Fall 2019

Communication Capability for a Simulation-Based Test and Evaluation Framework for Autonomous Systems

Ntiana Sakioti
Old Dominion University

COMMUNICATION CAPABILITY FOR A SIMULATION-BASED TEST AND EVALUATION FRAMEWORK FOR AUTONOMOUS SYSTEMS

by

Ntiana Sakioti
B.S. December 2017, Old Dominion University

A Thesis Submitted to the Faculty of
Old Dominion University in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

MODELING AND SIMULATION

OLD DOMINION UNIVERSITY
December 2019

Approved by:

James F. Leathrum Jr. (Director)

Yuzhong Shen (Member)

John Sokolowski (Member)

ABSTRACT

COMMUNICATION CAPABILITY FOR A SIMULATION-BASED TEST AND EVALUATION FRAMEWORK FOR AUTONOMOUS SYSTEMS

Ntiana Sakioti
Old Dominion University, 2019
Director: Dr. James F. Leathrum Jr.

The design and testing process for collaborative autonomous systems can be extremely complex and time-consuming, so it is advantageous to begin testing early in the design. A Test & Evaluation (T&E) Framework was previously developed to enable the testing of autonomous software at various levels of mixed reality. The Framework assumes a modular approach to autonomous software development, which introduces the possibility that components are not in the same stage of development. The T&E Framework allows testing to begin early in a simulated environment, with the autonomous software methodically migrating from virtual to augmented to physical environments as component development advances.

This thesis extends the previous work to include a communication layer allowing collaborative autonomous systems to communicate with each other and with a virtual environment. Traversing through the virtuality-reality spectrum results in different communication needs for collaborative autonomous systems, namely the use of different communication protocols at each level of the spectrum. For example, testing in a fully simulated environment might be on a single processor or allow wired communication if distributed to different computing platforms. Alternatively, testing in a fully physical environment imposes the need for wireless communication. However, an augmented environment may require the concurrent use of multiple protocols. This research extends the Test & Evaluation Framework by developing a heterogeneous communication layer to facilitate the implementation and testing of collaborative autonomous

systems throughout various levels of the virtuality-reality spectrum. The communication layer presented in this thesis allows developers of the core autonomous software to be shielded from the configuration of communication needs, with changes to the communication environment not resulting in changes to the autonomous software.

This thesis is dedicated to my family and friends, thank you for your unwavering support and patience; and to my grandfather for inspiring me to always try to be a better version of myself.

ACKNOWLEDGMENTS

First and foremost, I would like to thank my adviser Dr. Leathrum for his guidance and patience. Without his contributions and help the design and implementation of the system discussed would not have been possible. I would also like to thank the members of my committee, Drs. Shen and Sokolowski, for their guidance and participation in the thesis evaluation process. Additionally, I would like to thank Andrea Robey for her continuous help throughout this research and Nathan Gonda, Thomas Laverghetta, and Cierra Hall for implementing the foundation this research extends.

TABLE OF CONTENTS

Page

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

CHAPTER 1

INTRODUCTION

Autonomous systems are increasingly utilized in a variety of industries such as agriculture, space, military and transportation. Example applications include crop harvesting and weed control, space exploration, reconnaissance and security, transportation and package delivery [1]. Introducing collaboration in autonomous agents can further improve system performance, as multiple systems can potentially perform tasks more robustly and efficiently [2]. Furthermore, in certain applications, collaboration can be an integral part of system success. For example, road safety could increase if autonomous cars collaborated [3], while search and rescue and fire-fighting operations could be faster and more efficient with the introduction of collaboration. Depending on system application and degree of autonomy, collaborative capabilities can lead to increased reliability, performance, efficiency, as well as safety.

While developing an autonomous system is an intricate process, incorporating collaboration capabilities leads to more complex behavior and system requirements. This research develops a communication layer for a Test and Evaluation (T&E) framework meant to be utilized for the development and testing of autonomous systems [4]. By integrating a communication layer, the development of collaborative autonomous systems can be facilitated by the extended framework[1].

[1] IEEE Transactions and Journals style is used in this thesis for formatting figures, tables, and references.

1.1 Testing Throughout the Virtuality-Reality Spectrum [5]

The T&E Framework extended by this research facilitates the testing of autonomous systems throughout their development cycle. At the beginning of the development cycle, testing is conducted using a fully simulated system operating in a virtual world. As autonomous entities traverse the virtuality-reality spectrum, a physical system may first operate in a virtual world, moving to operation in a physical world when the required hardware has been integrated. Migrating from interacting with a fully virtual world to an augmented world to a physical world is done through configuration files defining sources of information such as sensor data. In this manner, the framework shields the autonomous software from knowledge of its test environment.

The communication needs of autonomous systems change depending on their stage of development and operating environment. In a fully simulated environment, testing may be conducted on a single processor or using wired communication if distributed. In a fully physical environment, wireless communication is required, while a combination might be necessary in an augmented virtuality or augmented reality environment.

1.2 Problem Statement

The development of collaborative autonomous systems is a rigorous process that requires extensive testing and time. Autonomous System (AS) developers not only have to implement the desired autonomous behavior but also depend on reliable communication among entities. While AS applications may vary across industries, they share the same basic communication requirements for collaboration, i.e. they need to be able to send and receive messages of various types. A communication layer that provides the ability to collaborate through various protocols would enable AS developers to simply utilize the functionalities provided without having to delve

into specific protocol intricacies or reconfigure the autonomous software each time the protocol utilized is changed.

1.3 Proposed Communication Layer

A communication layer is presented for the purpose of facilitating the development of collaborative autonomous systems throughout the virtuality-reality spectrum. The model provides the capability for the exchange of user-defined messages between autonomous systems as well as with the virtual environments they may operate in. Additional communication protocol functionality may be easily implemented into the layer, affording the autonomous system developer with flexibility in their hardware and robotic middleware choices.

The motivation for this research stems from the need to expand the T&E framework in [4] to provide more flexibility in autonomous system development, testing, and communication. Previous work, although very efficient, provides ad-hoc solutions tailored to specific applications. Their rigid structures and communication restrictions require the user to strip most of these architectures of their functionality or meticulously insert or modify their implementation and ensure they have hardware to support alternative communication mediums.

This communication layer, in conjunction with the framework it extends, is meant to mitigate these drawbacks. By providing the ability to use a multitude of communication protocols, and, if necessary, implement additional ones using the structure provided, autonomous developers are not limited to a particular communication medium. By isolating the layer components and providing clear interfaces, the autonomous developers do not need to be concerned with the protocol specific implementation details, nor does the software need to be modified for the use of

different communication protocols. Autonomous software developer focus can instead directly shift to developing the desired autonomous behavior.

1.4 Contents of the Thesis

Chapter 2 provides a background of topics important to understanding the contents of the thesis. The T&E framework that is extended by the thesis is introduced, followed by an examination of past approaches to autonomous collaboration. The requirements that should be imposed on the communication layer can thereby be identified, while the distinction between previous work and the proposed communication layer for the T&E framework presented will be highlighted. A model of the communication layer is detailed in Chapter 3, with the design features associated with communication and integration requirements being introduced. Chapter 4 discusses the software design for the model detailed in Chapter 3, highlighting algorithms and concepts key to achieving desired functionality. The capabilities afforded by the communication layer are demonstrated in Chapter 5 by studying an example application and the ability to address reliability of the success rate of message transmission.

CHAPTER 2

BACKGROUND AND RELATED WORK

This chapter will establish the context for the thesis by providing definitions and requirements for relevant systems and examining previous approaches. First, the different types of autonomous systems and their applications are discussed, followed by a definition of collaborative systems. Software requirements and previous approaches to conducting collaboration are then presented, followed by a discussion of communication frameworks already implemented and utilized. The framework for the development and testing of autonomous systems that this research extends is then presented.

2.1 Autonomous Systems

Autonomous systems are utilized in a multitude of industries in order to improve task efficiency and safety. Usually comprised of both hardware and software that collaborate on solving a problem or performing an action, they can operate under varying degrees of autonomy. An AS is considered truly autonomous when it can gather and analyze information, find an appropriate course of action and execute that action [6]. While autonomous systems can range from smart thermostats, home service robots, and smart houses, a crucial area of research and development centers around autonomous vehicles due to the complexity, safety concerns, and ethical ramifications associated with their development and operation. Autonomous vehicles (AVs) can be further subcategorized into distinct types. These are briefly detailed in the following sections.

2.1.1 Autonomous Ground Vehicles

An autonomous ground vehicle (AGV) is an autonomous vehicle that operates on the ground [7]. AGVs can be utilized for many applications where operator presence may be dangerous, inconvenient, or impossible. They are extensively used by the military in reconnaissance as well as bomb diffusion operations where the safety of field officers is paramount. Additionally, their use in the civil sector is studied for a variety of applications such as driverless delivery and transportation.

2.1.2 Autonomous Surface Vehicles

An autonomous surface vehicle (ASV) is an autonomous vehicle that operates on the water surface without a crew or operator. Current applications can range from surveillance, naval operations, as well as environmental and climate monitoring [8]. AGVs are especially valuable in oceanography, as they are more capable than weather buoys and far cheaper than manned research vessels [9].

2.1.3 Autonomous Underwater Vehicles

An autonomous underwater vehicle (AUV), is an autonomous vehicle that operates below the water surface. AUVs can be small or large, the largest weighing thousands of pounds, requiring their own support vessels. They can glide, stop, or hover, and are attractive options for ocean-based research, especially since they can avoid inclement weather by going below the sea-surface [10]. They have been continuously deployed in deep sea exploration as well as search operations for missing ships and airplanes, such as the U.S. Navy cruiser Indiana, and Air France Flight 447 [11].

2.1.3 Autonomous Aerial Vehicles

An autonomous aerial vehicle (AAV), commonly referred to as a drone, is an autonomous aircraft that operates without a pilot or operator [12]. While mostly utilized by the military in the past for surveillance and warfare applications, their use is rapidly expanding. Currently, AAV adoption for delivery services is in the testing stage for companies such as Dominos and Amazon [13], while AAV use in agriculture is already aiding in crop monitoring and soil assessment [14].

2.2 Autonomous System Model

Autonomous software is usually categorized into different functional modules. A popular modularized approach to autonomous software development is the *Sense*, *Plan*, and *Act* paradigm, the modules of which are interconnected using inputs and outputs [15]. Software under the *Sense* module receive input from all peripheral sensors and generate and output a perception of the environment. The *Plan* module in turn receives this information as input and generates a plan of actions based on the environment perception, desired functionality, and past actions. The *Act* module executes the plan of actions generated by the *Plan* module by converting them to control signals that are sent to the system's actuators. This paradigm can be seen in Figure 1. Alternative models exist, such as including a perception stage between sense and plan [16].

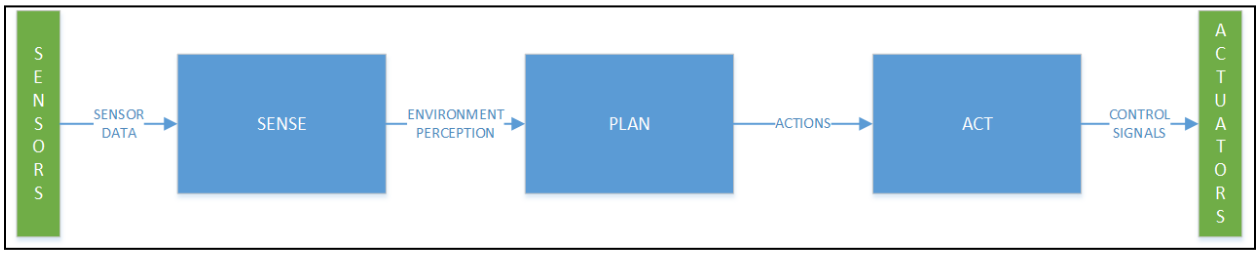

Fig. 1. Sense, Plan, and Act Paradigm Visual.

2.3 Collaborative Autonomous Systems

Autonomous collaborative systems, also referred to as autonomous collaborative agents, are autonomous systems that collaborate to achieve tasks. They can be part of a team comprised of a small number of systems or part of an autonomous swarm. Collaborative modes of operation, communication, and frameworks are discussed.

2.3.1 Modes of Operation

Depending on the desired functionality, collaborative autonomous systems can operate in a leader-follower mode or parallel mode [17]. In a leader-follower mode, autonomous decisions are made by the leader AS, while follower vehicles carry out commands sent to them by the leader. In a parallel mode, however, autonomous systems collectively collaborate to perform desired tasks. While a system can have more than one leader, communication in a leader-follower mode is only needed between leaders and followers. However, in a parallel or decentralized approach, each system must have the same communication capabilities and protocols to be able to communicate with all other agents [18].

2.3.2 Collaborative System Communication

Communication between autonomous agents can be achieved through a variety of mediums depending on on-board hardware. In [19], *AAV to AGV Collaboration* was conducted through a Secure Shell (SSH) wireless network connection. A different approach was presented by the authors of [20], who utilized a Radio-Frequency (RF) system along with an infrared transceiver and a Hertzian wave transmitter to achieve *Paralyzed and Non-Paralyzed Ground Robot Communication*. For future research, they expressed a need to incorporate Bluetooth as a

communication medium in their application. Both Bluetooth and TCP/IP protocols are utilized in [21], for a multi-agent robotic system called *SMART* to establish communication among robot and control software and between client and server respectively.

2.3.3 Collaborative Autonomous Frameworks

Collaborative autonomous architectures are designed to facilitate the development and testing of collaborative autonomous systems. In [22], the following guidelines and respective advantages of using these frameworks were highlighted:

- Offer tools and functions to simplify development of collaborative applications
- Offer high-level abstractions and interfaces to facilitate application integration, reuse, and development
- Hide heterogeneity of devices, platforms, and operating environments
- Hide distribution and communication details in the environment
- Facilitate communication among the different components of the systems
- Provide common services for general purpose functions in order to reduce development efforts and avoid duplication
- Provide a common architecture to add new services and features without changing system applications
- Offer properties such as security, reliability, and quality of services
- Supply the necessary tools to enhance the performance, stability, safety, and scalability of the collaborative autonomous application

Following these guidelines ensures that collaborative solutions are not tailored to a specific implementation but are instead compatible with a variety of applications.

2.3.3.1 Ad-Hoc Approaches to Collaborative Development

Many developers design autonomous software and communication tailored to a specific implementation in order to best fit their application needs. This approach, although sufficient and optimal for the current design, hinders modularity and reusability in future scenarios. The authors of [23] built a model *of ASV and AAV Synergetic Cruise* following the leader-follower mode. In this system, one ASV acts as the control station for three AAVs accomplishing several tasks. Collaboration is conducted using a wireless connection, with the control station (ASV) receiving sensor information from the three AAVs and sending back further instructions for the AAVs. Although the system exhibits robustness and efficiency, different applications and operating environments would require continuous improvement for the synergetic model. Additional information is required on the protocols used for communication between systems.

In [2], *Reactive and Deliberative Ground Vehicle Collaboration* is achieved, defining a model approach for heterogeneous robots. A paralyzed robot whose goal is to reach a destination emits a signal to request assistance by being pushed towards a particular destination. Non-paralyzed robots in turn roam about the environment until perceiving a signal emitted by the paralyzed robot. Once a signal is received, non-paralyzed robots work together to push the paralyzed robot in the desired direction. Communication is one-directional -- from a paralyzed to a non-paralyzed robot -- and achieved via an RF system, with the authors mentioning the desire to switch to Bluetooth communication in the future. Along with direct cooperation shown when a non-paralyzed robot attempts to push a paralyzed one, indirect cooperation occurred when mobile robots had to add their forces in order to make the robot move. Although the architecture presented resulted in a scalable robust system that is adaptable to changes due to environment disturbances, their approach restricts architecture use to very similar applications. Additionally, the potential

efficiency and reliability benefits of two-way communication both between the helper robots and the paralyzed robot were not studied.

A *Cooperative Architecture for a Robotic Swarm* based on dynamic fuzzy cognitive maps is presented in [18]. A swarm is a multi-agent system (MAS) that can be comprised of both heterogenous and homogeneous robots, thus allowing for both centralized and non-centralized control. In this application, a homogeneous swarm is assumed, exhibiting non-hierarchical control. The architecture was composed of three layers (reactive, deliberative, and cooperative), in order to support navigation system development, but according to the authors can be standardized and is applicable to other systems based on DFCM. Since testing and analysis was all simulated, communication protocols were neither used nor discussed in this paper.

2.3.3.2 Non-Application Specific Approaches to Collaborative Frameworks

Numerous architectures have been developed to meet the guidelines and provide the advantages highlighted at the beginning of this section in order to facilitate the development and testing of autonomous systems. One of the most widely used robotics frameworks in research is the *Robot Operating System (ROS),* a robotics middleware for robot software development [24]. ROS is used to conduct inter-process communication by the T&E framework this research enhances and is also integrated as one of the communication protocols for AS to AS communication provided by the collaboration layer detailed in this thesis. ROS provides services for hardware abstraction, low-level device control, inter-process and peer-to-peer message parsing, package management, as well as implementations for common functionalities. ROS implements a Publish/Subscribe scheme that is topic-based, where software nodes publish or subscribe to a topic. Messages can be user-defined with the source and destination of a message remaining anonymous.

Several distributions for ROS exist for different types and versions of operating systems, maintaining the same basic architecture.

Software developed using ROS is organized in packages, which can contain source code, any necessary configuration files or third-party libraries, and build files. Four primary functions are used to conduct communication with ROS: advertise, publish, subscribe, and callback. Advertise and subscribe are utilized to establish a topic, while publish sends a message to the chosen topic, and callback handles messages that have been received from a topic. The ROS Master is essentially a control station for ROS [25]. Nodes, topics, and callbacks are registered on the Master, which keeps track of these processes, and allows nodes and callbacks to locate their topics of interest. The ROS network can be further studied in Figure 2.

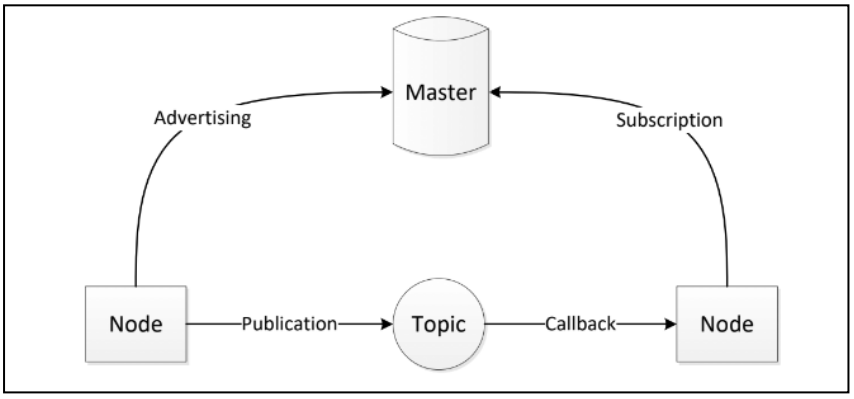

Fig. 2. ROS Network.

It is important to note that although ROS is very reliable and robust, hiding communication details from the developer and modularizing components for easy re-use, it restricts communication to the TCP/IP protocol.

In [26], a communication and control architecture was proposed to improve the capability and flexibility of autonomous systems. Using an object-oriented paradigm, the *Little-Object-Oriented Ground User Environment (LOGUE)* allows the sharing of both task and behavioral information among autonomous systems as well as behavior servers using Java RMI. Java RMI is a Java API that supports direct transfer of serialized Java classes among other capabilities [27]. The autonomous robots are comprised of a three-layer architecture consisting of modules for communication, action management, and device. The communication module handles system-to-system communication while the action management module translates messages into priority tasks, with the device interfacing directly with system sensors and actuators. *LOGUE* implementation is identical for autonomous robots, with a GUI interface and behavior database added to the behavior server. For successful object transmission, both systems are required to have the object class; transferring a class from one virtual machine to another is not possible, so the class file precedes the transmission of the object. Due to the use of Java RMI, this approach is highly scalable, overcoming basic technological difficulties that are handled by the API.

The author of [28] introduces a purely simulation-based layered framework for the development of collaborative autonomous systems (CAS), thus denoted as the *CAS* framework. Closely following the guidelines setup in the beginning of this subsection, the framework proposed retains the isolated development advantage of layered architectures, providing flexibility and tools to conduct collaboration while shielding the developer from protocol-specific intricacies. The architecture is designed to be compatible with many robotic platforms, supporting both internal

layer to layer communication, and limiting external communication to same layer levels, also requiring autonomous systems to contain identical layer architectures. The number of layers in the architecture is not limited and can be specified by the developer while a rigid structure to user-defined messages is embedded in order to ensure reliable communication. In order to increase reliability, the author mentions the need for integration to physical systems, leading to increased applications of this framework.

A different approach for a *Multi-Layer Architecture based on ROS and JADE Integration* for autonomous transport vehicles (ATVs) is detailed in [29]. The research was focused on providing social abilities to ATVs, utilizing a four-layer architecture. The upper (social) level was responsible for the interaction with other ATVs, while the lower (functional) level interfaced with all vehicle sensors and actuators to provide ATV control. The two intermediate layers were tasked with abstracting social behavior from functional behavior, pre-processing and storing information for fast response. The architecture was built on top of ROS, with the social layer consisting of a Multi-Agent System (MAS) JADE agent that offered transportation services and communicated with other ATVs. The upper intermediate layer in turn integrates ROS, used for ATV control, and JADE, used for social capabilities, communicating the agent at the social layer with the lower intermediate layer. Efficient layer division allows for modularity, while social abilities are abstracted from control functionalities, allowing isolated development.

The authors of [30] detail a more general non-vehicle-based approach, the *Knowledge Query and Manipulation Language (KQML)*, a language protocol for exchanging information. Using *KQML* provides agents the ability to transmit messages composed in their own representation language, wrapped in a *KQML* message. *KQML* can be viewed as a three-layer language. The content layer is the actual message content, while the communication layer encodes

message features describing parameters such as sender and recipient identity. Finally, the message layer's primary function is to identify the protocol required to deliver the message. Two specialized programs are needed to facilitate communication: a router and a facilitator, and a library of interface routines. Routers are content independent message routers, each agent associated with its own router. Routers are identical and are only concerned with the *KQML* arguments such as an Internet address for the message destination. Facilitators are in turn used to deliver incompletely addressed messages. Routers rely on facilitators to help them find message destinations. Typically, there exists one facilitator for each local group of agents. The *KQML* Router Interface Library (KRIL) lays between the application and the router, with the purpose of making access to the router as simple as possible for the programmer. It is embedded in the application and has access to tools that analyze the content field of the message. There can be various KRILs, i.e. one for each application type and one for each application language. *KQML* offers a standard protocol for autonomous agent communication, along with providing abstraction of an information source or destination and permitting the use of whatever language the programmer prefers.

Another general approach is the Advanced Message Queueing Protocol (*AMQP*) [31]. AMQP is an open standard application layer that features message orientation, queuing, routing, as well as reliability and security [32]. AMQP, like ROS, deals with publishers that produce messages, and consumers that obtain and process them. AMQP is a wire-level protocol, meaning that the data transmitted is a stream of bytes. This allows any tool available to conform to this data to create and interpret messages, increasing interoperability [31]. AMQP assumes a reliable transport layer protocol such as the Transmission Control Protocol (TCP). Publishers and consumers discover each other via exchanges created by the consumer with a given name that is public. Publishers send messages to an exchange, and consumers pull messages from a queue.

AMQP allows for application data to be of any form and in any encoding the application requires. A bare message is defined that allows an optional list of standard properties (id, user id, creation time etc.), followed by an optional list of application-specific properties and the message data [32].

2.4 The T&E Framework Extended by this Work

The communication layer presented in this thesis was designed as an extension to a *Test and Evaluation Framework* for autonomous systems [4]. The purpose of this framework was to facilitate testing of autonomous vehicles throughout the development cycle, by enabling testing capabilities throughout the virtuality-reality spectrum. A modular design approach for the autonomous software is assumed, which allows isolated testing of components even at the early stages of development. The architecture was designed with the *Sense, Plan, Act* module paradigm in mind for demonstration purposes, although it is not limited to that module configuration.

The focus of the framework was to decouple software components from their respective input and output sources in order to allow for additional components to control data augmentation [4]. Thus, communication between modules is handled by the framework, while the autonomous software is isolated from its operating system. Sensor data can therefore be replaced with simulated data based on the virtual environment to perform testing using virtual data. The source of information provided to modules is controlled by configuration files. The isolation of the software modules also allows for the framework to directly supply information to a particular module (i.e. the Sense module can be bypassed completely in order to supply the Plan module with a set world representation). A high-level view of the architecture is shown in Figure 3.

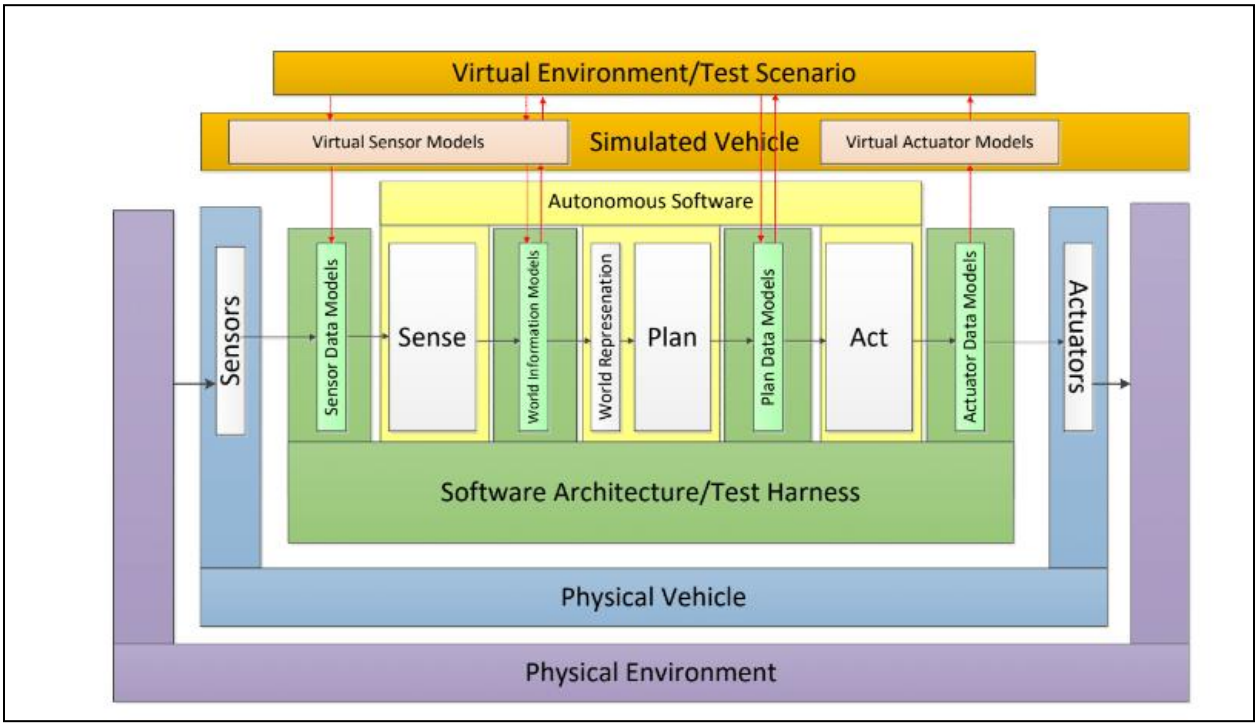

Fig. 3. High-Level T&E Framework Architecture [4].

The *Physical Vehicle* represents the physical autonomous system and is comprised of *Sensors* that provide information about the system and its environment, and *Actuators* that control system operation depending on control signals. The *Physical Environment* in turn represents all external factors that can influence and be influenced by the autonomous vehicle's operation. On the virtual side, the *Simulated Vehicle* contains *Virtual Sensor* and *Actuator Models* that mimic their physical counterparts. The *Virtual Environment* similarly represents a simulated version of the environment a vehicle operates in [4].

The *Test Harness* is placed between the autonomous system components and the rest of the framework, so that information is decoupled from its source so that data can be manipulated to

test throughout the virtuality-reality spectrum. Data can be injected before and after each autonomous system component, in order to manipulate *Sensor*, *Plan*, *Actuator* and *World Information Data Models*. ROS is in turn utilized to achieve communication between the separate framework and autonomous software components.

The T&E Framework API is a class denoted as *Node*, which provides the functionalities afforded by the framework. Each autonomous software module (i.e. *Sense*, *Plan*, *Act* etc.) inherits from *Node* to gain access to the framework capabilities. This relationship can be studied in Figure 4.

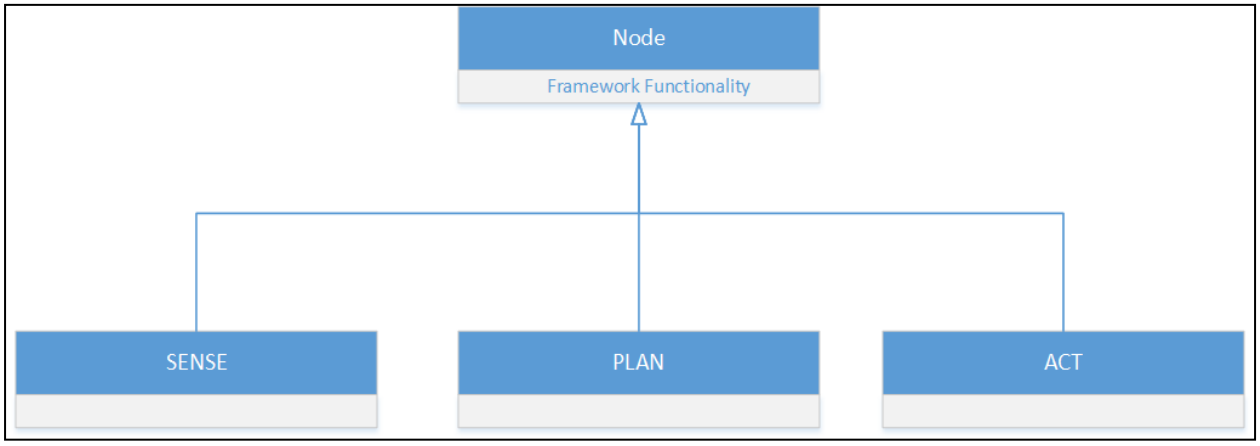

Fig. 4. Module Inheritance from Node Example.

CHAPTER 3

COMMUNICATION LAYER MODEL

The communication layer is intended to facilitate collaboration between autonomous systems developed using the framework described in [4]. Specifically, the communication layer is designed for autonomous module-to-module communication, with ease of use being one of the primary goals. It is also utilized behind the scenes to support communication with a remotely located virtual world to replace sensor or object presence data with simulated sensor data. This chapter introduces the communication layer design, a layered architecture intended to distance developers of varying expertise from the intricacies of establishing different communication protocols. A set of requirements is first presented to highlight capability expectations and design constraints. The layered structure proposed to satisfy these requirements is then discussed, including the object-oriented aspects of this layer and their interaction. Finally, the benefits of utilizing this layer in conjunction with the T&E framework it extends will be discussed.

3.1 Communication Layer Requirements

The requirements of the communication layer presented in this thesis were primarily derived from typical collaborative framework guidelines such as those listed in Section 2.3.3 [16]. With modularity, scalability, and reusability in mind, most of the requirements stemmed from the need to shield the autonomous developer from the implementation of protocol-specific communication. This supports testing throughout the virtuality-reality spectrum by allowing the components and communication to be reconfigured as needed, without any modification to the autonomous software. As the layer is intended to be an extension of the T&E framework described

in [4], the assumption of a modular autonomous software architecture was inherited. The requirements imposed on the communication layer for a T&E framework are:

- A communication interface to provide the user with clearly defined methods to perform communication, with the abstraction of protocol implementation allowing the developer to not worry about protocol-specific intricacies
- A flexible architecture to enable the use of various communication mediums, while not imposing restrictions on same level module-to-module communication provides system and hardware flexibility
- Support for direct external module-to-module communication
- Support for user-defined messages allows for a wide variety of content to be delivered

These four requirements are not only intended to facilitate the collaboration of autonomous systems but also enable the developer to focus on autonomous behavior implementation, expecting that communication is reliable and successful. This section elaborates on these requirements.

3.1.1 Communication Application Programming Interface

In order to ensure ease of use and application integration, a communication Application Programming Interface (API) is considered a necessity for the implementation of this layer. The API should be comprised of a variety of methods which the user can directly interface with to achieve desired communication. This allows for the abstraction and modularization of protocol specific communication and error-handling, thus reducing developer efforts. Use of general-purpose functions can also help avoid duplication of services. The API should extend the API presented in [4], which supports autonomous software interfacing with the T&E framework

3.1.2 Flexible Architecture

The communication layer is intended to provide communication capabilities required by the autonomous developer. As the communication mediums used (i.e. Bluetooth, IR, TCP/IP, ROS, etc.) depend on the specific application, the architecture should be extendable to allow use of additional protocols as needed. Adding new features and services should not require a modification of system applications. Therefore, providing a common structure for new features and services to follow is another requirement imposed on the system.

3.1.3 Direct External Module-to-Module Communication

As a modular autonomous software architecture is assumed, module-to-module communication is another requirement for the communication layer. Since the T&E framework this layer extends handles internal module-to-module communication, the layer only needs to establish external module-to-module communication between AS systems and the virtual environment (VE). Messages not only need to be directed to the designated AS and VE but also to the appropriate module of that AS and VE. The routing of information to the proper module should be hidden from the AS developer.

Unlike the autonomous system framework presented in [22], communication is not restricted to same level modules. For example, in a leader-follower mode with the autonomous systems following the Sense-Plan-Act module architecture, the Plan stage of the leader AS might want to send a “STOP” command to the ACT module of one of the follower autonomous systems. Thus, communication with unlike modules should not be precluded to allow for such scenarios and not restrict functionality.

3.1.4 User-Defined Messages

The information transferred using this communication layer is represented in the form of messages. Each message can have a distinct structure and contents depending on the application needs. Different types of messages can be classified under two categories, command and control, and information. Decision-making might require the exchange of high-level decision or requirement messages. Sensing modules in turn might need to transmit and receive messages with information about environment objects – i.e. location, dimensions, object type. For example, in the case of mapping a room, command and control messages would be used to coordinate partitioning the room to avoid overlap, while information messages would share the results. The variety of messages that the system handles should therefore be defined by the user to allow flexibility in message definition.

3.2 Model Architecture

The communication layer presented in this thesis was designed to extend the T&E Autonomous System (AS) Framework implemented in [4], the high-level architecture of which is shown in Figure 2. The *Autonomous Software* is comprised of the *Sense*, *Plan* – including the *World Representation*, and *Act* modules, which represent the main stages of an autonomous system model. The specific behaviors of each module might vary depending on the autonomous system application, with the general behavior of the modules following the pattern of *Sense* sensing the environment and providing a *World Representation*, *Plan* assessing the *World Representation* information and deciding on a course of action, and *Act* carrying out the actions selected by *Plan*.

The *Test Harness* is placed between the autonomous software components and the remainder of the T&E Framework with the purpose of decoupling the autonomous modules'

knowledge of the source and utilization of data outside of each component. The *Test Harness* therefore routes the information from module to module, allowing for the injection and manipulation of data in cases of virtual and augmented testing. The *Virtual Environment* represents a generated version of the environment the vehicle operates in, which could be a simulation of the environment or a testing module that supplies an approximation of data obtained from the environment [4].

The additional capability provided with the layer extension is communication between autonomous systems as well as the *Virtual Environment* – i.e. the *Communication Layer*. Specifically, the layer provides functionality for AS module to AS module (or AS to *Virtual Environment*) communication using implemented communication protocols. The expanded high-level architecture is shown in Figure 5. The sections pertaining to the physical and virtual vehicle and environment were removed so that the communication layer extension can be clearly identified and studied. The *Communication Layer* API is therefore integrated into the framework API to provide communication capabilities to all modules. User-defined messages are routed from AS module to AS (or AS to VE) module using the appropriate protocol and accessed by the software when desired.

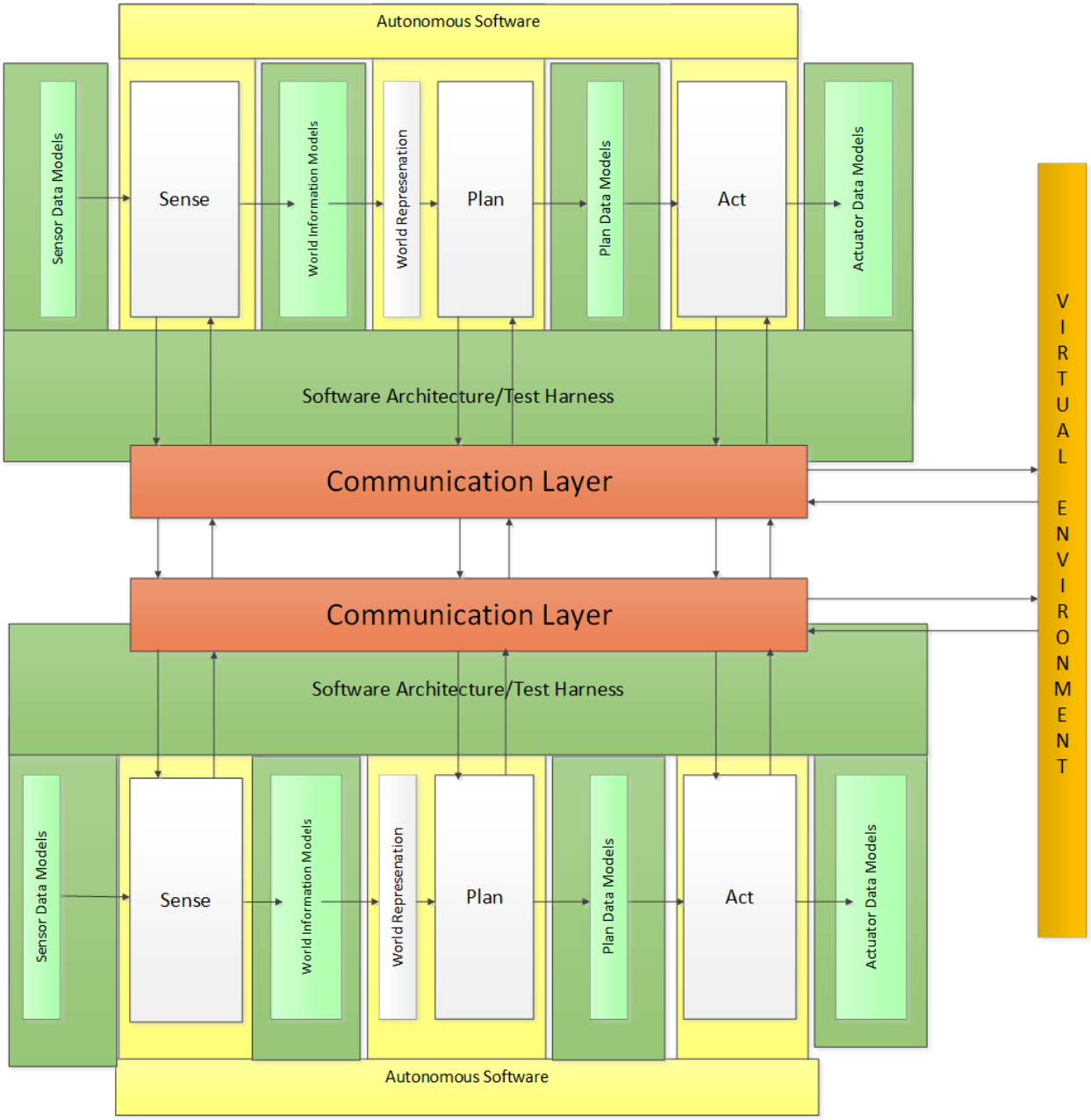

Fig. 5. Extended High-Level Architecture.

3.3 Developer Roles

In order to highlight the benefits and functionalities provided by the communication layer and T&E Framework it is important to discuss the different software developer roles involved when developing an autonomous system using this T&E Framework and communication layer extension. Different system components may have distinct lifecycles for design, development, and testing. Detailing these roles can aid in identifying their responsibilities as well as the division of component management among roles. The development roles identified are:

- Autonomy Developer – responsible for designing and implementing the system's autonomous software, e.g. the *Sense*, *Plan*, and *Act* modules
- Virtual Environment Developer – responsible for the development of the *Virtual Environment* in which the autonomous systems may operate
- Hardware Driver Developer – responsible for interfacing with the autonomous system -i.e. sensors, actuators etc.
- Communication Driver Developer – responsible for implementing the different communication protocol capabilities
- Framework Manager – responsible for integrating all developed components (i.e. *Virtual Environment*, Communication Protocols, Autonomous Software Modules) into the framework

It is important to note that the *Framework Manager* role allows the *Virtual Environment Developer* and *Autonomy Developer* to not know the particulars of the framework or communication protocol implementations.

3.4 Communication Interface

The API provides the *Autonomy Developer* access to the external communication capabilities developed in this layer. As collaboration is conducted through the transmission and receipt of messages, the operations provided are centered around sending a message and accessing received messages. The communication layer currently supports a polling approach for message retrieval by the autonomous software. As incoming messages are received, they are placed in a queue to be retrieved when the autonomous software is ready to perform message handling. Future work could implement an interrupt-based approach using callback functions for the communication layer to pass messages to the autonomous software. The methods provided via the interface therefore are:

- SendMessage
- CheckForMessage
- GetNextMessageType
- GetMessage

For the autonomous software to transmit a message the *Autonomy Developer* only needs to provide a destination and the message. The *CheckForMessage* function permits the user to check if there is a received message that has not yet been handled. If there is a message to be handled the developer can use the *GetNextMessageType* function to identify the message type, while the *GetMessage* method will return the oldest received message for handling. In future work, it would be beneficial to explore the integration of priorities to messages, enabling them to bypass any queue of incoming messages.

3.5 Protocol Configuration

For the communication layer to correctly handle the routing of messages configuration properties must be determined and set by the *Framework Manager*. This is currently done using configuration files. The layer classifies autonomous systems using integer ids, so one of the configuration files the *Framework Manager* must edit is file mapping an AS to a specific id. While the *Framework Manager* specifies the destination AS of a message using a "friendly" name (a string representation), the layer utilizes the aforementioned configuration file to map those "friendly" names to integer ids. Similarly, AS software modules are also classified using integer ids to ensure that a message is delivered to the correct module. This id is also set via a configuration file, specifically the module's configuration file, with the *Framework Manager* being tasked with ensuring that same level modules have the matching ids for all autonomous systems. Thus, AS ids are used to route messages between AS's, and module ids are used to route messages to the appropriate AS module.

The communication layer supports the use of multiple protocols in the same application. The *Framework Manager* can specify the desired protocol for each AS-AS or AS-VE communication. This is achieved through a configuration file in the form of a communication table shown in Table 2, which presents the expected format of the configuration file for 4 autonomous systems and one *Virtual Environment*. Table 1 presents the "friendly" name to id mapping utilized in Table 2.

TABLE 1

FRIENDLY NAME TO ID MAPPING FILE EXAMPLE

0	Vehicle1
1	Vehicle2
2	Vehicle3
3	Vehicle4
4	Virtual Environment

TABLE 2

COMMUNICATION CONFIGURATION FILE EXAMPLE

	0	**1**	**2**	**3**	**4**
0	*X*	R	R	R	R
1	R	*X*	B	B	R
2	R	B	*X*	B	R
3	R	B	B	*X*	R
4	R	R	R	R	*X*

Each table element is a character corresponding to a protocol type. Currently, two protocols have been implemented, Bluetooth (‘B’ in the table) and TCP/IP through ROS (‘R’ in the table). ‘X’ in the table signifies that communication should not be occurring between a system and itself. In this example, id 0 corresponds to the *Virtual Environment*, with ids 1 ,2, and 3 corresponding

to physical systems and 4 corresponding to a virtual AS. Communication between any AS and the Virtual Environment as well as between physical and virtual systems is thus conducted using ROS, while physical to physical system communication is achieved through Bluetooth. Figure 6 depicts the scenario described.

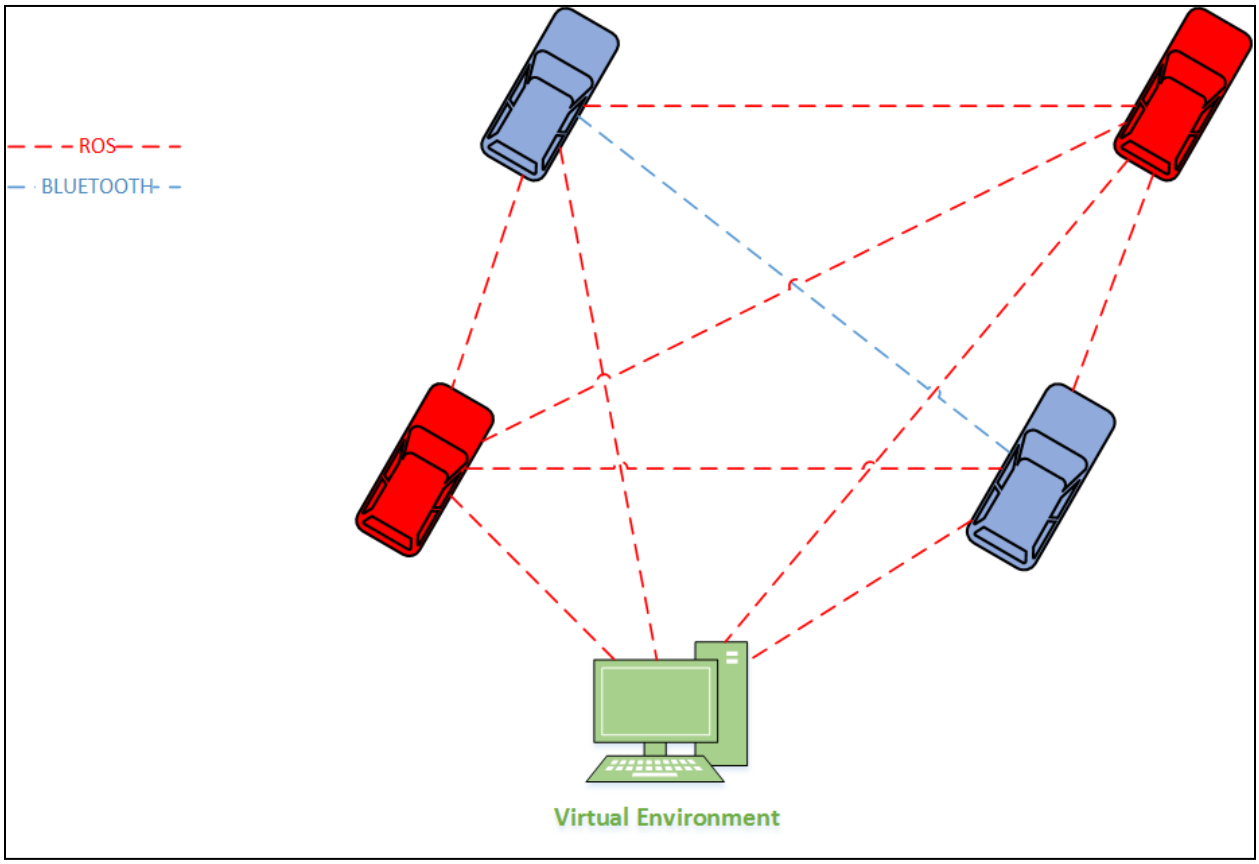

Fig. 6. Example Communication Depiction.

Bluetooth establishes communication using distinct Bluetooth addresses that are in the format “XX:XX:XX:XX:XX:XX”, where ‘X’ can be a capitalized character or a number. As Bluetooth addresses are specific to the devices used, the user must provide the Bluetooth addresses mapped to their corresponding autonomous systems in a configuration file. That configuration file follows the format shown in Table 3, with an application of 4 autonomous systems utilized as an example.

TABLE 3

BLUETOOTH ADDRESS CONFIGURATION FILE

0	08:ED:B9:B2:12:7A
1	68:A3:C4:4A:B3:BA
2	3C:95:09:8E:5B:6C
3	4C:ED:DE:9E:39:10

ROS in turn performs communication using distinct topic names, as mentioned in Section 2.3.3.2. To reduce complexity and the need for another configuration file, each AS was mapped and subscribed to a topic designated by its “friendly name”. A “friendly” name is a string chosen by the autonomous developer to denote each of the autonomous systems (i.e. Red, Charlie, Bravo etc.). Thus, when using ROS, an AS sends a message to another AS or the VE by publishing to the topic that bears the destination’s “friendly name”.

It is important to note that it is assumed that the framework manager is aware of the format configuration files should follow. The system will not attempt to identify erroneous configuration files; a fault in configuration files could thereby potentially lead to a crash and shutdown of the system or unexpected and undesired behavior. Framework managers are consequently expected to correctly perform their edits, maintaining the expected file format.

3.6 Communication Layer Model

The communication layer was designed as a layered architecture following an object-oriented approach. The layer architecture, closely resembling that of the Open Systems Interconnection (OSI) model, ensures compatibility with a plethora of applications. The object-oriented approach achieves modularity while affording flexibility in hardware and communication components.

3.6.1 Layered Approach

The autonomous software and framework architecture extended by the communication layer can split under four categories: *Application, API, Communication*, and *Protocol*. All autonomous software components (i.e. *Sense, Plan,* and *Act*) are classified under *Application*. The *API* in turn provides the autonomous software components with methodology to access the capabilities afforded by the T&E Framework and communication layer. Under *Communication*, outgoing and incoming messages are handled by managing the use of different protocols. Specific protocol implementations are in turn categorized under *Protocol*. These encompassing categories directly map to the OSI Model layers shown in Figure 7.

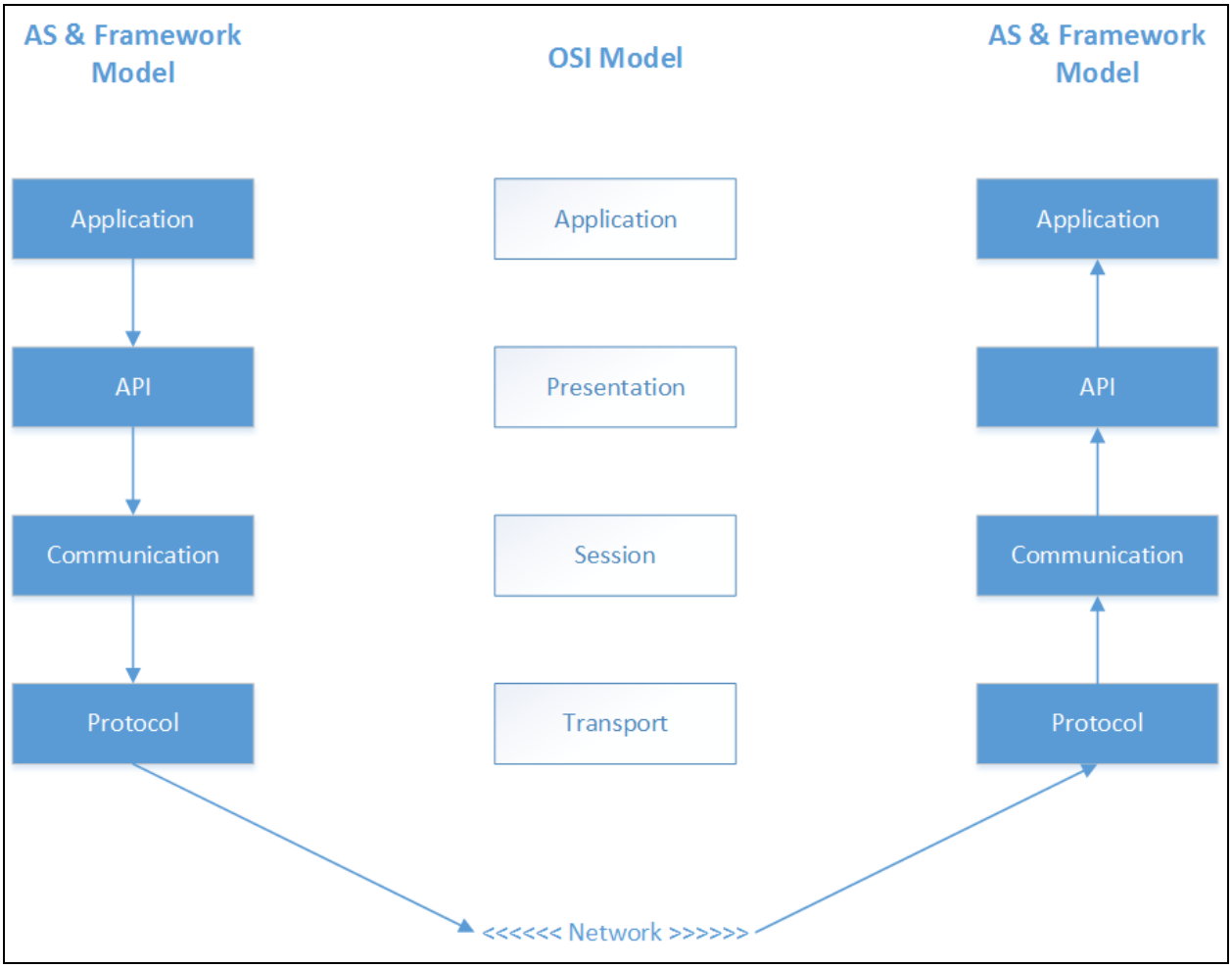

Fig. 7. Map of AS Application and T&E Framework to the OSI Model.

Application maps to OSI's Application layer, while *API* maps to OSI's Presentation layer. *Communication* in turn maps to the Session layer which manages and synchronizes the direction of data flow. *Protocol* similarly maps to the Transport layer of the OSI model, as both ensure end-to-end data transfer between applications [33].

3.6.2 Object-Oriented Design

The architecture components are represented as objects, with methods providing the required functionality. Using object-oriented concepts and data encapsulation, the desired communication functionalities are provided while hiding the implementation details from the user. This not only ensures that the user does not modify the architecture or data that should not be modified but also provides a modular structure that can accommodate additional communication protocols if needed. Figure 8 illustrates the layer class structure, basic class functionalities, along with the relationships between the different classes.

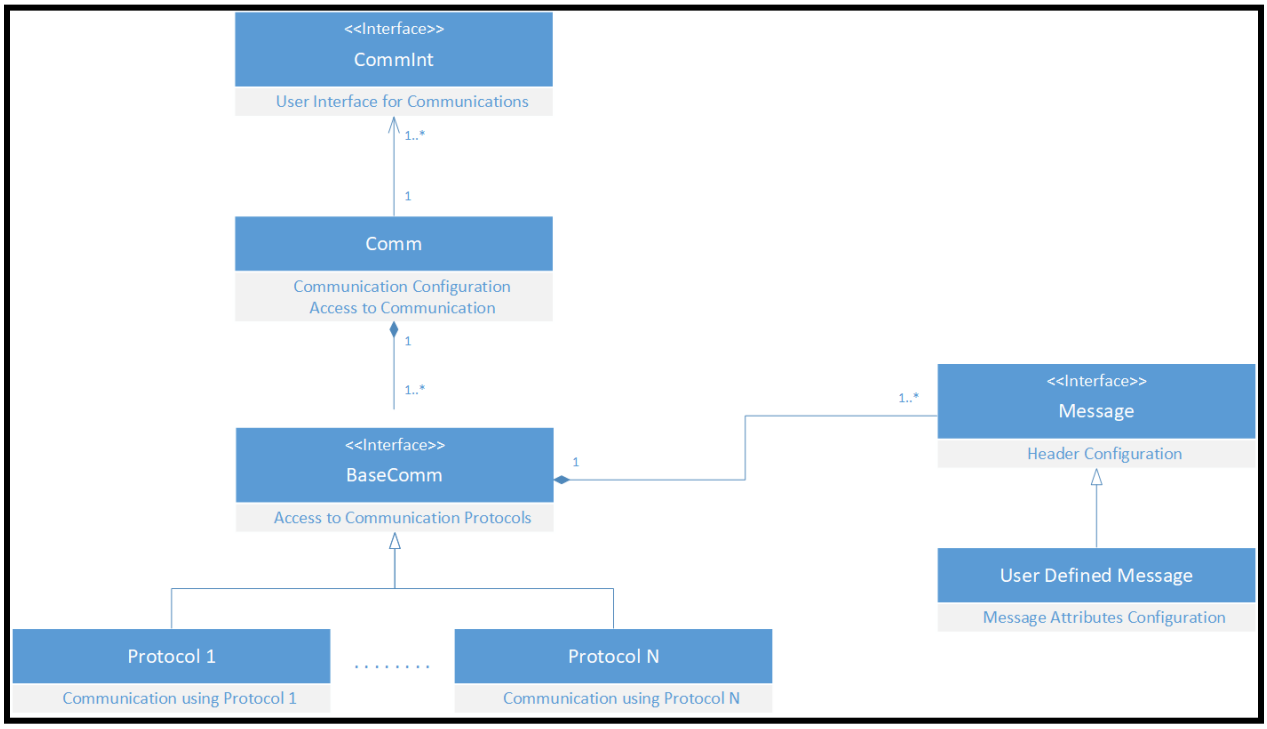

Fig. 8. High-Level Class Diagram.

The *CommInt* (Communication Interface) class is the communication layer API, which is integrated into the T&E Framework API to provide the *Autonomy Developer* with direct access to methods that transmit, check for, and get a message. The *Comm* class is tasked with configuring and initializing the communication parameters such as the configuration files for the AS ids and communication table. Along with initialization, the *Comm* class also provides access to communication by containing instances of the *BaseComm* class. To avoid unnecessary duplication, only one object instance of *Comm* can exist for each autonomous system. This led to a multiplicity relationship of one-to-many with *CommInt*, and many-to-one with *BaseComm*.

The *BaseComm* class acts as an interface to the different communication protocols, providing access to protocol specific methods and a message log where incoming messages are stored; this log is shared by all protocols and is a container of logs, containing one log for each AS module. It is important to note that each AS module is associated with a specific module id, to aid in the routing of messages. This is achieved by establishing an inheritance relationship between *BaseComm* and the corresponding protocol implementation classes. Polymorphism enables easy implementation of new communication protocols while having a common interface to the *Comm* class.

Although the message content and structure are user-defined, for the layer to correctly route messages a layer header must be attached to each message. The variables chosen to compose the header are:

- *Message Id* – an integer tuple; the first element is the id of the creating AS while the second element is a message id unique to the creating AS
- *Source Id* – an integer tuple; the first element is the id of the AS while the second element is the module id

- *Destination Id* – an integer tuple; the first element is the id of the AS while the second element is the module id
- *Communication Type* – a character denoting the communication protocol used
- *Message Size* – integer size of message data
- *Message Type* – an integer denoting the message type

The structure of messages including the header is presented in Figure 9.

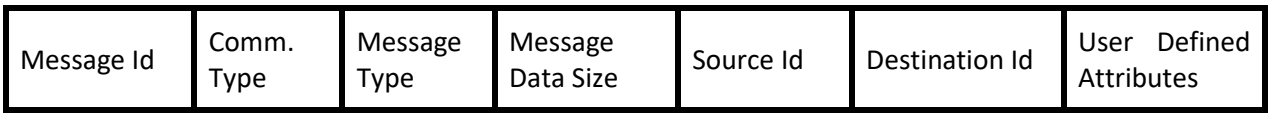

Fig. 9. User-Defined Message Structure.

The *Message* class provides and sets the header information, while the user must in turn define multiple *User-Defined* Message classes. To attach header information to user-defined messages an inheritance relationship between *Message* and *User-Defined Message* classes is realized. The *Message* class also maintains a multiplicity relationship of many-to-one with *BaseComm*, where received messages are stored upon receipt.

3.6.3 Component Interaction

Section 3.6.1 presented the layer components as well as the relationship between them. This section will illustrate the order of component interaction for the different communication

operations supported. The two operations that will be outlined are: send, and receive, the latter being comprised of check for message and get message operations.

As is the case for all operations, the *Autonomy Developer* initiates the send message operation using the send method provided by the *CommInt* class. The layer provides both a point-to-point and broadcast send operation, so depending on the chosen destination *Comm's* send point-to-point or send-broadcast method will be invoked. Depending on the desired protocol for the source-destination set, the corresponding protocol's send method will be called. The send method will then inform their caller whether transmission was successful, which will subsequently be passed on to the *Autonomy Developer* via the interface. It is important to note that some protocols provide functionality to determine success while others do not; if the application requires it, that functionality can be provided in the communication layer rather than the autonomous software. The sequence diagram for the send operation is shown in Figure 10. It is also important to note that the second protocol component *Protocol AS 2* represents a protocol implementation running on a separate AS system and is added to the figure to showcase communication between different autonomous systems.

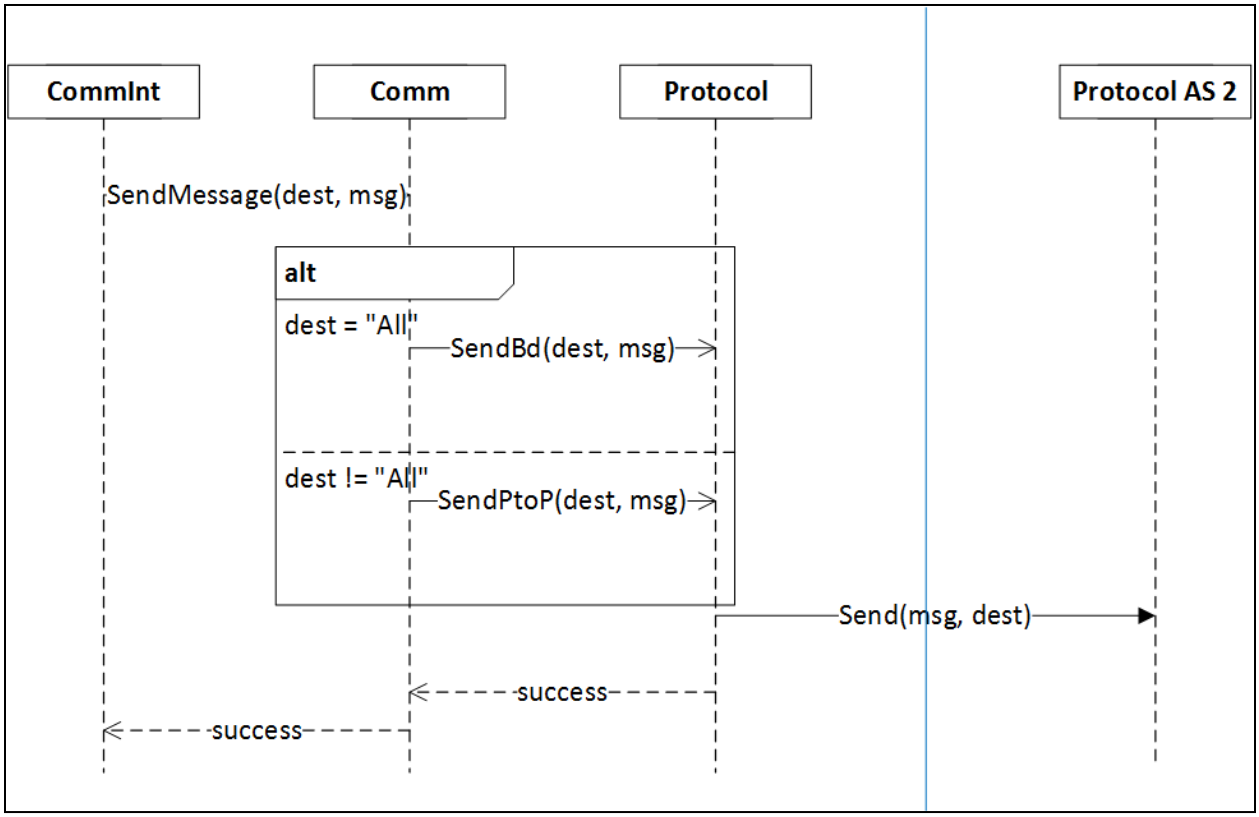

Fig. 10. Send High-Level Sequence Diagram.

Similarly, depending on the message source, the receive operation will be initiated by invoking the listening function of the specific protocol. Once an incoming message is detected, the cross-protocols' shared function updating the message log will be called, which inserts the message into the appropriate message queue denoted using the module id (*modId)*. Once the *Autonomy Developer* is ready to receive a message, the interface's function that checks for an incoming message will be invoked, followed by *Comm's*, and the appropriate protocol's shared and identically named functions; like the *UpdateMessageLog* method, the module id is utilized to check the correct message queue. An *EmptyFlag* variable is then returned via the interface. The

message type of the next message in the queue can be identified using *CommInt's GetNextMessageType* which will call *Comm's* as well as cross-protocols' shared and identically named functions. The message type is returned to create a correct object instance with the first message in the queue by calling *CommInt's* message retrieval method which will call *Comm's* as well as cross-protocols' shared and identically named functions. The message will then be retuned via the interface. The sequence diagram for the receive operation is shown in Figure 11.

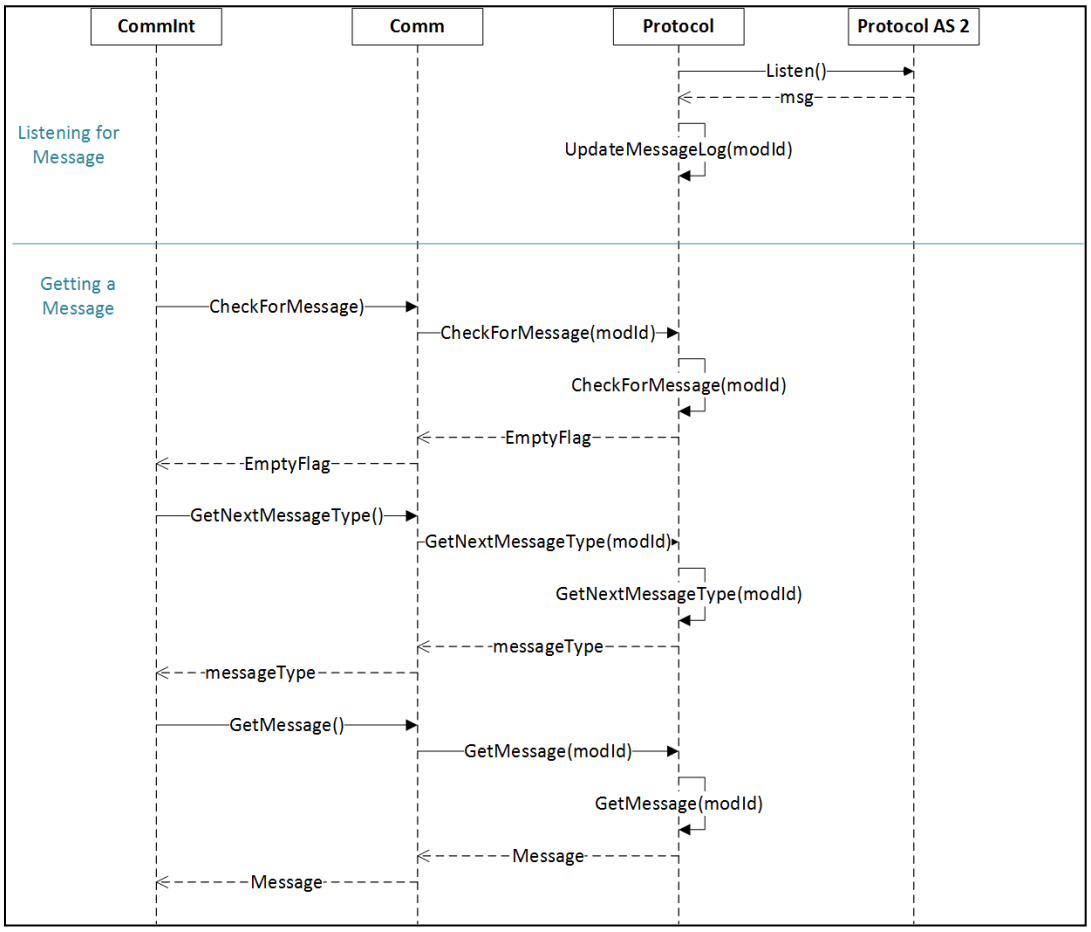

Fig. 11. Receive High-Level Sequence Diagram.

3.7 Layer Benefits

The Communication Layer architecture presented was designed to facilitate the collaboration of autonomous systems, particularly autonomous vehicles. To support reusability, the layer provides the capability to use multiple types of communication protocols. To unburden the user, the implementation of protocol specific communication and routing of messages were encapsulated in underlaying architecture layers not accessible by the *Autonomy developer*. The API designed to offer high-level abstractions for communication also allows for easy integration to existing framework architectures. Prompting the user to develop and structure the messages to be exchanged provides application flexibility. Autonomous system developers can therefore use this layer to achieve reliable collaboration for a wide range of applications, using multiple communication protocols depending on hardware capabilities.

CHAPTER 4

SOFTWARE DESIGN

Chapter 3 presented the communication layer model. This chapter details a software design to realize the model. The integration of the communication API with the T&E framework API is first detailed, followed by a discussion of the classes that compose the communication layer. The process of distinct protocol implementation is then detailed. Finally, two examples of communication media implementation – ROS and Bluetooth, are presented. The design is object-oriented and implemented using C++ in a Linux environment. The implementation is integrated with the API and node structure developed in [4].

4.1 Extending the API [4] to Include the Communication API

As mentioned in previous chapters, the communication layer and API are meant to be an extension of a T&E autonomous system framework and API [4]. The communication API must therefore be integrated into the framework API to provide communication capabilities to all AS and framework modules. The modularized design of the communication API thus enables an effortless integration to the framework.

4.1.1 Framework Application Programming Interface

The T&E framework [4] API is defined by a class *Node*, which encapsulates the behaviors for managing internal communication and processing. Its purpose is to automate AS internal communication while providing access to connect objects and functions that define a *Node's* state and behavior. All developer classes (i.e. the autonomous modules such as Sense, Plan, Act)

requiring access to the API must inherit *Node*, thus inheriting the API functions. The core API functions implemented by the *Node* class are listed in Table 4.

TABLE 4

API FUNCTIONS [4]

Function	Behavior
Initialization	Creates node and performs node initialization
Advertise	Advertises a topic and connects data for sending to the topic
Subscribe	Subscribes to a topic and connects data to receive
Notification	Connects function to be notified upon receiving from a topic
Publish	Sends data connected to topic for publishing
Callback	Receives data connected to subscribed topic

The *Initialization* function is used to create the *Node*, connect it to the framework, and set any initialization parameters. All subsequent functions implement the autonomous system's internal communication – i.e. the communication between the autonomous system's modules. Internal communication follows a publish-subscribe scheme which is currently achieved through ROS, though can be replaced by other protocols such as AMQP [31] without changes to the interface. The *Advertise* function initializes a topic – a named bus used to exchange data between modules, connecting the data that will be sent using that topic. *Subscribe* connects to a topic to

receive data, while *Notification* connects a function to be notified in the event that data is received. *Publish* and *Callback* respectively send and receive data to and from the topic.

4.1.2 External Communication Application Programming Interface

The external module-to-module communication API is implemented using the *CommInt* class, discussed in Section 3.6.2. *CommInt* allows the user to send, check for, and get a message. Currently the communication layer only allows for the autonomous software to retrieve messages using a polling approach – i.e. the software periodically checks if there is a message to be handled. Future work should also implement functionality to allow for the use of a callback approach, which would allow messages to be handled as soon as they are received. The API's main methods and their functionalities are listed in Table 5.

TABLE 5

COMMUNICATION API FUNCTIONS

Function	**Behavior**
Send	Sends a user-defined message to the destination chosen
CheckForMessage	Check if there is a message received that needs to be handled
GetNextMessageType	Get the type of the next message in the queue
GetMessage	Get a message to handle
setMsgFcnPtr	Receives and sets a pointer to a function that creates a message object given an id representing a message type
Serialize	Serializes message object data into an integer data buffer
DeSerialize	Deserializes an int data buffer into message object data
GetSize	Returns the number of integers needed to represent message object data

The *Send* function receives as parameters the user-defined message to be sent, as well as the destination of the message. The layer supports both point-to-point and broadcast communication, so the destination is either the "friendly" name of the AS the message will be delivered to, or "All" to signify that the message should be broadcast to all autonomous systems. A 0 will be returned if the send operation was unsuccessful, while 1 is returned if the message was delivered. The *CheckForMessage* function checks if there is an incoming message for the AS module to handle, true is returned if there is, false if there is not. The type of the next message in the queue can be identified using the *GetNextMessageType* method. The *GetMessage* function can in turn be called which returns the user-defined message to be handled. The *setMsgFcnPtr* function receives as a parameter a function pointer pointing to an autonomous developer implemented function, which given an integer corresponding to a message type, creates and returns an object of that message type. This function only needs to be called once at the beginning of the autonomous software implementation.

All aforementioned functions are called by the *Autonomy Developer* as needed, without being implemented or modified by the autonomous software developer. In order for the communication layer to support the capability to transmit various messages, the autonomous software developer must implement five functions that aid in translating message object data to and from integer representations of that data; these functions are *Serialize*, *DeSerialize*, and *GetSize,* and *Clone* and are all part of the API. Additionally, the autonomous software developer must implement the function that given an integer type that indicates the user-defined message

type, returns a message object associated with that integer. The function pointer that is given as a parameter to the *setMsgFcnPtr* function points to this function.

4.1.3 Extending the Framework Interface

API to API integration is very simple due to their modular designs. The methods that comprise the communication API described in Section 4.1.1 are merely transferred to become part of the framework API, i.e. part of the *Node* class. Communication initialization is accessed by the *Initialization* function of *Node*. The extended *Node* class can be seen in Figure 12. The additional methods that give access to communication are under the dotted line, while the *Communication* and *Message* classes represents the implementation to which those methods provide access to.

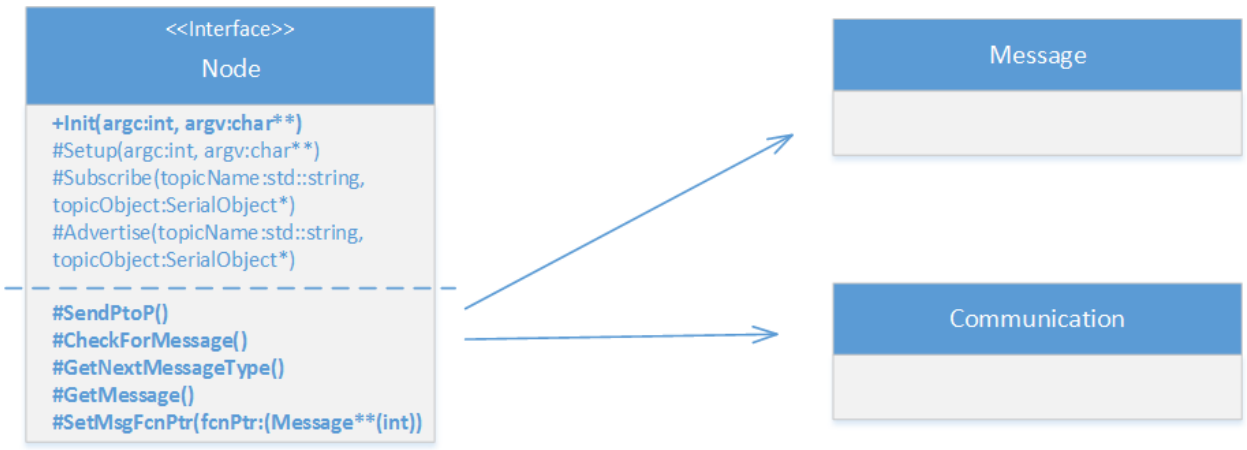

Fig. 12. Extended API Architecture.

4.2 Class Implementations for Communication

While the API supplies the user with access to communication, the communication functionality is implemented in the remaining layer classes shown in Figure 8. Inheritance is utilized extensively in this design to perform encapsulation and ensure modularity. Each communication process (i.e. Initialization, Transmitting, Receiving, etc.) is achieved using methods from all layer classes.

4.2.1 Comm Class

The *Comm* class's purpose is to configure and initialize communication parameters, as well as execute the processes supported by the layer. The class is implemented as a singleton [34], ensuring that only one instance of *Comm* can be instantiated. This provides a single object to manage all message traffic, to include routing of all incoming and outgoing messages. The class diagram for Comm is shown in Figure 13.

Fig. 13. Comm Class Diagram.

The functionality of the attributes and methods defined in *Comm* is as follows:

- *commTable* – A two-dimensional table denoting the communication protocol that should be used for all combinations of AS-to-AS or AS-to-Virtual Environment communication. Rows denote the source AS id and columns denote the destination AS id
- *nameIdMap* – Container mapping AS "friendly" names to AS ids
- *commPtrs* – Container of protocol class instances that are utilized to perform communication for a specific protocol
- *GetInstance()* – Returns a singleton instance of *Comm* class
- *Init()* – Initializes and populates attributes facilitating communication (i.e. *commTable* and *nameIdMap* and container of protocol pointers *commPtrs*)
- *SendPtoP(msg: Message *, dest: string)* – Sends a user-defined message to the destination denoted by the string *dest* and returns a flag denoting success or failure
- *SendBd(msg: Message *)* – Broadcasts a user-defined message to all autonomous systems specified in the configuration files and returns a flag denoting success or failure
- *CheckForMessage(modId: int)* – Checks if there is a message for the module specified by *modId*; returns true if there is and false if there is not
- *GetNextMessageType(modId : int)* – Returns the type of the next message of which the destination module was specified by *modId*
- *GetMessage(modId: int)* – Returns message of which the destination module was specified by *modId*

The *Init* function performs all necessary initializations by reading the configuration files to populate the communication table *commTable*, and AS friendly names to ids map *nameIdMap*. Depending on the number of communication protocols to be used, one instance for each protocol

class will be contained in *Comm*. The same protocol instance will be used to perform all communication required for the respective communication medium. This is the reasoning for the use of a singleton. Having one instance of *Comm* prohibits the redundant creation of multiple protocol-specific instances that perform the same processes.

The *SendPtoP* function is called when a point-to-point message needs to be sent. Using the "friendly" name destination and the *commTable*, the function finds which communication protocol should be utilized and continues the sending process by calling the protocol class's respective function. Similarly, the *SendBd* function iterates through a list of autonomous systems to send each AS the desired message. The protocol specific send functions return a 1 for success and 0 for failure to deliver the message. Due a small degree of unreliability associated with protocols such as Bluetooth, three attempts at transmission will occur for both transmission modes if the send operation continues to fail; after three attempts have been unsuccessfully conducted the respective function will notify the user of this failure by returning a 0, or of the success by returning a 1. This is a simple approach to try to mitigate transmission failure, if the Communication Driver Developer desired to ensure an 100% reliable protocol implementation, a more sophisticated approach could replace the simplistic one presented.

As with *Node*'s (the API's) *CheckForMessage*, *Comm*'s identically named *CheckForMessage* method acts as a middleman between the API and *BaseComm* where incoming messages are stored until retrieval. The function takes as a parameter a unique id representing an AS or virtual environment module (*modId*) and will return true if there is a message in that module's queue, and false if there is not. GetNextMessageType type will return the type of the next message in the queue specified by modId. *GetMessage* will in turn return a user-defined message if one exists within the queue of the module represented by *modId*.

4.2.2 BaseComm Class

The *BaseComm* class's main purpose is to act as an interface to the protocol specific class implementations. In addition, the class contains the backlog of messages that need to be handled as a static attribute so that all implementations have access to it. The messages are split into different queues, with one queue for each autonomous system's modules. A representation of these queues can be seen in Figure 14.

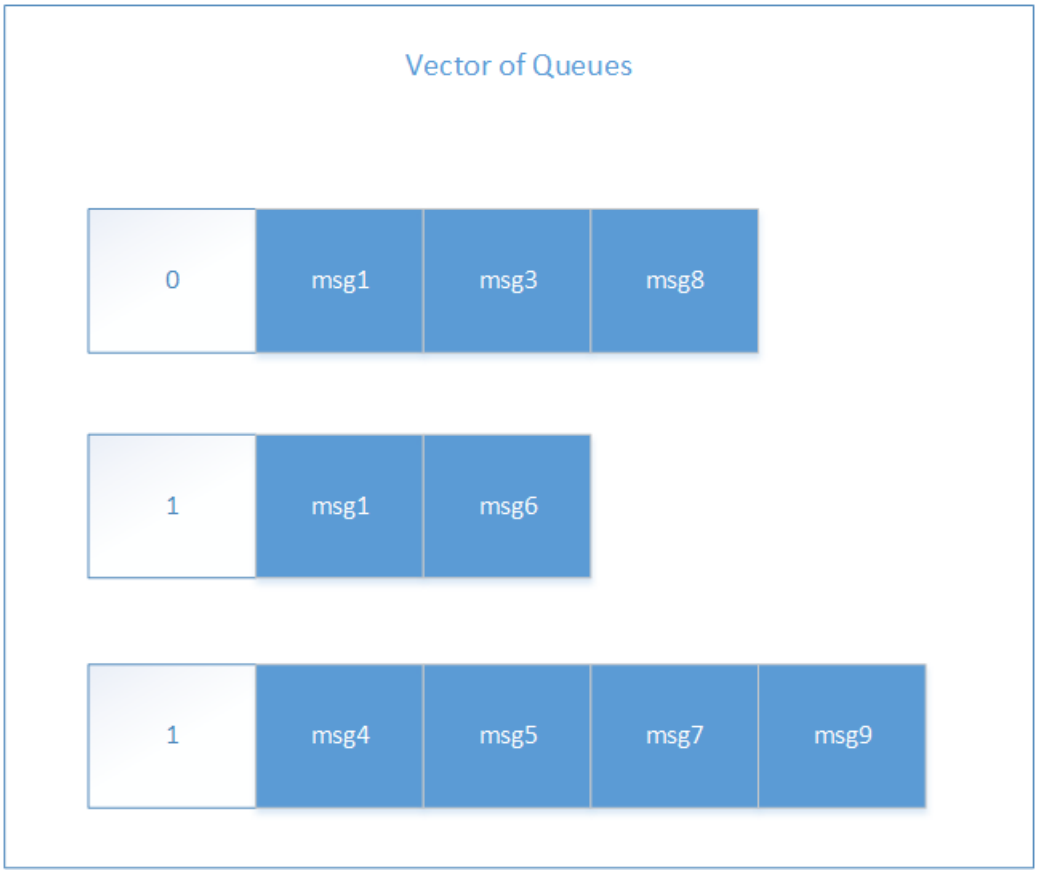

Fig. 14: Message Queue Representation.

The *BaseComm* class diagram is presented in Figure 15. As the class is meant to be an interface for subsequent communication protocol classes, two virtual methods are defined to be implemented by the child classes: *Setup* and *SendPtoP*. Once implemented, *Setup* will perform all the necessary initialization for the communication protocol, while *SendPtoP* will handle protocol-specific message transmission.

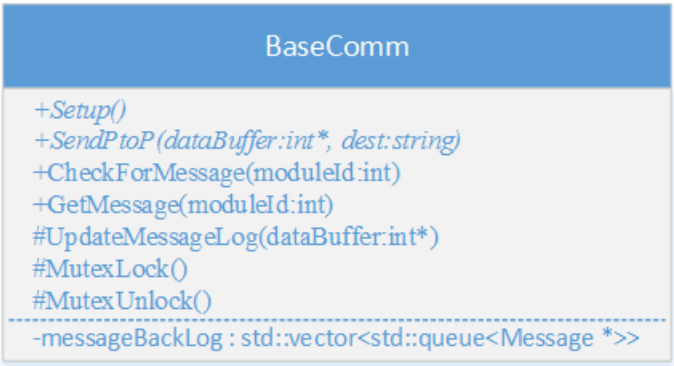

Fig. 15. BaseComm Class Diagram.

The remainder of the class functions handle the insertion and removal of messages from the appropriate queue of the message backlog - *messageBackLog*. Depending on the communication protocols utilized, a multithreading approach might be necessary to ensure that the layer can send and receive messages concurrently while also not interfering with the execution of the autonomous software. As this could lead to errors if two threads are trying to modify the same queue, the use of mutexes is required. In order to not handicap all queues when only one queue is modified, each message queue is associated with a separate mutex. Mutexes are locked using the

MutexLock function and unlocked using the *MutexUnlock* function; both receive as a parameter which module queue should be locked or unlocked. The *UpdateMessageLog* handles message insertions to the appropriate queue. The function receives the serialized message - in the form of an integer data buffer – and the module it is destined for, inserting it to the appropriate queue. *CheckForMessage* receives as a parameter the module whose queue it should be checking, if the respective queue is empty the function will return false, otherwise true will be returned – meaning there is a message that can be extracted from the queue. If there is a message to be extracted the *GetNextMessageType* that also receives the module id as a parameter will return the type of the next message in that module's queue. The *GetMessage* in turn removes and returns a message from the appropriate queue, which is selected via the function parameter. All three aforementioned functions utilize the *MutexLock* and *MutexUnlock* functions to ensure that only one thread has access to the queue when it is being modified.

It is recognized that some messages might be of high importance and that waiting in the log may be undesirable or detrimental to AS operation. Currently, all messages are handled in the order they are received; in future work, it might be useful to implement message priorities, allowing high-priority messages to bypass the message queues.

4.2.3 Message Class

The *Message* class acts as an interface for user-defined messages. As mentioned in Section 3.6.2, to successfully route messages, a layer header is attached to each user-defined message. The header structure and expected data can be seen in Figure 9. The *message id* is represented as a tuple, with the first element being the AS id, and the second element being a message id unique within the AS. Similarly, the *source* and *destination* are represented as tuples, with the AS id as

the first element and the module id as the second element. The *communication type* is a character specific to a protocol, while the *message type* and *size* are represented as integers. The Message class therefore contains methods to set and get the different header attributes. The class structure for the Message class is provided in Figure 16.

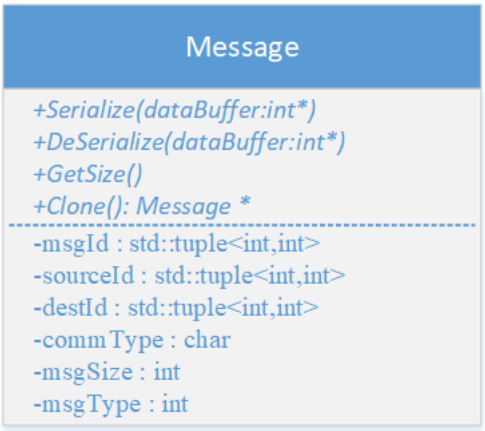

Fig. 16. Message Class Diagram.

To successfully transmit messages that can be reconstructed in the memory of another AS, message object data must be serialized and deserialized to and from a buffer of integers. Thus, additionally to the setter and getter functions for the layer header attributes, *Message* defines virtual functions to be implemented by the class's subclasses. These functions are *Serialize*, *Deserialize*, *GetSize*, and *Clone*.

Serialization is the process of translating an object's attributes into an ordered container of a single type, in this work's case an integer. The *GetSize* function is utilized to calculate and return the size of that container, which will be referred to as a data buffer. The *Serialize* method receives the data buffer as a parameter and populates it with integer data representing the object's attributes.

Similarly, *Deserialize* also receives an already populated data buffer as a parameter, and "unpacks" or deciphers the data in the buffer to the corresponding object's attributes. The *Clone* method in turn allows for the cloning of a message to another message of the same type.

Implementing Message's virtual functions in the subsequent subclasses is the most involved piece from the AS developer's perspective. Assigning data management to the AS developer allows for "typeless" communication of data buffers and enables the Comm class to remain independent of the application's structure and data types. The Message class is sufficiently abstract to allow the autonomous software developer to serialization/deserialization is deep or shallow. Deep serialization/deserialization would involve including objects/variables that are referenced or pointed to by the message in the data buffer.

4.2.4 Development of Protocol-Specific Classes

Protocol-specific communication is implemented using classes that inherit from the BaseComm class. Functionality is achieved by populating *BaseComm*'s virtual functions *Setup* and *SendPtoP* along with any other necessary methods. Along with implementing these classes for each supported protocol, the Communication Driver Developer also must modify the system and communication configuration files in order to reflect these additional capabilities. The process to implement a new protocol followed by the Communication Driver Developer is:

- *Inherit from BaseComm* – Create a protocol-specific class that inherits from *BaseComm*
- *Edit Configuration Files* – Edit the configuration file denoting which protocol should be utilized to conduct specific AS-to-AS communication to include this newly created protocol. This should be completed by the Framework Manager before full integration to the layer

- *Populate Setup* – Implement the virtual method *Setup* to handle any initialization needs pertaining to that protocol (i.e. map Bluetooth or TCP/IP addresses)
- *Populate SendPtoP* – Implement the virtual method *SendPtoP* to achieve the transmission of a message utilizing the new protocol

4.2.5 User-Defined Message Classes

The *Autonomy Developer* defines message types by implementing classes that inherit from the *Message* class. Along with the user-defined attributes and methods that the AS developer wants to implement, the virtual functions defined in *Message* – *Serialize*, *DeSerialize*, *GetSize*, and *Clone* – must be populated. The process to define a new message is:

- *Inherit from Message* – Create a protocol-specific class that inherits from *Message*
- *Implement GetSize* – Populate the *GetSize* function to return the summed integer size of all class attributes
- *Implement Serialize* – Populate the Serialize method to translate object attributes into integers and insert them in the data buffer provided as a parameter
- *Implement DeSerialize* – Populate the DeSerialize method to translate the integer data buffer provided as a parameter to object attributes
- *Implement Clone* – Populate the DeSerialize method to correctly "unpack" the integers contained in the data buffer to their corresponding object attributes

4.3 Example Protocol Implementations

The Communication Layer's main purpose is to provide communication capabilities for any number of communication protocols. As detailed in the previous section protocol-specific

functionality can be provided using a distinct class and modifying the configuration files to reflect this extension. Two protocols (ROS and Bluetooth) were implemented to showcase this capability, their implementation process is highlighted below.

4.3.1 Bluetooth

As mentioned in Section 3.5, for the layer to support Bluetooth communication a configuration file matching the AS ids to Bluetooth addresses is required. Using this file and the *Setup* function Bluetooth initialization is conducted. The Bluetooth class handles initialization using the *Setup* function, which sets the AS, reads the Bluetooth address configuration file and populates a structure – *btAddresses*, that maps AS ids to Bluetooth addresses.

Due to the nature of the protocol which halts execution while waiting for a message, a separate thread is required for listening. The thread is an attribute of the class and is implemented using the function *ListeningThread*, which is spawned from the initialization function. The *ListeningThread* will continuously "listen" for a connection until program termination. It will accept a data buffer from any address, and call *BaseComm's UpdateMessageLog* to insert a received message to the appropriate queue. The class diagram for the Bluetooth protocol implementation is depicted in Figure 17, while the sequence diagram for BlueComm's listening process can be seen in Figure 18.

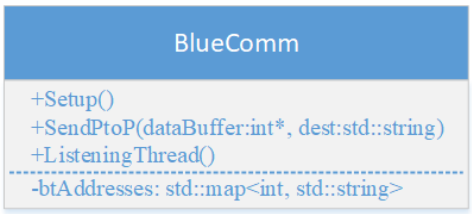

Fig. 17. Bluetooth Class Diagram.

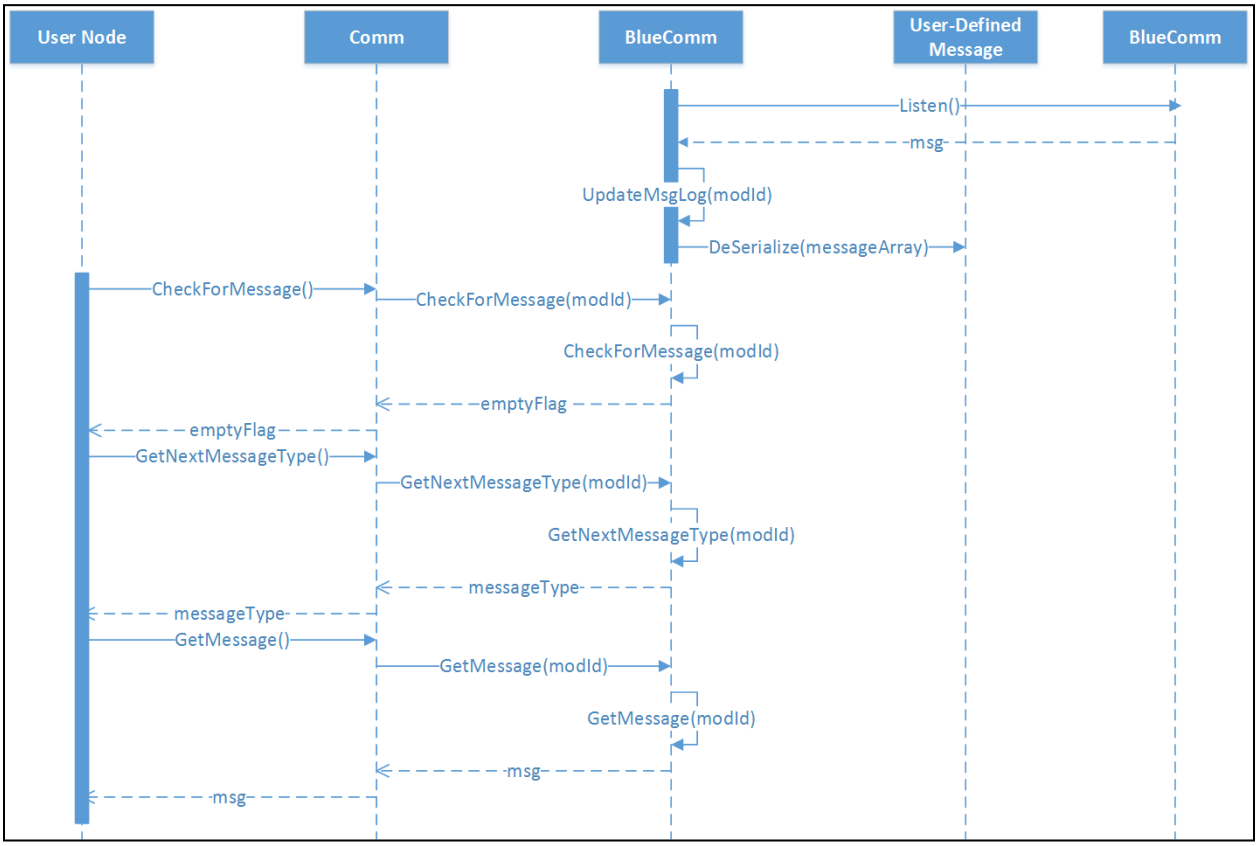

Fig. 18. Bluetooth Listening Sequence Diagram.

The *SendPtoP* function is utilized to transmit messages. The function receives as parameters the message data buffer and destination. The destination is represented by the destination autonomous system's "friendly" name. *SendPtoP* therefore identifies the Bluetooth address matching that "friendly" name in order to send the message. This process can be further studied in the sequence diagram shown in Figure 19.

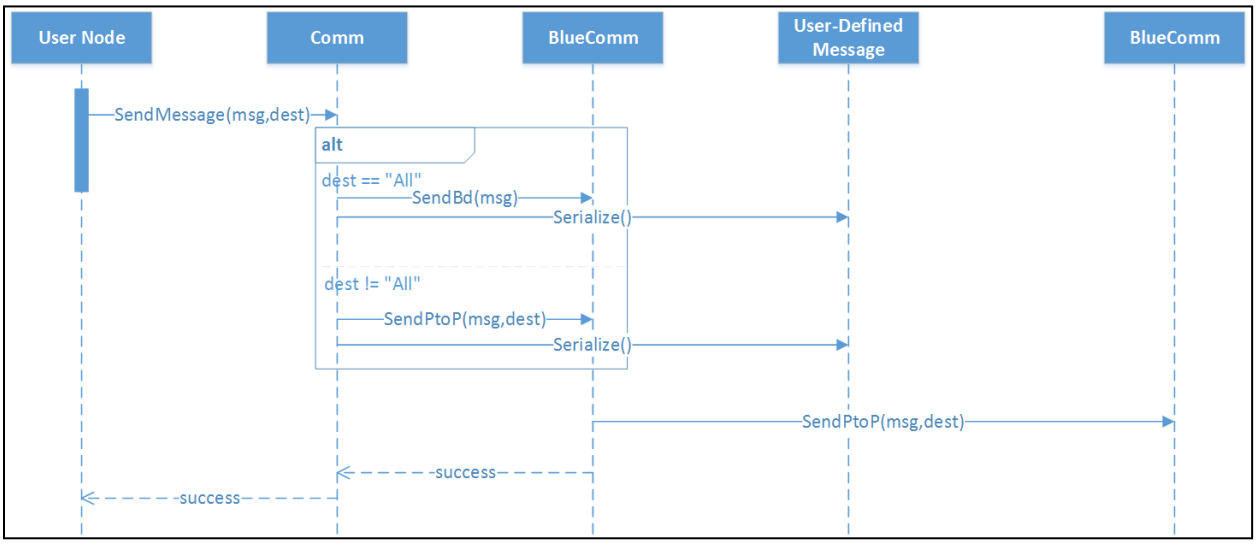

Fig. 19. Bluetooth Transmitting Sequence Diagram.

4.3.2 ROS

Communication through ROS does not require the creation of another configuration file to determine the topics. Instead, each autonomous system is associated with a topic denoted by its "friendly" name. In order to transmit a message to that AS, other autonomous systems publish messages to that topic. A different publisher is created for each topic; these are contained in the *publishers* map that maps each publisher to the id representing the topic it publishes to. Like

Bluetooth, the *Setup* function initializes ROS communication. Using a container of AS friendly names, a subscriber to the AS listening topic denoted by its "friendly" name is created, followed by the creation of Publishers for each AS.

The *MessageCallBack* function handles message insertion to the appropriate message queue, by calling *BaseComm's UpdateMessageLog* function. The *SendPtoP* function in turn receives as parameters the data buffer for the message to be sent, as well as the "friendly" name of the AS destination. The publisher pertaining to the destination AS is then called to transmit the message. The class diagram for the new class is shown in Figure 20, while the processes for sending and receiving a message using ROS are shown in Figures 21 and 22 respectively.

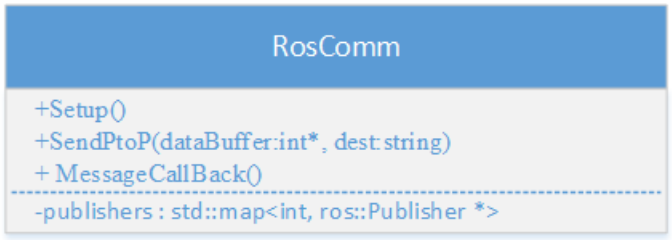

Fig. 20. RosComm Class Diagram.

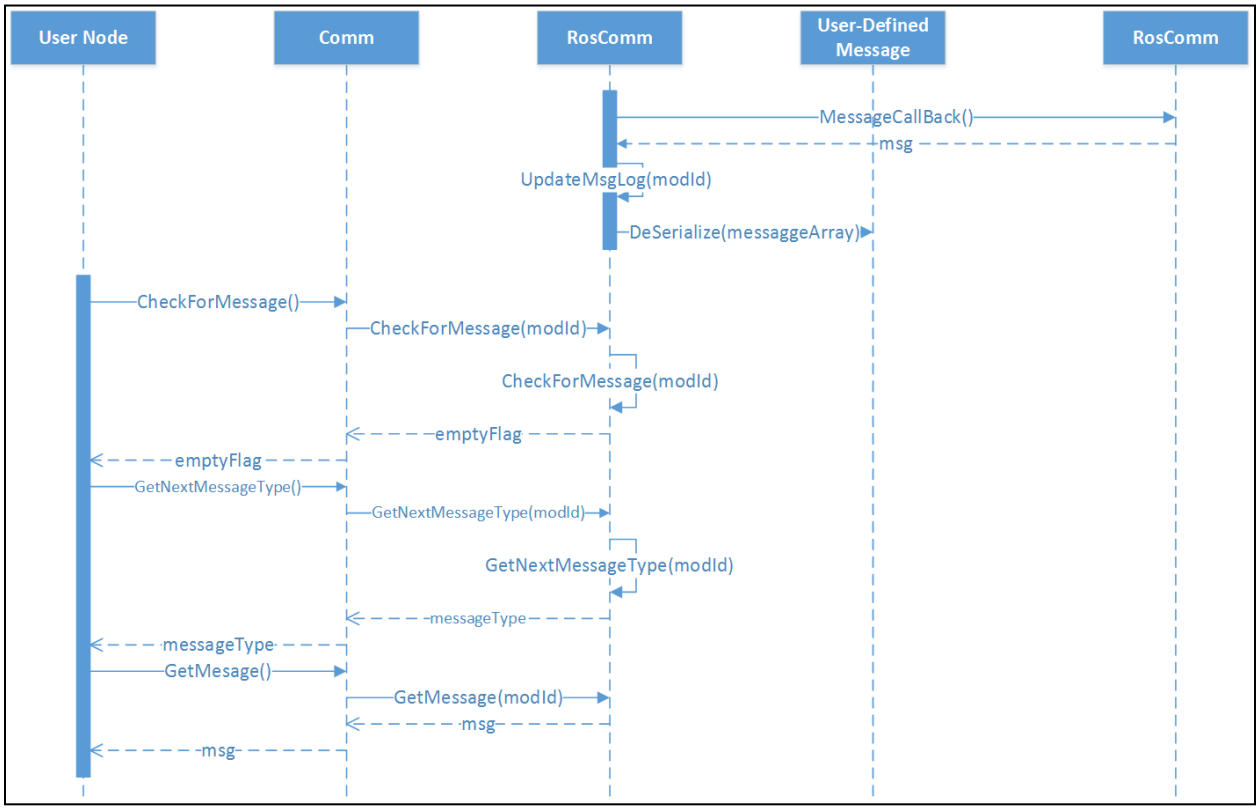

Fig. 21. ROS Receiving Sequence Diagram.

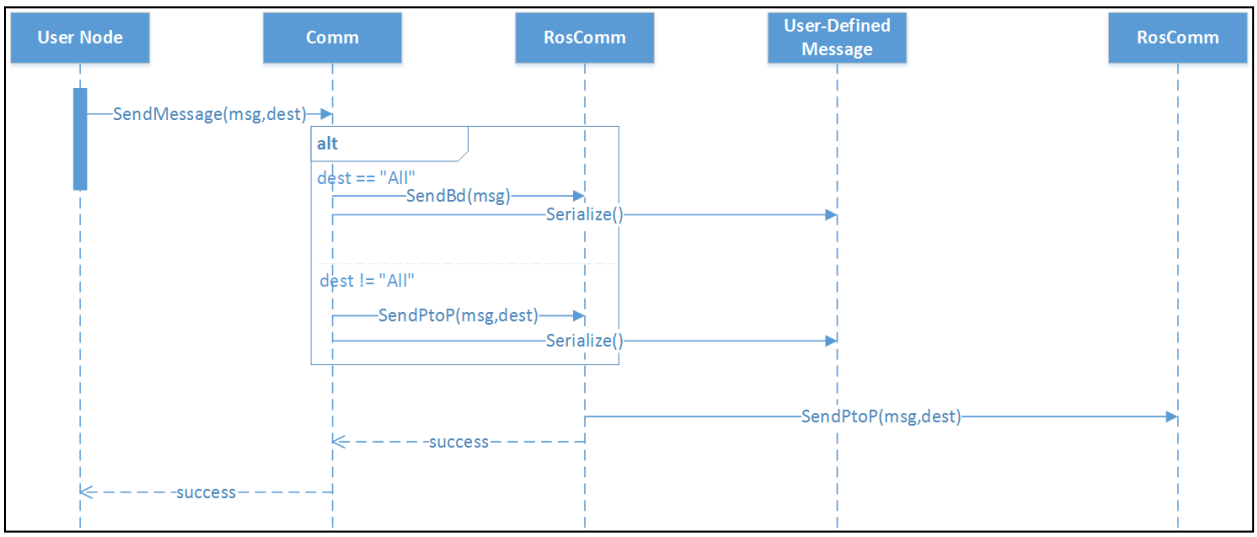

Fig. 22. ROS Transmitting Sequence Diagram.

CHAPTER 5

RESULTS

This chapter examines the development of a simple collaborative autonomous system utilizing the T&E Framework along with the communication layer extension for demonstration purposes. The process of mapping the system to the T&E Framework is discussed first, throughout which the capabilities and benefits of the layer extension are showcased. An analysis for communication reliability afforded by the communication layer is also presented.

5.1 Collaborative AS Example

To demonstrate the communication capabilities rendered by the framework extension, a simple collaborative autonomous system demonstration was developed. The encompassing system was comprised of three virtual autonomous ground vehicles denoted as *rovers*, and three virtual environments wherein the vehicles operated. The virtual environments act as data loggers, and do not impact the behavior of the rovers. The system followed a leader-follower approach, where one AS was chosen as the leader, with the remaining two systems classified as followers operating in a chain, only receiving messages from the previous *rover* in the chain. The AS behavior expected was to collaboratively perform the same movements, with the follower *rovers* moving as commanded by the leader AS. The leader AS determines which command the follower rovers must follow, sending that command to the first follower system, which in turn sends it to the second follower rover.

The communication protocols employed in this demonstration were Bluetooth and ROS. Bravo and Charlie operate as “simulated” rovers requiring wired communication – represented by

a ROS communication. Tango, however, operates as a “physical” rover requiring wireless communication – represented by a Bluetooth communication. Communication with each rover’s virtual environment requires “wired” communication – represented by communication through ROS. The rovers change their motion as directed by a command received by another AS. The command is represented by a single integer, with three possible values. A value of 0 indicates forward motion while a value of 1 indicates a left turn, and a value of 2 indicates that the rover should stop moving.

It is important to note that one virtual environment for each autonomous system was needed due to the layer’s utilization of a polling approach for message handling by the *Autonomy Developer*. This is because with the polling approach, only one message is handled at a time, whereas a virtual environment must handle location change messages from all *rovers* operating within it. An implementation of a callback approach for message handling would resolve this issue and is thus proposed for future work, which would allow one single virtual environment for all *rovers*.

5.1.1 Configuration Files

Before the autonomous software can be implemented, the required configuration files must be set by the *Framework Manager* to ensure communication functionality. First, the autonomous systems’ and virtual environments’ “friendly” names must be mapped to distinct integer ids. This configuration file is shown in Table 6, where the first row is the number of systems including the virtual environments. The corresponding communication table was realized as seen in Table 7.

TABLE 6

DEMONSTRATION ID CONFIGURATION FILE

6	
0	Alpha
1	Charlie
2	Tango
3	VirtualEnv
4	VirtualEnv2
5	VirtualEnv3

TABLE 7

DEMONSTRATION COMMUNICATION TABLE

	0	**1**	**2**	**3**	**4**	**5**
0	*X*	R	*X*	R	*X*	*X*
1	R	*X*	B	*X*	R	*X*
2	*X*	B	*X*	*X*	*X*	R
3	R	X	*X*	*X*	*X*	*X*
4	*X*	R	*X*	*X*	*X*	*X*
5	*X*	*X*	R	*X*	*X*	*X*

As previously mentioned, *R* denotes communication with ROS, while *B* denotes communication with Bluetooth, with *X* indicating where communication should not occur. The corresponding file mapping system ids to Bluetooth addresses is shown in Table 8. It is important to note that since the virtual environments do not perform communication using Bluetooth, their Bluetooth addresses are not required to be set in the configuration file.

TABLE 8

DEMONTRATION BLUETOOTH CONFIGURATION FILE

1	68:A3:C4:4A:B3:BA
2	E4:B3:18:09:09:06

5.1.2 User-Defined Messages

Two user-defined messages were developed for this demonstration. The *Command* message was realized to denote the action the leader is requesting the followers to perform. The *Wheels* message in turn was implemented to denote a vehicle's wheel actions, which were sent from all vehicles to the virtual environment so that the vehicle position can be updated.

5.1.2.1 Command Message

Following the process outlined in Section 4.2.5, the *Command* message class was defined as shown in Figure 23. The class implements the serialization and deserialization process

functions, while also providing a *GetCommand* function, which returns the integer command that a vehicle should perform. The possible command values are:

- 0 – specifies that the rover should commence full forward
- 1 – specifies that the rover should turn left
- 2 – specifies that the rover should halt all movement

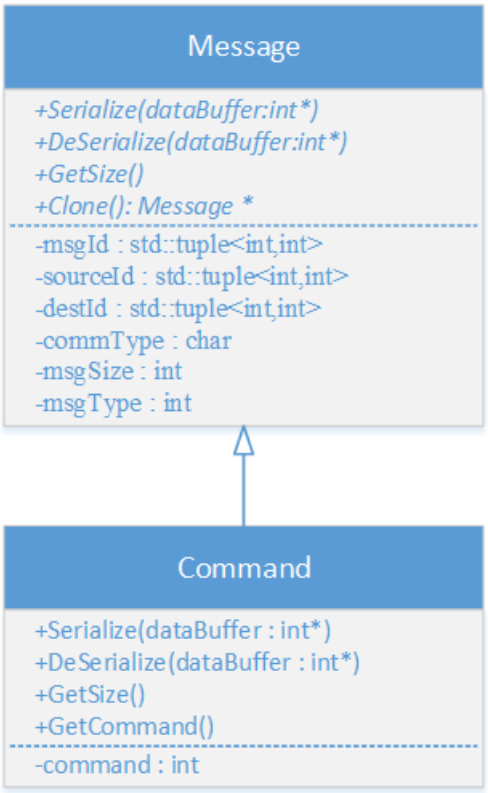

Fig. 23. Command Message Class Diagram.

5.1.2.2 Wheels Message

Similarly following the process outlined in Section 4.2.5, the *Wheels* message class was defined as shown in Figure 24. Two additional functions were developed, a *getRight* and *getLeft* function. They return the values of the of right and left wheels respectively. This message is meant to be utilized by the virtual environment to update each AS position.

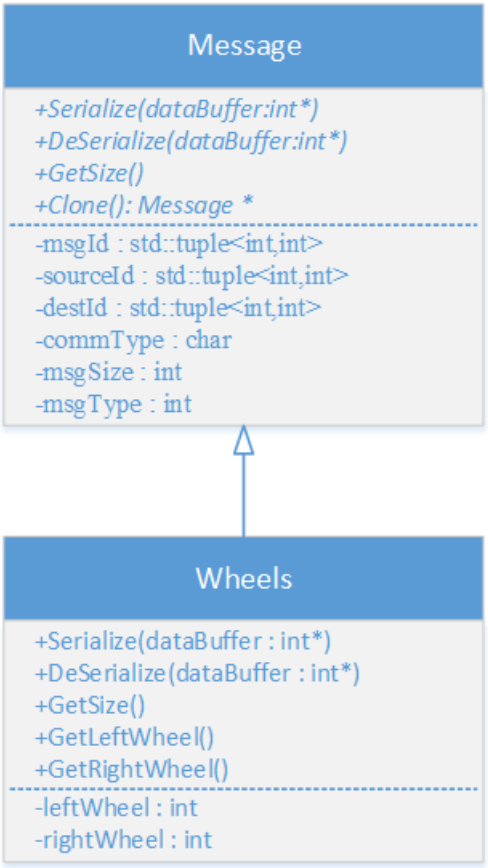

Fig. 24. Wheels Message Class Diagram.

As required by the *Message* creation process, a function was implemented that receives an integer as a parameter and returns an object of the *Command* message if the integer is 0, or the *Wheels* message if the integer is 1. A function pointer pointing to that function is then passed to the communication layer via parameter to the *SetFcnPtr* function.

5.1.2.2 Leader AS

The *leader* vehicle is responsible for continuously directing the movement of the follower vehicles along with its own. The autonomous software is composed of two modules, *Plan* and *Act*, where *Plan* performs the decision-making to select the action itself and the follower vehicles should perform, and *Act* interfaces with the virtual actuators that will carry out vehicle movement. No *Sense* module is required since the example does not require the rovers to observe their environment. The algorithm for the *Plan* module is shown in Figure 25. It is important to note that the variable timer is utilized to control when movement should be modified and is initially set to 0 in an initialization function, while the variable turn is utilized to keep track of whether a turn message has already been sent and is initially set to false.

```
while terminationSignal is false
   if timer < 5000000
      set left and right wheel to 1
      send Wheels message to VirtualEnv
      timer++
   else if timer >= 5000000 && timer < 9000000
      if turn is false
         set turn to true
         set command to 1
         send Command message to Charlie
      set left wheel to -10 and right wheel to 10
      send Wheels message to VirtualEnv
      timer++
  else if timer >= 9000000 && timer < 12000000
      if turn is true
        set turn to false
        set command to 0
        send Command message to Charlie
      set left and right wheel to 1
      send Wheels message to VirtualEnv
      timer++
  else if timer >= 12000000 && timer < 15000000
      if turn is false
        set turn to true
        set command to 1
        send Command message to Charlie
      set left wheel to -10 and right wheel to 10
      send Wheels message to VirtualEnv
      timer++
  else if timer >= 15000000 && timer < 19000000
      if turn is true
        set turn to false
        set command to 0
        send Command message to Charlie
      set left and right wheel to 1
      send Wheels message to VirtualEnv
      timer++
```

Fig. 25. Leader AS Plan Algorithm.

The values that *timer* is compared to where chosen arbitrarily. If the *timer* is less than 5000000, the rover should go straight, therefore both wheels are set to 1, and the *Wheels* message

is sent to the rover's virtual environment denoted as *VirtualEnv*, followed by an incrementation of *timer*. If *timer* is greater than or equal to 5000000 and less than 6000000, the program then checks if *turn* is false. If *turn* is false, a turn command has not been sent to the follower *rover*, the command is therefore set to 1, with the *Command* message being sent to *Charlie*. The *Wheels* message's left wheel value is then set to -10 while the right value is set to 10, with the message then being sent to *VirtualEnv*. This will cause the vehicle to perform a left turn pivot. The next two conditions directly follow the former two conditions with the only change being the values that *timer* is compared to.

5.1.2.3 Follower AS

The *follower* rover *Charlie's* main purpose is to wait for movement commands from the *leader*, forward those commands to *Tango*, and follow the commands once received. The autonomous software is composed of two modules, *Plan* and *Act*, where *Plan* deciphers the commands sent by the *leader*, forwards them to the next *follower*, and chooses the appropriate movement, while *Act* interfaces with the virtual actuators that will carry out vehicle movement. The algorithm for the *Plan* module is shown in Figure 26. It is important to note that two variables *prevL* and *prevR* are utilized to keep track of the previous values of the left and right wheels and are both initially set to 1. This allows the *rover* to keep its previous motion when a new command has not been sent.

```
while terminationSignal is false
   check whether there is a message
      if the message type is 0
         if command = 0
            set left and right wheels to 1
            send Wheels message to VirtualEnv2
            send Command message to Tango
            set prevL and prevR to 1
         if command = 1
            set left wheel to -10 and right wheel to 10
            send Wheels message to VirtualEnv2
            send Command message to Tango
            set prevL to -10 and prevR to 10
         if command = 2
            set left and right wheels to 0
            send Wheels message to VirtualEnv2
            send Command message to Tango
           set prevL and prevR to 0
   else
     set left wheel to prevL and right wheel to prevR
     send Wheels message to VirtualEnv2
```

Fig. 26. Follower AS Plan Algorithm.

While a termination signal has not been set, the software first checks if there is a message to receive. If there is a message and the message type is 0, the message command value is then checked. If the command received was 0, both the left and right wheels are set to 1 for forward motion, the *Wheels* message is then sent to the rover's virtual environment, followed by the forwarding of the *Command* message to *Tango*. *PrevL* and *prevR* are both set to 1. Similarly, if the command received was 1, the left wheel is set to -10 while the right wheel is set to 10 for a left pivot turn motion. The *Wheels* message is then sent to *VirtualEnv2* while the *Command* message is forwarded to *Tango*, and *prevL* is set to -10 with *prevR* being set to 10. However, if the command

is 2, both the left and right wheels are set to 0 to stop all motion. The *Wheels* message is then sent to the rover's virtual environment, while the *Command* message is once again forwarded to *Tango*. The variables *prevL* and *prevR* are also updated to 0. If there was no message received, the left and right wheel values are respectively set to *prevL* and *prevR*, with the *Wheels* message being sent to the virtual environment.

Tango's software algorithm is identical to *Charlie's* shown in Figure 26, except it does not forward the *Command* message to any AS, and it sends the *Wheels* messages to VirtualEnv3, i.e. its respective virtual environment. It is important to note that all rovers start moving straight as soon as the simulation begins; thus, an initial command to go straight is not needed.

5.1.2.4 Virtual Environment

The virtual environments' purpose was to monitor and display AS positions throughout the experiment. This was achieved by receiving periodic *Wheels* messages from their respective autonomous systems, which were utilized to calculate each vehicle's new position knowing its previous position.

5.1.3 Demonstration Conclusions

The demonstration indicates promise in utilizing the framework to conduct AS-to-AS or AS-to-VE communication. The successful transmission of the Command and Wheels message was studied, as well as whether there was appropriate response from receiving systems. The time-stamped location and orientation of each AS was recorded using data outputted by the virtual environments. The timestamp of when a *Command* message was sent by the *leader* (*Alpha*) was

also recorded, along with the value of the command. Similarly, the timestamp and command values were recorded each time a *Command* message was received by an AS. Tables 9, 10, and 11 show the recorded *Command* data, with the left column being the time a command was received and the right column denoting the command value. Table 12 depicts the timing that each different command was received for each of the three autonomous systems. Tables 13, 14, and 15 show the recorded location data, with columns containing the time, X, Y, and theta angle. The elements that are bold and italicized showcase where a change in one or more values should occurr due to a *Command* message. Orange highlighted blocks show where a change in angle should be observed, while blue highlighted blocks show where a change in X & Y should be observed.

TABLE 9

DEMONTRATION ALPHA COMMAND DATA

Time	**Command (Sent)**
08:36:54	1 – Turn Left
08:36:57	0 – Go Straight
08:37:00	1 – Turn Left
08:37:02	0 – Go Straight

TABLE 10

DEMONTRATION CHARLIE COMMAND DATA

Time	Command (Received)
08:36:54	1 – Turn Left
08:36:57	0 – Go Straight
08:37:00	1 – Turn Left
08:37:02	0 – Go Straight

TABLE 11

DEMONTRATION TANGO COMMAND DATA

Time	Command (Received)
08:36:56	1 – Turn Left
08:36:57	0 – Go Straight
08:37:00	1 – Turn Left
08:37:04	0 – Go Straight

TABLE 12

DEMONTRATION COMMAND TIMING

	Left Turn Command	Straight Command	Left Turn Command	Straight Command
Alpha	08:36:54	08:36:57	08:37:00	08:37:02
Charlie	08:36:54	08:36:57	08:37:00	08:37:02
Tango	08:36:56	08:36:57	08:37:00	08:37:04

TABLE 13

DEMONTRATION ALPHA LOCATION DATA

Time	X	Y	Theta
08:36:52	-190.494	-200	0
08:36:53	-177.319	-200	0
08:36:54	-155.246	-200	0.005
08:36:55	-155.246	-200	12.4502
08:36:56	-155.246	-200	61.2503
08:36:57	-155.26	-199.991	148.145
08:36:58	-157.6	-198.548	148.145
08:36:59	-168.346	-186.49	148.145
08:37:00	-179.959	-184.765	148.225
08:37:01	-179.959	-184.765	192.221
08:37:02	-179.979	-184.865	259.261
08:37:03	-181.371	-192.174	259.261
08:37:04	-183.719	-204.504	259.261
08:37:05	-186.126	-217.14	259.261

TABLE 14

DEMONTRATION CHARLIE LOCATION DATA

Time	X	Y	Theta
08:36:52	200	5.73321	90
08:36:53	200	15.7074	90
08:36:54	200	32.7212	90.005
08:36:55			
08:36:56	200	32.7212	91.37
08:36:57	199.999	32.7223	127.23
08:36:58	199.252	33.7033	127.23
08:36:59	191.85	43.4176	127.23
08:37:00	185.704	51.4844	127.24
08:37:01	185.704	51.4844	161.29
08:37:02	185.703	51.4835	222.206
08:37:03			
08:37:04	184.657	50.5424	222.206
08:37:05	176.74	43.3959	222.206

TABLE 15

DEMONTRATION TANGO LOCATION DATA

Time	X	Y	Theta
08:36:52	191.495	60	180
08:36:53	186.906	60	180
08:36:54	179.16	60	180
08:36:55	174.36	60	180
08:36:56	173.141	60	180.005
08:36:57	168.589	59.9996	196.135
08:36:58	164.879	58.9251	196.135
08:36:59	159.99	57.5111	196.135
08:37:00	158.474	57.07	196.14
08:37:01	158.474	57.07	209.38
08:37:02	158.474	57.07	226.901
08:37:03	158.474	57.07	244.655
08:37:04	158.482	56.8882	272.683
08:37:05	158.736	51.449	272.683

As can be seen in Tables 9 through 15, all Command messages transmitted are successfully received. It can also be inferred that the Wheels messages are being transmitted successfully due to the continuous change in data even when not all the points are listed. The response and delivery time afforded by the framework seem sufficient for this application. *Bravo* and *Charlie* show same

second delivery time, with equal response time. Delivery time from *Charlie* to *Tango,* however, ranges from same second to a two second delay. This could be the result of various reasons. A small delay could be attributed to the fact that *Charlie* sets and sends its wheel information before sending the command to Tango. Additionally, the communication between *Tango* and *Charlie* is conducted using Bluetooth, which can sometimes require more than one attempt at successful transmission. Charlie is also missing two seconds of data; it is inferred that the time delay associated with additional attempts at transmission resulted in no wheel information being sent to the environment during those seconds. The graphs for each rover's X,Y positions and orientation angles can be studied in Appendix A.

5.2 Communication Reliability

The communication layer's reliability for the two protocols implemented is addressed in this section. The purpose is to both demonstrate that the communication layer supports introducing protocols to improve message reliability and the inherent reliability of the communications implemented. Message transmission and delivery between three systems, all utilizing both protocols, was recorded. Each system sent a broadcast message, and five point-to-point messages (the destination of which was randomly selected). The experiment was conducted 20 times.

5.2.1 Configuration

Before the software for the reliability analysis could be implemented, the configuration files were set. Three systems were utilized, their "friendly" names being *Bravo, Charlie,* and *Delta*; the configuration file mapping the "friendly" names to distinct ids is shown in Table 16.

TABLE 16

RELIABILITY ANALYSIS ID CONFIGURATION FILE

3	
0	Bravo
1	Charlie
2	Delta

The chosen communication protocol for each system to system communication was arbitrarily chosen, ensuring that all systems perform communication using both the ROS and Bluetooth protocols. The communication table configuration file is shown in Table 17. The corresponding file mapping AS ids to Bluetooth addresses is shown in Table 18.

TABLE 17

RELIABILITY ANALYSIS COMMUNICATION TABLE FILE

	0	**1**	**2**
0	*X*	R	B
1	B	*X*	R
2	R	B	*X*

TABLE 18

RELIABILITY ANALYSIS BLUETOOTH ADDRESS CONFIGURATION FILE

0	68:A3:C4:4A:B3:BA
1	E4:B3:18:09:09:06
2	4C:ED:DE:9E:39:10

Two user-defined messages were developed to ensure successful transmission of differing message types. Their class diagrams are shown in Figures 27 and 28. As required by the *Message* creation process, a function was implemented that receives an integer as a parameter and returns an object of the *Message1* message if the integer is 0, or the *Message2* message if the integer is 1. A function pointer pointing to that function is then passed to the communication layer via parameter to the *SetFcnPtr* function. The algorithm for all systems is identical, with the only changes being the two destinations randomly selected. This can be studied in Figure 29.

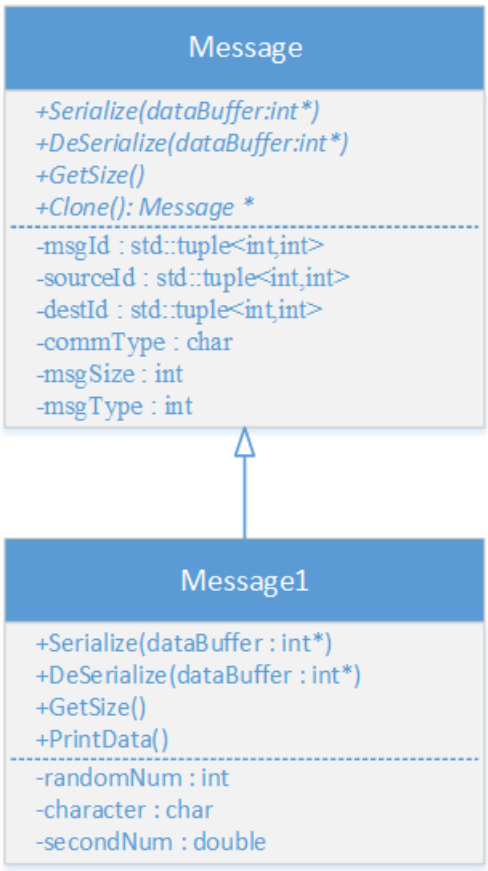

Fig. 27. Message1 Class Diagram.

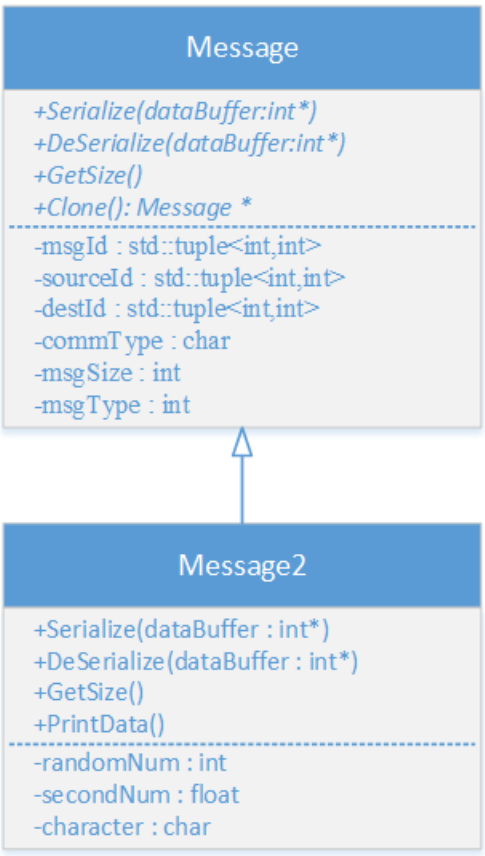

Fig. 28. Message2 Class Diagram.

```
initialization
create and initialize Message1
send broadcast of Message1
delete Message1
set counter = 0
create and initialize Message2
while counter < 5
  randomly select destination
  send point-to-point of Message2 to destination
  counter = counter + 1
delete Message2
```

Fig. 29. Reliability Analysis Algorithm.

5.2.2 Reliability Analysis Conclusions

As mentioned previously, 20 trials were conducted to analyze the transmission of messages utilizing the communication layer. For each trial, each of the three autonomous systems sent 7 messages, 5 point-to-point messages, and one message in broadcast form. The data collected is summarized in Tables 19 and 20. The total of each system's attempts, successful transmissions, and receival of messages for each of the two protocols are shown in Table 18, while the secondary transmission attempts are shown in 20.

TABLE 19

RELIABILITY ANALYSIS MESSAGE STATISTICS

	ROS Transmit Attempts	Bluetooth Transmit Attempts	ROS Transmit Successes	Bluetooth Transmit Successes	ROS Receivals	Bluetooth Receivals
Bravo	65	75	65	74	60	69
Charlie	69	71	69	69	60	82
Delta	54	86	54	82	68	74

TABLE 20

RELIABILITY ANALYSIS SECONDARY ATTEMPTS

	ROS Additional Attempts	Bluetooth Additional Attempts
Bravo	0	5
Charlie	0	7
Delta	0	13

A total of 420 messages were sent, but 413 were received, showing a 98.33% transmission success. All unsuccessful transmissions occurred when using the Bluetooth protocol, which shows that setting the number of attempts to three if a transmission was unsuccessful does not necessarily result in 100% successful communication. It does, however, lead to improved reliability. This is because the total additional attempts is not divisible by three, meaning that a first or second attempt led to the message being delivered. More extensive testing in different scenarios must be conducted in order to fully gauge the communication layer's reliability.

5.3 Communication Layer Impact

Both experiments only required the *Autonomous Software Developer* to implement the four functions pertaining to serialization/deserialization (Serialize, DeSerialize, GetSize, and Clone), as well as a function that given an integer id returns an object of the type that corresponds to this id, which is then populated by the incoming message. The *Autonomous Software Developer* was also required to set a function pointer to point to that function and pass it to the framework through a call to *setMsgFcnPtr*. The *Autonomous Software Developer* was then free to use the communication functionality as needed, while being shielded from the communication implementation. The *Communication Driver Developer* was in turn only responsible for developing protocol specific implementations, while the *Framework Manager* was responsible for modifying the configuration files as needed. The reconfiguration of configuration files or the addition of another protocol implementation are events the autonomous software and *Autonomy Developer* are oblivious to.

CHAPTER 6

CONCLUSIONS

The communication layer detailed in this thesis was designed to meet the communication needs of collaborative autonomous systems throughout their development cycle. By utilizing this layer in conjunction with the T&E Framework presented in [4], the development and testing of collaborative systems can be enhanced. The ability to concurrently utilize multiple communication protocols enables the communication between systems operating at any part of the virtuality-reality spectrum, thereby providing the ability to begin testing at the early stages of development. With the architecture closely following the layers of the OSI model [26], compatibility with many robotic applications can be expected. Communication capabilities are provided to all autonomous software modules, with the implementation of clear interfaces isolating the autonomous developer from the intricacies of specific protocol implementations. By requiring the definition of user-defined messages the transmission of various message types by the layer is ensured; this grants the autonomous developers a great degree of flexibility in the data that can be communicated. The support of different communication protocols, along with a well-defined process for additional implementations enables flexibility in hardware selection. Additionally, the communication layer provides the ability to easily reconfigure communication for testing to meet the changing communication needs as the operating environment moves through the levels of the virtuality-reality spectrum; this is achieved without any change to the autonomous software between reconfigurations.

In order to demonstrate the communication capabilities afforded by this layer a leader-follower navigation application was developed. The application not only illustrated the capability

to concurrently collaborate utilizing different communication protocols but also the ease of use of the functionalities provided. The autonomy developer only needs to provide the ability to serialize and deserialize message. At that point, the underlying communication configuration can be modified to meet the current state of testing in the virtuality-reality section. A reliability analysis of message transmission between three systems using two distinct protocols demonstrated the message transmission reliability that can be guaranteed by the communication layer when the development process is followed.

6.1 Future Work

While the layer developed demonstrates a valid proof of concept there are several improvements that could enhance communication capabilities. Communication speed can be improved by introducing another execution thread for sending messages. This would also allow the autonomous software to continue its execution and not be impeded by message transmission. A notable addition to the communication layer would be the implementation of a callback approach to message handling. This would grant autonomous developers the ability to immediately process messages if necessary. A priority could also be added to messages to allow them to bypass the message queues. Additionally, these extensions would allow the developer a choice in selecting which message processing approach would be most beneficial to their application (polling or callback). A time-stamp parameter may also be incorporated in the message header, which can in turn be utilized to ensure that messages are stored in the order they were intended to be received. An analysis of the delay associated with message delivery would also be beneficial in determining whether the communication layer can be utilized in applications where time is critical. Finally,

conducting additional testing with more sophisticated and intensive applications could examine the robustness of the encapsulating T&E framework expanded by this research.

REFERENCES

[1] A. Owen-Hill, "Ten Emerging Applications in Autonomous Logistics", Blog.robotiq.com, 2018. [Online]. Available: https://blog.robotiq.com/10-emerging-applications-in-autonomous-logistics. [Accessed: 05- Nov- 2019]

[2] R. Fierro et al., "A Framework and Architecture for Multi-Robot Coordination", The International Journal of Robotics Research, vol. 21, no. 10-11, pp. 977-995, 2002. Available: https://journals.sagepub.com/doi/abs/10.1177/0278364902021010981. [Accessed 5 November 2019].

[3] Y. Xie, H. Zhang, "Collaborative Merging Behaviors And Their Impacts On Freeway 1 Ramp Operations Under Connected Vehicle Environment 2 3 4 5 6", Symposium Celebrating 50 Years of Traffic Flow Theory, 2014. Available: https://www.semanticscholar.org/paper/COLLABORATIVE-MERGING-BEHAVIORS-AND-THEIR-IMPACTS-1-Xie-Zhang/7b828e91ca3b0b4661f0f52f12f2d7024dbab50d. [Accessed 6 July 2019]

[4] N. Gonda, "A Framework for Test & Evaluation of Autonomous Systems Along the Virtuality-Reality Spectrum.", *Master of Science (MS), thesis, Modeling Simul & Visual Engineering, Old Dominion University, 2019*. [Online]. Available: https://digitalcommons.odu.edu/msve_etds/47. [Accessed: 05- Jun- 2019]

[5] Milgram, P., Takemura, H., Utsumi, A., & Kishino, F. (1994). Augmented Reality: A Class of Displays on the Reality-Virtuality Continuum. SPIE Telemanipulator and Telepresence Technologies, Vol. 2351, p. 282-292.

[6] M. Salem, "What is an "Autonomous System?", Udacity, 2018. [Online]. Available: https://blog.udacity.com/2018/09/what-is-an-autonomous-system.html. [Accessed: 05- Jun- 2019].

[7] "Unmanned Ground Vehicle", *En.wikipedia.org*, 2019. [Online]. Available: https://en.wikipedia.org/wiki/Unmanned_ground_vehicle. [Accessed: 08- Jul- 2019].

[8] "Unmanned Surface Vehicles USV | Unmanned Marine Systems | L3 ASV", Unmanned Systems Technology, 2019. [Online]. Available: https://www.unmannedsystemstechnology.com/company/ autonomous-surface-vehicles-ltd/. [Accessed: 05- Nov- 2019].

[9] "Unmanned Surface Vehicle", En.wikipedia.org, 2019. [Online]. Available: https://en.wikipedia.org/wiki/Unmanned_surface_vehicle. [Accessed: 08- Jul- 2019].

[10] "Unmanned Aerial Vehicle", En.wikipedia.org, 2019. [Online]. Available: https://en.wikipedia.org/wiki/Unmanned_aerial_vehicle. [Accessed: 08- Jul- 2019].

[11] Hini, E. (2018). *Unmanned Maritime Systems for Search and Rescue*. [online] Medium. Available at: https://medium.com/@EliHini/unmanned-maritime-systems-for-search-and-rescue-6610a99c91a0 [Accessed 1 Nov. 2019].

[12] "What is an AUV?", Oceanexplorer.noaa.gov, 2019. [Online]. Available: https://oceanexplorer.noaa.gov/facts/auv.html. [Accessed: 06- Jun- 2019].

[13] D'Estries, M. (2016). *5 companies on the cutting edge of drone delivery*. [online] From the Grapevine. Available at: https://www.fromthegrapevine.com/innovation/companies-cutting-edge-drone-delivery [Accessed 18 Oct. 2019].

[14] Vroegindeweij, B., Wijk, S. and Henten, E. (2014). Autonomous unmanned vehicles fro agricultural applications. Available at: https://library.wur.nl/WebQuery/wurpubs/482638 [Accessed 7 Nov. 2019].

[15] "Robotic paradigm", *Revolvy.com*, 2019. [Online]. Available: https://www.revolvy.com/page/Robotic-paradigm. [Accessed: 05- Nov- 2019].

[16] Pendleton, S., Andersen, H., Du, X., Shen, X., Meghjani, M., Eng, Y., Rus, D. and Ang, M. (2017). *Perception, Planning, Control, and Coordination for Autonomous Vehicles*. [online] Available at: https://www.researchgate.net/publication/313834721_Perception_Planning_Control_and_Coordination_for_Autonomous_Vehicles [Accessed 7 Nov. 2019].

[17] Y. Zhao, W. Xing, H. Yuan and P. Shi, "A Collaborative Control Framework with Multi-Leaders for AUVs Based on Unscented Particle Filter", Journal of the Franklin Institute, 2015. Available: https://www.sciencedirect.com/science/article/abs/pii/S0016003215004512?via%3Dihub. [Accessed 6 July 2019].

[18] M. Mendonça, I. Chrun, F. Neves and L. Arruda, "A Cooperative Architecture for Swarm Robotic Based on Dynamic Fuzzy Cognitive Maps", *Engineering Applications of Artificial Intelligence*, vol. 59, pp. 122-132, 2017. Available: https://www.sciencedirect.com/science/article/abs/pii/S0952197616302524.

[19] H. Qin et al., "Autonomous Exploration and Mapping System Using Heterogeneous UAVs and UGVs in GPS-Denied Environments", *IEEE Transactions on Vehicular Technology*, vol. 68, no. 2, pp. 1339-1350, 2019. Available: https://ieeexplore.ieee.org/document/8598942.

[20] O. Simonin and O. Grunder, "A Cooperative Multi-Robot Architecture for Moving a Paralyzed Robot", *Mechatronics*, vol. 19, no. 4, pp. 463-470, 2009. Available: https://www.sciencedirect.com/science/article/abs/pii/S0957415808001827.

[21] C. García, P. Cárdenas, L. Puglisi and R. Saltaren, "Design and Modeling of the Multi-Agent Robotic System: SMART", *Robotics and Autonomous Systems*, vol. 60, no. 2, pp. 143-153, 2012. Available: https://www.sciencedirect.com/science/article/abs/pii/S0921889011001862.

[22] N. Mohamed, J. Al-Jaroodi, I. Jawhar and S. Lazarova-Molnar, "Middleware Requirements for Collaborative Unmanned Aerial Vehicles", *https://ieeexplore.ieee.org/document/6564794*, 2013. [Online]. Available: https://www.researchgate.net/publication/261429743_Middleware_requirements_for_collaborative_unmanned_aerial_vehicles. [Accessed: 06- Nov- 2019].

[23] M. Zhu and Y. Wen, "Design and Analysis of Collaborative Unmanned Surface-Aerial Vehicle Cruise Systems", Journal of Advanced Transportation, vol. 2019, pp. 1-10, 2019. Available: https://www.hindawi.com/journals/jat/2019/1323105/.

[24] "Documentation - ROS Wiki", *Wiki.ros.org*, 2019. [Online]. Available: http://wiki.ros.org. [Accessed: 06- Nov- 2019].

[25] "Master - ROS Wiki", *Wiki.ros.org*, 2019. [Online]. Available: http://wiki.ros.org/Master. [Accessed: 06- Nov- 2019].

[26] Z. Wang, T. Takahashi, T. Nitsuma, T. Ninjouji and E. Nakano, "LOGUE: an architecture for task and behavior object transmission among multiple autonomous robots", *Robotics and Autonomous Systems*, vol. 44, no. 3-4, pp. 261-271, 2003. Available: https://www.sciencedirect.com/science/article/abs/pii/S0921889003000769.

[27] "Java Remote Method Invocation", *En.wikipedia.org*, 2019. [Online]. Available: https://en.wikipedia.org/wiki/Java_remote_method_invocation. [Accessed: 04- Nov- 2019].

[28] I. Sakiotis, "A Simulation-Based Layered Framework Framework for the Development of Collaborative Autonomous Systems", *Master of Science (MS), thesis, Modeling Simul & Visual Engineering, Old Dominion University*, 2016. Available: https://digitalcommons.odu.edu/msve_etds/4. [Accessed 6 November 2019].

[29] J. Martin, O. Casquero, B. Fortes and M. Marcos, "A Generic Multi-Layer Architecture Based on ROS-JADE Integration for Autonomous Transport Vehicles", *Sensors*, vol. 19, no. 1, p. 69, 2018. Available: https://www.researchgate.net/publication/329907421_A_Generic_Multi-Layer_Architecture_Based_on_ROS-JADE_Integration_for_Autonomous_Transport_Vehicles.

[30] T. Finin, R. Fritzson, D. McKay and R. McEntire, "KQML- A Language and Protocol for Knowledge and Information Exchange", *Pdfs.semanticscholar.org*, 1994. [Online]. Available: https://pdfs.semanticscholar.org/c0da/82b917832ecb3dffc4dd16b0e1e1dbdf153b.pdf. [Accessed: 06- Nov- 2019].

[31] Amqp.org. (2019). *Home | AMQP*. [online] Available at: https://www.amqp.org/ [Accessed 7 Nov. 2019].

[32] En.wikipedia.org. (2019). *Advanced Message Queuing Protocol*. [online] Available at: https://en.wikipedia.org/wiki/Advanced_Message_Queuing_Protocol [Accessed 8 Nov. 2019].

[33] "ISO Reference Model for Open Systems Interconnection (OSI)", *Bitsavers.org*, 1991. [Online]. Available: http://www.bitsavers.org/pdf/datapro/communications_standards/2783_ISO_OSI.pdf. [Accessed: 05- Nov- 2019].

[34] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1994). *Design patterns*. 1st ed.

APPENDICES

APPENDIX A: DEMONSTRATION EXAMPLE GRAPHS

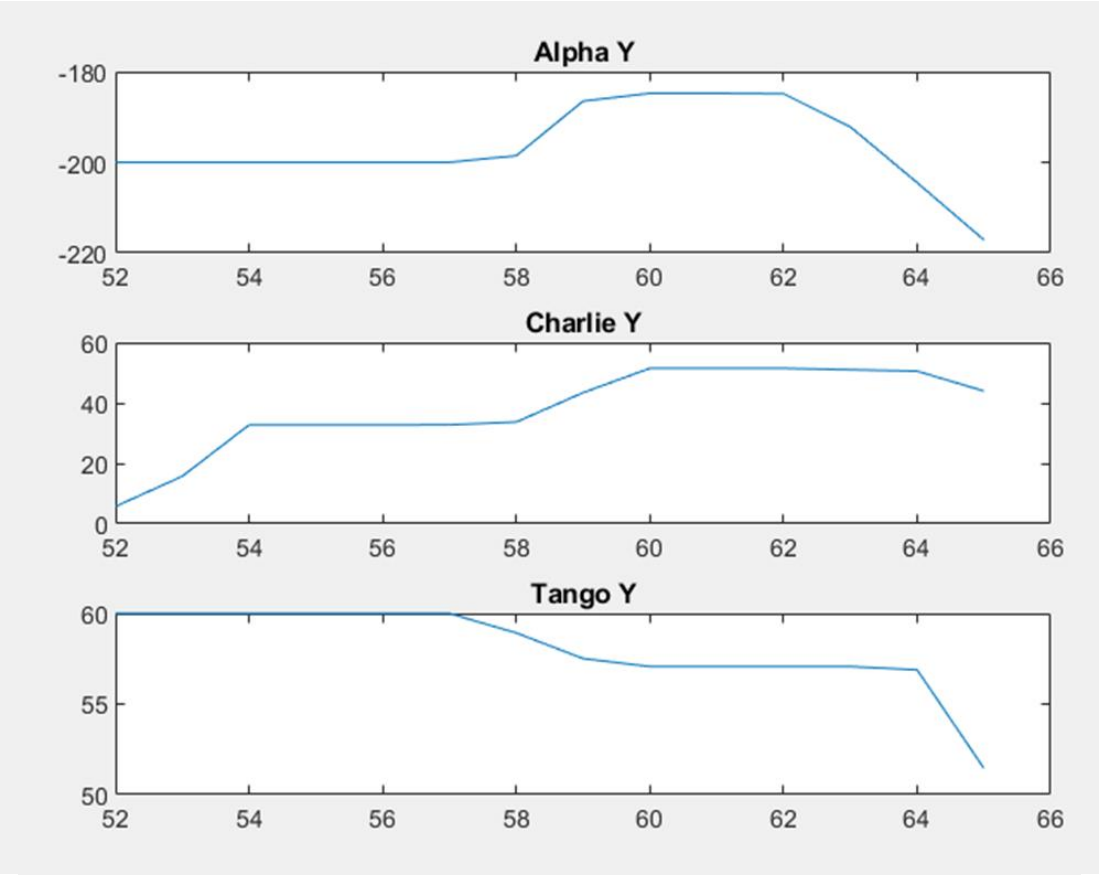

Fig. A-1 X-Positions of Rovers.

Fig. A-2 Y-Positions of Rovers.

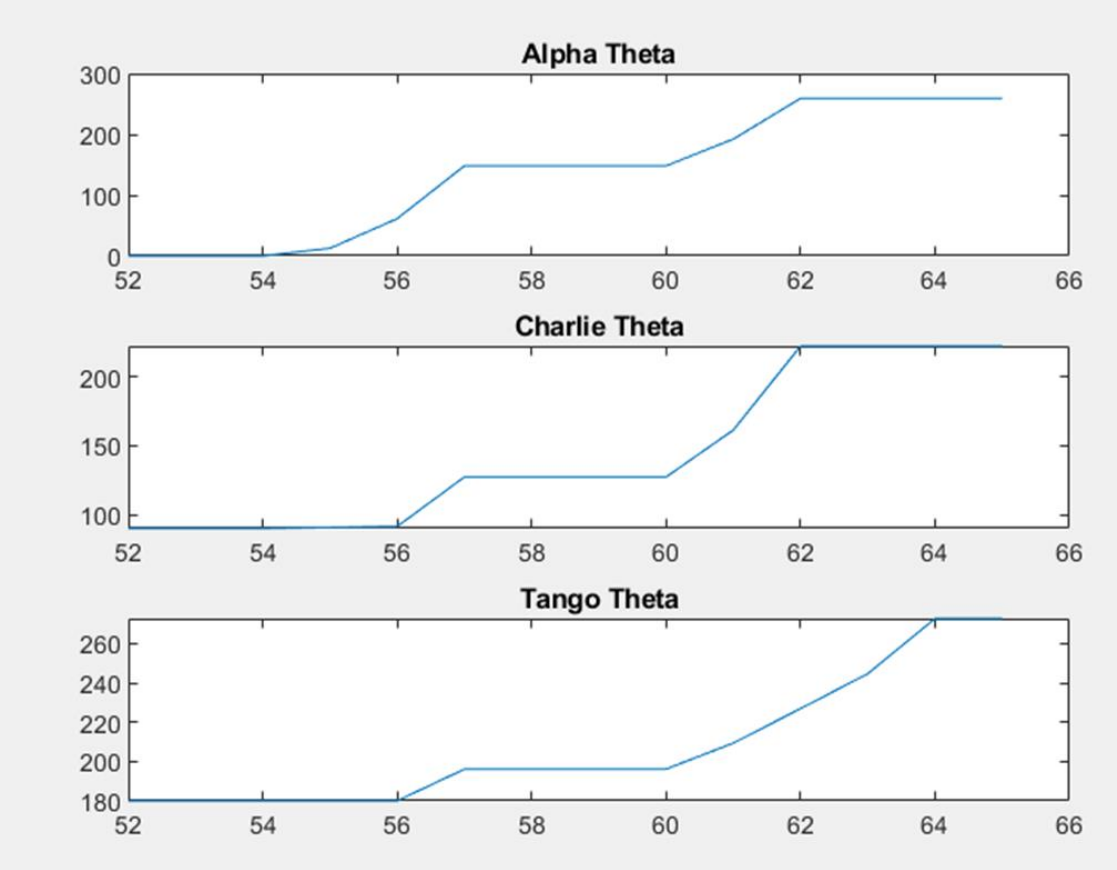

Fig. A-3 Orientation of Rovers.

VITA

NTIANA SAKIOTI

PERSONAL INFORMATION

Born: 09/22/1996
Citizenship: Greek
Email address: nsaki001@odu.edu

EDUCATION

Old Dominion University, Norfolk, VA
Frank Batten College of Engineering and Technology, Electrical and Computer Engineering Department
Bachelor of Science in Computer Engineering, 2017
Bachelor of Science in Electrical Engineering, 2017

AWARDS AND HONORS

VMASC Industry Association Undergraduate BS/MS Scholarship
AHEPA District Scholarship
AHEPA Academic Scholarship
Undergraduate Research Grant
Dean's List
Kovner Scholarship

PUBLICATIONS

- "Evaluating Kinect V1 and V2 For Chest Wall WALL Surface Scanning and Assessment" IWISH, 2019.
- "Applying a Test and Evaluation Framework to an Unmanned Maritime System", MSVE Capstone Conference, 2019
- "Experimental Validation of a ground robot simulation model during line following task", MODSIM WORLD 2017, 2017