FLEXIBLE AUTONOMOUS ROBOTIC TASK SCHEDULING
USING ADVANCED RISC MACHINES

A Thesis by

Zachary James Zaccagni

Bachelor of General Studies, Wichita State University, 2004

Submitted to the Department of Computer Science
and the faculty of the Graduate School of
Wichita State University
In partial fulfillment of
the requirements for the degree of
Master of Science

August 2008
FLEXIBLE AUTONOMOUS ROBOTIC TASK SCHEDULING USING ADVANCED RISC MACHINES

I have examined the final copy of this thesis for form and content, and recommend that it be accepted in partial fulfillment of the requirement for the degree of Master of Science with a major in Computer Science.

_____________________________________________________
Rajiv Bagai, Committee Chair

We have read this thesis
And recommend its acceptance:

_____________________________________________________
Kamesh Namuduri, Committee Member

_____________________________________________________
M. Edwin Sawan, Committee Member
DEDICATION

To my empowering, loving wife Jennifer, and my beautiful children Zane and Zora
It is not always what we know or analyzed before we make a decision that makes it a great decision. It is what we do after we make the decision, to implement and execute it, that makes it a good decision. - William Pollard
ACKNOWLEDGMENTS

I would like to express my gratitude to my supervising advisor, Professor Kamesh Namuduri, for his guidance, advice and sharing of his expertise. His confidence and support encouraged me to expand beyond my knowledge in computer science both during our research at NASA and my thesis.

A special thanks goes to my long-term undergraduate instructor and recent colleague, Keenan Jackson, who inspired me to think beyond the programming. His instruction changed the way I program, and has enabled me to be a true computer scientist.

Thanks are also due to the chair of my committee, Professor Rajiv Bagai, who helped me hone my skills and find a focus for my thesis.

I would also like to extend my gratitude to the other member of my committee, Professor M. Edwin Sawan.

My experience at NASA working with Jon Patterson and his team was instrumental in inspiring me to pursue my thesis topic, and fundamentally transformed who I am as a researcher. My profound thanks to them.

Finally, I would like to say thank you to all my family and friends, particularly my fantastic wife, my soulmate, whose continued encouragement and sacrifices have enabled me to finish this degree. Thank you to my wonderful Mom, who had given me the tools and opportunities for learning while growing up, and for buying me my first computer. And thanks to my parents-in-law who have encouraged and supported my return to school.
ABSTRACT

This research presents two methods for a group of Garcia robots to collaboratively decide which task to attend to, and to move to their respective locations. One method allows for a needed flexibility and dynamic analysis in this distributed system of any number of robots coupled with any number of target locations, but is tied too closely to distance measurements. The other method is the implementation of Peter Molnar’s approach, which is free from any specificity for determining preferences, but is shown to have some other limitations. The packet loss problem inherent of broadcast communications is addressed, as well. The robots make decisions interdependently with the other robots after the initial setup from a host computer of common environment variables such as a map, robot and target locations.

A distributed network was established first for the robots so that information could be shared. Beacon messages were broadcasted at random intervals, while the robots handled TCP and UDP messages in separate threads. Once the needed data was acquired, each robot began their task scheduling decision-making process. If additional information was required from another robot still, a robot requested it. Our algorithm for task scheduling converged to an agreement rapidly, and resolved any possible gridlocks that occurred when two robots are exactly the same distance and with the highest preference for a single target. This method is found to be too closely tied into distance measures being used for preferences. Changes in the environment, such as new robots or targets, are handled during subsequent calls to decide for both methods. The result of this thesis shows two flexible and quick methods for task scheduling and assignment.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>I.</td>
<td>INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>II.</td>
<td>LITERATURE REVIEWED</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>Communication fault tolerance in distributed robotic systems</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>Constructing spanning trees for efficient multi-robot coverage</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>Scalable and reliable data delivery in mobile ad hoc sensor networks</td>
<td>8</td>
</tr>
<tr>
<td>III.</td>
<td>TOOLS, EQUIPMENT, AND SYSTEMS USED</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>Garcia robots</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>Processors and Memory</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>Odometry</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>Range Finders</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>Ledge Detectors</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>Stargate board</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>AmbiCom Wave2Net Wireless 802.11 Compact Flash Card</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>GNU ARM-Linux compilers</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>Various laptop computers</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>Wichita State University Computer Science network computers</td>
<td>13</td>
</tr>
<tr>
<td>IV.</td>
<td>APPROACH TO TASK SCHEDULING</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>Network communication and programming languages</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>Host application (AppController)</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>Purpose</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>Structures used</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>AppController menu</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>AppController tokens and related data</td>
<td>28</td>
</tr>
<tr>
<td></td>
<td>AppController UDP commands and purpose</td>
<td>28</td>
</tr>
<tr>
<td></td>
<td>AppController TCP commands and purpose</td>
<td>29</td>
</tr>
<tr>
<td></td>
<td>Robot application (AppServer)</td>
<td>29</td>
</tr>
<tr>
<td></td>
<td>General</td>
<td>29</td>
</tr>
<tr>
<td></td>
<td>AppServer tokens and related data</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>AppServer UDP commands and purpose</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>AppServer TCP commands and purpose</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>Steps to set up environment for robots</td>
<td>31</td>
</tr>
</tbody>
</table>
## TABLE OF CONTENTS (continued)

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task scheduling algorithm</td>
<td>32</td>
</tr>
<tr>
<td>Implementing the coupled selection preference</td>
<td>33</td>
</tr>
<tr>
<td>Implementing the coupled selection preference (attempt 1)</td>
<td>33</td>
</tr>
<tr>
<td>Implementing the coupled selection preference (attempt 2)</td>
<td>34</td>
</tr>
<tr>
<td>Implementing column decided task selection algorithm</td>
<td>36</td>
</tr>
<tr>
<td>Implementing the coupled selection preference (attempt 3)</td>
<td>41</td>
</tr>
<tr>
<td>Thoughts on consensus building using distributed and grouped robots to converge</td>
<td>43</td>
</tr>
<tr>
<td>Robot mobility</td>
<td>47</td>
</tr>
<tr>
<td>V. CONCLUSIONS AND FUTURE WORK</td>
<td>52</td>
</tr>
<tr>
<td>Conclusions from the research</td>
<td>52</td>
</tr>
<tr>
<td>Future work</td>
<td>53</td>
</tr>
<tr>
<td>REFERENCES</td>
<td>56</td>
</tr>
<tr>
<td>APPENDICES</td>
<td>58</td>
</tr>
<tr>
<td>A. CODE – APPCONTROLLER.CPP</td>
<td>59</td>
</tr>
<tr>
<td>B. CODE – APPCONTROLLER.H</td>
<td>88</td>
</tr>
<tr>
<td>C. CODE – APPSERVER.CPP</td>
<td>89</td>
</tr>
<tr>
<td>D. CODE – APPSERVER.H</td>
<td>132</td>
</tr>
<tr>
<td>E. CODE – MDEFS.H</td>
<td>133</td>
</tr>
<tr>
<td>F. CODE – COMMON_FNS.CPP</td>
<td>135</td>
</tr>
<tr>
<td>G. CODE – COMMON_FNS.H</td>
<td>140</td>
</tr>
<tr>
<td>H. Functions Used in My Selection Implementation</td>
<td>141</td>
</tr>
</tbody>
</table>
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Preferences each robot (A-J) develops for targets (0-9) through time</td>
<td>4</td>
</tr>
<tr>
<td>2. Direction vector to Target 1</td>
<td>5</td>
</tr>
<tr>
<td>3. Random interval broadcasts</td>
<td>6</td>
</tr>
<tr>
<td>4. Garcia robot facing right</td>
<td>11</td>
</tr>
<tr>
<td>5. Stargate processor and daughter card</td>
<td>12</td>
</tr>
<tr>
<td>6. Red robot acts as server</td>
<td>16</td>
</tr>
<tr>
<td>7. Yellow robot acts as client</td>
<td>16</td>
</tr>
<tr>
<td>8. October 8, 2007 the robots communicate for the second time</td>
<td>17</td>
</tr>
<tr>
<td>9. ServerUDP program running on two “robots”</td>
<td>19</td>
</tr>
<tr>
<td>10. LOCATION structure</td>
<td>22</td>
</tr>
<tr>
<td>11. ROBOT structure</td>
<td>23</td>
</tr>
<tr>
<td>12. Robot’s location is appended to its beacon</td>
<td>23</td>
</tr>
<tr>
<td>13. AppController Menu</td>
<td>24</td>
</tr>
<tr>
<td>14. Display robots in long format</td>
<td>24</td>
</tr>
<tr>
<td>15. Building a 10x8 map</td>
<td>26</td>
</tr>
<tr>
<td>16. Map drawn 10x8 with debug flag turn on with robots and targets shown</td>
<td>26</td>
</tr>
<tr>
<td>17. Setting robots’ locations</td>
<td>27</td>
</tr>
<tr>
<td>18. Setting targets’ locations</td>
<td>27</td>
</tr>
<tr>
<td>19. Initial preference equation of Molnar</td>
<td>34</td>
</tr>
<tr>
<td>20. Coupled selection equation of Molnar</td>
<td>34</td>
</tr>
<tr>
<td>21. 3-by-3 matrix finally converges, using Molnar’s equations</td>
<td>35</td>
</tr>
</tbody>
</table>
LIST OF FIGURES (continued)

22. Determining preferences for targets ........................................... 36
23. Column decided task selection algorithm ...................................... 37
24. Eight iterations to finalize decision ............................................... 40
25. Third attempt at Molnar’s approach .............................................. 43
26. Multiple robots moving towards their determined scheduled tasks .......... 44
27. Path that each robot will take to their respective target locations ........... 51
CHAPTER I
INTRODUCTION

There has been a considerable increase in the use of robots during the last few decades. Robots are pervasive in our current society, being used in manufacturing, scientific investigations, and law enforcement. During the last decade, remote-controlled robots have played a vital role in significant discoveries in terrestrial, oceanic, and space exploration. We have explored the Titanic, probed volcanoes, and scrutinized Mars at a depth unimaginable without the aid of these machines.

We will see a sharper expansion in robotic applications as our society ages, exploration expands, and the desire for menial tasks diminishes. Most current techniques that use remote-controlled robots will be done using autonomous agents instead, and many new ways to utilize them will emerge. Scientists, law enforcement, military, medical professionals, and our older citizens will benefit from advancements in self-directed robotics.

As an example, autonomous machines will allow for people to live independently longer. Robot companions that remind and deliver needed medicine, measure biostatistics (e.g. heart rate, blood pressure), and possibly hold conversations will significantly change both their stability and mobility.

Geologists and oceanologists will be capable of probing areas that are inhabitable, or extraordinarily dangerous, for humans. Deep caves and trenches, especially beyond the inaccessible reach of remote-control, can be investigated at safe distances. Mapping terrain, sampling rocks and ice-cores, and even experimentation can be
performed without constant attention, since these next-generation robots will navigate to a
destination unaided.

They will allow scientists to explore almost any celestial body from afar, while
possibly preparing a habitat for humans arriving. Currently, NASA (National Aeronautics
and Space Administration) has two rovers on Mars, Spirit and Opportunity. These rovers
are typically not as self-directed as scientists would like them to be. Though they are far
better at navigating themselves than the 1997 Sojourner rover, the process is still slow and
painstaking. More than roving across planetary planes, NASA plans to have space vehicles
autonomously land on the lunar and Martian surfaces. A base is to be constructed by
autonomous robots for eventual human habitat. These robots are expected to decide what
to do and navigate to their destination to accomplish the task that has been decided, while
avoiding obstacles, other robots, and ultimately people.

This research is a continuation of the experiences and strategy development at
Marshall Space Flight Center NASA for the 2006 Exploration Systems Summer Research
Opportunities (ESSRO). The focus is in on task/target scheduling based upon the robots’
preferences for each task/target, and having each robot move to their decided locations. In
this research, each robot’s preference for each target is determined solely on distance
between the two. Other factors might be added later if needed. Other approaches to task
scheduling were looked into. Some were found to be robust theories, but lacked the true
flexibility that we were looking for. Other approaches, like consensus building, are very
flexible, but difficult to implement in experimenting with a small group of robots. What
was found is a needed alteration of Peter Molnar’s robust simulated theories, while
maintaining the flexibility of having any number robots and target locations, and resolving
some issues that Molnar stated regarding communication. This led to a decision-making process that is highly scalable and dynamic.

The challenge faced was multifaceted. Foremost, communication needed to be established among the robots first. Then, it was determined what information was needed and how it was going to be communicated. Once each robot had this needed information, we undertook the task of figuring how it would decide what to do, if anything. Finally, this culminated in hooking into the robot’s API to have it move to its determined destination.

This thesis presents a method for a group of robots to collaboratively decide which task to attend to, and to move to their respective locations. This method allows for a needed flexibility and dynamic analysis in this distributed system of any number of robots coupled with any number of target locations. The packet loss problem inherent of broadcast communications is addressed, as well as other task scheduling algorithms and equations. When started, one obstacle that needed to be addressed was the lack of sensors that could be utilized by the robot’s program. A host program resolved this. It serves to set up the environment for the robots (e.g. delivering the map and each robot’s location on the map). It does not solve any problems or make any decisions for the robots. The robots make decisions interdependently with the other robots after the initial setup. The host program also monitors communications, and attempts to keep an updated mapping of the robots for human observation.

The remaining chapters of this thesis are organized as follows. Chapter II covers related work. Chapter III gives a brief overview of the tools, equipment, and systems used. Chapter IV describes my approach to task-scheduling in detail from communications to moving the robot. Chapter V discusses the conclusion as well as possible future work.
CHAPTER II
LITERATURE REVIEWED

Communication fault tolerance in distributed robotic systems [1]

Peter Molnar addresses target acquisitions by a group of robots through a task scheduling convergence equation. The performing of tasks at those locations/targets is superfluous to this paper. The goal is to assign one robot to each target, while minimizing each robot’s travel distance and overhead cost to their respective target. Essentially, the paper discusses an approach to combine the aspects both target assignment and motion planning to allow for each robot’s mobility while its final target is being decided.

A dynamic model of preferences and negotiation is used to determine the final target for each mobile robot. Each robot’s movement is governed by its ascertained part in an overall matrix (as shown in Figure 1) of all the robots involved. This “preference matrix” is comprised of each robot and their respective preferences for each target/location. The preference value ranges from 0.0 to 1.0, where a larger value represents a stronger preference towards that target/location. He vaguely describes a few potential cost-based motives that could be included in deciding a preference, such as proximity and suitability. This dynamic model is referred to as a self-organizing behavior-based navigation.

Figure 1. Preferences each robot (A-J) develops for targets (0-9) through time. [1]
Molnar and Starke performed a simulation based on the dynamic behavior of pedestrian crowds [2]. “…each robot is guided by a destination vector $e^0$ which finally points directly to the selected goal after” this selection process. This destination vector ($e^0$) is the linear combination of each direction vector to each target (shown in Figure 2).

![Figure 2. Direction Vector to Target 1. [1]](image)

Problem that may arise includes data collision in a noisy wireless UDP network. This can result in incorrect direction vectors, assignment of multiple robots to one target, and target under/overshoots due to missed timely data from the other robots. The paper suggests avoiding common data traffic peaks that may lead to package collision (and thus loss), by having the robots choose update intervals at random rates instead of fixed (Figure 3).

There are multiple unstated limitations of this approach. First, the targets’ locations must be known. Without the robot’s ability to measure the distance to each target, the matrix would be incomplete or jagged. It may be infeasible, if not impossible at times, to measure the needed distances. There would need to be some compensation for that, especially if one target is behind another. Second, since the targets are known, a general mapping must exist (e.g. a grid, or program representation of this mapping). Third, since the targets are known on a known grid, coupled with the difficulty in (possibly not even feasible) measuring distances between robots and targets, then the robots’ location must be known as well. We have a known represented grid, with known target locations and known robot locations. Only obstacles or breakdowns would change the initial preference
matrix. The algorithm based on the equations was extremely costly and took ten to twenty
times the iterations than the final algorithm to finally converge.

Figure 3. Random interval broadcasts.

**Constructing spanning trees for efficient multi-robot coverage** [7]

This paper addresses the problem of efficiently covering an area using multiple
robots in an assumed distributed network. An example the paper cites would be covering
an entire area of terrain in the most minimum time possible. Their research is based upon
Gabriely and Rimon’s methods for a single robot using a Spanning Tree Coverage algorithm
(STC) [8], and expanded for use by multiple robots.

The spanning tree is simply the paths that the robot(s) take to cover the area
split into N number of cells. The spanning tree of a single robot system consists of a path to
cover every cell in the area. In this case, it takes N time to cover N cells. How the path,
spanning tree, is constructed is inconsequential to coverage time, since it is always N time.
Using multiple robots to cover the same N-cell area can see significant improvement in
time, given that the coverage is shared.

How the tree is constructed can change the resulting times it takes for a multi-
robot system to cover the area. Without even changing the locations of any robot’s
location, the spanning tree paths can be decided having each robot’s initial starting cell anywhere along the path. Essentially, even when the physical locations of the robots have them grouped together, the spanning tree may show them spread virtually throughout the tree. Their paper concentrates on a way to create these spanning trees for the most efficient coverage times, compared to the trees which are simply generated randomly.

They show a theoretical optimal coverage time for \( k \) number of robots covering \( N \) number of cells as the ceiling of \( N/k \). This time occurs when the initial distance between robots in the spanning tree is at most the ceiling of \( N/k \) for each robot. What they attempted to prove is that changing the spanning tree for optimal coverage time will be significantly better than solely using an algorithm that blindly follows a base initial spanning tree. Some spanning tree paths in a multiple robot system are better for coverage times than others. So they suggest analyzing multiple tree paths for best coverage time, instead of simply going with the first generated. The researchers also ascertained that the initial virtual placement of each robot on the spanning tree, with their distance being at most \( N/k \) between them, results in the best coverage times. So, their objective was to create a spanning tree contraction procedure that produces a tree with the robots within that maximal distance from each other. This would distribute the robots initial virtual locations evenly along the path, as much as possible. Compare to a related approach, called optimal multi-robot spanning tree coverage (Opt_MSTC) by Hazon and Kaminka[10], the time to cover an \( N \) cell grid ranges from \( N-k+1 \) to \( N/k \). Whereas the Opt_MTSC’s time for coverage ranges from \( N-k+1 \) to \( \left(\frac{N-k}{2}\right)+1 \). Not only did they prove that their new methods improve the coverage time over other multi-robot approaches, it was also shown that obstacles placed on the grid impacted the time insignificantly most of the trials. In fact
they noticed that occasionally, having obstacles would improve the coverage time due to their being less cells to cover.

**Scalable and reliable data delivery in mobile ad hoc sensor networks [9]**

This paper proposes an algorithm for disseminating information in a mobile ad-hoc sensor network, where each agent’s routing decisions are based on its own knowledge. The purpose was to show how a controlled delivery process can significantly improve a convergence of data even when conditions are such that delivery is unsure and the data is limited from each node in the distributed network.

They suggest that data delivery through an uncontrolled manner in a distributed network can lead to data loss and hence has a detrimental effect on any consensus that needs to occur. The performance of agents/nodes in the network is negatively impacted without assurance of accurate and/or complete data. The problem the researchers address is how to arrive at an optimal controlled data delivery system in a distributed network to achieve the best team coordination efforts.

Other methods mentioned consist of pooling data from specific or all sources into a sink node. The sink node receives data from multiple sources along with information of the best routing paths, only after it sends a query to those sources. This is considered a reactive routing protocol that this team suggests is not a good approach to mobile sensor networks. This approach relies upon stationary sensors, where locations are stable. In a mobile sensor network, this is obviously not true, and according to the team leads to heavy traffic and long latencies in gathering data into the sink. They propose a new scalable approach for mobile sensors that are directly unaware of most other sensors in the network. Instead of querying for data, each agent/node passes its data to its neighbors,
blind of any network topology. Where the data is passed is based solely on its own localized information; meaning, what it personally senses or knows about. Each node maintains a spanning tree of historical routing paths to ensure a high probability for any relevant data needed is delivered for an eventual convergence. By using its self-constructed routing spanning tree, data traverses a seemingly random network in a controlled manner. This approach also addresses issues of communication failure of sensors.

The paper discusses three algorithms. Random walk simply has each sensor randomly picking a neighbor to deliver its data to, then that node repeating that with also the forwarded data, until a consensus has been determined. The path reinforcement algorithm handles data deliverance of relevant data once it sees that it has already forwarded certain information before to a neighbor. The path learning algorithm is similar to random walk, but some serious issues that they address. A sensor does not ensure that the data its forwarding is actually received. This will most likely lead to significant failures in data convergence. They suggest that a way to resolve this is to check for failures, then retransmit to a different neighbor. This requires a form of a feedback system to gauge failures. The paper shows a new algorithm, and recommends each node tracking the reliability of transmissions to any neighbors it communicates with, and using that to reduce the number of times it needs to redeliver that data.

Their trials showed that this type of controlled data deliverance improves the time and probability for relevant data to converge in a mobile ad-hoc sensor network by two to five times.
CHAPTER III
TOOLS, EQUIPMENT, AND SYSTEMS USED

The research and development took place in two main systems, the computer science student network, and the robots’ network. The following is a list of tools, equipment, and systems that were used.

Garcia robots

The Garcia robot is a highly sophisticated machine produced by Acroname equipped with a Stargate board, Brainstem GP 2.0 and Brainstem Moto 1.0 processors. The following lists some of the more relevant specifications per Acroname [3]:

**Processors and memory**

Two separate 40MHz processors handle the robot’s functions. A BrainStem Moto 1.0 processor handles the motion control and several sensor inputs. A BrainStem GP 2.0 processor provides a serial interface, IR communication capability, and additional IO.

**Odometry**

The Garcia drive train produces 3648 encoder pulses per wheel revolution. This makes it possible to perform accurate turns and travel precise distances.

**Range finders**

Garcia has six IR range finders. They provide valid distance measurements in a range of 4 to 18 inches. These sensors enable the robot to wall-follow or detect obstacles while maneuvering. When not in use, pairs of sensors can be disabled to save battery power.
**Ledge detectors**

Under the front end, there are left and right floor proximity detectors. These sensors can tell the robot if it is about to roll over a ledge. If your robot is moving slowly toward a ledge, it can stop its wheels and skid to a stop before plummeting. Of course, if your robot is going too fast it will plummet anyway. Use caution near ledges.

![Garcia robot facing right](image)

**Figure 4.** Garcia robot facing right. [3]

**Stargate board**

The Crossbow Stargate is a single board computer with a 400Mhz XScale RISC processor, 64MB of RAM, 1 Type II Compact Flash slot, and 1 PCMCIA slot. With embedded Linux, and provision for wireless sensor networks, this is where the programming is stored and run. It has a Linux kernel 2.4.19 as its core and supports 802.11 and Bluetooth protocols. The XScale microprocessor is a 32-bit implementation of the fifth generation...
ARM architecture, with 32KB instruction cache. Programs written for the Stargate need to be compiled using the GNU ARM-Linux cross-compilers.

![Figure 5. Stargate processor and daughter card. [4]](image)

**AmiCom Wave2Net Wireless 802.11 Compact Flash Card**

This card plugs into the available PCMCIA slot on the Stargate to provide short-to-medium range high-speed wireless communication.

**GNU ARM-Linux compilers**

The Stargate system requires programs be compiled for the ARM architecture, since that is what the microprocessor is. Both arm-linux-gcc and arm-linux-g++ version 3.3.2 were used to compile the software that runs on the robots. This depended upon when in the development it was at. The host software (named AppController) was compiled on two different servers using gcc and g++ version 4.1.2. The robot’s software (named AppServer) was compiled using g++ version 4.1.2 while testing the network communication and decision algorithm.
Various laptop computers

Several different laptop computers were used to run the host software on throughout the development. The most recent laptop is a Hewlett Packard DV2700 with 2 GB of RAM, and up to 799 MB of shared video memory. It runs Ubuntu Linux in a Virtual Box on Windows Vista. Ubuntu has both the latest arm-linux compilers, which are too new for the Stargate, and version 3.3.2. The system connects wirelessly to the robots’ network and allows for secure shell tunneling into each robot. This permits movement of the software to the robot and sending commands to run it from within Ubuntu, which is running as a virtual machine. The host software runs directly on the Ubuntu system, and is compiled with non-ARM g++.

Wichita State University Computer Science server network

Most of the network communication and task scheduling algorithms were tested on the servers in the Computer Science department at Wichita State University. This allowed the freedom to design and develop from anywhere that had an internet connection. Setting up five puTTY (secure shell clients), where one is the host and the other four are simulated robots, gave the ability to reproduce a scenario close to the physical robot setup at the school. According to Tom Wallis, the Computer Science System Administrator, the department has six semi-autonomous subnets in CS using managed switches and VLANS. The subnet for student machines (e.g. kira, kirk, sisko, spock) has real IP addresses and its own servers, with nearly all internet communication passing through their firewall/router. When logging in to an account, that server communicates with authentication, file, and mail servers giving you access. Tom Wallis states that the topology of the network VLANS is flat, and that any node on the student subnet can communicate to
any other node on the same subnet. With this, we can assume that he means that the student subnet and the faculty subnet each use a full mesh topology with an Ethernet switch between them.
CHAPTER IV
APPROACH TO TASK SCHEDULING

Network communication and programming languages

The foremost component of this research was getting the robots to communicate with each other. An example network communication program was found in the Stargate documentation. It was a very simple client-server example with two files written in C. Stargate documentation showed how to send and receive TCP packets to and from the robots. The CLIENT program simply sent an empty message to the SERVER running at the ip address specified in the makefile as HOST_IP_ADDRESS. The SERVER program would send a confirmation message back to the CLIENT once it received the empty message. This was a very limited illustration of what could be done.

Quickly, the code was cleaned up and enhanced for testing on the Garcia robots. The first changes made were moving the macro definitions (e.g. host ip, port) from Makefile to a separate header file (mdefs.h), along with the global buffer variable named buf. After cleaning up the code, msg_out string was added to CLIENT so that an actual message could be sent, and error checking to the TCP command accept(). Both programs were compiled using arm-linux-gcc and transferred to two robots. The red Garcia had the SERVER code, and the yellow Garcia had the CLIENT code. Our colleague Syam Kavala helped set up the robots for testing. That day, October 5 2007 at 2:23pm, was the first time in the department that the robots had communicated (as shown in Figure). The next step was to integrate both this SERVER and CLIENT into a single application, since the robots would need to send and receive information. This endeavor was abandoned due to concluding that mere TCP connections were not sufficient for what the robots were to do.
A connectionless paradigm was embraced using the User Datagram Protocol (UDP). This was not an area in which we had any experience. Significant amount of information and instruction had to be assimilated before it could truly utilize UDP instead of TCP. This rebuild led to two new applications. The broadcaster and listener applications were originally written by Brian "Beej Jorgensen" Hall [5], and altered somewhat for this system. The broadcaster would only broadcast UDP packets, and the listener would only listen for them. This set up was similar to the original Stargate example, except that it worked with UDP packets instead of TCP.

The transformation of the original SERVER application was the next step. The definition HOST_IP_ADDRESS was changed to BROADCAST_ADD, seeing as it was now going to handling broadcasted messages, instead of being connected to a specific node in the network. It was determined that this new server, named SERVER_UDP, should be capable of multitasking. This meant that it needed to be able to broadcast messages, receive messages, and handle any other tasks programmed without waiting. The recvfrom socket function blocks until it receives data, and the program needs to broadcast beacon messages at specified intervals. Remembering UNIX system programming, one way to
allow multitasking is to fork a new process. This technique will change in later iterations, but it worked fantastic for the moment. The program forked new processes to handle broadcast message requests. This decreased the chance of missing a broadcast by hand- ing off the message handling to a new process, while the parent returns to the top of a loop to receive a new broadcast. Before it entered this UDP message handling loop, another process was forked to handle the interval beacon broadcast sending. The program needed to broadcast beacons at specified intervals, regardless of whatever else the program is doing. At that time, it was simply a dummy broadcast, with everything in place to put in the sendto() code. An infinite loop was used to print to stdout the beacon message (with its own ip address and current time). At the end of the broadcast loop, it sleeps for a random time interval. Random interval was opted for due to suggestion by Peter Molnar to help prevent avoid data collisions [1]. The results of this build were that the serverUDP ran in an infinite loop, catching broadcasted messages and printing out details to stdout (e.g. ip of who’s sending it, time, message), and the udp_broadcast_msg() function randomly "broadcasts" its beacon to stdout every 25-45 seconds (as shown in Figure 8).

```
> broadcaster 156.26.10.236 hello
sent 5 bytes to 156.26.10.236
> broadcaster 255.255.255.255 "It is me over here"
sent 18 bytes to 255.255.255.255
***** BEACON BEING BROADCAST at Mon Oct 8 20:10:03 2
*****
got packet from 156.26.10.239
packet is 5 bytes long
packet contains "hello"

Figure 8. October 8, 2007 the robots communicate for the second time. broadcaster at top send two messages, and the server_UDP program handles the messages it hears.
In between those times of robot communication, the server_UDP application was written and tested on the Computer Science network servers. It still hadn’t decided on the best structure to store information from the other robots that it hears. What was next was to finally bring broadcasting and listening into a single application, and allow binding the program on another port simultaneously to allow for TCP connections as well as the connectionless UDP. It was also desired to develop a simple "main desktop control" interface so that command could be sent to the robots. The next build, on October 11, was yet to see the reintegration of the TCP protocol, or a desktop control application, though certain changes were made in preparation of TCP’s return (e.g. handle_client_request() name changed to handle_udp_client_request()). What was accomplished was having a single program (serverUDP) to send and receive broadcasts. There was no longer a need for two separately functioning applications. It now sent a beacon broadcast every 25-45 seconds over UDP (see Figure 9), and can ignore its own broadcasts (a macro definition, TRUE by default). Finally resolved the issue of finding its own ip address regardless of system the application is running on. It needed to know its own ip address to ignore its own broadcasts. It displayed received messages to stdout (another macro definition, TRUE by default), and managed all the forked processes more efficiently. The signal handler was defined for the CHLD signal from child processes that have ended. This signal handler calls manage_zombies(), which handles cleanup of any orphan child processes. The reason for this signal handler is when a child process dies, a signal SIGCHLD is sent to the parent process. The parent process must “catch” this signal to help avoid zombie processes that remain in memory. Since the child process is no longer needed, it needs to be caught / handled when ended and released from memory.
The next iteration of the robots’ server code added in the ability to listen for TCP messages. Putting the TCP related functions (e.g. `manage_tcp_connections()`) and the UDP related functions (e.g. `udp_broadcast_msg()`) into external source files helped organize the code much more effectively. The TCP handling function is forked into a new process at the beginning of the program, to maintain the ability to multitask. The rest of October 15 build continues the functionality of the previous.

A name change occurred with the subsequent build. The `SERVER_UDP` was renamed to `AppServer`, and the basic `CLIENT` from earlier was renamed `AppController`. The `AppController` is the host program in its infancy, with a simple menu that allowed only for asking for an ip and message to send. The final host program has an extensive menu system and will be discussed later in this thesis. Other than code being cleaned up, the `AppServer` is nearly identical to the prior build.
The next step was a major overhaul of the code. The decision to move from the C programming language to C++, and from forked processes to threads, was an immensely proper transformation. Threading innately allows for data to be shared and is much lighter weight memory-wise, compared to forked processes. Threads do not have an entirely separate memory address from its parent, and therefore consume fewer resources than a full-blown newly copied process. The threads in this application are used to manage the network communication similar to the earlier builds with forked processes. This frees main() up to handle further tasks beyond simply the network like deciding what to do. To reiterate the purpose of using threads, without threads the application would only broadcast UDP packets after something else had been done (i.e. only after receiving a UDP packet, or after the program calculates). The application was rewritten in C++ to allow easier integration with Garcia robot API. C++ is easier and more frequently used language than C, making it easier for others to expand upon what has been done. At first, C++ socket wrapper classes were being written from scratch, since C++ does not have any unique libraries for TCP and UDP socket programming. A programmer has to draw upon the same traditional C libraries. Jeff Donahoo’s Practical C++ Sockets web site was discovered, where he was providing his own C++ wrapper for sockets from Baylor University under a GNU General Public License [6]. The from-scratch wrappers were discarded in favor of Jeff’s. This thesis is about task scheduling and the information passed, not C++ socket wrappers, so it was not necessary to waste time re-inventing it. With the addition of these socket classes, the code became more readable compared to past C implementations. The result is a transformed C/forked process server application into a C++/threaded server application. What was left out of the conversion was the ability to receive and manage UDP requests.
The final build of AppServer had the network ability to send and receive TCP and UDP messages, while broadcasting a beacon message. The host application, AppController, also has the same network functionality, but for different purposes that will be discussed next.

**Host application (AppController)**

**Purpose**

The host application, AppController, started out as a simple interface for sending TCP messages to a specified ip address. It transformed into a menu-driven interface for setting up the needed environment variables for the robots, and maintaining certain information for the user to monitor. Due to the lack of sensors available for integration into the robots' AppServer, like the Cricket GPS sensors, the capability to set up a number of initial environment information was needed. The goal was to have the robots decide what task/target to do. The robots needed to be aware of a map, their own locations, the locations of the targets on that map, and what direction they were facing. The AppController initializes, addresses, and transmits all these variables to the robots. It does not decide anything for the robots, nor tell any robot the locations of any other robots; that's the work of AppServer application running on each robot.

The host keeps track of known robots the same way the robots' AppServer does by processing beacon broadcasts and storing the heard robot's information in a vector. Before adding it to the ROBOT vector, it ensures that the robot is unique to known robots. There are many error-checking mechanisms involved with the user input. Through the function ensureValidInt(), it double-checks that the user had actually entered an integer and that it is within range (e.g. location specified is not outside the current map). Only one
thread exists in the host application. It is spurred at the beginning to handle UDP messages and robots' beacons. Like the AppServer, it also ignores its own broadcast messages and adds new robots to the robots vector.

**Structures used**

The map is two-dimensional vector of locations. Each location structure consists of row and column data (see Figure 10). Also included is additional information, like a Boolean valued occupied variable, Boolean valued target, and the ip address of an occupying robot.

```c
typedef struct location_type
{
    int row, col;
    bool occupied; // occupied by robot/target?
    bool target;   // is the target here?
    string ip;     // if occupied by robot, which?
} LOCATION;
```

Figure 10. LOCATION structure

There are two uses for the ROBOT type (see Figure 11) in the implementation. A vector of ROBOTS is used to store a list of all known robots and their respective information that was sent. Robots broadcast a UDP beacon that is essentially an "I'm here" message containing their ip address. Once a robot has its location, it appends that information to the beacon as well (Figure 12).

An instantiation of a ROBOT named 'me' is used to store the information about that robot. This structure holds an ip address, name (e.g. host name), the robot's location, which direction it is facing, and two-dimensional vector of the robot's preferences. The prefs vector is filled at the beginning of task deciding for the robot, and also updated.
for the robots in the robot vector with the latest preference data transmitted. More will be covered about the preference vector later in this thesis.

```c
typedef struct robot_type
{
    string ip;
    string name;
    LOCATION location;  // where the robot is on the map
    string direction;   // direction facing
    vector< pair<float,string> > prefs; // robot's task/target preferences
} ROBOT;
```

**Figure 11. ROBOT structure**

```
****** BEACON BEING BROADCAST by me, kira (156.26.10.236)
Message Sent: BEACON|156.26.10.236

TCP Received message # 1 from: 156.26.10.239:42729
Message: Message: CONTROL|156.26.10.239|LOC|2|6|N
Received MY location: 2x6(N)
****** BEACON BEING BROADCAST by me, kira (156.26.10.236)
Message Sent: BEACON|156.26.10.236|LOC|2|6|N
```

**Figure 12. Robot's location is appended to its beacon**

**AppController menu**

The first thing that the user sees when starting the application, beyond the network information bubble, is the AppController's extensive menu and submenus (see Figure 13). The user can send TCP messages to a specific robot or to all robots. This functionality is both called directly by the user and via other functions for different purposes. The program lists all known robots, and prompts the user for both the message to send and which robot to send it to, unless they are sending to all known robots. If there are no robots known at the time, it notes that to the user and prevents sending a message. Of course, a person can broadcast a message to all robots. The difference in choice is
dependent on whether the message is absolutely vital for any/all robots to hear and react to. Since UDP is a connectionless protocol, there is no guarantee that anything else actually received the message. Unlike a TCP connection, there is no network hand-shaking happening; therefore, no capabilities innate to request for a damaged or loss packet again.

There are two ways of displaying the robots. One is a longer form than the other as shown in Figure 14. The longer format displays the name as well as the ip address, location, and direction that robot is facing in a multiple-line format. The short format restricts each known robot’s information to one line. They are called directly from the menu and called indirectly inside other functions.

![Figure 13. AppController Menu](image1)

![Figure 14. Display robots in long format](image2)
Building a map is essential to the set up of the data needed for deciding. Without the map created first, you are unable to set locations of targets and robots. Both programs, AppController and AppServer, create the map the same way. The map is a two-dimensional vector of LOCATION, and will be erased before each time you create a map. This allows for a change in map size, row by column, inside the program instead of needing to restart AppController or AppServer. Once the map has been created for the host computer it broadcasts that map so that the robots can build the map also. The map is erased first on the AppServer as well before building a new one. This does not affect the current robots’ and targets’ locations. If their locations are outside the scope of the new map, the user will need to reset new locations.

After building a map, the user can display it from the main menu. This is only used for human monitoring, since robots don’t need a visual reference. It draws out a row-by-column grid of the map, placing known targets and robots in their respective squares if locations are known also (see Figure 15, 16). Originally wanted to have two methods of showing the map, like the two ways to display robots, but found it was not necessary. The 'p' option has been left on the menu preceded with a comment marking, if someone wanted to implement it. The showMap() function pulls information anew every time it is called, allowing for changes in robot locations to update on the map.
Setting locations menu choice takes the user to a submenu of three options.

The locations of each known robot can be set, any number of targets can be set, or the user can simply return to the main menu. When setting for the robots, it displays a short list of known robots first, showing their current location (Figure 17). After each robot is set, a TCP message is sent to that robot regarding its new location. This is typically used for initializing and setting up the environment. When setting targets, the user can set any number of targets (Figure 18). It is important to know that this will erase all currently known targets before asking for the first target location. The targets are simple row by column locations and do not have a direction assigned them. The user is restricted to the placement of the targets to ensure that they are actually within the boundaries of the map.
The user can redisplay the network information that was shown at launch of the application. It shows the host name, ip address, TCP and UDP ports, and the UDP broadcast address.

![Map showing robot locations](image1)

**Figure 17. Setting robots’ locations**

![Target locations](image2)

**Figure 18. Setting targets’ locations**
AppController tokens and related data

The purpose of tokens is to classify the communication a robot is receiving. With such, it can determine better what to do with the message it is receiving. Otherwise, a complex guessing algorithm is needed to determine what the message is. Using tokens take out the guesswork and is a common approach in passing data.

There exist multiple classifications of the AppController's token commands. They first can be split between UDP and TCP types of tokens as shown in Figure). Then in finer detail, some commands can be made directly (e.g. BEGIN), others both directly and indirectly. An indirect token type is typically sent from functions that need to convey information to one/all robots after doing something. A good example of this is sending LOC. It conveys the location information about a specific robot to that robot. When used directly, it simply sends the location to the specified robot. Indirectly, it is used for the same purpose after a user sets a specific robot's location.

AppController UDP commands and purpose:

(direct and indirect)

MAP / map - used to send the map (or map size) to the robots.
This is sent automatically after building the map. User calling Map will send the map to all robots. Prerequisites: map built.

(direct)
BEGIN / begin - start organizing to acquire targets. Prerequisites: the map has been built, locations set
STOP / stop - stop organizing, stop movement, and wait
QUIT / quit - used to tell all robots to exit the program.
AppController TCP commands and purpose:

(direct and indirect)

LOC / loc - used to send a specified robot its location on the map.

Sent automatically after "set locations (robots/targets)"
menu option is chosen. User sending loc, only via sendTCP,
will simply send the location of specified robot only.

Prerequisites: map built, location(s) been set for robot(s).

TAR / tar - used to send a specific robot the locations of all targets.

Sent automatically after target locations have been set by
the user. User sending tar will simply send the location of
all target(s) to specific/all robots. Prerequisites: map
built, location(s) been set for target(s).

(direct)

MAP / map - used to send a specified robot the map (map size).

Prerequisites: the map has been built.

BEGIN / begin - start organizing to acquire targets.

Prerequisites: the map has been built, locations set.

STOP / stop - stop organizing, stop movement, and wait

QUIT / quit - used to tell specific/all robots to exit the program.

Robot application (AppServer)

General

The robot application, AppServer, is capable of handling and distinguishing UDP
and TCP messages from both the AppController (the host application) and other robots. If
the BEACON token is seen, the robot knows that the message is from a robot. Otherwise, the message is originating from the host. While waiting for task scheduling to start, the robots gather information from messages being broadcasted or data being sent to it from the AppController.

**AppServer tokens and related data**

Similar to the AppController, the tokens are divided into UDP and TCP types (see Figure). The main difference is that the indirect and direct tokens are discrete sets of commands, with exception to LOC. UDP broadcasts are used to send BEACON and typically LOC messages. The robot distinguishes between query token commands and information tokens. The robots are able to query other robots for information directly, if needed. They need to know the other known robots’ preferences before beginning the task scheduling/deciding and will connect to another robot through TCP to ask WHERE and WHAT, if location or preferences are unknown.

**AppServer UDP commands and purpose:**

(indirect)

- **BEACON** - irregular interval broadcast used to alert others on the network that the robot is "here" / present.

- **LOC** - alert other robots on the network of its current location. It is appended to the UDP BEACON message, once this robot’s location is known/told/determined.

**AppServer TCP commands and purpose:**

(direct query requests)

- **Q** - All TCP queries from a robot begin with Q| to signify a Query.
WHERE - ask specific robot where it is located.

WHAT - ask specific robot for its task/target preferences.

(direct information sent)

LOC - provides location of robot sending this command

PREF - provides task/target preferences of robot sending it

**Steps to set up environment for robots**

Before the robots can decide on which task/target location, they need information set up first. As mentioned previously, robots need to be told of a map, their own locations, and the locations of any targets. This initial setup is accomplished through the AppController application. Once starting the program, the user is restricted in what they can do. No robots are known at the start. The host application gathers together a list of known robots as it receives beacons.

A map can be built before knowing any robots, but then a MAP message would need to be sent, once they are known, to relay the map. This can be accomplished through TCP or UDP. Without a map, robots and targets cannot be assigned locations.

First, build a map by choosing the size of the map. The application will send the map to all known robots automagically. Next, the user wants to set the locations of the robots. It displays the map size for reference if needed. A row and column will need to be entered for their location, and then the direction they're facing (e.g. N S NE SW). If desired, the current robot that is being worked with can be skipped by typing `s` (sans quotes) at either the row or column prompt. After setting a new location of a robot, a LOC message is sent directly to that robot. The default location of a robot is 9999 for both row and column.
This signifies in the programming that no location has been set for that robot. When finished with the locations of all the robots, it will return to the Set Locations menu.

Now, the targets will be created by simply declaring their locations. In a similar manner as the robots, the user will enter a row and column of each target. As many targets as desired can be entered, typing `d` when done. Please note that the setting of target locations is destructive to the prior target list. Your new list of targets overwrites whatever target list existed up to that point.

At this point, a map has been created and sent to the robots, the robots have their own locations and the locations of all known targets. This is referred to as the initial setup of the environment. The robots are waiting for a signal token from the host to start the task scheduling algorithm.

**Task scheduling algorithm**

The core of this thesis is task scheduling. The method of deciding which robot should go to which target is the implementation of this algorithm solution. There are a number of factors to consider in this decision. It should be flexible to handle any number of robots and any number of targets. It needed the capability to resolve any conflicts such as when two robots are the exact same distance from a target. A convergence to a conclusion of pairings must be reached relatively quickly. There also may be circumstances where there are more robots than targets, so the decision methods must handle and consider what to do with extra robots. Each robot’s decision must concur with the others. That is the basic framework of requirements for a decision-making algorithm. The preferences are based on only the distance to the targets. The preference matrix is a two-dimensional
vector of robots and their respective preferences. Visually, this would be a robot-by-target graph, with the targets along the top and robots down the side as shown in Figure 1.

**Implementing the coupled selection preference vector (attempt 1)**

First, the equations used in Peter Molnar's paper [1] for task scheduling were implemented. He had an equation for the initial preference vector (Figure 19), and an equation for convergence of a final preference matrix with all robots' preferences (Figure 20). At first, the kappa variable was of unknown value, so an arbitrary value was picked. The methods to converge seemed only to work for square matrices (e.g. 2 robots and 2 targets), and even then the kappa variable needed different values for differently sized matrices. After multiple attempts to find a pattern and tie kappa's value into some other varying value, like the number of robots, Molnar's email provided information that kappa was tied into an entirely separate motion equation. Having searched for and found that equation mentioned, it was discovered that this equation factors in prior knowledge of all obstacles on the map in its computation. The robots' prior knowledge of the obstacles, like they know of the targets, was not wanted in this implementation. The concept of this thesis is to have as little of information necessary to make these decisions. If specific sensors are integrated into the robot and application, the host application can be removed entirely. Though an obstacle avoidance system had not implemented, the desire was for the robots to eventually handle obstacles dynamically. So, back to finding some obstacle-independent pattern for kappa's value it went.

Each robot has a preference vector that holds values from 0.0 to 1.0 (a percentage value) for each target. A zero is the least preferred target, and a one is the highest preference value. Once a robot has received all the preference vectors from all
known robots, it then computes a larger two-dimensional preference matrix. Every cell in
the preference matrix needs to converge to converge to 0 or 1, and at most one 1 per robot,
for this to be successful. There ended up being three slightly different approaches to the
convergence equation (Figure 20). The first one computed each cell in the matrix based
upon the other values in its row and column. Each cell’s preference value between 0.0 and
1.0 was replaced with the new preference value. The next cell would then compute its new
preference value based on the other values in its row and column, including the previous
cell’s new value. It would continue doing this, going cell by cell, until every cell’s value
converged to 0 or 1. Basically, once it received the other robots’ preference vector, it would
attempt convergence of the entire matrix at the exclusion of the other robots (i.e. there
would be no further updates from the others). This would lead to a convergence half the
time with 3 robots by 3 targets, but only occasionally would all three robots concur in the

\[
\xi_{ij}(0) = 1 - \frac{||r_i(0) - g_j||}{\max_{i',j'} (||r_{i'}(0) - g_{j'}||)}.
\]

Figure 19. Initial preference equation of Molnar [1].
i represents a row, j represents a column

\[
\dot{\xi}_{ij} = \kappa \xi_{ij} \left( 1 - \xi_{ij}^2 - \beta \sum_{i' \neq i} \xi_{i'j}^2 - \beta \sum_{j' \neq j} \xi_{ij'}^2 \right)
\]

Figure 20. Coupled selection equation of Molnar [1].
i represents a row, j represents a column

**Implementing the coupled selection preference vector (attempt 2)**

The second approach was very similar to the first. It would loop through every
cell like the first approach, but the values in its row and column, which it used for
computation, originated from the original values instead of the new values. After computation, the new preference value for that cell was placed into a different temporary matrix. After it had looped through every cell, before it started a new iteration through the matrix, the old preference matrix was replaced with the matrix with the new values. With this approach, the same 50/50 convergence was achieved, but now had all three robots agreeing on their conclusion. Still did not work as well as wanted, and occasionally took 300-400 full matrix computing iterations to achieve convergence when it did (Figure 21). Other times, two of three would converge, while one got stuck between two numbers.
A solution could not be found for the flexibility strived for, but it did finally get convergence only for 2-by-2, 2-by-3, 3-by-3, and 3-by-2 setups using the second approach. Molnar’s approach was abandoned at this point for a new one.

**Implementing column decided task selection algorithm**

The preference vector for each robot is determined by its distance to the target. To resolve any possible conflicts of two robots having the same distance to a specific target, a random millionth was added to the distance. So, if the distance computed is 5.0, the new
distance with the added random value might be 5.0043. The preference vector sent to the other robots consists of pairings of target numbers with float distances to those targets ordered by highest to lowest preference (see Figure 22). The preference matrix, which holds the preferences of all the robots, is maintained in the same manner as mentioned above.

![Figure 22. Determining preferences for targets](image)

This method still uses ones to signify highest preference, and in the final iteration the robots that still have a one move to that target (Figure 23). Preferences in the preference matrix hold integer values to represent the highest preference (one) to the lowest preference (number of targets). Each robot maintains its own preference matrix, its own preference vector, and the preference vector of each of the other robots. So, it has access to the data necessary for any conflict resolution.
necessary = (rows-cols <= 0) ? rows : cols;

While necessary $\neq$ of decided (rows/cols) has not been reached
  Look at col (j)
    If there are NOT any 1's, continue to next col
    Else
      If there is only one 1
        Mark col (j) as decided,
        Increment decided,
        Move to next col.
      Else
        Decide which robot rows (i) claims that target
        Determine closest row (i) of all rows with 1,
        Claim row (i) that is closest, as decided,
        Increment decided,
        Decrement the prefs of other rows, having 1, by one,
        Example: Consider 3 cols, therefore prefs of 1, 2 or 3.
        1 becomes 0
        2 becomes 1
        3 becomes 2
        Mark col (j) as decided,
        Move to next col.
  While end.

Figure 23. Column decided task selection algorithm

The robot waits for the BEGIN command from the host computer in order to start the task scheduling procedure. Once that occurs, it checks to ensure that it has the preference vector from each known robot via a call to the CheckRobotsPrefs(). If there are any known robots found missing this vital data, the robot asks for it through a TCP message. Regardless of what the queried robot is doing, it quickly responds to all TCP requests, given that the TCP handling function is inside a separate thread. Until it is aware of the preferences of all known robots, it pauses/sleeps. This communication process happens very swiftly, so the pause is nearly insignificant.

A copy of all the robots' information and target locations is created at the beginning of the task decision. This is to maintain consistency through the process of deciding. New robots, targets, or other factors should not be allowed to interfere with the current decision. The copied data prevents this. Since it converges and decides so quickly,
concerns about the need to permit changes is irrelevant. Any changes will be a part of the
next decision-making process. This is a robust and flexible application, especially due to its
speedy processing.

The process of deciding what robots will do which task/targets begins. How it
determines whether it is done is by keeping track of the number of ones it needs. It
determines this before going into the loop. Due to this being flexible, it needs to handle
cases of differing number of robots and targets. If there are more robots than targets, then
the number of ones necessary is equal to the number of targets. Otherwise, the number of
ones necessary equals the number of robots. Then, the algorithm loops until all necessary
ones are found.

The procedure examines each column target in order (see Figure 24). If there
are not any ones present in that column, that means that no robot has selected that target
as its number one preference. It simply skips the column target, and goes on to the next,
wrapping around to the first column when necessary. If only one row (robot) has a one in
that column, it is considered a decided column and a decided variable is incremented.
When it finds a conflict in the column, multiple robots claiming it as their top preference, a
call to ResolveConflicts() is made.
ResolveConflicts() resolves the conflict of multiple rows vying for one as its preference. It does this by deciding which is closer, and then shifting the others’ preferences down (e.g. in a 3 target matrix, 1 becomes 0, 2 becomes 1, and 3 becomes 2). Only the other robot rows contending for one will be decremented, with the row
determined to be closest remaining unchanged (Figure 24). The decided variable is incremented by one. If the number of ones needed is now equal to the number of ones decided, this task scheduling decision is complete and the preferences of the losing robots are zeroed-out. This prevents any undoing of the decisions of previously decided columns. If a robot is to decide to do something differently after a column has been decided, it will happen at the next decision-making call.

Since that target column has been either decided or skipped, the next course of action is to move to the next column which may wrap back around to the first column again. As mentioned previously, it will continue this loop until has enough decided ones. Once that happens, the task becomes moving the robot to its destination.

**Implementing the coupled selection preference vector (attempt 3)**

Returning to Peter Molnar’s approach for a third time, it is believed that this third approach is the closest implementation of his convergence equation. After re-reading his paper [1] numerous times in a short time period, a different view of it emerged. It is similar to the first and second approach since they all are attempting to implement the same equation. Instead of computing a new preference value for every cell in the matrix, this only computes the values for its own preferences at each iteration. It still used the other values in its row and column. Along with this new view, thesis advisor Dr. Kamesh Namuduri pointed out a part of the equation that was not considered. For the other two attempts, a new preference value using the coupling selection equation was computed. He pointed out that it should be adding that computed value back into the current value. That made complete sense, considering the previous results. That was a simple change to plus-equal to the code, instead of equal. At each iteration, it would compute a new preference
vector for itself, send that data to the other robots, and then change its values in the full preference matrix. It would then wait for the others to send their newly computed preference vectors. Once preferences are known for all known robots, it would start the process again by computing its own preference vector based upon the other's values, and sending that data out. It would continue that loop until the entire preference matrix converged to 0 and 1's.

The last implementation successfully converged in the times reflected in the paper, as shown in Figure 25. The convergence performed precisely as described by Molnar, and worked for any NxN matrix of robots and targets. Though, it was still not as flexible as had hoped. Some issues encountered were due to the initial preferences. Since distance is used in determining initial setup, the preference for the farthest target was zero, causing problems for scenarios of only one target. It appeared to not handle robots with the same distance from a target, unless there was an offset of more targets. In both cases, the preference matrix would instantly stabilize to all zeros. The other problem arose from having more targets than robots. It successfully handled the reverse of that, with more robots than targets, just as Molnar's paper mentions. The kappa and beta values were both set to one, and this essentially made them insignificant in the calculations.
Thoughts on consensus building using distributed and grouped robots to converge.

Now that it has been shown that the implementation of Peter Molnar’s solution does converge for most cases, the exploration of Dr. Kamesh Namuduri’s, and his principal investigator (PI) colleagues’, proposal for consensus building was desired. They propose that we can extend Molnar’s approach by considering teams of robots which communicate through one or more virtual fusion centers (VFC). This would allow for an array of robots to share information for task scheduling, even when spaced far enough apart where direct communication is impossible. Their research derives from previous investigations into decision fusion in wireless sensor networks (WSN), and consensus building in cooperative...
control systems. Its purpose is to achieve a consensus on task management, similar to Molnar’s solution, using groups of robots indirectly connected via one or more VFCs.

Molnar’s methodology was to wait to receive a preference vector from every robot before aggregating the information into a preference matrix. At that point, a selection coupling algorithm would be applied to the individual robot’s preferences, utilizing the data of the other robots. Through numerous iterations of this, the preference matrix will stabilize by converging to zeros and ones, where a one signifies that robot’s scheduled task. Before convergence, each robot computes their initial preferences for each target, and begins moving generally towards the target for which they have the greatest preference. The own personal preference vector’s values change after each iteration of the selection coupling, which in turn changes the robot’s direction while moving. Most of the time, robots do not simply move directly to a decided target, but meander towards it due to its

Figure 26. Multiple robots moving towards their determined scheduled tasks [1]
preference coefficients being applied to its behavior (see Figure 1). Eventually, as mentioned above, the matrix stabilizes for each robot to definitively know their assigned task. This may signify a target location or for them to stop and idle.

Here are some ideas that were thought about regarding the implementation of consensus building. In this new proposal, the robots are split into groups. A group consists of two or more robots with one acting as a VFC. The VFC should have a distinguishing characteristic, such as the lowest ip address in the group. In addition to performing all the typical actions and thinking of a regular robot in the group, it also acts as a gateway for communicating data indirectly to-and-from the other robots in its group only. Since a robot may belong to any number of groups, and thus needs to communicate its preferences to each VFC of each group it belong to, the VFC will receive information about robots that are outside of its own group. It will then pass this additional data on to the robots. Any node/robot can become a VFC if necessary. This means that any robot is capable of taking over as a new VFC. This may occur when the distance of one of its VFCs is growing to be considered too far away, or to maintain a balance in number of team members. There could be a single or multiple reasons for such a switch.

Each robot needs to send both its own preferences, along with information about the other robots it knows about to each VFC of each group it belongs to. These two messages will need to be distinguished from each other, with the VFC then passing the data on. This allows for a VFC to learn of other nodes/robots/targets indirectly from another VFC via a shared member. There may not be any additional data.

Theoretically, there should be no reason for a VFC not to act as the other do as well. The VFC ought to calculate its own preferences and be a part of the entire task scheduling
process, since it is simply like any other member but with additional responsibilities. Of course, it must be explored whether these additional responsibilities could interfere with timely convergence. If it is determined that the extra processing negatively impacts both the VFC duties and its own typical member duties, then possibly it is better to have it act solely as a VFC. Another issue needing to be addressed is if the VFC is acting also as a mobile robot with a task to do, in regards to its own preference vector, the other nodes (all other members and VFCs) need to treat this VFC as just another robot in the task scheduling preference matrix. Otherwise, knowledge of the other VFCs is inconsequential.

Each node/robot needs to keep track of in which group(s) it belongs, along with a differentiated way of communicating to a specific VFC and nothing else. The structure holding these details should be capable of changing (e.g. grow when this member is added to a new group). Since nodes/robots may only communicate to a VFC in its group, according to the proposed research, it is absolutely necessary to discriminate the messages passed to-and-from them. Possibly approaches to manage this could be using the TCP protocol to connect to the VFC, or having each group designate a unique UDP port to bind on. This would need to be a dynamic binding, since robots can join and leave groups, and even possibly create their own group as a VFC.

Considering Molnar’s solution has the robots deciding and moving simultaneously, theoretically the addition of new robots, targets, and their respective preferences to a robot’s current preference matrix should have negligible negative impact. There are at least two different scenarios to look at. One is where a robot’s matrix has already converged. When new data is known, and then added, the robot should process the coupling selection anew using this latest sized matrix. One can hypothesize that any extra
targets, that were unknown before, will have little to no impact in changing the original scheduled task. If it did not know of them before, it is highly probable that these new targets are out of the range of action anyway. The other scenario is where the robot is in the middle of converging. Its preference vector still consists of numbers between 0.0 and 1.0. Adding the additional data may or may not affect the final decision. Consider the case where a target is known by one robot, but is unknown to its direct neighbor. The target may be out of sight or blocked from sensors of the neighbor robot. When it receives the additional information about that target, certain factors may influence its convergence towards the target. Think about if initial preferences are set using distance, and this previously hidden target is actually closer than any target location known prior. With that, the selection coupling is affected and may influence a change in the robot's movement direction. Both scenarios could be handled by computing an initial preference for any new targets, then by either including into the existing matrix, or by beginning the process anew. Further research and thought needs to go into which way to manage that. One could reason that the system will eventually and correctly converge for task scheduling when only partial information is known at the beginning, and additional information is added into the preference matrix as it is received from others.

**Robot mobility**

The most difficult part of moving the robot to its destination was an efficiently logical plan of action. There is a behavior queue used in the Garcia API that is used to order the actions and events given to a robot. As a base for the mobility part, an example application from Acroname [3] was used. For this, many of the advanced features were not needed, only a simple behavior implementation. Instead of moving the robot in a triangle,
it would be moving it to a specific location on its map. There were a couple issues that needed to be addressed.

It is desired to maintain the ability for the robot to convey its correct and current location and direction while moving. If it simply turns towards the target location and moves, there may be times that it overlaps multiple location (e.g. at the corner junction of four map cells). It was unsure how to effectively resolve that. A decision was made to restrict its turns to factors of 45 degrees (3.14159265 / 4 radians). Though it is less efficient than merely going directly to the target, this method helps eliminate location confusion. The map cells are square, so a 45 degree turn (which is actually a pivot on its center axis) aims it towards the center of a diagonally adjacent cell.

The robot must remain within the scope of the map, having it stop forward movement when its next move action will take it outside the map’s bounds. There may be times when it is possibly more cost-effective to get to the target location by going outside the borders of the map. Nevertheless, in most situations it is faster to stay inside, so that is how it was designed.

A separate movement queue was created to store an ordered list of actions to undertake. This queue, moveQ, is a vector containing pairs of actions and float values. The reason that a separate queue was wanted, from the robot’s internal behavior queue, is to have an established course of action before the robot implements it. When actions are queued into the behavior queue, it is implemented immediately in the order they are given. This way allowed the task decision and movement to be self-contained as a single feat. This will be discussed in the future work part of this thesis, regarding ways to interrupt the movement, if a new decision differs from the current one. If this had been written in the C
language, a link list based queue would have been used, since this requires a dynamic structure. In C++, vectors are very similar to arrays, but allow for a dynamic sizing that arrays innately can not do. A link list seemed excessive for what was needed, so a vector structure was chosen, seeing as traversing the vector or a link list would both have an O(n) runtime anyway.

Movement is fairly straightforward when the robots are located in the same row or column as the target location. It simply needs to turn towards the target and move there. When the robot’s and target’s location have differing rows and columns, then complexity sneaks into the method. The logic of actions to take was more complex due to the nature of a virtually inverted map coordinate system. It is more reflective of a spreadsheet grid which presents itself down and right instead of up and right. As an example, in a simple x-y mathematical coordinate system, y values (rows) get larger as it moves up the y-axis. In this implemented map, the row numbers get larger as it goes down (see figure). So, extra attention must be given when comparing location coordinates. The basis for creating such a map is due to how C++ implements two-dimensional vectors. A vector is similar to any array. With an array named ‘int myArray[10][5]’, you have created an array with ten rows of five integers, or fifty integers in total. myArray[0] is the first row of five integers. It is the same for vectors. In fact, vectors are basically arrays that can reallocate memory when it changes size. The vector structure has numerous other advantages and functionality over a straight array, similar to how string primitives compare to c-strings.
The move actions are placed into the moveQ paired with the computed distance to move in meters. The turn actions are paired with some factor of 45 degrees, in radians and can be positive or negative values depending on which direction the robot is to pivot.

It will first pivot towards the direction of the target location, facing the diagonally adjacent cell (see Figure 27). It will then move forward until reaching the target’s column or row, also stopping if out of bounds. Then, it pivots to move towards the target, similar to the above procedure. The last action it always performs is pivoting back to face North. North is an arbitrary direction at this point, related to the map’s virtual North (top of the map), and the algorithm always assumes that the robot is facing North at the start. That is a limitation that should be resolved in future work.
Figure 27. Path that each robot will take to their respective target locations.
CHAPTER V

CONCLUSIONS AND FUTURE WORK

This chapter is the summary of the research results. The conclusions from the results will be discussed and the ways that this thesis contributes to the knowledge of task scheduling.

Conclusions from the research

There are numerous approaches to task scheduling robots. The job to match robots to tasks/targets effectively is an important one. In Peter Molnar’s simulation, his paper introduced a robust manner through selection coupling. It was surprising that true convergence was not capable of being achieved consistently in a practical application using three different methodologies. The need to tie the variable kappa to a movement equation, which required prior knowledge of an array of known obstacles, is a rigid requirement. Tying convergence to known obstacles restricts the ability to remove a host computer for setting up the environment. Using the third technique of implementation, convergence only happened usually with small square matrices (same number of robots to targets), and still couldn’t handle a scenario with one target and multiple robots. Molnar’s approach needs to be revisited with changes.

The alternate column decided algorithm for task scheduling converged to an agreement rapidly. It is capable of handling any number of robots to any number of targets, with the robots in agreement. The possible gridlock that may occur when two robots have exactly same distance and highest preference for a single target has been resolved by adding a random minuscule value to each distance. Those distances are used is resolving any conflict. There are no set intervals for communication. The beacons are
broadcast at random intervals, and inter-robot queries and replies are obviously not tied to a specific time. This was done as a suggestion in Peter Molnar’s future work to help prevent packet collision and data loss. Once the robot has reached its determined location, it currently returns to a state of waiting until the AppController (the host computer) sends another BEGIN command. Regardless of what it is doing, even if in the task scheduling process or moving, the robot continues to receive and handle UDP and TCP messages. New information (e.g. a new robot or a location change for a target) will not affect the current task scheduling decision, for the reason that copies of the essential data are made and protected inside that procedure. Changes in the environment are handled during subsequent calls to decide. Due to it handling this so quickly, the changes in the robot’s behavior will be equally swift. This research has contributed a highly effective task scheduling algorithm, proven to be both flexible and quick in achieving a conclusion.

**Future work**

Though the column decided approach is both flexible and quick, there are some areas that should be addressed as future work. Some of these are purely extensions to my current procedures, and the others are possible changes over all to the approach. Further study into consensus building using Molnar’s work as a base is also an area deemed important for future work.

The first area that should be given great consideration is the disconnection of the host computer (AppController). In order to do this, multiple sensors would need to be integrated onto the robot and into the AppServer application. A GPS sensor network, like Cricket GPS, would be highly effective in getting and maintaining location information for each robot. The other sensors needed would be ones with map-building functionality, a digital compass, and a way to
sense targets and their locations. With those in place, there is no need for AppController for setting up the environment variables.

Furthermore, the current movement procedure waits for all of the behaviors queued to complete before returning back to main(). This could cause problems when the environment changes which causes a need for the robot to stop what it is doing. In this circumstance, the robot needs to stop, flush its behavior queue, and make a new decision based upon this new information. If that queue waiting is eliminated, it can return to main() to process task scheduling anew. By doing that, the robot could handle any possible changes like more/less robots or more/less target locations.

Once it completes moving, and the behavior queue is empty, it returns to main() and waits for another BEGIN command from AppController. It does this by resetting the Boolean begin variable to false. Removing that reset would cause the application to immediately process task scheduling again.

Another change that allows more flexibility would be removing any requirements for the robot to be facing North before beginning movement. This should help speed up the time it takes for a robot to reach its target, since it wouldn’t be wasting time orienting itself back to a start direction.

Future work that concentrates on real-time obstacle detection and avoidance may be appropriate for real world applications of task scheduling. Most practical employment would not have smooth obstacle-free surfaces, so I believe that research in this area is absolutely necessary.

Finally, Peter Molnar’s approach should be revisited with research in changing some areas. The selection coupling equation has a lot of promise, given that it seemed to work well in simulation and in my third attempt at implementation. If the movement
behavior equation of Molnar's could be merged back in, then it could be highly useful for algorithms involving consensus building.

As for consensus building, future work in comparing convergence correctness between waiting for all information and handling partial information over time should be looked at.
REFERENCES
LIST OF REFERENCES


APENDICES
APPENDIX A

CODE – APPCONTROLLER.CPP

/******************************************************************
* AppController                                                  *
*    Programmer: Zachary Zaccagni                                *
*                                                                *
*    BASIC abilities:   .                                        *
*      Sends and Receives UDP broadcast messages                 *
*      Sends and Receives TCP broadcast messages                 *
*      Handles requests (UDP and TCP) from AppServers (robots)    *
*                                                                *
* PURPOSE:   .                                                    *
*      Gather info about robots in area (running AppServer)       *
*      Build a map and Send the map to those robots               *
*      Send data regarding each robot’s location and orientation,*
*          in absence of sensors.                                *
*      Send data regarding task/target location(s),              *
*          in absence of sensors.                                *
*      Send command of BEGIN to all the robots, to commence       *
*          grouped self-organization to acquire tasks/targets.    *
*      When messages are sent, they are sent with a header of    *
*          who they are CONTROL|ip.add.res.s, with the msg at end*
*          e.g. CONTROL|156.123.232.34|begin                     *
*          This is for consistency, ease of access to source ip *
*                                                                *
* Special thanks to Jeff_Donahoo@Baylor.edu of                   *
*    Practical Sockets for providing a simple c++ wrapper for     *
*    sockets, along with examples.                                *
*                                                                *
******************************************************************/

#include <iostream>       // For cout, cerr
#include <sstream>         // For stringstream
#include <iomanip>         // For setw
#include <vector>
#include <cstdlib>        // For atoi()
#include <map>             // for mapping robots and targets to map
#include <pthread.h>       // For POSIX threads
#include <unistd.h>        // For sleep
#include <time.h>          // For time
APPENDIX A (continued)

// the following 6 are need for initialization of vars (get hostip, etc)
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#include "PracticalSocket.h"  // For Socket, ServerSocket, and SocketException
#include "mdefs.h"            // Defines, Macros - general for both apps
#include "common_fns.h"
#include "AppController.h"    // Defines, Macros, Declarations
using namespace std;

int main(int argc, char *argv[])
{
    pthread_t thread_UDP_Rec;          // Thread ID from pthread_create()
    char menu_choice;                           // For the menu options
    stringstream umsg;                         // used for sending msg
    ROBOTSMAP_DATA robotsAndMap;       // used to send info to functions
    ROBOT me;
    vector<ROBOT> robots;                          // list of known robots
    vector<vector<LOCATION> >map;     // X x Y map
    vector<LOCATION> targets;          // list of known targets
    _initializeHostVars(&me);              // initialize the host variables

    system("clear");
    _displayNetInfo();  // display information about the controller

    // Open UDP Socket
    UDPSocket udpSock((unsigned short) UDP_PORT); // open UDP socket

    // store socket information for easy passing and access
    robotsAndMap.udpSock = &udpSock;

    // store robot and map data for easy passing and access
    robotsAndMap.me =       &me;
    robotsAndMap.robots =   &robots;
    robotsAndMap.map =      &map;
    robotsAndMap.targets =  &targets;
// create UDP handling thread for incoming UDP msgs
if (pthread_create(&thread_UDP_Rec, NULL, UDPThread,(void *)&robotsAndMap)!=0)
{  cerr << "Unable to create UDP handling thread" << endl; exit(1); }

while ( menu_choice != 'x' )
{
    _displayMenu();
    menu_choice = getchar();
    getchar();

    switch (toupper(menu_choice))
    {
    case 'T': // send tcp msg to a specific robot
        //sendTCP(&robots);
        sendTCP(robotsAndMap);
        break;

    case 'U': // send udp msg to all robots
        sendUDP(robotsAndMap);
        break;

    case 'A': // send tcp msg to all robots
        sendTCP(robotsAndMap, 1);
        break;

    case 'R': // display known robots
        displayKnownRobots(&robots);
        break;

    case 'L': // display known robots
        displayKnownRobots(&robots, 1);
        break;

    case 'N':
        _displayNetInfo();
        break;
    }
case 'B': // build map
  umsg.str("");
  //if (map.size())
  if (buildMap(&map))
  {
    umsg << "MAP|" << map.size() << "|" << (map.at(0)).size();
    cout << "Broadcasting map " << map.size() << "x"
        << (map.at(0)).size() << endl;
    sendUDP(robotsAndMap, umsg.str());
  }
  break;

case 'S': // Set Locations
  setLocations(robotsAndMap);
  break;

case 'M': // show map
  system("clear");
  showMap(robotsAndMap);
  break;

case 'X': // exit the program
  break;

default:
  cout << "**Not a valid menu choice**\n" << endl;
} // end switch
  cout << "\n**Not a valid menu choice**\n" << endl;
} // end switch
  cout << "\n" << endl; // padding for menu
} // end while

return 0;
} // end main
/***************************************************************
* Function: _initializeHostVars
*    retrieve the host information (ip, name, etc...).
* Receives: ROBOT *me (ptr to controller’s info
*
* Returns: nothing
***************************************************************/
void _initializeHostVars(ROBOT *me)
{
    int retv;
    struct addrinfo hints, *ai = NULL, *aitop = NULL;
    int error, i;
    char ntop[NI_MAXHOST], strport[NI_MAXSERV];
    char udp_port[10];
    sprintf(udp_port, "%d", UDP_PORT);

    /***************************************************************
* get host name and address                                      *
****************************************************************/
    retv = gethostname ((char *)hostname, 199);

    // the following part that retrieves hostip/name modified from
    // David Reid http://www.beclan.org/code_view.php?id=19
    memset(&hints, 0, sizeof(hints));
    hints.ai_family = AF_INET;
    hints.ai_socktype = SOCK_STREAM;
    if ((error = getaddrinfo(hostname, udp_port, &hints, &aitop)) != 0)
    {  printf(": %.100s: %s", hostname, gai_strerror(error)); exit(-1); }
    for (ai = aitop, i = 1; ai; ai = ai->ai_next)
    {
        struct sockaddr_in *sa = (struct sockaddr_in*)ai->ai_addr;
        if (ai->ai_family != AF_INET && ai->ai_family != AF_INET6) {continue;}
        retv = strlen(inet_ntoa(sa->sin_addr));
        strncpy(hostip, inet_ntoa(sa->sin_addr), retv);
        hostip[retv] = '\0';    // force str termination
        (*me).ip = hostip;
        (*me).name = hostname;
        if (getnameinfo(ai->ai_addr, ai->ai_addrlen, ntop, sizeof(ntop),
            strport, sizeof(strport), NI_NUMERICHOST | NI_NUMERICSERV) != 0)
        {  printf("failed\n"); continue; }
    }
}
APPENDIX A (continued)

/***************************************************************
* Function: _displayNetInfo
*    displays the current network information
* Receives: nothing
*
* Returns: nothing
***************************************************************
void _displayNetInfo()
{
    cout << "\n\n\n\n\n\n" << "__________________________________________________\n"    \\
        << "|                Network Information              |
"    \\
        << "|================================================|
"    \\
    << endl;
    cout << " Ready to broadcast and receive packets." << endl;
    cout << " Running on Host Name: " << hostname << endl;
    cout << " With Address: " << hostip << endl;
    cout << " Broadcasting UDP on Port: " << BROADCAST_ADD << endl;
    cout << " UDP On Port: " << UDP_PORT << endl;
    cout << " TCP On Port: " << TCP_PORT << endl;
}

/***************************************************************
* Function: _displayMenu
*    displays the main menu for the controller
* Receives: nothing
*
* Returns: nothing
***************************************************************
void _displayMenu()
{
    cout << "Controller Menu" << endl;
    cout << "-----------------------------------" << endl;
    cout << " t -- send TCP message to a specific robot" << endl;
    cout << " u -- send UDP message to all robots" << endl;
    cout << " a -- send TCP message to all robots\n" << endl;
    cout << " --DISPLAY ROBOTS----------------" << endl;
    cout << " r -- display all known robots on network (long)" << endl;
    cout << " l -- list all known robots (short)\n" << endl;
    cout << " --MAP----------------" << endl;
}
APPENDIX A (continued)

```cpp
cout << " b -- build map" << endl;
cout << " m -- show map (with robots/targets)" << endl;
cout << " //p -- show map (with robots/targets and refresh)/" << endl;
cout << " s -- set locations (robots and/or targets)\n" << endl;
cout << " --OTHER----------------" << endl;
cout << " n -- display network information" << endl;
cout << " - -- --------------------" << endl << endl;
cout << " x -- exit controller program" << endl;
cout << "____________________________________" << endl;
cout << "Please select from above: ";
}
```

```cpp
/***************************************************************
* Function: *UDPThread
*    called in creation of the UDP thread. Listens for UDP
*    messages being broadcasted.
* Receives: robotsAndMap (ROBOTSMAP_DATA struct assigned in main)
* *
* * Returns: nothing
***************************************************************

void *UDPThread(void* robotsAndMap)
{
    // Guarantees that thread resources are deallocated upon return
    pthread_detach(pthread_self());

    try
    {
        char echoBuffer[BUFFER_SIZE];  // Buffer for rec
        int recvMsgSize;               // Size of received message
        string sourceAddress;          // Address of datagram source
        unsigned short sourcePort;     // Port of datagram source
        string sourceip;               // holds substr of ip address
        vector<string> tokens;         // holds tokens from msg
        int tokenCount;                // number of tokens in msg

        // recast to other than void
        ROBOTSMAP_DATA *ptrcast = (ROBOTSMAP_DATA*) robotsAndMap;
```
APPENDIX A (continued)

// infinite loop to listen and handle UDP broadcasting
// Once data is received, calls HandleUDPClient() to handle
// client request.
// Once that communication handling is finished, as indicated
// by HandleUDPClient() return, resets recStrLen.
// Infinite loop, it will return to wait for next data rcvd.
while(1) // loop forever
{
    // block and look for udp messages to receive
    recvMsgSize = ptrcast->udpSock->recvFrom(echoBuffer, BUFFER_SIZE,
        sourceAddress, sourcePort);
    if (recvMsgSize <= 0) // no bytes received or err in rec
        continue;

    tokens.clear();
    tokenCount = explodeStr(tokens, echoBuffer, '|');
    if (tokenCount)
        {
            sourceip.clear();
            sourceip = tokens[1];  // assign msg source's ip

            // tokens[0] could contain BEACON, LOC, etc
            // tokens[1] is the sender's ip address
            // tokens[2+] the message, if one exists
            // else recd some udp data
            if (IGNORE_SELF_BC) // ignore broadcasts from self
                {
                    if ((string)hostip != sourceip)
                        { UDPMsgHandler(tokens, ptrcast->robots); } 
                }
            else
                { UDPMsgHandler(tokens, ptrcast->robots); }
        }
    recvMsgSize = 0;
    bzero(echoBuffer, BUFFER_SIZE); // reset the buffer
} catch (SocketException &e)
    { cerr << e.what() << " UDP Thread" << endl; exit(1); }
return NULL;
}
/***************************************************************
* Function: UDPMsgHandler
*    UDP message handling function - handles UDP requests
* Receives: vector<string> tokens (message sent),
*    vector<ROBOT> *robots (the robots)
*
* Returns:  nothing
***************************************************************
*/
void UDPMsgHandler(vector<string> tokens, vector<ROBOT> *robots)
{
    bool exists = false;
    unsigned int curr;  // used for determining which robot is talking

    // check to see if robot has previously been added
    for (curr=0; curr<(*robots).size(); curr++)
    {
        if ((*robots).at(curr).ip == tokens[1]) // robot is in known robots
        {
            exists = true; break;
        }
    }

    if (!exists) // robot doesn't exist prior
    {
        if (DISPLAY_MSG_TO_CONSOLE)
        {
            cout << "Received packet from: |" << tokens[1] << "| To: |" << hostip << "|" << endl;
            cout << "Message: " << implodeStr(tokens, '|') << endl;
        }

        ROBOT tempRobot;
        LOCATION tempLoc = {9999,9999,false,false,tokens[1]};

        tempRobot.ip = tokens[1];
        tempRobot.name = "unknown";
        tempRobot.direction = "u";
        tempRobot.location = tempLoc;
        // add robot to vector list
        (*robots).push_back(tempRobot);
    }
}
// handle and update new robot location
if (tokens.size() >= 6 && tokens[2] == "LOC") {
    // change the msg-sending robot's location information
    (*robots).at(curr).location.row = strToInt(tokens[3]);
    (*robots).at(curr).location.col = strToInt(tokens[4]);
    (*robots).at(curr).direction = tokens[5];
}

('.')***************************************************************/
/* Function: sendUDP */
/*   sends UDP msg to all known robots */
/* Receives: ROBOTSMAP_DATA robotsAndMap */
/*    string msg (optional message to be sent) */
/* */
/* Returns: nothing */
/******************************************************************************/
void sendUDP(ROBOTSMAP_DATA robotsAndMap, const string msg) {
    string servAddress = BROADCAST_ADDR; // broadcast address
    string baseString = "CONTROL|" + (string)hostip + "|"; // all UDP have
    string msgString;
    int baseStringLength; // used in sendto function call

    cout << "\n\n\n\n\n\n"
         << "| SEND UDP msg to all robots |\n"
         << "|=================================================================================|\n" << endl;

    // first, check to see if any robots are known
    if ((*robotsAndMap.robots).size() == 0) {
        cout << "No robots known at this time" << endl;
    }
APPENDIX A (continued)

else
{
  if (msg.empty()) // get msg from user
  {
    // the msg
    cout << "There are " << (*robotsAndMap.robots).size() << " known robots" << endl;
    cout << "Please enter the message to send:" << endl;
    cin >> setw(BUFFER_SIZE - baseString.length()) >> msgString;
    cin.ignore(numeric_limits<streamsize>::max(),'
'); // discard xtr

    // handle special messages
    if (msgString == "quit") { strUpper(msgString); }
    else if (msgString == "begin") { strUpper(msgString); }
    else if (msgString == "stop") { strUpper(msgString); }

    else if (msgString == "map")
    {
      if ((*robotsAndMap.map).size())
      {
        stringstream umsg;
        umsg << "MAP|" << (*robotsAndMap.map).size() << "|
" << (("robotsAndMap.map).at(0)).size();
        msgString = umsg.str();
      }
      else
      {
        cout << "No valid map exists. You will need to build a map before sending a map" << endl; return;
      }
    }

    else if (msgString == "loc")
    {
      // do not allow sending LOC to all through this interface
      cout << "LOC not sent. To send each robot its own "
           << "LOC, please use the Set/Send Location interface. (note: you will need a map built "
           << "first)" << endl; return;
    }
}
APPENDIX A (continued)

    baseString.append(msgString);
    baseString.append(1,'\0');

} else  // use the msg sent to function
{
    baseString.append(msg);
    baseString.append(1,'\0');
}

baseStringLen = strlen(baseString.c_str()); // length of str broadcast

    // send the msg
    cout << "\n\n***** UDP MSG BEING BROADCAST by me, " << hostname
        << " (" << hostip << ")\n"
        << "Sent this message:" << baseString << endl;
    robotsAndMap.udpSock->sendTo(baseString.c_str(), baseStringLen,
                                 servAddress, (unsigned short) UDP_PORT);
    baseString.clear();
}
}

/*****************************************************************************/
*/ Function: displayKnownRobots
* displays the info on all known robots
* Receives: vector<ROBOT> *robots (the vector of robots)
*    int shortlist (whether to display long or short
*          listing. default is 0 or no)
*
* Returns: nothing
*/
void displayKnownRobots(vector<ROBOT> *robots, int shortlist)
{
    if (!shortlist) // not shortlist, so show header
        cout << "\n\n\n"  _______________________________________________________________________
        << "| DISPLAY all known robots |\n"
        << "|================================|=\n\n" << endl;
APPENDIX A (continued)

// first, check to see if any robots are known
if ((*robots).size() == 0)
{    cout << "\n\nNo robots known at this time"<< endl;}
else
{
    cout << "[[ " << (*robots).size() << " robot(s) known ]] \n" << endl;
    for (unsigned int i=0; i<(*robots).size(); i++)
    {
        if (shortlist) // show short listing of robots
        {
            cout << "[" << i << "] "
                 << "IP: " << (*robots).at(i).ip
                 << "  Loc: " << (*robots).at(i).location.row
                 << "  - " << (*robots).at(i).location.col
                 << "  (ROW: " << (*robots).at(i).location.row
                 << "  COL: " << (*robots).at(i).location.col << "]" << endl;
        }
        else
        {
            cout << "[" << i << "]\n
                IP Address     : " << (*robots).at(i).ip
                \n                Name          : " << (*robots).at(i).name
                \n                Direction Facing: " << (*robots).at(i).direction
                \n                Location      : " << (*robots).at(i).location.row
                \n                - " << (*robots).at(i).location.col
                \n                (ROW: " << (*robots).at(i).location.row
                \n                COL: " << (*robots).at(i).location.col << "]" << endl;
        }
    }
}
/***************************************************************
* Function: sendTCP
*    send tcp message to one specific robot or to all robots
* Receives: ROBOTSMAP_DATA robotsmap (ptrs to robots, map, etc)
*           int to_all (whether to send to one or all known
*                       robots. default is 0, or to one)
*           string msg (optional message to be sent)
* *
* Returns:  nothing
***************************************************************/
void sendTCP(ROBOTSMAP_DATA robotsmap, int to_all, string msg)
{
    string baseString = "CONTROL|" + (string)hostip + "|"; // base str to send
    string msgString;
    int baseStringLength; // Length of str to echo
    unsigned int which; // used as index of robot vector
    string which_str;   // index read from user

    unsigned int numOfRobots = (*robotsmap.robots).size();
    bool isValid = false;

    cout << "\n\n\n\n"
    << "|-----------------------------------------------|\n" \\
    << "|    SEND TCP msg to robot(s)      |\n" \\
    << "|======================================|\n" << endl;

    // first, check to see if any robots are known
    if (numOfRobots == 0)
    {   cout << "No robots known at this time" << endl; return; }

    else if (!to_all) // then need to select the robot
    {
        displayKnownRobots(robotsmap.robots, 1); // short listing of robots

        cout << "\nEnter index of robot to send message to (an integer)"
            << ", or x to return to the menu.\n"
        << "(e.g. enter 2   [2] == third robot): ";

        cin >> which_str;
        cin.ignore(numeric_limits<streamsize>::max(),"\n");

    }
APPENDIX A (continued)

do // loop until valid non-negative index within scope
{
    if (which_str == "x") return; // return to menu
    isValid = ensureValidInt(which_str, which, numOfRobots,
                          "Enter index of robot");
} while (!isValid);

} // end else if (!to_all)

if (msg.empty()) // get msg from user
{
    cout << "___________________________________________\n"
        << " Please enter the message to send\n"
        << "___________________________________________\n"
        << " Examples of what to send:\n"
        << " - begin (to start robot organization).\n"
        << " - stop (to stop robot organization).\n"
        << " - /*move/turn (move or turn robot)/.\n"
        << " - /*goto (tell to go to a space on map)/.\n"
        << " - quit (tell robot to shutdown program).\n"
        << " - or other message.\n"
        << "___________________________________________\n"
        << " ---> ";
    cin >> setw(BUFFER_SIZE - baseString.length()) >> msgString;
    cin.ignore(numeric_limits<streamsize>::max(),'
'); //rem xtr

    // handle special messages
    if (msgString == "quit") { strUpper(msgString); }
    else if (msgString == "begin") { strUpper(msgString); }
    else if (msgString == "stop") { strUpper(msgString); }
    else if (msgString == "map")
    {
        if ((*robotsmap.map).size())
        {
            stringstream umsg;
            umsg << "MAP" << (*robotsmap.map).size()
                 << "|" << ((*robotsmap.map).at(0)).size();
            msgString = umsg.str();
        }
APPENDIX A (continued)

else
{
    cout << "\n\nNo valid map exists. You will need to build" << " one before sending a map" << endl; return;
}
}
else if (msgString == "loc")
{
    if ((*robotsmap.map).size()) // check if map built
    {
        if (!to_all) // only if sending loc to indiv robot here
        {
            stringstream umsg;
            umsg << "LOC|" << (*robotsmap.robots)[which].location.row
                << "|" << (*robotsmap.robots)[which].location.col
                << "|" << (*robotsmap.robots)[which].direction;
            msgString = umsg.str();
        }
        else // do not allow sending LOC to all through this interface
        {
            cout << "\n\nLOC not sent. To send each robot its own "
                << "LOC, please use the Set/Send Location "
                << "interface." << endl; return;
        }
    }
    else
    {
        cout << "\n\nNo valid map exists. You will need to build" << " one before sending a location" << endl; return;
    }
}

    // create the msg to send
baseString.append(msgString);
baseString.append(1,\0);
}
else  // use the msg sent to function
{
    baseString.append(msg);
    baseString.append(1,\0);
}

baseStringLen = strlen(baseString.c_str()); // length of str
try
{
    if (!to_all) // send message to specified robot
    {

        // open a tcp socket with robot specified
        cout << "...contacting " << (*robotsmap.robots)[which].ip << ".";
        TCPSocket sock((*robotsmap.robots)[which].ip, TCP_PORT);
        cout << "OK\n" << endl;

        // Send the message
        cout << "...sending message to "
        << (*robotsmap.robots)[which].ip << "...";
        sock.send(baseString.c_str(), baseStringLen);
        cout << "OK\n" << endl;
    }
    else // send message to all robots known
    {
        // open a tcp socket with each robot in turn, send msg
        for (unsigned int which=0; which<numOfRobots; which++)
        {
            cout << "...contacting "
            << (*robotsmap.robots)[which].ip << "...";
            TCPSocket sock((*robotsmap.robots)[which].ip, TCP_PORT);
            cout << "OK\n" << endl;

            // Send the message
            cout << "...sending message to "
            << (*robotsmap.robots)[which].ip << "...";
            sock.send(baseString.c_str(), baseStringLen);
            cout << "OK\n" << endl;
        }
    }
}

} catch(SocketException &e) { cerr << e.what() << endl; }
/***************************************************************
* Function: ensureValidInt
*   ensures that the string sent to it is an integer of
*   a non-negative value, and lower than max sent it.
*   calls strToInt to convert from string to int.
* Receives: string &s (string to convert)
*           unsigned int &i (int to put it into)
*           unsigned int max (max value limit)
*           string msg (optional additional msg to display)
*   Returns: true (if valid), false (if not)
***************************************************************/

bool ensureValidInt(string &s, unsigned int &i, unsigned int max, string msg)
{
    bool isValid = true;
    if (!strToInt(s, i))
    {
        cout << "Whoah there! " << s 
        << " is not a non-negative integer. " 
        << " Please try again. \n" << msg 
        << " (a non-negative integer): ";
        cin >> s;
        cin.ignore(numeric_limits<streamsize>::max(),'\n');
        isValid = false;
    }
    // index specified within scope?
    if ((isValid) && (i >= max || i < 0))
    {
        cout << "Whoah there! " << s 
        << " is not within the scope. Please try again. \n" 
        << msg << " (a non-negative integer): ";
        cin >> s;
        cin.ignore(numeric_limits<streamsize>::max(),'\n');
        isValid = false;
    }
    return isValid;
}
/***************************************************************
* Function: buildMap
*    builds/rebuilds an rowXcol map of specified size, altering
*    the 2d vector map sent to it by reference
* Receives: vector< vector<LOCATION> >*map (the 2d map)
*
* Returns: true (if new map was built), false (if not)
***************************************************************/
bool buildMap(vector< vector<LOCATION> >*map)
{
    unsigned int rows, cols;
    string rows_str, cols_str; // index read from user
    bool isValid = false;

cout << "\n\n\n\n"
    << "| Build the map |
    << "|----------------|
    << "\n" << endl;

    // get number of rows
    cout << "\nEnter # of rows (an integer)"
        << ", or x to return to the menu: ";
    cin >> rows_str;
    cin.ignore(numeric_limits<streamsize>::max(),'
');
    do // loop until valid non-negative int within scope
    {
        if (rows_str == "x") return false; // return to menu
        isValid = ensureValidInt(rows_str, rows, MAX_ROWS_COLS,
                                  "Enter # of rows");
    }while (!isValid);

    // get number of cols
    isValid = false; // reset isValid
    cout << "\nEnter # of columns (an integer)"
        << ", or x to return to the menu: ";
    cin >> cols_str;
    cin.ignore(numeric_limits<streamsize>::max(),'
');
    do // loop until valid non-negative int within scope
    {
        if (cols_str == "x") return false; // return to menu
        isValid = ensureValidInt(cols_str, cols, MAX_ROWS_COLS,
                                 "Enter # of columns");
    }while (!isValid);
// create rowxcol map
cout << "\n\nCreating the map of size " << rows << " x " << cols << "...";

// reset map back to 0-0
for(unsigned int i=0; i<(*map).size();i++)
    {(*map).at(i)).clear();
    (*map).erase((*map).begin(),(*map).end());

// resize the map to new specified size
(*map).resize(rows, vector <LOCATION>(cols));

// display map here?
if (DISPLAY_MSG_DEBUG)
{
    cout << "\nmap size " << (*map).size() << endl;
    for(unsigned int i=0; i<(*map).size();i++)
        cout << " element size " << ((*map).at(i)).size() << endl;
    return true;
}

/****************************
* Function: showMap
*    displays a representation of the map for the observer
* Receives: ROBOTSMAP_DATA robotsAndMap (ptrs to map, robots, etc)
* *
* Returns: nothing
****************************/
void showMap(ROBOTSMAP_DATA robotsAndMap)
{
    map<string,string> mapping;
    mapRobotsTargets(robotsAndMap, mapping); // mapping robot/target-to-map
    stringstream strkey; // used for key in mapping of robots/targets
APPENDIX A (continued)

if ((*robotsAndMap.map).size())
{
    int rows = (*robotsAndMap.map).size(),
        cols = ((*robotsAndMap.map).at(0)).size();
    cout << "\nMap Size: " << rows << " X " << cols << endl;

    cout << setfill ('-') << setw (cols*3);
    cout << "-" << endl; // draws a line of proportional size
    cout << "   ";
    for (int j=0; j<cols; j++)
    {
        cout << setfill (' ') << setw (3) << j;
    }
    cout << endl;
    for (int i=0; i<rows; i++)
    {
        cout << setfill (' ') << setw (3) << i;
        for (int j=0; j<cols; j++)
        {
            strkey.str("\"");
            strkey << i << "-" << j;
            // check if location is occupied by robot or target
            // and display that on map (e.g. R1 or T1)
            if (mapping.find(strkey.str()) != mapping.end())
                cout << "|" << mapping[strkey.str()];
            else
                cout << "|  ";
        }
        cout << "|\n   ";
        cout << setfill ('-') << setw ((cols*3)+1);
        cout << "-" << endl; // draws a line of proportional size
    }
    else
    {
        cout << "\n\nNo valid map exists. You will need to build one"
             << " before a map can be shown" << endl;
    }
}
/***************************************************************/
* Function: mapRobotsTargets
* creates an associative container for robots or targets.
* the key is the location (e.g. 3-4), and value is r/t and
* associated robot/target # (e.g. r2)
* Receives: ROBOTSMAP_DATA robotsAndMap (ptrs to map, robots, etc)
*       map *mapping (ptr to mapping)
* *
* Returns: nothing
***************************************************************/
void mapRobotsTargets(ROBOTSMAP_DATA robotsAndMap, map<string,string>& mapping)
{
    // loop through robots and targets, mapping them
    int numOfTargets = (*robotsAndMap.targets).size(),
        numOfRobots  = (*robotsAndMap.robots).size();
    stringstream strkey, strval; // used in creating the key, value
    //map<string,string>::iterator it; // iterator for debugging

    if (numOfTargets > 0)
    {
        for (int current = 0; current < numOfTargets; current++)
        {
            strkey.str("" );
            strval.str("" );
            strkey << (*robotsAndMap.targets).at(current).row << ":" << 
                   (*robotsAndMap.targets).at(current).col;
            strval << "t" << current;

            mapping.insert (pair<string,string> (strkey.str(), strval.str()));
            if (DISPLAY_MSG_DEBUG)
            {
                cout << "Target at " << strkey.str() << " is " 
                     << mapping[strkey.str()] << endl;
            }
        } // end for loop
    }
}
if (numOfRobots > 0)
{
    for (int current = 0; current < numOfRobots; current++)
    {
        strkey.str("\n");
        strval.str("\n");
        strval << "r" << current;

        mapping.insert (pair<string,string> (strkey.str(), strval.str()));
        if (DISPLAY_MSG_DEBUG)
        {
            cout << "Robot at " << strkey.str() << " is " << mapping[strkey.str()] << endl;
        }
    } // end for loop
}

/***************************************************************
* Function: _displaySetLocMenu
*    displays the menu for setting locations of robots/targets
* Receives: nothing
*
* Returns: nothing
***************************************************************/
void _displaySetLocMenu()
{
    cout << "\n\n\n\n\n____________________________________________\n| Set Location(s) of Robots and/or Targets |
|==========================================|\n\n" r -- set robot(s) location(s)\n" t -- set target(s) locations(s)\n" (note: this clears all known targets)\n" //s -- send current locations to each robot/\n" (note: each robot will recv their loc)\n" x -- exit back to menu\n" Please select r, t or x: ";
}
/***************************************************************
* Function: setLocations
*   user choice to set locations for each robot or target.
*   calls setRobotLocations or setTargetLocations
* Receives: ROBOTSMAP_DATA robotsmap (ptrs to robots,map,targets)
*
* Returns: nothing
***************************************************************/

void setLocations(ROBOTSMAP_DATA robotsmap)
{
    string locmenu;
    unsigned int numOfRobots = (*robotsmap.robots).size();

    if (!(*robotsmap.map).size()) // check if map has been built
    {
        cout << "

        No valid map exists. You will need to build one before robots and targets can be assigned
        locations" << endl;
        return;
    }

    do
    {
        _displaySetLocMenu();
        cin >> locmenu;
        cin.ignore(numeric_limits<streamsize>::max(),'n');

        if (locmenu == "r") // set robot locations
        {
            if (numOfRobots)
            {
                cout << "
OK Let's set the locations of the robots.

        Directions: enter row and col coordinate for each robot.
You will see the current location of each robot, then will be asked to give a new ROW/COL coordinate.
(Type s to skip that robot at either prompt for ROW or COL)"
            };
        }
    }
cout << "Map's Size:" << (*robotsmap.map).size() 
   " x " << ((*robotsmap.map).at(0)).size() 
   " # of Robots: " << numOfRobots;

for (unsigned int i=0; i<numOfRobots; i++)
{
    setRobotLocation(robotsmap, i);
}
else
{
    cout << "\n\nNo robots known at this time" << endl;
}

}

else if (locmenu == "t") // set target locations
{
    cout << "\n\nOK Let's set the locations of the targets.\n\n" 
    "Directions: You will be prompted to give a ROW/COL " 
    "coordinate for each target.\n (or type d when done," 
    "to return to the menu)\n\n";
    setTargetLocation(robotsmap);
}

else if (locmenu == "s") // send location to each robot
{
    cout << "\n\nSending location to each robot is not yet" 
    " functional\nYou will need to send a TCP " 
    "message to each robot seperately with\n" 
    "using the loc command\n\n";
}
else
{
    cout << "\n\n**Not a valid menu choice**\n\n";
}

cout << "\n\n"; // padding for menu shown again
} while (locmenu != "x");
}
/***************************************************************
* Function: setRobotLocation
* user sets locations for each robot
* when location is set, sends info to robots.
* Receives: vector<ROBOT> *robots (ptr to robots, map)
*           unsigned int which (robot to set location)
*
* Returns: nothing
***************************************************************/
void setRobotLocation(ROBOTSMAP_DATA robotsmap, unsigned int which)
{
    unsigned int rows, cols;
    unsigned int mapX = (*robotsmap.map).size();         // rows
    unsigned int mapY = ((*robotsmap.map).at(0)).size(); // cols
    string rows_str, cols_str, direction;      // read from user
    string baseString = "CONTROL|" + (string)hostip + "|"; // base str to send
    bool isValid = false; // control for valid input

    cout << "\n[" << which << "] "
    << "IP: " << (*robotsmap.robots).at(which).ip
    << "-" << (*robotsmap.robots).at(which).location.col
    << " (facing " << (*robotsmap.robots).at(which).direction
    << "]" << endl;

    // get robot's new row coordinate
    cout << "\nROW (or s to skip):";
    cin >> rows_str;
    cin.ignore(numeric_limits<streamsize>::max(),'
');
    do // loop until valid non-negative int within scope
    {
        if (rows_str == "s") return; // skip this robot
        isValid = ensureValidInt(rows_str, rows, mapX, "ROW (or s to skip)");
    }while (!isValid);

    // get robot's new col coordinate
    isValid = false; // reset isValid
    cout << "\nCOL (or s to skip):";
    cin >> cols_str;
    cin.ignore(numeric_limits<streamsize>::max(),'
');
}
do // loop until valid non-negative int within scope
{
    if (cols_str == "s") return; // skip this robot
    isValid = ensureValidInt(cols_str, cols, mapY, "COL (or s to skip)");
}while (!isValid);

cout << "\nStarting Direction (n,s,e,w):";
cin >> direction;
cin.ignore(numeric_limits<streamsize>::max(),'\n');
strUpper(direction);

// set robot's new X Y coordinate location
(*robotsmap.robots).at(which).location.row = rows;
(*robotsmap.robots).at(which).location.col = cols;
(*robotsmap.robots).at(which).direction = direction;

try
{
    // create the msg to send
    stringstream umsg;
    umsg << "LOC|" << rows << "|" << cols << "|" << direction;
    baseString.append(umsg.str());
    baseString.append(1,'\0');
    TCPSocket sock((*robotsmap.robots)[which].ip, TCP_PORT);

    // send robot its new location
    cout << "...sending location to " << (*robotsmap.robots)[which].ip << "...";
    sock.send(baseString.c_str(), strlen(baseString.c_str()));
    cout << "...DONE";
} catch(SocketException &e) { cerr << e.what() << endl; }
}
/***************************************************************
* Function: setTargetLocation
*    user sets locations for each target.
*    when all locations are set, sends info to robots.
* Receives: vector<ROBOT> *robots (ptr to robots, map, targets)
*
* Returns: nothing
***************************************************************/
void setTargetLocation(ROBOTSMAP_DATA robotsmap)
{
    unsigned int rows, cols;
    unsigned int mapX = (*robotsmap.map).size(); // rows
    unsigned int mapY = ((*robotsmap.map).at(0)).size(); // cols
    int numOfTargets; // number of targets, used in creating msg
    string rows_str, cols_str, direction; // read from user
    string baseString = "CONTROL|" + (string)hostip + "|"; // base str to send
    stringstream umsg; // used to create the msg to send
    bool isValid = false; // control for valid input
    bool isDone = false; // control for if done
    LOCATION tempLoc;

    // begin by clearing all targets
    (*robotsmap.targets).erase((*robotsmap.targets).begin(),
        (*robotsmap.targets).end());

    while(!isDone) // loop to get each new target
    {
        cout << "Target [" << (*robotsmap.targets).size() << "] " << endl;
        // get target’s new ROW coordinate
        cout << "ROW (or d when done):"; cin >> rows_str;
        cin.ignore(numeric_limits<streamsize>::max(),’\n’);
        do // loop until valid non-negative int within scope
        {
            if (rows_str == "d") isDone = true; // done
            else
                isValid = ensureValidInt(rows_str, rows, mapX,
                        "ROW (or d when done)’);  
        }while (!isDone && !isValid);
    }
if (!isDone) // get target's new COL coordinate
{
    isValid = false; // reset isValid
    cout << "\nCOL (or d when done):";
    cin >> cols_str;
    cin.ignore(numeric_limits<streamsize>::max(),'\n');
    do // loop until valid non-negative int within scope
    {
        if (cols_str == "d") isDone = true; // done
        else
            isValid = ensureValidInt(cols_str, cols, mapY,
                "COL (or d when done)");
    }while (!isDone && !isValid);
}

if (!isDone) // add target to vector
{
    tempLoc.row = rows;
    tempLoc.col = cols;
    tempLoc.occupied = true;
    tempLoc.target = true;
    tempLoc.ip = "";
    // add target to vector list
    (*robotsmap.targets).push_back(tempLoc);
    isValid = false; // reset isValid
}

// create msg and call sendTCP
numOfTargets = (*robotsmap.targets).size();
if (numOfTargets > 0)
{
    umsg << "TAR|" << numOfTargets;
    for (int current = 0; current < numOfTargets; current++)
    {
        umsg << "|" << (*robotsmap.targets).at(current).row
        << "|" << (*robotsmap.targets).at(current).col;
    }
    cout << "...sending targets to all robots...";
    sendTCP(robotsmap, 1, umsg.str());
    cout << "DONE\n";
}
APPENDIX B

CODE – APPCONTROLLER.H

// contains declarations and macros for AppController

#ifndef APPCONTROLLER_H
#define APPCONTROLLER_H

/*=================================*/
| function declarations  |
/*=================================*/
void *UDPThread(void* robotsAndMap); // UDP thread handling function
void UDPMsgHandler(vector<string> tokens, vector<ROBOT> *robots);
void sendUDP(ROBOTSMAP_DATA robotsAndMap, const string msg=""); //send udp msg
void sendTCP(ROBOTSMAP_DATA robotsmap, int to_all=0, string msg="");

void displayKnownRobots(vector<ROBOT> *robots, int shortlist=0);
bool ensureValidInt(string &s, unsigned int &i, unsigned int max, string msg);

bool buildMap(vector<vector<LOCATION> >*map);
void showMap(ROBOTSMAP_DATA robotsAndMap);
void mapRobotsTargets(ROBOTSMAP_DATA robotsAndMap, map<string,string> &mapping);

void setLocations(ROBOTSMAP_DATA robotsmap);
void setRobotLocation(ROBOTSMAP_DATA robotsmap, unsigned int which);
void setTargetLocation(ROBOTSMAP_DATA robotsmap);

void _displayMenu();
void _displaySetLocMenu();

#endif
APPENDIX C

CODE – APPSERVER.H

/******************************************************************
* AppServer                                                      *
*    Programmer: Zachary Zaccagni                                *
*                                                                *
*    BASIC abilities:      .                                     *
*      Sends and Receives UDP broadcast messages                 *
*      Sends and Receives TCP broadcast messages                 *
*      Handles requests (UDP and TCP) from other                 *
*      AppServers (robots) and the AppController.                *
*      Sends UDP broadcast messages at specified intervals       *
*                                                                *
*    PURPOSE:      .                                             *
*      Gather info about other robots in area (running AppServer)*
*      Self-organize with those other robots to aquire           *
*        tasks/targets.                                        *
*                                                                *
*    Special thanks to Jeff_Donahoo@Baylor.edu of                *
*    Practical Sockets for providing a simple c++ wrapper for    *
*        sockets, along with examples.                           *
*                                                                *
******************************************************************/

#include <iostream>           // For cout, cerr
#include <sstream>            // For stringstream
#include <iomanip>            // For setw
#include <vector>
#include <set> // used in determining pref to targets
#include <list> // used in determining pref to targets
#include <cstdlib>            // For atoi()
#include <cmath>              // For sqrt(), pow()
#include <pthread.h>          // For POSIX threads
#include <unistd.h>           // For sleep
#include <time.h>             // For time

// the following 6 are need for initialization of vars (get hostip, etc)
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
```cpp
#include "PracticalSocket.h"  // For Socket, ServerSocket, and SocketException
#include "mdefs.h"            // Defines, Macros
#include "AppServer.h"        // common functions
#include "common_fns.h"       // used in instantiating Garcia robot
#include "acpGarcia.h"        // used in instantiating Garcia robot

int messageNum; // used in TCP msgs, if logs/msg keeping
bool begin;     // used in signaling BEGIN

int main(int argc, char *argv[]) {
    messageNum = 1;
    begin = false;
    ROBOT me;
    vector<ROBOT> robots;   // list of known robots
    vector<vector<LOCATION>> map;  // the map
    vector<LOCATION> targets;  // list of known targets

    ROBOTSMAP_DATA robotsAndMap;  // used to send info to functions

    pthread_t thread_TCP, thread_UDP_Rec;  // thread ID from pthread_create()
    pthread_t main_thr = pthread_self();  // if needed to be referenced

    robotsAndMap.me = &me;
    robotsAndMap.robots = &robots;
    robotsAndMap.map = &map;
    robotsAndMap.targets = &targets;

    _initializeHostVars(&me);  // initialize the host variables

    // defined globally
    squareSize = 0.5; // size of each side of each square on map
    diagDist = 1.41421356237 * squareSize; // distance from 1 sqr
    // to another diagonally
```
APPENDIX C (continued)

if (DISPLAY_MSG_TO_CONSOLE)
{
    cout << "Main thread = " << main_thr << endl;
    _displayNetInfo();
}

// Create TCP handling thread
if (pthread_create(&thread_TCP, NULL, TCPThread,
    (void *)&robotsAndMap) != 0)
{  cerr << "Unable to create TCP handling thread" << endl; exit(1); }

// Create UDP handling thread
if (pthread_create(&thread_UDP_Rec, NULL, UDPThread,
    (void *)&robotsAndMap) != 0)
{  cerr << "Unable to create UDP handling thread" << endl; exit(1); }

while (1) // Run forever - core decision making for task mgmnt
{
    if (begin) // start deciding preferences / organizing
    {
        // first, check if my loc && loc of other robots are known
        if (begin && me.location.row != 9999
            && CheckRobotsLocs(&robotsAndMap))
        {
            DetermineOwnPrefs(&robotsAndMap);
            SendOwnPrefs(&robotsAndMap);

            while (begin && !CheckRobotsPrefs(&robotsAndMap))
            {  sleep(1);  }
            // check to ensure that all prefs for all robots is known
            // before continuing on to the decision making
            if (begin)
            {
                // now it is time to decide which robot will
                // do what. Decision making time.
                TaskDecision(&robotsAndMap);

                ResetAllRobotsPrefs(&robotsAndMap, true);

                begin=false; // for now, run through only once.
                // needs a BEGIN to do it again.
            }
        }
    }
}
APPENDIX C (continued)

    sleep((rand()%5));     // temporary--used in debugging only
} // end if begin

return 0;// NOT REACHED
}

/******************************************************************************
* Function: HandleTCPClient
*    TCP message handling function - handles TCP messages
* Receives: TCPSocket *sock (tcp socket to recv from)
*    ROBOTSMAP_DATA *robotsAndMap (ptrs to robots and map
*
*    * Returns: nothing
*******************************************************************************/
void HandleTCPClient(TCPSocket *sock, ROBOTSMAP_DATA *robotsAndMap)
{
    extern int messageNum;
    stringstream tcpMsg;
    vector<string> tokens;
    char echoBuffer[BUFFER_SIZE];
bzero(echoBuffer, BUFFER_SIZE); // zero out the str
    int recvMsgSize;
    unsigned int curr; // used for determining which robot is talking
    bool robotExists = false;

    if (DISPLAY_MSG_TO_CONSOLE)
    {
        cout << "=== TCP RCVD (from: ";
        try
        {  cout << sock->getForeignAddress() << ":"; }
        catch (SocketException &e)
        {  cerr << "Unable to get foreign address" << endl; }
        try
        {  cout << sock->getForeignPort(); }
        catch (SocketException &e)
        {  cerr << "Unable to get foreign port" << endl; }
        cout << ")\n";
    }

    // receive msg until the end of transmission
    while ((recvMsgSize = sock->recv(echoBuffer, BUFFER_SIZE)) > 0)
    {   // Zero means end of transmission.
        tcpMsg << echoBuffer;
    }
}
tokens.clear();
explodeStr(tokens, tcpMsg.str().c_str(), '|');
if (DISPLAY_MSG_TO_CONSOLE)
{   cout << "=== WITH MSG #" << messageNum++
    << ": " << implodeStr(tokens, '|') << "\n\n"; }

// CONTROL: perform action / handle command if exists
if (tokens[0] == "CONTROL" && tokens.size() > 2)
{
    HandleControlMsg(tokens, robotsAndMap);
}

// Other Robots are communicating (WHERE, LOC, PREF, etc)
else if (tokens.size() > 2)
{
    // check to see which robot
    for (curr = 0; curr < (*robotsAndMap->robots).size(); curr++)
    {
        // check if robot is in known robots
        if (((*robotsAndMap->robots).at(curr).ip == tokens[1])
            {
                robotExists = true; break;
            }
    }

    if (!robotExists) // robot is new, add it
    {   addRobot(&tokens, robotsAndMap->robots); }

    HandleRobotMsg(tokens, robotsAndMap, curr);
}
void HandleUDPClient(vector<string> tokens, ROBOTSMAP_DATA *robotsAndMap)
{
    // tokens[0] could contain BEACON, LOC, etc
    // tokens[1] is the sender's ip address
    // tokens[2+] the message, if one exists
    bool robotExists = false;
    unsigned int curr; // used for determining which robot is talking

    // check to see if robot has previously been added
    for (curr=0; curr<(*robotsAndMap->robots).size(); curr++)
    {
        // check if robot is in known robots
        if ((*robotsAndMap->robots).at(curr).ip == tokens[1])
        {
            robotExists = true; break;
        }
    }

    if (!robotExists && tokens[0]!="CONTROL") // robot doesn't exist prior
    {
        addRobot(&tokens, robotsAndMap->robots);
    }

    if (DISPLAY_MSG_TO_CONSOLE)
    {
        cout << "+++ UDP RCVD (from: " << tokens[1] << " to: "
        << hostip << ")" << endl;
        cout << "+++ WITH MSG: " << implodeStr(tokens, '|') << 
        "\n\n";
    }

    // CONTROL: perform action / handle command if exists
    if (tokens[0]="CONTROL" && tokens.size() > 2)
    {
        HandleControlMsg(tokens, robotsAndMap);
    }
}
// Other Robots are communicating
else if (tokens.size() > 2)
{
    // handle robot commands -- updated locations, prefs for targets, etc
    HandleRobotMsg(tokens, robotsAndMap, curr);
}

/**************************************************************
* Function: TCPThread
*    called in creation of the TCP thread. Listens for TCP
*    messages being sent to it
* Receives: void* robotsAndMap (ptrs to robots and map)
* Returns:  nothing
***************************************************************/
void *TCPThread(void* robotsAndMap)
{
    // Guarantees that thread resources are deallocated upon return
    pthread_detach(pthread_self());

    try
    {
        TCPServerSocket servSock(TCP_PORT);    // Socket descriptor for server
        while(1) // loop forever
        {
            TCPSocket *clntSock = servSock.accept();
            HandleTCPClient((TCPSocket *) clntSock,
                             (ROBOTSMAP_DATA*) robotsAndMap);
        }
    } catch (SocketException &e)
    {
        cerr << e.what() << " TCP Thread" << endl; exit(1); }
    return NULL;
}
/***************************************************************
* Function: *UDPThread
*    called in creation of the UDP thread. Listens for UDP
*    messages being broadcasted and branches a thread for
*    the UDP broadcasting
* Receives: void* robotsAndMap (ptrs to robots and map)
*    
*    Returns: nothing
***************************************************************/
void *UDPThread(void* robotsAndMap)
{
    // Guarantees that thread resources are deallocated upon return
    pthread_detach(pthread_self());
    try
    {
        UDPSocket udpSock((unsigned short) UDP_PORT); // open UDP socket
        char echoBuffer[BUFFER_SIZE];     // Buffer for rec
        int recvMsgSize;                  // Size of received message
        string sourceAddress;             // Address of datagram source
        unsigned short sourcePort;        // Port of datagram source
        vector<string> tokens;            // holds tokens from msg
        int tokenCount;                   // number of tokens in msg
        pthread_t thread_UDP_Brd;         // thread for UDPBroadcast

        ROBOTSMAP_DATA *ptrcast = (ROBOTSMAP_DATA*) robotsAndMap;
        ptrcast->udpSock = &udpSock;

        // Create UDP Brodacasting thread
        if (pthread_create(&thread_UDP_Brd, NULL, UDPBroadcastThread,
                        (void *) robotsAndMap) != 0)
        { cerr << "Unable to create UDP BCast thread" << endl; exit(1); }

        // Infinite loop for server to listen and handle UDP broadcasting
        //  Once data is received, HANDLEUDPCLIENT() to handle client request.
        //  Once that communication handling is finished, as indicated
        //  by HANDLEUDPCLIENT() return, resets recStrLen.
        while(1) // loop forever
        {
            // block and look for udp messages to receive
            recvMsgSize = udpSock.recvFrom(echoBuffer, BUFFER_SIZE,
                                            sourceAddress, sourcePort);
            if (recvMsgSize <= 0) // no bytes received or err in rec
            { continue; }
    }
APPENDIX C (continued)

tokenCount = explodeStr(tokens, echoBuffer, '|');
if (tokenCount)
{
    // recd some udp data. handle it
    if (IGNORE_SELF_BC) // ignore broadcasts from self
    {
        if ((string)hostip != tokens[1])
        {
            HandleUDPClient(tokens, (ROBOTSMAP_DATA*) robotsAndMap);
        }
    }
    else
    {
        HandleUDPClient(tokens, (ROBOTSMAP_DATA*) robotsAndMap);
    }
    tokens.clear();
    recvMsgSize = 0;
    bzero(echoBuffer, BUFFER_SIZE); // reset buffer
}
} catch (SocketException &e)
{
    cerr << e.what() << " UDP Thread" << endl; exit(1);
} return NULL;

/***************************************************************************/
* Function: *UDPBroadcastThread
*    called in creation of the UDP Broadcasting thread.
*    Broadcasts a beacon over UPD at random intervals.
*    Receives: void* robotsAndMap (ptrs to robots and map)
*    Returns: nothing
***************************************************************************/
void *UDPBroadcastThread(void* robotsAndMap)
{
    pthread_detach(pthread_self());

    string servAddress = BROADCAST_ADD; // broadcast address
    string echoString = "BEACON|" + (string)hostip; // string to send
    int echoStringLen; // Length of str to end
    stringstream msgString; // used if loc is known
    time_t t1; // for timestamp
    srand(((unsigned)time(0))); // seed random generator

    ROBOTSMAP_DATA *ptrcast = (ROBOTSMAP_DATA*) robotsAndMap;

    /* */
try
{
    while(1) // loop forever
    {
        // if location is known, add it to the broadcast
        if (ptrcast->me->location.row != 9999) // not default
        {
            msgString.str(""');
            msgString << "|LOC|" << ptrcast->me->location.row
            << "|" << ptrcast->me->location.col
            << "|" << ptrcast->me->direction;
        }
        echoString.append(msgString.str());
    }
    echoString.append(1,\0');
    echoStringLen = echoString.length();

    (void) time(&t1); // timestamp in seconds since epoch
    if (DISPLAY_MSG_TO_CONSOLE)
    {
        cout << "****** BROADCASTING BEACON (" << hostname;
        cout << " " << hostip << ")", at " << ctime(&t1)
        << "****** WITH MSG: " << echoString << \"n\n";
    }
    // Send the string to the server
    ptrcast->udpSock->sendTo(echoString.c_str(), echoStringLen,
                            servAddress, (unsigned short) UDP_PORT);
    echoString = "BEACON|" + (string)hostip;
    sleep(5+(rand()%15)); // Take a rest 5-20 secs.
}
// Destructor closes the socket

} catch (SocketException &e)
{
    cerr << e.what() << " UDP Broadcast Thread" << endl; exit(1); }

return NULL;
}
/* Function: sendTCP
   * send tcp message to one specific robot or to all robots
   * Receives: ROBOTSMAP_DATA *robotsAndMap (ptrs to robots,me,etc)
   *   string msg (message to be sent)
   *   int which (which robot to send msg to.
   *   default is 9999, or to all)
   *
   * Returns: nothing
   *********************************************/
void sendTCP(ROBOTSMAP_DATA *robotsAndMap, string msg, int which)
{
    string baseString = "Q|" + (string)hostip + "|"; // base str to send
    int baseStringLength; // Length of str to echo

    unsigned int numOfRobots = (*robotsAndMap->robots).size();

    // first, check to see if any robots are known
    if (numOfRobots == 0)
    {
        cout << "No robots known. Msg not sent." << endl; return;
    }

    baseString.append(msg);
    baseString.append(1, '\0');

    baseStringLength = strlen(baseString.c_str()); // length of str

    try
    {
        if (which == 9999) // send message to all robots known
        {
            // open a tcp socket with each robot in turn, send msg
            for (unsigned int curr=0; curr<numOfRobots; curr++)
            {
                if (DISPLAY_MSG_TO_CONSOLE)
                    cout << "...contacting " << (*robotsAndMap->robots).at(curr).ip
                        << "...";
                TCPSocket sock((*robotsAndMap->robots).at(curr).ip, TCP_PORT);
                if (DISPLAY_MSG_TO_CONSOLE)
                    cout << "OK\n" << endl;
            }
        }
    }
}
APPENDIX C (continued)

// Send the message
if (DISPLAY_MSG_TO_CONSOLE)
    cout << "...sending message to "
    << (*robotsAndMap->robots).at(curr).ip << "...");
sock.send(baseString.c_str(), baseStringLen);
    if (DISPLAY_MSG_TO_CONSOLE) cout << "OK\n" << endl;
} // end for

} else // send message to specified robot
{
    // open a tcp socket with robot specified
    if (DISPLAY_MSG_TO_CONSOLE)
        cout << "...contacting " << (*robotsAndMap->robots).at(which).ip
        << "...");
    TCPSocket sock((*robotsAndMap->robots).at(which).ip, TCP_PORT);
    if (DISPLAY_MSG_TO_CONSOLE)
        cout << "OK\n" << endl;

    // Send the message
    if (DISPLAY_MSG_TO_CONSOLE)
        cout << "...sending message to "
        << (*robotsAndMap->robots).at(which).ip << "...");
    sock.send(baseString.c_str(), baseStringLen);
    if (DISPLAY_MSG_TO_CONSOLE) cout << "OK\n" << endl;
} catch(SocketException &e) { cerr << e.what() << endl; }

/***************************************************************/
* Function: sendUDP
* send udp message to all robots
* Receives: ROBOTSMAP_DATA *robotsAndMap (ptrs to robots, me, etc)
* string msg (message to be sent)
*
* Returns: nothing
***************************************************************/
void sendUDP(ROBOTSMAP_DATA *robotsAndMap, string msg)
{
    string baseString = "Q|" + (string)hostip + "|"); // base str to send
    int baseStringLen; // Length of str to echo

    unsigned int numRobots = (*robotsAndMap->robots).size();
APPENDIX C (continued)

// first, check to see if any robots are known
if (numOfRobots == 0)
{   cout << "No robots known. Msg not sent." << endl; return; }

baseString.append(msg);
baseString.append(1,'\0');

baseStringLen = strlen(baseString.c_str()); // length of str

try
{
    robotsAndMap->udpSock->sendTo(baseString.c_str(), baseStringLen,
    BROADCAST_ADD, (unsigned short) UDP_PORT);
}
catch(SocketException &e) { cerr << e.what() << endl; }

/***************************************************************
* Function: _initializeHostVars
*    retrieve the host information (ip, name, etc...)
* Receives: ROBOT *me (ptr to this robot's info)
* Returns:  nothing
***************************************************************/
void _initializeHostVars(ROBOT *me)
{
    int retv;
    struct addrinfo hints, *ai = NULL, *aitop = NULL;
    int error, i;
    char ntop[NI_MAXHOST], strport[NI_MAXSERV];
    char udp_port[10];
    sprintf(udp_port, "%d", UDP_PORT);

    //***************************************************************
    * get host name and address
    ***************************************************************
    retv = gethostname ((char *)hostname, 199);
/****************************************/
/* modified following code by David Reid
   http://www.beclan.org/code_view.php?id=19 */
memset(&hints, 0, sizeof(hints));
hints.ai_family = AF_INET;
hints.ai_socktype = SOCK_STREAM;
if ((error = getaddrinfo(hostname, udp_port, &hints, &aitop)) != 0)
{  printf(": %.100s: %s", hostname, gai_strerror(error)); exit(-1); }
for (ai = aitop, i = 1; ai; ai = ai->ai_next)
{
    struct sockaddr_in *sa = (struct sockaddr*)ai->ai_addr;
    if (ai->ai_family != AF_INET && ai->ai_family != AF_INET6) {continue;}
    retv = strlen(inet_ntoa(sa->sin_addr));
    strncpy(hostip, inet_ntoa(sa->sin_addr), retv);
    hostip[retv] = '\0';    /* force str termination */
    LOCATION tempLoc = {9999,9999,false,false,hostip};

    (*me).ip = hostip;
    (*me).name = hostname;
    (*me).direction = "u";
    (*me).location = tempLoc;
    if (getnameinfo(ai->ai_addr, ai->ai_addrlen, ntop, sizeof(ntop),
        strport, sizeof(strport), NI_NUMERICHOST | NI_NUMERICSERV) != 0)
    {   printf("failed\n"); continue; }
}

/***************************************************************/
/* Function: _displayNetInfo
   * displays the current network information
   * Receives: nothing
   * Returns: nothing
***************************************************************/
void _displayNetInfo()
APPENDIX C (continued)

{  
cout << "\n\n\n\n\n"
  << "|                  Network Information                  |\n"  
  << "|=================================================================|\n"  
  << endl;

cout << " Ready to broadcast and receive packets." << endl;

cout << " Running on Host Name: " << hostname << endl;

cout << " With Address: " << hostip << endl;

cout << " Broadcasting UDP on Port: " << BROADCAST_ADD << endl;

cout << " UDP On Port: " << UDP_PORT << endl;

cout << " TCP On Port: " << TCP_PORT << endl;

}
/***************************************************************
* Function: addRobot
*    adds a new robot to the robots vector
* Receives: vector<string> *tokens (the tokens of msg)
*           vector<ROBOT> *robots (the robots)
*
* Returns:  nothing
***************************************************************/
void addRobot(vector<string> *tokens, vector<ROBOT> *robots)
{
    ROBOT tempRobot;
    LOCATION tempLoc = {9999,9999,false,false,(*tokens)[1]};

    tempRobot.ip = (*tokens)[1];
    tempRobot.name = "unknown";
    tempRobot.direction = "u";
    tempRobot.location = tempLoc;

    // add robot to vector list
    (*robots).push_back(tempRobot);
}

/***************************************************************
* Function: vectorRowSum
*    sum adds the values of the vector it is passed, ignoring
*    the column sent it.
* -modified from example of Fred Swartz (fredosaurus.com)
*
* Receives: const vector<float>& x (vector to sum)
*           int ignoreCol (the column to ignore)
*
* Returns:  nothing
***************************************************************/
float vectorRowSum(const vector<float>& x, int ignoreCol)
{
    float total = 0.0; // the sum is accumulated here
    for (unsigned int i=0; i<x.size(); i++)
    {
        if (i != (unsigned int)ignoreCol)
            total += x[i];
    }
    return total;
}
/**************************************************************
* Function: vectorColSum
* sum adds the values in the column of the vector it is passed,
* ignoring the row sent it.
* -modified from example of Fred Swartz (fredosaurus.com)
*
* Receives: vector< vector<float> >&prefMatrix (vector used)
*     int ignoreRow (the row to ignore)
*     int col (column to sum)
*
* Returns: nothing
**************************************************************/
vfloat vectorColSum(const vector< vector<float> >&pMat, int ignoreRow, int col)
{
    float total = 0.0; // the sum is accumulated here
    for (unsigned int i=0; i<pMat.size(); i++)
    {
        if (i != (unsigned int)ignoreRow)
            total += pMat.at(i).at(col);
    }
    return total;
}

/**************************************************************
* Function: HandleControlMsg
* handles the messages from both TCP and UDP from CONTROL
*
* Receives: vector<string> tokens (message sent)
*         ROBOTSMAP_DATA *robotsAndMap (ptrs to robots and map)
*
* Returns: nothing
**************************************************************/
void HandleControlMsg(vector<string> tokens, ROBOTSMAP_DATA *robotsAndMap)
{
    LOCATION tempTarget; // used in adding new targets
    int numOfTargets;   // used in adding new targets
APPENDIX C (continued)

    //##### MAP ###########################################################
if (tokens[2]=="MAP") // control is sending a map
{
    if (DISPLAY_MSG_TO_CONSOLE)
        cout << "Received new map: " << tokens[3] << "x"
            << tokens[4] << endl;
    // resize map to new size here
    buildMap(robotsAndMap->map, // map
              strToInt(tokens[3]), // rows
              strToInt(tokens[4])); // cols
}

    //##### LOC ############################################################
else if (tokens[2]=="LOC") // control is sending this robot's location
{
    if (DISPLAY_MSG_TO_CONSOLE)
        cout << "Received MY location: " << tokens[3] << "x"
    (*robotsAndMap->me).location.row = strToInt(tokens[3]);
    (*robotsAndMap->me).location.col = strToInt(tokens[4]);
    (*robotsAndMap->me).direction = tokens[5];
}

    //##### TAR ############################################################
else if (tokens[2]=="TAR") // control is sending list of targets
{
    if (DISPLAY_MSG_TO_CONSOLE)
        cout << "Received new targets" << endl;

    numOfTargets = strToInt(tokens[3]);
    if (numOfTargets > 0)
    {
        // begin by clearing all targets
        //(*robotsAndMap->targets).erase((*robotsAndMap->targets).begin(),
        //(*robotsAndMap->targets).end());
        (*robotsAndMap->targets).clear();
    }
for (int current = 0; current < numOfTargets; current++)
{
    tempTarget.row = strToInt(tokens[(2*current)+4]);
    tempTarget.col = strToInt(tokens[(2*current)+5]);
    tempTarget.occupied = true;
    tempTarget.target = true;
    tempTarget.ip = "";

    // add target to vector list
    (*robotsAndMap->targets).push_back(tempTarget);
}
if (DISPLAY_MSG_DEBUG)
{
    cout << "Targets Received ( "
        << (*robotsAndMap->targets).size() << ")" << endl;
    for (unsigned int i=0; i<(*robotsAndMap->targets).size(); i++)
    {
        cout << "[" << i << "] "
            << (*robotsAndMap->targets).at(i).row << "|" 
            << (*robotsAndMap->targets).at(i).col << endl;
    }
} // end if numOfTargets
} // end if TAR

//##### QUIT #########################
else if (tokens[2]=="QUIT")
{
    if (DISPLAY_MSG_TO_CONSOLE)
        cout << "Quit command from CONTROL. Exiting program." << endl;
    exit(1);
}

//##### BEGIN ########################
else if (tokens[2]=="BEGIN")
{
    if (DISPLAY_MSG_TO_CONSOLE)
        cout << "Begin command from CONTROL. beginning" << endl;
    begin = true;
}
else if (tokens[2] == "STOP")
{
    if (DISPLAY_MSG_TO_CONSOLE)
        cout << "Stop command from CONTROL.stopping." << endl;
    begin = false;
}

void HandleRobotMsg(vector<string> tokens, ROBOTSMAP_DATA *robotsAndMap, unsigned int which)
{
    
    REFERENCE:
    Q - All TCP queries from a robot begin with Q| to signify a Query (or answer) is being recvd directly from another robot.
    (direct)
    queries/information requested:
        WHERE  - ask specific robot where it is located
        WHAT   - ask specific robot for its task/target preferences
    
    answers/information recvd
        LOC    - provides location of robot sending this cmd
        PREF   - provides task/target preferences of robot sending this cmd
    */
unsigned int curr; // used in debugging
vector<ROBOT> *robs = robotsAndMap->robots;
int numOfPrefs;
float distance; // used in adding to prefs vector
stringstream msg; // used in answer to WHERE, WHAT

//##### LOC ##########################
if (tokens[2]=="LOC" && tokens.size() >= 6) // robot sent its location
{
    // via beacon or directly
    if (DISPLAY_MSG_TO_CONSOLE)
    {
        cout << "Received location from robot [" << which << "]"
            << "(" << tokens[1] << ")"
            << tokens[5] << ")" << endl;
    }

    // change the msg-sending robot's location information
    if (tokens.size() >= 6)
    {
        (*robs).at(which).location.row = strToInt(tokens[3]);
        (*robs).at(which).location.col = strToInt(tokens[4]);
        (*robs).at(which).direction = tokens[5];
    }

    if (DISPLAY_MSG_DEBUG)
    {
        cout << "\nCurrent known robots' locations:\n";
        for (curr=0; curr<*robs.size(); curr++)
        {
            cout << "[" << curr << "][" << (*robs).at(curr).ip << ") ROW: "
                << (*robs).at(curr).location.row
                << " COL: "
                << (*robs).at(curr).location.col
                << " (facing: "
                << (*robs).at(curr).direction << "]\n";
        }
    }
}

} // end else if LOC
//##### PREF ####################################################################
else if (tokens[2]=="PREF") // robot sent its task/target preferences
{
    numOfPrefs = strToInt(tokens[3]);
    if (numOfPrefs > 0)
    {
        // begin by clearing prefs
                                        ((*robs).at(which).prefs).end());

        for (int current = 0; current < numOfPrefs; current++)
        {
            // add pref
            distance = strToFloat(tokens[(2*current)+4]);
            (*robs).at(which).prefs.push_back(make_pair(distance,
                                                        tokens[(2*current)+5]));
        }
    }
}

//##### WHERE ####################################################################
else if (tokens[2]=="WHERE") // robot is seeking this robot's LOC
{
    if (robotsAndMap->me->location.row != 9999) // not default
    {
        msg.str(""); // ensure the stringstream is clear at start
        msg << "LOC|" << robotsAndMap->me->location.row
            << "|" << robotsAndMap->me->location.col
            << "|" << robotsAndMap->me->direction;
        sendTCP(robotsAndMap, msg.str(), which);
    }
}

//##### WHAT ####################################################################
else if (tokens[2]=="WHAT") // robot is seeking this robot's PREF
{
    numOfPrefs = (*robotsAndMap->me).prefs.size();
APPENDIX C (continued)

// first, check to see if preferences have been set
if (numOfPrefs == 0)
{
    if (DISPLAY_MSG_DEBUG)
        cout << "Preferences not set, so preferences not sent." << endl;
} else
{
    msg.str(""'); // ensure the stringstream is clear at start
    msg << "PREF" << numOfPrefs;

    for (int curr=0; curr<numOfPrefs; curr++) // sorted order ascending by float component of the pair
        msg << "|" << (*robotsAndMap->me).prefs.at(curr).first << "|" << (*robotsAndMap->me).prefs.at(curr).second;
    sendTCP(robotsAndMap, msg.str(), which);
} // end else if WHAT

/***************************************************************
* Function: DetermineOwnPrefs
*    determines preferences for each known target,
*    currently based solely on distance.
* *
* Receives: ROBOTSMAP_DATA *robotsAndMap (ptrs to robots and map)
* *
* Returns:  nothing
***************************************************************/
void DetermineOwnPrefs(ROBOTSMAP_DATA *robotsAndMap)
{
    int myrow = (*robotsAndMap->me).location.row,
        mycol = (*robotsAndMap->me).location.col;
    float distance, max; // distance and max distance
    unsigned int curr; // used in debugging sorted prefs
    stringstream keystore; // used for key-x of pair in vector myPrefs
    list<float> distList; // used for max distance

    int numOfTargets = (*robotsAndMap->targets).size();
    (*robotsAndMap->me).prefs.clear(); // clear out the current prefs
if (numOfTargets > 0 && myrow != 9999)
{
  if (DISPLAY_MSG_TO_CONSOLE)
  {   cout << "Calculating my preferences..."; } 

  // calculate distance between robot and each target
  for (int current=0; current<numOfTargets; current++)
  {
    distance = sqrt( 
      (pow((float)myrow-(*robotsAndMap->targets).at(current).row,2)) + 
      (pow((float)mycol-(*robotsAndMap->targets).at(current).col,2)));

    distList.push_back (distance);
    keystr.str(""); 
    keystr << "t" << current; 
    // make_pair creates a pair<float,string> object and add to vector 
    (*robotsAndMap->me).prefs.push_back(make_pair(distance,keystr.str()));
  }

  distList.sort();
  max = distList.back(); // grab the largest distance value

  // linear transformation to Euclidean distance
  for (curr=0; curr< (*robotsAndMap->me).prefs.size(); curr++)
  {
    (*robotsAndMap->me).prefs.at(curr).first =
      1 - ((*robotsAndMap->me).prefs.at(curr).first / max);
  }

  if (DISPLAY_MSG_TO_CONSOLE)
  {   cout << "DONE.\n"; }

  if (DISPLAY_MSG_DEBUG)
  {   for (curr=0; curr< (*robotsAndMap->me).prefs.size(); curr++)
      {   // order of how they were added
          cout << (*robotsAndMap->me).prefs.at(curr).second << "="
               << (*robotsAndMap->me).prefs.at(curr).first << endl;
      }
  }
}

} // end if numOfTargets
/***************************************************************
* Function: ComputeOwnCoupledSelectionPrefs
* compute new preference vector for me based upon
* the prefMatrix sent it. sends this to all known robots
* (via TCP). The last robot in the prefMatrix is me (this
* robot).
*
* Receives: vector< vector<float> > *prefMatrix (robotsXtargets)
* ROBOTSMAP_DATA *robotsAndMap (ptrs to robots, etc)
* 
* Returns: true (if converged), false (if not)
***************************************************************/

bool ComputeOwnCoupledSelectionPrefs(vector< vector<float> > *prefMatrix,
                                    ROBOTSMAP_DATA *robotsAndMap)
{
    int row = (*prefMatrix).size()-1;  // this robot's row in the matrix
    int prefSize = (*prefMatrix).at(row).size();:///(*robotsAndMap->me).prefs.size();
    bool isConverged = false;

    // vars used in coupled selection equation
    float kappa = 1;    // time scaling parameter
    float curVal;       // current value at row x col
    float B = 1;        // Beta coefficient represents the coupling strength
    float B2 = 1;       // Peter Molnar suggests 0 < B < .5, though paper states B>.5
    float addedVal;

    // Peter Molnar states that kappa is linked to the motion
    // equation (force model), and is meaningless if one just considers
    // the decision process. If Kappa is very small the decision process
    // is slowed down with respect to the movement of the robots, (or in
    // general, without robots moving).
    for (int j=0; j<prefSize; j++)
    {
        curVal = (*prefMatrix).at(row).at(j);

        if (curVal == 1) { isConverged = true; continue; } // it converged
        else if (curVal == 0) {} // it converged zero
APPENDIX C (continued)

```cpp
addedVal = kappa * curVal * (1-(pow(curVal,2)) - (B * (pow(vectorColSum(*prefMatrix, row, j),2)) - (B2 * (pow(vectorRowSum(*prefMatrix).at(row), j),2)))
// curVal += kappa * curVal *
// (1 - (pow(curVal,2))
// - (B * (pow(vectorColSum(*prefMatrix, row, j),2)))
// - (B2 * (pow(vectorRowSum(*prefMatrix).at(row), j),2)))
curVal = curVal + addedVal;

// rounding correction for float value comparisons
if (curVal <= 0.00001) curVal = 0;
else if (curVal >= .99999) curVal = 1;
// update prefMatrix and this robot's prefs
(*prefMatrix).at(row).at(j) = curVal;
(*robotsAndMap->me).prefs[j].first = curVal;
// end for

SendOwnPrefs(robotsAndMap);
return isConverged;
}

/***************************************************************
* Function: SendOwnPrefs
* sends this robot's task/target preferences (via TCP) to
* all known robots
*
* Receives: ROBOTSMAP_DATA *robotsAndMap (ptrs to robots,me,etc)
*
* Returns: nothing
***************************************************************/
void SendOwnPrefs(ROBOTSMAP_DATA *robotsAndMap)
{
    int numOfRobots = (*robotsAndMap->robots).size();
    int prefSize = (*robotsAndMap->me).prefs.size();
    stringstream msg; // used in creating the message

    // first, check to see if preferences have been set
    if (prefSize == 0)
    {
        if (DISPLAY_MSG_DEBUG)
            cout << "Preferences not set, so preferences not sent." << endl;
        return;
    }
```

114
// second, check to see if any robots are known
if (numOfRobots == 0)
{
    if (DISPLAY_MSG_DEBUG)
        cout << "No robots known. Preferences not sent." << endl;
    return;
}

msg.str(""); // ensure the stringstream is clear at start
msg << "PREF|" << prefSize;

for (int curr=0; curr<prefSize; curr++)
{   // sorted order ascending by float component of the pair
    msg << "|" << (*robotsAndMap->me).prefs.at(curr).first
        << "|" << (*robotsAndMap->me).prefs.at(curr).second;
}

if (DISPLAY_MSG_DEBUG)
    cout << "\nSending preferences to all known robots..." << endl;

sendTCP(robotsAndMap, msg.str());
}

/***************************************************************
* Function: CheckRobotsLocs
*    checks each robot to see if its location is known.(row/col
*    ==9999 means unknown loc). If unknown, send a request for
*    that robot's location (via TCP)
* *
* Receives: ROBOTSMAP_DATA *robotsAndMap (ptr to robots, etc)
* *
* Returns: true (if all locs are known), false (>=1 unknown)
***************************************************************/
bool CheckRobotsLocs(ROBOTSMAP_DATA *robotsAndMap)
{
    int numOfRobots = (*robotsAndMap->robots).size();
    stringstream msg;     // used in creating the message
    bool allLocsKnown = true;     // return value
for (int curr=0; curr<numOfRobots; curr++)
{
    if ((*robotsAndMap->robots).at(curr).location.row == 9999) // unkn loc
    {
        allLocsKnown = false;
        msg.str(""); // ensure the stringstream is clear at start
        msg << "WHERE";
        sendTCP(robotsAndMap, msg.str(), curr); // request loc
    }
} // end for

return allLocsKnown;

/***************************************************************
* Function: CheckRobotsPrefs
* checks each robot to see if its task/target preference
* is known. If unknown, send a request for that robot's
* preferences (via TCP)
*
* Receives: ROBOTSMAP_DATA *robotsAndMap (ptr to robots, etc)
*
* Returns: true (if all prefs are known), false (>=1 unknown)
***************************************************************/
bool CheckRobotsPrefs(ROBOTSMAP_DATA *robotsAndMap)
{
    int numOfRobots = (*robotsAndMap->robots).size();
    int prefSize; // used for each robot
    stringstream msg; // used in creating the message
    bool allPrefsKnown = true; // return value
    for (int curr=0; curr<numOfRobots; curr++)
    {
        prefSize = (*robotsAndMap->robots).at(curr).prefs.size();
        // if (prefSize == 0) // prefs are not known
        if (prefSize == 0) // prefs are not known
        {
            allPrefsKnown = false;
            msg.str(""); // ensure the stringstream is clear at start
            msg << "WHAT";
            sendTCP(robotsAndMap, msg.str(), curr); // request pref
        }
    }
APPENDIX C (continued)

} // end for

return allPrefsKnown;
}

/***************************************************************
* Function: TaskDecision
* decide/determine which robot to which task/target.
* implementation of my algorithm to pair robots to
* targets based upon everyone's preferences.
*
* Receives: ROBOTSMAP_DATA *robotsAndMap (ptr to robots, etc)
*
* Returns: nothing
***************************************************************/
void TaskDecision(ROBOTSMAP_DATA *robotsAndMap)
{

    //ROBOTSMAP_DATA robotsDataCopy = *robotsAndMap;   // copy
    //vector<ROBOT> *robs = robotsDataCopy.robots;     // short form ptr
    //vector<LOCATION> *tars = robotsDataCopy.targets; // short form ptr

    vector<ROBOT> *robs = robotsAndMap->robots;  // short form ptr
    vector<LOCATION> *tars = robotsAndMap->targets; // short form ptr
    vector<vector<float> >prefMatrix;       // the preference matrix

    int numOfRobots  = (*robs).size();  // number of robots known
    int numOfTargets = (*tars).size();  // number of targets

    // find # of ones to consider matrix converged
    int needed = ((numOfRobots+1)-numOfTargets <= 0) ? (numOfRobots+1) : numOfTargets;

    int decided = 0;  // count of how many are decided
    unsigned int iterations = 0;          // used in debugging
    unsigned int myPrefSize = 0, targetToDo = 9999;
APPENDIX C (continued)

// RESET prefMatrix by erasing and resizing with current data
for(unsigned int i=0; i<prefMatrix.size();i++)
    prefMatrix.at(i).clear();
prefMatrix.erase(prefMatrix.begin(),prefMatrix.end());
prefMatrix.resize(numOfRobots+1, vector<float>(numOfTargets));

// create one-to-one correspondence of robots to tasks
while (decided != needed && begin)
{

decided = 0; // reset number of ones seen

// FILL prefMatrix
//FillPrefMatrix(robotsDataCopy, prefMatrix);
FillPrefMatrix(*robotsAndMap, prefMatrix);

// check here: if numOfTargets==1 and there are no other known robots,
// then assign that target location to this robot. Don't converge.
if (numOfRobots==0 && numOfTargets==1)
{
    prefMatrix.at(0).at(0) = 1;
    break; // leave the while loop
}

// DISPLAY current preference matrix
if (DISPLAY_MSG_DEBUG)
{
    cout << "Iteration #" << ++iterations << endl;
    DisplayPrefMatrix(&prefMatrix);
}

// COMPUTE own coupled selection pref vector
if (ComputeOwnCoupledSelectionPrefs(&prefMatrix, robotsAndMap))
{
    decided++; } // this robot's prefs converged, so increment

// DETERMINE # of other robots' pref vectors converged at this point
decided += DetermineConvergedNumber(&prefMatrix, false);

// ERASE all the known robot's preference, except self.
ResetAllRobotsPrefs(robotsAndMap);
APPENDIX C (continued)

    // show decided vs needed
    if (DISPLAY_MSG_DEBUG)
    {
        cout << "needed:" << needed << " decided:" << decided << endl;
    }

    // WAIT for new preferences from other known robots
    while (begin && !CheckRobotsPrefs(robotsAndMap))
    {
        sleep(1);
    }
}

    // DISPLAY final preference matrix
    if (DISPLAY_MSG_DEBUG)
    {
        cout << "Iteration Final" << endl;
        DisplayPrefMatrix(&prefMatrix);
    }

    // find the target that this robot needs to do
    myPrefSize = prefMatrix.at(numOfRobots).size();
    for (unsigned int curpref=0; curpref<myPrefSize; curpref++)
    {
        if (prefMatrix.at(numOfRobots).at(curpref) == 1)
        {
            targetToDo = curpref; break;
        }
    }

    // if robot has a task/target to do, then move it there
    if (targetToDo != 9999) // a task/target has been decided
    {
        //robotsAndMap ONLY to update the actual location
        //targetToDo is the task/target for this robot
        //tars is the current listing (the copy) of targets
        // and their respective locations.
        MoveRobotToTarget(robotsAndMap, targetToDo, tars); // move to target
    }

    // ResetAllRobotsPrefs(robotsAndMap, true);
}
/***********************************************************************/
* Function: FillPrefMatrix                           
*    fill the prefMatrix with preferences from known robots      
*                                                           
* Receives: ROBOTSMAP_DATA *robotsAndMap (ptrs to robots, etc.)  
*    vector< vector<float> >*prefMatrix (ptr prefMatrix)        
*                                                           
* Returns: nothing                                           
***********************************************************************/

void FillPrefMatrix(ROBOTSMAP_DATA &robotsAndMap,
                    vector< vector<float> >&prefMatrix)
{
    vector<ROBOT> *robs = robotsAndMap.robots; // short form
    int numOfRobots  = (*robs).size() + 1;  // number of robots known + me
    int currob, curpref;                    // used in setting up matrix
    int prefSize;

    for (currob=0; currob<numOfRobots-1; currob++)
    {
        prefSize = (*robs).at(currob).prefs.size();
        for (curpref=0; curpref<prefSize; curpref++)
        {
            // add pref for target to prefmatrix
            prefMatrix.at(currob).at(curpref) =
                (*robs).at(currob).prefs[curpref].first;
        }
    }

    // add me to the preference matrix at [numOfRobots]
    prefSize = robotsAndMap.me->prefs.size();
    for (curpref=0; curpref<prefSize; curpref++)
    {
        // add pref for target to prefmatrix
        prefMatrix.at(numOfRobots-1).at(curpref) =
            robotsAndMap.me->prefs[curpref].first;
    }
}
/***************************************************************
* Function: DisplayPrefMatrix
* displays a row x col graph of each robot’s preferences
* as existing in the prefMatrix sent to it. typically
* used for debugging
* 
* Receives: vector< vector<float> > *prefMatrix (ptr prefMatrix)
* 
* Returns: nothing
***************************************************************/
void DisplayPrefMatrix(vector< vector<float> > *prefMatrix)
{
    int nRob = (*prefMatrix).size();       // number of robots/rows
    int nTar = (*prefMatrix).at(0).size(); // number of targets/cols
    stringstream strkey;                 // used in debugging

cout << "   ";
for (int j=0; j<nTar; j++)
{
    cout << setfill (')') << setw (11) << j;
}
cout << endl;
cout << "   ";
cout << setfill ('-') << setw (nTar*12);
cout << "-" << endl; // draws a line of proportional size

cout << "  ";
for (int i=0; i<nRob; i++)
{
    cout << setfill ('') << setw (3) << i;
    for (int j=0; j<nTar; j++)
    {
        strkey.str(""لة"'");
        strkey << setw (11) << (*prefMatrix).at(i).at(j);
        cout << '|' << strkey.str();
    }
    cout << "|
    
    cout << setfill ('-') << setw ((nTar*12)+1);
    cout << "." << endl; // draws a line of proportional size
}
}
/***************************************************************
* Function: ResetAllRobotsPrefs
* erases all robots' prefs, returning their vector size to 0x0
*
* Receives: ROBOTSMAP_DATA *robotsAndMap (ptrs to robots, etc).
* bool includeSelf (whether to include this robot,
* default is false)
*
* Returns: nothing
***************************************************************/
void ResetAllRobotsPrefs(ROBOTSMAP_DATA *robotsAndMap, bool includeSelf)
{
    for (unsigned int curr=0; curr<*robotsAndMap->robots.size(); curr++)
    {
        (*robotsAndMap->robots).at(curr).prefs.erase (
            (*robotsAndMap->robots).at(curr).prefs.begin(),
            (*robotsAndMap->robots).at(curr).prefs.end() );
    }
    if (includeSelf)
    {
        // erase me's pref
        (*robotsAndMap->me).prefs.erase (
            (*robotsAndMap->me).prefs.begin(),
            (*robotsAndMap->me).prefs.end() );
    }
}

/***************************************************************
* Function: DetermineConvergedNumber
* determines how many rows in the prefMatrix have converged.
* it skips the last row, which is this robot's prefs, if
* var self is false (default is true...to include self)
*
* Receives: vector< vector<float> > *prefMatrix (ptr prefMatrix)
* bool self (whether to include self, default true)
*
* Returns: # of robots converged in the prefMatrix
***************************************************************/
int DetermineConvergedNumber(vector< vector<float> > *prefMatrix, bool self)
{
    int ones=0; // return value
    float curVal; // current value at rowXcol
    for (unsigned int row=0; row<prefMatrix->size(); row++)
    {
        int count=0;
        for (unsigned int col=0; col<prefMatrix->at(row).size(); col++)
        {
            if (fabs(prefMatrix->at(row).at(col)) > 0.002)
                count++;
        }
        if (count == prefMatrix->at(row).size() - 1)
            ones++;
    }
    return ones;
}
int nRob = (*prefMatrix).size(); // number of robots/rows
int nTar = (*prefMatrix).at(0).size(); // number of targets/cols

int rows = (self) ? nRob : nRob-1;

for (int i=0; i<rows; i++)
{
   for (int j=0; j<nTar; j++)
   {
      curVal = (*prefMatrix).at(i).at(j);
      if (curVal == 1) { ones++; continue; } // it converged
   }
} // end for

return ones;

// used to hook into robot, from Acroname example for garciaApp
class acpGarciaAppExecute :
    public acpCallback
    {
    public:
        acpGarciaAppExecute(
            acpGarcia* pcGarcia,
            acpObject* pBehavior
        ):
            m_pcGarcia(pcGarcia),
            m_pBehavior(pBehavior)
        {}

        aErr     call();

    private:
        acpGarcia*   m_pcGarcia;
        acpObject*   m_pBehavior;
    };

123
APPENDIX C (continued)

aErr acpGarciaAppExecute::call()
{
    cout << "behavior "
        << m_pBehavior->getNamedValue("name")->getStringVal()
        << " with id "
        << m_pBehavior->getNamedValue("unique-id")->getIntVal()
        << " executing...\n";

    return aErrNone;
}

} /* sExecute callback*/

class acpGarciaAppComplete :
public acpCallback
{
public:
    acpGarciaAppComplete(
        acpGarcia* pcGarcia,
        acpObject* pBehavior
    ) :
        m_pcGarcia(pcGarcia),
        m_pBehavior(pBehavior)
    {}

    aErr  call();

private:
    acpGarcia* m_pcGarcia;
    acpObject* m_pBehavior;
};

/*******************

aErr acpGarciaAppComplete::call()
{
    cout << "completed with status = "
        << m_pBehavior->getNamedValue("completion-status")->getIntVal();

    return aErrNone;
}

} /* sComplete callback*/
/* Function: MoveRobotToTarget
   instantiate and move this robot to the task/target location
   specified. (Altered heavily from the triangle demo application
   of the Garcia API examples)
   *
   * Receives: ROBOTSMAP_DATA *robotsAndMap (only to update this
   *          robot's actual location)
   *           unsigned int toDo (the task/target for this robot)
   *           vector<LOCATION> *tars (current listing (the copy)
   *                   of knwon targets)
   *
   * Returns: nothing
   */
void MoveRobotToTarget(ROBOTSMAP_DATA *robotsAndMap, unsigned int toDo,
                       vector<LOCATION> *tars)
{
    vector< pair<string,float> > moveQ;  // list of commands to do
    float rotAngle45 = 3.14159265 / 4;
    float rotAngle90 = 3.14159265 / 2;
    float rotAngle180= 3.14159265;
    float lastAngle = 0; // used in queuing

    LOCATION myTarget; // location of task/target
    myTarget.row = (*tars).at(toDo).row;
    myTarget.col = (*tars).at(toDo).col;

    int myrow = (*robotsAndMap->me).location.row,
                mycol = (*robotsAndMap->me).location.col;

    int moveQSize; // used in enqueuing into behavior queue

    float dist;  // distance to move

    aErr err;
    aIOLib ioRef;
    acpGarcia garcia;
    acpObject* pBehavior;

    aIO_GetLibRef(&ioRef, &err);
    cout << "waiting for garcia\n";
while (!garcia.getNamedValue("active") -> getBoolVal())
{
    cout << "still waiting\n";
    a1O_MSSleep(ioRef, 100, NULL);
}

// future work:
// flush the queue
//garcia.flushQueuedBehaviors(); // stop previous behavior queue, if any

// set direction (internally and externally) facing north, after
// flushing queue

// determine how to get there, and place in a vector moveQ

// #### IN THE SAME ROW #######
if (myrow == myTarget.row) // in same row
{
    if (mycol <= myTarget.col)
    {
        moveQ.push_back(make_pair("pivot", -rotAngle90));
        dist = (myTarget.col - mycol) * squareSize;
        moveQ.push_back(make_pair("move", dist));
        moveQ.push_back(make_pair("pivot", rotAngle90)); // face north
    }
    else if (mycol > myTarget.col)
    {
        moveQ.push_back(make_pair("pivot", rotAngle90));
        dist = (mycol - myTarget.col) * squareSize;
        moveQ.push_back(make_pair("move", dist));
        moveQ.push_back(make_pair("pivot", -rotAngle90)); // face north
    }
}
}
else if (mycol == myTarget.col) // in same col
{
  if (myrow <= myTarget.row) // turn around, move to target
  {
    moveQ.push_back(make_pair("pivot", rotAngle180));
    dist = (myTarget.row - myrow) * squareSize;
    moveQ.push_back(make_pair("move", dist));
    moveQ.push_back(make_pair("pivot", -rotAngle180)); // face north
  }

  else if (myrow > myTarget.row) // move forward to target
  {
    dist = (myrow - myTarget.row) * squareSize;
    moveQ.push_back(make_pair("move", dist));
  }
}

else // target's row != myrow && target's col != mycol
{
  // first, move diagonally to the closest we can get
  if (myrow < myTarget.row) // turn around, move to target
  {
    moveQ.push_back(make_pair("pivot", rotAngle180));
  }

  // we're going to move diagonally
  if (myTarget.col < mycol)
  {
    moveQ.push_back(make_pair("pivot", -rotAngle45));
    lastAngle = -rotAngle45;
  }
  else
  {
    moveQ.push_back(make_pair("pivot", rotAngle45));
    lastAngle = rotAngle45;
  }
}
// enqueue how far to go diagonally
while (myTarget.col != mycol && myTarget.row != myrow)
{
    dist = diagDist; // one square diagonally
    moveQ.push_back(make_pair("move", dist));
    if (myTarget.col < mycol) { mycol--; }
    else { mycol++; }
} // end while

lastAngle = lastAngle * 3; // used in returning to start state

} // end if

else // myrow > myTarget.row
{
    // we're going to move diagonally
    if (myTarget.col < mycol)
    {
        moveQ.push_back(make_pair("pivot", rotAngle45));
        lastAngle = -rotAngle45; // pivot rev dir afterwards
    }
    else
    {
        moveQ.push_back(make_pair("pivot", -rotAngle45));
        lastAngle = rotAngle45; // pivot rev dir afterwards
    }

    // enqueue how far to go diagonally
    while (myTarget.col != mycol && myTarget.row != myrow)
    {
        dist = diagDist; // one square diagonally
        moveQ.push_back(make_pair("move", dist));
        if (myTarget.col < mycol) { mycol--; }
        else { mycol++; }
    } // end while
}
// check if we have arrived at myTarget via diagonal moving
if (myTarget.col == mycol && myTarget.row == myrow) // we're there
{
    moveQ.push_back(make_pair("pivot", lastAngle)); // face north
}
else // not there, but in same row or col as target
{
    // #### IN THE SAME ROW, NOW #################
    if (myrow == myTarget.row) // in same row
    {
        if (mycol <= myTarget.col)
        {
            moveQ.push_back(make_pair("pivot", -rotAngle90));
            dist = (myTarget.col - mycol) * squareSize;
            moveQ.push_back(make_pair("move", dist));
            moveQ.push_back(make_pair("pivot", rotAngle90)); // face n
        }
        else if (mycol > myTarget.col)
        {
            moveQ.push_back(make_pair("pivot", rotAngle90));
            dist = (mycol - myTarget.col) * squareSize;
            moveQ.push_back(make_pair("move", dist));
            moveQ.push_back(make_pair("pivot", -rotAngle90)); // face n
        }
    }
    else if (mycol >= myTarget.col) // in same col
    {
        if (myrow <= myTarget.row) // turn around, move to target
        {
            moveQ.push_back(make_pair("pivot", rotAngle180));
            dist = (myTarget.row - myrow) * squareSize;
            moveQ.push_back(make_pair("move", dist));
            moveQ.push_back(make_pair("pivot", -rotAngle180)); // face n
        }
        else if (myrow > myTarget.row) // move forward to target
        {
            dist = (myrow - myTarget.row) * squareSize;
            moveQ.push_back(make_pair("move", dist));
        }
    }
}
APPENDIX C (continued)

} // end else
} // end else
} // end else NOT SAME ROW OR COL

// ####### QUEUE THE BEHAVIORS INTO ROBOT #######
moveQSize = moveQ.size(); // how many commands/actions
acpValue executeCB;
acpValue completeCB;
acpValue lengthVal;
acpValue rotationVal;

// go through vector moveQ and process each (string,float) pair
for (int curr=0; curr<moveQSize; curr++)
{
    // ##### MOVE #####################
    if (moveQ.at(curr).first == "move")
    {
        lengthVal = moveQ.at(curr).second;
pBehavior = garcia.createNamedBehavior("move", "makeitso");
pBehavior->setNamedValue("distance", &lengthVal);
executeCB.set(new acpGarciaAppExecute(&garcia, pBehavior));
pBehavior->setNamedValue("execute-callback", &executeCB);
completeCB.set(new acpGarciaAppComplete(&garcia, pBehavior));
pBehavior->setNamedValue("completion-callback", &completeCB);
garcia.queueBehavior(pBehavior);
    }

    // ##### PIVOT ####################
    else if (moveQ.at(curr).first == "pivot")
    {
        rotationVal = moveQ.at(curr).second;
pBehavior = garcia.createNamedBehavior("pivot", "boldlygo");
pBehavior->setNamedValue("angle", &rotationVal);
executeCB.set(new acpGarciaAppExecute(&garcia, pBehavior));
pBehavior->setNamedValue("execute-callback", &executeCB);
completeCB.set(new acpGarciaAppComplete(&garcia, pBehavior));
pBehavior->setNamedValue("completion-callback", &completeCB);
garcia.queueBehavior(pBehavior);
    }
}
APPENDIX C (continued)

```c
// ####### OTHER CMDS #############################################
else // other commands, if needed
{}
} // end for

//acpValue lengthVal(0.5f);
//acpValue rotationVal((float)(3.14159265 / 4)); // 45 degrees

// this loop checks to see when the list above is complete
// and checks for callbacks in 100 mSec intervals
while (!garcia.getNamedValue("idle")->getBoolVal())
garcia.handleCallbacks(100);

cout << "done\n";

// update robot with new location
(*robotsAndMap->me).location.row = myrow;
(*robotsAndMap->me).location.col = mycol;

aIO_ReleaseLibRef(ioRef, &err);
```
APPENDIX D

CODE – APPSERVER.H

// contains declarations and macros for AppServer

#ifndef APPSERVER_H
#define APPSERVER_H

/*=============================*
      function declarations    |
*=============================*/

void HandleTCPClient(TCPSocket *sock, ROBOTSMAP_DATA *robotsAndMap);
void HandleUDPClIENT(vector<string> tokens, ROBOTSMAP_DATA *robotsAndMap);
void *TCPThread(void* robotsAndMap);  // TCP thread handling function
void *UDPTThread(void* robotsAndMap);  // UDP thread handling function
void *UDPBroadcastThread(void* robotsAndMap); // UDP bcast thread function
void sendTCP(ROBOTSMAP_DATA *robotsAndMap, string msg, int which=9999);
void sendUDP(ROBOTSMAP_DATA *robotsAndMap, string msg);

void buildMap(vector< vector<LOCATION> >*map, int rows, int cols);
void addRobot(vector<string> *tokens, vector<ROBOT> *robots);
float vectorRowSum(const vector<float>& x, int ignoreCol);
float vectorColSum(const vector< vector<float> >&pMat, int ignoreRow, int col);

void HandleControlMsg(vector<string> tokens, ROBOTSMAP_DATA *robotsAndMap);
void HandleRobotMsg(vector<string> tokens, ROBOTSMAP_DATA *robotsAndMap,
unsigned int which);
void DetermineOwnPrefs(ROBOTSMAP_DATA *robotsAndMap);
bool ComputeOwnCoupledSelectionPrefs(vector< vector<float> >*prefMatrix,
ROBOTSMAP_DATA *robotsAndMap);

void SendOwnPrefs(ROBOTSMAP_DATA *robotsAndMap);
bool CheckRobotsLocs(ROBOTSMAP_DATA *robotsAndMap);
bool CheckRobotsPrefs(ROBOTSMAP_DATA *robotsAndMap);
void TaskDecision(ROBOTSMAP_DATA *robotsAndMap);
void FillPrefMatrix(ROBOTSMAP_DATA &robotsAndMap,
    vector< vector<float> >&prefMatrix);
void DisplayPrefMatrix(vector< vector<float> >*prefMatrix);
void ResetAllRobotsPrefs(ROBOTSMAP_DATA *robotsAndMap,bool includeSelf=false);
int DetermineConvergedNumber(vector< vector<float> >*prefMatrix, bool self);

void MoveRobotToTarget(ROBOTSMAP_DATA *robotsAndMap, unsigned int toDo,
    vector<LOCATION> *tars);

#endif
APPENDIX E
CODE – MDEFS.H

/*********************************************************************************/
* FILE: mdefs.h                                                            *
* This file contains the program macro definitions, macro functions,       *
* typedefs, and function declarations used                                          *
*********************************************************************************/

#ifndef MDEFS_H
#define MDEFS_H

#define UDP_PORT 8025         // UDP port
#define TCP_PORT 8100         // TCP port
#define BUFFER_SIZE 512
#define BROADCAST_ADD "192.168.3.255"  // robot network
#define MAX_ROWS_COLS 100     // maximum # of rows or cols for map

/* configuration settings: TRUE or FALSE */
#define IGNORE_SELF_BC         true   /* ignore broadcasts from self */
#define DISPLAY_MSG_TO_CONSOLE false  /* print all messages to console */
#define DISPLAY_MSG_DEBUG      true   /* print debug messages to console */

// used to replace bzero with memset
#define bzero(b,len) (memset((b), '\0', (len)), (void) 0)

// #define LOG_ALL_MESSAGES       false  /* save log of all communication */
// #define LOG_FILE_NAME "comm.log"      /* file name of the log file */

// used to replace bzero with memset
#define bzero(b,len) (memset((b), '\0', (len)), (void) 0)

typedef struct location_type
{
    int row, col;
    bool occupied; // occupied by robot/target?
    bool target;  // is the target here?
    string ip;   // if occupied by robot, which?
}LOCATION;
APPENDIX E (continued)

typedef struct robot_type
{
  string ip;
  string name;
  LOCATION location;  // where the robot is at on the map
  string direction;  // direction facing
  vector<pair<float,string>> prefs;  // robot's task/target preferences
}ROBOT;

// holds ptrs to robots, map, targets, me and the udp sock
typedef struct robotsmap_data
{
  UDPSocket *udpSock;  // udp socket
  ROBOT *me;  // info on this robot
  ROBOT *robots;  // ptr to known robots
  vector<vector<LOCATION>> *map;  // ptr to map
  vector<LOCATION> *targets;  // ptr to known targets
}ROBOTS_MAP_DATA;

/*=============================*
 | global                      |
 *=============================*/
char hostip[50];
char hostname[200];

// lengths are in meters (float), angle in radians for robots API
float squareSize;  // size of each side of each square on map
float diagDist;  // distance from 1 sqr to another diagonally

/*=============================*
 | function declarations       |
 *=============================*/
void _initializeHostVars(ROBOT *me);
void _displayNetInfo();

#endif
APPENDIX F
CODE – COMMON_FNS.CPP

/********************************************
* common_fns
* Programmer: Zachary Zaccagni
* Common functions shared by both Controller/Observer and Server/Robot applications
* ********************************************/

#include <iostream>
#include <vector>             // For vectors
#include <sstream>            // For stringstream
#include <cmath>              // For pow()
#include "PracticalSocket.h"  // For Socket, ServerSocket, and SocketException
#include "common_fns.h"
using namespace std;

/********************************************
* Function: explodeStr
* split a null-terminated character array on a given character
* into a vector of strings. Vector is passed by reference and cleared each time. number of strings split out is returned.
* function by William S. Lear rael@deja.com
* on mail.python.org/pipermail/python-list/1999-May/002439.html
* example call: explodeStr(v, line.c_str(), '|');
* Receives: vector<string>& v (holds the split out strings)
* const char* s (str to work on)
* char c (delimiter)
* Returns: size of vector
********************************************/
int explodeStr(vector<string>& v, const char* s, char c)
{
    v.clear();
    while (true)
    {
        const char* begin = s;

        while (*s != c && *s) { ++s; }

        v.push_back(string(begin, s));
        if (!*s)
        {   break; }

        if (!*++s)
        {
            v.push_back(""");
            break;
        }
    }
    return v.size();
}

/***************************************************************
* Function: implodeStr
* glues together each element of a string vector using the "glue"  
* char specified. Returns a string which contains all the
* vector elements in the same order as their index.
* 
* modified from function by Sang-drax (member in forum) 
* on cboard.cprogramming.com/archive/index.php/t-64181.html 
* example call: implodeStr(v, '|');
* Receives: vector<string>& v (vector to work on)
* char glue (seperator, defaults to empty str)(optional)
* 
* Returns: nothing
***************************************************************/

string implodeStr(vector<string> v, char glue)
{
    stringstream out;
    for (unsigned int p=0;p<v.size()-1;++p)
        out << v[p] << glue;
    out << v[v.size()-1];
    return out.str();
}
APPENDIX F (continued)

/***************************************************************
* Function: strUpper
* transform function to convert an entire string to uppercase
* transform statement from function by Danny Kalev, author of
* example call for string conversion:
*    strUpper(s)
* Receives: string s (string to convert)
* Returns: nothing
***************************************************************/
void strUpper(string &s)
{
    transform(s.begin(), s.end(), s.begin(), (int(*)(int)) toupper);
}

/***************************************************************
* Function: strLower
* transform function to convert an entire string to lowercase
* transform statement from function by Danny Kalev, author of
* example call for string conversion:
*    strLower(s)
* Receives: string s (string to convert)
* Returns: nothing
***************************************************************/
void strLower(string &s)
{
    transform(s.begin(), s.end(), s.begin(), (int(*)(int)) tolower);
}

/***************************************************************
* Function: strToInt
* converts a string to int using string streams
* provided by cprogramming.com - Alexander Allain
* call eg: if (strToInt(s1, result)){}
* Receives: const string &s (string to convert)
*           unsigned int &i (int to put it into)
* Returns: true (if successful), false (if not)
***************************************************************/
bool strToInt(const string &s, unsigned int &i)
{
    istringstream myStream(s);
    if (myStream>>i)
        return true;
    else
        return false;
}

/*******************************************************************************
* Function: strToInt (overload)
* converts a string to int using string streams.
* programmer should ensure that str contains integer before calling. This is primarily used inline for streams.
* modified from function provided by Alexander Allain
* - cprogramming.com.
* call eg: strToInt(s1)
* Receives: const string &s (string to convert)
* Returns: int i
*******************************************************************************/
int strToInt(const string &s)
{
    int i;
    istringstream myStream(s);
    if (myStream>>i)
        return i;
    else
        return 0;
}

/*******************************************************************************
* Function: strToFloat
* converts a string to float using string streams.
* programmer should ensure that str contains float before calling. This is primarily used inline for streams.
* modified from function provided by Alexander Allain
* - cprogramming.com.
* call eg: strToFloat(s1)
* Receives: const string &s (string to convert)
* Returns: int i
*******************************************************************************/
float strToFloat(const string &s) {
    float i;
    istringstream myStream(s);
    if (myStream >> i) {
        return i;
    } else {
        return 0;
    }
}

/***************************************************************
 * Function: roundFloat
 * To round Value to NumPlaces decimal places.
 * modified from function provided by Br. David Carlson at
 * cis.stvincent.edu in his tutorials..
 *
 * Receives: value    (a floating point number)
 *            numPlaces (a positive integer giving the number of
do
decimal places to which to round the answer)
 *
 * Returns: a rounded float number
 ***************************************************************/
float roundFloat(float value, int numPlaces) {
    float factor;
    int temp;

    factor = pow(10.0, numPlaces);
    temp = (int)(value * factor + 0.5);

    return temp / factor;
}
APPENDIX G

CODE – COMMON_FNS.H

// contains declarations and macros for common_fns.h

#ifndef COMMON_FNS_H
#define COMMON_FNS_H

/*=============================*
 | function declarations       |
 *=============================*/

int explodeStr(vector<string>& v, const char* s, char c);
string implodeStr(vector<string> v, char glue='\0');

void strUpper(string &s);
void strLower(string &s);
bool strToInt(const string &s, unsigned int &i);
int strToInt(const string &s);
float strToFloat(const string &s);
float roundFloat(float value, int numPlaces);

#endif
/**
 * Function: TaskDecision
 *  decide/determine which robot to which task/target.
 *  implementation of my algorithm to pair robots to
 *  targets based upon everyone’s preferences.
 *
 *  * Receives: ROBOTSMAP_DATA *robotsAndMap (ptr to robots, etc)
 *  *
 *  * Returns: nothing
 */

void TaskDecision(ROBOTSMAP_DATA *robotsAndMap)
{
  ROBOTSMAP_DATA robotsDataCopy = *robotsAndMap; // copy
  vector<ROBOT> *robs = robotsDataCopy.robots;     // short form ptr
  vector<LOCATION> *tars = robotsDataCopy.targets; // short form ptr
  vector<int> >prefMatrix; // the preference matrix
  vector<int> conflictedRows; // rows that are conflicted

  int numOfRobots = (*robs).size() + 1;  // number of robots known + me
  int numOfTargets = (*tars).size();    // number of targets

  int needed = (numOfRobots - numOfTargets <= 0) ? numOfRobots : numOfTargets;
  int decided = 0;  // count of how many are decided
  int curCol = 0;  // current column (target)(j) to look at
  unsigned int iterations = 0;          // used in debugging
  unsigned int myPrefSize = 0, targetToDo = 9999;

  // RESET prefMatrix and resize to appropriate size
  for(unsigned int i=0; i<prefMatrix.size();i++)
    prefMatrix.at(i).clear();
  prefMatrix.erase(prefMatrix.begin(),prefMatrix.end());
  prefMatrix.resize(numOfRobots, vector<int>(numOfTargets));

  // FILL prefMatrix
  FillPrefMatrix(robotsDataCopy, prefMatrix);
while (decided != needed) {
    if (curCol == 0) decided = 0;

    // DISPLAY current preference matrix
    if (DISPLAY_MSG_DEBUG) {
        cout << "Iteration #" << ++iterations << " COL: " << curCol << endl;
        DisplayPrefMatrix(&prefMatrix);
    }

    conflictedRows.clear();
    FindOnesInCol(prefMatrix, conflictedRows, curCol);

    // resolve conflicts if there are two+ rows/robots vying for 1
    if (conflictedRows.size() > 1) {
        ResolveConflicts(prefMatrix, conflictedRows,
                         robotsDataCopy, decided, curCol);
    } else if (conflictedRows.size() == 1) {
        decided++;
    }

    //curCol = ((curCol+1) < numOfTargets) ? curCol+1 : 0; // wrap around cols
    curCol++;
    if (curCol >= numOfTargets) {
        curCol = 0;
    }

    // sleep to show slower iterations
    if (DISPLAY_MSG_DEBUG) {
        sleep(1);  // used in debugging only
    }
} // end while (decided != needed)

// DISPLAY final preference matrix
if (DISPLAY_MSG_DEBUG) {
    cout << "Iteration Final" << endl;
    DisplayPrefMatrix(&prefMatrix);
}
// find the target that this robot needs to do
myPrefSize = prefMatrix.at(numOfRobots-1).size();
for (unsigned int curpref=0; curpref<myPrefSize; curpref++)
{
    if (prefMatrix.at(numOfRobots-1).at(curpref) == 1)
    {
        targetToDo = curpref; break;
    }
}

// if robot has a task/target to do, then move it there
if (targetToDo != 9999) // a task/target has been decided
{
    //robotsAndMap ONLY to update the actual location
    //targetToDo is the task/target for this robot
    //tars is the current listing (the copy) of targets
    // and their respective locations.
    MoveRobotToTarget(robotsAndMap, targetToDo, tars); // move to target
}
ResetAllRobotsPrefs(robotsAndMap, true);

/***************************************************************/
/* Function: FindOnesInCol */
/*       find number of 1's in specified column */
/* */
/* Receives: vector< vector<float> >&prefMatrix (vector used) */
/*     vector<string> &conflictedRows (store rows with 1) */
/*     int col (column to look at) */
/* */
/* Returns: nothing */
/***************************************************************/
void FindOnesInCol(const vector<vector<int>>& pMat, 
    vector<int>& conflictedRows, int col)
{
    unsigned int psize = pMat.size();
    for (unsigned int i = 0; i < psize; i++)
    {
        if (pMat.at(i).at(col) == 1)
        {
            conflictedRows.push_back(i);
        }
    }
}

/***************************************************************
* Function: ResolveConflicts
*    resolves the conflict of multiple rows vying for 1 as
*    its preference. It does this by deciding which is closer,
*    and then shifting the others' preferences down.
*
*    Receives: vector<vector<float>>& prefMatrix (vector used)
*               vector<string>& conflictedRows (conflicted rows)
*               int& decided (once resolved, incremented)
*               int col (col/target to be resolved)
*
*    Returns:  nothing
***************************************************************/
void ResolveConflicts(vector<vector<int>>& pMat, 
    vector<int>& conflictedRows, 
    ROBOTSMAP_DATA& robotsAndMap, int& decided, int col)
{
    vector<ROBOT>* robs = robotsAndMap.robots; // short form ptr
    ROBOT* me = robotsAndMap.me;               // short form ptr
    unsigned int prefSize = pMat.at(0).size(); // short form ptr
    unsigned int rowSize = pMat.size();        // number of robots/rows
    int closest, row, newpref;
    int needed = (rowSize-prefSize <= 0) ? rowSize : prefSize;

    // Determine closest row (i) of all rows with 1,
    closest = DetermineClosest(conflictedRows, *robs, *me, col);
APPENDIX H (continued)

/*
Mark row (i), that is closest, as decided,
Decrement the prefs of other rows, having 1, by one,
Example: Consider 3 cols, therefore prefs of 1, 2 or 3.
   1 becomes 0
   2 becomes 1
   3 becomes 2
since the winning robot is the only 1, set the losing robot's
row at this col to 0, but still decrement other col's prefs.

if needed==decided, simply zero-out all prefs for the losers,
since THIS decision is done. This prevents undeciding of
previously decided cols. (note: If a robot is to decide to do something
differently after a col has been decided, it will happen at
the next decision-making call.)

*/

decided++; // this col has been decided

// shift prefs of other rows down by 1 in prefMatrix
while (!conflictedRows.empty())
{
    row = conflictedRows.back(); // get the row
    conflictedRows.pop_back();   // remove from vector
    if (row != closest)          // then shift prefs
    {
        for (unsigned int curPref=0; curPref<prefSize; curPref++)
        {
            newpref = pMat.at(row).at(curPref) - 1;
            if ( (newpref > 0) && (decided != needed) )
            {  pMat.at(row).at(curPref) = newpref; }
            else
            {  pMat.at(row).at(curPref) = 0; }
        }
    } // end if
} // end while
/***************************************************************
* Function: DetermineClosestDistance
*    determines the closest of the conflicted robots to a target.
* *
* Receives: vector<string> &conflictedRows (conflicted rows)
*           vector<ROBOT> *robs (ptr to robots)
*           int col (col/target specified)
* *
* Returns:  int (the row/robot that is closest)
***************************************************************/

int DetermineClosest(vector<int> &conflRows, vector<ROBOT> &robs,
                     ROBOT &me, int col)
{
    unsigned int cur, conflSize = conflRows.size();
    int numOfRobots = robs.size();
    int closest = conflRows[0];
    int curRob;
    vector< pair<float,string> > rob1, rob2; // pref sorted by 2nd

    for (cur=1; cur < conflSize; cur++)
    {
        curRob = conflRows[cur];

        if (curRob >= numOfRobots) // then comparing against me
        {
            rob1 = me.prefs;
            rob2 = robs[closest].prefs;

            // sort by target #, instead of distance
            sort(rob1.begin(), rob1.end(), UnsortPairs);
            sort(rob2.begin(), rob2.end(), UnsortPairs);

            if (rob1[col].first < rob2[col].first)
            {
                closest = curRob;
            }
        }
    }
}
else
{
    rob1 = robs[curRob].prefs;
    rob2 = robs[closest].prefs;

    // sort by target #, instead of distance
    sort(rob1.begin(), rob1.end(), UnsortPairs);
    sort(rob2.begin(), rob2.end(), UnsortPairs);

    if (rob1[col].first < rob2[col].first)
    {   closest = curRob;  }
}
} // end for loop

return closest;

/***************************************************************
* Function: UnsortPairs
*    used to sort robots' pref vectors by target, instead of
*    by distance. used in determining closest.
* *
* Returns:  bool
/***************************************************************/
bool UnsortPairs(const pair<float,string>& lft, const pair<float,string>& rht)
{
    return lft.second < rht.second;
}

/***************************************************************
* Function: DetermineOwnInitialPrefs
*    determines preferences for each known target,
*    currently based solely on distance.
* *
* Receives: ROBOTSMAP_DATA *robotsAndMap (ptrs to robots and map)
* *
* Returns: nothing
***************************************************************/
void DetermineOwnInitialPrefs(ROBOTSMAP_DATA *robotsAndMap)
{
    int myrow = (*robotsAndMap->me).location.row,
            mycol = (*robotsAndMap->me).location.col;
    float distance; //, max; // distance and max distance
    unsigned int curr, scurr; // used in debugging sorted prefs
    stringstream keystr; // used for key-x of pair in vector myPrefs
    int numOfTargets = (*robotsAndMap->targets).size();
    (*robotsAndMap->me).prefs.clear(); // clear out the current prefs
    (*robotsAndMap->me).prefs.erase((*robotsAndMap->me).prefs.begin(),
                                    (*robotsAndMap->me).prefs.end());

    if (numOfTargets > 0 && myrow != 9999)
    {
        if (DISPLAY_MSG_TO_CONSOLE)
            { cout << "\nCalculating my preferences..."; }

        // calculate distance between robot and each target
        for (int current = 0; current < numOfTargets; current++)
        {
            distance = sqrt(
                            (pow((float)myrow-(*robotsAndMap->targets).at(current).row, 2)) +
                            (pow((float)mycol-(*robotsAndMap->targets).at(current).col, 2)));

            // add rand thousandth to resolve "coin-flipping" issues resulting
            // from robots being exact same distance to a specific target.
            distance += ((float)rand()/RAND_MAX) * .01; // add rand thousandth
            keystr.str(""");
            keystr << current;
            // make_pair creates a pair<float,string> object and add to vector
            (*robotsAndMap->me).prefs.push_back(make_pair(distance, keystr.str()));
        }
    }

    if (DISPLAY_MSG_TO_CONSOLE)
    { cout << "DONE.\n"; }
}


APPENDIX H (continued)

```c++
if (DISPLAY_MSG_DEBUG)
{
    for (curr=0; curr<(*robotsAndMap->me).prefs.size(); curr++)
    {
        // order of how they were added
        cout << (*robotsAndMap->me).prefs.at(curr).second << "="
             << (*robotsAndMap->me).prefs.at(curr).first << endl;
    }
}

sort((*robotsAndMap->me).prefs.begin(),
     (*robotsAndMap->me).prefs.end()); // sort float first, then string

if (DISPLAY_MSG_DEBUG)
{
    cout << "Sorted: " << endl;
    for (scurr=0; scurr<(*robotsAndMap->me).prefs.size(); scurr++)
    {
        // sorted order ascending by 'x' component of the pair
        cout << (*robotsAndMap->me).prefs(scurr).first << "="
             << (*robotsAndMap->me).prefs(scurr).second << endl;
    }
}
```

} // end if numOfTargets

}