless control networks,” in *Object-Oriented Real-Time Distributed Computing, 2004. Proceedings. Seventh IEEE International Symposium on*, pp. 155–162, IEEE, 2004.

[28] M. Valenti, B. Bethke, J. P. How, D. P. de Farias, and J. Vian, “Embedding health management into mission tasking for uav teams,” in *American control conference*, pp. 5777–5783, 2007.

[29] P. Bouffard, “starmac-ros-pkg,” 2012.

[30] H. Huang, G. M. Hoffmann, S. L. Waslander, and C. J. Tomlin, “Aerodynamics and control of autonomous quadrotor helicopters in aggressive maneuvering,” in *Robotics and Automation, 2009. ICRA’09. IEEE International Conference on*, pp. 3277–3282, IEEE, 2009.

[31] S. Lupashin, A. Schöllig, M. Hehn, and R. D’Andrea, “The flying machine arena as of 2010,” in *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pp. 2970–2971, IEEE, 2011.

[32] H. C. Christmann, “Self-configuring ad-hoc networks for unmanned aerial systems,” 2008.

[33] M.-S.-B. J. O. W. Group, “Mil-std-499b,” tech. rep., Andrews AFB, Washington, D.C. 20331, 1993.

[34] B. Boehm and W. J. Hansen, “Spiral development: Experience, principles, and refinements,” tech. rep., DTIC Document, 2000.

[35] U. DoD, “Systems engineering fundamentals,” *DoD, Department of Defense (USA)*, 2001.

[36] M.-S.-G. W. Group, “Environmental engineering considerations and laboratory tests,” tech. rep., 314 Longs Corner Road, Aberdeen Proving Ground, MD 21005-5055, 2008.

[37] F. A. Administration, “Summary of small unmanned aircraft rule (part 107),” 2016.

[38] P. Guo, “Python is now the most popular introductory teaching language at top u.s. universities,” 2014.

[39] D. R. Inc, “Dronekit by 3d robotics: Introducing to dronekit-python,” 2015.

[40] K. Varda, “Protocol buffers: Googles data interchange format,” *Google Open Source Blog, Available at least as early as Jul*, 2008.

[41] L. Meier, J. Camacho, B. Godbolt, J. Goppert, L. Heng, M. Lizarraga, *et al.*, “Mavlink: Micro air vehicle communication protocol,” *Online]. Tillgänglig: http://qgroundcontrol. org/mavlink/start.[Hämtad 2014-05-22]*, 2013.

[42] H. D. Mathias, “An autonomous drone platform for student research projects,” *Journal of Computing Sciences in Colleges*, vol. 31, no. 5, pp. 12–20, 2016.

[43] W. Harrington, *Learning Raspbian*. Packt Publishing Ltd, 2015.

[44] A. F. Molisch, K. Balakrishnan, C.-C. Chong, S. Emami, A. Fort, J. Karedal, J. Kunisch, H. Schantz, U. Schuster, and K. Siwiak, “Ieee 802.15. 4a channel model-final report,” *IEEE P802*, vol. 15, no. 04, p. 0662, 2004.

[45] A. Haerter, “Pixhawk autopilot: Key features,” 2016.

[46] A. D. Team, “Ardupilot autopilot suite,” 2016.

[47] R. P. Foundation, “Camera module,” 2016.

[48] M. Asadpour, D. Giustiniano, K. A. Hummel, and S. Heimlicher, “Characterizing 802.11 n aerial communication,” in *Proceedings of the second ACM MobiHoc workshop on Airborne networks and communications*, pp. 7–12, ACM, 2013.

[49] T. Andre, K. A. Hummel, A. P. Schoellig, E. Yanmaz, M. Asadpour, C. Bettstetter, P. Grippa, H. Hellwagner, S. Sand, and S. Zhang, “Application-driven design of aerial communication networks,” *IEEE Communications Magazine*, vol. 52, no. 5, pp. 129–137, 2014.

[50] C. T. Rogers, , C. Noren, and J. Valasek, “Heterogeneous multi-vehicle modular control framework with payload integration,” in *Unmanned Aircraft Systems (ICUAS), 2017 International Conference on*, IEEE, 2017.

APPENDIX A

SYSTEMS ENGINEERING DOCUMENTATION

A.1 Needs Statement

The VSCL is considering a system that would connect multiple UAS and other vehicles together and allow them to be controlled for testing UAS control algorithms. In addition to connecting multiple vehicles, the system needs to integrate sensors, avionics, and payloads that will be used for controlling the vehicles. The system would need to operate outdoors when conditions are ideal for testing. The systems need to be easy to use and quick to change for a variety of laboratory testing needs.

A.1.1 Highlighted Needs

- Outdoors
- Quick and easy to use
- Connect multiple vehicles
- Connect sensors, payloads, and avionics

A.2 Hypothetical Example

A hypothetical example of a series of tests that the system would support could be as follows: The first test is to implement an RL cooperation algorithm to image an area using an Anaconda aircraft with a multi-spectral camera and direct another vehicle to areas of interest. A second test is implementing a formation-flying algorithm using the same Anaconda vehicle. The algorithm requires sonar in a tight formation to simulate a refueling operation. The need is to have a system that can support both without months of development and testing on independent systems.

A.3 CONOPS

Clark shall to be an open architecture where hardware changes can be made easily and quickly with little to no software development. Clark should be accessible to VSCL students current and future. Students should be able to understand the system architecture and how to modify quickly. Clark shall be easy to modify or swap the controller and/or number of agents used in testing. As a research tool, Clark shall providing data logs, debugging information, test supervision, and failsafes to the mission control station operator. Clark will be capable of operating four vehicles with up to five payloads by a single mission control operator. For safety, pilots will oversee each vehicle and if necessary retake control.

A.3.1 Physical Description

The proposed system, Clark will be a software package and testing suite with hardware recommendations.

A.4 Requirement Tree

1. Defined Reuquirement

    1.1. Derived Requirement

        1.1.1. Design Requirement

1. Clark shall be used in an outdoor environment excluding extreme weather

    1.1. Clark shall to be used in temperature ranges outside of [0, 70] deg C

    1.2. Clark shall not be used in wet weather such as rain or snow

    1.3. The wireless network shall extend to the limit of line-of-sight flight

        1.3.1. The wireless network shall have a range greater than 1 mile

2. Clark shall be easy to use by both developers and operators

    2.1. Clark shall be programmed in a language which can be quickly learned by Aerospace engineering students

        2.1.1. Clark shall be programmed using the Python language

        2.1.2. Clark shall be executed as a single process

    2.2. Clark shall be accompanied by extensive software and operations documentation

        2.2.1. Docstrings shall be used to manage code documentation

        2.2.2. A *what if* document should be included to demonstrate changes that can be made and how to do that.

        2.2.3. Procedures shall be documented for general use

        2.2.4. General troubleshooting practices should be provided

    2.3. Clark should provide testing tools for individual modules to verify functionality before integration with the system.

        2.3.1. A testing module for the radio interface shall be provided

        2.3.2. A testing module for the autopilot interface shall be provided

        2.3.3. A testing module for the payload interface shall be provided

    2.4. Clark shall provide data logs for use in debugging, controller analysis, and general system performance.

        2.4.1. There shall be a lower level log consisting of interface information

        2.4.2. There shall be a high level log consisting of mission level information

        2.4.3. Logs should be able to be set to various logging levels (DEBUG, INFO, WARNING, ERROR, and CRITICAL)

2.4.4. Logs should contain the time, module, and function as a preface to the log message

3. Clark shall connect multiple vehicles (agents) over a wireless network

 3.1. The wireless network shall be distributed

  3.1.1. The wireless network shall be a mesh network

 3.2. Clark shall be able to communicate with a radio transceiver both sending commands and retrieving network data

  3.2.1. The radio interface shall have a function to check for data in waiting on the network

  3.2.2. The radio interface shall have a function to retrieve data in waiting

  3.2.3. The radio interface shall have a function to send data from a buffer to a specific destination(s)

  3.2.4. The radio interface shall have a function to retrieve the connected transceivers network address

  3.2.5. The radio interface shall have a function to change parameters of the radio transceiver

  3.2.6. The radio interface shall have a function to verify that the connection is functioning properly

  3.2.7. The radio interface shall have a function to close its connection with the radio transceiver

4. Clark shall connect multiple sensors, payloads, and/or avionics aboard a single agent over a wired network

 4.1. The wired network shall use a consistent protocol for all payloads connected

4.2. The wired network shall use MAVLink over serial as the protocol to connect to the autopilot

4.2.1. DroneKit and Pymavlink should be used for interfacing with the autopilot

4.3. Clark shall connect to the autopilot through a wired connection

4.3.1. Clark shall connect to the autopilot through an autopilot interface

4.3.2. The autopilot interface shall have a function for retrieving state information from the autopilot

4.3.3. The autopilot interface shall have a function for commanding the autopilot

4.3.4. The autopilot interface shall have a function for verifying the connection with the autopilot and that it is functioning properly

4.3.5. The autopilot interface shall provide Clark with GPS time

4.3.6. The autopilot interface shall have a function for closing the connection with the autopilot

4.4. Clark shall connect to the payload or payload network through a wired connection

4.4.1. Clark shall connect to the autopilot through an payload interface

4.4.2. The payload interface shall have a function for receiving data from the payload(s)

4.4.3. The payload interface shall have a function for sending data to the payload(s)

4.4.4. The payload interface shall have a function for closing the connection with the payload(s)

5. Clark shall provide a manned mission control station with tools to observe and control all agents and payloads on the same network

5.1. The second human operator shall be for backup

5.2. Clark shall provide a basic graphical user interface for mission supervision

5.2.1. The GUI shall provide each agents state data (lat, lon, alt, vel, mode)

5.2.2. The GUI should provide time since last heartbeat

5.2.3. The GUI shall provide the ability to activate a multi-agent controller

5.2.4. The GUI should provide the ability to set individual agents to different modes

5.2.5. The GUI shall be able to terminate the mission

6. Clark shall be an open architecture

6.1. Clark shall consist of multiple interchangeable modules that provide an API set (specified by requirements of this document) to a central module

6.1.1. Clark shall be the central module to which all interfaces connect

6.1.2. Clark shall communicate with the user interface through a UDP connection

6.2. Clark shall utilize a open source UNIX based operating system

6.2.1. Clark shall employ a UNIX based mission computer

6.2.2. The mission computer shall have the necessary hardware ports to connect to the autopilot, radio transceiver, and payload

6.3. Clark shall not use machine specific libraries or function sets

7. Clark shall provide a modular multi-agent control architecture

7.1. Clark shall be able to communicate with the autopilot both sending commands and retrieving state data

7.1.1. The autopilot interface shall have a function to send commands with specified parameters

7.1.2. The autopilot interface shall have a function to retrieve autopilot state data (GPS time, latitude, longitude, altitude, velocity, heading, and mode)

7.1.3. The autopilot interface shall have a function to get the GPS time

7.1.4. The autopilot interface should be able to set autopilot parameters

7.1.5. The autopilot interface should be able to request an autopilot state not contained in 7.1.2

7.2. Clark shall be able to store information of several agents on each agent

7.2.1. The agent interface shall be able to unpack all clarklink messages

7.2.2. The agent interface shall be able to raise new data flags

7.2.3. The agent interface shall be able to pack all clarklink messages

7.2.4. The agent interface should be able to store all data for all agents on the network

7.2.5. The agent interface shall use clarklink

7.2.6. Clark shall provide a function to send data to the autopilot, radio, or payload from a buffer.

7.2.7. Clark shall provide a function to send data to the autopilot, radio, or payload directly.

7.2.8. Clark shall provide a function to read data to the autopilot, radio, or payload.

7.2.9. Clark shall be capable of being run in a testing mode where any interface can be enabled/disabled.

7.2.10. Clark shall dynamically add agent as their heartbeat messages are received

7.2.11. Clark shall be able to sync GPS time with system time

7.2.12. Clark shall provide an outbound data buffer

7.3. Clark shall provide an abstraction layer for defining mission communications, computation, and control

7.3.1. The mission module shall have access to all functions within clark

7.3.2. The mission module shall be able to have a defined sample period

7.3.3. The mission module shall be able to modify the order and frequency of communication within a sample period

7.3.4. The mission module shall be able to modify the order and frequency of commands within a sample period

7.3.5. The mission module shall be able to terminate all connections and shutdown

7.3.6. The mission module shall export updated states to a CSV file

7.3.7. The mission module shall have a function for checking for heartbeat timeouts

7.3.8. The mission module shall have a function for adding data to a buffer

8. Clark shall utilize hardware with specifications conducive to use on a specified agent

APPENDIX B

XBEE PRO 900HP CONFIGURATION FILE (XML)

```xml
<?xml version="1.0" encoding="UTF-8"?>

<data>
  <profile>
    <description_file>XBP9B_8074.xml</description_file>
    <settings>
      <setting command="CM">00FFFFFFFFFFF7FFFF</setting>
      <setting command="HP">0</setting>
      <setting command="ID">5ABC</setting>
      <setting command="MT">3</setting>
      <setting command="PL">4</setting>
      <setting command="RR">A</setting>
      <setting command="CE">0</setting>
      <setting command="BH">0</setting>
      <setting command="NH">7</setting>
      <setting command="MR">1</setting>
      <setting command="NN">3</setting>
      <setting command="DH">0</setting>
      <setting command="DL">FFFF</setting>
      <setting command="TO">C0</setting>
      <setting command="NI">0x20</setting>
```

```
<setting command="NT">82</setting>
<setting command="NO">0</setting>
<setting command="CI">11</setting>
<setting command="EE">0</setting>
<setting command="KY"></setting>
<setting command="BD">6</setting>
<setting command="NB">0</setting>
<setting command="SB">0</setting>
<setting command="RO">3</setting>
<setting command="FT">13F</setting>
<setting command="AP">1</setting>
<setting command="AO">0</setting>
<setting command="D0">1</setting>
<setting command="D1">0</setting>
<setting command="D2">0</setting>
<setting command="D3">0</setting>
<setting command="D4">0</setting>
<setting command="D5">1</setting>
<setting command="D6">0</setting>
<setting command="D7">1</setting>
<setting command="D8">1</setting>
<setting command="D9">1</setting>
<setting command="P0">1</setting>
<setting command="P1">0</setting>
<setting command="P2">0</setting>
<setting command="P3">1</setting>
```

```
      <setting command="P4">1</setting>
      <setting command="PD">7FFF</setting>
      <setting command="PR">7FFF</setting>
      <setting command="M0">0</setting>
      <setting command="M1">0</setting>
      <setting command="LT">0</setting>
      <setting command="RP">28</setting>
      <setting command="AV">1</setting>
      <setting command="IC">0</setting>
      <setting command="IF">1</setting>
      <setting command="IR">0</setting>
      <setting command="SM">0</setting>
      <setting command="SO">2</setting>
      <setting command="SN">1</setting>
      <setting command="SP">12C</setting>
      <setting command="ST">BB8</setting>
      <setting command="WH">0</setting>
      <setting command="CC">2B</setting>
      <setting command="CT">64</setting>
      <setting command="GT">3E8</setting>
      <setting command="DD">B0000</setting>
    </settings>
  </profile>
</data>
```