Multiple Agent Architecture for a Multiple Robot System

by

Bram Aaron Bakst Gruneir

A thesis

presented to the University of Waterloo

in fulfilment of the

thesis requirement for the degree of

Master of Applied Science

in

Systems Design Engineering

Waterloo, Ontario, Canada, 2005

AUTHOR'S DECLARATION FOR ELECTRONIC SUBMISSION OF A THESIS

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Controlling systems with multiple robots is quickly becoming the next large hurdle that must be overcome for groups of robots to successfully function as a team. An agent oriented approach for this problem is presented in this thesis. By using an agent oriented method, the robots can act independently yet still work together. To be able to establish communities of robots, a basic agent oriented control system for each robot must first be implemented. This thesis introduces a novel method to create Physical Robot Agents, promoting a separation of cognitive and reactive behaviours into a two layer system. These layers are further abstracted into key subsections that are required for the Physical Robot Agents to function. To test this architecture, experiments are performed with physical robots to determine the feasibility of this approach.

A real-time implementation of a Physical Robot Agent would greatly expand its field of use. The speed of internal communication is analyzed to validate the application of this architecture to real-time tasks.

It is concluded that the Physical Robot Agents are well suited for multiple robot systems and that real-time applications are feasible.

Acknowledgments

I would first off like to thank my supervisors, Dr. Mohamed Kamel and Dr. Hamada Ghenniwa, for their time, advice and support.

I would like to thank my readers, Dr. Hamid Tizhoosh and Dr. Otman Basir, for their time reviewing this thesis.

I would like to express my gratitude to the Autonomous Systems and Real Time Distributed Design Focus Group for being a sounding board for most of the ideas presented in this thesis. The focus group consists of Alaa Khamis, Ali Tehrani, Mohamed El-Abd, Ben Miners, Insop Song, Hongwei (Howard) Zhu and others.

I would also like to thank Ben Miners who not only wrote most of the Action Layer code, but also edited this thesis. I need to thank Alaa Khamis who edited this thesis as well.

I am forever in debt to my parents, Robert and Marilyn Gruneir, who have now stuck by me for two engineering degrees. They have always supported me and have even edited very rough drafts of this dissertation. Not an envious task.

And finally, Maria Mammoliti, for editing the thesis, again and again, filming and rendering the DVD and making the graphics look pretty. That; and she put up with me.

Dedication

I dedicate this thesis to my grandmothers, Ellen Gruneir and Doris Bakst. Without them, I would not be the person that I am today.

They both will be missed.

Table of Contents

List of Figures

List of Tables

Table of Acronyms

- ACL – Agent Communication Language
- CIR-Agent – Coordinated Intelligent Rational Agent
- FIPA – Foundation for Intelligent Physical Agents
- IPC – Inter-Process Communication
- JADE – Java Agent Development Framework
- KQML – Knowledge Query and Manipulation Language
- PAMI Lab – Pattern Analysis and Machine Intelligence Lab
- PATA – Parallel Advanced Technology Attachment
- PRA – Physical Robot Agent
- RAM – Random Access Memory
- SATA – Serial Advanced Technology Attachment
- SCSI – Small Computer System Interface

- TCP – Transmission Control Protocol

- UDP – User Datagram Protocol

1 Introduction

Both multiple agent (multi-agent) systems and multiple robot (multi-robot) systems have existed for many years. However, it is only recently that the integration of these fields is being explored. This thesis delves into the field of multi-agent approaches for multi-robot systems and presents a framework in which multiple robot systems can be quickly programmed and tested. This first chapter contains an overview of the motivation and structure of this thesis.

1.1 Motivation

With recent advances in agent oriented languages and frameworks, such as the Java Agent DEvelopment Framework (JADE) [1] and Agent-0, it is now possible to both design and implement complex multiple agent systems on a wide variety of systems. A list of agent oriented languages can be found in [25]. The application of agent oriented approaches to multiple robot systems has focused mainly on robotic simulations or using very simple robots. This work in this thesis is designed to help bridge that gap and foster an environment which allows quick implementations of multi-agent, multi-robot system on complex mobile robots.

In current research, no mention is made of the internal software structure of the robots and the specifics of this area are left up to each individual implementation. When implementations are robot and system dependant, there is no portability in these systems' software. By describing in detail the internal structure of the robots and using this

information to find a common level of abstraction, all future systems contribute to improving portability.

To accomplish these goals, a new software architecture describing the internal software components is presented for the robots. By structuring and restricting the flow of information within it, the robots can be autonomous and co-operative. The architecture allows for the implementation of most types of hierarchies, using different types of robots. The agent level of robots is also standardized and interchangeable. This architecture is described in detail and implemented within this thesis.

1.2 Thesis Scope

This thesis covers both the theoretical and implementation aspects of a new architecture for multi-robot systems based on an agent oriented design. This architecture uses a multi-agent solution to enable a fully distributed system. Due to the wide variety of possible systems, no single code base is applicable, so implementation guidelines are put forward as opposed to hard rules. As well, a functional implementation is presented. The source code for this implementation and a DVD demonstrating the functioning of the experiment can be found in Appendix D.

With the constraints of real-time systems, the basic requirements of a multi-robot system are greatly increased. This thesis discusses in depth both the methods and requirements of the communications in the presented architecture. A comparison of the speed of the communication within the architecture is presented.

1.3 Contribution

The primary goal of this thesis is to present a new architecture for automated multi-robot systems based on a multi-agent oriented approach. It allows for a large variety of systems including multiple hierarchies including centralized and distributed systems. It looks not only at the method that the robots use to communicate with each other, but also describes the internal software structure of each robot. This structure allows completely different robots to work together; it allows the robot's lower levels to be treated as a "black box" so a robot can be programmed regardless of how its hardware functions. This fosters an environment whereby the same code can be used on multiple robots for different tasks, greatly increasing portability. The architecture can be used with small (single robot) and large (hundreds of robots) systems. The internal communications between layers is examined for the possible use for real-time applications.

1.4 Thesis Organization

This thesis is split into 6 chapters which can be seen as four parts. The first 2 chapters contain the background information for both this thesis and other work in the field. Chapter 1 is the introduction in which the motivation, scope and organization of the thesis are discussed. Chapter 2 contains the background information required for the rest of the thesis. This chapter contains basic definitions and some of the important concepts addressed throughout the rest of the thesis. It also contains reviews of relevant research in the field. It examines where other research is currently with respect to the contributions made within this thesis.

The second part, consisting of 2 chapters, describes the new architecture. Chapter 3 describes this architecture in detail including the nature of the communications that are required within the architecture. Chapter 4 details how an implementation of the architecture could proceed.

The third part, chapter 5, is that of experimentation in which two different experiments are performed. The first demonstrates a fully realized version of the architecture with an analysis of its performance. An examination of some of the different communication methods for intra-robot communications is presented in the second experiment.

The conclusions are located in chapter 6. It contains the conclusions, the limitations of architecture and recommendations for future directions and further research.

The appendices consist of both the source code used to create the architecture and communications experiments as well as a DVD video of the architecture experiment being performed.

2 Background and Literature Review

This chapter contains definitions and information on the key terms and concepts presented within this thesis as well as reviews of relevant literature.

2.1 Robots

This section describes some of the challenges in dealing with multiple robot systems design and presents a definition of the term 'robot'.

2.1.1 Definition

The following definition of a robot was taken from Wikipedia [28]:

> In practical usage, a robot is a mechanical device which performs automated physical tasks, either according to direct human supervision, a pre-defined program or, a set of general guidelines using artificial intelligence techniques. Robots are typically used to do the tasks that are too dirty, dangerous, difficult, repetitive or dull for humans.

Most research robots are computers with sensors, a means of locomotion and a method of communication. Although the work in this thesis is performed on mobile robots, the architecture is applicable to all robot types.

2.1.2 Multi-Robot Design

A multi-robot system is a system in which several robots function at the same time to achieve a goal. Typically, the use of more than one robot will either allow the completion of said goal, or make a marked improvement over a single robot system.

Multi-Robot design does however bring with it the complications of increased complexity of communications and the possibility of physical collisions between robots. As the number of robots increases, functioning as a team can be daunting. One way to facilitate the programming of multi-robot systems is to use an agent oriented approach. An example of a multi-robot system can be found in [23], where two robots try to push a box together. Even when not explicitly mentioned, as in Mataric, Nilsson and Simsarian's paper, a agent oriented design is being presented. Agents and multi-agent design are discussed in the following section.

2.2 Agents and Agent Oriented Design

This subsection defines agents, examines multi-agent design and some of the more common hierarchies. A specific type of agent is examined and some of the complications from real-time systems are discussed.

2.2.1 Definition

Before moving forward in this thesis, it is important to establish an understanding on the nature of agents. Agents are a fairly new and highly contested programming abstraction and it is difficult to find a consensus of their full definition. For the purposes of this thesis, an agent is defined as an entity that has the following concepts: it must be autonomous; it must have a memory; it must be able to communicate with other agents; it requires a method to understand the data communicated to it and finally, it must contain some form of a problem solver. These restrictions are taken from the work of Wooldridge, a highly respected researcher in this new field [29].

2.2.2 Multi-Agent Design

Similar to multi-robot design, multi-agent design occurs whenever more than one agent works together to complete a task. In [18], Jennings et al. present a history of multi-agent design. This is the area where agents are designed to be able to function well. Since each agent has a means of communicating with the other agents, they have the ability to solve problems collectively. There are many different schemes for assigning duties to different agents, including an auction or by delegation of duties by a captain. As the number of agents grow, more multi-layered structured approaches, or hierarchies, are required. The most common hierarchies are discussed in the following subsection. The structure of the agents themselves is also discussed.

2.2.2.1 Multi-Agent Hierarchies

Every multi-agent system needs an overall organizational hierarchy [24]. A two agent system can either function as a partnership or as a chief and an assistant. If one extends this example to any number of agents, a system where there is one chief and many assistants is known as a centralized system. A hierarchy where all of the agents derive collective decisions is known as a distributed system.

There are a number of advantages to using a centralized system hierarchy. By having a single controlling agent, co-ordination and co-operation are relatively easy to achieve. The managing agent's primary role is to monitor the overall system and direct the subordinate agents' actions. All commands must pass through this central agent to ensure the stability of the system. A centralized system is the easiest type of system to program but with this advantage also come some disadvantages. A single agent-dependent system

can collapse if the chief agent is disabled or its communications are interrupted. To prevent this from occurring, the leader agent must be designed to handle a large number of communications and process large amounts of data. As the number of agents in the system increase, additional responsibility is designated to the chief agent.

On the other end of the spectrum is a distributed system. In a true distributed system, no one agent is in charge and decisions are based on collective agreements. Unlike the centralized system, the removal of an agent and the addition of a new one is a simple process. Distributed systems can become large but this size should not overwhelm any single agent's ability to communicate or process information. As well, since decisions are not being made by one chief agent, communications will not be as strained and a special agent does not need to be created to handle communications between agents. The cost of these benefits occurs within each agent's complexity. As each agent must assist in the decision making, problem solving abilities must be built into them. In addition, if not create properly, distributed systems can easily become unstable and not solve problems, due to agent disagreement (infighting).

To deal with some of the disadvantages presented in a centralized system, a hierarchical system can be used. It is similar to a centralized system but it does not have a single chief, but a full chain, or, more accurately, a tree of command. A good example of a hierarchical system is an army. Every agent, except for the head one, reports to a chief agent to which all questions are sent and from which all orders are received. The primary concept in a hierarchical system is to lessen the amount of work which the head agent has to resolve. This way, the head agent may delegate a full range of tasks to lower level

groups. This type of system solves most of the communication problems with the exception of times when an agent is added or removed from the system. If any agent is removed, all agents under its command are suddenly removed from the system as well. Using a hierarchical system allows for a much larger number of agents to work together than that which could work within a simple centralized system. However, with this added complexity, the agents themselves need to be more complicated to handle giving and receiving orders.

To circumvent some of the problems of distributed systems, a holonic system hierarchy can be used [30]. Holonic systems use teams to handle tasks. When a task needs to be performed by more than one agent, a group of agents organize themselves into a team to allow decisions to be made faster and to reduce the complexity of using a fully distributed system. An agent can be part of more than one team at the same time with completely different agents or even two teams with exactly the same agents. If more agents are required to perform a task, they can be recruited onto the team. More programming complexity is required within a holonic system than a basic distributed one, but it ensures faster decision making and quicker response times.

There are many other examples of systems that can be used from multi-agent systems hierarchy, but most are hybrids of the ones already presented. When choosing which type of hierarchy to use, the goals of the systems are paramount. It may be pointless to create an extremely complex holonic system when a hierarchical one will perform sufficiently. Some of the more important factors in making this decision are the number of agents being used and the response time of the system. For the most part, agent

oriented design tends to push towards more distributed style systems to allow the agents greater autonomy.

The architecture presented in this thesis allows for all of the aforementioned hierarchies to be implemented. By creating an agent friendly environment, the architecture facilitates the development of all types of systems.

2.2.3 Multi-Agent Robotic Systems

While there has been extensive research involving multi-agent system hierarchies, there is very little on the internal structures of the agents themselves. This section will present systems that have some similarity to the architecture presented in this thesis.

Tigli's and Thomas's paper [27] explores the different agent configurations that can be used to control a robot. The mobile robot agent discussed in the paper is, in essence, the Physical Robot Agent (PRA) used in this thesis. By dividing the internal workings of the mobile robot agent, an internal multi-agent system can be used to control each robot. This is similar to the work that is performed in this thesis which uses a software multi-agent system to control each robot's Cognitive Layer (see chapter 3). However, it is not as structured as the one proposed in this thesis and there are no clear information pathways that must be followed. This could allow for conflicts to occur within the robot, making design much more difficult. See [15] for another multi-agent system that is used to control mobile robots, but one in which the tasks are somewhat more defined.

Kawamura's paper [20] looks at using miniature or atomic agents as small components of a larger agent. Kawamura's system however, is more structured than Tigli's and Thomas's work mentioned above. Even though his system is designed for human-robotic agent interaction, it could be used for agent-agent interaction as well. This system is similar to the software agents used within the Cognitive Layer.

Lucidarme, Simonin and Liégeois present a single layer system in [22]. This system contains a separation between an interaction element and an action element which is the primary difference between the structure demonstrated in their paper and the one described in this thesis. The interaction element is not similar to the Cognitive Layer as it does not make any decisions; it just is directed by the action element. Furthermore, the interaction element can only receive messages from the action element, not reply. By taking the interaction away from the decision making, it is more difficult for multi-agent systems to come to agreements.

In Cossentino, Sabatucci and Chella's paper [4], a very strict methodology is presented to streamline the development of multi-agent systems. One of the main points in the paper is to maximize the amount of pattern reuse, as it can greatly decrease the amount of work needed to produce new systems. This can occur by reusing an agent, similar to what is discussed in chapter 5, or even better is the use of a repository of tasks, which is implemented in the architecture presented in this thesis (see section 4.1.2).

2.2.4 CIR Agent

The Coordinated Intelligent Rational Agent (CIR-Agent) model is an agent layout taken from work by Kamel, Ghenniwa and Eze [11] [6] in which each agent consists of four key blocks: a knowledge base; a problem solver; an interaction component and a communication method. These can all be seen in Figure 2.1. This is the main structure of an agent that is used throughout this thesis.

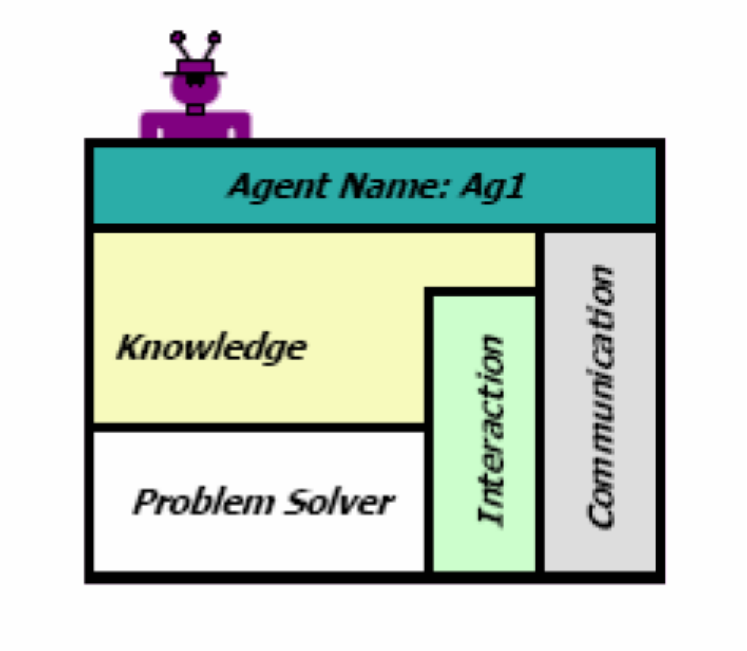

Figure 2.1: CIR Agent

The knowledge base is where all agent information and memory is stored. This includes its goals, its current state and its history. This is the pool from which it can access the required protocols for communication and which other agents have been contacted.

The problem solver determines the exact solution and how it should be applied to a problem. In basic agents, this can simply be a state machine; in a more complex agent,

the problem solver can construct a goal based strategy which incorporates the cooperation of other agents.

The interaction component is the method that is used for communication with other agents. This includes methods of handshaking, bartering and bidding.

The communication method is how the agent communicates with other agents using an agent communication language (ACL). There are a few common ACLs such as the Foundation for Intelligent Physical Agent's communication language (FIPA's ACL) [8] and the Knowledge Query and Manipulation Language (KQML) [7]. An introduction to ACLs can be found in [10].

2.2.5 Real-Time Multi-Agent Design

When an agent is designed to function in real-time, it must have the ability to guarantee meeting specific deadlines. This further complication to multi-agent design means that the system must be able to respond within this constraint, even in the worst-case scenario. In order to achieve this, all possible time components of the system must be considered. As an experiment in this thesis, some of the communications that occur within the robot are examined.

2.2.6 Agent Oriented Design and Software Agents

Throughout this thesis, two types of agents are discussed. The first is the overall agent, such as the CIR agent or the Physical Robot Agent discussed in future chapters. The second is a software agent which arises when programming in an agent oriented

language. A software agent is a small amount of code that acts as an agent and is helpful when programming an agent oriented system, but it is not representational of a fully fledged Physical Robot Agent. Software agents can be seen as components of these larger constructs.

2.3 Communications

The exchanging of data between agents is vitally important to the health of a multi-agent system. This section will briefly examine some pertinent literature dealing with methods and protocols of communication.

2.3.1 Agent/Robot Intercommunication

Berna-Koes et al. in [2], try to improve communication efficiency within a multi-agent context by having the agents use a "back-channel" to handle all high load communication requests. Standard methods of communications can handle most requests, but when a high load is required, a back-channel that exists between agents can be used. The result is a similar method to shared memory except that requests must be made to receive data. For systems that require large amounts of data exchange, as discussed in chapter 4 of this thesis, this system should work. However, unlike shared memory, it uses up large amounts of bandwidth, the same bandwidth that is needed for standard communications.

2.3.2 Communication Protocol Performance

In *Network Protocols for Mobile Robot Systems* [13], Harmon and Gage conclude that the best communications protocol is UDP. This is based on the slow throughput of the

transport layer protocol and the extra processing power required for it. This conclusion mirrors the conclusions reached in section 5.2, where three types of communication are examined.

Howell et al [14] explore the challenges associated with using the user datagram protocol (UDP) in ad hoc networks. Their results indicate the best packet size to reduce the signal to noise ration is 784 bytes. This result adds to the results found in section 5.2, where an exponential relationship between the number of packets sent and the delays associated with their transfer is explored.

Gao, Yan, Ding and Huang [9] attempt to create a new protocol for multi-agent multi-robot communications. This protocol can only be used for mobile agents and should only be used to transmit specific types of commands. With many other pre-existing protocols that will perform just as well, there does not seem to be a need for its use.

In [21], Lei Cheng and Yong-Ji Wang examine fault tolerances when trying to get a group of mobile robots into a formation. They use a combination of both UDP and the transmission control protocol (TCP) for communication between robots. For broadcast messages, they use UDP and for inter-robot communication TCP is used. This is a good strategy, as when network faults occur, TCP can be used, see section 5.2.

3 Architecture of a Physical Robot Agent

Many issues must be considered when designing a multi-robot system such as autonomy, cooperation, communication structure and coordination. Collective autonomy refers to the ability of the robots to work individually and without human intervention. Cooperation is the ability of the robots to work with each other and requires communication whenever the robots actions depend critically on knowledge that is accessible only from another agent. Coordination addresses the interdependency management among the cooperative robots to achieve individual or collective goal(s).

All of these issues can be addressed using an agent oriented approach. Taking into account that the system deals with physical robots, not simulated ones, a completely agent-based solution is difficult due to the lack of low level control (e.g. actuators and sensors) in agent-based languages. In addition, as the agents do not necessarily exist within the robot, having the low level controls reside in the agent would not be practical. To solve this problem, the concept of Physical Robot Agent (PRA) described in [5] is adopted. As shown in Figure 3.1, a PRA splits up the sensory/action (physical) and decision making (cognitive) processes into a two layer system.

Based on the PRA concept, a new architecture has been developed as shown in Figure 3.2. It consists of the Action Layer, which handles all of the sensory and movement functions; and the Cognitive Layer, which handles the decision making. All of the internal software components are new concepts presented by this thesis.

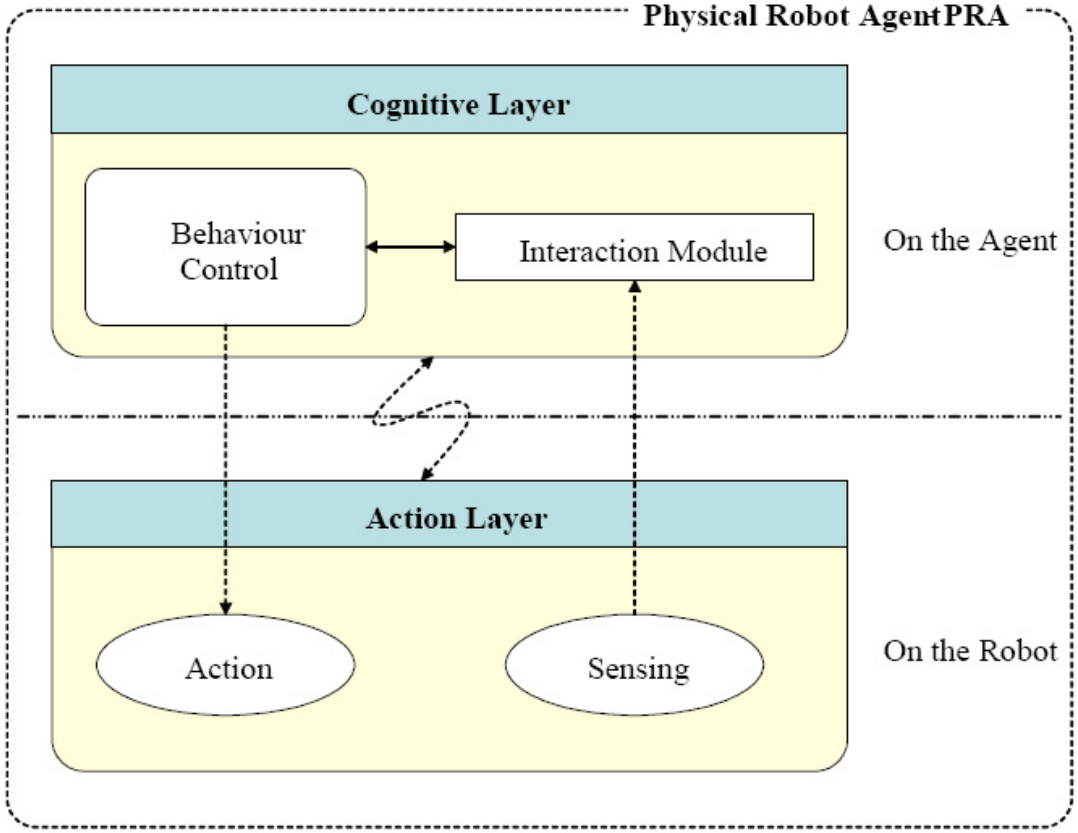

Figure 3.1: Physical Robot Agent

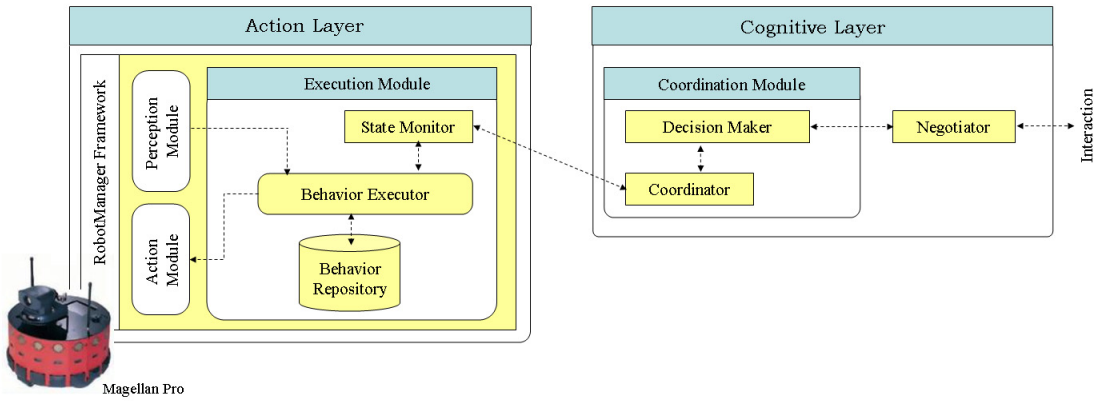

Figure 3.2: Proposed Architecture

This layering system is inspired from ethology, the science of studying animal behaviours, where the Cognitive Layer represents the conscious brain and the Action Layer represents a combination of both the body and the unconscious brain. For example, when controlling a limb, the Action Layer understands the inner workings of the movement as well as the touch and heat sensors, but the overall goal is described by the Cognitive Layer.

The proposed architecture in this thesis has been published in [12]. In this paper, a simplified version of the work presented in this thesis is discussed. The paper includes the proposed architecture located in this chapter, and the experiment located in section 5.1. The following subsections describe these two layers and their internal elements.

3.1 Action Layer

The Action Layer is where the physical actions and sensory parts of the robot are located. In this layer, tasks or reactions are controlled and executed. These tasks and reactions are simple programs that are controlled from the Cognitive Layer. The Action Layer consists of three key elements: the Executor, the Repository and the State Monitor as shown in Figure 3.2. This subsection will briefly describe both the function and importance of these three elements.

3.1.1 Executor

For a PRA to affect the physical world, it must have the ability to manifest itself in some way. The Executor is responsible for controlling and performing all of the physical

actions of the robot. It manages the actuators and receives feedback from the perception modules. The State Monitor communicates with the Executor negotiation the tasks to be executed. The Repository captures how and action should be performed. In return, the Executor communicates all updated variables to the State Monitor. The Executor is the only element in the PRA that has access to sensors and actuators. This part of the abstraction is critical, as it ensures that all physical manifestations are controlled in the same element. If more than one element has access to the sensors and actuators, then the possibility exists, of more than one system trying to control the same component. By having the Executor perform this task, optimizations can occur. If a control algorithm needs adjustment, it must only be changed in a single place. The worst-case execution time of a task in a hard real-time system must be known to determine scheduling. When information is taking multiple paths, it is much more difficult to estimate the time required for an operation. Without the Executor, a robot will not be able to interact with its surrounding world as it could not affect it in any way.

3.1.2 Repository of Tasks

The Repository is the knowledge base of the Action Layer. For a single use robot, the Repository element is not essential as there are very few tasks that will be required. When a PRA becomes more detailed however, the Repository becomes essential. The Repository is a collection of known tasks that can be run by the Executor. It communicates with the State Monitor to determine which tasks are required in the Executor and to respond to queries about the available tasks. In a learning PRA, the

Repository should update its tasks or even create new ones, based on feedback from the Cognitive Layer (through the State Monitor).

By having a library of common tasks available, the Repository supports development of the system in which the Action Layer can be considered a "black box". Development of the two layers can be considered independent of each other. A large repertoire of tasks is needed for a truly separated design; nonetheless, with a few basic movement commands being stored, very complicated Cognitive Layer designs can be created. Furthermore, when using the Repository as part of the dual layer system, this independent development can be extended to robots that are completely different in low level functionality. As the Action Layer provides a universal abstraction for the underlying hardware (e.g. 3 or 4 wheels for the robot), the Cognitive Layer can be exactly the same on different PRA and still achieve the exact same results.

3.1.3 State Monitor

The State Monitor is the Action Layer's communication channel through which is the only method that it can communicate with the Cognitive Layer. The complex nature of all the interactions between the layers gives rise to an element in both layers designed to handle the intricacies of these communications. In order for the Executor and the Repository to run tasks, they must know which ones are required. Further, in order for the Cognitive Layer to have a full picture of the current status of the PRA, the State Monitor must both package the data and inform it of the updates.

The State Monitor's information restriction is important to ensure that data flows through the same channels. This way, all PRAs will have a similar interface with which to connect despite the type of robot being used. For real-time considerations, where the worst-case is the most important, variable updates and task command latency can be measured through it. The State Monitor ensures the isolation of the Action Layer and allows only the proper Cognitive Layer to contact it. For these reasons, the State Monitor element is required.

3.2 Cognitive Layer

Unlike the Action Layer, in which a set of tasks plays the most important role, in the Cognitive Layer, all high level decision making is performed. It is the Cognitive Layer that makes the PRA autonomous. The structure of the Cognitive Layer can be varied, however it must be able to control the robot via commands to the Action Layer and communicate with the other agents when in a multi-PRA situation. These two types of communication, inter-robot (between PRAs) and intra-robot (or inter-layer within the PRA) are the only means that the Cognitive Layer uses to receive information (see discussion in chapter 3.5). It receives status updates from the Action Layer's State Monitor and uses these updates to determine which course of action to pursue. Even with the vastly different requirements for different systems, three main elements are always required in the Cognitive Layer: the Decision Maker, the Negotiator and the Coordinator (see Figure 3.2). The Cognitive Layer is where the agents reside in this system.

This section will outline the abstracted elements of the Cognitive Layer.

3.2.1 Decision Maker

The Decision Maker represents the problem solver of the PRA. Everything that occurs within the PRA of any consequence must be sanctioned by the Decision Maker. Without this element, the robot would not be autonomous, would not be able to adapt to new situations and would not be able to form consensus with other robots. From the Decision Maker, commands are sent to the Action Layer via the Coordinator and inter-robot communications are facilitated through the Negotiator. The Decision Maker is a composition of several components including memory and problem solving by extrapolating the elements of a CIR Agent.

3.2.2 Negotiator

The Negotiator is the element by which the agents interact with other agents. What makes the Negotiator significant is the concept that all communications from other robots must pass through the Decision Maker. This ensures that the Decision Maker is always aware of the Action Layer's status. Without the Decision Maker's consent, no external command will ever be executed (such as a command from another PRA) and no variable update to an external source will ever occur.

Most robot architectures have a negotiator of within their PRA description; it is typically called the interaction component. What makes this one unique is the restriction on information flow; that all information passes through the Decision Maker.

3.2.3 Coordinator

The Coordinator is the element that is in charge of communications with the Action Layer. It should only receive communications from the Action Layer and the Decision Maker. It maintains this single pathway of communications; it ensures that the Decision Maker is always in command of all aspects of the PRA. By emphasizing this element, this strict dataflow is enforced.

Two additional reasons for the existence of the Coordinator as a separate element are to mark it as distinct from the Negotiator and emphasize its repetitive nature. Once the Coordinator has been created on a system, it should not need to be changed often, even for different tasks.

3.3 Hierarchies

The separation between the Cognitive and Action Layers is a buffer that allows most types of hierarchies to be used in the system. The restriction to the types of hierarchies allowed occurs when one PRA has control of another. This is not allowed to occur directly as it would remove the autonomous nature of the agent.

3.4 CIR-Agent

Even with the dual layer nature of the system, the CIR-Agent is still present within a PRA. However, many components of the CIR-Agent are abstracted slightly differently. The knowledge base and problem solver are placed within the Decision Maker and the Repository. The interaction and communication components are moved into the State

Monitor, the Coordinator and the Negotiator. The Executor can be seen as a new addition in which the CIR Agent now has the ability to interact with the physical world. It is important to note that these components remain essentially intact, but due to the nature of the dual layer system, they may be moved into the non-agent Action Layer or split up between the layers.

3.5 Communication

The role of communication among mobile robots is one of the most important issues in multi-agent robot systems design. Communication is required to ensure cooperation between robots. Each robot's actions depend critically on knowledge that is accessible only from another robot. The communication structure of a group determines the possible modes of inter-agent interaction. These modes of interaction are sometimes classified into interaction via environment, interaction via sensing and interaction via communication [3]. In interaction via environment, the surroundings are used as a shared medium (or memory) for storing information so that it can be interpreted by other cooperating entities. This method is known as 'cooperation without communication' or 'stigmergy'. Like an ant pheromone trail, a stigmergic signal can be picked up by any other entity at any time. This is accomplished by storing the information in a stable medium. Interaction via sensing refers to the local interactions that occur between agents as a result of one agent sensing another, but without an explicit communication. On the other hand, interaction via communication involves explicit communication with other agents by either directed or broadcast intentional messages. In the proposed architecture, interaction via communication is adopted. As shown in Figure 3.3, there are two types of

communication in the system: inter-layer communication and inter-agent communication.

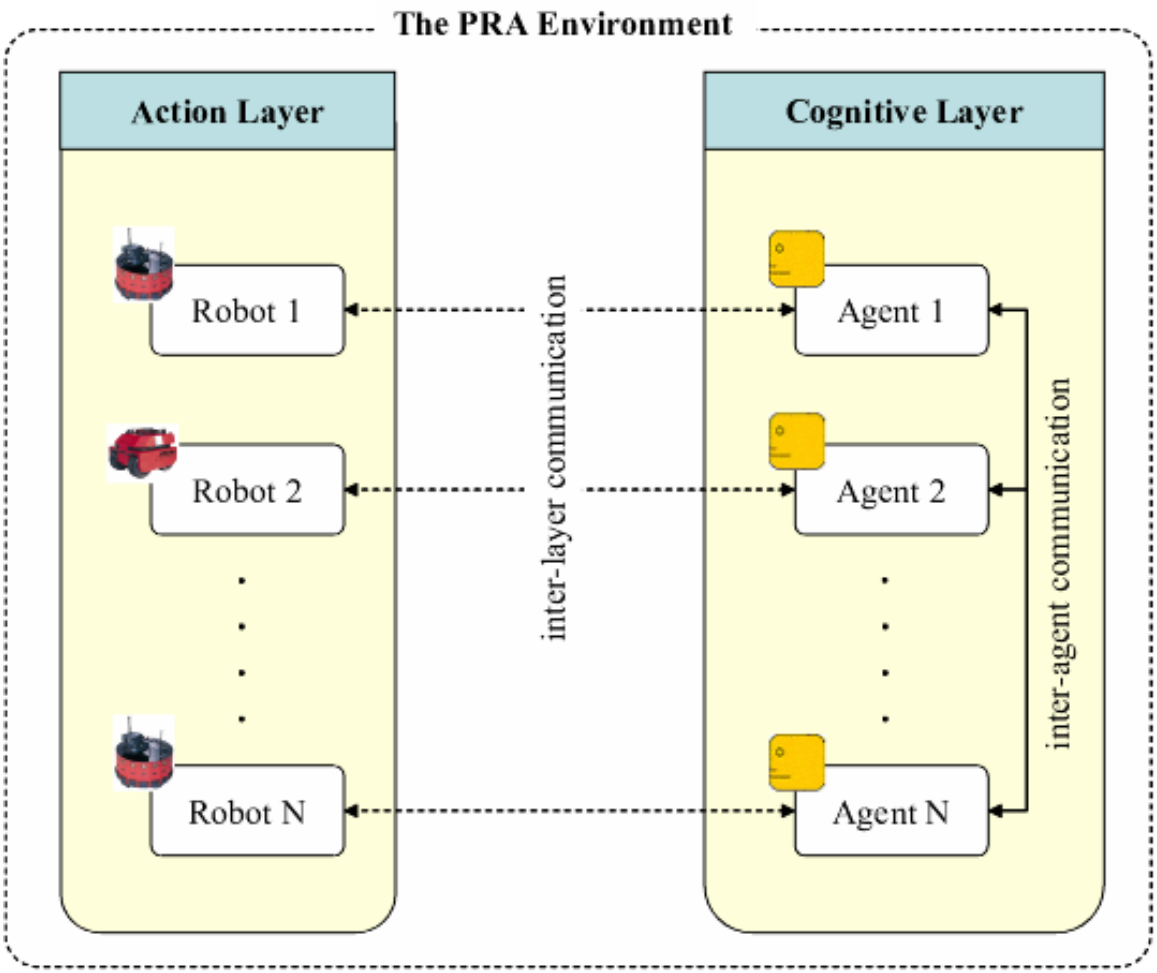

Figure 3.3: Communication Links

3.5.1 Inter-Agent (Inter-Robot)

Communication is essential for multiple agents to work together. Agents must be able to communicate between one another to locate other agents and interact with them. To accomplish this, two main components are required: a communication protocol for inter-agent communication and a network protocol in which to send the messages. Inter-Agent

(or Inter-Robot) communication is how different PRAs correspond. The most important aspect of inter-agent communication is that it should be limited to as small and as few messages as possible. If there is too much data or there are too many messages being sent, the load they create can severely slow down a system. This is precisely where network constraints define the total amount of communication that can occur. Inter-agent communication should be concise yet meaningful. This helps to reduce the amount of network traffic.

The communication between robots is performed between the Negotiator elements in the different PRAs. An agent communication language (ACL) can be used to facilitate messaging. ACLs allow for large groups of robots to be able to communicate with each other. The total number of PRAs that could be used at once is limited only by the selected platform.

3.5.2 Inter-Layer (Intra-Robot)

The communication between layers is vitally important if this architecture is to be used for real time applications. The maximum speed of the robots and hence the ability for the PRAs to function as real-time robots will be restricted by a bottleneck of either communication or processing speed. The processing speed is determined by the speed of the processors and the algorithms themselves. There are only two locations where a communication bottleneck can occur. The first is between PRAs, in inter-agent communication, in which network speed and traffic are the main culprits for slower speed. The second location is within a PRA between the two layers. As the dual layer architecture is relatively new, this type of communications has not been examined before.

The volume of data transferred between layers is only bounded by the specifics of the robot; however, by limiting the amount of data being transmitted, the abstraction and separation between the two layers becomes apparent. For example, there is often no need for the Cognitive Layer to receive an image from a camera for processing. Data intensive processing can be handled and large volumes of data passed between tasks in the Action Layer, with only the relevant results sent to the higher level.

3.5.3 Throughput

When using the architecture describe in this thesis, extremely complicated PRAs can be created and they should be able to react quickly. Most importantly, the worst-case completion time at which they can react is essential for any real-time applications. The throughput and latency of inter-layer communication is examined in depth in section 5.2.

4 Implementation

This chapter explores some of the aspects dealing with the implementation of the architecture described in the previous chapters. An example of a fully functional version of the architecture can be seen in section 5.1. This chapter's contents are all original concepts presented by this thesis.

When implementing the proposed architecture on mobile robots, only a few components are required for it to function. Firstly, the robots must have most of the capabilities of a notebook computer. This includes a hard drive, a quick processor and a network card of some sort, (a wireless connection is recommended) to be able to contact other robots. The robot also must be able to run Java for the Cognitive Layer implementation. Java is not required, but highly recommended because JADE was designed specifically to work within Java [17]. If another agent oriented language is used, it must be able to run on the robots. If a robot has the basic capabilities of a notebook computer than it should already be able to run the Java programming language. Apart from the necessary components, this architecture can be used on almost any robot.

4.1 Action Layer

This section describes some of the issues dealing with the implementation of the elements within the Action Layer. It is important to note that although they are separate elements, they are not necessarily separate programs. They could be different objects within a multi-threaded program to facilitate faster communication within the layer. Unlike the

stricter separation between the two main layers, these elements are more of a guideline than a boundary.

4.1.1 Executor

Implementation of the Executor will be different on each distinct type of robot. The Executor should be able to run both single and multiple tasks simultaneously or concurrently based on the requirements and restrictions of the system.

The types of tasks that will be required are also varied:

- Initializers which are run once, at start-up, to initialize some aspect of the robot, e.g. resetting the pan and tilt of a camera;

- Actions or steps which are run as part of a state machine, such as rotating to face a target;

- Alerts which are run and sit idle until a specific condition occurs, e.g. waiting for target movement; and

- Reflexes which are required to react quickly to a situation without consulting the Cognitive Layer, such as crash avoidance.

Multiple tasks should be able to be run at the same time, allowing one to set up a crash avoidance task, e.g. sonar detection that activates when the robot is quickly approaching a

wall while allowing the robot to perform its other duties. It is also possible to execute complex recognition tasks using sensor data, or sensor fusion. The results of these tasks are sent to the Cognitive Layer. Although tasks and reactions can perform low-level functions autonomously, they are all directed by the higher-level Cognitive Layer.

4.1.2 Repository of Tasks

As the Repository is simply a collection of available tasks, implementation can be as simple as the tasks available in the code or as complicated as a collection of all tasks that could be executed, perhaps stored in a hash table or dynamically linked library. Even though the Repository may not always be a large component, it still is an important one.

4.1.3 State Monitor

In the Action Layer, the Communicator is the element that relays information between the two layers. It must be able to both send and receive data from the Cognitive Layer. The Communicator must be able to inform the Executor of the tasks that need to be run, variables and constraints for those tasks, and how and when to run them. It also might need to assist in finding the index or location of tasks found in the Library, or even update or alter the stored tasks. As well, the Communicator must also send the results of these tasks (if any) and updates on a list of internal variables as requested by the Cognitive Layer.

4.2 Cognitive Layer

Since two of these key elements of the Cognitive are already included within an agent (by definition,) the Decision Maker and the Negotiator, an agent oriented solution is a good solution for the problem. This section details some aspects that should be considered when implementing the Cognitive Layer.

The Cognitive Layer does not necessarily exist on the robot in a PRA. It could exist on a separate machine, or in a combination of machines. What is important is that there is a clearly separate section for each PRA. This ensures the autonomy of each individual PRA.

4.2.1 Decision Maker

The way in which the Decision Maker is designed and implemented is determined on a per system basis. Possible examples include a collection of small programs, a collection of threads, or a collection of simple software agents. Each system will have a different requirement and thus the resulting Decision Maker will be vastly different, except for the methods in which it interacts with the other elements.

4.2.2 Negotiator

Assuming an agent oriented design is already being used, this element is already included in every agent based in the definition of an agent. The Negotiator must have the ability to communicate with other robots and to understand what is being communicated. The

Negotiator does not only rely on messages, but can also perform any bidding or handshaking required for decisions and consensus building.

4.2.3 Coordinator

Typically when implementing the Coordinator should be able to start, stop and pause tasks in the Action Layer, based on the whim of the Decision Maker. This includes reflexes if required, as a robot may need to actively suppress a reflex to accomplish a task. For example, if a robot needs to pick up an item, it may need to suppress its automated obstacle avoidance reflex. It also may need to be able to receive variable updates from the Action Layer's State Monitor and inform the Decision Maker of these new conditions.

4.3 Communications

The communications between the robots and agents are very important and this section details the crucial aspects of their implementation.

4.3.1 Inter-Layer (Intra-Robot)

As the inter-layer communication has not previously been examined, a need existed to create a system with which the two layers can interface. The method with which a microprocessor interprets assembly language is chosen as model for the means of communication between layers. This choice is made because of (1) the simplicity of the manner in which the communications are processed and (2) it reinforced the idea that the communications should be minimized. Microprocessors have a series of registers for

both input and output. These registers were replaced with monitored variables. As this is now implemented in a software domain, there is no limit on the number of variables monitored. There is, however, a cost associated with monitoring a variable; as more are monitored, the slower the system and the slower the transmission. Secondly, tasks can be run on microprocessors and this same system is used for inter-layer communication. The difference is that not only can these tasks be run, but multiple versions of both of the same task and others can be run in parallel, again due to less restrictive software domain. This allows a reflex, such as an anti-collision detection task, to be run while the robot accomplishes its goals.

The Coordinator element enables communication between layers and resides in the Cognitive Layer. This element receives commands from the Decision Maker and creates the messages that will be passed to the Action Layer's State Monitor. The main type of communication is commands. With a command, tasks are initiated and halted and specific Action Layer variables are requested for higher level processing.

The most common types of commands issued are:

- Initiate a task: tasks can be called by name with parameters
- Alter a task: a currently running task's parameters are changed
- Cancel a task: this command stops a task from running

- Watch a variable: this command asks for updates whenever the value of that variable is changed

- Stop watching a variable: this cancels the watch command

- Alter variable: this allows the Cognitive Layer to set a variable that can be used by a task

4.3.2 Message Types

The speed of communication is vital to knowing the delivery time of the system if this architecture is to allow for real-time applications. Before calculating the latency and throughput of the communication, the message requirements had to be determined. This section will briefly discuss the three main types of messages that could be sent between the layers. To design a better system, knowledge of the most common forms of communication is required. In this case, three distinct types of messages can be passed between the layers: commands, variable updates and large data updates. They are explained in detail below.

4.3.2.1 Commands

A command message is a very simple communication. Typically, a command only needs to be initiated, occasionally with a variable or two. This means that the message will be very compact. Typical commands should look like the following:

- *Forward 30* – move 30 units forwards

- *Pan 100* – pan the camera to 100 degrees

- *Halt*

Commands will typically be sent one or two at a time. These commands will be sent from the Cognitive Layer's Coordinator to the Action Layer's State Monitor.

4.3.2.2 Variable Updates

Variable updates are sent from the Action Layer to the Cognitive Layer. These can occur in a few different forms. Firstly, they can be a single variable being updated whenever that variable is altered. Secondly, they can be a timed update of all or most of the variables, whether they have changed or not. Thirdly, they can be a complete update of all of the variables whenever any or a number have been altered. Finally, they can be a combination of all of these techniques. The method utilized is dependant on what information is required in the Cognitive Layer. If only a few updated variables are required and the single variable method is used, then these messages look similar to a command message.

Many systems require a large number of variables of differing types to be updated on a regular basis. These messages can be implemented in a stub of some sort that would encapsulate the data into a hash table or a similar structure. Nevertheless, these messages, when broken down, can be considered as a number of command messages strung together.

4.3.2.3 Large Data Updates

The variables being updated examined so far are in the form of integers and occasionally real number and strings. In a large data update message, large data objects such as pictures can be sent. A well designed system should minimize the number of these messages, as they take a long time to process. However, there are certain cases where this type of information is required.

When multiple robots are mapping a room full of objects, at some point they require each other's maps to find each other. This can be accomplished, assuming the maps are fairly intricate, by sending a large amount of data. The structure of these messages should seem like a stream of bits that would be unrecognizable without prior knowledge of what information is contained within them. Essentially, without knowledge of the data structure, they would seem useless. It is preferred to send only the relevant data to minimize this type of communication. For example, instead of sending a raster image of an internal map, a vector representation could be used to substantially reduce the amount of data transferred.

5 Experimentation

In this chapter, two experiments are presented. This first, the application experiment, focuses on the implementation of the architecture. The second examines various transmission protocols that can be used for inter-layer communication.

5.1 Application Experiment

The first experiment implemented to test the validity of the architecture is the Best-View Demonstration in which one or more robots try to encircle a target. The overall goal of this project is to test the feasibility of using the architecture, and to do so, a task is required that can be accomplished using a different number of robots. The chosen activity is to have the robots encircle an object. This section deals with both the implementation issues and the results associated with the experimentation.

The robots' goal in the experiment is to encircle a target. In this case, the term 'encircle' is meant a target object is completely surrounded. As shown in Figure 5.1, a coloured can on top of a basketball is chosen as the target. The basketball makes the target easy to spot from afar and the can allows for a good estimate of distance to be found when up close. The can has four colours evenly placed along its surface. These colours are used by the robots to determine from which angle they are viewing the target. In this way the robots have a common form of perception of the target and can discuss which one should move to where to surround it. The method with which this is achieved is dependant on the number of robots participating. If there is only one robot, it should find the front of the target. The front of the target is an arbitrary point, in this case, the line between the

yellow and blue colours is chosen. If there are two robots then they both should be 180 degrees apart from each other. With three robots, they should all be 120 degrees apart. Figure 5.2 shows the relative positions of the robots with respect to the target for each of these situations. The arrow on the robot represents the front of the robot, and the red triangles represent the camera view.

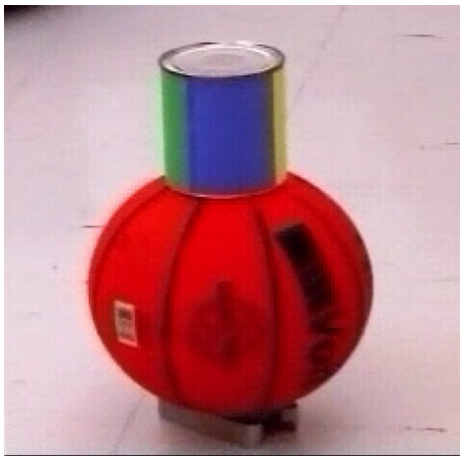

Figure 5.1: Target

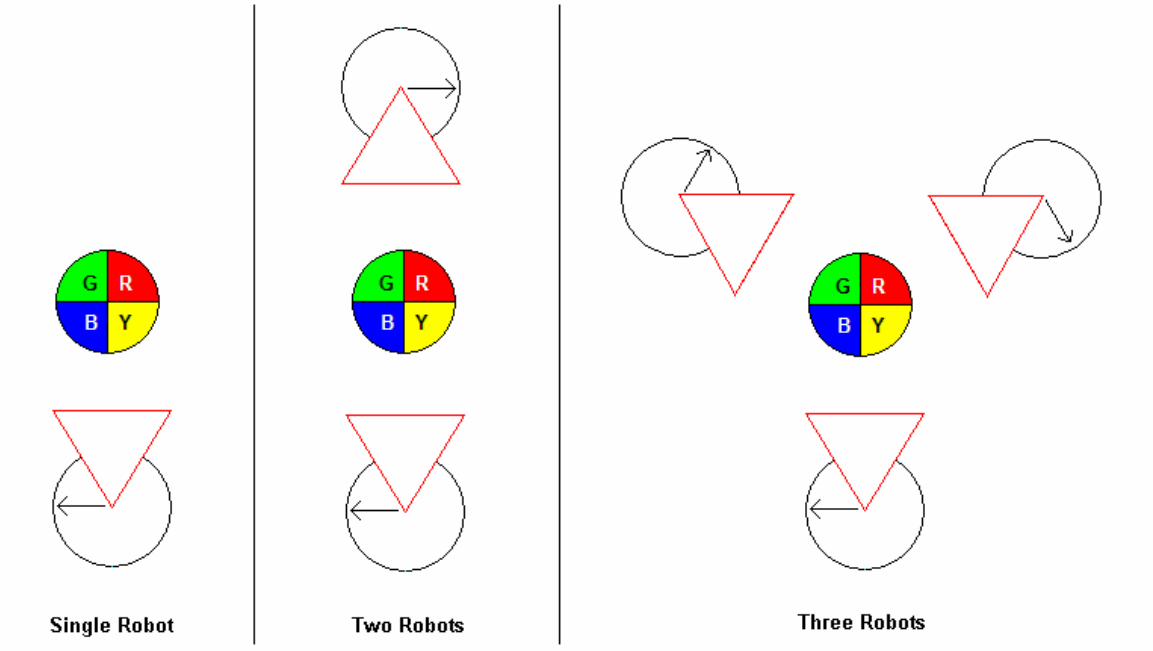

Figure 5.2: Goal Positions

One important aspect of this experiment is that it must be possible to add or remove robots from the system. If one robot is attempting to look at the front of the target when another joins, the system should adjust and both robots should now attempt to place themselves at 180 degrees apart. Similarly, if there are three robots encircling the target, when one is removed, the remaining two should compensate.

The tests performed to demonstrate the full functionality of the system are dependant on the number of robots present. The most important tests are "does the system function correctly and encircle the target with one, two or three robots?" At the time of writing the maximum number is three due to limited availability of robots. The key issues in this experiment are communication and architecture. The movements of the robots are not reliant on the architecture; it is the coordination that is important. To this end, the tests performed are basic functionality testing, systems of one to three robots and the possibility and coordination of adding or removing a robot from the system.

The robots used for this experiment were the Magellan by the iRobot Corporation, see Figure 5.3 and [16]. There ware three identical robots available, which made them ideal subjects for this experiment.

Figure 5.3: Magellan Pro

The Magellan Pro robot has all of the required components to run the architecture and the tests for this experiment. All of the robots used in this experiment have an installed rotating and panning camera, which can be seen on top of the robot in Figure 5.3. The robots are equipped with sonar and other sensors, however the camera is the only sensor used in the experiment.

5.1.1 Layer Implementation

The main purpose of this thesis is to present a new architecture for multi-robotic systems that takes advantage of the benefits of a multi-agent paradigm. This section will discuss the implementation of this system.

5.1.1.1 Agent Model

In order to facilitate the implementation of the proposed design, the CIR Agent model described in section 2.2.4 is used. This model can be used to develop an agent which is autonomous, has a method of coordination (for both interaction and communication), has the ability to cooperate with other agents and it is adaptable to different situations. The PRAs are all based on this model.

5.1.1.2 Cognitive Layer

Ideally, if a large number of known Action Layer tasks exist, then it is possible to write the 'brain' behind a robot without having to re-write any of the lower level functionality. This simplified development process is supported by the Repository element. This intentionality limits the abilities of the Cognitive Layer to help keep the abstraction between high and low level tasks. To enable the Cognitive Layer to communicate with other robots yet still be independent, autonomous and cooperative, an agent oriented solution is the natural course.

When conducting experiments using this architecture, the Java Agent DEvelopment Framework (JADE) [1] is used. Any multi-agent environment could be used, but JADE is chosen because it is FIPA compliant built for the Java environment (see [17]), which supports portability. This decision is an important one, as it has ramifications throughout the design of the entire system.

To create the Cognitive Layer, a number of software agents are used. Each robot can have as many agents as is required acting collectively as the Cognitive Layer of a particular robot. This collection of software agents make up the PRA. Currently, only two agents per robot are required. The first software agent is the main 'brain' of the Cognitive Layer. It encompasses both the Decision Maker and the Negotiator. A second software agent acts as the Coordinator. This agent acts as the intermediary between the Decision Maker and the Action Layer of the robot. The reasoning behind this two software agent implementation is simply that every robot requires a Coordinator; thus incorporating it into the main 'brain' agent seemed to be complicating the matter. In this

manner, every robot can have the exact same Coordinator agent, no matter how the rest of the Cognitive Layer is designed. This agent is discussed in detail in below in section 5.1.2.1.

The agent that encompasses both the Decision Maker and the Negotiator is again just a convenience. It is possible to separate them, but because most projects require completely new negotiators, having the two together inside a software agent seemed the sensible answer. This again is completely up to the designers of the multi-robot system.

Inter-Agent communication methods are built into JADE and hence they do not require their own agents. There are also a few necessary software agents required by JADE but they are only used to enable the agent oriented environment to function.

Each robot may have completely different software agents or they may all be the same. This depends entirely on the project. For the most part, the Coordinator should be fairly independent of the project. As mentioned previously, all of these agents, including the required infrastructure can be placed on one or multiple servers and not actually reside on the robots themselves. This enables robots with limited processing power to still take advantage of the proposed multi-level architecture. This architecture is designed to be independent of any specific goal, so that it can be tailored to satisfy each project's requirements.

5.1.1.3 Action Layer

The Action Layer on the robots was primarily designed by Ben Miners, co-author of the architecture paper [12]. He has allowed the inclusion of this description on the implementation of the Action Layer in the robots:

> The Action Layer serves two important purposes; to abstract variations in physical hardware from the Cognitive Layer, and to carry out local time-critical tasks. Abstracting the hardware in this layer is an approach to allow the same action logic to be carried out on several different hardware platforms. Latency is minimized using an event-driven approach to ensure appropriate tasks or reactions are carried out for each external stimulus.
>
> As illustrated in Figure 5.4, the Action Layer communicates with physical robot hardware through an abstraction interface. This interface maps each received sensor value to a specific location and orientation, and translates generic motion control commands to hardware specific values. All action logic is defined using a set of simple concurrent tasks. Each of these tasks can be in one of two states as decided by the Cognitive Layer; passive or active. Active tasks can carry out their actions when triggered, while passive tasks do nothing until activated from the Cognitive Layer. Activation and deactivation of tasks is the primary method of control from the Cognitive Layer.

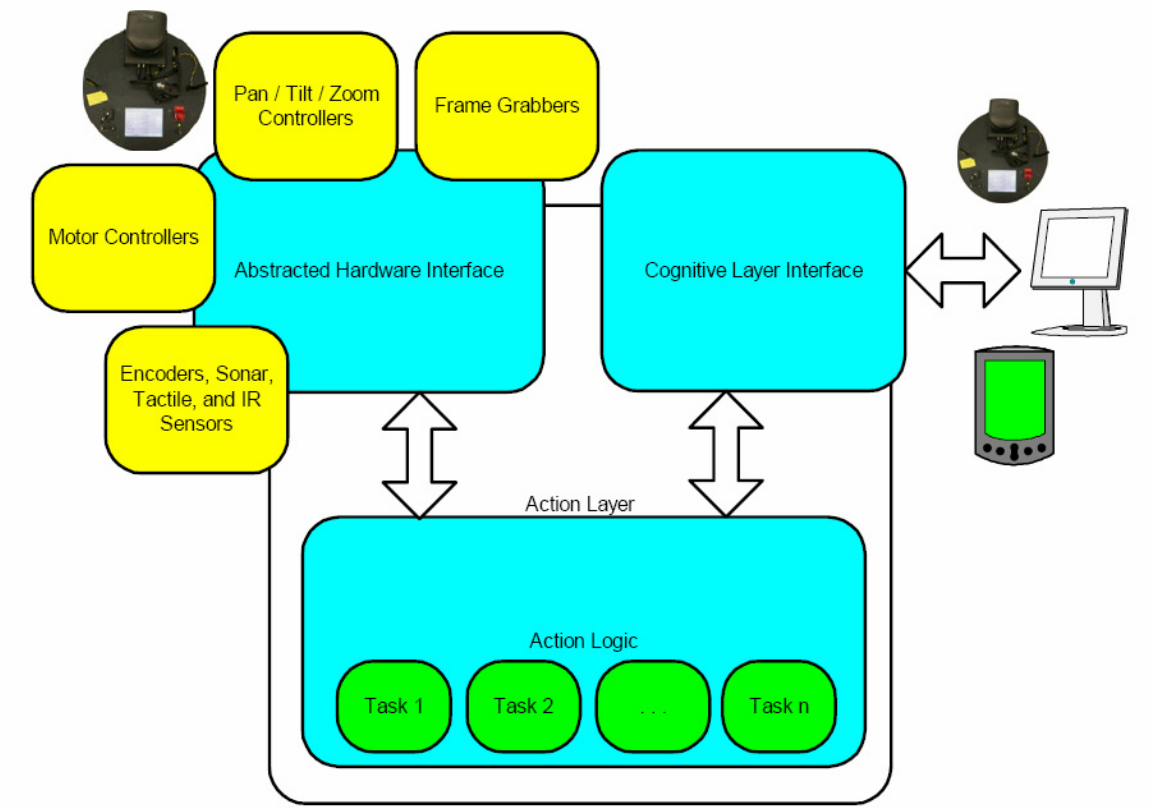

Figure 5.4: Action Layer

A specific precondition based on external stimuli is defined for each task. Examples of these conditions include the arrival of new a video frame, a sonar measurement, or a change in robot position. Including these preconditions outside task logic helps to keep internal logic simple and allows a single task to easily respond to different triggers or external stimuli. As soon as a task's precondition is met, the task is executed. During execution, tasks can process sensor data, control robot movement and sensor parameters in addition to exposing high-level task state as feedback to the Cognitive Layer. Processing sensor data locally in the Action Layer eliminates unnecessary communication of low-level data while ensuring relevant high-level information is available to the Cognitive Layer.

5.1.2 Software Agents

A basic state machine with a small amount of reasoning is all that is necessary to complete the exercise. For the purposes of this experiment, all of the PRAs are identical. Each of which consists of three software agents: the Coordinator agent; the single robot

agent and the multiple robots agent. In this section, each of these agents is discussed in detail.

5.1.2.1 Coordinator Agent

The Coordinator software agent performs a few key functions. It acts as an intermediary between the agents representing the cognitive level and the low level functions of the Action Layer. It accepts messages based on a specified format and performs a variety of operations. The most important function is to send a command to initiate a task in the Action Layer. This command is sent to the Action Layer through an open port, a computer connection location through which network data can be transmitted, to which the Action Layer is sensitive. This agent can request that a variable be monitored and it can also cancel a task. Meanwhile, whenever a variable being monitored is altered, the Action Layer sends an update through a Java stub. The Coordinator sees this and sends an update to the subscribed software agent. Only the variables to which an agent is subscribed are sent through the Java stub. This means that even though a task may expose a variety of variables, only the ones that are needed in the Cognitive Layer are actually sent. Thus, the Repository in the Action Layer tasks can be used in future experiments. The Coordinator agent needs to be quick to allow for seamless operation of the robot, however, this agent should only need to be created once and then it can be used on all future projects. The speed of the Coordinator is examined further in the next experiment, section 5.2.

5.1.2.2 Single Robot Agent

This software agent was created to determine if the communication between the Cognitive Layer and the Action Layer was functioning. It was also created to prove the feasibility of the multi-robot system. As it was designed for a single robot, only the Decision Maker element was included. This state machine of the agent can be seen in Figure 5.5.

Each bubble represents the current task being run and the arrows represent the transition in state based on a monitored variable. The first task is that of finding the target. The robot rotates until the target is located. The second task requires the robot to approach and then centre in on the target. Once completed, the robot prepares for pivoting by panning the camera 90 degrees while simultaneously backing off from the target. Finally, the robot calculates the shortest route to get to the 0 degree mark, the front of the target and pivots accordingly. Once there, the robot remains on alert for any new commanders or changes to the target. If at any time the robot loses site of the target it will return to the first state and start searching for the target anew.

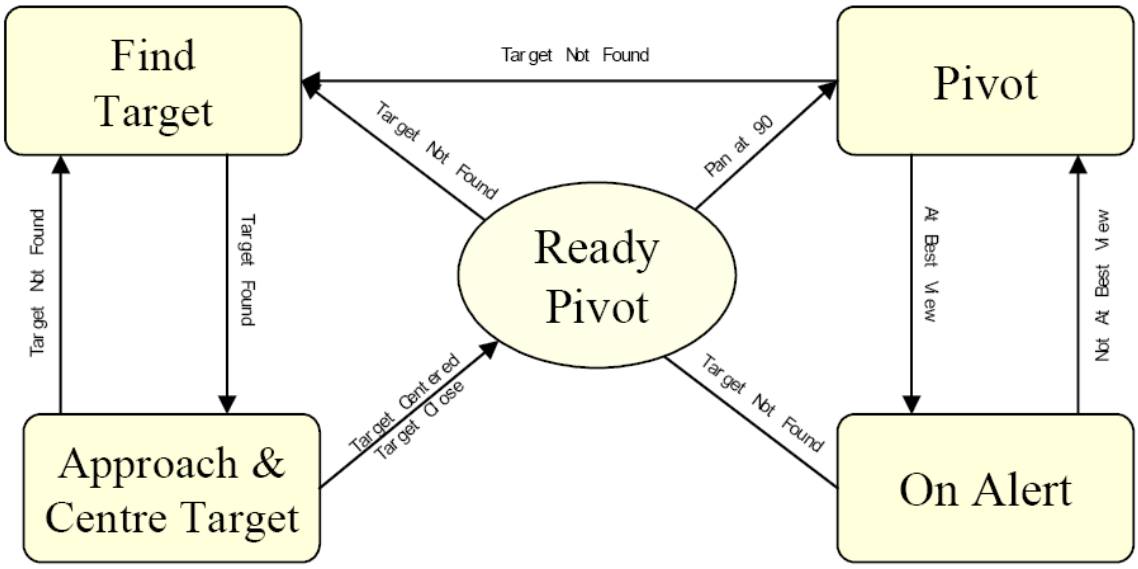

Figure 5.5: Single Robot State Machine

5.1.2.3 Multiple Robots Agent

This multi-robot agent encompasses both the Decision Maker element and the Negotiator element. This software agent is used on multiple robots communicate not only with the Action Layer through the Coordinator Agent, but also with other software agents on other robots. This agent builds upon the single robot version, as well as coordinating its actions with the other software agents running in the Cognitive Layers of the other robots. The state machine of this agent can be seen in Figure 5.6. This state machine is similar to that of the single robot version. The differences lie within the new states located between the old states of the previous implementation. New steps that are designed for negotiation with other PRAs were added. Each of these new steps is a Cognitive Layer task and not an Action Layer task. These steps include: greeting the other agents; reporting to the other agent that this PRA is ready to pivot; receiving and sending the cost to travel to the different location around the can and telling the other

robots that an error has occurred and the whole process needs to be restarted. It is important to note that the Action Layer tasks did not have to be altered for this new agent or for the more complicated goal. For both the signal and multiple robot implementations, the Action Layer performs the exact same tasks.

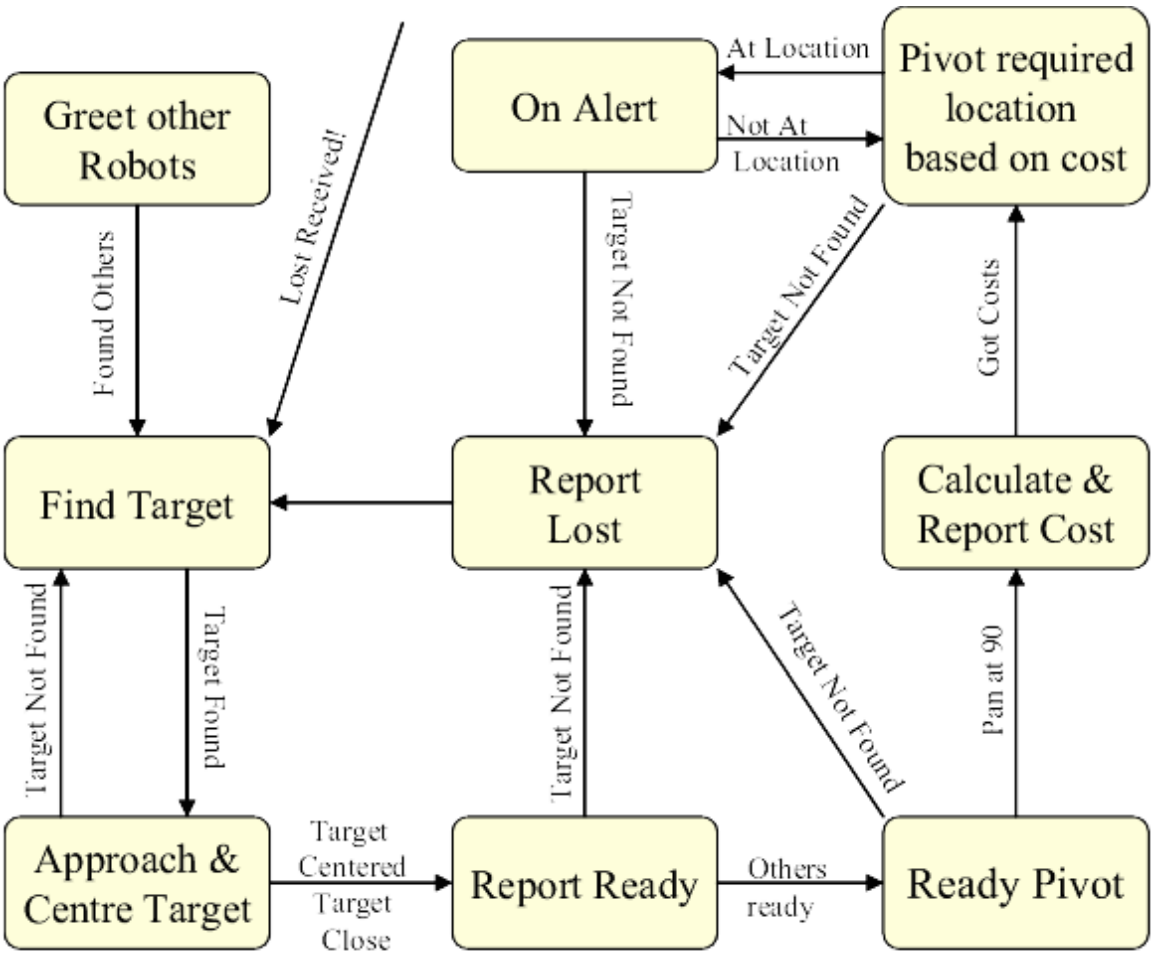

Figure 5.6: Multiple Robots State Machine

5.1.3 Results

After more than 20 tests on each of the three robots, the single robot agent performed as expected. Each robot would find the target, approach it and pivot to the correct location. One problem noted with the experiment was in the Action Layer where there were

complications with colour detection under different lighting conditions. Because of this, the PRA's state machine was observed to function correctly, even when conditions were not optimal. The communication between the layers was fully functional as evidenced by the fact that the robots reacted quickly when the target was removed or replaced.

The multi-robot experiment again responded to the same lighting problems as the single robot scenario; nevertheless, the robots were able to function well as a team. A new challenge was noted as occasionally a PRA saw the red colour of another robot and mistook it for an extension of the basketball. This does not seem to adversely affect the final outcome of the program as this only occurred at far distances. Similarly to in the single robot implementation, when a robot lost sight of the target, it reacted quickly and reset. Furthermore, the other robots also reacted and reset shortly afterwards having received the error message from the lost robot. Typically, the robots all locate the target and encircle it while staying in sync with one another. Table 5.1 shows the results obtained during this experiment and Figure 5.7 is a photograph of the robots in action. More than 25 tests were performed for each of the scenarios listed below with no errors occurring that could be attributed to the architecture or the communications.

Table 5.1: Best-View Results

Scenario	Number of Robots	Results
Encircle Target	1 (find front of target only)	The robots performed as expected. All three robots were individually tested.
	2	Any two robots performed as expected.
	3	The robots performed as expected.
Remove Robot	3 down to 2	The robots performed as expected. Choice of robots did not affect the results.
	2 down to 1	The robots performed as expected. Choice of robots did not affect the results.
Add Robot	1 to 2	The robots performed as expected. Choice of robots did not affect the results.
	2 to 3	The robots performed as expected. Choice of robots did not affect the results.

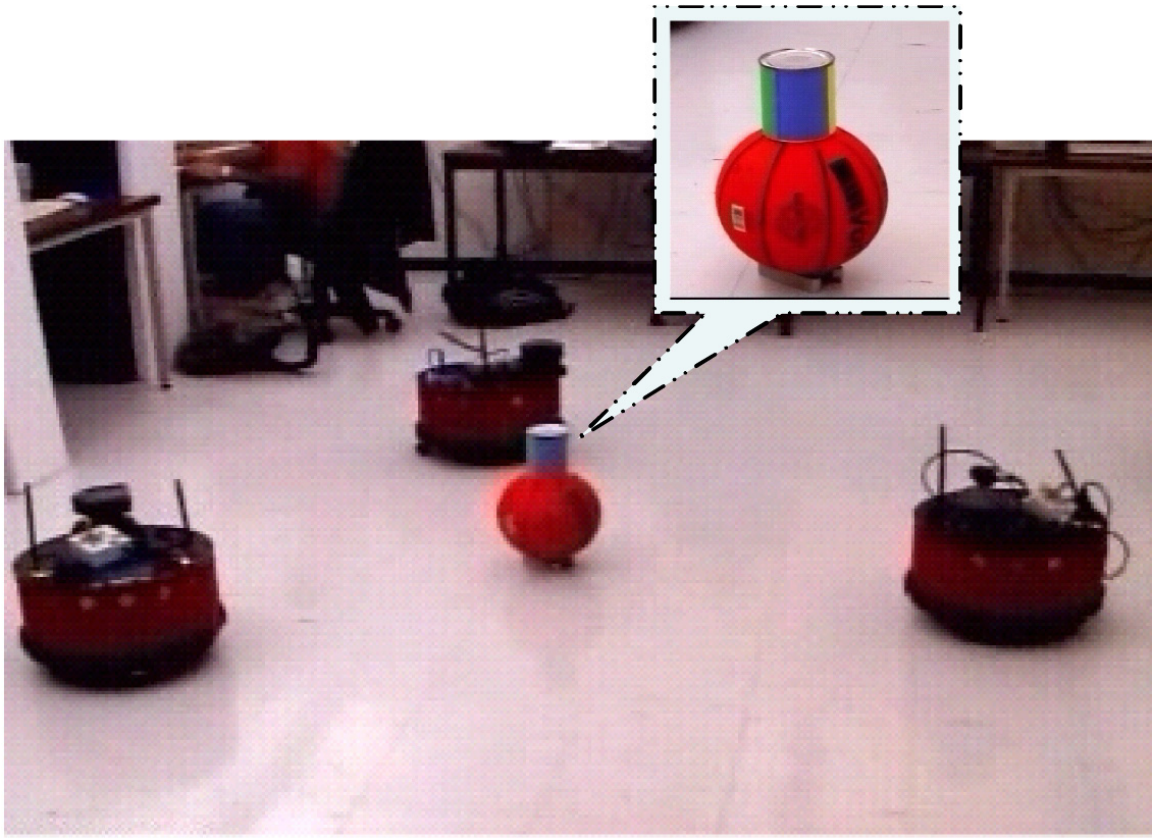

Figure 5.7: Robots in Action

The communication between agents functioned flawlessly and the robots performed as expected. All communications between robots was monitored to ensure that the correct messages were being sent and received. Failures arose due to errors in detection of the distance from the target and a robot hitting a piece of furniture or wall (currently there is no detection or avoidance for other objects). When a failure occurred, the robots reset as the state machine dictated.

5.1.4 Conclusions

This architecture defines clear boundaries between the processes that occur within a robot and allows multiple robots with different specifications to communicate with each other and perform meaningful tasks. As was evident by the experiment presented, this architecture works well and solved the encircling of an object problem sufficiently. The architecture functions as expect in both a single and a multi-robot scenario.

5.2 Real-Time Feasibility Experiment

To test the real-time performance of this architecture, the intra-robot (between layers) and inter-robot (between robots) communication time must be examined. In this experiment, intra-robot communication is explored with a variety of message types, methods and protocols.

The primary goal of this experiment is to test the communication delays that occur between the Coordinator in the Cognitive Layer and the State Monitor in the Action Layer. Thus this experiment should answer the following: which method of

communication has the lowest latency, has the highest throughput; is the easiest to implement; is exportable, not restricted to a single operating system; and is expandable, can it be used if the layers are on different systems? To do this, a few demanding tests were created. This section describes these tests, the nature of the test system and the results.

The tests needed to use different types of messages, different communication styles and a different number of transmissions. Conclusions will be based by weighing both the positive and negative aspects of each type of communication.

Each test has two main aspects. Randomized messages, which are already in memory, are sent from the Cognitive Layer to the Action Layer. Once fully received, the message is returned to the Cognitive Layer using the same communication method to ensure that both directions of communication are tested. This test is then repeated a varying amount of times. The total time for this whole process to occur is then recorded and analysed.

The system used to perform the testing is contained on a single robot. This type of robot was chosen as it is the same robot that is currently used in the labs for multi-robot experiments, including the architecture experiment described in chapter 4. Thus, all data recorded would be relevant for determining delays when these robots are used for real-time experimentation. The software used to test the system was a simplified version of both of the Action and Cognitive Layers also used in the previous experiment. The following subsections describe the robots and the simplified layers.

The robot used for this experiment was the Magellan by the iRobot Corporation, see Figure 5.3 and [16]. Only a single robot is required, so one of the robots used in the previous experiment, section 5.1, is used for this experiment. All tests are performed on the same robot, to ensure high quality data. For the purposes of this experiment a connection to other robots is not required.

5.2.1 Layers

As this experiment is designed to test only the throughput of the different communication methods, only the basic communication shell of the layers is required. A simple test application is used in place of the Action Layer and a basic Java program was required for the Cognitive Layer. The code used for this experiment can be found in Appendix B.

The Cognitive Layer was created in Java to attempt to be similar to the Cognitive Layers in fully functioning robotic systems. In this case, all of the extra functionality in the layer was removed leaving only a random message generator as well as sending and receiving routines. By doing so, it is possible to time the period it takes to send and receive the messages.

The Action Layer was programmed in C. Usually, the Action Layer spends a lot of time monitoring the robot and controlling its movements, however, in this case, all of the superfluous parts have been removed to accurately measure only the communication speed and delays. To simulate the Action Layer, very simple programs were used. These programs wait for a packet or file and when one is detected, it receives the full packet. Once received, it sends the packet or file back along a similar communication channel

that it was received in. Again, these small programs are designed to only have the functionality required to perform these communication operations.

5.2.2 Communication

If the type of messages described in section 4.3.2, which need to be sent between the concurrently running processes in the action and Cognitive Layers are to be tested, then the method for sending the messages is of great importance. This section describes the most common types of communication methods and protocols over which these types of messages can be sent and received and describes how there were implemented. Each method of communication to be tested has different characteristics and hence, needs a slightly different implementation.

5.2.2.1 User Datagram Protocol

The following definition of the User Datagram Protocol (UDP) was taken from Wikipedia [28]:

> The User Datagram Protocol (UDP) is a minimal message-oriented transport layer protocol that is currently documented in IETF RFC 768.
>
> In the TCP/IP model, UDP provides a very simple interface between a network layer below and an application layer above. UDP provides no guarantees for message delivery and a UDP sender retains no state on UDP messages once sent onto the network. UDP adds only application multiplexing and data check-summing on top of an IP datagram.

Lacking reliability, UDP applications must generally be willing to accept some loss, errors or duplication. Most often, UDP applications do not require reliability mechanisms and may even be hindered by them.

UDP packets were implemented by opening two ports. The Action Layer received packets through one port, and once fully received, sent a new packet, with the same information as the one received, back to the Cognitive Layer using a secondary port. The total time to perform this operation was recorded for different sizes and numbers of messages sent.

5.2.2.2 Transmission Control Protocol

The following definition of the Transmission Control Protocol (TCP) was taken from Wikipedia [28]:

> Transmission Control Protocol (TCP) is a connection-oriented, reliable delivery byte-stream transport layer communication protocol, currently documented in IETF RFC 793.
>
> In the Internet protocol suite, TCP is the intermediate layer between the Internet Protocol below it, and an application above it. Applications most often need reliable pipe-like connections to each other, whereas the Internet Protocol does not provide such streams, but rather only unreliable packets.
>
> TCP checks to make sure that no packets are lost by giving each byte a sequence number, which is also used to make sure that the data is delivered to the entity at the other end in the correct order. The TCP module at the far end sends back an acknowledgement for bytes which have been successfully received; a timer at the sending TCP will cause a timeout if an acknowledgement is not received within a reasonable round trip time, and the (presumably lost) data will then be re-transmitted. The TCP checks that no bytes are damaged by using

> a checksum; one is computed at the sender for each block of data before it is sent, and checked at the receiver.

To summarize, unlike UDP, there is a large amount of error checking and the order of the messages is guaranteed. This should add a bit of processing overhead. As TCP uses a handshake method, these handshakes were performed once at the start of each test. Overhead should be significantly reduced by allowing the connection to remain open. Even though it is possible to send and receive though the same port, two ports were used to make the operation similar to the other tests. Again, similar to the UDP method, only once full messages had been received in the Action Layer did they get sent back to the Cognitive Layer. The total time required to perform the sending and receiving operations, while varying the number and types of messages and the time spent on the handshake was recorded.

5.2.2.3 File Sharing

For large amounts of data, files are a viable option. By writing and reading from a file, messages can be passed. While this may not be the fastest method of message passing, it is very reliable. As well, using a file does not necessarily mean using a hard drive. To increase transfer speed, a random access memory (RAM) drive or memory-mapped files could be substituted. Files are of interest because they can contain huge amounts of data. As well, data can be stored in subsections of a file requiring only some of file to be altered. More than one program can read from a file simultaneously and multiple files can be worked on concurrently.

This communication method is dependant on the speed of the drive being used. Newer drives are faster and larger than older ones. The type of interface with the drive is important (i.e. Small Computer System Interface (SCSI), Parallel Advanced Technology Attachment (PATA) or Serial Advanced Technology Attachment (SATA)). As well, the type of partition also plays a significant role. All of these factors greatly affect performance. In the case of the tests performed, the interface, partition type, age and model of the drive were determined by what was present in the robot.

The implementation for file communication was realized differently. The Cognitive Layer writes to a file, when the file is complete, the Action Layer reads in the file and then writes the contents to a new file. This new file is then fully read in by the Cognitive Layer. The Action Layer only writes to the new file once the old file has been completely read to ensure that this has the same restrictions as the UDP and TCP communication methods. Files are deleted after they are read. The total time required to write the message from the Cognitive Layer to the time received, using different sizes and number of messages, was recorded.

5.2.2.4 Inter-Process Communication

Inter-process communication (IPC) messaging was designed to send messages between concurrently running process. This may seem to be a perfect method of communication but it has many limitations. Firstly, on most operating systems, the messages' maximum size is very restrictive and typically very small. Secondly, the means used to invoke this communication method differ between operating systems. An additional challenge with IPC messaging is that there is no native Java library for it. There are some libraries that

exist for each type of operating system, such as JTux [19]. The methods used in C++ also differ between operating systems. When using files and ports, there is no stringent requirement that both processes need to be running on the same machine, but with IPC messaging, this would also be required. This type of messaging was not implemented in the current experiment because of the aforementioned issues; it would, however be interesting to test this out in the future.

5.2.2.5 Shared Memory

Shared memory can be used to send large amounts of data by having both processes look at the same area in memory. Implementing this is not simple. When using C++, direct management of memory is a simplistic task; in Java this is not the case. Being able to directly alter memory is purposely not allowed with Java. Therefore, as in the IPC method, third party libraries [19] are required to be able to affect memory directly. These are again dependant on the operating system and both processes must be running on the same computer. For similar reasons to IPC messaging, difficulty of implementation and the inability to have the Cognitive Layer on a separate machine, shared memory was not implemented in this experiment.

5.2.3 Messages

The three message types, commands, variable updates and large data updates were implemented as three different types of tests. For the purposes of this experiment, to simulate the message types, each is constructed of completely random characters with varying lengths.

Commands were constructed to be between 5 and 10 characters in length. This was designed to be of similar length to a typical command. Depending on the speed of the communication method, between 100 and 100,000 messages were sent and received.

Variable updates were between 500 and 1,000 characters in length, simulating a large number of updated variables. Depending on the speed of the communication method, between 100 and 100,000 messages were sent and received.

The large data updates were all 64,000 characters in length. This was to emulate an 8-bit per pixel 320x200 greyscale image. The cameras used on the Magellan Pro capture still images of this size which would likely be the largest size of data that would need to be sent. Between 100 and 10,000 of these messages were sent and received based on the speed of the communication method.

5.2.4 Observations

This section will describe the results achieved by running the tests. A table of all the results can be found in Appendix C.

In Figure 5.8, the results from using UDP can be seen. The dashed line (top) shows the results for large data update messages and the solid line (bottom) shows the results for the command messages. Clearly, the larger a message, the longer it takes and this appears to be an exponential relationship. Figure 5.9 through Figure 5.11 show an exponential fit for each of the packet sizes. If a PRA was to use this type of communication,

implementation would be fairly easy and, with 100,000 small messages being transferred in under 15,000 ms, probably very efficient.

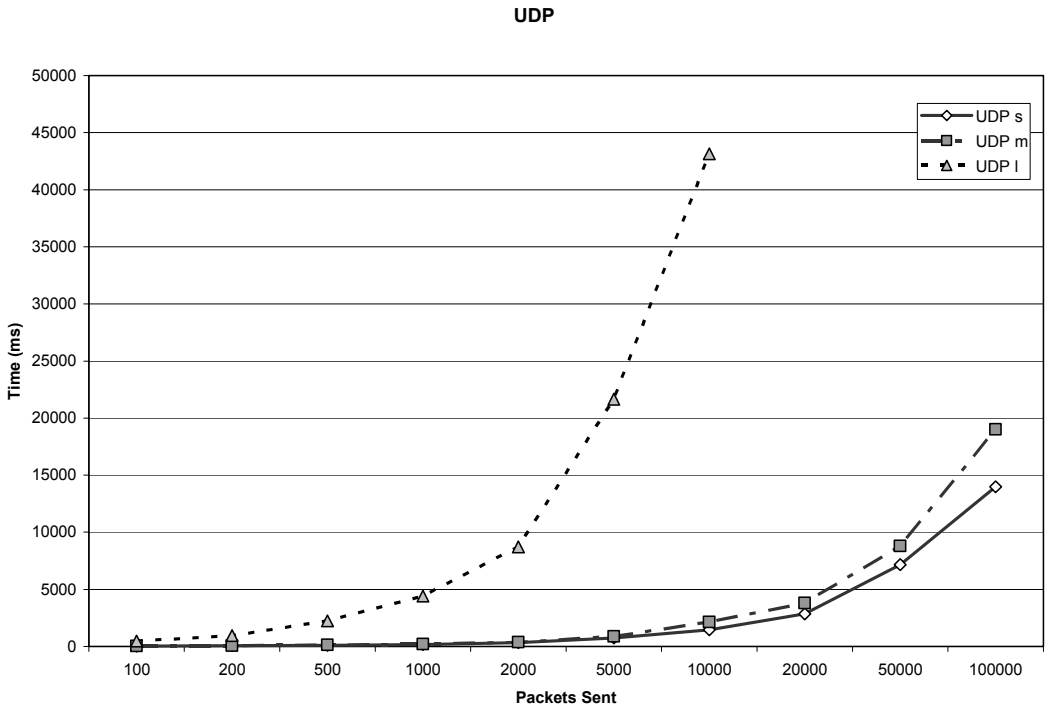

Figure 5.8: UDP Results

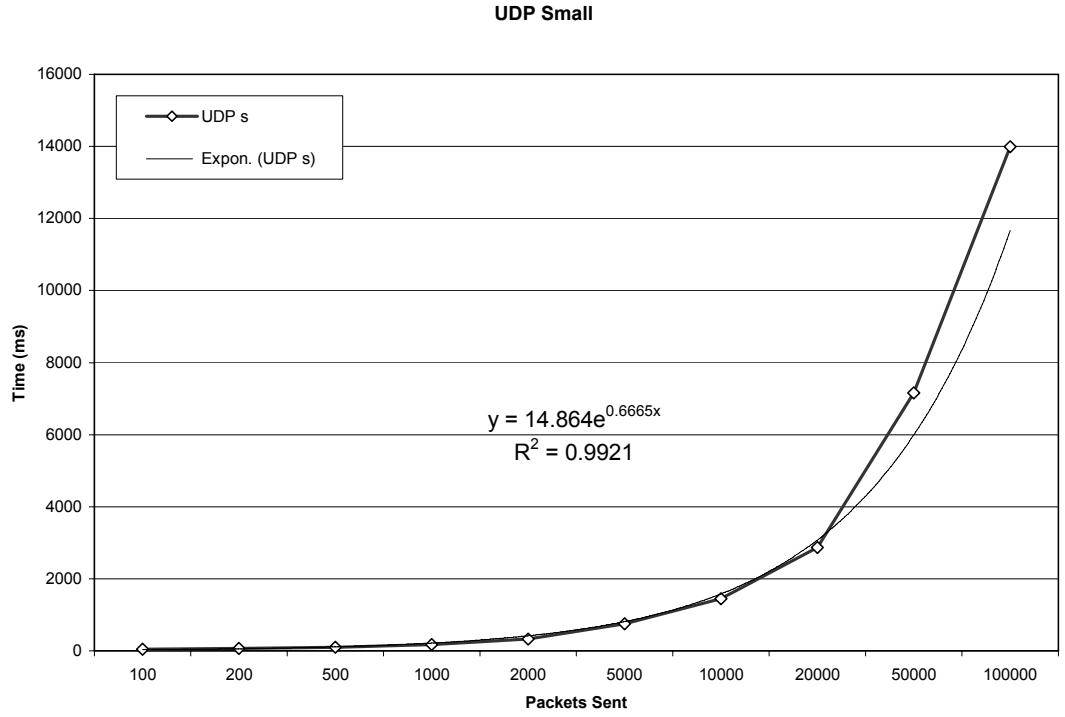

Figure 5.9: UDP Small Exponential Trend

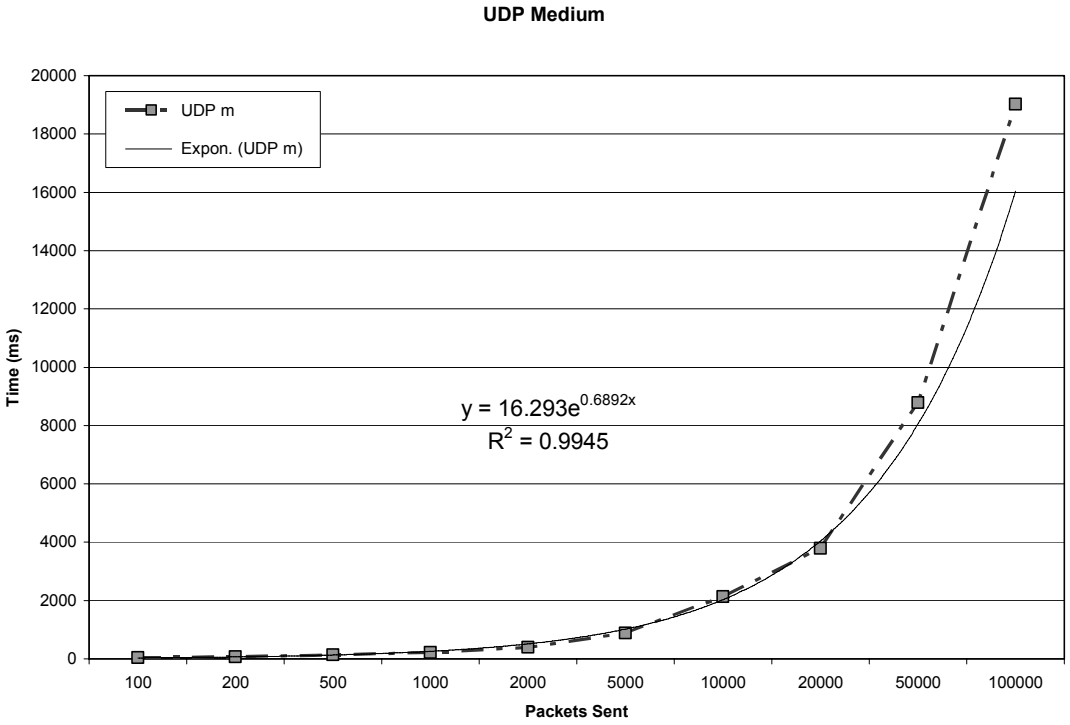

Figure 5.10: UDP Medium Exponential Trend

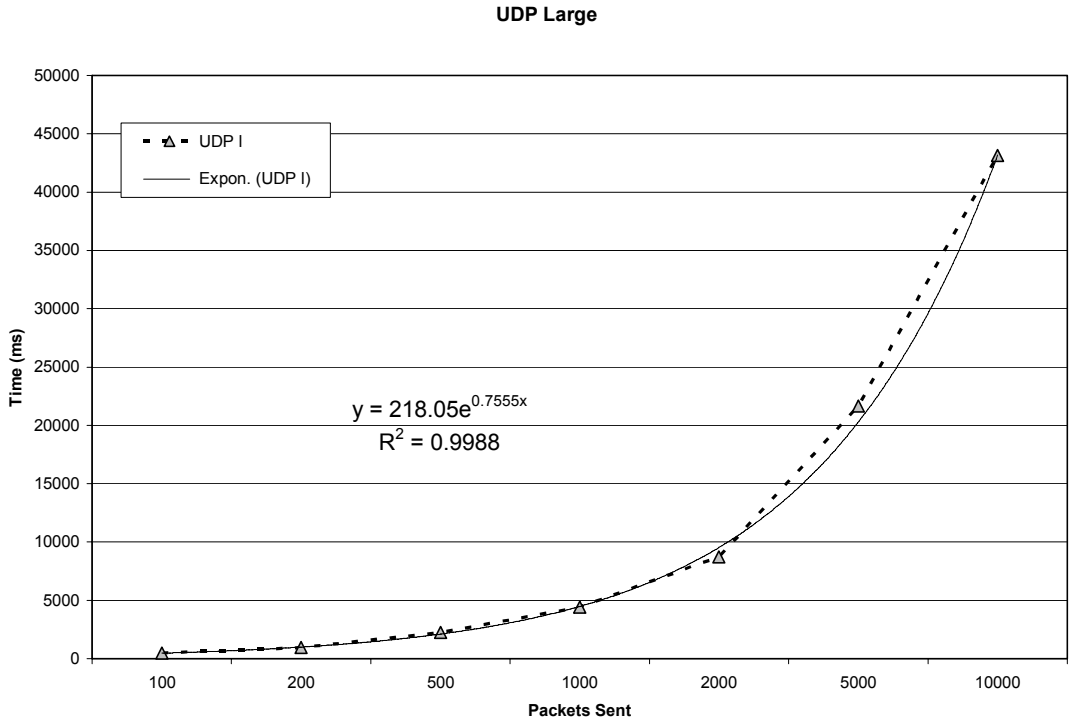

Figure 5.11: UDP Large Exponential Trend

The results achieved for TCP messages, which can be seen in Figure 5.12, are the opposite of those obtained for UDP messages. Again, there appears to be a non-linear relationship but clearly, larger messages take less time to process than smaller ones. The largest messages, large data updates (dashed line, but this time at the bottom) seem to take significantly less time to process. This may be due to the length of the buffer in the Action Layer. If the buffer was not filled, the TCP stream may have needed to time-out before being processed. This could be the cause of what appears to be backward results. More information on this type of delay can be found in [26]. While noticeably slower than the UDP messages, 5,000 message taking over 22,000 ms, if there is a chance of a fault between the layers, then this delay could be worth knowing that the message was received properly.

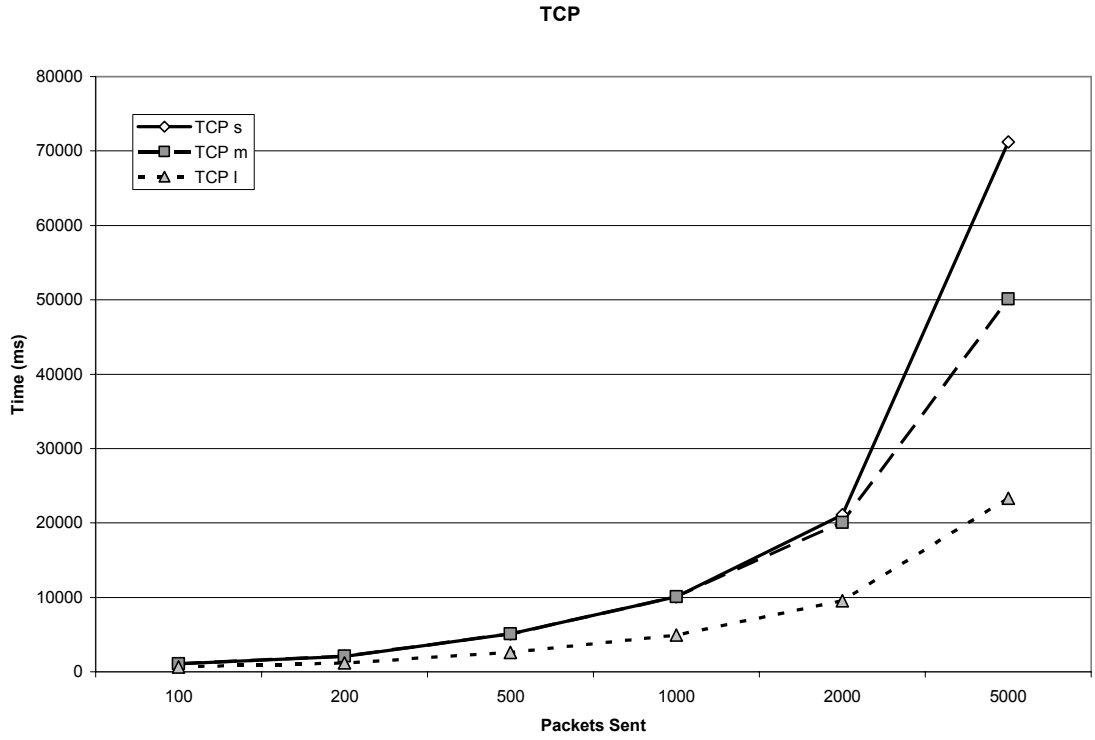

Figure 5.12: TCP Results

File communication seemed to be consistent in time and the differences between large and small messages clearly were miniscule compared to the time required to create and delete files. Figure 5.13 demonstrates that although large messages take a slightly longer time, the size of the file is not the major determining factor in communication latency. If the PRAs had some type of shared memory resource, this slower method of communications could be used, especially if the required messages were very large in size.

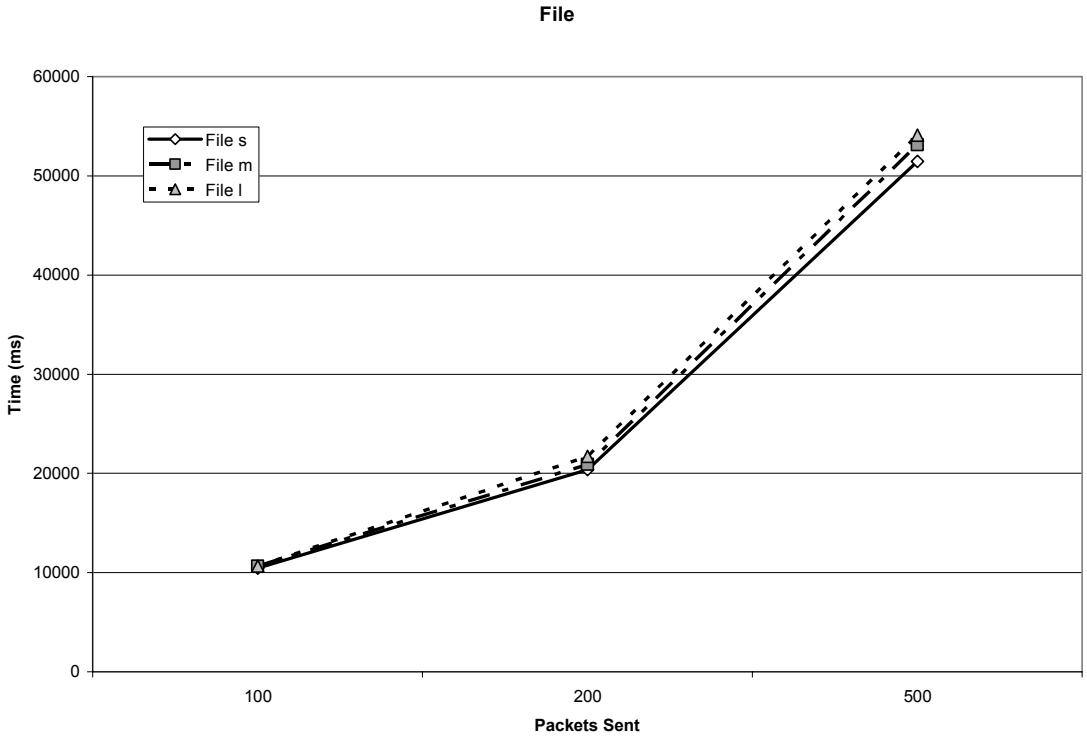

Figure 5.13: File Sharing Results

5.2.4.1 Comparison

When comparing the three types of communication, UDP is clearly the fastest, in bytes per second. All the tests were averaged and the results are presented in Figure 5.14. File communication functioned at an average of 0.21 bytes per second. TCP functioned at 2.02 bytes per second and UDP functioned at the very quick throughput of 101 bytes per second. With these speed estimates, the real-time performance of the inter-layer communication of a PRA can be calculated.

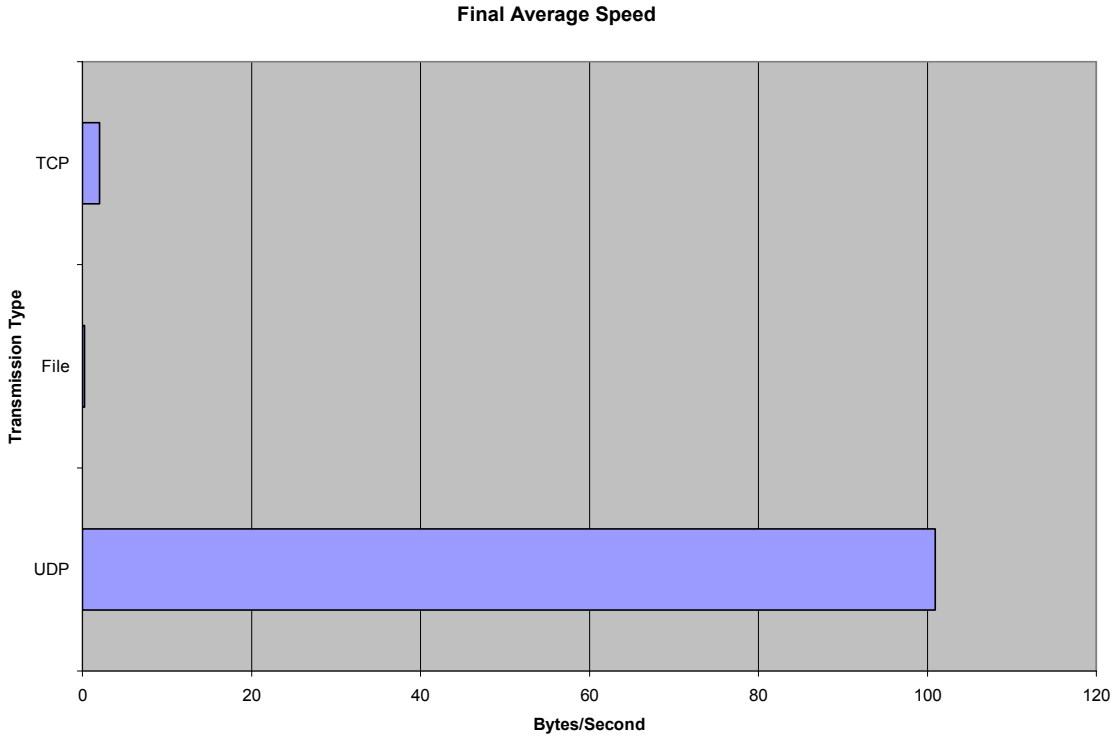

Figure 5.14: Average Throughput

These results can be attributed to a number of different factors. UDP is a bare bones type of communication; it does not have any of the error handling features that exist in TCP and thus, runs at a much quicker speed. Even if the TCP packet size was the same size as each transmitted message and timeout did not occur, it would still be slower than UDP due to the error correction.

5.2.5 Conclusions

UDP messages are sent 50 times faster than TCP messages and sent 500 times faster than files. It was expected that file messages would have the worst throughput, but the difference between TCP and UDP was not. The buffer problem greatly affected performance of the TCP communications. When examining most of the messages sent

between the layers and both layers existing on the same robot, the order of the messages, UDP greatest detriment, does not play an important role. For the most part, the variable updates are typically sent at intervals; the order of the commands received will not be important and large data updates would be infrequent. Clearly, using UDP, perhaps with a data structure of some sort, is the best choice of the three examined in this experiment for sending messages and is reliable on stable networks. If network stability is a potential problem, TCP messages should be considered instead. With an average throughput of 101 bytes per second, it is also possible to calculate, with knowledge of the information being sent back and forth between layers, how long it will take for a robot to be able to respond to a command in real-time.

6 Conclusions, Limitations and Recommendations

This chapter contains the conclusions reached through this thesis. It also contains the limitations of the research and the conclusions as well as recommendations for

6.1 Conclusions

Multiple robot system control is a quickly emerging field in which an agent oriented solution seems to hold the most potential. Currently, there is no well known, effective architecture that exists that will allow agents to be both independent and control low level systems. The goal of this thesis is to present an agent based architecture that has the ability to control both the physical aspects of the robot and coordinate with other robots. Furthermore, a secondary goal is to determine the feasibility of using this architecture for real-time systems.

A novel architecture for multiple robot systems was developed. The architecture is designed to help multiple robots cooperate and coordinate with each other using an agent oriented solution. To accomplish this, each robot is designated to be a Physical Robot Agent (PRA). The PRA is then abstracted into two different layers: the Cognitive and Action Layers. The Action Layer controls all of the sensors and actuators of the robot. All of the physical tasks that the PRA will pursue are controlled by this layer. The Cognitive Layer controls all of the planning and coordination and cooperation with the other PRAs. The Cognitive Layer gives commands to the Action Layer, which in turn returns status updates to the Cognitive Layer.

The Action Layer is further divided into three main components. The first of these components is the Executor. The Executor has access to all of the sensors and actuator on the robot. It controls exactly how the robot moves and reacts. It receives all of its commands from the State Monitor and exactly how to perform these commands from the library of tasks, the Repository element. The Repository is a collection of tasks that the Cognitive Layer can request the Action Layer to perform. If these tasks are 'learning' tasks, the optimized versions are updated within the Repository. The State Monitor tells the Executor what tasks to run or stop based on messages received from the Cognitive Layer. It also sends the Cognitive Layer state variable updates as requested.

Similar to the Action Layer, the Cognitive Layer is split into three elements: the Decision Maker, the Negotiator and the Coordinator. The Decision Maker is the 'brain' of the PRA. It performs all of the planning associated with the robot. To communicate with other robots, the Negotiator element is used. All messages that are sent to the PRA must go through the Negotiator and the Decision Maker to ensure the autonomy of the robot. The Coordinator element handles the communications between the Action Layer and the Decision Maker. It ensures that all variables requested are updated correctly and that all the commands given by the Decision Maker are followed.

To implement the Action Layer, a C++ programme that contained all of the required elements was utilized. The Cognitive Layer was designed using agents. The Java Agent DEvelopment Framework (JADE) was used in conjunction with the Java programming language to create two software agents. The Coordinator is its own software agent and the Decision Maker and Negotiator share a software agent.

Experiments were conducted in which the architecture performed correctly. Experiments with a single robot proved that the architecture was feasible. Further experiments with multiple robots, some in which robots were added or removed, also demonstrated that the multiple robot system functioned perfectly. Based on these results, it is concluded that the architecture is fully functional, robust, portable and practical.

For the architecture to be considered for real-time applications, the communication delays must be tested. The two locations where these delays can occur are between the two layers and between PRAs. The communication between the layers was tested for latency and throughput. The type of messaging that should be used between them was determined.

To test the communication between layers, a basic setup was used. Both layers were emulated and messages were sent using the transmission control protocol (TCP), the user datagram protocol (UDP) and using files on the robot's hard drive. File communications functioned at an average of 0.21 bytes per second. TCP functioned at 2.02 bytes per second and UDP functioned with a throughput of 101 bytes per second. Messages sent using UDP are 50 times faster than TCP messages and 500 times faster than files. Based on this test, it is concluded that all messages passed between layers should be performed using UDP. The cost of using UDP is the possibility of lower reliability due to the lack of error detection. On most systems, where both layers are on the same robot, this is not an issue.

Based on the experiments conducted, it is concluded that the architecture proposed in this thesis is functional, adaptable and reliable in both a single robot and a multi-robot environment. Furthermore, all communications between layers should be conducted using the UDP unless extremely large amounts of data are required. The architecture allows for small and large systems of robots to use many different hierarchies. The abstraction of the two layers and the sub elements allow for the portability of code between completely different robots. The agent level programming can take place independently from the needs of a specific robot. All of these features allow for a shorter time between simulation, implementation and realization.

6.2 Limitations

The architecture could be limited by the precise definition of 'agent' that is required. Furthermore, the architecture can potentially increase the amount of data exchange needed due to the strict information pipelines between elements within each layer. These pipelines ensure that the autonomy of each PRA is intact, however their existence will add some processing time.

There is also a limitation enforced by the separation of the layers. Communications between layers in the dual-layer architecture require more effort than in a single layer system. This more formal type of communication fosters a better Action Layer design.

The application experiment presented in section 5.1 was effective at using up to three robots. A constraint on the results was that there were a limited number of available robots. Theoretically, the framework could sustain a large number of robots, but when

dealing with very large systems (i.e. swarms) it could break down. The limit on the number of robots that could be used in this specific experiment is based on maximum that could fit around the target without colliding.

6.3 Recommendations and Future Work

Based on the conclusions drawn from this thesis, the following are recommended:

1. All future work with mobile robots within the Pattern Analysis and Machine Intelligence (PAMI) Lab at the University of Waterloo should be done using this dual layer architecture.

2. A library of tasks, the Repository element, should be built up to allow for faster development of new multiple robot experiments.

3. Different hierarchies should be tested on the robots to ensure that this architecture is fully compatible with the robots.

4. Real-time experiments should be continued, concentrating on the communications between robots.

5. Further communication tests using more protocols and a random access memory drive to determine a comparison to the communication methods tested should be performed.

6. Experiments involving different types of robots should be conducted using this architecture to explore portability.

References

[1] Bellifemine, F.; Poggi, A.; Rimassa, G.; "JADE - A FIPA-compliant Agent Framework" *CSELT internal technical report, part of this report has been also published in Proceedings of PAAM'99*, London, April, 1999, pp 97 - 108.

[2] Berna-Koes, M.; Nourbakhsh, I.; Sycara, K.; "Communication efficiency in multi-agent systems" *International Conference on Robotics and Automation*, April 26 - May 1, 2004, pp 2129 – 2134, Vol.3

[3] Cao, Y.; Fukunaga, A.; Kahng, A.; "Cooperative Mobile Robotics: Antecedents and Directions" Autonomous Robots, 4, 1-23, Kluwer Academic Publishers (1997)

[4] Cossentino, M.; Sabatucci, L.; Chella, A.; "A possible approach to the development of robotic multi-agent systems" *IEEE/WIC International Conference on Intelligent Agent Technology*, October 13 - 16, 2003, pp 539 – 544

[5] Eze, J.; Ghenniwa, H.; Shen, W.; "Distributed Control Architecture for Collaborative Physical Robot Agents" *IEEE International Conference on Systems, Man & Cybernetics*, Washington DC, 2003, pp 2977 – 2982

[6] Eze, J.; Ghenniwa, H.; Shen, W.; "Integration Framework for Collaborative Remote Physical Agents" *3rd International Symposium on Robotics and Automation,* 2002, Toluca, Mexico

[7] Finin, Tim; Labrou, Yannis; Mayfield, James; "KQML as an agent communication language" Software Agents, MIT Press, Cambridge (1997)

[8] FIPA – Foundation for Intelligent Physical Agents, *http://www.fipa.org* (August, 2005)

[9] Gao Zhijun; Yan Guozheng; Ding Guoqing; Huang Heng; "Research of communication mechanism of multi-agent robot systems", *International Symposium on Micromechatronics and Human Science,* September 9 – 12, 2001, pp 75 – 79

[10] Genesereth, M. R.; Ketchpel, S. P.; "Software Agents", *Communication of the ACM*, July, 1994, Vol. 37, No. 7

[11] Ghenniwa, H.; Kamel, M.; "Interaction Devices for Coordinating Cooperative Distributed Systems", *Automation and Soft Computing*, 2000, pp.173 - 184, Vol 6, No 2

[12] Gruneir, Bram; Miners, Ben; Khamis, Alaa; Ghenniwa, Hamada; Kamel, Mohamed; "Agent oriented Design of a Multi-Robot System" *International Workshop on Multi-Agent Robotic Systems, International Conference on Informatics in Control, Automation and Robotics*, 2005

[13] Harmon, Scott Y.; Gage, Douglas W.; "Protocols for Robot Communications: Transport and Content Layers" *International Conference on Cybernetics and Society*, Cambridge Massachusetts, October 8, 1980, pp 1090 – 1097

[14] Howell, Whitney; Patel, Seema; Minten, Brian; "UDP Performance over an Ad Hoc Network for Mobile Robots" *International Conference on Wireless Network,* 2004, pp. 520 – 526

[15] Huancheng Zhang; Miaoliang Zhu; "An architecture for outdoor mobile robot navigation with self-organization" *International Control, Automation, Robotics and Vision Conference*, December 6 - 9, 2004, pp 267 – 272, Vol. 1

[16] iRobot Corporation, *http://www.irobot.com/home/default.asp* (August, 2005)

[17] Java – Java Technology, *http://java.sun.com/* (March, 2005)

[18] Jennings, N.; Sycara, K.; Wooldridge, M.; "A roadmap of agent research and development" *Journal of Autonomous Agents and Multi-Agent Systems*, 1998, pp 275 – 306, Vol. 1

[19] JTux: Java to Unix Package, *http://www.basepath.com/aup/jtux* (August, 2005)

[20] Kawamura, K.; "The Role of Cognitive Agent Models in a Multi-Agent Framework for Human-Humanoid Interaction" *11th IEEE International Workshop on Robot and Human Interactive Communication,* September 25 – 27, 2002, pp 81 – 86

[21] Lei Cheng; Yong-Ji Wang; "Fault tolerance for communication-based multirobot formation" *International Conference on Machine Learning and Cybernetics*, August 26 – 29, 2004, pp 27 – 132, Vol.1

[22] Lucidarme, P.; Simonin, O.; Liégeois, A.; "Implementation and Evaluation of a Satisfaction/Altruism Based Architecture for Multi-Robot Systems" *International Conference on Robotics and Automation*, May, 2002, pp 1007 – 1012

[23] Mataric, M.J.; Nilsson, M.; Simsarian, K.T.; "Cooperative multi-robot box-pushing", *IEEE/RSJ IROS*, 1995, pp 556 - 561

[24] Moulin, B.; Chaib-draa, B.; "An overview of distributed artificial intelligence", *Foundations of Distributed Artificial intelligence*, John Wiley & Sons, New York, NY, 1996, pp 3 - 55

[25] Multi Agent Systems, *http://mas.colognet.org/implementation.html* (August, 2005)

[26] Padhye, J.; Firoiu, V.; Towsley, D.; Kurose, J.; "Modeling TCP throughput: a simple model and its empirical validation" *ACMSIGCOMM*, September, 1998

[27] Tigli, J. Y.; Thomas, M. C.; “Use of multi agent systems for mobile robotics control” *IEEE International Conference Systems, Man, and Cybernetics: Humans, Information and Technology*, October 2 – 5, 1994, pp 588 – 592, Vol.1

[28] Wikipedia, *http://en.wikipedia.org/wiki* (February, 2005)

[29] Wooldridge, M.; “An Introduction to MultiAgent Systems”, West Sussex, England: John Wiley & Sons Ltd. (2002)

[30] Zhang, X., Norrie, D.H.(1999), "Holonic Control at the Production and Controller Levels", *IMS 99*, Leuven, Belgium, September 22-24, 1999, pp. 215 - 224.

Appendix A: Implementation Application Source Code

This appendix contains the Java source code to the Cognitive Layer software agents from section 5.1.

A.1 Greeter

```
import jade.core.Agent;
import jade.core.behaviours.*;
import jade.lang.acl.*;
import jade.core.AID;
import java.util.*;

public class Greeter extends Agent {
  public final int Robots = 3;

  protected void setup() {
    addBehaviour(new  GreetRobot(this));
    System.out.println("Greeter has started as " + getName());
  }

  class GreetRobot extends CyclicBehaviour {
    public GreetRobot(Agent a) {
      super(a);
    }

    public void action() {
      boolean exitcode = false;
      boolean exitcode2 = false;
      LinkedList agentlist = new LinkedList();
      agentlist.clear();
      ACLMessage hello;
      System.out.println("Waiting for robots...");

      while (exitcode == false) {
        System.out.println("Number of Robots reporting:  " +
agentlist.size());
        ListIterator it2 = agentlist.listIterator();
        AID t;
        int c = 0;
        while (it2.hasNext()) {
          c = c + 1;
          t = (AID)it2.next();
          System.out.println("Robot " + c + ":" + t.getLocalName());
        }
```

```
        ACLMessage welcome = myAgent.blockingReceive();
        System.out.println("Received message");
        if (welcome.getPerformative() == ACLMessage.FAILURE) {
          System.out.println("Exiting program");
          exitcode = true;
        } else if (welcome.getPerformative() ==
ACLMessage.REJECT_PROPOSAL) {
          ListIterator it = agentlist.listIterator();
          while (it.hasNext()) {
            if (welcome.getSender().equals((AID)it.next())) {
              it.remove();
              System.out.println("Robot removed: " +
welcome.getSender().getLocalName());
            }
          }
        } else if (welcome.getPerformative() == ACLMessage.DISCONFIRM)
{
          System.out.println("Removing all robots.");
          agentlist.clear();
        } else if (welcome.getPerformative() == ACLMessage.CANCEL) {
          ListIterator it = agentlist.listIterator();
          AID you;
          while (it.hasNext()) {
            you = (AID)it.next();
            if (welcome.getContent().compareTo(you.getLocalName()) ==
0) {
              ACLMessage goodbye = new ACLMessage(ACLMessage.REFUSE);
              goodbye.addReceiver(you);
              myAgent.send(goodbye);
              System.out.println("Sending goobye to: " +
you.getLocalName());
            }
          }
        } else {
          ListIterator it = agentlist.listIterator();
          exitcode2 = false;
          while ((exitcode2 == false) && (it.hasNext())) {
            if (welcome.getSender().equals((AID)it.next())) {
              System.out.println("Already found instance of:  " +
welcome.getSender().getLocalName());
              exitcode2 = true;
            }
          }
          if (exitcode2)
            it.remove();
          agentlist.addFirst(welcome.getSender());
          System.out.println("Adding robot "+ agentlist.size() + ":  "+
welcome.getSender().getLocalName());
          if (agentlist.size() > 1) {
            ListIterator agentlistit = agentlist.listIterator();
            agentlistit.next();
            while (agentlistit.hasNext()) {
              hello = new ACLMessage(ACLMessage.CONFIRM);
              hello.setContent(Integer.toString(agentlist.size()));
              hello.clearAllReceiver();
              hello.addReceiver(welcome.getSender());
              hello.setSender((AID)agentlistit.next());
```

```
              System.out.println("Sending message to " +
welcome.getSender().getLocalName());
              myAgent.send(hello);
            }
          }
          hello = new ACLMessage(ACLMessage.DISCONFIRM);
          hello.setContent("All Robots Reported");
          hello.addReceiver(welcome.getSender());
          myAgent.send(hello);
        }
      }
    }
  }
}
```

A.2 Commander

```
import jade.core.Agent;
import jade.core.behaviours.*;
import jade.lang.acl.*;
import jade.core.AID;
import java.util.Iterator;
import java.util.HashMap;


public class Jade2Robot extends Agent {
  ParallelBehaviour par;
  AID reportto = null;
  String RunBeh = "";
  private ACLMessage resend = null;
  private int sentlast;

  private BehaveState bs;
  private robotMgrStub stub;

  protected void setup() {
    bs = new BehaveState();
    sentlast = 15;
    stub = new robotMgrStub(this.getLocalName(),7392,0);
    par = new ParallelBehaviour(this, ParallelBehaviour.WHEN_ALL);
    par.addSubBehaviour(new RecMsg(this));
    par.addSubBehaviour(new Report(this));
    addBehaviour(par);
    System.out.println("Jade 2 Robot (J2R) has started on " +
getLocalName());
  }

  class RecMsg extends CyclicBehaviour {
    public RecMsg(Agent a) {
      super(a);
    }

    public void action() {
      ACLMessage msg = myAgent.receive();
      while (msg!=null) {
```

```
        System.out.println("J2R <- " + msg.getSender().getName() + " ("
+ ACLMessage.getPerformative(msg.getPerformative()) + "):" +
msg.getContent() );
        if (msg.getPerformative() == ACLMessage.INFORM)
          par.addSubBehaviour(new SendStub(myAgent, msg));
        if (msg.getPerformative() == ACLMessage.SUBSCRIBE)
          par.addSubBehaviour(new Subscribe(myAgent, msg));
        if (msg.getPerformative() == ACLMessage.CANCEL)
          par.addSubBehaviour(new Unsubscribe(myAgent, msg));
        if (msg.getPerformative() == ACLMessage.REQUEST)
          par.addSubBehaviour(new Monitor(myAgent,msg));
        msg = myAgent.receive();
      }
      block();
    }
  }

  class SendStub extends OneShotBehaviour {
    private ACLMessage msg;
    public SendStub(Agent a, ACLMessage Mess) {
      super(a);
      msg = Mess;
    }
    public void action() {
      int val = 0;
      String stuff[] = msg.getContent().split(":",2);
      if (stuff[0].compareTo(RunBeh) == 0)
      {
        System.out.println("J2R Maintaining Behaviour:" + stuff[0]);
      }
      else
      {
        stub.ActivateBehaviour("","");
        stub.ActivateBehaviour(stuff[0],stuff[1]);
        System.out.println("J2R Activate Behaviour: " + stuff[0] + " -
" + stuff[1]);
        RunBeh = stuff[0];
      }
    }
  }

  class Subscribe extends OneShotBehaviour {
    private ACLMessage msg;

    public Subscribe(Agent a, ACLMessage Mess) {
      super(a);
      msg = Mess;
    }

    public void action() {
      reportto = msg.getSender();
      System.out.println("J2R New Subscriber set to:" +
reportto.getName());
      par.addSubBehaviour(new Reply(myAgent,msg,ACLMessage.AGREE));
    }
  }
```

```
  class Unsubscribe extends OneShotBehaviour {
    private ACLMessage msg;
    public Unsubscribe(Agent a, ACLMessage Mess) {
      super(a);
      msg = Mess;
    }
    public void action() {
      if (reportto == msg.getSender())
        reportto = null;
      System.out.println("J2R Subscriber removed");
    }
  }

  class Monitor extends OneShotBehaviour {
    private ACLMessage msg;
    public Monitor(Agent a, ACLMessage Mess) {
      super(a);
      msg=Mess;
    }
    public void action() {
      while (stub.MonitorVariable(msg.getContent()) == false);
      stub.SetVariable(msg.getContent(),0);
      System.out.println("J2R Now Monitoring:" + msg.getContent());
      par.addSubBehaviour(new Reply(myAgent,msg,ACLMessage.AGREE));
    }
  }

  class Report extends CyclicBehaviour {
    public Report(Agent a) {
      super(a);
    }
    private ACLMessage msg;
    private String stuff = new String();
    private Integer temp = new Integer(1);
    public void action() {
      int i;
      if (!(reportto == null)) {
        if ( stub.GetState(bs,1) ) {
          i = bs.mVariable.size();
          stuff = i + "!";
          for (Iterator it = bs.mVariable.keySet().iterator();
it.hasNext(); ) {
            String key = (String)(it.next());
            stuff += key + ":" + (Integer)bs.mVariable.get(key) + "!";
          }
          stuff = stuff.substring(0,stuff.length()-1);
          msg = new ACLMessage(ACLMessage.INFORM);
          resend = new ACLMessage(ACLMessage.INFORM);
          msg.addReceiver(reportto);
          resend.addReceiver(reportto);
          msg.setContent(stuff);
          resend.setContent(stuff);
          //System.out.println("J2R Reporting: " + stuff);
          myAgent.send(msg);
          sentlast = 0;
        }
        else
```

```
        {
          sentlast++;
          if (sentlast == 10)
            myAgent.send(resend);
          if (sentlast == 10000)
            sentlast=11;
        }
      }
    }
  }

  class Reply extends OneShotBehaviour {
    ACLMessage msg;
    int perf;
    public Reply(Agent a, ACLMessage Mess, int per) {
      super(a);
      msg = Mess;
      perf = per;
    }
    public void action() {
      ACLMessage reply = new ACLMessage(perf);
      reply.addReceiver(msg.getSender());
      myAgent.send(reply);
    }
  }
}
```

A.3 Single Robot Implementation

```
import jade.core.Agent;
import jade.core.behaviours.*;
import jade.lang.acl.*;
import jade.core.AID;

public class BestView extends Agent {
  ParallelBehaviour par;
  SequentialBehaviour seq;
  private AID j2r = new AID("mag1",AID.ISLOCALNAME);

  protected boolean postFound = false;
  protected boolean postCentered = false;
  protected boolean postClose = false;
  protected boolean panAt90 = false;
  protected boolean atBestCorner = false;
  protected int state = 0;


  protected void setup() {
    seq = new SequentialBehaviour(this);
    seq.addSubBehaviour(new Startup(this));
    par = new ParallelBehaviour(this, ParallelBehaviour.WHEN_ALL);
    par.addSubBehaviour(new RecMsg(this));
    seq.addSubBehaviour(par);
    System.out.println("Bestview Agent is running as " + getName());
    addBehaviour(seq);
  }
```

```
  class Startup extends OneShotBehaviour {
    public Startup(Agent a) {
      super(a);
    }
    public void action() {
      SequentialBehaviour seq2 = new SequentialBehaviour(myAgent);
      seq2.addSubBehaviour(new Send2J2R(myAgent,"postCentered",
ACLMessage.REQUEST));
      seq2.addSubBehaviour(new Send2J2R(myAgent,"postFound",
ACLMessage.REQUEST));
      seq2.addSubBehaviour(new Send2J2R(myAgent,"postClose",
ACLMessage.REQUEST));
      seq2.addSubBehaviour(new Send2J2R(myAgent,"panAt90",
ACLMessage.REQUEST));
      seq2.addSubBehaviour(new Send2J2R(myAgent,"atBestCorner",
ACLMessage.REQUEST));
      seq2.addSubBehaviour(new Send2J2R(myAgent,"",
ACLMessage.SUBSCRIBE));
      par.addSubBehaviour(seq2);
    }
  }

  class Send2J2R extends OneShotBehaviour {
    String var;
    int perf;
    public Send2J2R(Agent a, String inside, int type) {
      super(a);
      var = inside;
      perf = type;
    }
    public void action() {
      ACLMessage reply = new ACLMessage(ACLMessage.REFUSE);
      while (reply.getPerformative() != ACLMessage.AGREE) {
        ACLMessage msg = new ACLMessage(perf);
        msg.addReceiver(j2r);
        msg.setContent(var);
        myAgent.send(msg);
        reply = myAgent.blockingReceive();
      }
    }
  }

  class RecMsg extends CyclicBehaviour {
    public RecMsg(Agent a) {
      super(a);
    }
    public SequentialBehaviour seq3;
    public void action() {
      ACLMessage msg = myAgent.receive();
      if (msg!=null) {
        System.out.println(myAgent.getLocalName() + " <- " +
msg.getSender().getName() + " (" +
ACLMessage.getPerformative(msg.getPerformative()) + "):" +
msg.getContent() );
        //if (msg.getSender().getName().compareTo(j2r.getName()) == 0)
{
```

```
          if (msg.getPerformative() == ACLMessage.INFORM)
          {
            seq3 = new SequentialBehaviour(myAgent);
            seq3.addSubBehaviour(new UpdateVar(myAgent, msg));
            seq3.addSubBehaviour(new StateMachine(myAgent));
            par.addSubBehaviour(seq3);
          }
        //}
        //else {
          //This is for later...
        //}
      }
      block();
    }
  }

  class StateMachine extends OneShotBehaviour {
    public StateMachine(Agent a) {
      super(a);
    }

    public void action() {
      System.out.println("State = " + state);
      switch (state) {
        case 0: if (postFound)
                  state++;
        break;
        case 1: if (!postFound)
                  state = 0;
                else if (postCentered && postClose)
                  state++;
        break;
        case 2: if (!postFound)
          state = 0;
        else if (postCentered && panAt90)
          state++;
        break;
        case 3: if (!postFound)
          state = 0;
        else if (atBestCorner)
          state++;
        break;
        case 4: if (!postFound)
          state = 0;
        else if (!atBestCorner)
          state--;
        break;
      }
      switch (state) {
        case 0: par.addSubBehaviour(new SetNewBehaviour(myAgent,
"findPost:0")); break;
        case 1: par.addSubBehaviour(new SetNewBehaviour(myAgent,
"approachPost:0")); break;
        case 2: par.addSubBehaviour(new SetNewBehaviour(myAgent,
"readyPivot:0")); break;
        case 3: par.addSubBehaviour(new SetNewBehaviour(myAgent,
"Pivot:0")); break;
```

```
        case 4: par.addSubBehaviour(new SetNewBehaviour(myAgent,
"onAlert:0")); break;
        default: par.addSubBehaviour(new SetNewBehaviour(myAgent,
":0")); break;
      }
    }
  }

  class SetNewBehaviour extends OneShotBehaviour {
    private String behave;

    public SetNewBehaviour(Agent a, String beh) {
      super(a);
      behave = beh;
    }

    private ACLMessage msg = new ACLMessage(ACLMessage.INFORM);

    public void action() {
      msg.addReceiver(j2r);
      msg.setContent(behave);
      //System.out.println("Setting Behaviour to" + behave);
      myAgent.send(msg);
    }
  }

  class Subscribe extends OneShotBehaviour {
    public Subscribe(Agent a) {
      super(a);
    }

    private ACLMessage msg = new ACLMessage(ACLMessage.SUBSCRIBE);

    public void action() {
      msg.addReceiver(j2r);
      myAgent.send(msg);
    }
  }

  class Unsubscribe extends OneShotBehaviour {
    public Unsubscribe(Agent a, ACLMessage Mess) {
      super(a);
    }

    private ACLMessage msg = new ACLMessage(ACLMessage.CANCEL);

    public void action() {
      msg.addReceiver(j2r);
      myAgent.send(msg);
    }
  }

  class UpdateVar extends OneShotBehaviour {
    private ACLMessage msg;
    public UpdateVar(Agent a, ACLMessage Mess) {
      super(a);
      msg = Mess;
```

```
    }
    public void action() {
      boolean temp;
      String updates[] = msg.getContent().split("!");
      for (int i=1; i <= Integer.parseInt(updates[0]); i++)
      {
        String var[] = updates[i].split(":",2);
        temp = false;
        if (var[1].compareTo("1") == 0)
          temp = true;
        System.out.println(var[0] + "-" + var[1] + "=" + temp);
        if (var[0].compareTo("postFound") == 0)
          postFound = temp;
        if (var[0].compareTo("postClose") == 0)
          postClose = temp;
        if (var[0].compareTo("postCentered") == 0)
          postCentered = temp;
        if (var[0].compareTo("panAt90") == 0)
          panAt90 = temp;
        if (var[0].compareTo("atBestCorner") == 0)
          atBestCorner = temp;
      }
      //System.out.println(myAgent.getLocalName() + ": pf=" + postFound
+ ", pcent=" + postCentered + ", pc=" + postClose);
    }
  }
}
```

A.4 Multiple Robot Implementation

```
import jade.core.Agent;
import jade.core.behaviours.*;
import jade.lang.acl.*;
import jade.core.AID;
import java.math.*;
import java.io.*;
import java.util.*;

public class MBestView extends Agent {
  ParallelBehaviour par;
  SequentialBehaviour seq;
  private AID j2r;
  //private AID friend1 = new AID();
  //private AID friend2 = new AID();
  private AID Greeter;

  class AIDitem {
    public AID agent;
    public boolean playing = false;
    public boolean ready = false;
    public boolean degreefound = false;
    public int degree = -1;
    public int goodDegree = -1;
    public AIDitem(ACLMessage msg) {
      agent = msg.getSender();
      ready = false;
```

```
    degreefound = false;
    degree = -1;
    goodDegree = -1;
  }
  public AIDitem() {
    agent = null;
    ready = false;
    degreefound = false;
    degree = -1;
    goodDegree = -1;
  }
}

AIDitem findinlist(LinkedList l, AID i) {
  Iterator it = l.iterator();
  AIDitem me = null;
  AIDitem me2 = null;
  while (it.hasNext()) {
    me = (AIDitem)it.next();
    if (me.agent.equals(i))
      me2 = me;
  }
  return me2;
}

private LinkedList friends = new LinkedList();
private LinkedList playFriends = new LinkedList();

protected boolean postFound = false;
protected boolean postCentered = false;
protected boolean postClose = false;
protected boolean ready = false;
//protected boolean other1ready = false;
//protected boolean other2ready = false;
protected boolean panAt90 = false;
protected boolean atBestCorner = false;
protected boolean HALTER;
protected boolean QUITER = false;
protected boolean RESET = false;
protected boolean newAngleRec = false;
protected boolean newAngleFound = false;
protected int state = 0;
protected boolean degreeFound = false;
protected int gooddegree = 0;
//protected boolean f1degreeFound = false;
//protected int friend1degree = -1;
//protected boolean f2degreeFound = false;
//protected int friend2degree = -1;
protected int curdegree = -1;

protected void setup() {
  seq = new SequentialBehaviour(this);
  seq.addSubBehaviour(new Startup(this));
  par = new ParallelBehaviour(this, ParallelBehaviour.WHEN_ALL);
  par.addSubBehaviour(new RecMsg(this));
  seq.addSubBehaviour(par);
  System.out.println("Bestview Agent is running as " + getName());
```

```
    HALTER = true;  //This should send a hello to each robot
    addBehaviour(seq);
  }

  class Startup extends OneShotBehaviour {
    public Startup(Agent a) {
      super(a);
    }
    public void action() {
      SequentialBehaviour seq2 = new SequentialBehaviour(myAgent);
      seq2.addSubBehaviour(new sendGreeterHello(myAgent));
      seq2.addSubBehaviour(new Send2J2R(myAgent,"postCentered",
ACLMessage.REQUEST));
      seq2.addSubBehaviour(new Send2J2R(myAgent,"postFound",
ACLMessage.REQUEST));
      seq2.addSubBehaviour(new Send2J2R(myAgent,"postClose",
ACLMessage.REQUEST));
      seq2.addSubBehaviour(new Send2J2R(myAgent,"panAt90",
ACLMessage.REQUEST));
      seq2.addSubBehaviour(new Send2J2R(myAgent,"atBestCorner",
ACLMessage.REQUEST));
      seq2.addSubBehaviour(new Send2J2R(myAgent,"degree",
ACLMessage.REQUEST));
      seq2.addSubBehaviour(new Send2J2R(myAgent,"",
ACLMessage.SUBSCRIBE));
      par.addSubBehaviour(seq2);
    }
  }

  class sendGreeterHello extends OneShotBehaviour {
    public sendGreeterHello(Agent a) {
      super(a);
    }

    public void action() {
      // Setup J2R
      try {
        BufferedReader in = new BufferedReader(new
FileReader("myname.txt"));
        String str;
        str = in.readLine();
        j2r = new AID(str,AID.ISLOCALNAME);
        in.close();
        System.out.println("The J2R is " + str);
      } catch (IOException e) {
      }

      //Send Message to Greeter
      ACLMessage msg = new ACLMessage(ACLMessage.CONFIRM);
      //Read Greeter Corba Address From File
      Greeter = new
AID("Greeter@fusion.uwaterloo.ca:1099/JADE",AID.ISGUID);
      try {
        BufferedReader in = new BufferedReader(new
FileReader("Greeter.txt"));
        String str;
        str = in.readLine();
```

```
        Greeter.addAddresses(str);
        in.close();
      } catch (IOException e) {
      }
      msg.addReceiver(Greeter);
      myAgent.send(msg);

      //Wait for messages from the greeter:
      ACLMessage welcome = myAgent.blockingReceive();
      while (welcome.getPerformative() != ACLMessage.DISCONFIRM) {
        friends.addLast(new AIDitem(welcome));
        System.out.println("Hello new friend:  " +
welcome.getSender().getName());
        welcome = myAgent.blockingReceive();
      }
    }
  }

  class Send2J2R extends OneShotBehaviour {
    String var;
    int perf;
    public Send2J2R(Agent a, String inside, int type) {
      super(a);
      var = inside;
      perf = type;
    }
    public void action() {
      ACLMessage reply = new ACLMessage(ACLMessage.REFUSE);
      while (reply.getPerformative() != ACLMessage.AGREE) {
        ACLMessage msg = new ACLMessage(perf);
        msg.addReceiver(j2r);
        msg.setContent(var);
        myAgent.send(msg);
        reply = myAgent.blockingReceive();
      }
    }
  }

  class RecMsg extends CyclicBehaviour {
    public RecMsg(Agent a) {
      super(a);
    }
    public SequentialBehaviour seq3;
    public void action() {
      ACLMessage msg = myAgent.receive();
      while (msg!=null) {
        System.out.println("***************  GOT MESSAGE!:
********************");
        System.out.println(myAgent.getLocalName() + " <- " +
msg.getSender().getName() + " (" +
ACLMessage.getPerformative(msg.getPerformative()) + "):" +
msg.getContent() );
        if (msg.getPerformative() == ACLMessage.INFORM) {
          seq3 = new SequentialBehaviour(myAgent);
          seq3.addSubBehaviour(new UpdateVar(myAgent, msg));
          //          seq3.addSubBehaviour(new StateMachine(myAgent));
Don't think I need two here.
```

```
  seq3.addSubBehaviour(new StateMachine(myAgent));
  par.addSubBehaviour(seq3);
}
else if (msg.getPerformative() == ACLMessage.CFP) {
  RESET = true;

  //Do we know this robot yet?
  Iterator friendsit = friends.iterator();
  AIDitem me = null;
  AIDitem me2 = null;
  while (friendsit.hasNext()) {
    me = (AIDitem)friendsit.next();
    if (me.agent.equals(msg.getSender()))
      me2 = me;
  }
  if (me2 == null) {
    me2 = new AIDitem(msg);
    friends.addLast(me2);

    ACLMessage hello = new ACLMessage(ACLMessage.CFP);
    hello.addReceiver(msg.getSender());
    hello.setContent("Nice to meet you!");
    myAgent.send(hello);
  }

  //Reset the robot's variables
  me2.ready = false;
  me2.degreefound = false;
  me2.degree = -1;

  //Add robot to working robots
  Iterator pfit = playFriends.iterator();
  AIDitem me3 = null;
  AIDitem me4 = null;
  while (pfit.hasNext()) {
    me3 = (AIDitem)pfit.next();
    if (me3.agent.equals(me2.agent))
      me4 = me3;
  }
  if (me4 == null) {
    playFriends.addLast(me2);
  }
  par.addSubBehaviour(new StateMachine(myAgent));
}
else if (msg.getPerformative() == ACLMessage.FAILURE) {
  HALTER = true;
  playFriends.clear();
  par.addSubBehaviour(new StateMachine(myAgent));
}
else if (msg.getPerformative() == ACLMessage.PROPOSE) {
  Iterator playFriendsit = playFriends.iterator();
  AIDitem me = null;
  AIDitem me2 = null;
  while (playFriendsit.hasNext()) {
    me = (AIDitem)playFriendsit.next();
    if (me.agent.equals(msg.getSender()))
      me2 = me;
```

```
          }
          if (me2 == null) {
            System.out.println("ERROR FRIEND NOT FOUND: " +
msg.getSender().getLocalName());
            ACLMessage err = new ACLMessage(ACLMessage.NOT_UNDERSTOOD);
            err.setContent("You have not registered with me.");
            err.addReceiver(msg.getSender());
            myAgent.send(err);
          }
          else {
            me2.ready = true;
          }
          par.addSubBehaviour(new StateMachine(myAgent));
        }
        else if (msg.getPerformative() == ACLMessage.INFORM_REF) {

          Iterator playFriendsit = playFriends.iterator();
          AIDitem me = null;
          AIDitem me2 = null;
          while (playFriendsit.hasNext()) {
            me = (AIDitem)playFriendsit.next();
            if (me.agent.equals(msg.getSender()))
              me2 = me;
          }
          if (me2 == null) {
            System.out.println("ERROR FRIEND NOT FOUND: " +
msg.getSender().getLocalName());
            ACLMessage err = new ACLMessage(ACLMessage.NOT_UNDERSTOOD);
            err.setContent("You have not registered with me.");
            err.addReceiver(msg.getSender());
            myAgent.send(err);
          }
          else {
            me2.degreefound = true;
            me2.degree = Integer.parseInt(msg.getContent());
            System.out.println("GOT DEGREES:  " + msg.getContent() + ":
Degrees = " + me2.degree);
            newAngleRec = true;
          }

          par.addSubBehaviour(new StateMachine(myAgent));
        }
        else if (msg.getPerformative() == ACLMessage.REJECT_PROPOSAL) {
          //Robot is told that it this other robot is quiting.
          Iterator friendsit = friends.iterator();
          AIDitem me = null;
          while (friendsit.hasNext()) {
            me = (AIDitem)friendsit.next();
            if (me.agent.equals(msg.getSender())) {
              friendsit.remove();
              playFriends.clear();
              HALTER = true;
            }
            //Might need to put some check in here.
            System.out.println("This robot has QUIT the system: " +
msg.getSender().getLocalName());
          }
```

```
        par.addSubBehaviour(new StateMachine(myAgent));
      }
      else if (msg.getPerformative() == ACLMessage.NOT_UNDERSTOOD) {
        //Not in their lists, this means we need to resart
        System.out.println("RESARTING FOR NOT BEING IN ACTIVE LIST");

        ACLMessage halty = new ACLMessage(ACLMessage.FAILURE);
        AIDitem me = null;
        Iterator it = friends.iterator();
        while (it.hasNext()) {
          me = (AIDitem)it.next();
          halty.addReceiver(me.agent);
        }
        myAgent.send(halty);
      }
      else if (msg.getPerformative() == ACLMessage.REFUSE) {
        //This robot is told to quit.
        QUITER = true;
        par.addSubBehaviour(new sendQuit(myAgent));
      }
      msg = myAgent.receive();
    }
    block();
  }
}

class sendRobotHello extends OneShotBehaviour {
  public sendRobotHello(Agent a) {
    super(a);
  }

  public void action() {
    ListIterator friendsit = friends.listIterator();
    ACLMessage hiRobot = new ACLMessage(ACLMessage.CFP);
    while (friendsit.hasNext()) {
      AIDitem robot = (AIDitem)friendsit.next();
      hiRobot.addReceiver(robot.agent);
      hiRobot.setContent("Greetings!");
    }
    myAgent.send(hiRobot);
  }
}

class StateMachine extends OneShotBehaviour {
  public StateMachine(Agent a) {
    super(a);
  }

  public void action() {
    int oldstate = -1;
    while (oldstate != state) {
      oldstate = state;
      if (HALTER || RESET )
        state = 11;
      if (QUITER)
        state = 12;
      switch (state) {
```

```
            case 0: if (postFound) //Find the post
              state++;
            break;

            case 1: if (!postFound) //Centre in on the post and approch
it
              state = 0;
            else if (postCentered && postClose) {
              state++;
              par.addSubBehaviour(new sendReady(myAgent));
            }
            break;

            case 2: Iterator it = playFriends.iterator();
            AIDitem me = null;
            boolean friendsready = true;
            while (it.hasNext()) {
              me = (AIDitem)it.next();
              if (me.ready != true)
                friendsready = false;
            }
            if ((friendsready) && (ready))
              state++;
            break;

            case 3: newAngleFound = false;
            if (!postFound)
              state = 10;
            else if (postCentered && panAt90) {
              state++;
              degreeFound = false;
              par.addSubBehaviour(new sendDegree(myAgent));
            }
            break;

            case 4: newAngleRec = false;
            if (!postFound)
              state = 10;
            else if (newAngleFound) {
              newAngleFound = false;
              state = 3;
            }
            else if (degreeFound)
              state++;
            break;

            case 5: if (!postFound)
              state = 10;
            else if (newAngleRec) {
              newAngleRec = false;
              degreeFound = false;
              state = 4;
            }
            else if (newAngleFound) {
              newAngleFound = false;
              state = 3;
            }
```

```
else if (atBestCorner)
  state++;
break;

case 6: if (!postFound)
  state = 10;
else if (newAngleRec) {
  newAngleRec = false;
  degreeFound = false;
  state = 4;
}
else if (newAngleFound) {
  newAngleFound = false;
  state = 3;
}
else if (!atBestCorner)
  state--;
break;

case 10: par.addSubBehaviour(new sendHalt(myAgent));
state++;
break;


//Halt & Reset State
case 11:
  degreeFound = false;
  ready = false;
  gooddegree = -1;

  AIDitem me2 = null;
  Iterator it2 = friends.iterator();
  while (it2.hasNext()) {
    me2 = (AIDitem)it2.next();
    me2.ready = false;
    me2.degree = -1;
    me2.degreefound = false;
  }
  Iterator it3 = playFriends.iterator();
  while (it3.hasNext()) {
    me2 = (AIDitem)it3.next();
    me2.ready = false;
    me2.degree = -1;
    me2.degreefound = false;
  }

  if (!RESET)
    par.addSubBehaviour(new sendRobotHello(myAgent));
  HALTER = false;
  RESET = false;
  state = 0;
  break;

case 12:
  System.out.println("Robot has quit");
  break;
default:
```

```
            System.out.println("Not in normal state.");
            state = 10;
            break;

        }
        switch (state) {
          case 0: par.addSubBehaviour(new SetNewBehaviour(myAgent,
"findPost:0")); break;
          case 1: par.addSubBehaviour(new SetNewBehaviour(myAgent,
"approachPost:0")); break;
          case 3: par.addSubBehaviour(new SetNewBehaviour(myAgent,
"readyPivot:0")); break;
          case 4: par.addSubBehaviour(new findDegree(myAgent));  break;
          case 5: par.addSubBehaviour(new SetNewBehaviour(myAgent,
"Pivot:"+Integer.toString(gooddegree))); break;
          case 6: par.addSubBehaviour(new SetNewBehaviour(myAgent,
"onAlert:0")); break;
          case 12: par.addSubBehaviour(new SetNewBehaviour(myAgent,
"stop:0")); break;
          default: par.addSubBehaviour(new SetNewBehaviour(myAgent,
":0")); break;
        }  //Hey Darren, "Heh Heh, Alright."
        System.out.println("State = " + state + " Number of Robots = "
+ playFriends.size() + " GoodDegree = " + gooddegree);
      }
    }
  }

  class SetNewBehaviour extends OneShotBehaviour {
    private String behave;

    public SetNewBehaviour(Agent a, String beh) {
      super(a);
      behave = beh;
    }

    private ACLMessage msg = new ACLMessage(ACLMessage.INFORM);

    public void action() {
      msg.addReceiver(j2r);
      msg.setContent(behave);
      System.out.println("Setting Behaviour to" + behave);
      myAgent.send(msg);
    }
  }

  class sendHalt extends OneShotBehaviour {
    public sendHalt(Agent a) {
      super(a);
    }
    private ACLMessage halty = new ACLMessage(ACLMessage.FAILURE);

    public void action() {
      AIDitem me = null;
      Iterator it = friends.iterator();
      while (it.hasNext()) {
        me = (AIDitem)it.next();
```

```
        halty.addReceiver(me.agent);
      }
      myAgent.send(halty);
    }
  }

  class sendQuit extends OneShotBehaviour {
    public sendQuit(Agent a) {
      super(a);
    }
    private ACLMessage halty = new
ACLMessage(ACLMessage.REJECT_PROPOSAL);

    public void action() {
      AIDitem me = null;
      Iterator it = friends.iterator();
      while (it.hasNext()) {
        me = (AIDitem)it.next();
        halty.addReceiver(me.agent);
      }
      halty.addReceiver(Greeter);
      myAgent.send(halty);
      System.out.println("Robot has quit the system");
      while (true);
    }
  }

  class sendReady extends OneShotBehaviour {
    public sendReady(Agent a) {
      super(a);
    }
    private ACLMessage msg = new ACLMessage(ACLMessage.PROPOSE);

    public void action() {
      ready = true;
      AIDitem me = null;
      Iterator it = playFriends.iterator();
      while (it.hasNext()) {
        me = (AIDitem)it.next();
        me.degreefound = false;
        me.degree = -1;
        msg.addReceiver(me.agent);
      }
      myAgent.send(msg);
    }
  }

  class sendDegree extends OneShotBehaviour {
    public sendDegree(Agent a) {
      super(a);
    }
    private ACLMessage msg = new ACLMessage(ACLMessage.INFORM_REF);
    public void action() {
      AIDitem me = null;
      Iterator it = playFriends.iterator();
      while (it.hasNext()) {
        me = (AIDitem)it.next();
```

```
        msg.addReceiver(me.agent);
      }
      msg.setContent(Integer.toString(curdegree));
      System.out.println("SENDING DEGREES:  " + msg.getContent());
      System.out.println("CURRENT DEGREES:  " + curdegree);
      myAgent.send(msg);
    }
  }

  class findDegree extends OneShotBehaviour {
    public findDegree(Agent a) {
      super(a);
    }

    public void action() {
      int tempdegree = curdegree;  //this should lock the variable
curdegree for processing
      int numrobots = playFriends.size();

      boolean friendsready = true;
      Iterator it = playFriends.iterator();
      int i = -1;
      AIDitem me = null;
      while (it.hasNext()) {
        me = (AIDitem)it.next();
        i++;
        if (me.degreefound == false)
          friendsready = false;
        System.out.println(me.agent.getLocalName() + " has a degree
setting of " + me.degree);
      }


      if ((friendsready) && (degreeFound == false)) {
        //Insertion Sort
        //Aldash, how about some more white spirit?
        LinkedList ordfri = new LinkedList();
        ListIterator ito;
        ordfri.clear();
        AIDitem pme = new AIDitem();
        pme.degree = tempdegree;
        ordfri.addFirst(pme);
        it = playFriends.iterator();
        AIDitem meo = null;
        while (it.hasNext()) {
          me = (AIDitem)it.next();
          ito = ordfri.listIterator();
          boolean breakout = false;
          while ((ito.hasNext()) && (!breakout)) {
            meo = (AIDitem)ito.next();
            if (me.degree < meo.degree)
              breakout = true;
          }
          if (breakout)
            ito.previous();
          ito.add(me);
        }
```

```
        Iterator errit = ordfri.iterator();
        AIDitem errd = null;
        while (errit.hasNext()) {
          errd = (AIDitem)errit.next();
          if (errd.agent == null)
            System.out.println("Robot " + "me" + " has a degree setting
of " + errd.degree);
          else
            System.out.println("Robot " + errd.agent.getLocalName() + "
has a degree setting of " + errd.degree);
        }

        //Test the 2 solutions
        int div = (360/(numrobots+1));
        int test1 = 0;
        int test2 = 0;
        int tempy1 = 0;
        int tempy2 = 0;
        int c;
        ito = ordfri.listIterator();
        for (c=0;c<(numrobots+1);c++) {
          meo = (AIDitem)ito.next();
          tempy1 = meo.degree - c*div;
          tempy2 = meo.degree - (c+1)*div;
          if (meo.agent == null) {
            System.out.println("Robot " + "me" + " with a degree of " +
meo.degree + " going to " + c*div + " = " + tempy1);
            System.out.println("Robot " + "me" + " with a degree of " +
meo.degree + " going to " + (c+1)*div + " = " + tempy2);
          } else {
            System.out.println("Robot " + meo.agent.getLocalName() + "
with a degree of " + meo.degree + " going to " + c*div + " = " +
tempy1);
            System.out.println("Robot " + meo.agent.getLocalName() + "
with a degree of " + meo.degree + " going to " + (c+1)*div + " = " +
tempy2);
          }
          if (tempy1 < 0)
            tempy1 = tempy1 * (-1);
          if (tempy2 < 0)
            tempy2 = tempy2 * (-1);
          test1 = test1 + tempy1;
          test2 = test2 + tempy2;
        }
        System.out.println("Test 1: " + test1 + ", Test 2: " + test2);
        c = 0;
        if (test2 > test1)
          c = c + div;
        ListIterator ito2 = ordfri.listIterator();
        while (ito2.hasNext()) {
          meo = (AIDitem)ito2.next();
          c = c + div;
          if (c >= 360)
            c = 0;
          meo.goodDegree = c;
        }
```

```
        gooddegree = pme.goodDegree;

        errit = ordfri.iterator();
        errd = null;
        while (errit.hasNext()) {
          errd = (AIDitem)errit.next();
          if (errd.agent == null) {
            System.out.println("Robot " + "me" + " should go to " +
errd.goodDegree);
            gooddegree = errd.goodDegree;
          }
          else
            System.out.println("Robot " + errd.agent.getLocalName() + "
has a degree setting of " + errd.goodDegree);
        }
        System.out.println("Going to: " + gooddegree);
        degreeFound = true;
      }
      else
        degreeFound = false;
    }
  }

  class Subscribe extends OneShotBehaviour {
    public Subscribe(Agent a) {
      super(a);
    }

    private ACLMessage msg = new ACLMessage(ACLMessage.SUBSCRIBE);

    public void action() {
      msg.addReceiver(j2r);
      myAgent.send(msg);
    }
  }

  class Unsubscribe extends OneShotBehaviour {
    public Unsubscribe(Agent a, ACLMessage Mess) {
      super(a);
    }

    private ACLMessage msg = new ACLMessage(ACLMessage.CANCEL);

    public void action() {
      msg.addReceiver(j2r);
      myAgent.send(msg);
    }
  }
  //Maria, she's just this girl, you know?

  class UpdateVar extends OneShotBehaviour {
    private ACLMessage msg;
    public UpdateVar(Agent a, ACLMessage Mess) {
      super(a);
      msg = Mess;
    }
    public void action() {
```

```
      boolean temp;
      int tempdegree;
      String updates[] = msg.getContent().split("!");
      //Jessica & Noah - I'm actually done.  Really.  No Joke.
      for (int i=1; i <= Integer.parseInt(updates[0]); i++) {
        String var[] = updates[i].split(":",2);
        temp = false;
        if (var[1].compareTo("1") == 0)
          temp = true;
        //System.out.println(var[0] + " -- " + var[1] + " = " + temp);
        if (var[0].compareTo("postFound") == 0)
          postFound = temp;
        if (var[0].compareTo("postClose") == 0)
          postClose = temp;
        if (var[0].compareTo("postCentered") == 0)
          postCentered = temp;
        if (var[0].compareTo("panAt90") == 0)
          panAt90 = temp;
        if (var[0].compareTo("atBestCorner") == 0)
          atBestCorner = temp;
        if (var[0].compareTo("degree") == 0) {
          tempdegree = Integer.parseInt(var[1]);
          if (((curdegree - tempdegree) > 20) ||
             ((tempdegree - curdegree) > 20)) {
            newAngleFound = true;
          }
          curdegree = tempdegree;
        }
      }
    }
  }
}
```

Appendix B: Real-Time Experiment Source Code

This appendix contains the Java and C++ source code from the experiment conducted in section 5.2.

B.1 Cognitive Layer

```
/*
 * speedtest.java
 *
 * Created on August 27, 2004, 12:43 AM
*/

/**
 *
 * @author  Bram Gruneir
 */

import java.util.*;
import java.lang.*;
import java.io.*;
import java.net.*;

public class speedtest extends java.lang.Object {

  static Random r = new Random();
  static long ts;
  static long te;
  static long total;

  /** Creates a new instance of speedtest */
  public speedtest() {
  }

  /**
   * @param args the command line arguments
   */
  public static void main(String[] args) {
    System.out.println("Starting Communications Tester");
    r.setSeed(System.currentTimeMillis());
    System.out.println("The current time in ms is = " +
System.currentTimeMillis());

     if (args[0].equalsIgnoreCase("s")) {
      System.out.println("Short data");
```

```
      }
      else if (args[0].equalsIgnoreCase("m")) {
      System.out.println("Medium data");
      }
      else {
        System.out.println("Long data");
   }

    int max = 100;
    int maxTimes = Integer.parseInt(args[1]);

    System.out.println("Data sent:  " + (max*maxTimes));

    String[] s = new String[max];

    int c;
    int c2;

    for (int c3=0; c3<1; c3++) {
    for (c = 0; c < max; c++) {
     if (args[0].equalsIgnoreCase("s")) {
        s[c] = createshort();
      }
      else if (args[0].equalsIgnoreCase("m")) {
        s[c] = createmedium();
      }
      else {
        s[c] = createlong();
       }
    }

    //System.out.println("The Length of the string is:  " +
s[c].length());
    ts = System.currentTimeMillis();
    connectTCP(6004,6005);
    for (c = 0; c < maxTimes; c++) {
      for (c2 = 0; c2 < max; c2++){
        //sendOverFiles(s[c2], "/usr/tmp/outtest.txt",
"/usr/tmp/intest.txt");
        //sendOverUDP(s[c2], 6000, 6001);
        sendOverTCP(s[c2]);
      }
      //System.out.println(c);
    }
    disconnectTCP();
    te = System.currentTimeMillis();
    total = te - ts;
    System.out.println("The total number of ticks is:  " + total);
}
    System.exit(0);
  }

  /*simulates commands (5-10 characters)*/
  public static String createshort() {
    int l = r.nextInt(6) + 5;
    byte[] b = new byte[l];
    r.nextBytes(b);
```

```
  String test = new String(b,0,l);
  //System.out.println("-" + test + "-");
  return test;
}

/*variables updates (100-1000) characters*/
public static String createmedium() {
  int l = r.nextInt(901) + 100;
  byte[] b = new byte[l];
  r.nextBytes(b);
  String test = new String(b,0,l);
  //System.out.println("-" + test + "-");
  return test;
}

/*simulates a 320x200x256 size image - 64000 bytes*/
public static String createlong() {
  int l = 64000;
  byte[] b = new byte[l];
  r.nextBytes(b);
  String test = new String(b,0,l);
  //System.out.println("-" + test + "-");
  return test;
}

// 64000 byte limit on UDP!
public static void sendOverUDP(String s, int outPort, int inPort) {
  try {
    int l = s.length();
    /*Set to local host*/
    InetAddress addr = InetAddress.getByName("0.0.0.0");
    DatagramSocket din = new DatagramSocket(inPort,addr);
    DatagramSocket dout = new DatagramSocket();
    byte[] bout = s.getBytes();
    DatagramPacket pout = new DatagramPacket(bout,l,addr,outPort);
    dout.send(pout);

    /* This should block */
    byte[] bin = new byte[l];
    DatagramPacket pin = new DatagramPacket(bin,l);
    din.receive(pin);

    din.close();
    dout.close();
  }
  catch (java.net.SocketException e) {
  }
  catch (java.io.IOException e2) {
  }
}

public static ServerSocket tcpServSock;
public static Socket tcpOutSock;
public static Socket tcpInSock;
public static DataInputStream ins;
public static PrintStream ps;
```

```
public static void connectTCP(int outPort, int inPort) {
  try {
    /*Set to local host*/
    InetAddress addr = InetAddress.getByName("0.0.0.0");
    tcpServSock = new ServerSocket(inPort);
    tcpOutSock = new Socket(addr,outPort);
    tcpInSock = tcpServSock.accept();

    ps = new PrintStream(tcpOutSock.getOutputStream());
    ins = new DataInputStream(tcpInSock.getInputStream());
  }
  catch (java.net.SocketException e) {
  }
  catch (java.io.IOException e2) {
  }
}

public static void disconnectTCP() {
try {
  System.out.println("TCP Disconnect");
  tcpOutSock.close();
  tcpInSock.close();
  tcpServSock.close();
}
catch (java.net.SocketException e) {
  }
  catch (java.io.IOException e2) {
  }
}

// No limit on data, but must have handshakes
public static void sendOverTCP(String s) {
  try {
    int l = s.length();
    /*Set to local host*/
    //System.out.println("TCP");

    //Send data
    ps.print(s);

    //Get Data
    byte[] bin = new byte[l];
    int cur = 0;
    while (cur < l)
      cur = cur + ins.read(bin,0,l);
  }
  catch (java.net.SocketException e) {
  }
  catch (java.io.IOException e2) {
  }
}

public static void sendOverStub(String s) {
}

/*Possible future implementation*/
public static void sendOverMessaging(String s) {
```

```
  }

  /*Possible future implementation*/
  public static void sendOverSharedMemory(String s) {
  }

  public static void sendOverFiles(String s, String sendName, String
getName) {
    try {
      File outputFileTEMP = new File("/usr/tmp/tempjava.txt");
      File outputFile = new File(sendName);
      File inputFile = new File(getName);
      FileWriter out = new FileWriter(outputFileTEMP);

      out.write(s);
      out.close();
      while (!outputFileTEMP.renameTo(outputFile));


      int l = s.length();
      char[] ch = new char[l];
      //System.out.println("File " + sendName + " sent, waiting for
reply in "+ getName);
      while (!inputFile.exists());
      while (!inputFile.canRead());
      FileReader in = new FileReader(inputFile);
      in.read(ch);
      in.close();
      inputFile.delete();
    }
    catch (java.io.IOException e) {
    }
  }

  public static float testshort(int t) {
    return 0;
  }

  public static float testmedium(int t) {
    return 0;
  }

  public static float testlong(int t) {
    return 0;
  }
}
```

B.2 UDP Action Layer

```
#include <stdio.h>      /* for printf() and fprintf() */
#include <sys/socket.h> /* for socket() and bind() */
#include <arpa/inet.h>  /* for sockaddr_in and inet_ntoa() */
#include <stdlib.h>     /* for atoi() and exit() */
#include <string.h>     /* for memset() */
#include <unistd.h>     /* for close() */
```

```
#define ECHOMAX 255     /* Longest string to echo */

void DieWithError(char *errorMessage)
{  /* External error handling function */
      printf(errorMessage);
      exit (-1);
}

int main(int argc, char *argv[])
{
    int insock;                          /* Socket */
    struct sockaddr_in inechoServAddr; /* Local address */
    struct sockaddr_in remoteechoClntAddr; /* Client address */
    unsigned int remotecliAddrLen;         /* Length of incoming
message */
    char echoBuffer[ECHOMAX];        /* Buffer for echo string */
    unsigned short inechoServPort;     /* Server port */
    int recvMsgSize;                 /* Size of received message */

    int outsock;                          /* Socket */
    struct sockaddr_in outechoServAddr; /* Local address */
    unsigned short outechoServPort;     /* Server port */

    if (argc != 3)         /* Test for correct number of parameters */
    {
        fprintf(stderr,"Usage:  %s <UDP SERVER PORT> <UDP CLIENT
PORT>\n", argv[0]);
        exit(1);
    }

    inechoServPort = atoi(argv[1]);  /* First arg:  local port */
    outechoServPort = atoi(argv[2]);  /* First arg:  local port */

    /* Create socket for sending/receiving datagrams */
    if ((insock = socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP)) < 0)
        DieWithError("socket() failed");

    /* Construct local address structure */
    memset(&inechoServAddr, 0, sizeof(inechoServAddr));   /* Zero out
structure */
    inechoServAddr.sin_family = AF_INET;                /* Internet
address family */
    inechoServAddr.sin_addr.s_addr = htonl(INADDR_ANY); /* Any incoming
interface */
    inechoServAddr.sin_port = htons(inechoServPort);      /* Local port
*/

    memset(&outechoServAddr, 0, sizeof(outechoServAddr));
    outechoServAddr.sin_family = AF_INET;                /* Internet
address family */
    outechoServAddr.sin_addr.s_addr = htonl(INADDR_ANY); /* Any
incoming interface */
    outechoServAddr.sin_port = htons(outechoServPort);      /* Local
port */

    /* Bind to the local address */
```

```
    if (bind(insock, (struct sockaddr *) &inechoServAddr,
sizeof(inechoServAddr)) < 0)
        DieWithError("bind() failed");

    for (;;) /* Run forever */
    {
        /* Set the size of the in-out parameter */
        remotecliAddrLen = sizeof(remoteechoClntAddr);

        /* Block until receive message from a client */
        if ((recvMsgSize = recvfrom(insock, echoBuffer, ECHOMAX, 0,
            (struct sockaddr *) &remoteechoClntAddr,
&remotecliAddrLen)) < 0)
            DieWithError("recvfrom() failed");

        //printf("Handling client %s\n",
inet_ntoa(remoteechoClntAddr.sin_addr));

        /* Send received datagram back to the client */
        if (sendto(insock, echoBuffer, recvMsgSize, 0,
           (struct sockaddr *) &outechoServAddr,
sizeof(outechoServAddr)) != recvMsgSize)
          DieWithError("sendto() sent a different number of bytes than
expected");


    }
    /* NOT REACHED */
}
```

B.3 TCP Action Layer

This code could not be located at the time of printing. Please contact the author for more information.

B.4 File Action Layer

```
#include <stdio.h>      /* for printf() and fprintf() */
#include <stdlib.h>     /* for atoi() and exit() */
#include <string.h>     /* for memset() */
#include <unistd.h>     /* for close() */


int main(int argc, char *argv[])
{
  FILE *infile, *outfile;

  int temp;
  int eof;
  char buffer[64001];
```

```
  while (1)
  {
    while ((infile = fopen(argv[1],"r")) == NULL) {
    }
    outfile = fopen("/usr/tmp/tempc.txt","w+");
    eof = 0;
    while (feof(infile) == 0)
    {
       temp = fread(buffer,1,64000,infile);
       fwrite(buffer,1,temp,outfile);
    }
    fclose(infile);
    unlink(argv[1]);
    fclose(outfile);
    while (rename ("/usr/tmp/tempc.txt", argv[2]) != 0) {}
  }
}
```

Appendix C: Real-Time Experiment Table of Results

The following tables, Table C.1 to Table C.9 contain the results from the experiment conducted in section 5.2. The results obtained in each trial are in milliseconds.

Table C.1: UDP Short Results

Packets Sent	100	200	500	1000	2000	5000	10000	20000	50000	100000
Trial 1	79	109	195	345	627	1084	1775	3232	7536	14394
Trial 2	29	66	141	289	305	723	1458	2819	7107	13968
Trial 3	29	39	229	136	275	714	1426	2874	7096	13926
Trial 4	38	84	83	136	307	715	1359	2815	7140	13961
Trial 5	24	58	54	137	268	693	1405	2793	7099	13958
Trial 6	21	72	81	164	292	740	1429	2866	7095	13917
Trial 7	45	33	54	137	277	692	1395	2796	7138	13939
Trial 8	64	94	55	140	345	710	1413	2815	7095	13899
Trial 9	18	23	80	133	265	736	1413	2839	7101	13941
Trial 10	52	22	54	156	288	689	1369	2802	7143	13957

Table C.2: UDP Medium Results

Packets Sent	100	200	500	1000	2000	5000	10000	20000	50000	100000
Trial 1	99	132	252	430	760	1239	2103	3714	8875	20285
Trial 2	30	73	177	291	334	950	1686	3305	8272	16286
Trial 3	32	52	255	172	335	843	1655	3355	11289	26842
Trial 4	29	97	116	176	359	822	3232	3272	8303	16306
Trial 5	24	71	83	177	325	797	4228	3271	8350	16474
Trial 6	25	45	83	165	338	822	1658	4314	8566	16281
Trial 7	51	44	83	173	344	810	1657	6224	8909	25688
Trial 8	66	103	83	176	331	828	1751	3294	8586	19136
Trial 9	20	28	84	261	320	914	1675	3426	8299	16302
Trial 10	24	27	189	167	432	823	1642	3759	8372	16625

Table C.3: UDP Large Results

Packets Sent	100	200	500	1000	2000	5000	10000
Trial 1	526	993	2414	4758	9342	22466	44312
Trial 2	479	928	2354	4569	8732	21659	43201
Trial 3	464	920	2331	4332	8733	21712	42984
Trial 4	467	1082	2228	4344	8731	21565	42895
Trial 5	463	932	2168	4295	8615	21426	42958
Trial 6	476	936	2176	4350	8664	21598	43003
Trial 7	468	914	2164	4441	8646	21569	42985
Trial 8	485	922	2172	4327	8588	21474	43058
Trial 9	462	885	2160	4312	8620	21478	42968
Trial 10	475	892	2186	4351	8665	21619	42977

Table C.4: TCP Short Results

Packets Sent	100	200	500	1000	2000	5000	10000
Trial 1	1046	2066	5081	10085	25175	50085	185216
Trial 2	1074	2084	5082	10064	20089	85183	100094
Trial 3	1085	2085	5087	10093	20079	85403	100200
Trial 4	1084	2082	5089	10067	20081	85192	100081
Trial 5	1085	2088	5083	10075	20083	50093	185178

Table C.5: TCP Medium Results

Packets Sent	100	200	500	1000	2000	5000	10000
Trial 1	1069	2170	5089	10077	20080	50093	100095
Trial 2	1090	2176	5078	10087	20062	50076	100069
Trial 3	1085	2117	5087	10083	20105	50089	100095
Trial 4	1088	2089	5078	10063	20082	50189	100158
Trial 5	1090	2087	5082	10069	20087	50160	192660

Table C.6: TCP Large Results

Packets Sent	100	200	500	1000	2000	5000	10000
Trial 1	613	1179	2614	4947	9565	23302	46353
Trial 2	622	1166	2601	4888	9569	23461	46333
Trial 3	632	1186	2573	4936	9528	23236	46409
Trial 4	642	1194	2598	4908	9531	23203	46367
Trial 5	628	1176	2552	4902	9560	23295	46431

Table C.7: File Short Results

Packets Sent	100	200	500
Trial 1	10917	20157	53286
Trial 2	11515	21058	52186
Trial 3	10858	20619	51909
Trial 4	9718	21419	50518
Trial 5	10020	19918	51599
Trial 6	10241	19438	51418
Trial 7	10378	20638	49978
Trial 8	10139	20158	51958
Trial 9	9717	19980	50158
Trial 10	10978	20457	51597

Table C.8: File Medium Results

Packets Sent	100	200	500
Trial 1	11146	21880	52532
Trial 2	10791	22010	51951
Trial 3	10222	21455	52493
Trial 4	10344	20194	54714
Trial 5	11274	20892	55312
Trial 6	11326	19494	52074
Trial 7	10495	20623	52435
Trial 8	10615	20043	52194
Trial 9	10495	20874	53154
Trial 10	9712	21594	54353

Table C.9: File Large Results

Packets Sent	100	200	500
Trial 1	10851	21828	54484
Trial 2	11017	22232	54457
Trial 3	10594	21500	54637
Trial 4	10783	22222	54014
Trial 5	10873	21499	54367
Trial 6	10610	21673	54154
Trial 7	10425	21390	54460
Trial 8	10522	21544	53005
Trial 9	10649	21751	54014
Trial 10	10222	21632	53620

Appendix D: DVD Video of Implementation Experiment

Attached to the back cover of this thesis is a DVD video demonstrating the fully implemented architecture. The video is region free and will play in all DVD players that are compatible with the DVD-R format.

If this is an electronic copy of this thesis or if this video is damaged or missing, a new copy can be obtained by contacting the author of this thesis at bgruneir@alumni.uwaterloo.ca.

Some electronic versions of this thesis may have the video stored with it. To access it, try this link:

Architecture Demonstration Video