Summer 2016

A Simulation-Based Layered Framework Framework for the Development of Collaborative Autonomous Systems

Ioannis Sakiotis
Old Dominion University

A SIMULATION-BASED LAYERED FRAMEWORK FOR THE DEVELOPMENT OF COLLABORATIVE AUTONOMOUS SYSTEMS

by

Ioannis Sakiotis
B.S. December 2014, 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
August 2016

Approved by:

James F. Leathrum, Jr. (Director)

Roland R. Mielke (Member)

Frederic D. McKenzie (Member)

ABSTRACT

A Simulation-Based Layered Framework for the Development of Collaborative Autonomous Systems

Ioannis Sakiotis
Old Dominion University, 2016
Director: Dr. James F. Leathrum, Jr.

The purpose of this thesis is to introduce a simulation-based software framework that facilitates the development of collaborative autonomous systems. Significant commonalities exist in the design approaches of both collaborative and autonomous systems, mirroring the sense, plan, act paradigm, and mostly adopting layered architectures. Unfortunately, the development of such systems is intricate and requires low-level interfacing which significantly detracts from development time. Frameworks for the development of collaborative and autonomous systems have been developed but are not flexible and center on narrow ranges of applications and platforms. The proposed framework utilizes an expandable layered structure that allows developers to define a layered structure and perform isolated development on different layers. The framework provides communication capabilities and allows message definition in order to define collaborative behavior across various applications. The framework is designed to be compatible with many robotic platforms and utilizes the concept of robotic middleware in order to interface with robots; attaching the framework on different platforms only requires changing the middleware. An example Fire Brigade application that was developed in the framework is presented; highlighting the design process and utilization of framework related features. The application is simulation-based, relying on kinematic models to simulate physical actions and a virtual environment to provide access to sensor

data. While the results demonstrated interesting collaborative behavior, the ease of implementation and capacity to experiment by swapping layers is particularly noteworthy. The framework retains the advantages of layered architectures and provides greater flexibility, shielding developers from intricacies and providing enough tools to make collaboration easy to perform.

ACKNOWLEDGMENTS

First and foremost, I would like to express my sincere gratitude to my adviser, Dr. Leathrum for the effort that he put towards the completion of this thesis. His guidance and contributions were absolutely necessary in the design and implementation process of the system that is discussed in this thesis. I would also like to thank the members of the Committee Drs. Mielke and McKenzie and the graduate program director, Dr. Y. Shen, for their guidance and participation in the thesis evaluation process. Finally, I would like to thank Brandon Waddell, Darren Rose, and Duncan Leathrum for their participation and assistance in the development of the CAS Framework.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

Page

CHAPTER 1

INTRODUCTION

The development of autonomous systems becomes increasingly relevant as applications become more complex (space exploration, rescue) and difficult to be controlled. Increased autonomy leads to an improvement in productivity, through consistency and elimination of human error. Furthermore, autonomy reduces operational costs since operators are no longer required and, in certain cases, can eliminate technical difficulties. Autonomous robots are utilized in space exploration missions due to the communication latency, which makes direct operational control unfeasible. Autonomous robots can potentially be used in a plethora of fields, from industrial and space, to agricultural and military applications. For a significant number of those applications, the presence of multiple robots is beneficial and sometimes necessary, especially if they have the capacity to communicate and collaborate toward the accomplishment of specific tasks. Autonomous systems bear many requirements such as high reliability, capability to handle uncertain scenarios, multi-hierarchical goals, status evaluation, capability for human-robot interfacing, and more, which make their design complex and difficult to carry out. If such systems are required to collaborate, then their behavior becomes even more complex and the requirements of communication capabilities steeper. This research develops a simulation-based software framework to facilitate the design and implementation of collaborative autonomous systems[1].

---

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

1.1. Problem Statement

Developing collaborative autonomous applications is an intricate process. The developers have to establish software architectures for the behavioral aspects of the robot; communication capabilities have to be implemented and the software has to interface with a specific platform. These processes take time and require rigorous testing. Despite the variance of the autonomous system applications, several characteristics are shared. Most systems have a similar structure and order of operations; robots use sensors to receive data and upon processing and planning, they attempt to accomplish a sequence of tasks. If the application requires communication capabilities, that functionality largely remains the same, only requiring interfacing with the specific platform. These similarities provide the potential for a software framework to be utilized in the development process. A framework should provide substantial flexibility and should be moldable enough to allow the use of the provided functionality and structures across various applications. The adoption of a software framework would allow developers to simply utilize its features instead of wasting time in order to accomplish something that has already been done.

1.2. Proposed Framework

A layered software framework is presented for the purpose of facilitating the development process of collaborative autonomous systems, termed the Collaborative Autonomous Systems (CAS) Framework. The framework will provide moldable structures for the user to insert behavioral algorithms, without denying the capacity for expansion and addition of further structures. Furthermore, communication capabilities will be integrated to the structures, allowing autonomous robots to exchange user-defined messages under the condition that their layered structures are identical. The internal modules of an agent will also have the capacity to utilize the framework's communication capabilities. Clear communication interfaces

will be provided, allowing the replacement of communication components without affecting the surrounding systems. For the same purposes, the framework is intended to interface with a robotic middleware. A robotic middleware is a platform-specific component that allows software to interface with heterogeneous hardware by translating common commands (sensor instructions, move, rotate, etc.), to instructions that can be carried out by the robot [1]. By placing the middleware between the application and the operating system of the robot, it is possible to field the framework on different platforms simply by using the appropriate middleware.

The motivation behind the development of the framework stems from the lack of flexibility in the designs presented in past literature. The architectures that have been previously proposed, both for collaborative and autonomous systems, are efficient and robust. Unfortunately, they are designed for specific applications and platforms making their adaptation to other applications difficult to perform. The proposed structures are rigid and contain already developed and interconnected structures for planning, scheduling, navigation, obstacle avoidance, etc. Thus, the user would either have to strip the system of most functionality if a high-level structure is desirable or spend time filtering compatible portions and merging them to his system.

The proposed framework seeks to mitigate these difficulties by imposing few restrictions and providing moldable structures without imposing any algorithms for their internal processing. The framework is intended to allow the development of the autonomous and collaborative behavior in harmony, by isolating these components and providing clear interfaces. Aspects such as layered structures, isolated development, and concurrent layer execution, which were prevalent in past architectures, have been adopted by the framework. Since compatibility with a plethora of applications is desired, the design is intended to bridge the gap and allow the isolated

development of decision-making and collaborative models by definable layers and clear interfaces, not only between components but also with the physical hardware. By enforcing a high degree of modularity, the framework is intended to be compatible with multiple hardware and communication components (swapping interfaces) and supports the evolution of the system by allowing specific components to be replaced or added.

1.3. Contents of the Thesis

Before delving into details of the CAS Framework's design, a review of important aspects in the design of collaborative autonomous systems is provided in Chapter 2. By examining past literature, important features that should be adopted are mentioned as well as the importance of the framework's benefits and how it differs with past work in the field. In Chapter 3, the framework model is described; important design choices that stem from common application requirements are discussed in addition to main characteristics of the layered model. Chapter 4 describes the software model in detail, highlighting key concepts and algorithms that are instrumental in the framework's functionality. Furthermore, the frameworks' capabilities are demonstrated in Chapter 5 by analyzing the design process of an example application and highlighting both the use of already defined functionality and the effect of user interaction. The results of this application are discussed and evaluated in terms of observed collaborative behavior and ease of implementation.

CHAPTER 2

BACKGROUND

There are numerous applications for autonomous robotic systems (ARS): un-manned space exploration, agricultural, satellites, manufacturing, and autonomous cars. The benefits of achieving autonomy, include a potential increase in reliability, response time, performance, and much more depending on the application and the degree of autonomy that is implemented. For some of these systems, such as satellites and autonomous cars, collaboration is an integral component of their functionality. Self-driving cars perform better when capable of communicating in order to avoid collisions. Multi-satellite systems are required to collaborate and coordinate in order to accomplish certain tasks, thus eliminating the issues with dynamic response and flexibility that are present in centralized control systems. Autonomous systems are largely designed without considering the collaboration element—probably due to the fact that the only communication requirements of many systems are related to overriding autonomy by operators or supplying parametrized requirements; such is the case with exploration land-rovers and manufacturing robots. The design requirements of these isolated autonomous systems are still relevant and in fact they provide the foundations for the development of autonomous systems that have collaborative capabilities. Similarly, collaborative systems have been developed, focusing on the transferring and organizing of information, sometimes without requiring a physical platform. A discussion of the characteristics and aspects of both autonomous and collaborative systems takes place in this chapter, highlighting the requirements for a software based framework whose purpose is to facilitate the development of systems that share characteristics from both categories. Furthermore, aspects that were not adopted from these

already developed architectures are discussed. Finally, the potential and the limitations of the framework, in addition to the future development and testing that take place are described.

2.1. Autonomous Systems Architectures

In the development of autonomous robotic systems, layered structures seem to be prevalent. The benefit of this approach is in its flexibility and modularity. Developers of different layers can work in isolation. This becomes extremely relevant with the multi-disciplinary aspect of ARS, which include elements of computer science, artificial intelligence, operations research, and communications [2]. Furthermore, such architectures, allow entire layers to be substituted under the conditions that the appropriate interfaces are honored. The modular aspect of the layered approach allows developers of different expertise to be distanced from the implementation details of other layers. While there are alternatives to the layered approach, they do not offer the same amount of flexibility and are thus significantly less popular.

2.1.1. Three Layer Architectures

The distribution of systems into layers follows several patterns which can be noticed across various works that are focused on different fields. The layers tend to be organized hierarchically were goals and actions are passed down from layer to layer. The context of the information is relevant to the layer level; higher level layers aggregate lower level layer information. With this approach, layers can execute portions of missions, often in an iterative manner that is coordinated by the above layers. Such is the case with the architecture described in [2]. The author, introduces three layers: management and organization, coordination, and execution. The management and organization layer serves as the controller executive and is responsible for the higher level decision-making and planning. The coordination layer performs

middle management and can algorithmically process (partition, parameterize, etc.) the plans of the control executive, eventually passing commands to the execution layer, which deals with hardware and the low-level software that operates it. In fact, this three-layer approach is encountered often in such systems. In [3], the same layers are introduced, albeit with different names: symbolic, behavior, and sensorimotor. Emphasis is given to the behavioral layer which translates the abstract navigation requirements of the symbolic layer, and either activates or deactivates specific modules (internal to the layer) that are responsible for coordinating specific tasks. In addition, a programming environment is mentioned, ARCA, which was used to develop this highly modular layer and supports multi-threading. The authors of [3], mentioned that the intricacies of layer interfacing were eliminated by using ARCA, allowing the developer to disregard such intricacies and focus on actual layer development.

In [4] a complex layered model was introduced which adopted a hybrid layered structure. The layers were organized according to a hierarchy but modules within the layer were also hierarchically organized. These modules could interact even outside the layer, regardless of whether the layers that contained those modules were neighbors. The three layers are: goal directed actions (GDA), reactive, and reflexive. The GDA layer receives sensory information and supplies it to the Path Planner, an independent module within the layer that analyzes the environment and formulates a path which will in turn be sent to the Navigation System, another in-layer module. This module will generate navigational commands by comparing the current and goal states. Then, the Reflexive layer becomes relevant which has uses its access to sensors to react accordingly; based on pre-defined rules it avoids issues such as collision. The Reflexive and GDA layers are not linked in the layered hierarchy, yet the components inside still interact. This is a departure from the models in [2] and [3] where only neighboring layers could

communicate directly. Components on the top of the hierarchy can be seen interacting regarding map creation, relying on the Reactive layer to build a local view, and on the GDA layer to update and store the global map. The sensors are external to the system but interface with both the Reflexive and Reactive layers.

A three layer architecture was also adopted in [5], in terms of organizing the functionality of a single construction vehicle. The layers are: vehicle, subsystem, and primitive. The functionality of these layers is outlined throughout [5] and reveals that these layers resemble those of the common three layer structure (high-level decision-making, low level decision-making, and action controller). The top layer (vehicle) supplies mobility or construction related plans to the layer below, and depending on the plan, a separate in-layer module formulates a lower level plan which is passed on the third layer. The primitive layer eventually interfaces with the sensors and actuators of the platform and executes actions that are required by the low level plan. This architecture was inspired by the subsystem layered structure that was proposed in [6], which is a framework for autonomous system development. In that framework, the layers are organized as: subsystem, primitive, and servo. Regarding the execution of these layers, the authors of [5] recognize the need for parallel processing by pointing out that construction vehicles perform various actions at the same time (control bucket for digging, moving, etc.)

Another three layer architecture was introduced in [7], with the sensors, brain, and effectors comprising the layered structure. The departure in this case was that further behavioral layers populated the brain layer. In fact, the focus of the architecture is these behavioral layers which can involve path following, shooting, obstacle avoidance and moving.

Architectures of such systems often require the capacity for human intervention regardless of the desired degree of autonomy. The system in [8], which was designed for a

specific platform, has a three layer structure: supervisor and plan, control, perception/actuator. In the literature mentioned throughout this section, the top two layers of the hierarchy involved planning and coordinating. These two layers abstracted decision making into two levels in order to reduce the complexity and apply a command-and-conquer approach to mission accomplishment. In [6], this decision-making/coordination functionality has been integrated to the middle layer. Instead, the top layer in [8], is purely a human-robot interface which allows an operator to formulate and issue plans of execution to the control layer. While the mission is issued by an operator, the execution is completely autonomous and relies on the control layer to coordinate actions and issue specific commands to the perception/actuator layer. The difference in the context of the layers in this case, is particularly noteworthy. It demonstrates that despite popular architectures, distinctiveness still occurs and is something that must be taken into consideration in the design of the framework.

There are many instances of three layer architectures for autonomous agents. The authors of [9] and [10] also used three layers for their architectures. The pattern that is observed is that the same functionality is either integrated or disintegrated into layers, depending on whether additional functionality must be inserted in other layers (i.e. human intervention layer), Regardless, the management, control, and execution layers are prevalent in such architectures. In [11], the reason behind the popularity of this structure was examined, highlighting the common algorithms of such systems which tend to fall into three categories, each being assigned to a layer. Reactive control algorithms are implemented on the lowest level layer that deals with hardware control (sensors, actuators). Activity sequence algorithms on the other hand, rely extensively on internal state parameters in order to formulate plans; these algorithms populate the middle layer. Finally, searched-based algorithms are effective in the highest decision-making

layer which has to select pre-defined goals according to certain system criteria. Of course, this structure offers requires expansion to include aspects like world modeling or sensor processing.

2.1.2. Less Common Layered Architectures

As mentioned in Section 2.1.1., the most common structure for autonomous agents is comprised of three layers. However, several other configurations exist, utilizing a different number layers. The same aspects of the structures that were mentioned in 2.1.1., still exist within other layered systems, but significant integration or disintegration has taken place.

In [12], the CLARAty architecture is described, which comprises of two layers: Decision and Functional, and was designed for space related control applications. The functional layer contains generic robot capabilities while the decision layer contains the high-level planning functionality. This architecture was utilized in [13], which added domain models, as well as verification and validation related functions that can be accessed by both CLARAty layers.

A quite different model was described in [14], consisting of four layers: modeling, model translation, planning, and execution. Aspects such as the concurrent execution of layers, and the definition of message types centered on collaboration are distinguished within the model. In terms of structure, the same processes, with small variations, are executed as the other systems mentioned throughout the chapter. A representation of the system is created, based on sensor data on the modeling layer; the environment is defined in terms of states while agents are assigned beliefs (mission and parameters) and possible actions. The model translation and planning layers interact until a sequence of actions is scheduled. The execution layer, not only executes actions, but also initiates a re-planning process in case it is unable to perform an action. Another four layer architecture was introduced in [15]. The layers are: sensing, modeling,

planning, and acting, which is not a significant variation compared to all the other architectures that have been mentioned throughout the section.

A slightly different approach was taken in the five layered architecture presented in [16], which includes the layers: visualization, reasoning, intermediate, feature, and data. This architecture was applied for autonomous indoor navigation robots. The data and reasoning layer's function are similar to that of the decision-making and action layers of the architectures in Section 2.1.1; planning/scheduling occurs in the reasoning layer and in the data layer, data extraction occurs while there are direct connections to hardware components such as motors. The other three layers are introduced due to application specific requirements. The feature layer processes the raw data that are collected from the data layer and identifies obstacles such walls and objects, thus allowing the robot to orient itself. The intermediate layer is present for interfacing the feature and reasoning layers. The reasoning layer's planning methodology involves creates 2D-Models and a view of the environment which must be compared against actual sensor data for validation purposes. Non-polygon related data (robot state, etc.) that are collected from sensors will be processed in this layer, since the feature layer only involves polygon identification. Finally, the visualization layer has been included, purely for human-machine interfacing. This layer, creates 3D and 2D representations of the environment and its interaction with the robot, allowing operators to monitor and intervene if such functionality has been included in the robot.

Non three-layered architectures are significantly less common than the three-layered basis that was introduced by [12]. The reason for this can be accredited to the fact that additional layers are mostly application specific[16] while the functionality that has been abstracted across multiple layers, can be integrated into three-layers, or even two, as demonstrated in [12].

Regardless, the fact that the non-three layer aspect is not entirely uncommon and is in fact needed for the development of specific applications, has to be taken into consideration when developing a software development framework.

2.1.3. Non-Layered Architectures

Throughout this section, the benefits of adopting layered architectures, has been discussed and demonstrated by the numerous examples in past literature. Despite the popularity of such architectures, alternatives exist, with traditional components interacting in a non-hierarchical manner. This subsection will introduce some of those systems.

In [17] a navigation framework for autonomous ground vehicles was introduced, which did not encompass any layered structures, relying instead on a sequential, one-directional flow of information regarding vehicle control. The planning and control functionality is still present; the main components of this framework are: sensors, global map, planning module, signal process, local map, artificial intervention, collaborative control, and vehicle control.

The goal of this structure is to provide a specific course that can be followed by the autonomous vehicle, while continuously monitoring the state of both the environment and vehicle, allowing course corrections or artificial intervention to occur. With a global map available, the planning module calculates a coarse trajectory which, in turn, is passed to the collaborative control module. This module, taking into consideration any external collaboration, calculates the ideal trajectory and supplies it to the vehicle control component, which interfaces with the physical hardware. Throughout this process, the sensor component is accessible by the signal process module. Depending on the data received, a secondary path can be re-calculated or the artificial intervention module seizes control, either stopping the vehicle or implementing a specific algorithm tailored to the situation. In direct contrast to the layered architectures

previously discussed in this section, the flow of information is one directional. For example, the planning module does not receive feedback from the collaborative control module which, in turn, does not receive data from the vehicle control module. All feedback and course corrections that could have been handled with bi-directional layer communication, are accomplished by sensors providing data to intermediate components which will eventually reach the planning module.

The layered approach was also rejected in [18], claiming that the upwards flow of information(action to planning layers) only works for navigation related applications due to the simplicity of the world model which essentially only contains robot coordinates. The authors of [18] proposed a framework for behavior-based robots; the main components of this framework are: plan, act, world, sense, and model. Sensing allows a model to be created, according to which "plan" generates a mission and selects the actions that comprise it. Then, "act" executes those actions leading to changes in the environment, which in turn affects the model, leading to more missions. Both the "plan" and "act" modules consist of two interacting components; plan generation and action selection belong to "plan", and activation dynamics and target dynamics belong to "act". The components within "plan" and "act" can interact with each other. The bi-directional communication aspect of the components inside "plan" and "act" resemble that of layers, but this element is not present in the rest of the system; act does not directly communicate with plan, instead relying on "plan" to evaluate the model for feedback.

Despite the robustness of the architectures presented in [17] and [18], the flexibility and modularity of the layered approach seems more appropriate for autonomous robotic systems. The difficulty of upwards information flow that was mentioned in [18] can be counteracted with proper communication protocols and mechanisms which can allow complex information to be easily transferred. If such functionality is implementable, then no issue remains with layered

approaches regarding autonomous systems. Communication, is also important for collaborative systems, whether they are robotic or not, since the transfer of complex information is an integral part of their functionality.

2.2. Collaborative Systems

The architectures introduced in Section 2.1 had structures that emphasized the autonomous aspect of their respective application. For some, collaboration and/or communication modules were included but such functionality was not the priority of the design. This section will introduce architectures of purely collaborative systems, and identify features that would be suitable for the CAS framework.

2.2.1. Collaborative models for development processes

The authors of [19] manage to highlight two of the key reasons behind the importance of collaboration, which are the difference in context of the processes that occur within a system and the difference in the knowledge between developers. In [19], a layered collaborative process model for the development of expert medical systems is introduced. The collaboration aspect of this model is extremely different from those of autonomous robotic systems; this model does not involve collaboration between homogenous agents. Instead, it distinguishes components within the development process that require collaboration regarding the development of an application, not its execution. The necessity for such collaboration on medical systems, arises from the fact that expert medical knowledge, as well as technical and administrative knowledge, are required. This knowledgebase is composed by multiple individuals, who nonetheless need to communicate effectively and collaboratively reach certain solutions, whether that is an agreement over specifications or the execution of verification and validation procedures.

Three hierarchical layers were proposed: conceptual, societal, and computational. At the top of the hierarchy is the conceptual layer which involves the iterative process of creating conceptual and formal models until they are appropriate for verification and validation. This process is expected to be collaborative and is to be performed only by appropriate clinical experts. By isolating these processes into a layer, it is guaranteed that non-experts will not hinder the development. Once the processes within the conceptual layer have been completed, the societal layer takes control. This layer aims to resolve socio-organizational issues that arise specifically for medical applications; ethical dilemmas regarding privacy and legality often need to be handled. Resolving such issues requires collaboration from clinical and administrative experts. The two can bridge issues regarding their knowledgebase differences by utilizing educational resources (workshops, documents, etc.). With all administrative issues resolved, the computational layer takes control which involves the actual implementation of the system. Technical and clinical experts have to collaborate in the implementation process; the technical experts will build the system while clinical experts will supervise and make sure that all clinical requirements have been met.

The difference in the context of each layer is important, and makes the layered isolation appropriate. Administrative experts should have no involvement outside the processes of the societal layer. Similarly, clinical experts should be present throughout the entire development and not leave technical experts to judge the quality of the system since they don't have the appropriate knowledge. By organizing the collaboration process and defining the duties of each participant, the development of medical systems is expected to be facilitated.

2.2.2. Collaboration for Autonomous Agents

Collaboration is relevant in [20] which discusses the distribution of satellite control and its effectiveness towards autonomous coordination. The necessity of adopting a distributed approach is caused by the lack of flexibility and implementation difficulties (dynamic response, processing power, etc.) in the communication capabilities of centralized satellite systems.

In order to eliminate the slave-master relationship of satellites, control is distributed among homogenous satellite systems. All satellites have the same capabilities, but nonetheless, during a collaborative mission, they are assigned into two layers: decision-management and behavior performance. Despite this distribution, all satellites retain their characteristics; the only effect of this organization is that the decision-management satellites will request and thus initiate collaborative actions and that the behavior layer satellites will have to reply. In order to allow flexible missions to be accomplished, all satellites that are assigned in the same layer can communicate with each other directly. Intra-layer communication is restricted to robots that have collaborative relations.

Despite the restrictions that are imposed by this layered structure, the satellites are fully autonomous and are capable of internal decision-making and processing. The structure that defines the behavior of each system is also distributed among two layers: intelligence core and outer shell. The intelligence core—which is further partitioned to “multiple satellites coordination” and “single satellite autonomy” layers—is responsible for decision making and task planning (collision avoidance, formation planning, etc.). The outer shell handles sensor control, receiving state information about the environment and other satellites as well as passing them to the intelligence core. Furthermore, it contains a response agent model which contains state information about the satellite.

The layered approach was also adopted in [21], which introduced a seven layer architecture for location-based applications: sensors, measurements, fusion, arrangements, contextual fusion, activities, and intentions. The layers communicate with each other in an upward manner. Sensors extract raw data, which are then handled by the measurements layer. The Fusion layer, with the data provided by Measurements, keeps track of object locations, with the potential to also include, speed, acceleration, and coordinate history. Arrangements calculates the relationships between objects while contextual fusion can interpret and use such information to identify noteworthy states or events. Then, the activity layer can identify what actions need to take place. Finally, the Intentions layer contains user related information and requirements.

2.2.3. Organizational Collaboration

The aforementioned applications are not the only examples of layered architectures on collaborative systems. In fact, such architectures are applicable in other fields as well; in [22] a four layered architecture was introduced for autonomous coordination in urban logistics, while [23] introduced seven layers for organizing collaborative information and distribution design concerns. Despite the popularity of layered architectures, alternatives always exist. In [24] a non-layered modular architecture was proposed for autonomous collaboration of unmanned heterogeneous aerial vehicles. The main modules of the architecture are: mission management, autonomous entity management, autonomous entity executive, and world model. The group mission management module handles planning and monitoring for groups of entities while the entity management plans individual actions. The entity executive module supervises these actions while the world model provides information about the environment and other vehicles in the system. The components of the architecture do not universally communicate with each other, thus forbidding erroneous interactions such as Group Mission Management with Entity

Executive. Since the agents are heterogeneous, adopting a layered approach would yield complications (layer-layer communication would require extra overhead, etc.), which would render it unsatisfactory.

2.3. CAS Design Approach

The CAS software framework, is intended to facilitate the development of collaborative autonomous systems. Frameworks for this purpose have been built in the past. By mentioning the main aspects of those frameworks, the differences and benefits of the CAS framework will be highlighted.

The 4D/RCS framework [6] is intended for development of intelligent vehicle systems with certain degrees of autonomy and was the basis for the architecture presented in [5]. The premise of 4D/RCS is not much different from the CAS framework; both support the development of intelligent systems that are intended to pursue the accomplishment of goals. On the other hand, the basis for 4D/RCS is coordination and for this reason, the layered distribution applies to the agents themselves and not only their contents. One layer can represent a battalion of vehicles, or company, platoon, etc. The architecture of the agent itself is layered in [6], comprising of a subsystem, primitive, and servo layers. Another framework related architecture was presented in [25]. The authors proposed a hierarchy of layers that correspond to specific robotic capabilities. Furthermore, upstream and downstream flow of information is be present, allowing the issuing of commands and the receipt of feedback. While the collaboration capabilities within components are not explored in [25], the layered approach mirrors those of Section 2.1.

The CAS framework has a layered structure that is intended to mirror both the autonomous and the collaborative systems in Sections 2.1 and 2.2. The systems in Section 2.1

vary greatly in structure, implementing different number of layers, and bear the disadvantage of being designed for specific fields and platforms. The CAS framework is intended to allow the development of a plethora of systems, regardless of variation in the formal model or the platform that is to be implemented. It is postulated that all layered systems, including the ones in Sections 2.1 and 2.2, are implementable in the CAS framework. This requires a flexibility in the molding of the layered structure and communication capabilities. By allowing the developer to define the number of layers and the context of the communication that flows between agents and layers, significant flexibility is allowed, certainly enough to implement most of the aforementioned systems. The organizational partitioning of the agents that can take place in [6] is not present in the current version of the CAS framework. Such functionality would have to be implemented by the user instead. Furthermore, the framework assumes that the layered structure of all robots is homogenous. Non-homogeneity can potentially pose issues to the communication capabilities due to the context difference among the layers.

CHAPTER 3

CAS FRAMEWORK MODEL

The purpose of the CAS Framework is to facilitate the design and development of collaborative autonomous systems. Such systems require intricate decision-making and collaborative capabilities to be employed across different levels of abstraction. These levels can represent management, coordination, control, communication, and execution but can vary significantly across systems. In [13] management and coordination are merged; in [2] and [11] they are separate. On the other hand, [16] combines the management and coordination levels but also proposes further level abstraction with the inclusion of additional levels for human-machine interfacing, data extraction, and data translation. Partitioning a system according to such levels is a common theme among robotic systems who often impose a hierarchy that highlights level dependencies, sequence of operations, and information flow. The purpose of those levels is to perform processing that will result in the achievement of individual and collective goals. Since the levels are applied to similar systems, robots that use similar technology, there are bound to be many similarities across systems, but also quite a few differences due to application variance.

This chapter introduces a model which involves the mapping of these levels to independent partitions called layers. By adopting a layered structure, developers of varying expertise will be able to distance themselves from the intricacies of other layers. This is more difficult to achieve with non-layered systems since it requires a more intensive definition of interfaces between all interacting components as well as further organization. Despite the independence of layers in this model, layer to layer interaction will be possible through the framework's communication capabilities, which are also described in this chapter. The

requirements of this layered approach will be explored in depth based on the characteristics of the architectures mentioned in Chapter Two, and the role of the system as a framework. Then, a layered architecture will be proposed and evaluated in its capacity to satisfy the requirements. Finally, the object oriented aspects of the CAS Framework will be introduced and will illustrate the mapping of the model to the resulting structure.

3.1. Layered Model

Collaborative autonomous systems can intuitively be organized according to a layered hierarchy by assigning layers to collaboration, decision-making, and execution. In [26] the authors list an approach to partitioning layers: what to do, how to do it, how to automatically adjust the process, and how to execute instruction sequences. Such layers are distinct in their functionality, making them appropriate for isolated development. Of course, the dependency aspect is present as well. Collaboration can potentially take precedence to all other layers, since it can provide data that influences all functionality (new requirements, new mission, etc.). The intricacy of these layers often allows further partitioning, resulting in more than three layers dedicated to the same functionality. Layers can also be integrated resulting in a smaller number layers such as the CLARAty architecture in [13].

The model represents each level of abstraction as an independent layer (Fig. 1). These layers have the capacity to communicate, allowing them to collaborate regarding the accomplishment of both individual and collective goals. Communication avenues, illustrated in Fig. 1 with directed red lines, are dependent on the hierarchical structure. External communication (robot to robot) is restricted to same level layer interaction; layer 1 of Robot 1 can communicate with layer 1 of Robot 2, not layer 2, 3, 4 or 5. In addition to this parallel communication, layers within a robot can communicate with their neighbors; layer 3 can only

interact with layers 2 and 4 as long as they belong to the same robot. This internal communication aspect of the layered model, can be used to implement layer dependencies such as the one in [13] where the Functional Layer is dependent on the Executive/Planner Layer and requires information to determine what actions it must execute.

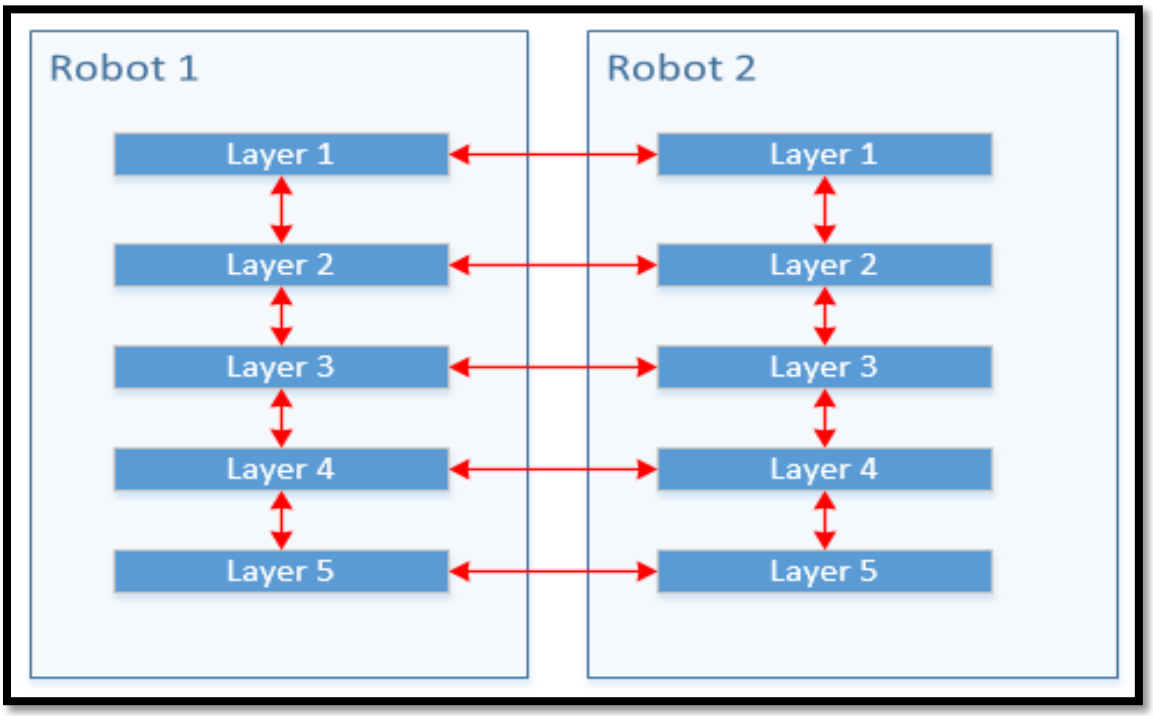

Fig. 1. Layered Model.

Adopting a layered approach might be compatible with many applications due to the modularity of modern autonomous systems but most importantly it offers flexibility and a clearly defined structure in the development process. With clear interfaces, layers can be developed independently and at the same time prevent any implementation details from affecting the rest of the system. Furthermore, layered partitioning allows testing alternate structures by swapping entire layers with ease, as long as the interface is matched, and even perform maintenance and

testing independently across layers. Despite the additional overhead that is required by layered architectures (layer interface, communication interface, layer communication), the flexibility that is awarded justifies its adoption, especially since frameworks must be moldable in order to allow the development of highly varied applications.

A framework based on this model, allows a user to develop applications by defining the contents of the layers and the information that flows between them. While the layered aspect alone aids in the development of most systems, an embedded communication capability, provided that it is not application specific, facilitates the development process since it is an important component which would not have to be rewritten.

3.2. Requirements

The requirements of the CAS Framework are derived from typical aspects of collaborative and autonomous systems as well as by studying patterns in the implementation of compatible applications. There are numerous systems across a wide selection of fields (engineering, sociology, etc.) that can be developed in such a framework. The multi-level aspect alone in some applications is an indicator of compatibility and it has been heavily utilized both for collaborative and autonomous systems. Among such systems, requirements such as flexible topologies [25], collaboration [20], isolated development [25], and concurrent layer execution [3] [5] [14] are listed. The requirements that stem from these categories as well as additional suggestions will be explored in this section.

3.2.1. Flexible Topology

Among a number of systems, autonomous and/or collaborative, the adoption of layers was prevalent. Architectures introduced in [1] through [16], and [19] through [23], all employed

modularity and distributed the system's functionality across several layers. Despite this common characteristic, the number of layers differed often and even when that was not the case, the roles of these layers were different.

In [23], a collaborative model was organized in seven layers based on areas of design concerns: goals, products, activities, patterns of collaboration, techniques, tools, and scripts. In this approach, each layer is dependent on the layer above it and thus requires direct communication, just as in Fig. 1, which illustrates the collaborative layered model that is adopted by the CAS framework. The authors of [2] recognize that there are multiple layer partitioning approaches based on the application's needs. This is evident by comparing the architectures in Section 2.1.1 with those of Sections 2.1.2, 2.1.3 and 2.2. These architectures not only have a different number of layers, but in some cases there are application specific layers which are not encountered elsewhere; such is the case with the SPARTAN system in [20] which has layers dedicated to image processing, and the indoor navigation robot of [16] which has a human-machine interfacing layer.

An example is presented to highlight the possible layered approaches. An application that allows two robots to collaboratively move boxes from a source to a destination does not require the common three-layer structure or any particular number of layers. In this scenario, a possible layer partitioning could be Decision-Making, Task, Coordination, and Action. The Decision-Making layer would collaborate with the other robot regarding the high level objective which could involve the transportation of a specific object to a destination. The Task Layer would determine if the transportation would be a joint process or if the robot could handle it on its own. The Coordination Layer would collaborate in terms of moving the object (balancing, etc.) Finally, the Action layer would execute specific actions such as moving the left robotic arm in

order to lift a box or moving to a point on the plain with a specific velocity. This example shows that a framework that is designed for the implementation of collaborative systems on a layered architecture should not have a fixed number of layers or even fixed layer roles. Thus, a requirement of the CAS framework is to allow the user to decide the number of user defined layers and their roles.

3.2.2. Communication

The layer dependencies within a robot and the collaborative aspects of the system that will be applied, require communication capabilities. The layers must be able to communicate with other internal layers (dependencies) and external layers which belong to other robots (collaboration). Of course, the context of each layer in the hierarchy is different. Due to this context difference, layers cannot be guaranteed meaningful communication with external layers of a different hierarchy level. For the same reason, the hierarchy must be identical across the robots that are required to collaborate. Thus, by limiting external communication to only identical level layers and by requiring the same number of layers and functionality across all robots, reliable communication can be guaranteed. Internal communication is also limited; information can be exchanged only with neighboring layers. This limitation reduces the complexity of highly interdependent layers and maintains a significant degree of clarity and robustness in the hierarchy. Of course, communication can still occur between non-neighboring layers by passing information to intermediate layers.

All the information that is transferred by the framework will be represented as messages, each with different contents and structure according to the needs of the application. In Chapter 2, several systems were introduced which had variations on the number of layers and their roles. This variation in modularity implies that the information that is passed across layers will be

different for each system. Management layers are expected to pass high level decisions (requirements) for a certain mission to the Coordination layer, which will receive this message and request an action from the Execution Layer. In the CLARAty two layer architecture [12], the Management and Coordination layers are merged and thus the mission requirement message will not be sent across layers. On the other hand, the two robot example box moving example mentioned in Section 3.2.1, will require that message to be split. In this case, the Decision-Making layer would send a message that includes information about the box that is to be moved and nothing else. Then, the Coordination Layer, upon receiving that message, would communicate externally with the other robot about the balancing issues in the moving process (what side will each robot lift, etc.) There are great differences in the context of the messages that must be passed based on the layer hierarchy and the application itself. Thus, the messages that will be handled by the system should be user defined.

3.2.3. Isolated Layer Development

One of the key benefits of the layered architecture implemented in the CAS Framework is that it allows developers to work on different components of the system in isolation and with minimal dependencies to other developer implementations. The development of autonomous robotic systems has a high degree of modularity, which is why layered architectures are prevalent in such applications. The development of an Execution Layer is vastly different from the Decision-Making layers; one requires knowledge of interfacing with robotic hardware while the other is entirely behavioral and can potentially be programmed by psychologists and sociologists who do not have expert-level programming skills. While these two layers will interact, the implementation of one should not be dependent on the other; the intricacies of robotic instructions should not affect the behavioral algorithms of the coordination based levels,

and vice versa. The framework's structure and functionality should fully endorse this approach and allow all layers to be developed independently and provide the necessary interfaces for their integration and potential interactions.

3.2.4. Concurrent Layer Execution

In order to solidify the aspect of isolated development, concurrent layer execution has been imposed as a requirement on the CAS Framework. Without the aspect of concurrency, the execution of the individual layers would become co-dependent, relying on a cycle of execution that is inefficient and leads to several complications that can be mitigated with concurrency. By implementing layer concurrency, execution becomes independent and allows the robot to function at different levels and thus fully utilize its processing capabilities; the higher level layers (strategic, decision-making, coordination) would be able to evaluate the lower level execution, interrupt actions and perform internal processing regarding future actions, all while an action is being performed. In addition, collaboration would very restrictive without concurrent layer execution, since the communication component would only function when lower levels were not busy executing actions. This would make applications very inefficient in terms of processing power, severely limiting the compatibility of the framework with many robotic systems. Even if the communication component alone was isolated from the execution of the layers (sequential layer execution while communication is constantly running), the response time of the layers to both external and internal messages would be extremely long; layers would have to wait for their turn to access the messages that were received by the communication component. Without such restrictions, the layers can freely communicate internally regarding evaluations and interruptions as well externally between concurrent layers in terms of collaboration and collective decision-making.

Table I lists all the requirements been imposed on the framework. The requirements for unrestricted number of layers and roles, the need for user-defined messages, isolated development, and concurrent layer execution are all derived from several examples of autonomous and collaborative systems. On the other hand, the restrictions on the layers' communication capabilities (only parallel layer interaction is allowed for external communication and only neighbor interaction for internal communication) have not been prevalent in collaborative autonomous applications. This feature does not allow the user to overcomplicate the communication avenues and keeps the design compact without eliminating flexibility; this applies to all requirements. Concurrent execution is a functionality that will not affect the user since it does not require any manipulation and runs seamlessly in the background. Furthermore, the restrictions on communication can be bypassed by additional framework functionality (requirement 5). The inclusion of these eight requirements is intended not only to facilitate the development but also to shield the user from over-complicated decisions without limiting flexibility.

TABLE I

CAS FRAMEWORK REQUIREMENTS

1	Unrestricted number of user-defined layers
2	No restriction on the role of user-defined layers
3	Forbid external communication between different level
4	Direct internal communication between neighboring
5	Indirect internal communication between layers
6	User-Defined Messages
7	Concurrent layer execution
8	Isolated layer development

3.3. Layer Structure

The most important aspects of the collaborative model, are the use of independent layers that encompass decision-making/collaboration capabilities and the communication between layers. The components of the layered structure satisfy both aspects and are designed to meet the framework requirements.

There are two categories of layers in the framework (Fig. 2): application, and robotic. The application layers allow the user to define the collaboration and decision-making aspects of the robot. The robotic layers encompass the methods that control the hardware, which can also be simulated. Messages from an application layer to a robotic layer, include commands that must be carried out by the hardware, such as move, rotate or get sensor readings. These commands and instructions that are embedded within the lowest level software of a robot, vary from platform to platform. Due to this variation, it is proposed that a middleware component is embedded at the top of the robotic layer hierarchy. The middleware, is robot specific software that translates commands, to instructions that the robot can execute. Thus, the only requirement for a CAS Framework application to be mounted on different robots, would be to swap the middleware. In case where such software has not been developed, the user can develop his or her own middleware inside a robotic layer. In addition to the middleware, a virtual model can reside within the robotic layers. In the absence of a physical robot, this model could simulate physical actions and sensor readings. Whether a component is simulated or not, the interfaces remain the same, allowing the developer to experiment with simulated runs and later port it to a physical system.

The number of layers that can be created has not been restricted and the user is responsible for determining both their number and their roles (Requirement 1 and 2). In addition,

the order in which the user inserts the layers to the architecture, determines the hierarchy; in the structure that is illustrated in Fig. 2, Layer 1 was inserted first. Therefore, this order will determine communication channels for all layers.

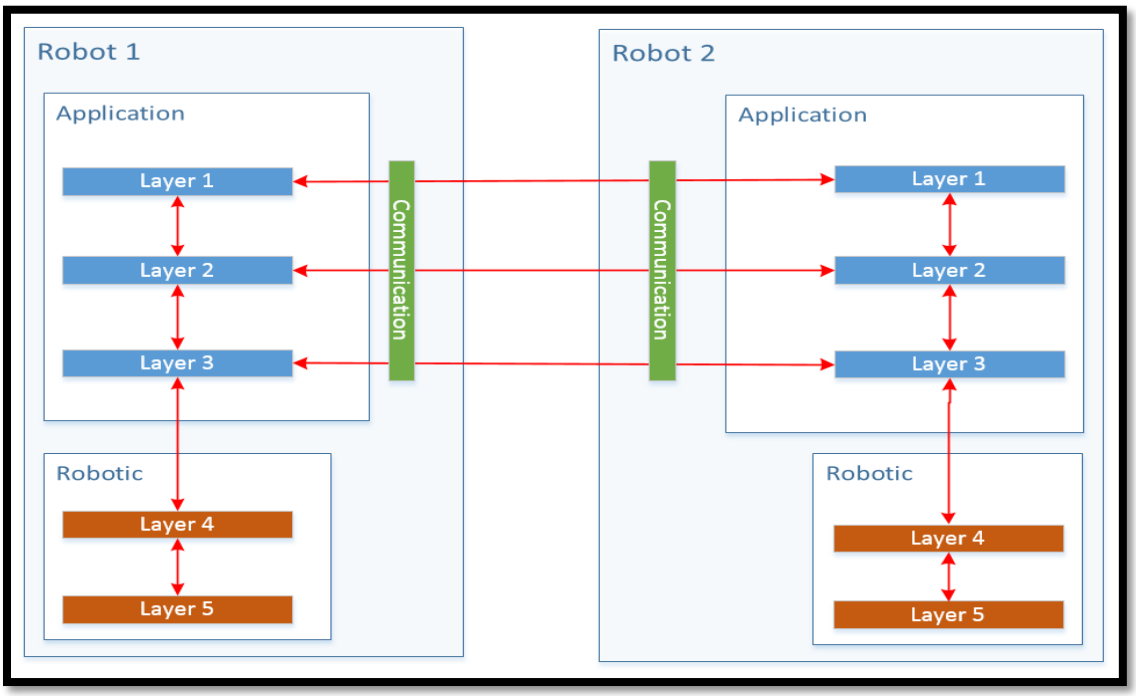

Fig. 2. Layered Structure.

Another critical aspect of the framework is the communication component, which handles external communication (robot to robot). By utilizing this component, the layers can be isolated from the network and thus the user will not be hindered by the intricacies of network communication. Instead, this functionality is assigned to the communication component which is responsible for receiving and sending messages to other robots as well as serializing messages before sending and deserializing upon receipt. The Communication component is isolated and easily replaceable, which might be required when applying the software on a robotic platform;

compatibility issues might arise or a different communication process might be required. Serialization is necessary for physically transferring messages that will be exchanged among robots. Such external messages are restricted to layers of the same level (Requirement 3) as illustrated in Fig. 2. Of course, layers can also communicate internally with layers directly above and below them. This process does not involve the communication component and instead relies on the layers themselves which can interact directly (Requirement 4). Communication between layers farther in the hierarchy can also occur by sending information through all intermediate layers (Requirement 5).

The information that is handled by the layers and the communication component is in the form of user-defined messages which can be manipulated freely as long as the framework specified header is included (Requirement 6). The number of distinct messages to be included in the system is unrestricted. Alternatively, the manipulation of messages is handled differently depending on whether they are internal or external. While internal messages can be manipulated in their original form, all external messages must be serialized and deserialized, a process that is performed by the communication component. External messages are intended to be used by the application layers, since robotic layers usually do not collaborate; of course nothing in the architecture forbids robotic layers from sending external messages. Internal messages, on the other hand, are expected to be passed among all types of layers, both application and robotic.

Furthermore, the messages must compatible with the communication component of the framework. This is accomplished by embedding a rigid structure (Fig. 3) to the user defined messages. This framework defined component includes the information that is required by the framework's message handling components. The Message Id, Message Type, Robot ID, Source Layer ID, and Destination Layer ID, are all utilized during the sending/receiving process. These

attributes are used from the framework to perform serialization and distribute the messages to the layers. Of particular importance is the Message Type attribute which can have three values: Negotiation, Accept, and Deny. If a message is negotiation typed, it signifies that a reply is expected while Accept signifies that a reply is not needed and that the layer simply has to accept it. Deny is used to signify a rejection of particular negation typed messages. This allows for the clear distinction of accepting or denying queries or collaboration requests without requiring manipulation of a message's contents. Despite the necessary inclusion of the message header, no other restrictions are imposed on the messages. The user is free to include any fundamental type attribute, and transfer information among all layers.

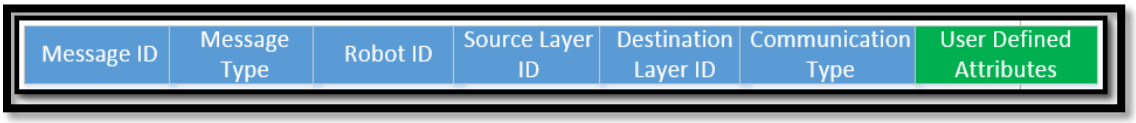

Fig. 3. Structure of User-Defined Messages.

Finally, the robotic layers handle the action that is determined in the application layers. A robotic middleware is utilized to interface with the physical or simulated robot and translate the system's desired actions to commands that the robot can interpret and execute. In order for the middleware to be utilized, it needs to interface with the framework. This is accomplished by inserting a component called pseudo-middleware between the middleware and the robotic layer that issues commands. Implementing the system on different robotic platforms might require the middleware to be swapped. In that case, the only changes in the system would occur in the pseudo-middleware.

The robotic and application layers are executed concurrently by default, without requiring any manipulation by the developer (Requirement 7). Furthermore, one of the most fundamental benefits of the layered architectures, which is the isolated development of layers (Requirement 8), is allowed by the inclusion of interfaces, which define the communication avenues and structure of the framework.

3.4. Object-Oriented Model

An object oriented approach was utilized regarding the design of the framework. The components of the layer structure are represented as objects and their functionality is defined through methods. Additionally, object-oriented inheritance relations and data encapsulation were heavily utilized. Through data encapsulation, the user is denied access to methods and data that should not be manipulated, such as the layered structure, layer id numbers, and the communication methods. Furthermore, through inheritance relations the interfaces are clearly defined and can illustrate which methods are available for use and which are expected to be defined by the user.

3.4.1. Class Structure

Capabilities and features such as internal/external communication and layer structure, are a result of the interactions between classes that represent components of the model. All components that require user manipulation have interfaces which are defined as parent classes and provide access to communication functionality, moldable structures and methods, and access to information, whether that is messages or framework related attributes.

The class diagram in Fig. 4, illustrates the relationship between every class in the framework, including the classes that must be added by the user. Both layer categories and the

communication component are implemented as classes, with the distinction that Communication is a single class while the Robotic and Application classes represent a collection of instantiation.

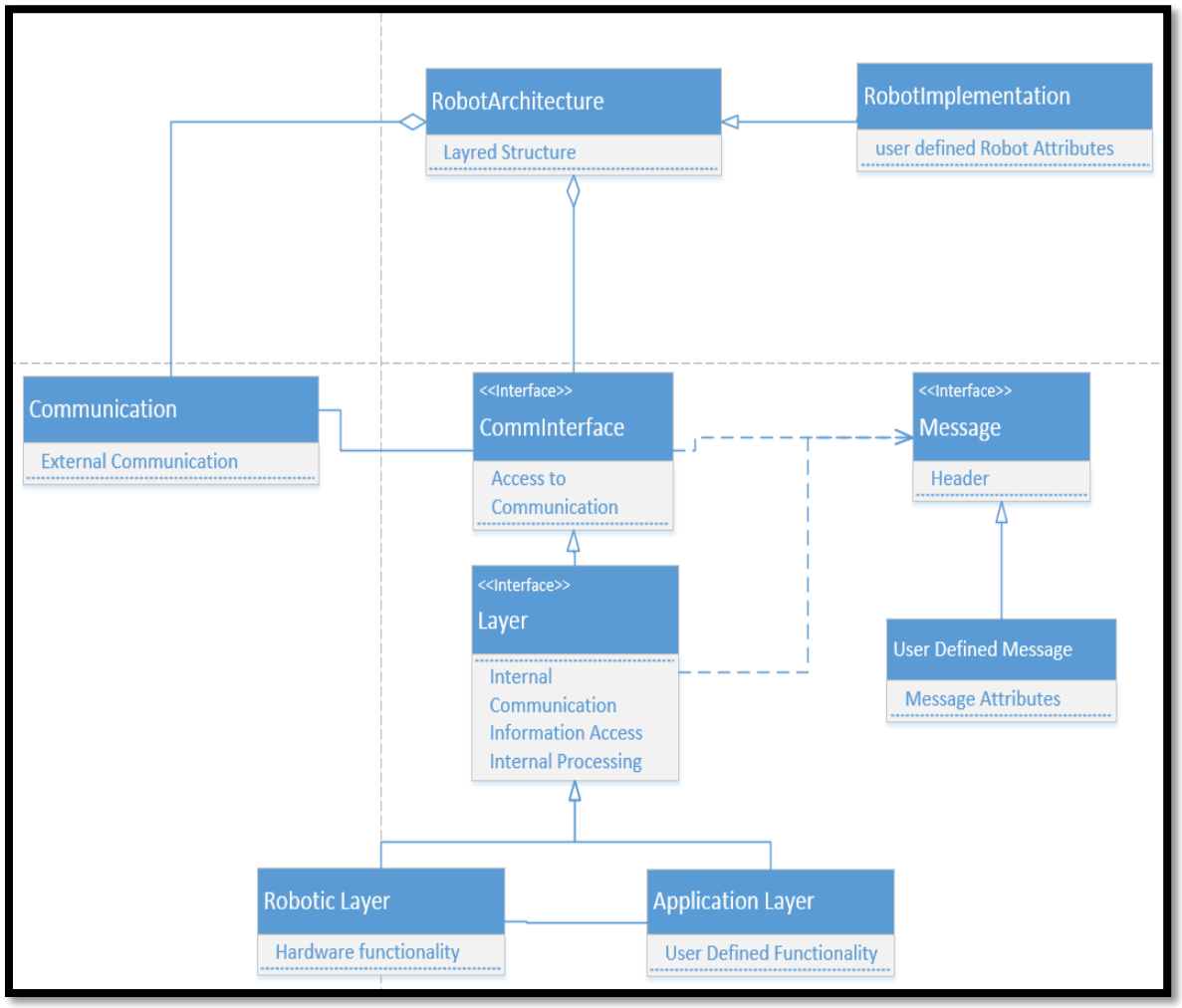

Fig. 4. UML Class Diagram of CAS Framework.

The association relation in the figure, illustrates that two objects can communicate directly. Such is the case with instances of the application and robotic layers. In addition, classes from both layer categories have an inheritance relation with the Layer class, which is in fact an interface. The Layer interface provides methods for internal communication and information access, as well as internal processing methods which automatically hook the layer to the

framework. The same inheritance relationship exists between the Layer and the CommInterface class. This relation provides an avenue for the Layer to use the methods of the Communication class—hence the association between CommInterface and Communication. This indirect association between Layer and Communication eliminates the amount of changes in the Layer in the case of the communication component's replacement or manipulation. In such a case, changes will only occur in the CommInterface class.

On the other hand, the Communication class does not inherit from Layer even though it shares some—but not all—of the Layer's characteristics such as internal processing algorithms and message related structures. In this case, an inheritance relation between Layer and Communication would mitigate the isolation of the user-defined layers and the benefits that were derived from it. This class is responsible only for sending and receiving external messages but has to rely on the CommInterface for providing information to a Layer.

For the transfer of any information, all user defined messages must honor the Message interface. The inheritance relation between Message and User Defined Message allows for a pre-defined header (mentioned in Section 3.3) to be embedded in all messages. Populating this header accordingly, is an absolute requirement which is illustrated by the dependency relation in Fig. 4 between the Layer/CommInterface and the Message class. Without the proper message header, the framework is not be capable of handling messages properly.

All components, with the exception of messages, which are created and then disposed of, reside within the RobotArchitecture class, hence the aggregation relationship between CommInterface/Communication and RobotArchitecture. This class owns attributes (layer list, communication, commInterface) that hold references to each component in the framework and provide this structure—via inheritance—to the RobotImplementation class. This class is where

the user will instantiate all components and perhaps insert additional attributes and methods if deemed necessary.

The inheritance of attributes and methods from an interface to a user-defined class is a common pattern in the framework. This approach, shields critical components from improper manipulation and guarantees the functionality of important features, all the while highlighting which functions must be implemented for user-defined structures to be properly integrated with the framework. The methods that must be defined by the user are related to messages and layer internal processing. Regarding the layers, the logic and sequence of execution will be user-defined. The influence of the user on the layered structure is illustrated in Fig. 5. The figure illustrates that the user injects code inside each layer. The user-defined portions only communicate with the framework-defined portions of the layer. Thus, the user does not have to define any interfaces for user-defined interaction of modules.

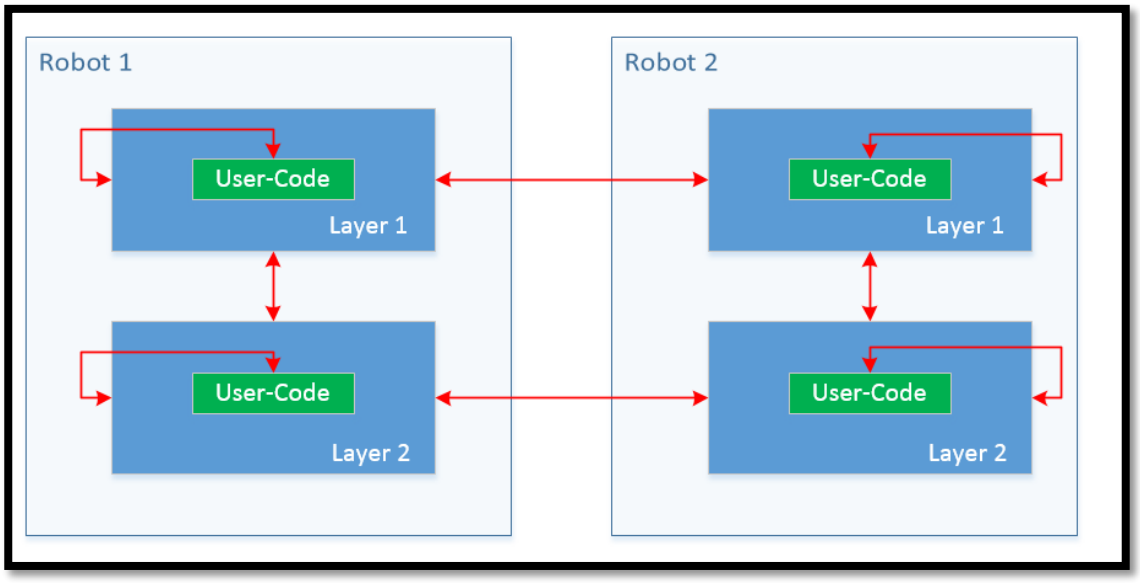

Fig. 5. Placement of User Defined Functionality on Layers.

The structure of the layer and the embedded functionality which is a result of inheritance relations is illustrated in Fig. 6. Every user-defined layer will inherit methods and attributes from CommIntface and Layer. One additional attribute that is inherited from Layer and is also present on the Communication component, is that of a Message Queue. The queue is a structure that holds a list of incoming messages. Populating a queue occurs differently depending on whether the incoming message is a product of internal or external communication. External communication is handled by Communication and CommInterface, which utilize the methods Send and Receive (black lines in Fig. 6) to redistribute the messages of the Communication queue to the appropriate layer's queue. Internal communication (yellow and brown lines) are handled by the layer itself which can directly load messages to the queues of its neighboring layers.

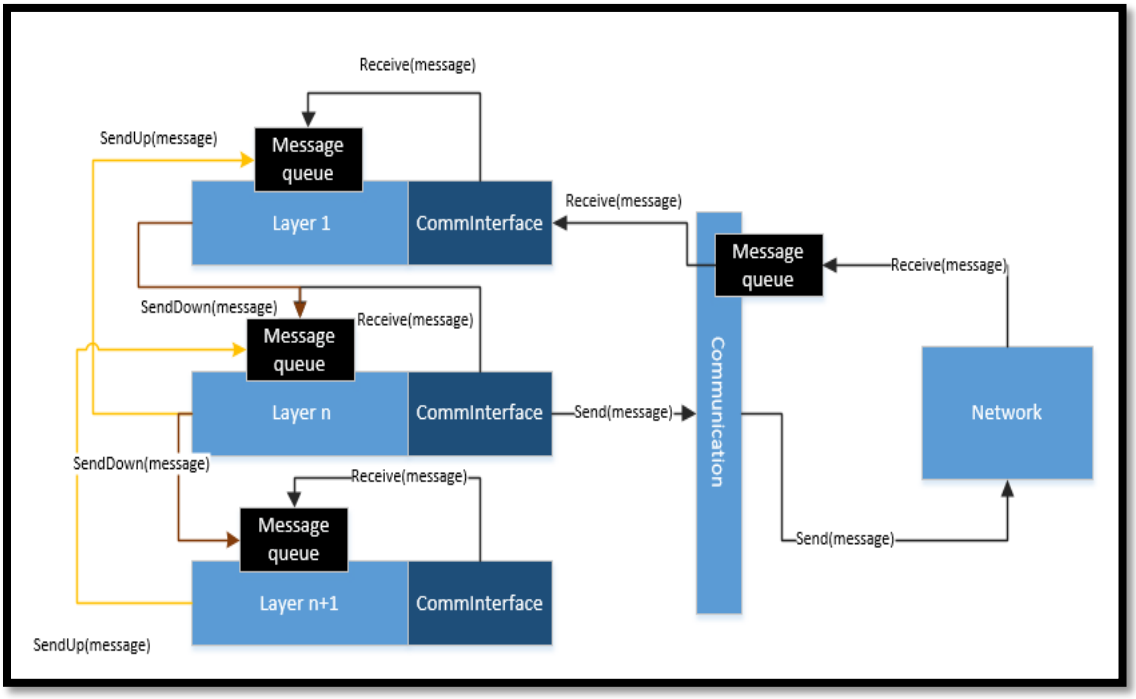

Fig. 6. Detailed Layered Structure of CAS Framework.

3.5. Component Interaction

Sub-section 3.4 described the relationships between the components of the framework and that effect of those relationships on the layered structure. This sub-section will illustrate, via sequence diagrams, the order of operations behind the communication, which is the main interaction between the components of the framework.

The sequence of the external communication process is described in Fig. 7. The process of sending an external message is initiated inside a layer but is executed through the send function which is inherited by CommInterface. Then, Communication's send function is invoked, which will serialize the message and transfer the message to a network and send it to another robot. When receiving a message on the other hand, the first step is the Communication's Receive call, which takes a serialized message that was stored in its queue, extracts it, deserializes it and passes it on to CommInterface's Receive. Finally, this method stores the message in the proper layer's queue. When the layer is finished processing, it will extract this message from the queue and provide it to the user.

As for internal communication (Fig. 8), the process is much simpler. A message is created in the same fashion, but instead of using the CommInteface's methods, the user can call Layer's inherited SendUp or SendDown—internal communication is only possible with neighboring layers—which will directly load the message on the proper queue. It is worth noting that serialization is not necessary during internal communication.

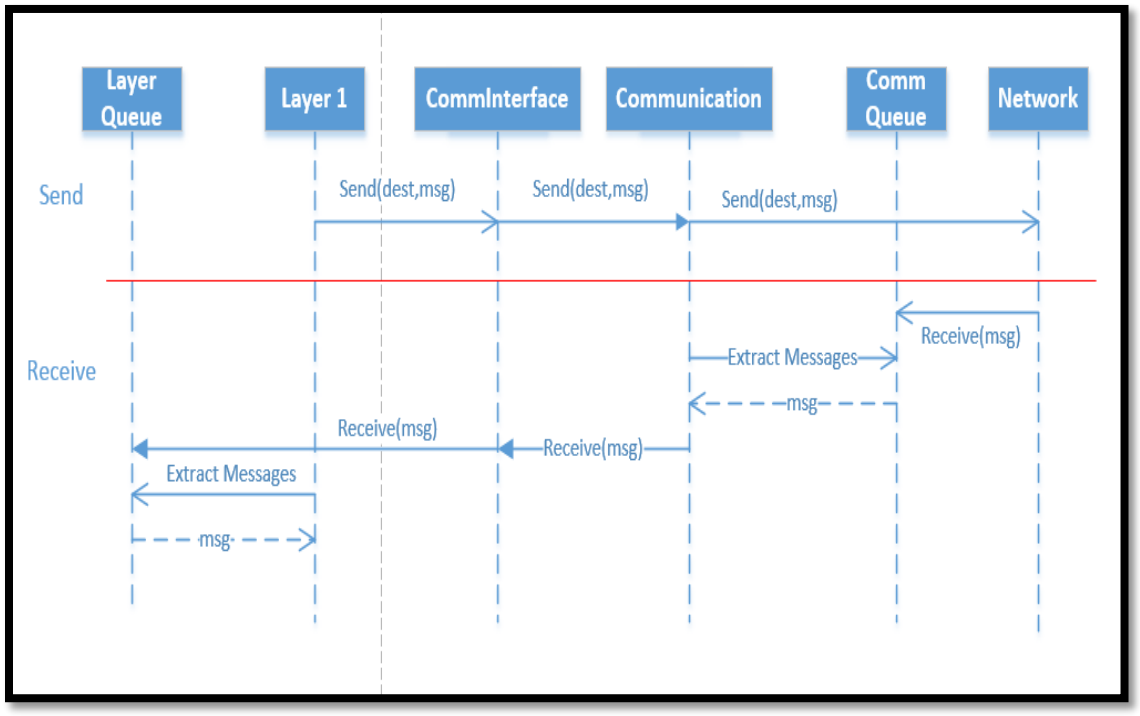

Fig. 7. External Communication Sequence Diagram.

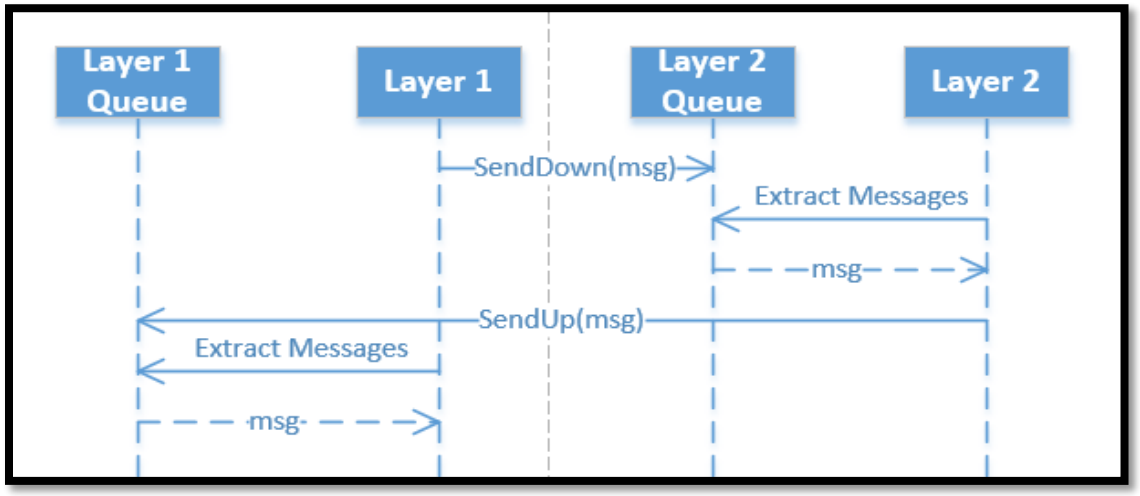

Fig. 8. Internal Communication Sequence Diagram.

3.6. Layered Architecture Benefits

The CAS Framework architecture introduced in Section 3.3 and analyzed through its object-oriented elements in Section 3.4 is intended to facilitate the development of a collaborative autonomous system. While maintaining several aspects that were prevalent in collaborative and autonomous systems (layered structure, concurrent execution, etc.), additional restrictions and features were included (parallel and internal communication).

By utilizing this framework, the user already has access to internal and external communication methods that are not application specific, all the while maintaining the capability to define the information that must be exchanged by creating messages. Despite the inclusion of communication, which can be a very significant portion of a system and can require extensive testing to guarantee integrity, the framework provides a clear structure for different levels of the system to be implemented. Creating a hierarchy of levels is easy and intuitive since the entire structure is already provided and only requires a number of layers to be specified and mapped to the provided structures. These structures are modular and so are the supporting elements of the framework, allowing not only for fast isolated development, but also for easy swapping of entire components. Entire layers could be replaced and even the communication component without affecting the surrounding system. The same modular approach is what allows for the framework to be implemented across multiple platforms given that the appropriate middleware have already been developed.

The framework provides significant functionality and an effective layer structure allowing the user to develop the logic of the layers without worrying about the intricacies of structure and communication. Since the modularity aspect framework also allows the swapping of elements that might be unsatisfactory (different communication system, middleware), the user

is awarded significant flexibility all the while saving time from redeveloping widely used functionality.

CHAPTER 4

SOFTWARE DESIGN

In the Chapter 3, the model of the CAS Framework was introduced. This chapter introduces the various methodologies and concepts behind the software design of the CAS Framework; all components are defined and their algorithms described. In-depth discussion of aspects like time management, concurrent layer execution and message-handling illustrate the developer's role in the algorithmic design, highlighting concepts that he must be aware of and how to effectively use the framework's features. Finally, an example application is introduced and the process of implementing it in the CAS Framework is described and evaluated in terms of effectiveness and ease of implementation.

4.1. Modes of Execution

The CAS Framework has two modes of execution which distinguish whether the Application layers interface with physical or simulated hardware. In the Operational mode, robotic hardware is accessible and can be utilized to gather data from the environment. In the Simulation mode, such hardware is absent and models are used to simulate physical actions and gather data from a virtual environment. Those two modes affect only the Robotic layers which contain either simulated models or software that interfaces with the hardware. While there are certain differences and characteristics among the two modes which either expand or limit potential features, there is no effect on the Application layers which function independently and remain unchanged regardless of the execution mode.

The Simulation mode relies on kinematic models to simulate the execution of physical actions. By utilizing such models, calculations regarding the duration of an action are performed

and the Robotic layers sleep for the calculated duration, thus simulating the action. Based on this approach, the Robotic Layer's processing functions are delayed in real-time and can provide insight regarding realistic running times and system behaviors that would be identical if the application was mounted on a physical robot. Time management within the Simulation mode is necessary in order to effectively simulate actions but it also provides potential that is not possible in the Operational mode. Since it is the Robotic layers that determine the sleeping time, the simulated duration of all actions can be scaled down, leading to decreased running times for simulated systems, without any consequences on system behavior.

The potential for execution time scaling and the responsibilities that are associated with time management are not present in the Operational mode. In this case, the physical hardware simply performs certain actions and is not susceptible to any time-related manipulation. Time requirements and shortened execution times can only be handled if the actual commands contain related parameters such as velocity and acceleration, in which case the hardware functions at a pre-defined pace.

Similar to the Operational mode of the Robotic layers is the execution of the Application layers. Despite the isolation of each layer and the aspect of concurrent execution which leads to different processing times, the application layers are not associated with any time management and do not need to be synchronized; their execution and knowledge of time is collectively based on the Wall-Clock and they perform series of operations without time requirements. The Application layers are iteratively involved in the issuing of commands to the Robotic layers and time management does not influence this process. A layer completing operations faster than the layer above it does not signify a lack of synchronization, because the layer above has already completed relevant calculations (or is entirely uninvolved depending on the application).

Otherwise the current layer would not have initiated its current calculations; any parallel processing of higher level layers is directed at future planning.

Since the Robotic Layers run in real-time and the Application layers share the same clock without regard for time, the user does not need to be concerned about layer synchronization issues or time management. Fig. 9 illustrates the structure and dependencies of the components associated with the two execution modes. Only the kinematic models of a simulated robotic layer depend on time whereas the Robotic layers with hardware interfacing and application layers remain independent. The layers simply react to external and internal input, regardless of communication latency and without causing complications for the lower layers. If the concept of time is vital for an application, the user will have to include a user-defined clock. Such a clock should be shared by all layers, in order to avoid synchronization issues.

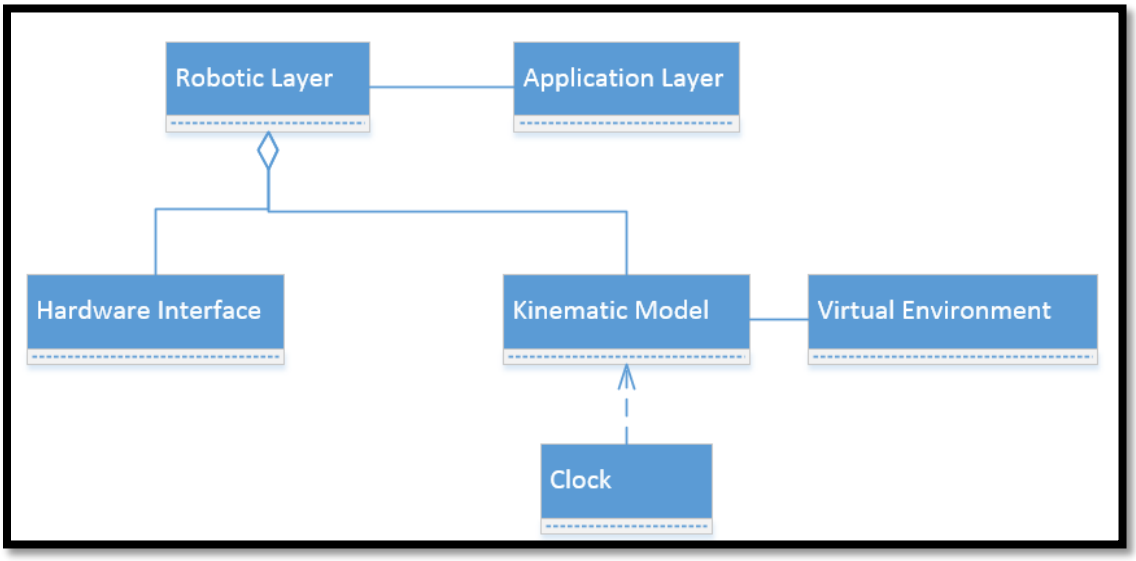

Fig. 9 Time Dependencies.

4.2. Class Definitions

This sub-section introduces the class definitions, describing the main functionality of each component and the most important methods that comprise them. The main components are Layer, CommInterface, Communication, and Message.

Layer is the base class of every layer in the framework. It serves as an interface for the application and robotic layers, providing methods for internal communication, internal processing, and accessing information. Layer processing involves alternating between message checking and executing user-defined functions; Fig. 10 illustrates this process. MessagesPending is the method responsible for checking the contents of a layer's queue, which contains messages, internal and external, for the layer. The next method executed is InnerProcess which is an empty method, intended to be populated with function calls to user-defined classes that define the layer's behavior and sequence of operations. When this sequence of user-defined function calls is completed, control returns to MessagesPending and the same process is completed iteratively so long as a message exists in the queue. The loop containing these two function calls, concludes with an evaluation of the layer's termination conditions. TerminationCondition is another empty method defined by the user for each layer and determines when the continuous loop of message checking and executing user-defined functions will end.

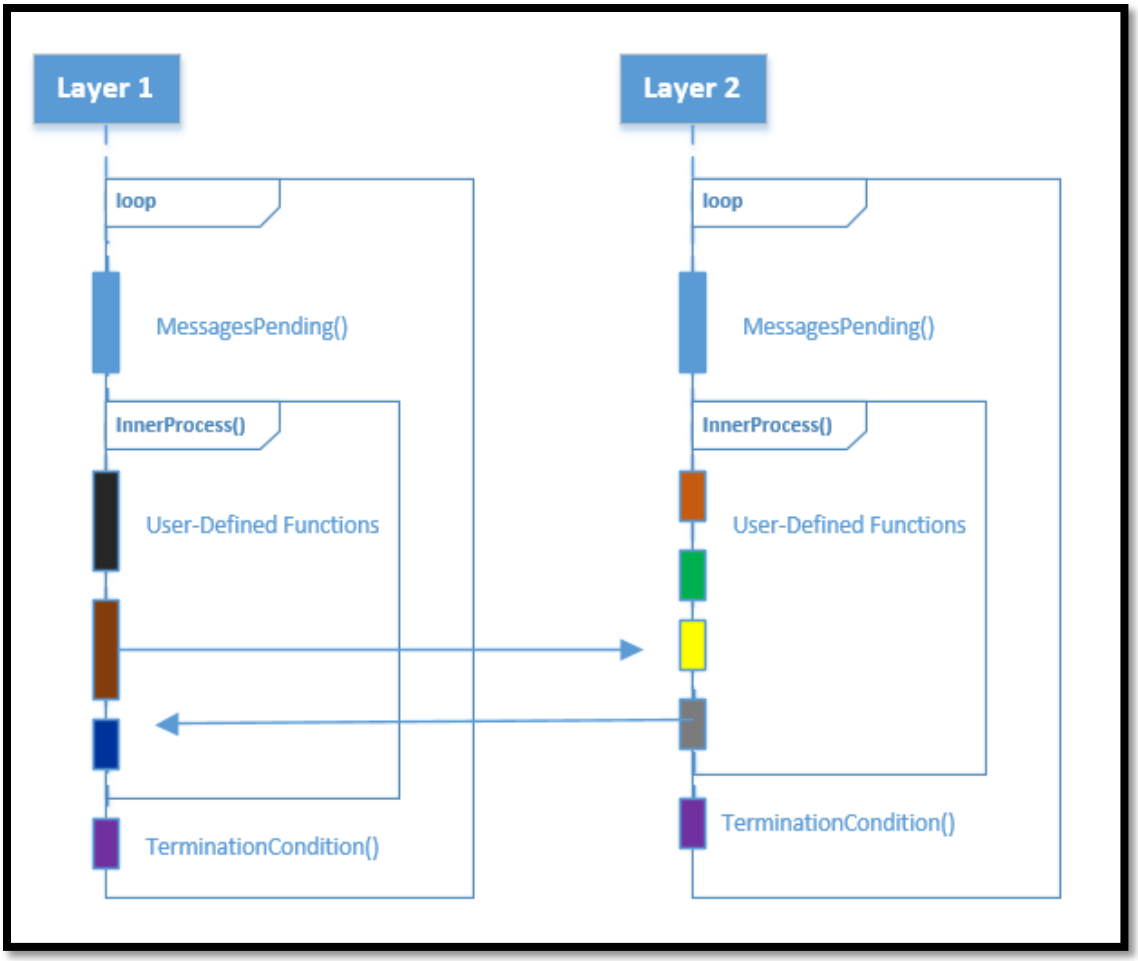

Fig. 10. Internal Processing Sequence Diagram of the Layer Interface.

Layer also provides information access methods which are invoked when MessagesPending detects a message in the queue. Accessing a message occurs by defining an empty method that includes the message as a parameter. The user is responsible for storing the message or calling additional user-defined functions before control leaves the scope of this function and the message is lost. This empty access method is different for external and internal messages; external messages come from a same level layer that belongs to another robot while internal messages come from within the robot, either from a layer above or below in the hierarchy. If the message is external then the empty HandleMsg function is called. If the message

is internal (Fig. 11), the layer-defined ReceiveIntraMsg method will be called. This function does not immediately give control of the message as the HandleMsg method did. Instead, it either forwards the message to another layer in the hierarchy or invokes StoreIntraMsg (as in Fig. 12) which is the equivalent of HandleMsg for internal messages and gives access to the message. It is worth noting that internal communication can be initiated by the Layer's methods, SendUp and SendDown. Since the ReceiveIntraMsg is already defined by the framework, the user's only responsibility regarding message handling is defining HandleMsg and StoreIntraMsg.

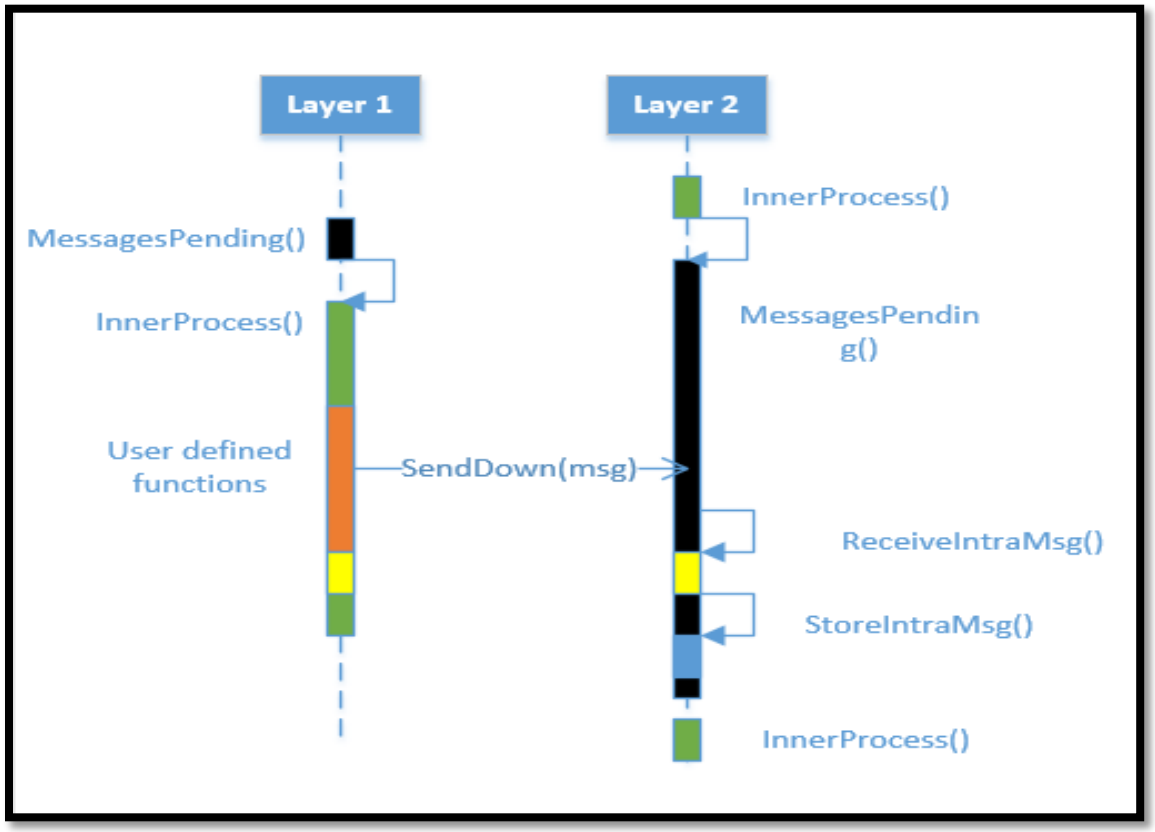

Fig. 11. Internal Layer Message Handling.

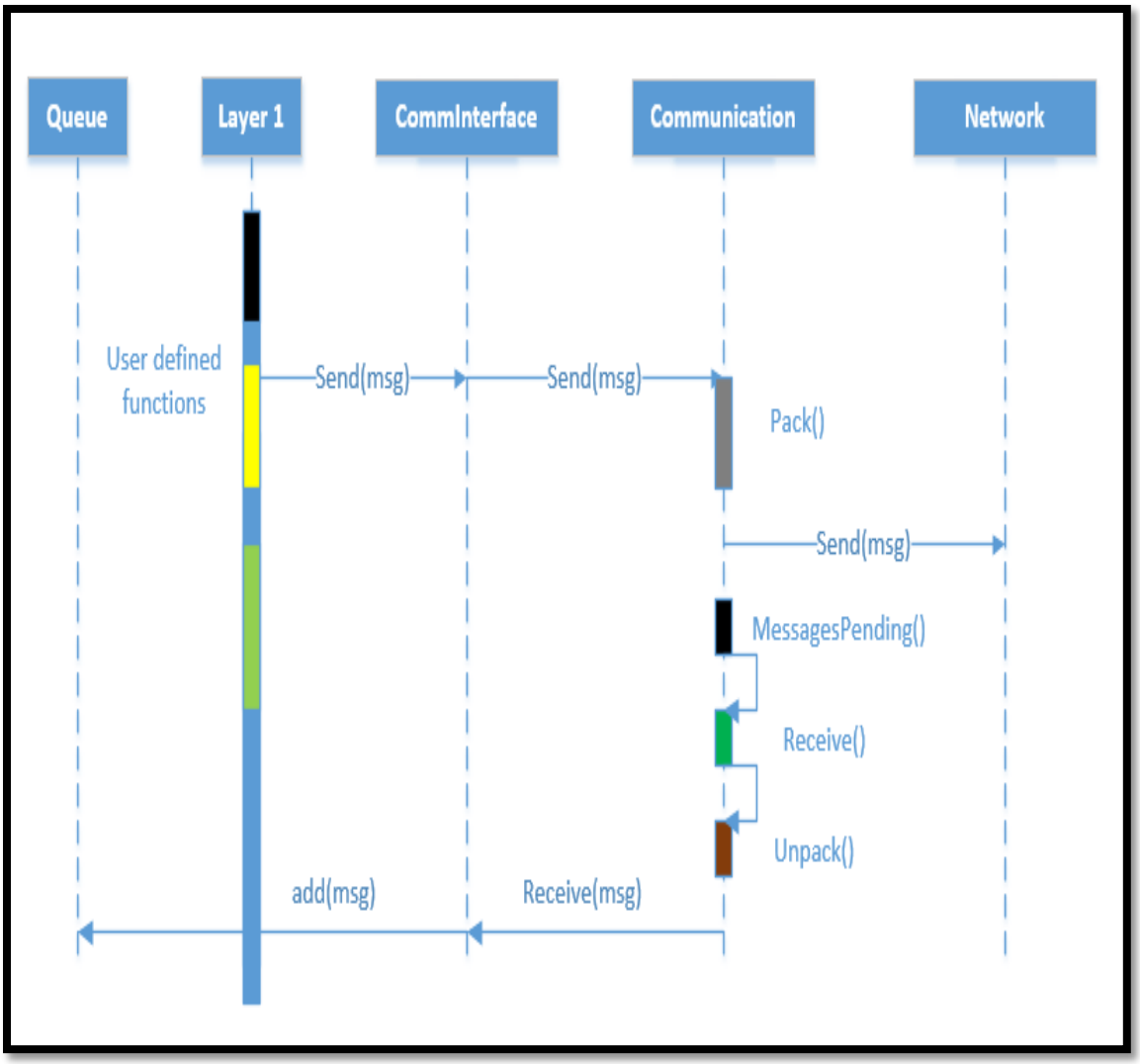

Fig. 12. Interaction of CAS Framework's Main Components.

CommInterface is the base class of Layer and allows all user-defined layers to interface with Communication and vice versa. This is made possible due to the association relation of CommInterface and Communication, which allows the base class of the user-defined layer, to invoke Communication's public methods; since Layer has access to CommInterface's public and protected methods through inheritance, Layer indirectly has access to Communication's methods. Besides being an intermediate for sending external messages, CommInterface is

responsible for placing incoming messages from external robots to the layer's queue, which is accomplished by invoking the Receive method. Unlike the Layer class, CommInterface is completely defined and does not require any user manipulation, in order to interact with Layer and Communication.

Communication contains the external communication functionality allowing it to send messages to other robots through a wireless network. This class shares some characteristics with Layer; it contains a queue of serialized messages (layer queues contain deserialized messages) that is continuously checked through the MessagesPending method (this method is identical to Layer's counterpart). Of course, the communication component does not inherit from CommInterface or Layer and thus has no direct access to the user-defined layers. Instead, it relies on CommInterface to physically load messages on layer's queues. Finally, Communication is responsible for serializing (Pack) and deserializing (Unpack) messages.

Message is the base class of any user-defined message. The user is free to define a user-specific message class as long as the Message base class is its parent. By inheriting from Message, the framework imposed message header is attached to the user-defined message. This header is what ensures compatibility of a message with the CAS Framework's communication mechanisms. The Message interface also provides methods for serializing (Pack) and deserializing (Unpack) the header but the user must provide similar methods for the user-defined attributes.

With the conclusion of all the component's definitions, detailed component interactions are described. Fig. 12 illustrates the basic interactions between Communication, CommInterface, and a user-defined layer. In the figure, Layer 1 sends a message by invoking CommInterface's Send method, which in turn calls Communication's Send. Then, Communication will serialize

the message with the Pack method and send it to the network. This concludes the sending process. Communication then continuously checks its queue for external messages. Upon finding a message, MessagePending calls the Receive method, indicating that a message has been placed in Communication's queue. The Receive method deserializes the message through the Unpack method, and CommInterface's Receive is called to place the message on the appropriate layer's queue.

4.3. Algorithm Description

User-defined functionality is inserted to the framework in the form of user-defined message and layer classes. Of course, their methods are invoked in specific places of communication, message, and layer processing algorithms. Knowledge of these algorithms and critical aspects such as concurrent layer execution and the sequence of their operation, while not necessarily required, provide great insight to the role of the developer and how to fully utilize the framework's features. Message processing, interfacing, communication types, and the process of creating layers and messages, are fully explored in this section. An example application is also introduced to demonstrate potential interpretations for the layer structures and show how user-defined algorithms are injected in the framework.

4.3.1. Concurrent Execution

One of the aspects that greatly facilitates the development of autonomous robotic systems is concurrent layer execution. The benefit of following such an approach is that it isolates the execution of the layers and eliminates the nuances involved in the alternative, sequential execution. Concurrently executed layers can independently traverse the internal states with an

event based approach and communicate with each other without affecting the surrounding system.

Fig. 13 illustrates the event-based approach behind concurrent execution. Layers are inactive by default and are activated by messages, which carry information about external events that have an implication on the layer's execution; a query message from another layer and the issue of a specific actions are examples of such events. Upon activation, the layer reacts based on the contents of the message by executing a user-defined function. Upon completion, the layer's execution is suspended until a message triggers further reactions. This approach is also followed by the Communication component which also runs concurrently to the layers. This isolates the Communication component, allowing it to react to incoming messages regardless of what the surrounding system is doing. The distinction is that the Communication component's reaction is limited to transferring messages to CommInterface and then immediately suspends execution.

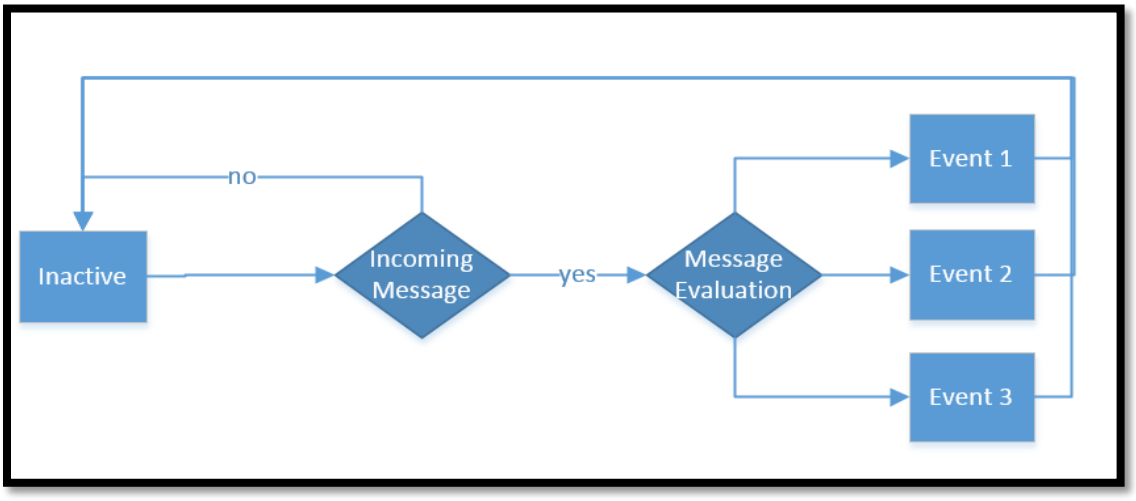

Fig. 13. Event Based Layered Execution.

Concurrency provides several benefits. Layer interruptions can be implemented; a new message arrival could stop the current reaction and lead to another event. This would be especially useful in the robotic layers where actions take longer and are more likely to be interrupted. Without concurrent execution, control would remain with a specific layer, making further processing impossible to perform and thus forbidding interruptions. Multi-level processing is another benefit of the concurrency approach and allows the robot to fully make use of its processing power which would otherwise be limited to one layer at a time.

Concurrent execution was implemented with the use of threaded functions. The implementation of this feature is made under certain assumptions. The implementation language is assumed to support multi-threading. Furthermore, multi-threading requires significant computational capabilities which is assumed to be present in the robotic hardware. It is unrealistic to impose this requirement on all current robotic platforms. However, computational capabilities are advancing and it is expected that near-future platforms will meet this requirement.

Threads allow for a specified function to run concurrently with the function that invoked it and other threaded functions. This functionality is embedded to the RobotArchitecture and only requires two calls per robot: StartProcess, and EndProcess. These two functions resolve the intricacies of thread coordination and termination. StartProcess will initiate internal layer processing for all layers while EndProcess will be called when layer termination conditions have been met. It is worth noting that the concurrent execution can cause complications regarding structures that are accessible by the threads. In this case, the queues can be subject to corruption due to multiple threads inserting and extracting messages at the same time. This complication is avoided by having the layers reserve the queues through using a mutex to lock the queue access

for the duration of the insertion/extraction and releasing the reservation as soon as this process is completed.

While the threads run in the background and do not require further user interaction, their effect can be noticeable in the processing of the layers and the timing of their interactions. The processing times of layers will inevitably vary, causing a non-consecutive order of execution that the user must be aware of. In Fig. 14, which illustrates the processing of three distinct layers, we see that Layer 2 sends an internal message towards Layers 1 and 3, with Layer 1 taking precedence. The layer's thread alternates between the execution of user-defined functions inside InnerProcess and MessagesPending. In this example, the orange bar in Fig. 14, represents a function that will be called upon receiving the sent message.

Layer 1 might be expected to react first since the message towards it was sent earlier. As the figure illustrates, the inner processing of each layer can vary greatly. In this case, Layer 1 has a significantly longer processing time, allowing Layer 3 to check for messages earlier and thus execute the user-defined function first.

The threads are terminated only when layer specific conditions have been met. Such conditions are placed within the TerminationCondition method which is invoked at the end of each process cycle. Concurrent layer execution is not necessary, but since its advantages are numerous and the fact that it does not require any user input provide no reasons for not including it. With this feature, the layers and the communication component are more effectively isolated and user is able to better utilize his platform's processing capabilities.

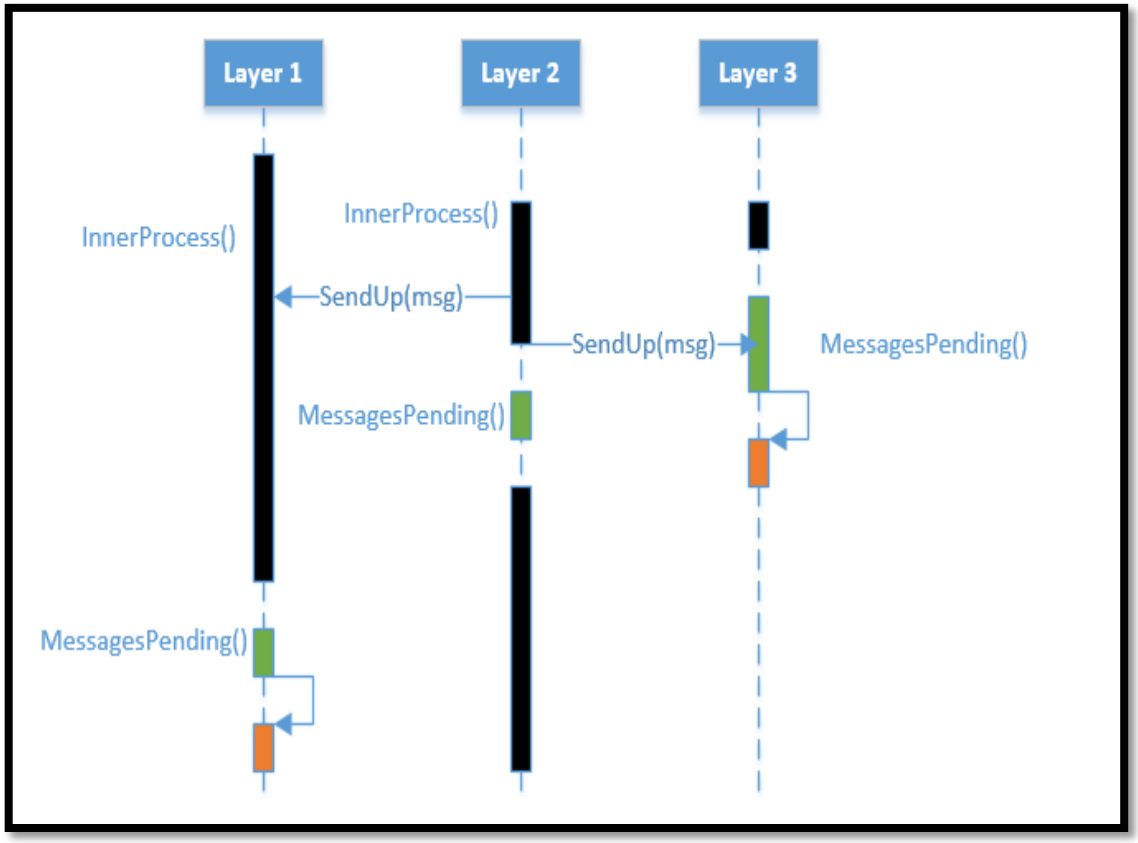

Fig. 14. Communication between Concurrently Executed Layers.

4.3.2. Message Processing

In Section 4.1, it was mentioned that Communication and Layer distribute the message which are in turn made available to the user by HandleMsg and StoreIntraMsg. Message processing relies on the proper implementation of the message header which includes the layer id of the source layer, the layer id of the destination, the robot id, the message id, and the message type discussed in Sub-section 3.3. While the message id is mostly used for statistical and testing purposes, the remaining attributes must always be included in the user defined message. These attributes, which are inherited from the Message interface, must always be populated since the communication mechanisms rely on them for distribution and proper deserialization.

There are two places where message extraction occurs in the framework: in the communication component and the layer. In the first case, message handling is always the same and it involves deserializing the message (by calling unpack) and passing it to the CommInterface by calling the latter's Receive method, which loads the message to the queue by invoking the Add method. Message extraction from the Communication queue and distribution (Fig. 15) is simple because it only contains external messages. The queue in the Layer on the other hand, can contain both internal and external messages. The process in this case depends on the destination and layer source id numbers. When those two attributes are the same, indicating the message was sent and received at the same level of the layer hierarchy requiring that the message originated on another robot, the message is identified as external and the HandleMessage method is called, thus giving the user immediate access to the message. When they differ on the other hand ReceiveIntraMsg is called (Fig. 16). This function compares the id numbers and determines whether the message must be forwarded upwards, downwards, or simply accepted by the layer.

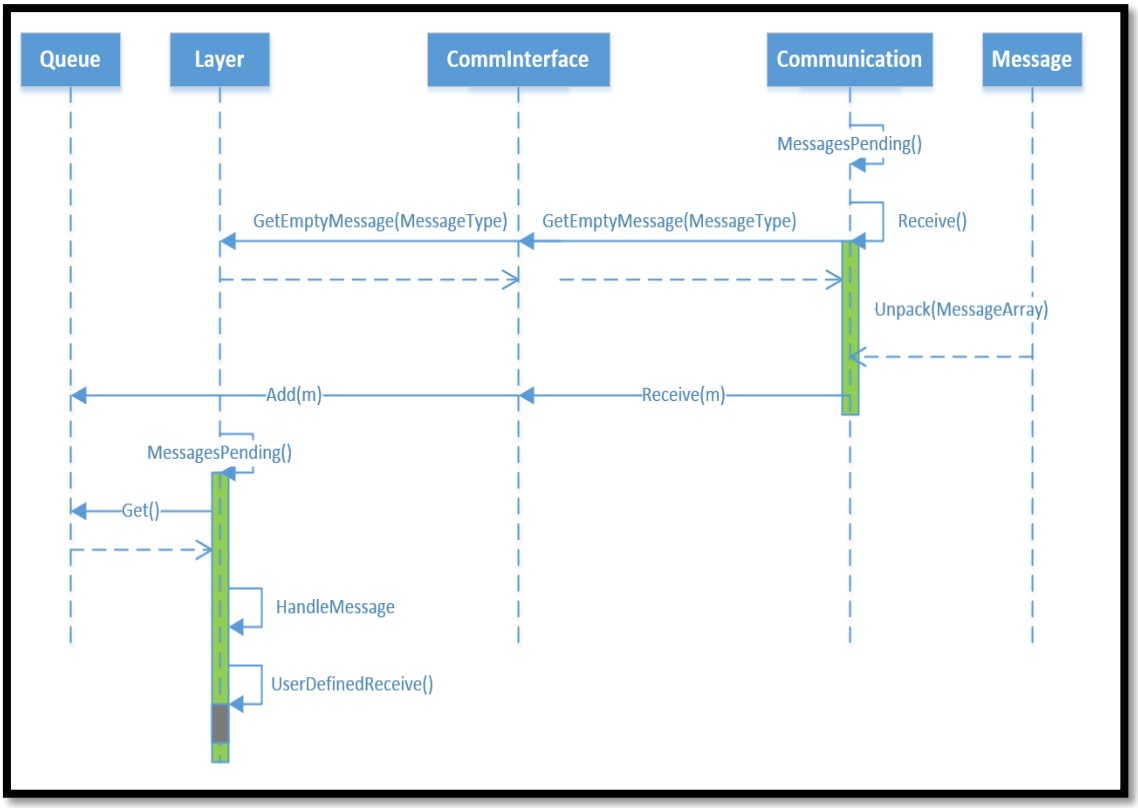

Fig. 15. External Message Extraction and Distribution.

Forwarding messages occurs when a message is received by a layer that is not the destination. This occurs during internal communication between non-neighboring layers since messages have to pass through all intermediate layers. Forwarding is performed by the methods SendUp/SendDown. When the message reaches its destination, the message type is evaluated. If an internal message is branded as a Negotiation type, it means that the user expects another internal message to be sent back to the source. In that case, the process is automated and all the user has to do is define the function GetIntraMessageReply which supplies a reply message based on the type of the incoming message. The usefulness of this method, is that it removes the burden of additional message coordination and delays, thus keeping the message processing components separate and compact. If the internal message reaches its destination without a

negotiation mark (Accept or Deny), then StoreIntraMsg will be called and the user will be given access to the message. This process is illustrated in Fig. 17.

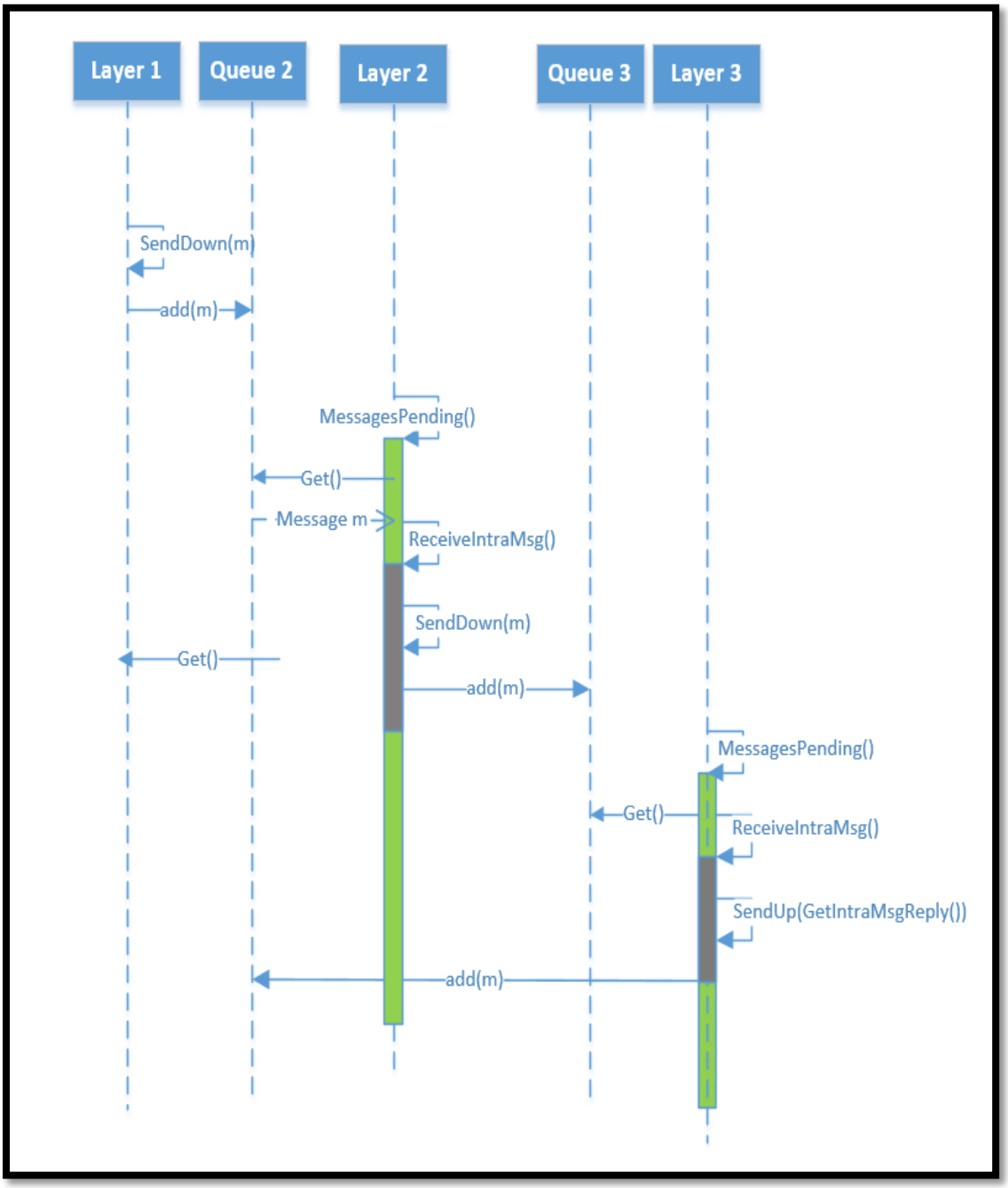

Fig. 16. Internal Message Handling Sequence Diagram.

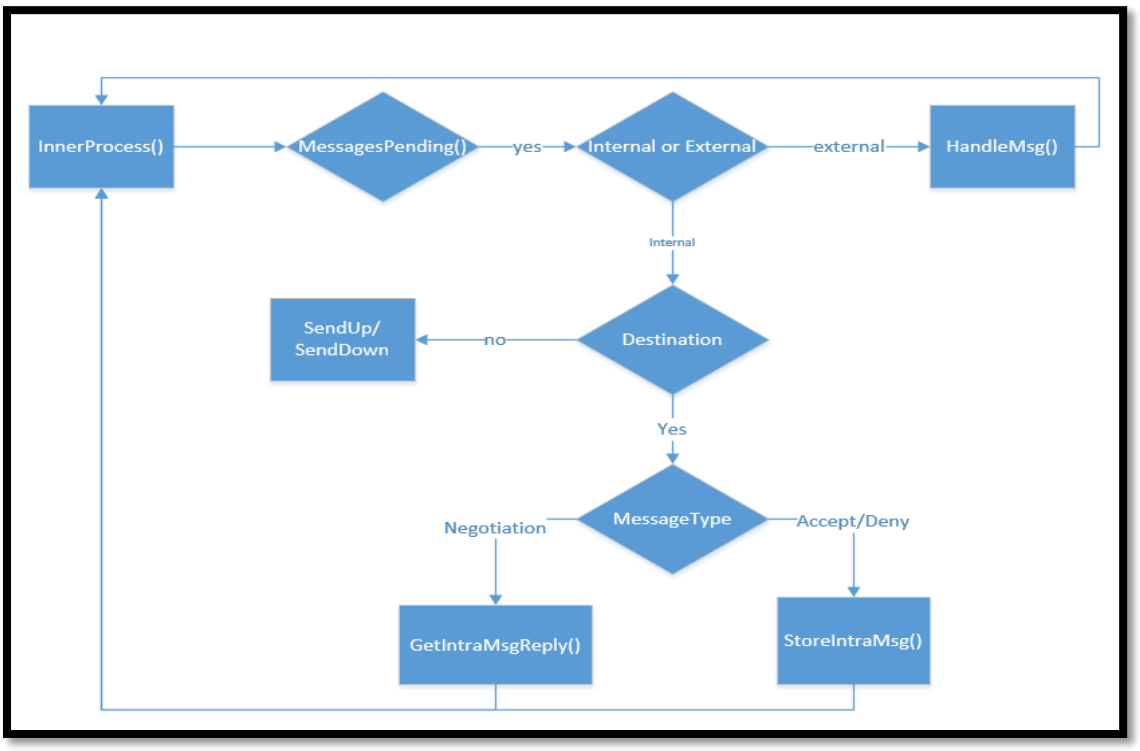

Fig. 17. Message Handling Flowchart.

4.3.3. User Code Interfaces

The purpose of the CAS Framework is to allow development of applications, which requires the freedom to define layer and message classes as well as adding any number of methods that are required by the application. In order for those classes to be fully integrated to the framework, the user must define specific methods that are inherited from interfaces and populate specific structures. This subsection will explore the processes of interfacing with the framework.

The layered structure resides within the RobotArchitecture class, which is itself an interface and thus should not be manipulated. This empty structure is inherited by the RobotImplementation class which is where the user makes the first step in application development. This involves instantiating objects of the user-defined layers and inserting them

into an array structure that was inherited by RobotArchitecture. The integration of the layers in the framework requires the additional step of defining the connection of each layer to the queues of its neighbors; this enables the internal communication functionality. In order for this to be accomplished, a simple call of RobotArchitecture's ConnectLayers is required. This automatic integration process, is performed under the assumption that the layers will be inserted in decreasing hierarchical order. In the aftermath of the layer integration, the framework would resemble the structure of Fig. 18.

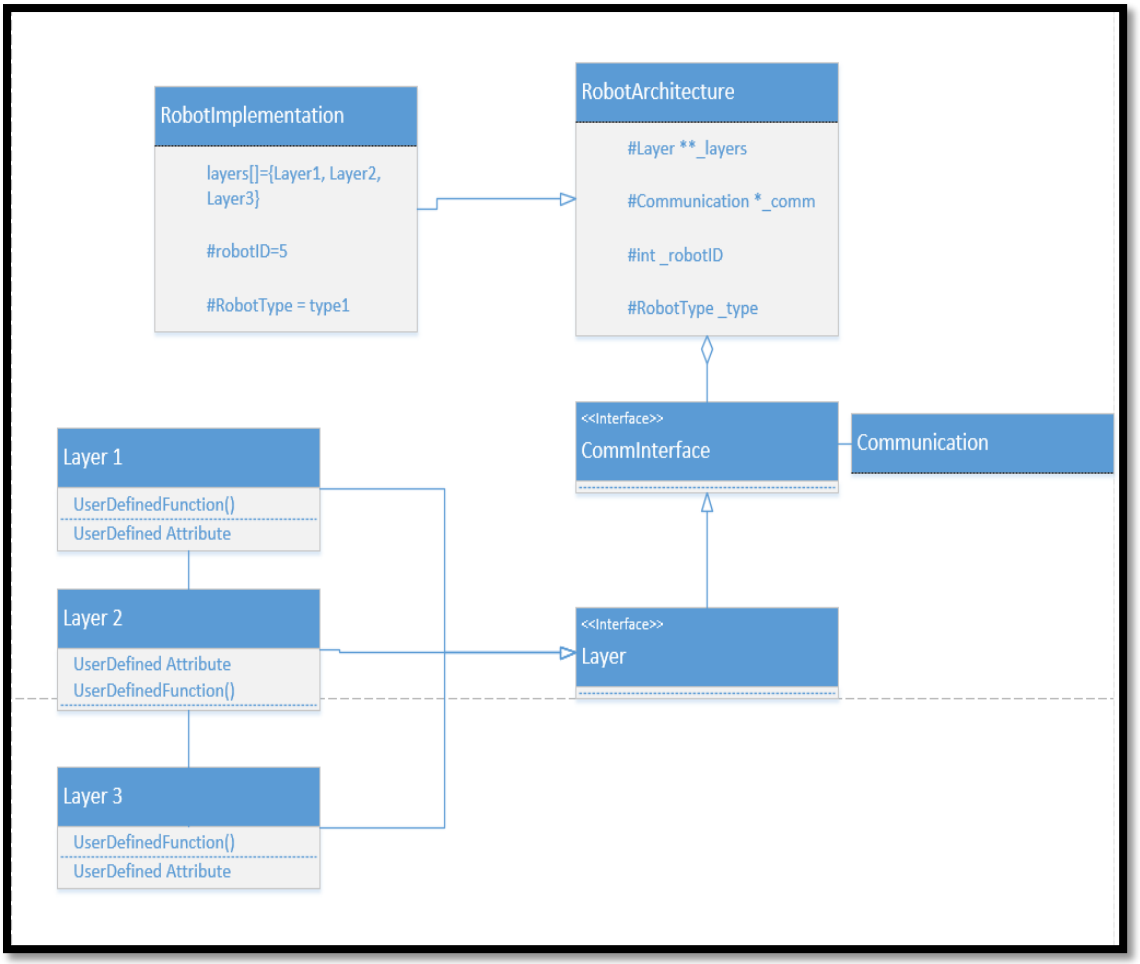

Fig. 18. CAS Framework with Three Layers Defined.

With the layers properly defined and interconnected, the user can begin to define new methods to provide system behavior. Certain functions, marked as virtual in the interfaces, must be defined as well. Layer creation requires the definition of the Layer interface's virtual methods such as InnerProcess and HandleMsg which are in fact the places where the user can make use of his newly defined methods.

The contents of InnerProcess will determine the layer's sequence of operations. Functions that are called inside this method are expected to initiate collaboration with other robots and request certain actions from lower layers in the hierarchy. HandleMsg, on the other hand, defines event based reactions (state changes) of a layer to incoming messages. In this method, the user can call newly defined functions based on specific input and force the layer to react in a way that could potentially alter inter processing as well. Defining all virtual methods is the last step of the layer definition process (Table II.) and thus the layer has the capability to use all of the framework's features.

TABLE II

LAYER CREATION PROCESS

1	Create a Class for every desired layer
2	Create inheritance relation between each new class and the Layer class
3	Instantiate Layers array in RobotImplementation
4	Call RobotArchitecture::ConectLayers()
5	Define all virtual methods of Layer in every user defined class
6	Use CommInterface methods for external communication
7	Use Layer methods for internal communication

Defining a message is a much simpler process (Table III). The number of virtual methods is very limited and only includes Pack, Unpack, and Bodysize. Pack and Unpack are the methods that perform message serialization and deserialization while Bodysize provides the size of the message in bytes. Additionally, supplying a default constructor is also required for the deserialization purposes.

TABLE III

MESSAGE CREATION PROCESS

1	Create a Class for every desired message type
2	Create inheritance relation between each new class and the Message class
3	Define a Pack method for each class based on the defined Message:Pack
4	Supply an empty, default constructor

The framework requires the user to create and manipulate content on very specific places as mentioned in Tables II and III. In the same fashion, there are specific elements of the framework that the user should not manipulate due to the risk of malfunction. The interfaces: Layer, CommInterface, and Message, should remain intact, since they provide guidelines on the user's definition responsibilities and define the structure and internal communication capabilities. Similarly, the communication component is fully functional and does not require additional editing.

4.4. Application Development Example

In Section 3.2.1, a simple collaborative autonomous system was introduced for demonstrating its translation to a layered model. The scenario involved two robots that were tasked with picking up objects and moving them in a two dimensional plane. A possible layered interpretation of this example contained the Decision-Making, Task, Coordination and Action layers. Fig. 19 illustrates the resulting structure. For the architecture to take the illustrated shape, the Decision-Making layer must be inserted at index 0 of the RobotArchitecture's layer array (Task at index 1, Coordination at index 2, and Action at index 3). Finally, a call to ConnectLayers must be made.

In terms of the user-defined methods of the layers, some are layer specific and others are identical for all. The TerminationCondition will be return true, when five objects have been moved to their destination; this condition will be the same across all four layers. InnerProcess on the other hand will vary. Then inner process of Decision-Making will include the following functions: SelectObject, and InformTask. The Task layer's inner process function will include HelpRequest and InformCoordination. Coordination's methods will be AgreeOnBalancing and InformAction. Finally, the Action layer's inner process functions will be PickUp, Move, and Scan.

With an object selected, the two robots must communicate this selection via the SelectedObject message and make sure that there is no conflict. Then, an identification of the object that must be moved is sent to the Task layer via the InformTask function and the ChangeStatus message.

When the task is informed, it will call the HelpRequest function, which will determine if the object can be moved without assistance based on the robot's specifications. If the

specifications are adequate, then the InformAction method will be used; otherwise, a RequestPartner message will be sent to the other robot. After several messages have been exchanged, the robot will have knowledge on whether it can move the object on its own or not. If collaboration is required the coordination level will be informed and AgreeOnBalancing will be called, exchanging coordination information. When this process is complete, InformAction will be called which will send a ChangeStatus message. The Action layer will then proceed with alternate calls between PickUp and Move until the process is complete. The entire interaction between the two robots can also illustrated in Fig. 20, which does not include layers but instead the robot as a whole.

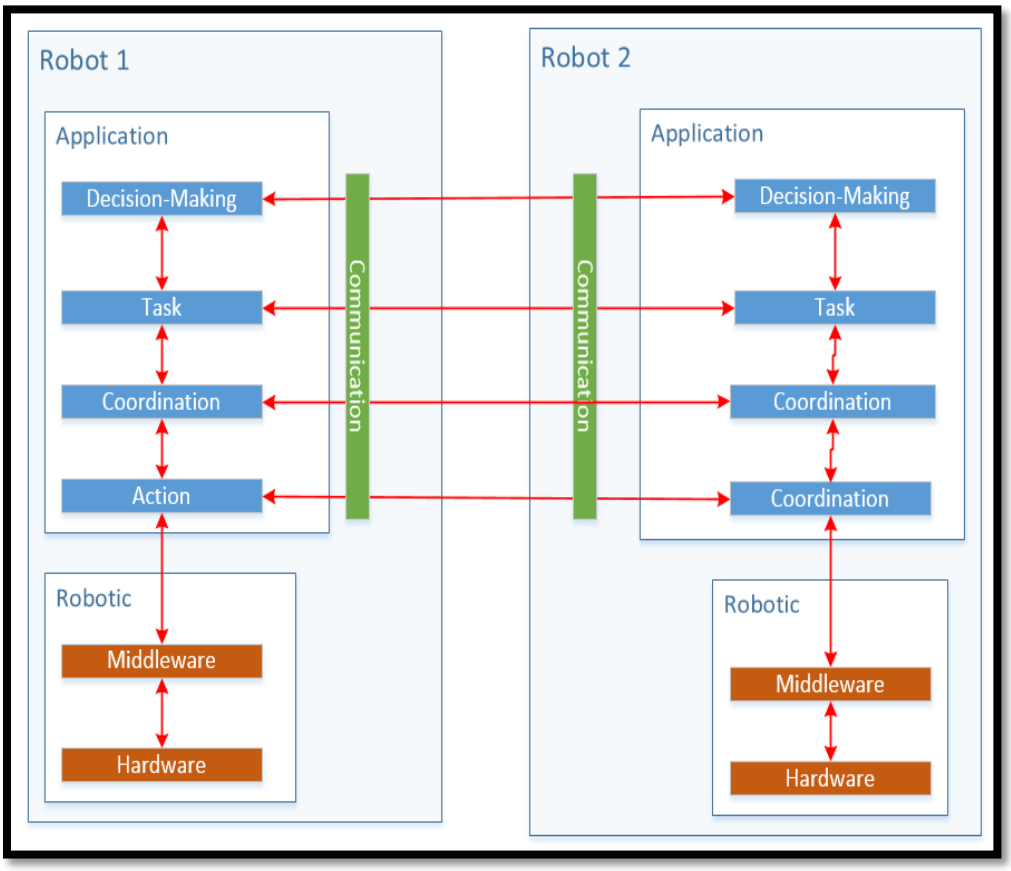

Fig. 19. Two Robot Object Moving Layered Structure.

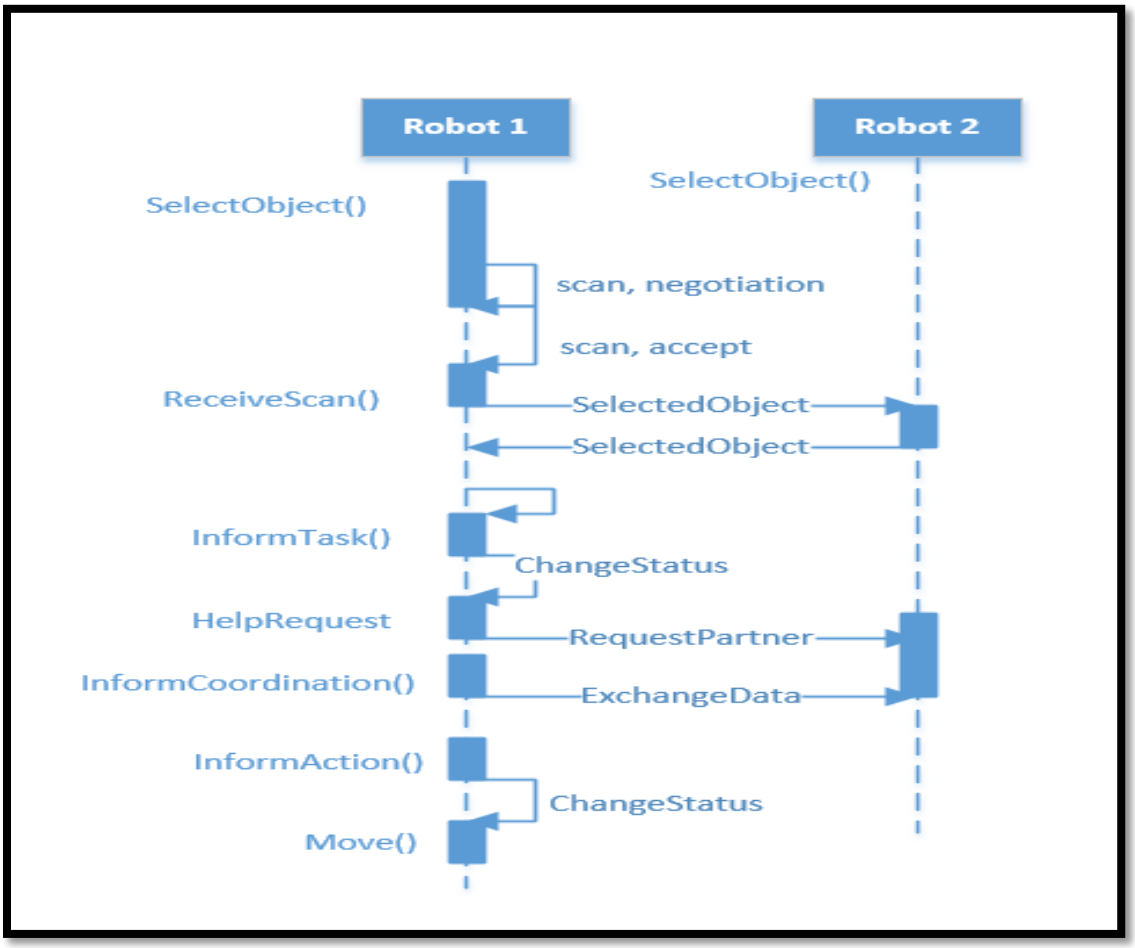

Fig. 20. Two Robot Application, Agent Interaction.

By implementing the methods InnerProcess, HandleMsg, and StoreIntraMsg the user will have defined a layer's internal processing, state changes, and methods for accessing information. Within these functions, the potential reactions and conditions of layer activation/suspension will be defined. Fig. 21 illustrates the logic within the Decision-Making layer's InnerProcess().

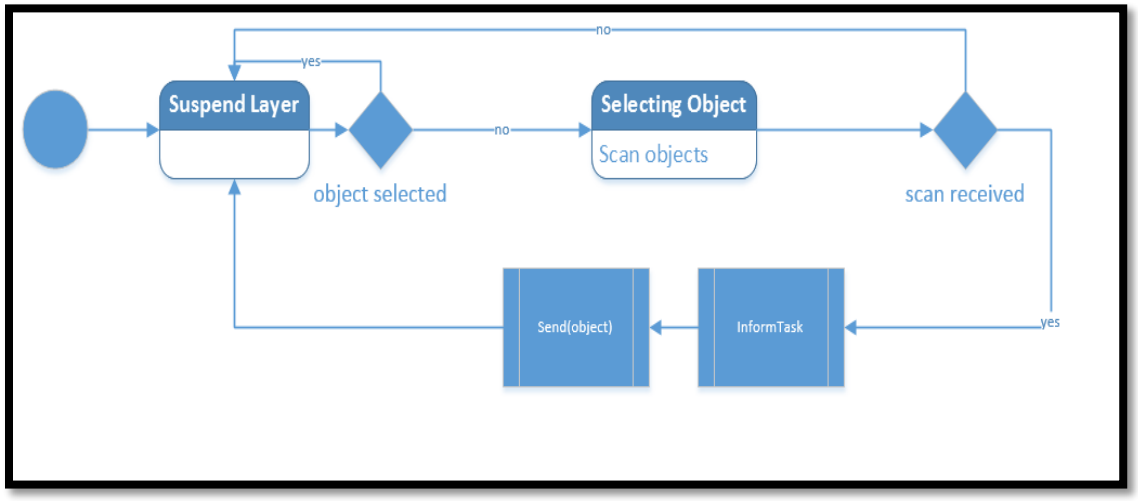

Fig. 21. Decision-Making Layer's State Machine Diagram.

The layer is initially suspended until an incoming message activates it and checks whether an object has been selected. If that is not the case, the layer enters the "Selecting Object" state which involves sending a scan request to the Action Layer and suspending execution. Upon receiving a scan message, the layer informs the Task Layer and sends the object information to Robot 2. A similar pattern is repeated for all layers which are activated by events. Activation events vary from scan messages to inform and ChangeStatus messages. Upon activation, certain attributes are checked and depending on their status, or the attributes of incoming messages, certain reactions are triggered either in the form of sending external messages or proceeding with execution of user-defined functions such as Move and Scan.

The process of creating these layers and injecting the proposed behavior in their virtual methods does not seem complicated. The four layer structure was one interpretation of the system. It would be possible for the Task and Coordination to be integrated, simply by combining their state transitions in HandleMsg and InnerProcess. The coordination that was required for each layer is also simplistic; it involves checking an attribute and depending on its

value, the layer would remain idle or it would send a message. The framework's structure does not hinder or disallow any of the proposed model's functionality; messages can reach their destination whether internal or external and the concurrent layer execution's subtleness is evident by its absence in the figures. The application example that was introduced posed no problems for the CAS Framework. Of course, the example was rather simplistic. In the next chapter, a more complicated application will be introduced and allow a more proper and detailed evaluation of the framework's capacity to facilitate an application's development process.

CHAPTER 5

EXAMPLE APPLICATION DEVELOPMENT

This section will examine the development of an autonomous collaborative system on the CAS Framework. The process of mapping the system to the layered architecture will be discussed to demonstrate the ease and flexibility of implementing systems on the framework. Furthermore, the capabilities of the framework and the burden that is avoided by utilizing them will become evident from the description of the system's detailed algorithmic aspects. By providing results and evaluating the implementation process, the benefits and effectiveness of using the framework will be showcased. Analysis of the results will reveal if meaningful collaborative behavior was observed among the autonomous agents and if such patterns aided the system. Finally, a comparison of the user-implemented content and the already-defined framework functionality will illustrate the degree of support that was provided by the CAS framework.

5.1. Problem Description

The example system used for development is a fire brigade that consists of autonomous robots. The system is fairly simple in the complexity on individual robots but is sufficient to illustrate the layered framework. In the example, a set of robots is tasked with transporting buckets from a set of sources to a set of pre-defined destinations. There are three types of robots: Generators, Sinks, and Transporters (Fig. 22). Generators are immobile robots whose purpose is to generate buckets filled with water and pass them on to a Transporter for transport to a sink. Sinks are also immobile robots whose sole purpose is to receive buckets and dispose of them.

Transporters are mobile robots that transport the buckets between Generators and Sinks. Several assumptions are made:

- Buckets move solely from source to sink. Empty buckets are not passed back to the source.
- Transporters have a fatigue property when carrying a full bucket. Thus, when carrying a bucket they will slowly decelerate; without the possession of a bucket, Transporters can maintain full speed. The deceleration will stop taking effect when a minimum velocity has been reached. At this point, the robot will continue to move with this minimum velocity.

The last assumption encourages collaboration in order to keep the velocity a bucket is moved as high as possible by passing the bucket off to a fresh Transporter.

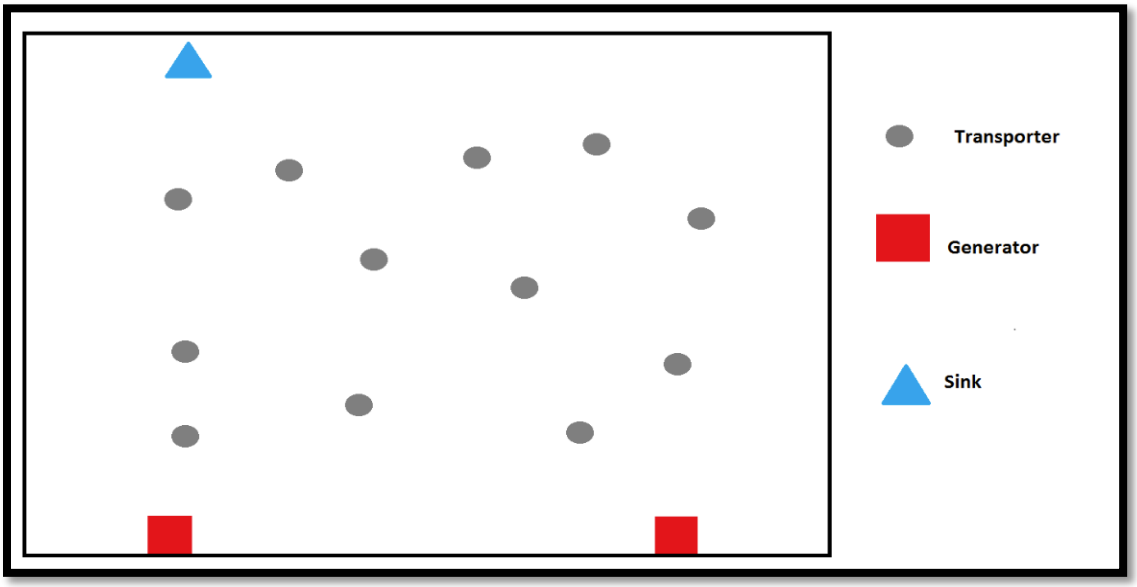

Fig. 22. Fire Brigade System.

The purpose of the system is to transport buckets efficiently from Generator to Sink. By utilizing the CAS framework, the user can examine if a particular collaborative approach will aid in the bucket transportation process by increasing the bucket throughput or/and decreasing the average bucket travel time.

5.2. Approach

Collaboration in this system is hypothesized to be advantageous due to the deceleration aspect of carrying a bucket. This encourages Transporters to pass the bucket to another Transporter in order to have the bucket maintain a high velocity. Taking that into consideration, the proposed transportation process will consist of a series of bucket exchange between which Transporters move buckets until the bucket reaches its destination. A bucket exchange, from its formulation to its execution, involves several steps (Fig. 23) that require collaboration between the two involved robots as well as internal processing.

During this process, the robot will reach several states (Fig. 24), Idle, Finding Partner, Meeting, and Coordinate Meeting. The initial state is determined by the robot's type. Transporters and Sinks are in the Idle state until they are involved in a partnership. Generators have a bucket in their possession by default and will immediately enter the Finding Partner state.

- If robot has a bucket and no partner
  - Determine Destination (obtain coordinate positions and types of all robots)
  - Obtain Coordinates of all robots within range and select a Partner
  - Agree on meeting point with partner
  - Move to meeting point
  - Pass the bucket
- If a robot does not have a bucket but has a partner
  - Agree on meeting point with partner
  - Move to meeting point
  - Accept the bucket
- If a robot does not have a bucket or a partner
  - Wait for partnership request

Fig. 23. Bucket Transportation Algorithm.

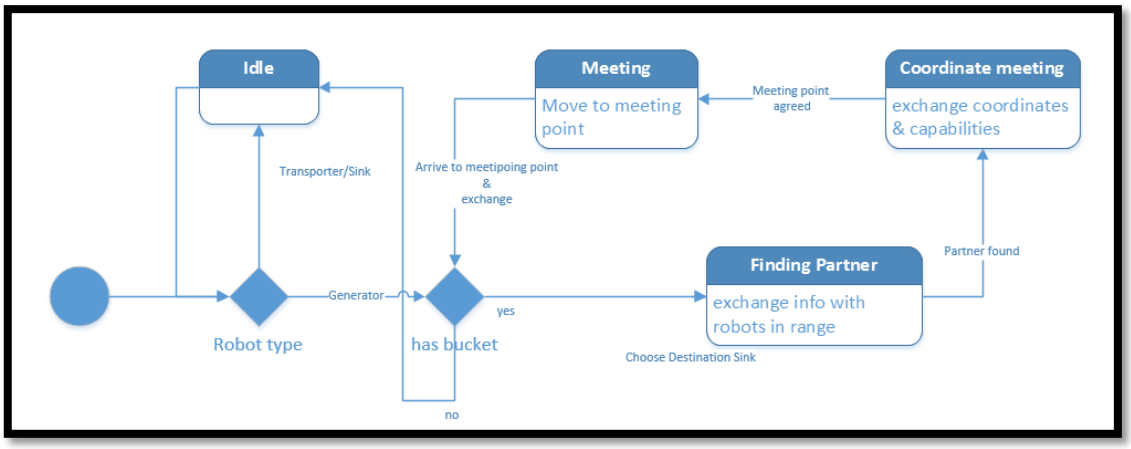

Fig. 24. Fire Brigade Agent State Machine Diagram.

A Transporter enters the same state when it has a bucket. When in this state, regardless of type, the robot desires the partner resulting in the fastest movement of the bucket to the sink. A

large number of robots in the system leads to numerous partnership options. Thus, a robot must evaluate which bucket exchange is most advantageous. This is accomplished by calculating an estimate of the time required for the bucket to reach the Sink. This involves determining the time that is required for the two robots to meet, in addition to the time that the partner will spent transporting the bucket to the Sink on its own. Since velocity and acceleration are known values, only the coordinates of the two robots are required for this calculation to take place. First, the distance between the two robots is calculated with Equation (1). If rearranged, Equation (2) can be used to calculate time, given acceleration and velocity. Unfortunately, the distance required by the robots is not yet known.

$$distance = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \quad (1)$$

$$distance = velocity \cdot time + \frac{1}{2}\, acceleration \cdot time^2 \quad (2)$$

The total distance that will be traveled by the two robots (regarding the exchange) is equal to the distance between them. Inserting the values of each robot to Equation (2) and adding them, results in Equation (3). The only unknown variable is the time, which represents the time required for the two robots to meet. Through the quadratic formula, the time can be determined.

$$2 \cdot velocity \cdot time + \frac{1}{2}\, acceleration \cdot time^2 - \text{ distance} = 0 \quad (3)$$

The next step is to calculate the time required by the potential partner to carry the bucket towards the Sink, after it has come to its possession. Equation (2) is once more, but since no other robot is involved, the equation can be immediately rearranged to Equation (4). Solving for time, using the quadratic formula will yield the solution.

$$velocity \cdot time + \frac{1}{2}\, acceleration \cdot time^2 - \text{ distance} = 0. \quad (4)$$

As mentioned in the Problem Description, the robot can enter a minimum velocity mode when traveling with a bucket for an extended period of time. This can cause problems in the

calculations. Thus, the results must be evaluated in order to ensure whether the minimum velocity mode was activated or not. If the mode was activated, the time until its activation is calculated by Equation (5)

$$time = (\frac{velocity - velocity_0}{acceleration}) \quad (5)$$

Then, the distance traveled during this time is calculated for both robots separately by Equation (4). The remainder of the distance between them, will be traveled in constant velocity by both robots. Once more, by taking the sum of the two instances of Equation (3) the time until the two robots meet is calculated and added to the time required for the minimum velocity mode to be activated.

This process is repeated for each robot within a specified. When all results have been calculated, a partnership preference list is created, by sorting the partnerships based on the time metric. Then, the robot will send a partnership a request to the first robot in the list. In case of a rejection, the robot will send a request to the next best partner if one exists. If no partner is found, the robot will transport the bucket without collaboration, unless it is a Generator in which case a partner has to be found due to the lack of mobility. Partnerships are non-interruptible and will last until the bucket is exchanged.

The Finding Partner State is exited as soon as a partnership is formed. Then, the two robots enter the Coordinate Meeting state. In this state, the robots exchange coordinate and type information. Then, they calculate the point of exchange and ensure that it is reachable by both robots. With the two robots in agreement about all aspects of the exchange, they enter the Meeting state. In this state, the two robots move toward the meeting point and informing their partner when they have arrived. When both partners are ready, they will exchange the bucket.

As soon as the exchange takes place, if the receiving robot is an entity it enters the Finding Partner state, while Sinks enter the Idle state. For Generators, losing possession of the bucket, requires the creation of a new one which will initiate the entire process again and lead the robot to the Finding Partner state. Loss of bucket possession for Entities leads to the Idle state, in which they wait for partnership request.

There are certain limitations to this approach. Partnerships are not re-evaluated after they are formed. This means that better qualified partners that become available after the partner selection, cannot be involved. Such robots, will remain idle (unless called into a partnership) and will not seek to improve their position by moving to a place where partnership requests are more likely to be received. Since the purpose of the example is not to provide an optimal solution to the fire brigade problem but instead to demonstrate the capabilities and benefits of using the framework, these limitations will not be addressed further.

5.3. Layer Structure

As mentioned in Section 1.3, the layered structure of the Framework allows the abstraction of a collaborative/decision-making process into independent layers. The internal communication is limited to neighboring layers, implicitly organizing the layers on a hierarchy which can be utilized by enforcing layer precedence and requirements for operations. The steps of the main algorithm (Fig. 25.) have a clear order of execution and require the completion of previous steps. While the user is free to apply the algorithm on the layered structure without many restrictions, it is intuitive to create an application layer for each step of the algorithm or combine some steps to a single layer (Table IV). In this case, the processes find destination, find partner, and agree on meeting point, can be implemented in separate layers. Moving to a meeting point and passing the bucket resemble actions and can thus be jointly implemented in a layer

which can coordinate them. Finally, a robotic layer is required for the execution of actions. These layers are named: Goal, Strategy, Group Activity, Task, and Action.

TABLE IV

LAYERED MODEL

Layer	**Purpose**	**Function**
Goal	Find Destination	Request types and Coordinates of all Robots
Strategy	Find Partner	Get robots in range and send partnership request
Group Activity	Agree on Meeting Point	Exchange coordinates and capability information
Task	Meet & Exchange Bucket	Coordinate actions: move to meeting point & bucket exchange
Action	Execute	Move, rotate, exchange bucket

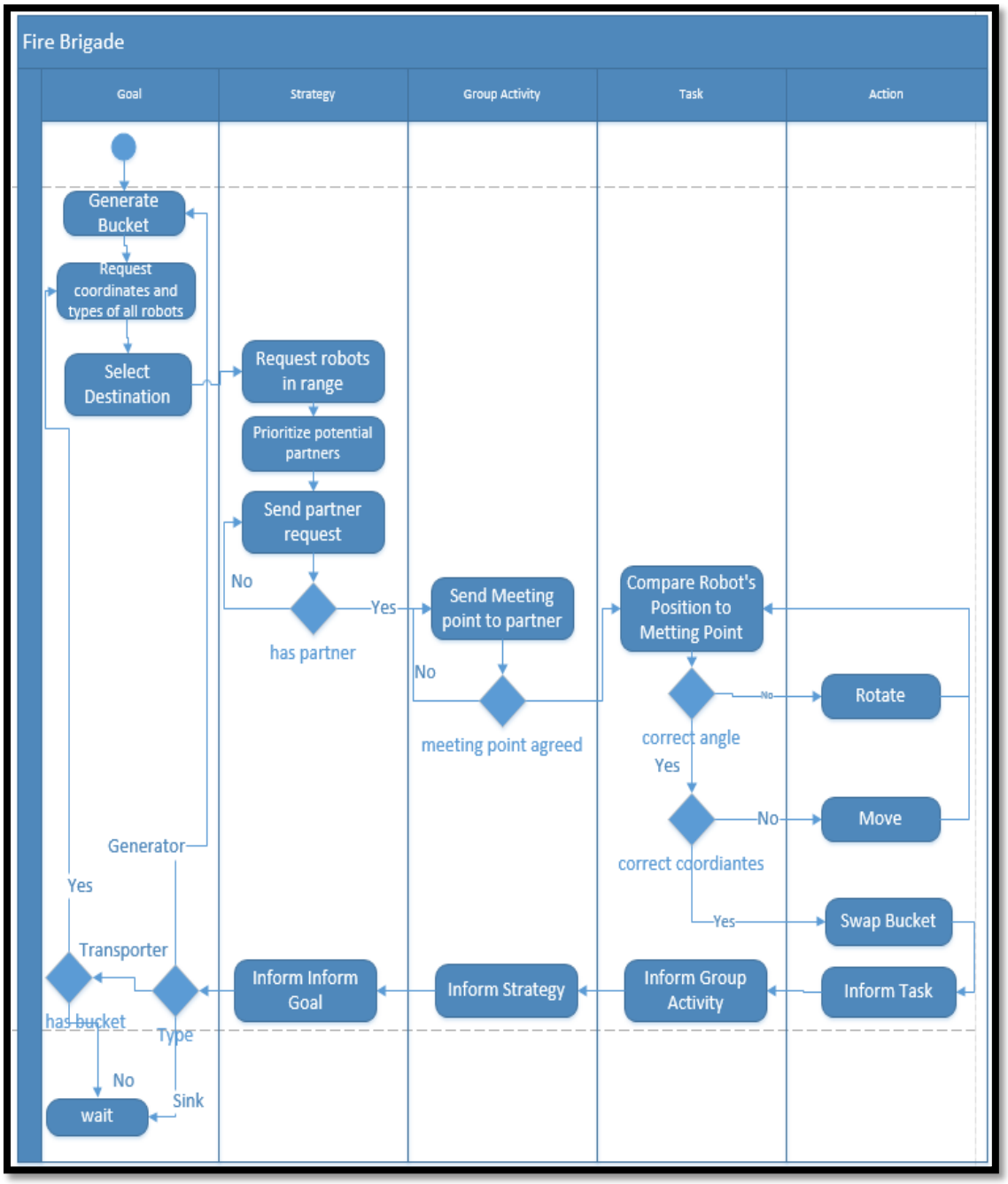

Fig. 25. Layer Activation Activity Diagram.

A pattern that is observed is that due to the order of execution of the algorithm that led to this layered structure, a layer is by definition activated by another layer higher in the hierarchy. In addition, a layer can be activated when it receives an external message from one of its counterparts on another robot. The layer activation process, which is illustrated in Fig. 25 is initiated by the Goal layer which performs a sequence of action and waits for the layers below accomplish a mission. The activities of each layer, which are illustrated in Fig. 25. are necessary for achieving the layer's purpose. The Goal layer is responsible for determining the Goal of the robot, which are either move a bucket or stay idle. The strategy layer includes all the functionality of the Finding Partner state; it performs the necessary partner evaluation and collaborates toward forming a partnership. Group Activity allows the robots to agree on a meeting point by exchanging information. The Task layer is responsible for organizing the execution of physical actions. It compares the robot's current state with the intended state and determines if the robot has to move, rotate, or wait for its partner. All the layers higher in the hierarchy, collaborate by exchanging information and then reacting. The Task layer differs in this aspect, since it evaluates all executed actions and upon moving to the meeting point, it collaborates and only when both robots are at the same point, will a command for a bucket exchange be given. All these decisions that the Task layer is responsible for are given to the Action layer for execution. This layer, which belongs in the robotic category, is evaluated after every execution by the Task layer. Upon the final evaluation, which consists of the bucket exchange, all layers will be notified.

5.4. Structure Modifications

One of the benefits of the framework is that due to its compact form and with the use of inheritance and association relationships, expansions can be made by the developer without

manipulating the interfaces. In certain systems some expansions might facilitate the development process or even be necessary. Due to the variety of systems that are intended to be compatible with the CAS Framework, such expansions should be supported by system. This subsection will demonstrate the roles of some additional non-framework components, which are illustrated in Fig. 26, and will describe the relationships that were defined in order for the expansion to take place.

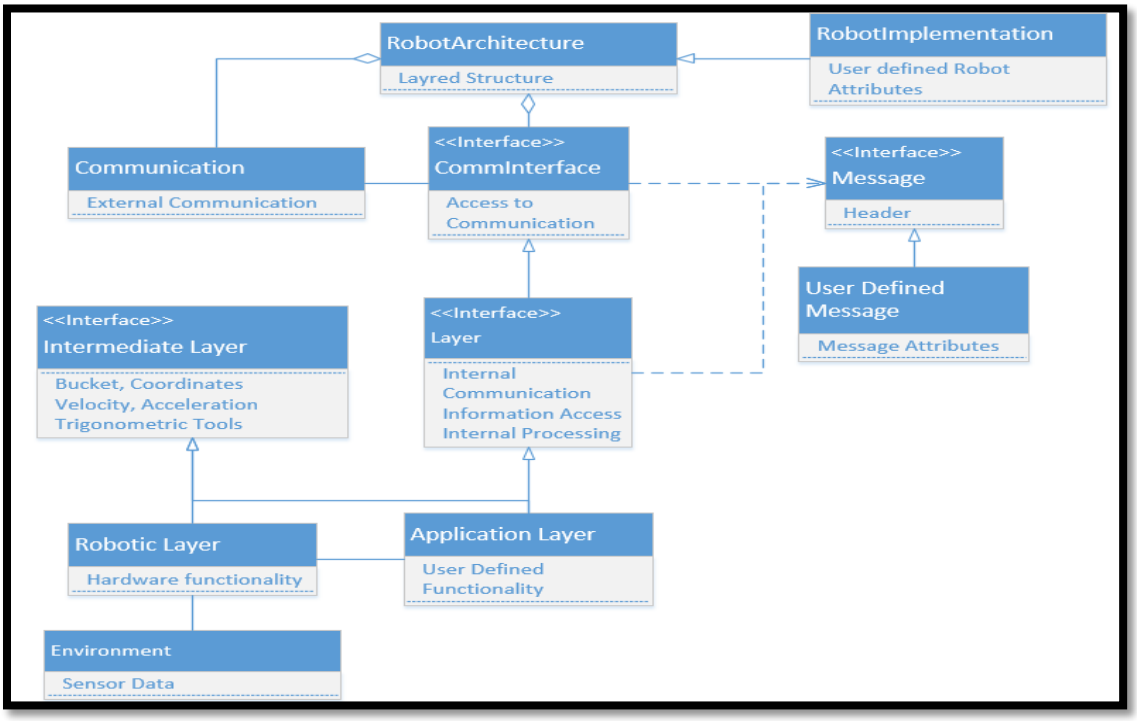

Fig. 26. Expanded Architecture of Fire Brigade Application.

In the Fire Brigade application, layers contain duplicate information (bucket possession, coordinates, etc.) that must be kept synchronized and up to date. Note that this duplication can be easily avoided but was included in order to demonstrate that such coordination is easy to be performed. In that case, it seems logical to want those common attributes and functions to be

passed down to all layers through inheritance, instead of redefining them in each layer. This distribution could take place by inserting these attributes on the Layer interface. Unfortunately, the manipulation of any interface can potentially pose problems and should be avoided. The solution was the insertion of an intermediate user-defined layer, which acts as another interface to the layers. This intermediate layer is simply another parent class to all user-defined layers, the Layer interface being the other, and is not associated with any framework defined components, making its inclusion risk free. It is worth noting, that an inheritance relation is also established between the intermediate layer and the Robotic layers (Action). The Action Layer does not contain behavioral algorithms and all functions and attributes required by the Simulation mode have already been inherited by the Layer interface. Regardless, access to the common attributes of the Intermediate Layer is useful since it allows the insertion of safety checks, such as checking bucket possession before performing bucket exchanges. Thus, establishing inheritance relations between the layers and the intermediate layer, will make the duplicate attributes accessible by all layers. It is important to note, that these attributes are duplicated and not shared; each layer has its own copy. This can be avoided by declaring the attributes as static; for demonstration purposes, they remain non-static. In the Fire Brigade application, this intermediate layer provided attributes such as Cartesian coordinates and a Boolean that represents bucket possession. Trigonometric methods that determine angles and distances between points were also added since they were heavily used by multiple layers. Though the inclusion of the intermediate layer was not necessary, it allowed the avoidance of mundane and non-user friendly re-definitions.

Another addition was the Environment class which is accessible only by the Robotic Layer. The Environment contains information such as robot coordinates and sensor data; these

data are made available to the Robotic Layer through an association relation that was defined. References to the environment are not included in any framework-defined modules, making its inclusion safe. This class is necessary due to the lack of robotic hardware. The Environment can simulate sensor readings and hold the locations of all simulated robots, making it vital in virtual systems. If the application was attached on a physical robot with access to functioning sensors, the Environment class would not be needed (Fig. 27). For simulation purposes, placement of the Environment on a user-defined robotic layer would allow the developer to use the framework's internal communication capabilities for accessing sensor data. This would enforce the structure of a layer on the Environment, requiring internal processing functions and receive functions to be defined. Furthermore, the response to sensor queries would not be immediate since the Environment would have to access such messages and reply to each one individually. An alternative approach is to not include the Environment as a robotic layer, but as a simple class that is accessible from a robotic layer. While this would deny internal layer communication functionality, it would allow direct access to sensor data and will not require the definition of any framework related functions.

Further manipulation in the architecture can potentially take place. More associations, dependencies, and inheritances could link additional components. Such an example is an intermediate message, which could provide a basic structure to all messages in the system. The capacity to perform such expansions without risk, is awarded to the proper interfacing of the CAS Framework components, and provides significant flexibility the user, whose application might require very specific structures.

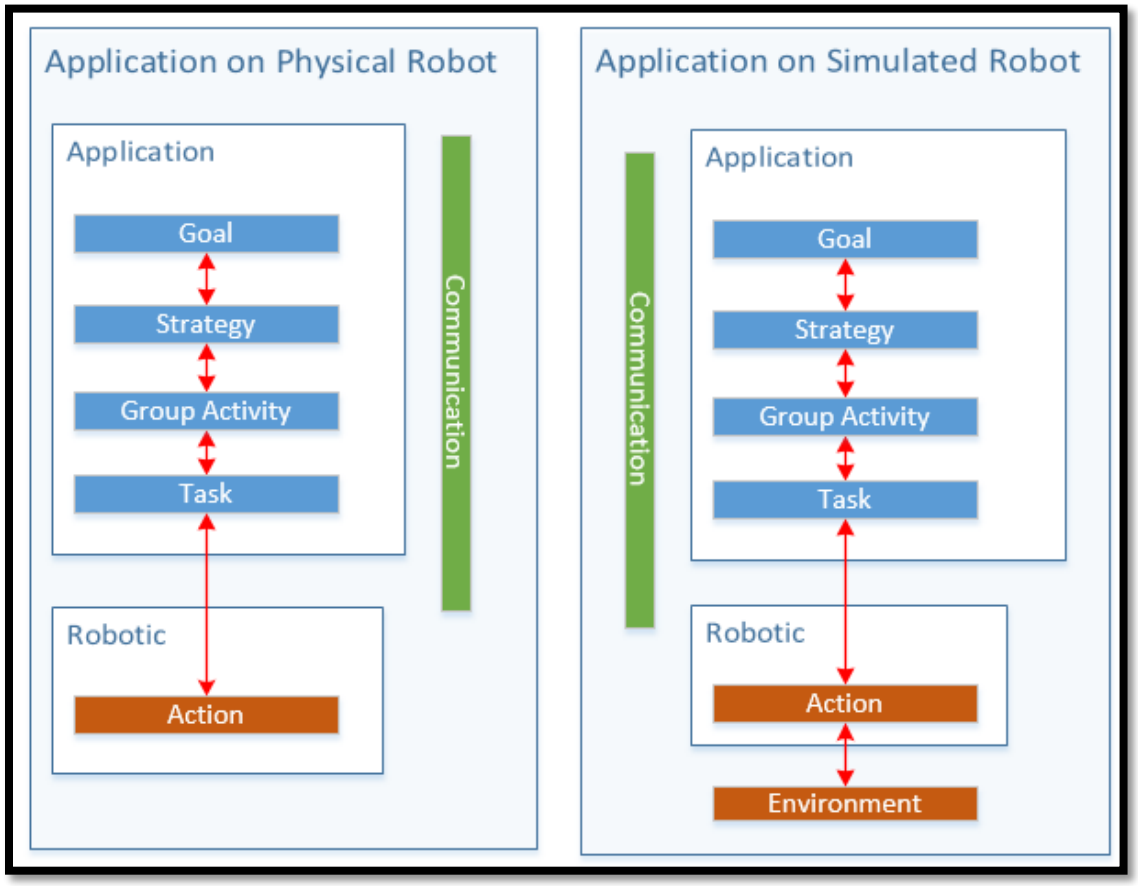

Fig. 27 Physical vs. Simulated Robots.

5.5. User-Defined Layer Structure

As mentioned in Section 3.2.4, layers run concurrently with the use of threads. The layers run until a termination condition is met; this is accomplished by iteratively executing the methods MessagesPending and InnerProcess. The latter is user-defined and in this scenario consists of three functions: CreateParallelLayerMsg, CreateUpperLayerMsg, and CreateLowerLayerMsg. These functions evaluate certain attributes that reveal the state of the layer and execute accordingly.

Fig. 28 illustrates the state machine diagram of the Goal layer. If the layer is idle and does not have a goal set, it will communicate with all robots in order to find a destination for a

bucket. When the goal is set, the “MoveBucket” state is reached which involves sending a message to the Strategy layer, signaling it to find a partner. If the lower layers function properly, the bucket will be exchanged at some point in the future. When this exchange takes place, the possession of a bucket will be evaluated. If the bucket is no longer in the robot’s possession, it means that it has been given to another robot. In that case, if the robot is a generator it will generate another bucket. Otherwise, it will remain idle until a new goal is set through a partnership formation. If after an exchange, the robot is in the possession of a bucket, it will remain in the “MoveBucket” and will inform the layer below that a new partner must be found.

The state machine diagram of the Strategy layer is illustrated in Fig. 29. When Strategy receives a goal (Sink destination) from the Goal layer, it initiates the partner selection process. This requires knowledge of the robots in range, possession of a bucket, partnership status, and a complete evaluation of all possible partners. Until these requirements are satisfied, the Strategy layer will alternate between the Evaluating Partners, Get Robots in Range, and Requesting Partner states. When a partnership has been formed, a message will be sent to the Group Activity layer. The layer will remain in this state until the Group Activity layer notifies that the bucket has been exchanged; this will lead to the notification being forwarded to the Goal Layer and switching to an idle state.

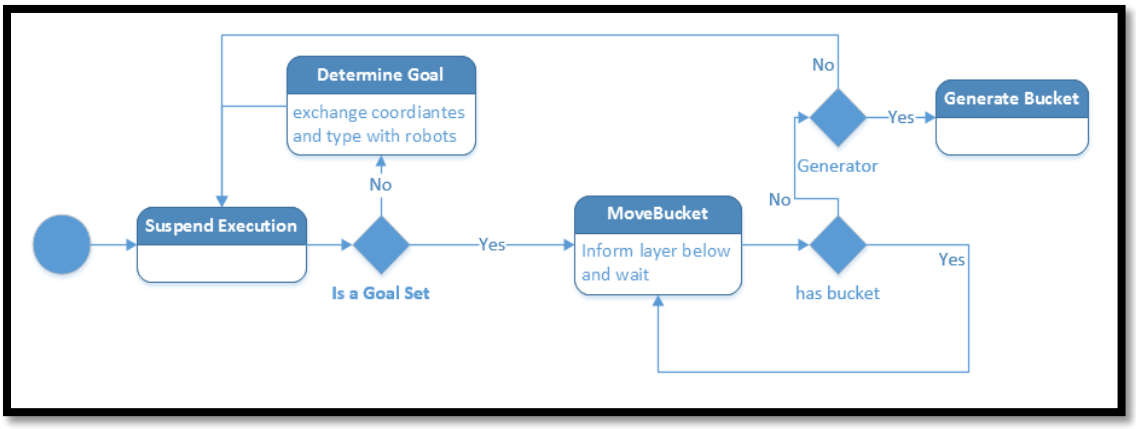

Fig. 28. Goal Layer State Machine Diagram.

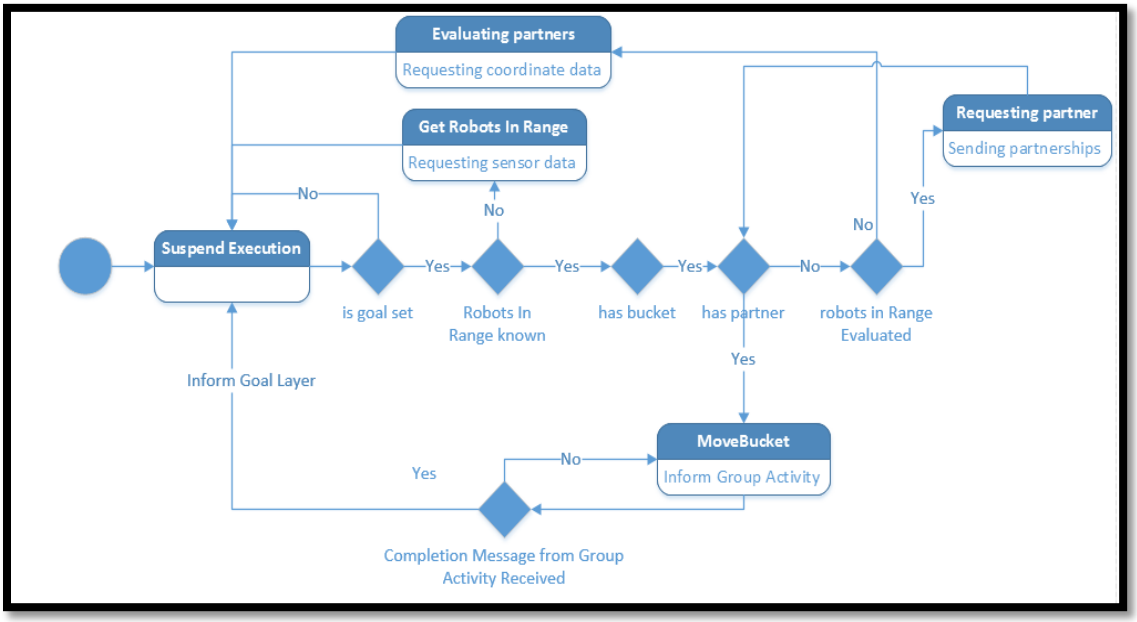

Fig. 29. Strategy Layer State Machine Diagram.

The state machine diagram of the Group Activity layer is illustrated in Fig. 30. The Group Activity layer is activated upon receiving a message from the Strategy Layer. On message receipt, the layer switches to the Meet state, in which a meeting point must be agreed upon with

the partner. Thus, a message with the meeting coordinates is sent and upon receipt of an acceptance the layer enters the Meeting state and notifies the Task layer about where to move in order for the bucket exchange to occur. When the Task layer notifies regarding the completion of a bucket exchange, the notification will be forwarded to the Strategy Layer.

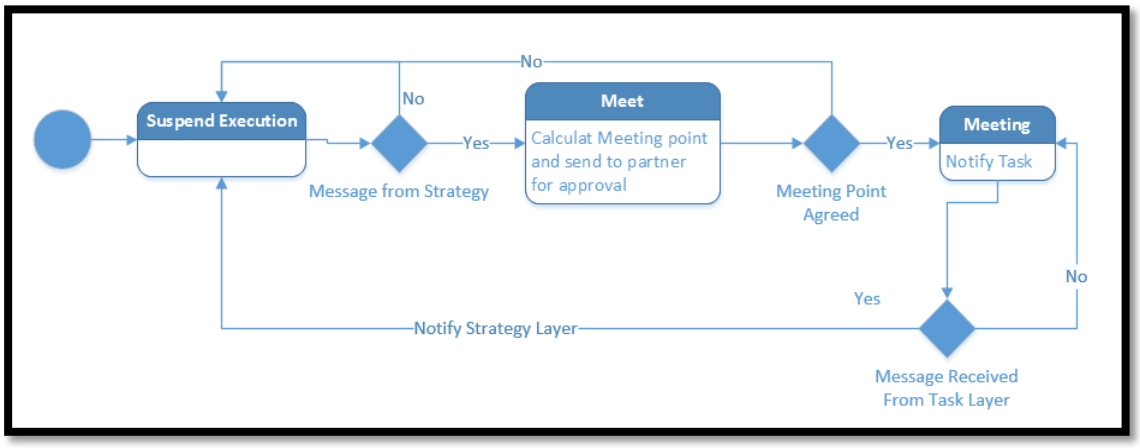

Fig. 30. Group Activity Layer State Machine Diagram.

The state machine diagram of the Task layer is illustrated in Fig. 31. Upon receipt of a message from Group Activity, the Task layer determines whether to enter the Rotate or Move state based on its coordinates and orientation. In both states, a lower layer message will be sent to the Action Layer. As soon as a notification is received by the Action Layer, that the requested action has been completed, the updated coordinates and orientation values will be compared to their desired values. Depending on the results of this evaluation, the Task layer will request a move or rotate from the Action layer, or it will enter the "SwapBucket" state, in which the signal for a bucket exchange will be send to the Action Layer.

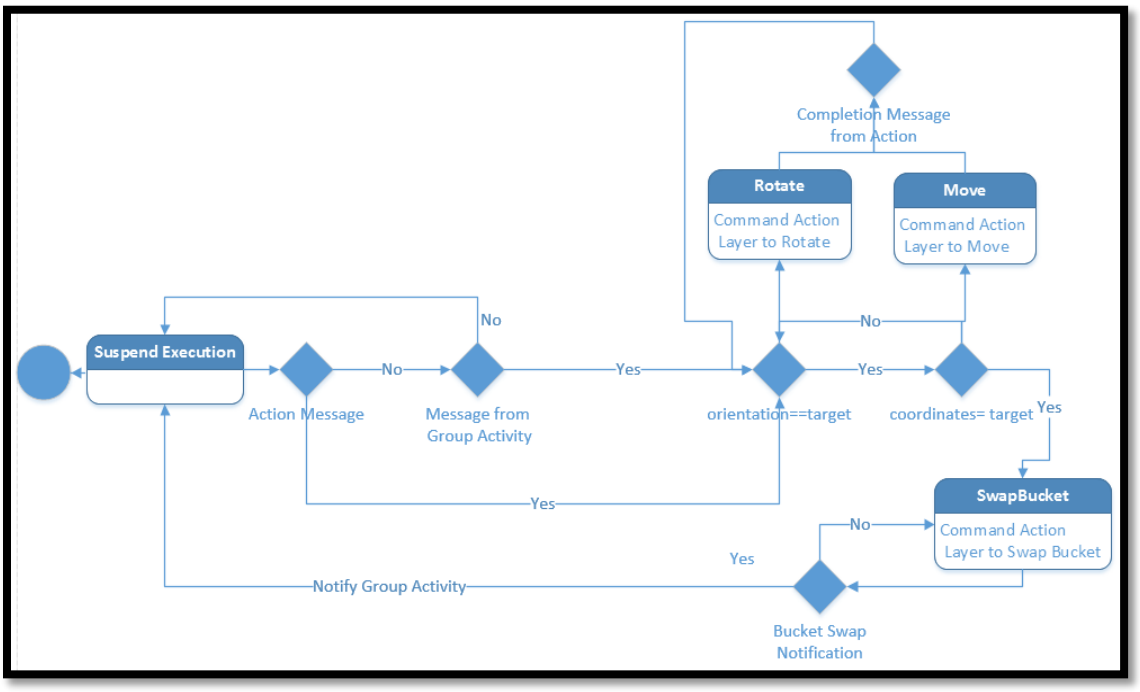

Fig. 31. Task Layer State Machine Diagram.

The state machine diagram of the Action layer is illustrated in Fig. 32. The Action Layer simply executes the command that is given by the Task Layer and notifies it for evaluation. If the evaluation is successful, the next action will be calculated or in the case of a bucket swap, all above layers will no notified. Otherwise, the same action will be repeated per instruction of the Task layer.

The state machine diagram of the Action layer is illustrated in Fig. 32. The Action Layer simply executes the command that is given by the Task Layer and notifies it for evaluation. If the evaluation is successful, the next action will be calculated or in the case of a bucket swap, all

above layers will no notified. Otherwise, the same action will be repeated per instruction of the Task layer.

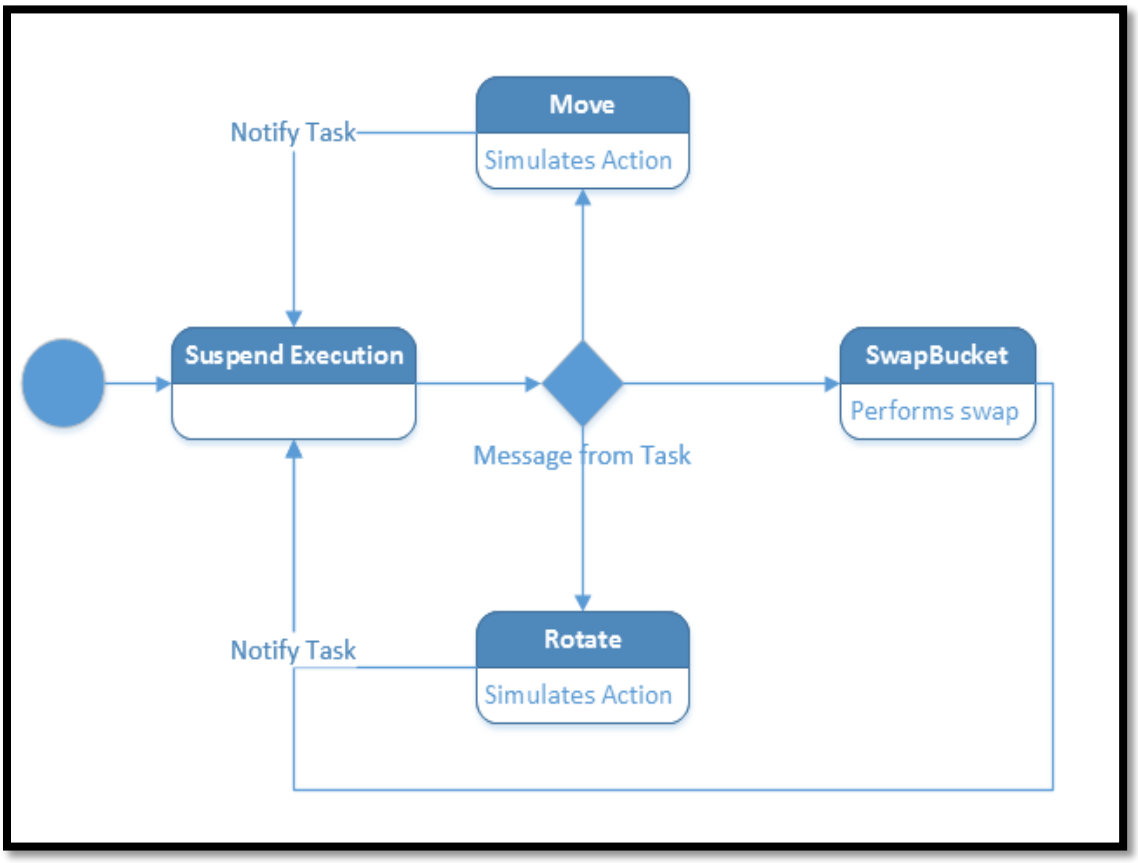

Fig. 32. Action Layer State Machine Diagram.

5.6. Layer Communication

In the Fire Brigade application, eight message types have been created based on the functionality of each layer their communication requirements. Table IV lists all messages with their function and layer associations. Their functionality will be discussed in order of execution throughout this subsection.

The RequestInfo message, which contains Cartesian coordinates, robot type (Generator, Sink, Entity), and communication type (negotiation, accept, deny), allows a robot to determine the closest sink and select the destination of a bucket.

TABLE III

FIRE BRIGADE APPLICATION MESSAGES

Message	Layers Involved	Function
RequestInfo	Goal-Goal	Transfer coordinate and type information for destination selection
RequestRobotsInRange	Strategy-Action	Find the robots inside a specific range
RequestPartner	Strategy-Strategy	Request partnership
Meet	Group Activity-Group Activity	Agree on meeting point
RequestExchange	Task-Task	Inform of arrival at meeting point
ChangeStatus	Available to all layers	Inform of a change that will require neighbor layer to change state
CompletionStatus	Action-Task	Inform of completed task and terminating parameters
RequestCoordinates	All Layers(except Action)- Action	Extracts coordinates from grid and returns them

When a destination has been determined, the Goal layer forwards this information to the strategy layer through the ChangeStatus message. This message is designed for notifying neighboring layers of status changes which lead to new responsibilities (Goal-Strategy) or notify regarding the completion of these responsibilities (Strategy-Goal). The attributes are:

- a string where layers can store their current responsibility,
- an integer and two doubles that allow the transfer of coordinates,

- a Boolean, that can express the presence or absence of a bucket, and
- a communication type which specifies if a message is a query or not. The communication type can also specify acceptance or rejection of requests.

In the Fire Brigade application, the Goal layer populates this message by assigning the partnership criteria “time” to the string, indicating that the shortest travel time to a Sink is what governs partner selection. The coordinates of the Sink destination are assigned to the two doubles, and the possession of a bucket to the Boolean. The integer is unpopulated because no further information is required. Depending on the layer that uses this message, different attributes will be utilized. It is possible to create a respective ChangeStatus(internal) message class for each layer, thus avoiding unutilized attributes. In the Fire Brigade example, a single ChangeStatus message was created in order to keep the design compact and to refrain from defining messages classes that are almost identical

The purpose of the strategy layer is to send partnership requests to all robots that are in range, or to reply to such requests. The RequestRobotsInRange(internal) message is an intra-message that is sent to the Action layer from the Strategy Layer, since knowledge of the robots in range is required for partnership requests to be made.. The action layer through its access to the environment, can determine a list of all the robots in range and sends their IDs to the Strategy Layer. This message contains an integer that is assigned the destination layer of the message, an integer pointer which we use to pass a dynamic array of IDs that represent the robots in range, an integer to represent the size of the list that contains the robot IDs, and a communication type which allows the Action layer to recognize that it must sent a reply when it is populated with the Negotiation type.

Then, the Strategy layer can send a RequestPartner(external) message to another robot. This message will contain the coordinates of the robot, the coordinates of the destination, the robot type, and the communication type, and the robot ID. The same message is used to reply and will contain the communication type (Accept or Deny) in addition the coordinates and ID of the replying robot.

Then, ChangeStatus is used to notify the Group Activity layer of the changes. The message will include the partner ID, destination coordinates, the presence of a bucket, and the change in the strategy's responsibility which will be "MoveBucket".

The Group Activity will send a Meet (external) message to its partner in order to initiate the bucket exchange coordination. This message will include coordinates, robot type, and communication type. Upon receiving such a message, the partner will evaluate the coordinates and mobility capabilities of the two robots, and reply by specifying the meeting point. Once the meeting point has been agreed upon, a ChangeStatus message will be sent to the Task layer, containing the new responsibility ("Meeting"), the partner ID, and the presence of a bucket.

With the meeting point established, the Task layer must coordinate the bucket exchange and the moving process. Alternating between the move and rotate actions is a result of inner processing whose result is forwarded via the ChangeStatus. When the Action layer has notified that it has moved to the meeting point, the robot must wait for its partner to do the same. In order to avoid unnecessary message exchanges, a Meeting message is sent, which will contain the Negotiation communication type. The partner will reply by using the same message with a communication type of Accept, only when it reaches the meeting point. Then, the Task layers of the two robots can send a new ChangeStatus message to the Action layer, requesting to swap the bucket.

When the exchange has occurred, a CompletionStatus(internal) message will be sent to the Task layer which will in turn send a ChangeStatus message to the Group Activity layer. With each ChangeStatus message that includes the completion of the layer's responsibility, the above layer will be forwarded the same message until it reaches the Goal layer. These instances of ChangeStatus will simply populate the presence of a bucket and the new responsibility. When the Goal layer has been notified of an exchange, the same process will be repeated.

The messages utilized in the Fire Brigade application are very similar in structure and are utilized in a similar manner. The handling of external/internal messages and intra-messages is identical, since the framework handled the distribution. A specific receive function was defined for each message and it called within the HandleMsg, StoreIntraMsg, and GetIntraMsgReply, according to the message type; utilization of these functions required interfacing to the Message class. Regardless of the message type or its content, the definition and utilization process was facilitated by the CAS framework which provided both the means for user-defined message creation and manipulation.

5.7. Alternative Strategy Layer

The layered structure, which keeps each level of the application separate, allows for easy manipulation of layers without jeopardizing the functionality of the surrounding system. Since layer manipulation/replacement is one of key reasons behind the adoption of the layered structure, the effects of manipulating a layer and the effort that is required by the user must be evaluated. Such a change was implemented for the partner selection process which occurs at the Strategy layer. The original version involved the evaluation of all robots in range in terms of the time it would take for the bucket to reach its destination. Swapping this approach with an alternative where a random robot within the range would be chosen is quite simple.

The Strategy layer's function CreateParallelMessage is responsible for sending partner requests. In order to change this approach, this is the only function that requires modification. While this function originally involved a coordinated series of queries regarding robots in range and partner evaluation related data, the new version only requires the generation of a random integer. This random number will determine which in-range robot will be sent a partner request.

Additionally, the ReceiveRequestInfo method which prioritized the potential partnerships, can be removed since it is required by the random approach. Thus, by inputting three lines of code in the CreateParallelLayerMsg function, the entire system's behavior is changed. Without the layered approach, the partner selection methods would be interconnected and less isolated, making their manipulation complicated. The layered architecture, on the other hand, is demonstrated to be significantly versatile and compact, requiring minimal effort for the manipulation of significant functionality and without interfering with the surrounding layer's execution.

5.8. Results

The execution of the Fire Brigade system in the CAS Framework is analyzed in terms of its observed behavior and the impact of the framework on the development process. For the scenarios presented, the positions of the entities are randomly generated within a subarea and will thus vary, however, the positions of the Generators and the Sinks are constant. Generators are placed at coordinates (4, 0) and (8, 0) while the Sinks are placed at (4, 9) and (8, 9). Furthermore, in order to contain the spreading of the robots in the initial position, the random positioning of the entities will be restricted within the square, (3, 1), (9, 1), (7, 1), (3, 7). Fig. 33 illustrates the initial robot positions in a scenario random run. Note the two Generators at y = 0 and the two Sinks at y = 9.

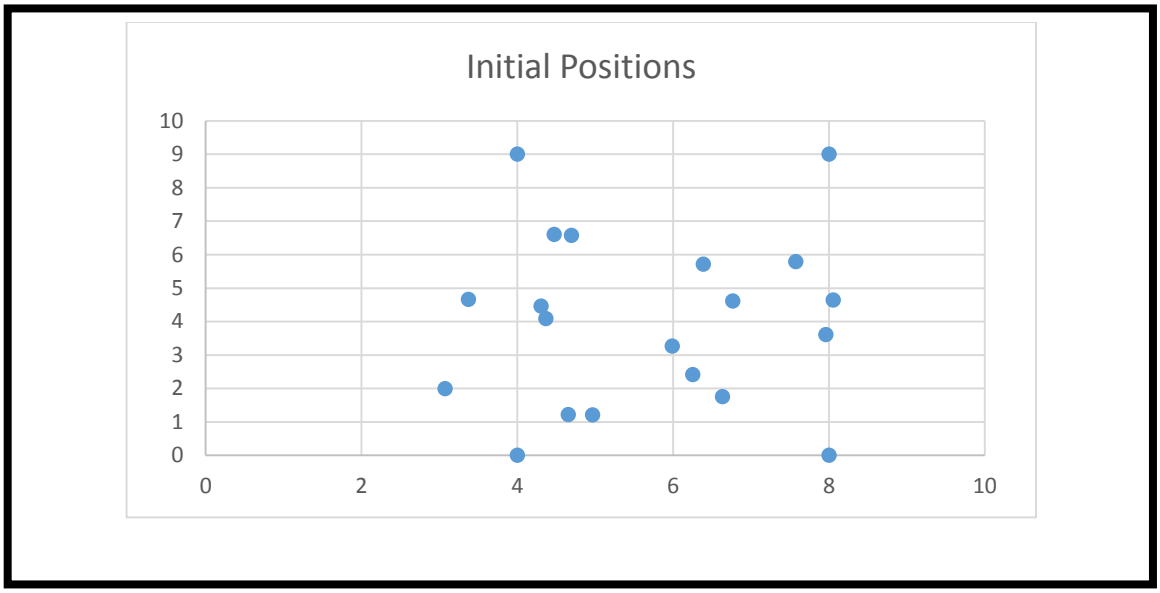

Fig. 33. Initial Robot Positions.

The hypothesis regarding the execution of the scenario, is that robots will coordinate to form lines through their local user-defined decision to exchange buckets with entities that can transport the bucket sooner. The robots that make the transportation process shorter tend to be positioned in relatively small angles in reference to the Sink. There is expected to be a constant convergence of the robots towards the path between Generator and Sink, eventually forming a line which will allow the robots to pass the buckets among them with minimal movement. The results of select scenarios will be compared against the hypothesis. Furthermore, an alternative approach will be introduced in order to demonstrate the ease of select layer manipulation. Finally, the impact of the framework in the development of this example will be discussed.

5.8.1. Scenario 1 Output

In order to test the hypothesis, a scenario was run. The executed scenario involves a set of twenty robots, randomly placed on a grid of dimensions, ten by ten. The Entities are set move at a velocity of 5 (distance units/time unit) when not carrying a bucket and decelerate at 1 (distance units/time units^2) when they do.

Finally, partnerships can be made with robots that are within a range of 5 meters. As the Generators pass on buckets and continue to generate new ones, the Entities begin to interact with each other. Throughout the processes of partner selection and bucket swapping, the robots form clear patterns. The positions of all robots were collected after each bucket exchange in the system and were used to demonstrate the effects of collaborative behavior. Fig. 34 illustrates six snapshots across the running time of the system, each showing robot positions at that moment in time. By the 35th bucket exchange in the system, all entities have been pulled towards the Generators. This is understandable for the initial steps since Generators have to pull Entities towards in order to pass the bucket; those entities have to do the same, causing an initial convergence towards the Generators. Another pattern takes place in later iteration, due to the imposed partner selection metric. Partnerships are evaluated based on the time that is required for the exchange to take place in addition to the time required by the partner to transport the bucket to the Sink. Due to this approach, robots that are closer to the Sink are be more favorable for an exchange as long as long as the time required to perform the exchange does not overshadow the rest of the transportation.

As this process continues and the robots manage to pass buckets to Sinks, they begin distancing themselves from the generators. This distance makes immediate partnerships (with the same robots) impossible due to the limited range that determines which robots are eligible for

partnership. This way, the robots that are in the lower half of the grid begin to form partnerships more often and act as buffers for the robots that are higher in the grid. By the 95th iteration, rough lines have started to form and by the 125th iteration, the entities tend not to deviate far from the line. By the 200th iteration, 18 buckets have been received by the two Sinks and the two lines have converged even further, thus establishing a clear exchange sequence. This pattern becomes useful in the bucket transportation process since it allows the robots to keep the bucket's average speed relatively high. The robots cooperate with other robots in the vicinity, while at the same time performing minimal movement. Fig. 34. illustrates how the robots 17 and 18 (black colored dots) will initially maneuver around the grid only to occupy a specific area in the line in the later iterations.

It is worth noting that even with a velocity of 5 (distance units/time unit), which is quite sufficient for a robot to efficiently move the bucket by itself, the robots still formed a chain in the process of transporting 20 buckets, which is the termination condition of this scenario. Small inconsistencies in this structure are possible to occur due to the randomness of the position and some robots might not be involved in the transportation process at all due to their initial position resulting in them not being included in any partnerships. In some rare cases, the chain can be broken due to robots going out of range, caused by uneven initial positions.

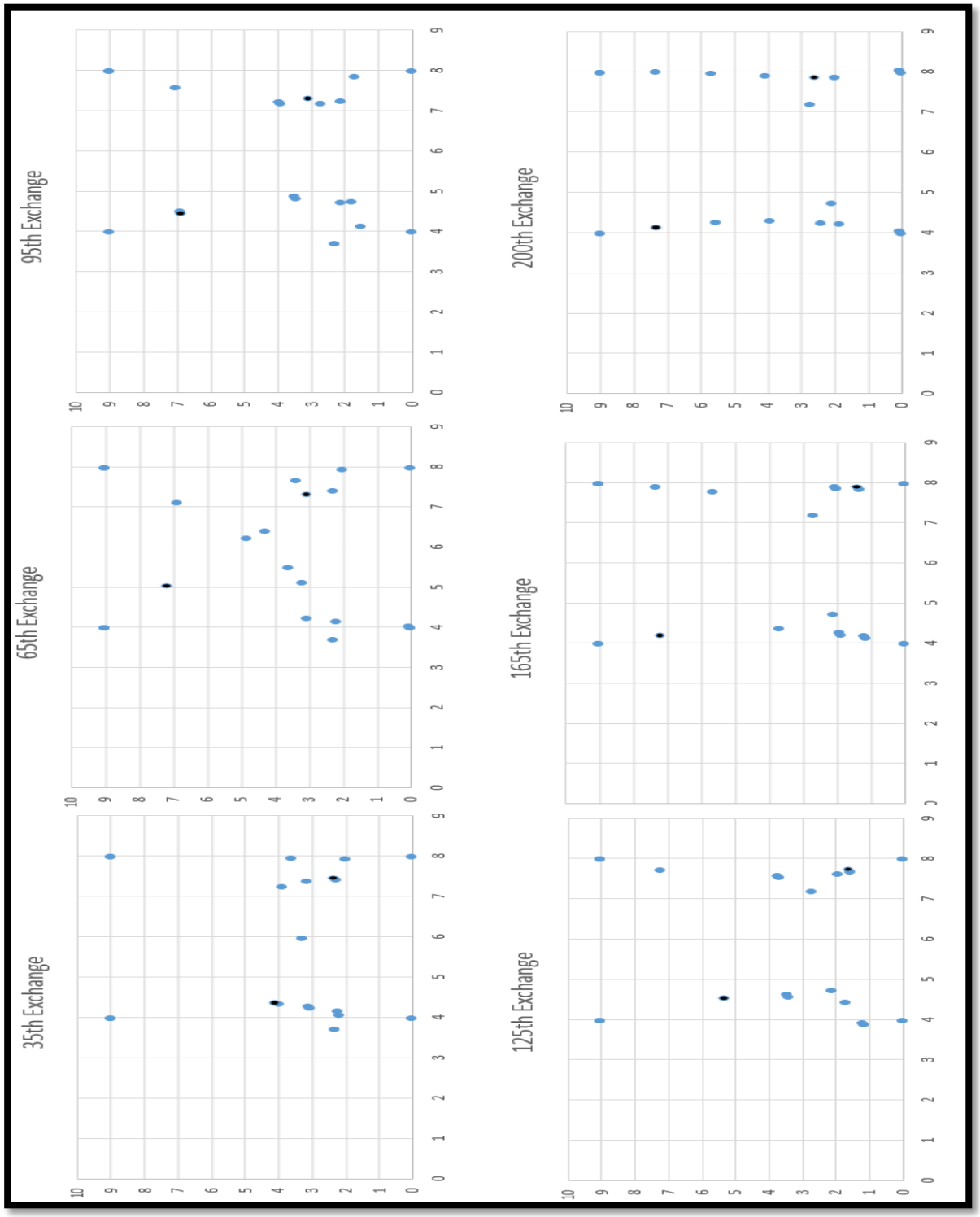

Fig. 34. Positions of Two Robots.

The range, velocity, and deceleration of an Entity are critical in the transportation process and the long term behavior of the Fire Brigade. A smaller range can leave the system susceptible to breaks in the line of sight leading to decreased performance since robots will forced to transport the bucket toward the Sink without further collaboration. The velocity on the other hand establishes the criteria for partner selection. The same scenario was run with initial velocity set to 3 m/s, deceleration to .5 m/s, and twenty-five robots. Fig. 35. illustrates the results in terms of bucket exchanges while Fig. 36. illustrates the results of the same scenario run in increments of fifty seconds.

While the difference between the results of the two approaches is minimal, both views are useful. Displaying the positions of the robots across time demonstrates the effect of collaboration and the time required for this effect to take place. For the first fifty seconds of the scenario's run time, the robots collaborate without managing to form any meaningful structures. Fifty seconds later, though, rough lines have been formed by the robots. With 150 seconds passed, the lines are distinct. This pattern does not change for the next 100 seconds, demonstrating that an efficient structure was formed which allowed the robots to continue the transportation process without minimal movement. This is further reinforced by the fact that only four buckets had been collected by the 50^{th} second. When the structure started forming, the buckets moved faster, resulting in the termination of the application at the 230^{th} second

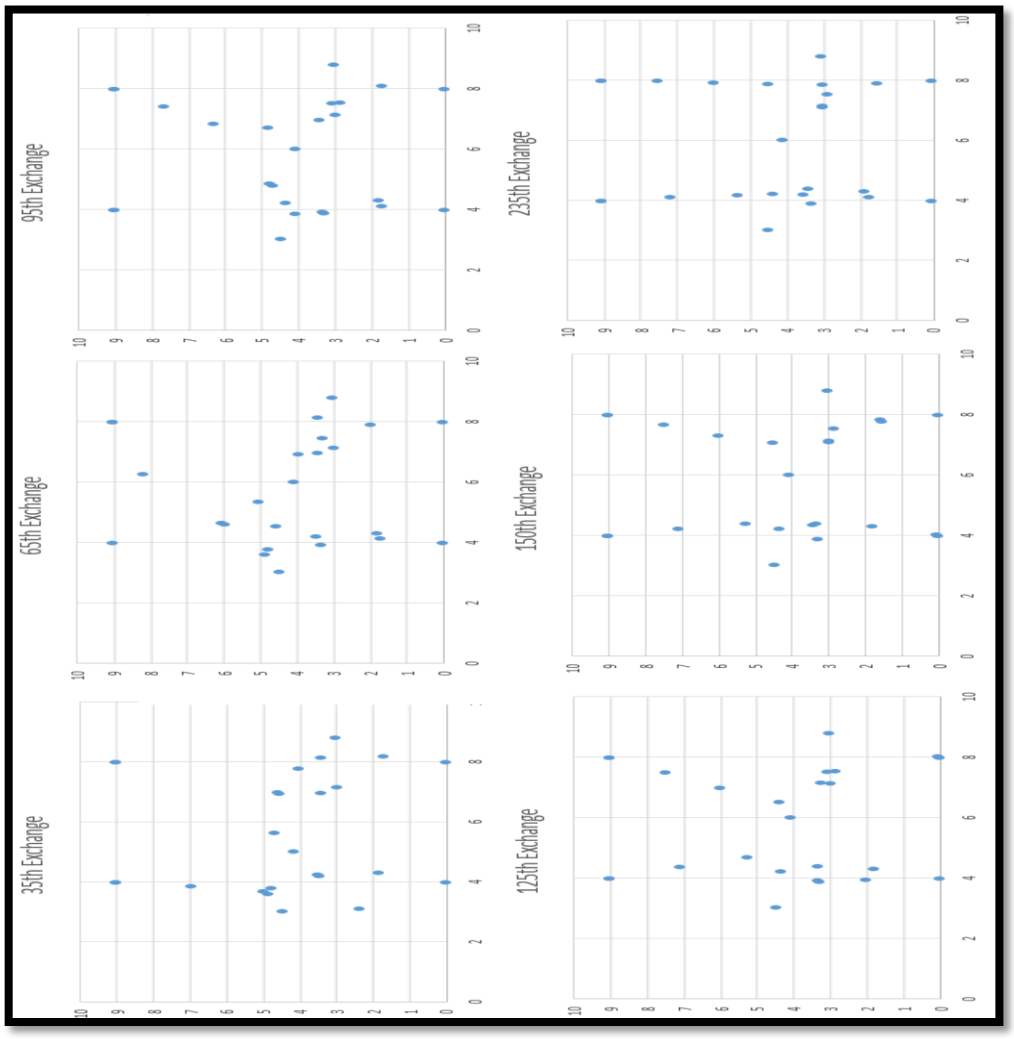

Fig. 35. Robot Positions of Updated Scenario.

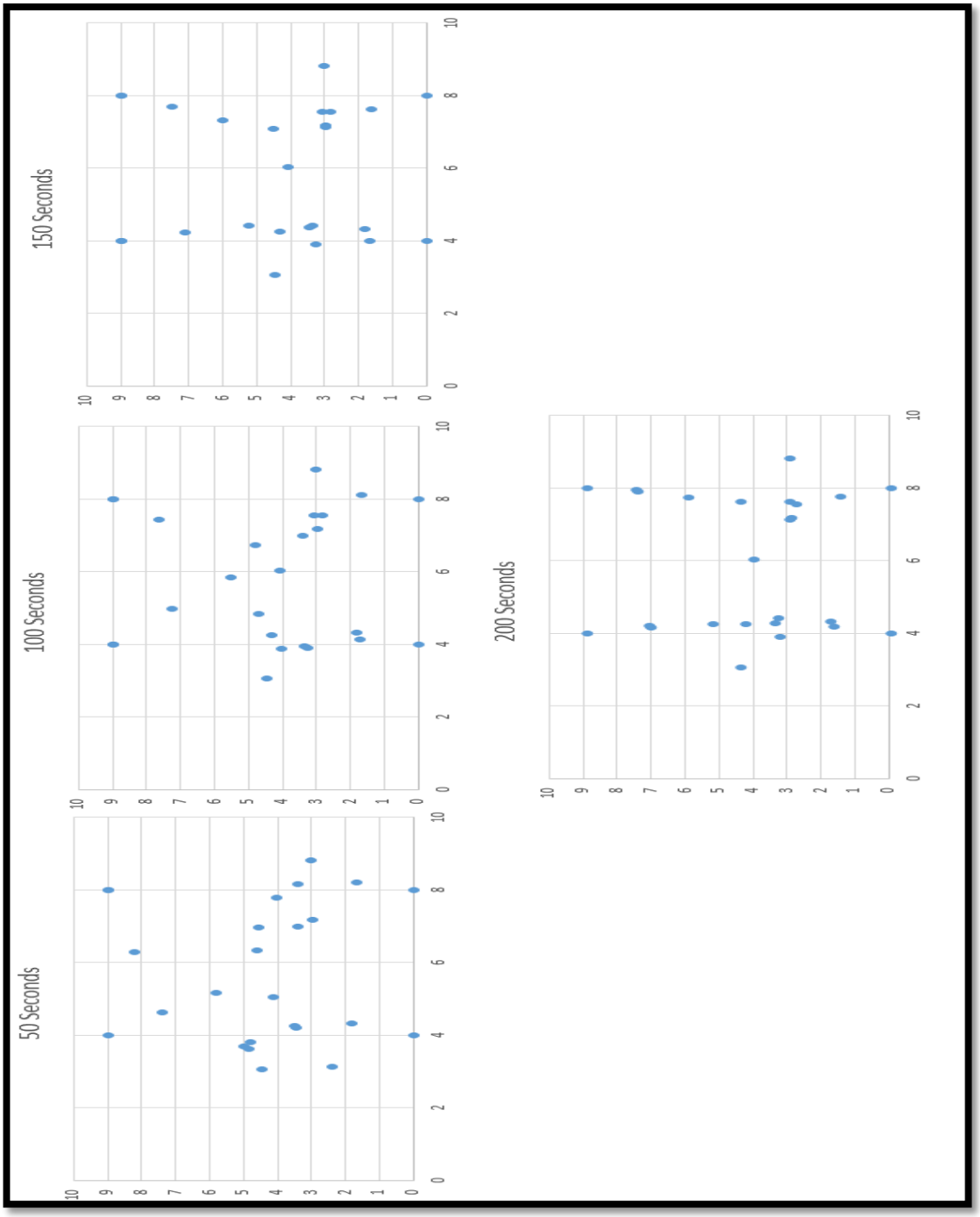

Fig. 36. Robot Positions Across Time.

5.8.2. Alternate Approaches

One of the most efficient methods of understanding the functionality of a system is to experiment with alternative approaches and compare the performance or other important metrics. The CAS framework and its layered structure greatly facilitate this process. A user can easily implement different approaches regarding the execution of specific layers and evaluate the system's behavior. An alternative approach will be investigated in this subsection.

An alternative approach was implemented regarding partner selection. In the original scenario, robots evaluate partnerships in terms of total transportation duration. One alternative approach is to disregard collaboration. As soon as Generators handle a bucket to an Entity, the Entity will simply transport the bucket to a Sink. A comparison between the run times of the two systems will illustrate the effectiveness of the collaboration method that was introduced in Section 5.2. This scenario run, which shared all the parameters of the scenario that was executed in Subsection 5.8.1, (robots:25, velocity: 3, deceleration: .5) was terminated at the 270th second; the original scenario was terminated at the 230th second. The collaborative approach showed a significant performance improvement. Taking into consideration the delay that is experienced by the five layer message traversal and the continuous request of coordinates and robots in range, the difference is substantial. According to the results, the hypothesis of the expected robot behavior that was expressed in the beginning of Section 5.8 is correct and the CAS framework made it easy to validate.

5.8.3. Impact of Framework

The framework-provided functionality assumed significant responsibility leaving to the user only the development of the actual layered behavior and the matching of interfaces. The framework consists of 669 lines of code. Of these 669 lines, only the interfaces need to be known

to the user, which allows the user to hook his or her behavior layers to the framework and utilize its functionality. Of course, detailed knowledge of the frame-work defined components would provide greater insight into the framework's functionality.

The Communication component spans 130 lines and its entire functionality runs in parallel to the layers and in the background, leaving the user oblivious to its execution. This component provides initiates automatic message receipt and interfaces with network. Without this component, the user would have to worry about the intricacies of network interfacing and would need to coordinate message handling by deserialization and distributing messages.

Layer (172 lines) handles internal communication. Of particular use is the intra-message functionality and the automatic extraction of messages. Coordinating communication would be more intricate if not for automatic intra-message replies and automatic forwarding. Another important aspect of Layer is the automatic layer connection which organizes the layer hierarchy and connects the layer to its neighbors, thus making internal communication possible. The user simply has to call a function (ConnectLayers) and the entire process is be completed on the background.

The framework contained some intricate functionality but the interaction between user and framework was rather simple. The implications of concurrent layer execution were unknown to the user. Parallel data manipulation, thread initialization, and termination were all handled by the framework. While, knowledge regarding concurrent layer execution should be known to the user, the effect and interaction between the two is non-observable, leaving the user to enjoy the benefits without any responsibility for that matter. Matching the interfaces was easy as well. Creating layers simply required an inheritance relation to be established, which involves the insertion of half a line of code, while the methods that the framework required to be defined and

were clearly distinguished by the word virtual in the interfaces. The virtual functions themselves did not require complex functionality, excluding the message serialization methods. Communication was also handled easily and only required calls to intuitive functions such as Send, SendUp, and SendDown.

The bulk of the Fire Brigade's running time was spent on user-defined methods. This is not surprising since the framework's functionality is either called once (structure formation and integration) or is rapidly executed (Send, Receive). The lines specific to the Fire Brigade example are 2,989. Nonetheless, the burden that was relieved from the user is significant and the fact that the interaction required is minimal and intuitive makes the CAS Framework beneficial in the development of collaborative autonomous systems.

CHAPTER 6

CONCLUSION

The CAS Framework was designed to meet the requirements of collaborative and autonomous systems and to allow their fusion by providing flexible structures and mitigating application specific characteristics. In the field of robotics, layered architectures are prevalent, which was demonstrated in Chapter 2 and even applies to hardware structures; a layered architecture for robotic hardware was described in [27]. By adopting a hierarchical layered architecture for the behavior of autonomous agents, similar to that of the Open Systems Interconnection (OSI) [28], the framework became compatible to many robotic applications and allowed for the desired feature of isolated layer development. The collaborative aspect was handled by providing communication capabilities to all layers and defining clear interfaces as well as imposing rules that shield against out of context interactions. These rules are minimal and the framework is very flexible, allowing the user to define both the number of layers and the information that flows between them.

In order to demonstrate the effectiveness of the frameworks' structure and capabilities, a Fire Brigade system was implemented. The design process of this application illustrated not only collaborative behavior in the results, but also how easy and intuitive it was to use the framework. In addition, its modularity and ease of swapping modules was demonstrated by changing a component of the system without any ramifications on the surrounding system's functionality. This replacement (alternative strategy layer) supplied additional results that provided insight to the system's behavior. This ease of experimentation regarding the system's behavior is a result of the layered structure's modularity.

The layered architecture provides substantial potential regarding the development of additional applications as well as hybrid virtual/physical systems. Of particular interest are applications that are implemented on physical robots but can interact with virtual environments. Such experimentation is expected to be easier with this layered approach which allows layer swapping. Similarly, using the same architecture, virtual and physical robots could interact, providing further possibilities for testing. This can be extremely useful and cost effective for numerous applications that deal with hazardous environments. Another benefit of the CAS Framework is that it can be compatible with multiple robotic platforms. The design of the robotic layers and the middleware that is placed between the framework and the hardware allows the user to mount an application to any robot, simply by swapping middleware which are hardware-specific software packages.

The CAS Framework was tested in a simulated environment successfully. The next step involves mounting the software to a physical robot and attempting to communicate with other agents, either physical or virtual, thus evaluating the interfacing process with both robotic and communication related hardware. Algorithmic improvements regarding the performance of concurrent layer execution are also intended, in the hope of making the system more accessible to less advanced hardware. The object oriented approach that was introduced requires substantial processing power, which is not that prevalent in many robotic platforms. Instead, the CAS framework can become more relevant as the processing power of autonomous robots increases. Finally, testing the integration of physical and virtual agents/environment is a very exciting prospect which can lead to more elaborate and demanding application evaluations and, thus, an increase in system reliability.

Frameworks that allowed the development of collaborative autonomous systems with significant flexibility do not currently exist in the literature. However, past literature did hint towards several of the requirements. The literature showed a great degree of cohesion, thus reinforcing the design choices that were made, such as the layered structure and the concurrent layer execution. While there are more features that can enhance the facilitation aspect of the framework, the current layered structure and communication capability are the basic building blocks of most collaborative autonomous systems. Their inclusion is intended to serve the CAS Framework's intended application to numerous other applications, just as it did for the Fire Brigade.

REFERENCES

[1] A. Elkady and T. Sobh. (2012, Jan.). Robotics Middleware: A Comprehensive Literature Survery and Attribute-Based Bibliography. *Journal of Robotics*. [Online]. Vol. 2012, pp. 1-15. Available: www.hindawi.com/journals/jr/2012/

[2] P. Antsaklis, K. Passino and S. Wang. (1990, Sept.). An Introduction to Autonomous Control Systems. Presented at Intelligent Control. [Online]. Available: http://ieeexplore.ieee.org.proxy.lib.odu.edu/xpl/

[3] R. Cattoni, G. Caro, M. Aste, and B. Caprile. (1994, Sept.). Bridging the Gap Between Planning and Reactivity: A Layered Architecture for Autonomous Indoor Navigation. Presented in Intelligent Robots and Systems Advanced Robotic Systems and the Real World. [Online]. Available: http://ieeexplore.ieee.org.proxy.lib.odu.edu/xpl/

[4] M. Alwan, E. Obeid, N. Kharma, and P. Cheung. (1994, Nov.). A Three-Layer Hybrid Archutecture for Planning in Autonomous Agents. Presented in Intelligent Information Systems. [Online]. Available: http://ieeexplore.ieee.org.proxy.lib.odu.edu/xpl

[5] S. Dertsen, J. Axelsson and J. Froberg. (2015, Apr.). An Analysis of a Layered System Architecture for Autonomous Construction Vehicles. Presented in Systems Conference.[Online]. Availble: http://ieeexplore.ieee.org.proxy.lib.odu.edu/xpl/

[6] 4D/RCS: A Reference Model Architecture For Unmanned Vehicle Systems Version 2.0. 2nd Edition., J. Albus, et. al., National Institute of Standards and Technology, Gaithersburg, MD., 2002, pp. 10-19.

[7] M. Naseer, M. Bokhari, and A. Ahmad. (2005, Dec.). A Multi-Layered Behavioral Architecture for Semi-Autonomous Agents. Presented in Pakistan Section Multitopic Conference. [Online]. Available: http://ieeexplore.ieee.org.proxy.lib.odu.edu/xpl

[8] H. Tang et.al. (2009, Aug.). Human-Robot Collaborative Teleoperation System for Semi-Autonomous Reconnaissance Robot. Presented in International Conference on Mechatronics and Automation. [Online]: Available: http://ieeexplore.ieee.org.proxy.lib.odu.edu/xpl

[9] L. Xiaopen, M. Hongwei, and Z. Zhihui. (2010, Mar.). Research on Open Control Architecture of Autonomous Mobile Robot with Multi-Layer and Modularization. Presented in Informatics in Control, Automatin and Robotics. [Online]. Available: http://ieeexplore.ieee.org.proxy.lib.odu.edu/xpl/

[10] I. Kostavelis, et al. (2012, Dec). Spartan: Developing a Vision System for Future Autonomous Space Exploration Robots. *Journal of Field Robotics*. [Online]. Vol. 31, pp. 107-140. Available: www.onlinelibrary.wiley.com/doi/10.1002/rob.21484

[11] E. Gat. On Three-Layer Architectures. *Artificial Intelligence and Mobile Robots*. [Online]. Vol. 195, pp. 210-220. Available: www.cs.uml.edu/~holly/91.549/readings/tla.pdf

[12] G. Brat, et. Al. (2006, Dec). A Robust Compositional Architecture for Autonomous Systems. Presented in IEEE Aerospace Conference. [Online]. Available: http://ieeexplore.ieee.org.proxy.lib.odu.edu/xpl/

[13] T. Estlin, R. Volpe, I. Nesnas, D. Mutz and F. Fisher. (2001, Jun.). Decision-Making in a Robotic Architecture for Autonomy. Presented in Proceedings of the International Symposium on Artificial Systems. [Online]. Available: www.ai.jpl.nasa.gov/public/planning/papers/

[14] F. Amato, F. Moscato, and P. Dario. (2015, Nov.). An Agent-based Model For Autonomous Planning in Distributed Critical Systems. Presented in 10th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing. [Online]. Available: http://ieeexplore.ieee.org.proxy.lib.odu.edu/xpl/

[15] K. Franke et al. (2014, Nov.). Architecture for CISS/CDAS within the Field of Cooperative Driving. Presented in International Conference on Connected Vehicles and Expo. [Online]. Available: http://ieeexplore.ieee.org.proxy.lib.odu.edu/xpl/

[16] H. Surmann and A. Morales. (2000, Nov.). A Five Layer Sensor Architecture for Autonomous Robots in Indoor Environments. Presented in International Symposium on Robotics and Automation. [Online]. Available: www.publica/fraunhofer.de/documents

[17] Y. Wenjian, S. Funchun, L. Huaping, S. Yu. (2009, Dec.). A Collaborative Navigation System for Autonomous Vehicle in Flat Terrain. Presented in IEEE International Conference on Control and Automation. [Online]. Available: http://ieeexplore.ieee.org.proxy.lib.odu.edu/xpl/

[18] J. Hertzberg, et. al. (1998, Sept.). A Framework for Plan Execution in Behavior-Based Robots. Presented in Intelligent Control. [Online]. Available: http://ieeexplore.ieee.org.proxy.lib.odu.edu/xpl/

[19] I. Fernando, F. Henskens and M. Cohen. (2012, Jun.). A Collaborative and Layered Approach (CLAP) for Medical Expert System Development: A Software Process Model. Presented in IEEE/ACIS 11th International Conference on Computer and Information Science. [Online]. Available: http://ieeexplore.ieee.org.proxy.lib.odu.edu/xpl/

[20] L. Gao, Z. Liu, and Z. Li. (2011, Jun.). Research on Architecture Model with Autonomous Coordination for Distributed Satellite Systems. Presented in Computer Science and Service System. [Online]. Available: http://ieeexplore.ieee.org.proxy.lib.odu.edu/xpl/

[21] J. Hightower, B. Brumitt and G. Borriello. (2002, Jul.). The Location Stack: A Layered Model for Location in Ubiquitous Computing. Presented in Proceedings of the Fourth IEEE Workshop on Mobile Computing Systems and Applications. [Online]. Available: http://ieeexplore.ieee.org.proxy.lib.odu.edu/xpl/

[22] M. Serna, C. Uran, and K. Uribe. (2011, Jul.). Collaborative Autonomous Systems in Model sof Urban Logistics. *Network of Scientific Journals from Latin America*. [Online]. Vol. 79, pp. 171-179. Available: www.revistas.unal.edu.co/index.php/dyna/article/view

[23] R. Briggs, et al. (2009, Dec.). A Seven-Layer Model of Collaboration: Separation of Concerns for Designers of Collaboration Systems. Presented in International Conference on Information Systems. [Online]. Available: www.aisel.aisnet.org/icis2009/26/

[24] Y. Chen, et. al. (2006, Jun.). Market-Based Collaborations for Autonomous Operations of Unmanned Air Vehicles. Presented in IEEE Worshop on Distributed Intelligent Systems. [Online]. Available: http://ieeexplore.ieee.org.proxy.lib.odu.edu/xpl/

[25] N. Goerke. "A Hierarchical Layered Architecture for Intelligent Control of Autonomous Robots," unpublished.

[26] H. Yin, et. al. (2013, Sept.) Research of Space Robot Autonomous Operation Architecture. Presented in 5th International Conference on Intelligent Networking and Collaborative Systems. [Online]. Available: http://ieeexplore.ieee.org.proxy.lib.odu.edu/xpl/

[27] M. Berger, et al. (1997, Jul.). A Modular, Layered Client-Server Control Architecture for Autonomous Mobile Robots. Presented in Industrial Electronics Conference. [Online]. Available: http://ieeexplore.ieee.org.proxy.lib.odu.edu/xpl/

[28] H. Zimmerman. (1980, April). OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection. *IEEE Transactions on Communications*. [Online]. Vol. 28, , pp. 425-432. Available: http://ieeexplore.ieee.org.proxy.lib.odu.edu/xpl/

VITA

IOANNIS SAKIOTIS

PERSONAL INFORMATION

Born: 12/27/1993
Citizenship: Greek
Email address: isaki001@odu.edu

EDUCATION

Old Dominion University, Norfolk, VA
Frank Batten College of Engineering and Technology, Modeling, Simulation and Vizualization Engineering Department
Bachelor of Science in Modeling and Simulation Engineering, 2014

AWARDS AND HONORS

2016 Damalas Family Scholarship from the AHEPA organization
2015 Kotarides Family Scholarship from the AHEPA organization

PUBLICATIONS

- "Discrete Event Simulation for Supporting Production, Planning, and Scheduling Decision in Job Shop Facilities" ModSim World Conference, Norfolk 2014.
- **"**Discrete Event Simulation Implementation of a Production Planning and Scheduling Tool", Capstone Design Conference, Hampton 2014.