COOPERATIVE ROBOTICS USING WIRELESS COMMUNICATION

Except where reference is made to the work of others, the work described in this thesis is my own or was done in collaboration with my advisory committee. This thesis does not include proprietary or classified information.

_____

Adam A. Ray

Certificate of Approval:

_____	_____
A. Scottedward Hodel Associate Professor Electrical Engineering	Thaddeus A. Roppel, Chair Associate Professor Electrical Engineering
_____	_____
Stuart Wentworth Associate Professor Electrical Engineering	Stephen L. McFarland Acting Dean Graduate School

COOPERATIVE ROBOTICS USING WIRELESS COMMUNICATION

Adam A. Ray

A Thesis

Submitted to

the Graduate Faculty of

Auburn University

in Partial Fulfillment of the

Requirements for the

Degree of

Master of Science

Auburn, Alabama
December 16, 2005

COOPERATIVE ROBOTICS USING WIRELESS COMMUNICATION

Adam A. Ray

_____
Signature of Author

_____
Date of Graduation

VITA

Adam A. Ray, son of Travis and Susan Ray, was born in January 20, 1981, in Anniston, Alabama. He graduated from Talladega High School as the Salutatorian. He entered Auburn University in September, 1999, and graduated summa cum laude with a Bachelor of Science degree in Electrical Engineering in December, 2003. In January, 2004, he accepted a job with Bytewise Measurement Systems in Columbus, Georgia as an electrical engineer. Also in January, 2004, he entered graduate school at Auburn University. He married Rebecca L. Armstrong, daughter of James and Tanya Armstrong, on December 18, 2004.

THESIS ABSTRACT

COOPERATIVE ROBOTICS USING WIRELESS COMMUNICATION

Adam A. Ray

Master of Science, December 16, 2005
(B.S., Auburn University, 2003)

183 Typed Pages

Directed by Dr. Thaddeus A. Roppel

Cooperative Robotics has to do with the use of multiple robotic agents assisting each other to perform a task that is either too difficult or impossible for one robot to perform alone. It is a multi-disciplinary field that spans the areas of computer science, electrical engineering, and artificial intelligence. Research scenarios often include tasks that are difficult, monotonous, or dangerous for humans to perform.

This thesis presents a search-and-rescue algorithm, referred to as SARA-1, that is designed to enable a team of cooperative autonomous robots to search an area for a stationary target. The robots use wireless communication to build and share collective maps of the environment. They attempt to spread out their cooperative search, taking care not to explore the same area twice. This algorithm is pertinent to both indoor and outdoor applications. The range of applications is limited only by the user's imagination,

and might include such tasks as hazardous waste location and removal, planetary exploration, warehouse organization, and human search-and-rescue.

Several different experiments are performed using SARA-1 on the Player/Stage simulation platform, in two different simulated environments. Experiments involve varying the number of robots and the communication interval (how often the robots exchange data) to see how the time and success of a search-and-rescue task is affected. It is found, in the number of robots experiments, that there is a threshold at which the positive effects of cooperation are outweighed by inter-robot interference and communication overhead. Increasing the number of robots past this threshold has undesirable consequences including an increase in the time for task completion and a higher frequency of robot failure.

In the communication experiments, it is found that a group of robots using any communication interval is superior to a group of robots that does not communicate. Choosing the communication interval that is most efficient is shown to be a much more complex task. More frequent communication provides more cooperation between robots, but requires more overall time for task completion. Less frequent communication slightly speeds up the time for task completion, but produces more interference among robots, increasing the risk of failure. The optimal selection of the number of robots and the communication interval depends highly on the specific task that is being performed.

ACKNOWLEDGEMENTS

I am deeply indebted to Dr. Thaddeus Roppel for his time, advice, and encouragement. He has been a great inspiration to me over the past two years. I would also like to thank the members of the Cooperative Robotics Research team, especially Isaac, Aaron, Rama, Arun, and Joel, for their conversation, advice, and friendship. Special thanks to Schaoen Wu for hours of help with Linux and to Isaac Rieksts for learning the Player/Stage software with me. I am also very appreciative of Dr. A. Scottedward Hodel and Dr. Stuart Wentworth for their interest and advice during my research and thesis. Thanks to Dr. Hodel for pushing me to be confident in myself.

I would also like to thank my wife, parents, sister, and brother-in-law for their years of love and support. A final thanks to all of my friends at Bytewise for providing a relaxed and friendly place to work while I pursue my degree.

Style manual or journal used: Institute of Electrical and Electronic Engineers Journal style

Computer software used: Player/Stage, Microsoft Office Word 2003, Microsoft Office Excel 2003, MATLAB 7.0

TABLE OF CONTENTS

NOMENCLATURE

Acronyms

CRR	Cooperative Robotics Research
SARA-1	Search-And-Rescue Algorithm, version 1

Equation Variables

$\alpha_{x,y}$	temporary variable for the grid cell $[x, y]$; used in an effort to separate and simplify the map-combination equations of section 4.5.4
$\alpha^{i}_{x,y}$	temporary variable for the grid cell $[x, y]$ and robot i; used in an effort to separate and simplify the map-combination equations of section 4.5.4
$C_{x,y}$	cost of reaching the grid cell $[x, y]$ from a robot's current position
$C_{x+\Delta x, y+\Delta y}$	cost of reaching the grid cell $[x + \Delta x, y + \Delta y]$; used for finding the minimum cost of reaching a grid cell and its surrounding eight neighbors
$C^{i}_{x,y}$	cost of reaching the grid cell $[x, y]$ for robot i
d	a sonar reading of a particular distance in the range (0,5 meters)
d_j	a particular sonar reading associated with the index j
$h(d_j)$	the number of times that a sonar reading, d_j, was measured by a robot
i	index number assigned to a particular robot
N	the total number of robots

$P(occ_{x,y})$	the probability that grid cell $[x, y]$ is occupied by an obstacle
$P(occ_{x+\Delta x,y+\Delta y})$	the probability that grid cell $[x + \Delta x, y + \Delta y]$ is occupied by an obstacle; used for the calculations of the probability of occupancy for a grid cell and its surrounding eight neighbors
$P(occ^{i}_{x,y})$	the probability that grid cell $[x, y]$ is occupied by an obstacle in the occupancy grid of robot i
$T(x_G, y_G)$	the transformation of a goal point, $[x_G, y_G]$, into a robot's coordinate system
$U_{x,y}$	the utility of the grid cell $[x, y]$ (utility is defined in section 4.5.5)
$V(d)$	visibility of a sonar reading of a particular distance, d (visibility is defined in section 4.5.5)
$[x, y]$	two-dimensional grid cell coordinates
$[x_G, y_G]$	two-dimensional Cartesian coordinates of a goal point
$[x_i, y_i, \theta_i]$	translational and rotational location of a robot [meters, meters, radians]
Δx	the shift of a grid cell x-coordinate
Δy	the shift of a grid cell y-coordinate
$\%Diff$	percent difference between two variables being compared

Units

cm	centimeter
m	meter
m^2	square meter
m/s	meter/second

Behaviors

communication	allows robots to share information about the environment and task
dispersion	allows robots to spread out during the initial moments of a search-and-rescue task
homing	directs a robot toward a specific node on a planned path
map-building	allows a robot to create a grid-like representation of its two-dimensional environment
obstacle-avoidance	directs a robot away from an obstacle
path-following	allows a robot to follow a pre-determined path
path-planning	allows a robot to locate unexplored areas of a global map, and plan a path to those areas
robot initialization	allows a robot to determine the number of functional robots at the beginning of a search-and-rescue task, and initialize channels for communication with each of those robots
target-homing	allows a robot to go to the target once it is found

Experiments and Environments

Broun Hall	the complex, simulated environment used with the search-and-rescue experiments, based on part of the third floor floor-plan of Broun Hall, Auburn University
continuous communication	a communication experiment which allows robots to exchange maps each loop of the code
no communication	a communication experiment that does not allow robots to exchange data
occasional communication	a communication experiment which allows robots to exchange maps every third loop of the code
Simple Rooms	the simple, simulated environment used with the search-and-rescue experiments

Robot States

search state	the first part of a search-and-rescue task in which robots search in parallel for a target
rescue state	the second part of a search-and-rescue task that is initialized when any robot finds the target; while in this state, all robots attempt to navigate to the target's location and surround it

Terms

cost	a mathematical value associated with the effort that a robot will expend to get from its current location to a remote point; takes into account the distance to the point and the obstacles that lie in between
cost grid	a grid of values representing the cost of going to each cell in an occupancy grid
direct combination	a way of summing the effects of robotic behaviors by running them concurrently on a processor and summing their outputs in a mathematical expression
explicit communication	communication that occurs for the sole purpose of transmitting data between two robots
frontier cell	a grid cell on the boundary of a robot's explored environment
frontier region	a group of grid cells that represents a large enough area to be of interest to a robot
grid cell	an $[x, y]$ coordinate in an occupancy grid representing some amount of two-dimensional space in a robot's environment
implicit communication	communication that occurs when a robot senses the actions of another robot; a "through the world" approach to communication

occupancy grid	a grid of probabilistic values representing the probability of occupancy for each grid cell in a robot's environment; this provides a dynamically updated map of the robot's environment
occupied	a grid cell that has a high probability of occupancy—i.e. it contains a probabilistic value in the range (0.5,1)
open	a grid cell that has a low probability of occupancy—i.e. it contains a probabilistic value in the range (0,0.5)
probability of occupancy	a value in the range (0,1) that represents the probability that the associated grid cell contains an obstacle
temporal combination	a way of summing the effects of robotic behaviors by running them separately and switching between them
unknown	a grid cell that has a probability of occupancy equal to 0.5
utility	the amount of unexplored area that a robot can cover with its sensors upon reaching a frontier region; this is used to spread the robots out while in the *search state*
visibility	the probability that a robot's sensors can cover objects at a certain distance—i.e. $V(d)$

LIST OF FIGURES

LIST OF TABLES

CHAPTER ONE: INTRODUCTION

On September 12, 2001, in the aftermath of the World Trade Center terrorist attacks, mobile robots made their first appearance in an actual human search-and-rescue mission [1]. While fully autonomous robots are not yet practical, mobile robots assisted rescue workers in locating more than two percent of the victims that were discovered. These robots were sent into tiny crevices, used to explore buried rooms with camera vision, and were well accepted by the rescue community. While this was a landmark event for the multi-disciplinary field of robotic search-and-rescue, it also showed the great need for further research and development. Fully autonomous and cooperative robots remain an unrealized goal for researchers worldwide [3].

Robotic search-and-rescue research has typically focused on either human-robot interaction or single autonomous robots. Many of the robots used at the World Trade Center were remotely controlled by humans or had network cables trailing behind them that were used to transmit camera images [21]. The robots that did have some autonomy were sent out alone or in non-cooperative pairs creating a high risk of failure. What is lacking in the field of robotic search-and-rescue is a team of human-independent cooperating robots that, with the press of a single button, can be sent out on a mission with a high level of confidence.

This thesis presents a search-and-rescue algorithm, referred to as SARA-1, for a group of fully autonomous robots that uses inter-robot cooperation to accomplish the assigned task much more efficiently than could be achieved by a single robot. During the first part of the task, the robots are in a *search state*. While in this state, the robots search in parallel for a single, stationary target. In a collaborative effort, the robots spread out and do not cover the same area twice. The first robot to find the target communicates its location to the other robots. This causes all robots to enter a *rescue state*. While in this state, all robots navigate to and surround the target. To avoid the complex physics of cooperative object manipulation, the search-and-rescue task is defined as complete when all robots are within a 2-meter radius of the target.

The task described above requires an algorithm that is capable of communication, location-awareness, path-planning, and the ability to distinguish obstacles, other robots, and the target. The proposed algorithm, SARA-1, uses local maps that are built and shared by the robots in a decentralized manner to provide many of the functions necessary to complete the search-and-rescue task. SARA-1 requires that the robots start with an awareness of their relative locations in order to have a reference point with which to build maps. This requirement is not a limitation in many search-and-rescue applications.

SARA-1 is robust and scalable. It is robust in the sense that it does not break down with the loss of robots or the degradation of communication. Each robot runs the same code and is capable of performing the task alone in the event that communication or team members fail; however, when either of these events occurs, the benefits of

cooperation are lost. SARA-1 is scalable in the sense that any number of robots can be used depending on the requirements of the specific task.

SARA-1 incorporates the concepts of a relatively new field known as cooperative robotics [23, 28]. Cooperative robotics deals with the study of multiple autonomous agents "working together" to perform some task that is either too difficult or impossible for one agent to perform while acting alone. It merges the disciplines of computer science, electrical engineering, and artificial intelligence. Research scenarios include box-pushing [33], exploration [39], area mapping [41], fire-fighting [9], hazardous waste removal [27], water treatment [35], and mine detection [43].

Liu and Wu [23] suggest that the use of multiple cooperating robots has several advantages over single-robot systems. These advantages include greater efficiency, a wider defined task domain, inherent parallelism, and distributed sensing. Also, building several cheap "off-the-shelf" robots can be less expensive than one large, custom-made robot [44]. This is particularly beneficial in hazardous environments where a failed robot may never be retrieved. Another benefit of many multi-robot systems, including the one presented in this thesis, is their decentralized nature. Decentralization produces a robust algorithm that is more fault-tolerant than a centralized approach [24].

There have been few attempts to incorporate ideas of cooperative robotics into search-and-rescue tasks. Jennings *et. al.* [24] present two large robots searching for and "rescuing" warehouse-like boxes. However, this task is performed in a single, obstacle-free room and involves no intentional cooperation; the robots simply move in random

patterns until one robot spots the target. Vainio *et. al.* [35] present a control architecture for a group of underwater search-and-destroy robots. While this experiment makes great use of unintentional cooperation to provide speed-up over a single-robot, more general applications would benefit from direct communication and intentional cooperation. Intentional and unintentional cooperation are discussed in more detail in Chapter Two.

SARA-1 is implemented using a software simulator known as Stage and a robot device server known as Player. The Player/Stage project [37] has become a very popular open-source tool in the cooperative robotics arena. Player runs under Linux, BSD, and several other Unix-based platforms and can be run on robots with embedded computers. Player provides a simple interface to a robot's sensors and actuators and optionally allows remote client programs to connect to them over a TCP socket. Stage is a two-dimensional simulator that provides virtual Player robots that can move, sense, and interact in any bitmapped environment. It is stated by the Player/Stage developers that "agents developed in simulation will work with little or no modification on the real devices and visa-versa" [5].

The research presented in this thesis is one step toward a long-term goal of the Cooperative Robotics Research (CRR) team at Auburn University [14]. Eventually, we wish to have several robots, each with embedded computers running the Player software and each robot performing cooperative tasks such as search-and-rescue. For this reason, SARA-1 is tailored to the projected needs and desires of the hardware side. The current state of the hardware development is discussed in Chapter Two of this thesis.

Following the implementation of SARA-1, several experiments are performed on two parameters: the number of robots and the communication interval. Performing experiments with the number of robots is a common topic of research in cooperative robotics [2, 34, 35, 40]. The question asked is "How many robots make this task most efficient?" It is important to define what is meant by "efficient". In most search-and-rescue tasks—certainly in urban search-and-rescue—the most important factor is how much time the task takes. This is assuming that the task is actually completed (i.e. time for task completion is not infinity). Therefore, the two metrics used to measure the efficiency of SARA-1 are how long the mission takes and whether or not each mission is successful.

Performing experiments with inter-robot communication is also a common topic of research in cooperative robotics [29, 33, 39, 40]. In experiments of this type, typical questions asked are "How much communication is necessary (if any)?" or "How often should robots communicate to produce the greatest efficiency?"

In the research described in this thesis, experiments are performed in two different simulated environments of differing complexity, referred to as Simple Rooms and Broun Hall. The number of robots is varied from one to seven and the communication interval is varied from *continuous*, to *occasional*, to *none*. The time required for task completion, the number of unsuccessful runs, and the time spent in each robotic behavior is presented and discussed.

The remainder of this thesis is organized as follows: Chapter Two gives an overview of the field of cooperative robotics. It is divided into several key areas of interest and concludes with a summary of the current research of the CRR team. Chapter Three describes the Player software and Stage simulator. Chapter Four presents SARA-1, the search-and-rescue algorithm. Chapter Five discusses the experiments performed and compares their results to other similar experiments found in the cooperative robotics literature. Chapter Six provides a summary, conclusions, and suggestions for future work.

CHAPTER TWO: LITERATURE SURVEY

The young and dynamically growing field of cooperative robotics has become a diverse research area that often seems to go in several different directions at once. Areas of interest range from high-level human-interactive robots [1] to biologically inspired autonomous gnat-like agents [4]. In the past fifteen years many different research areas have emerged, each generating significant amounts of progress. However, the field is so new that no topic area within cooperative robotics can be considered mature [28].

This chapter introduces the key areas of research within the field of cooperative robotics that pertain to the work presented in this thesis. The first two multi-agent robotic systems are presented in the Origins section, followed by six main areas of research: Approaches, Biological Inspirations, Behavior-Based Robotics, Communication, Mapping and Exploration, and Object Transport and Manipulation. The chapter concludes with the Cooperative Robotics Research section that presents the work of the CRR team at Auburn University.

There are several other key areas in the field of cooperative robotics that are outside the scope of this thesis. Parker [28] presents progress in the areas of localization, motion coordination, reconfigurable robots, and multi-robot learning.

2.1 Origins

As Liu and Wu [23] suggest, the field of cooperative robotics began in the late 1980's when researchers began investigating issues in multiple mobile robot systems. Up to this point, most of the research had focused on either single robot systems or distributed problem-solving involving non-robotic components. When these two ideas were merged, the field of cooperative robotics (also referred to in the literature as distributed robotics) was born. The work of two of the groups to first present the ideas of distributed robotic systems is presented below.

Fukuda and Nakagawa [45] introduce the idea of a dynamically reconfigurable robotic system (DDRS) which allows a robot to autonomously reconfigure its parts based on the goals of a specific task. DDRS consists of robotic "cells" which are defined as fundamental components with a single mechanical function such as a mobile base, gripper, or arm joint. These cells communicate with each other and can approach, detach, and combine themselves in different ways depending on task definition and allowable workspace. The research is motivated by biological cells which, although they have simple single functions, show very complex and new behaviors when combined in groups. DDRS is proposed for use in space, factory, and hostile environments. This theoretical research progresses to an actual robotic system called CEBOT [46].

Asama *et. al.* [18] present ACTor-based Robots and Equipments Synthetic System (ACTRESS) which is a distributed multi-robot system designed for maintenance

tasks in nuclear power plants. The autonomous components of this system are termed "robotors" and can be mobile robots or any component that has at least two basic functions: 1) the ability to sense surroundings, make decisions, and act on these decisions and 2) the ability to communicate to other robotors for purposes of cooperation and interference avoidance. ACTRESS is shown in simulation with the cooperative task of two mobile robots pushing boxes to the sides of a room.

Inspired by the novel ideas introduced by these two groups and a few others, the field of cooperative robotics grew rapidly. The field gained popularity due to its applications in hazardous environments and the promise of performing tasks more efficiently. Several different approaches to cooperation emerged early on and have remained with the field to this day. These approaches are presented in the next section.

2.2 Approaches

Most of the work in cooperative robotics can be categorized into two approaches: swarm-type cooperation and intentional cooperation [27]. The swarm-type approach deals with a large number of lower-level robots that are typically (though not always [32]) unaware of each other's actions. Tasks proposed for swarm robots usually have a parallel nature such as collecting rock samples on Mars and sorting mail. Cooperation occurs due to the statistical result of a large number of repeated actions. The goal in the swarm approach is to design each robot's control laws such that numerous simple interactions with the environment produce a globally desirable behavior.

Intentional cooperation, on the other hand, typically deals with a limited number of higher-level robots in which each robot is not only aware of the actions of other agents, but uses this information to determine its own best action [16]. Task examples include moving furniture and building space stations. Each robot's interaction with the environment or other robots has purpose and contributes to the robot's predefined goals. Cooperation occurs on local and global levels, as opposed to the swarm approach in which there is typically only global cooperation. Also, in intentional cooperation approaches, there are often numerous goals to achieve in a logical or optimal order [24].

It is often difficult to distinguish between swarm and intentional approaches because many examples of applications have characteristics from both categories. One possible way to distinguish between the two approaches is to apply Matarić's definitions of explicit and implicit cooperation [31]. Explicit cooperation occurs as a result of one agent performing actions to benefit another agent's goals. In contrast, implicit cooperation occurs as a result of selfish motivations that help an agent to achieve its own goals, but also have an effect on the environment that help other agents achieve their goals as well. Swarm approaches tend to take advantage of implicit cooperation, whereas intentional cooperation approaches require explicit cooperation.

Research in cooperative robotics can be further categorized according to the type of robots that are used: homogeneous or heterogeneous [31]. Homogeneous approaches consist of identical robots running identical code whereas heterogeneous robots may run different code and often have different sensing and/or manipulation capabilities. Homogeneous robots allow for faster multi-robot design and are typically used in swarm

approaches. Heterogeneous robots take longer to design and are more difficult to debug in multi-robot systems, but allow for more specialized tasks. For this reason they are often used in intentional cooperation approaches. However, homogeneous robots are also used in many intentional cooperation approaches [33].

The definitions of swarm vs. intentional cooperation and homogeneous vs. heterogeneous robots are not presented as a means of categorizing all approaches. They are simply a way to introduce the general ideas and approaches used by many researchers in the field. Examples from the literature of several different combinations of these approaches are presented in the following paragraphs.

Matarić [32] uses a group of twenty homogeneous robots in a swarm approach to show the benefits of increased awareness during a homing behavior. She gives results from three experimental cases: ignorant coexistence, informed coexistence, and intelligent coexistence. The ignorant case uses a traditional swarm approach in which robots are not able to distinguish obstacles from other robots. In this case, robots spend a significant amount of time avoiding each other. In the informed case robots are able to detect and yield to each other. They consequently spend less time avoiding and more time homing. In the intelligent case, robots can detect other robots in a 36-inch radius and measure the local population density in an effort to avoid the two extremes of collision and isolation. Each increased level of awareness improves the overall time for task completion.

Halme *et. al.* [2] use both swarm and intentional cooperation in a simulated stone-collecting experiment. In this experiment, there are two different types of robots: work units which collect stones and support units which carry energy to the work units. The work units use implicit cooperation (swarm), whereas the support units use explicit cooperation (intentional). Kube and Zhang [9] apply a homogeneous swarm approach to a group of box-pushing robots and a simulated follow-the-leader task. The selfishly motivated robots push the box as if there are no other robots there, even though the box is too large for one robot to push alone. An interference avoidance algorithm keeps the robots from hitting one another and a backward motion detector keeps a robot from pushing against the others.

Hutin *et. al.* [36] take a heterogeneous intentional cooperation approach in two simulated experiments: exploration and mapping, and convoying (follow the leader). Each experiment uses five agents—three robots, a map builder, and an interface. While the robots are homogeneous, the overall system is heterogeneous. Dadios and Maravillas [16] use an intentional cooperation approach with two homogeneous soccer-playing robots performing a pass-shoot task. Matarić *et. al.* [33] use a homogeneous intentional cooperation approach on two six-legged box-pushing robots. It is compared to both a single robot approach and two non-cooperating robots.

SARA-1 uses homogeneous robots in an intentional cooperation approach. The *search state* is parallel in nature and could possibly benefit from a swarm approach. However, as is shown in the experimental results in Chapter Five, an increase in the

number of robots produces a negative effect due to inter-robot interference, even with the help of intentional cooperation.

2.3 Biological Inspirations

There are numerous examples of biological societies that achieve collective tasks [9]. Many researchers in the field of cooperative robotics have attempted to mimic in robots the behaviors observed in these societies [28]. Replicating the behaviors of various biological societies has led to an entire field of research known as behavior-based robotics [30]. Behavior-based robotics is the foundation of much of the multi-robot work over the past fifteen years and will be explored in detail in the next section. Several examples of applying biological inspirations to multi-robot teams are presented below.

Kube and Zhang [9] define a list of five biologically inspired mechanisms for invoking group behavior: non-interference, following, responding to environmental cues, group realization, and auto-stimulation. Non-interference and following are closely related to behaviors (as opposed to behavior triggers) and are discussed in the next section. The last three behavior triggers can easily be mimicked in the robotic world to obtain very useful results. Many insects use the environmental cue of sunlight to begin foraging tasks at dawn. In a like manner, the environmental cue of a fire could invoke a group of robots to begin firefighting tasks. Many insects use a group realization mechanism as a trigger to perform different tasks depending on their proximity to the rest of the group. For example, ants will quit working when taken away from their bed. In

the fire-fighting example, robots in a near vicinity to the group could be responsible for aiming water toward the center of the fire, whereas robots that found themselves away from the group could invoke a behavior to manage hoses or to get more supplies. The auto-stimulation allows insects (or robots) to self-invoke certain behaviors. In the firefighting example, a robot that noticed the water getting low could self-invoke a behavior to get more water, even if it was in the vicinity of the group.

Many researchers have focused their efforts toward the study of ant colonies. Halme *et. al.* [2] give their robots an ant-inspired "energy equalizing" ability that allows two robots (or ants) to equalize their energy when they meet. In simulation, this ability allows for a more efficient use of energy and results in less "dead" robots at the end of a stone collecting task. Kube and Bonabeau [11] study ants that cooperate to move large prey. The ants systematically rearrange their positions until the prey can be lifted and transported back to the bed. This behavior is mimicked in a group of box-pushing robots. At the time of this writing, videos of Kube's box-pushing robots can be found at [10].

Biological inspirations have greatly impacted the study of cooperative robotics, and have given birth to the field of behavior-based robotics which will be discussed next.

2.4 Behavior-Based Robotics

Behavior is a biologically inspired term defined as a regularity in the interaction of a robot with its environment [33]. Behaviors need not be complex; as Kube and Zhang [9] observe, many behaviors can be realized in combinational circuits. Examples of

simple behaviors include wandering and obstacle avoidance. In a typical robot system, behaviors are embedded in the control architecture and are intended as building blocks for achieving higher-level goals. In multi-robot systems, the simple and complex behaviors of each robot are combined to form a group behavior that is both new and desirable [12].

To make this point clear, consider two robots that have the task of pushing a box from point A to point B. One robot is a pusher and the other is a steerer. Notice that this is a heterogeneous, intentional cooperation approach. On a single robot level, the pusher will have a contact behavior that will keep it in contact with the box and a drive behavior that will keep it in forward motion. Taken individually, these behaviors accomplish nothing, but when combined the robot will push a box. Likewise, the steerer will have a homing behavior to get to point B and some behavior to keep the box between itself and point B. On the multi-robot level, the pushing and steering behaviors of each robot are useless when acting alone; but when combined, the task is accomplished.

Behavior-based robotics started in the late 1980's after Rodney Brooks [38] proposed a new method, known as subsumption architecture, for controlling autonomous mobile robots. This architecture allows for increasing levels of competence that run concurrently and whose results are joined to form a single action. Each level of competence forms a layer of the control system. For example, a first level of competence could be obstacle avoidance and a second level of competence could be to wander around while avoiding obstacles. Once the first level is built and debugged, then the second level is built on top of the first level; it does not replace the first level, but uses

information from it to achieve the next level of competence. The levels communicate asynchronously and higher priority levels can suppress outputs or ignore inputs from lower priority levels. Each level was originally intended to run its own separate processor. This provides for great extensibility albeit at high cost.

Outside of Brooks' subsumption architecture control schemes typically lean toward one of two extremes. One extreme is a planner-based strategy that involves a world model and a centralized planning approach for all robots involved in a task. The opposing extreme is a purely reactive strategy that relies on direct coupling between sensors and actuators. Behavior-based strategies follow a subsumption-style approach and generally fall between these two extremes [30]. While behavior-based approaches are distributed and decentralized like the reactive approaches, their computation is not limited to lookup.

Matarić [31] divides behavior-based robotics into two categories based on how the behavioral layers are combined: direct combination and temporal combination. Approaches in the direct combination category execute multiple behaviors concurrently in a strict subsumption-style approach. The behaviors are summed using a mathematical function. Approaches in the temporal combination category execute only one behavior at a time and switch between them in a more reactive-style approach. This is not a strict reactive approach since identical sensor readings will produce very different results depending on the current and recent behaviors.

The remainder of this section provides examples from the literature of behavior sets for different tasks and implementations of different control architectures. Strict planner-based architectures are not distributed by nature and therefore fall outside of the mainstream of cooperative robotics research. For this reason, they are not discussed any further in this thesis. Several examples of strict planner-based strategies are given in [30].

Matarić [31] offers a set of six behaviors for mobile robots interacting and moving around in a two dimensional plane: avoidance, wandering, aggregation, dispersion, following, and homing. Avoidance is a commonly used behavior that keeps robots from hitting obstacles and each other. A robust avoidance behavior would also deal with accidentally hitting an obstacle. Wandering is a behavior that keeps a robot in motion. Aggregation and dispersion enable robots to form a group (as in flocking) or to spread out (as in foraging). Following is a behavior that is useful to avoid collisions, and homing directs robots to a goal location (the goal is not always stationary). Matarić performs many experiments on the combinations of these behaviors and compares the results of direct and temporal combinations to centralized planner-based approaches.

Kube *et. al.* [12] propose a set of four behaviors necessary for a box-pushing group of robots: *find, slow, goal, avoid*. His approach is a selfishly-motivated swarm approach in which each robot is attempting to push the box by itself, as opposed to the intentionally cooperative push-steer method mentioned earlier. The *find* behavior keeps the robot in motion and *avoid* keeps it from colliding with other robots. The *Slow* behavior allows the robot to push the target or box without causing damage to itself. The

Goal behavior keeps the robot moving the box toward a lighted goal. The transition in these behaviors is controlled through environmental cues.

Parker [27] has developed the ALLIANCE architecture based on a strict subsumption-style approach. Her architecture is non task-specific and is designed for maximum fault tolerance. She adds two novel mathematically-modeled motivations called *impatience* and *acquiescence*. *Impatience* is a parameter associated with a particular behavior set that initializes to zero and grows as long as a particular task is not being accomplished. When it reaches a threshold, a robot may decide to take over that particular task. While *impatience* reflects the fact that other robots may fail, *acquiescence* reflects the fact that a robot itself may fail. When this parameter reaches a threshold, a robot may decide to give the task over to another robot. These parameters work together to solve problems such as malfunctioning hardware and the sudden need for additional tasks to be performed.

Asama *et. al.* [19] use a reactive control approach on the ACTRESS robotic system in which a temporary master/slave configuration is set up between two robots on a collision path. A series of rules are applied that progress from communicating in an attempt to negotiate movement, to using a deadlock solver on a remote computer, and finally to problem solving by a human operator.

Kube *et. al.* [12] propose an adaptive logic network (ALN) for behavior arbitration. They point out that as the complexity of a collective task increases the problem of behavior arbitration increases as well and that a subsumption architecture

only works well when the interaction and complexity of behaviors is low. Their methods are tested on a simulated group of box-pushing robots in which the robots are manually guided through the task in order to train the ALN's.

SARA-1 uses a behavior-based approach with both temporal and direct combination of behaviors. For the *search state*, the behaviors include *robot initialization, dispersion, obstacle-avoidance, map-building, path-planning, path-following,* and *communication*. The *rescue state* uses the *obstacle-avoidance, map-building, path-planning,* and *path-following* behaviors as well as one additional behavior: *target-homing*. The specifics of SARA-1 and its internal behaviors are discussed in Chapter Four.

2.5 Communication

Communication is a very popular area of study in cooperative robotics, especially since methods for communication are changing rapidly. Instead of presenting all of the different methods and their pros and cons, we will first discuss the two distinct types of multi-robot communication and then give a few examples of each from the literature. It is important to note that applications taking advantage of either form of communication generally fall into the intentional cooperation category.

Two distinct types of communication are commonly used in many types of cooperative tasks, whether robotic or biological. They are usually referred to in the robotic literature as explicit and implicit [28]. Explicit communication occurs with the

sole purpose of transmitting a message, such as speech, growls, gestures, or radio transmissions. Implicit communication, referred to as stigmergic communication in biology [31], occurs through an awareness of the side-effects of other actions—a "through the world" approach.

As an example, consider a human trying to lift a box. If his arms started shaking and a look of strain came upon his face, you might guess, through implicit communication, that the box is too heavy and he needs help. If he were to yell, "Help!", this would be explicit communication. Similarly, consider two robots unloading a truck, one taking boxes off the truck and the other stacking them in a storage location [27]. If the stacking robot runs out of boxes to stack, it might realize, through implicit communication, that either the unloading robot is having trouble or that the truck is empty. The robot could distinguish between these two possibilities if a blinking light was triggered when the truck became empty. This would be explicit communication.

Dadios and Maravillas [16] use implicit communication in a team of two soccer-playing robots. They use fuzzy logic and an overhead camera for navigation. Fuzzy logic incorporates into robotic algorithms the idea that humans communicate in non-exact concepts as opposed to crisp numbers—right/left for example. The overhead camera is used by both robots for implicit communication, but the robots are not allowed to explicitly communicate with each other. In simulation the robots are able to effectively pass and shoot the ball into a goal in the presence of opposing team members represented by stationary obstacles.

Asama *et. al.* [19] use explicit communication by putting laptops, equipped with wireless modems, on each mobile robot. Although this is an expensive and bulky approach it allows a large amount of information, such as global maps, to be passed and processed. Simsarian and Matarić [25] use explicit communication by equipping two box-pushing robots with radio communication. They send "my turn, your turn" messages along with each robot's sensory data and are able to successfully push a box toward a moving infrared-emitting source.

Researchers using explicit forms of wireless communication must remember that these types of communication are not always reliable in certain environments. Arkin and Diaz [39] attempt to solve this problem in a multi-robot exploration task with the constraint that the robots must maintain line-of-sight. In an experiment called "anchored wander" one robot moves at a time until the line-of-sight is lost. At this point, the robot backtracks until line-of-sight is regained and becomes a communication "anchor" for the next robot to move beyond the first robot, and so on.

While much of the research in multi-robot communication is devoted to the type of communication that is used, another popular research area focuses on how much communication is optimal for a particular task. Arkin *et. al.* [40] explore how the amount of communication affects two robots on a puck-gathering mission. This simulated experiment allows cooperative carrying of one puck to produce speedup. They compare the absence of communication with simple communication of the "behavioral state", a three-parameter message consisting of the robot's coordinates and current behavior. This

simple communication protocol allows for more efficient work and less random wandering than in the non-communication approach.

Matarić *et. al.* [33] perform communication experiments on two box-pushing robots. In a non communication experiment, each robot tries to push the box toward a goal using a reactive control strategy as if it were the only robot pushing. This is compared to a second experiment in which a simple communication protocol is used to enable the robots to intentionally cooperate in pushing the box. With no communication, the actions of one robot frequently undo the actions of the other and result in either pushing the box out of bounds or abandoning it. Using simple communication, the robots are able to push the box to the goal in every run.

Inspired by the experiments described above, several different communication strategies for SARA-1 are explored and presented in Chapter Five. The code uses explicit wireless communication to pass information used in cooperative map-building and path-planning. The specific protocol for message-passing is described in Chapter Four.

2.6 Mapping and Exploration

Exploration can be a time-consuming part of many cooperative tasks and, quite often, time is of great importance (i.e. search-and-rescue). Therefore, it is important in multi-robot approaches to ensure that additional robots enhance the time for exploration over a single robot approach [41]. One may assume that the addition of robots inevitably

speeds up an exploration task. However, without coordination all robots might follow the same path and require the same amount of time as a single robot. Even worse, interference between robots may cause the exploration to take more time than with a single robot. Therefore, a key problem in multi-robot exploration is to cause the robots to simultaneously explore different areas of the environment [49].

A simple, common-sense approach to spread robots out in an exploration task is to force a robot to do an about-face when another robot is encountered. Matarić [32] takes this approach a step further and allows robots to sense each other within a certain radius and to calculate a population density. The robots can then move to areas of less density. While these approaches do help to spread out the robots, they do not ensure that the robots will move on to explore different areas. They may just move back and forth in the same area, attempting to remain spread out.

The next level of complexity in multi-robot exploration involves the development of some form of a shared map that tells not only where robots are, but where they have been. Most of the work in multi-robot exploration deals with map-building of two-dimensional environments and use either range sensors (such as sonar and laser) or camera vision [28]. In many applications, the map-building takes place on each robot in a decentralized approach [7]. While it may be faster to process images and maps from multiple robots in a remote location [36], this does not have the advantages of a decentralized system, one of the main reasons for using multiple robots in the first place.

Before proceeding to examples from the literature, it is necessary to define the idea of an occupancy grid. As a representation of the robot's environment, an occupancy grid is a grid of equally spaced cells that store the probability that the representative cell is occupied by an obstacle [49]. The use of probability attempts to make up for faulty readings due to imperfect sensors and inexact robot locations. The grid is typically initiated to 0.5 for every cell, representing a completely unexplored map, and the probability of occupancy is updated as each robot explores its environment. Burgard *et. al.* [49] offer a method for integrating maps from multiple robots. The method assumes that the robots know their relative positions.

Yamauchi [7] presents the idea of frontier-based exploration. He labels the cells of an occupancy grid as open, unknown, or occupied according to their probabilistic values. He then defines a frontier cell as any open cell that is adjacent to an unknown cell. With this information, a robot can go to one of these open cells in order to explore the frontier cell. For maximum efficiency a robot chooses an area of the occupancy grid with the maximum number of frontier cells to explore. The algorithm is demonstrated using two real robots.

Burgard *et. al.* [49] build on frontier-based exploration by allowing a robot to calculate the tradeoff between the cost and utility of reaching a frontier cell. The cost is defined by how much effort it will take to reach a frontier cell and the utility is defined as the size of the unknown area a robot can explore upon reaching the frontier cell. Each robot is assigned its next frontier cell to explore using a simple formula, calculated by all robots, that attempts to minimize cost and maximize utility for each robot. The algorithm

is implemented on two and three robots in a large simulated area and on two real robots in a smaller area.

Simmons *et. al.* [41] take a slightly different centralized approach. Local occupancy grids are built and processed by each robot. Every tenth map is sent to a remote computer and used for global map-building and path-planning. Each robot constructs "bids" consisting of estimated utilities for the robot to travel to various locations. The bids are sent to and processed by the centralized path-planner which assigns tasks to each robot. This approach is tested on real robots exploring much larger areas (62 x 43 meters) [41] than the decentralized approach (15 x 8 meters) [49].

The search-and-rescue code presented in this thesis implements frontier-based exploration and uses the cost/utility formulas presented in [49]. Occupancy grids built by simulated robots are shown in Chapter Four.

Another popular area of research that coincides with mapping and exploration is localization. Localization deals with the ability of a robot to determine its location based on information such as nearby landmarks and maps. Localization is outside the scope of this thesis since it is assumed in the search-and-rescue code that all robots begin with the knowledge of their positions relative to one another. For a good list of references dealing with localization issues, see [23, 28].

2.7 Object Transport and Manipulation

Due to a limitation of the simulation software that is used to demonstrate the search-and-rescue task presented in this thesis, no object manipulation will be performed. However, since it would be a useful addition to any search-and-rescue algorithm, some examples of the current research in object manipulation are presented in this section. It is a goal of the CRR team at Auburn University to include this ability in future cooperative tasks.

Most of the research dealing with object manipulation focuses on the physics of grasping, lifting, etc. in single robot applications and has not yet been incorporated into the cooperative robotics arena. A popular alternative that focuses on inter-robot cooperation rather than device physics is box-pushing. Several researchers have chosen this cooperative task and their approaches and experiments are discussed below.

Jennings *et. al.* [24] use a push/steer method with two real robots moving large warehouse-like boxes. The pusher pushes the box in a trajectory specified by the steerer. The steerer works to keep the object pinned between the two robots and heading in the correct direction. In this method, the box can be reoriented as well as relocated.

Kube and Zhang [9] use a swarm approach with multiple robots pushing a lighted box that is too heavy for one robot to move by itself. The pushing is accomplished by the selfish motives of each robot. However, a progression sensor is added to ensure that a robot is not pushing against the movement of the box. In [9] there is no goal location to push toward, but this ability is added in [12].

In contrast to Kube and Zhang's swarm approach Matarić *et. al.* [33] use an intentional cooperation approach to box-pushing with two six-legged robots. They use a synchronized "my turn/your turn" algorithm which involves two behaviors: a correcting behavior that keeps the robot in contact with the box and a light-following behavior that keeps the robot moving toward a goal. The robots must begin in contact with the box. If the box is abandoned the robot will not be able to find it again. Simsarian and Matarić [25] add learning and a moving target to this approach.

2.8 Cooperative Robotics Research at Auburn University

The work presented in this thesis is one of many collaborative projects of the Cooperative Robotics Research (CRR) Team at Auburn University. This section briefly describes several of the other projects that are currently underway. At the time of this writing, more information about these projects, including videos, can be found at [14].

On the software side, several cooperative behaviors have been designed and implemented using the Stage simulator. SARA-1 takes advantage of several of these behaviors including avoidance and homing. A follow-the-leader task has also been implemented in which a line of up to three robots follow a leader around a single room while avoiding carefully placed, large obstacles. Getting the robots to follow each other out of a room and around corners is a difficult programming task and is left to future research.

On the hardware side, the first robot platform has been designed and built and is shown in Figure 1. GRR-1 (General Reconnaissance Robot) is equipped with a Basic Stamp microcontroller, a sonar sensor, and two infrared sensors. Several behaviors have been implemented on GRR-1 including obstacle-avoidance, infrared beacon-tracking, and object-following. The next projected step is to build two more identical robots and explore multi-robot cooperative tasks such as follow-the-leader. Follow-the-leader would take advantage of both the obstacle-avoidance and object-following behaviors.

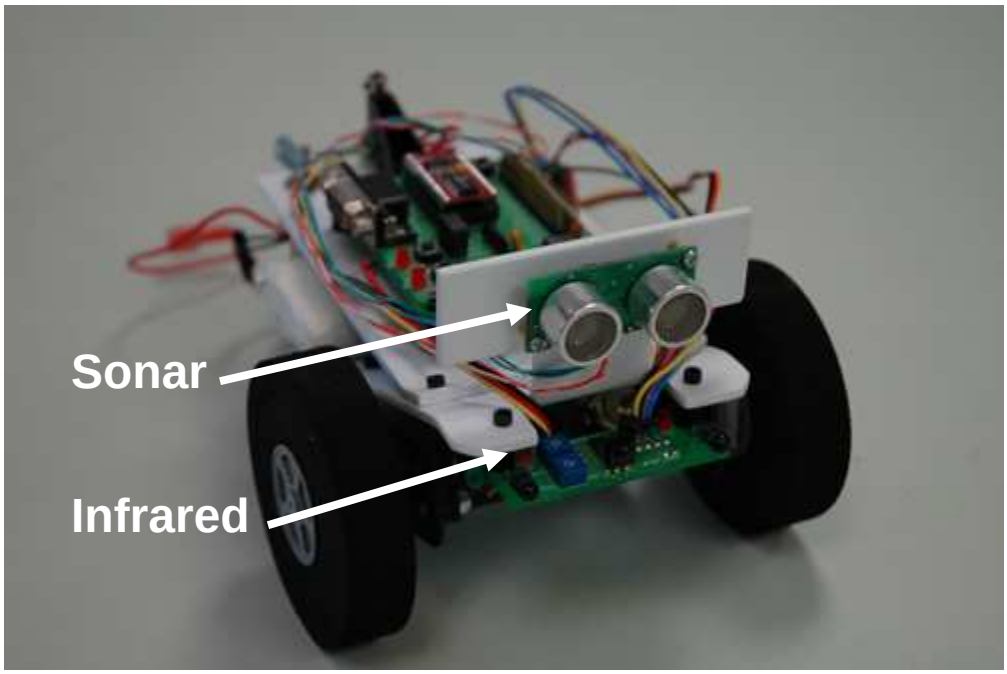


Figure 1: GRR-1: the first mobile robot platform

A second generation robot platform is currently in the design phase. It will have multiple sonar sensors and a Gumstix™/Robostix™ [17] embedded computer that will run the Player device server under Linux. The use of Player as an interface between the control programs and the robot will allow for faster development of code that is more

reusable. For example, in the event that a device or sensor is changed or upgraded, only the device driver will need to be rewritten—the control program will remain unchanged. Also, any control programs written for the robot will be tested and debugged in the Stage simulator first. This second generation robot platform is a major step toward the hardware implementation of many complex cooperative algorithms such as SARA-1 presented in this thesis.

In conjunction with the robotic software and hardware development, the CRR team is also investigating the use of sensor "motes" (wireless communication boards) on mobile robot platforms. The team currently has eight Moteiv Tmote Sky™ sensor motes [47]. The motes are equipped with several sensors such as temperature, humidity, and light sensors as well as onboard radio communication capabilities. These sensor motes are typically used in a wireless network to share environmental information between units and a base station. The CRR team hopes to incorporate these sensor motes with future generations of mobile robot platforms to allow inter-robot communication and possibly robot-human communication. The communication protocol used in SARA-1 was designed with these sensor motes in mind.

CHAPTER THREE: THE PLAYER/STAGE SOFTWARE

SARA-1, which forms the core of this research, is implemented using a robot device server called Player and a software simulator called Stage. The Player/Stage project has become a very popular open-source tool in many areas of robotics research and is available for download at [37]. There are other very powerful robot simulation tools available [15, 42]; however, to the author's knowledge, the Stage simulator is the only one aimed at multi-robot cooperation. Prior to the first release of the Player/Stage software in May 2001, many cooperative robotics researchers were forced to write their own simulation tools [9, 40]. These simulation tools tend to be very simple and task-specific. This chapter provides a brief overview of the Player/Stage software, its capabilities, and the simulated devices used by SARA-1.

3.1 The Player Server

Player can run on many Unix-based computers providing a simple interface to a robot's sensors, actuators, and other devices. The software contains many powerful function calls to non-specific drivers allowing the programmer to reuse control programs on different robots without re-writing any code—only the software drivers are updated.

As with many open-source tools, there is a vast community of users who share drivers and other software written for Player. The function calls are contained in client libraries and are currently available in C, C++, Tcl, Python, Java, and Common LISP [6]. SARA-1 uses the C++ client library.

Several instances of Player can be combined into a network, each instance being able to exchange data, send commands, or share computational load. A network of Player instances could range from several robots cooperating autonomously with each other to several robots cooperating with a centralized planner. Outside of the cooperative robotics realm, a network could consist of a single lightweight robot out in the field whose information is processed and whose actions are controlled by several powerful remote computers. Within the network, client programs can be written that "subscribe" (via TCP/IP) to a set of devices in order to control or manipulate them. Examples include robot controllers and sensor data processors.

To illustrate the power and versatility of Player, Figure 2 is reprinted from [5]. "P" represents an instance of the Player server, and "C" represents a client program. The scenario shown includes three robots patrolling a building while keeping a tight formation. The top robot runs a formation-control program by subscribing to the sonar sensors and wheel motors on all three robots. An audio processing algorithm runs on the right-most robot that subscribes to its closest teammate in order to find the direction of unusual sounds. While the robots are running their control programs, a researcher is debugging the formation-control program on a remote workstation. A data logger keeps track of the robots' sonar ranges and ground-truth positions by subscribing to an

overhead vision system. Meanwhile, another university across the country has access to a supercomputer that runs an online mapping client that subscribes to a wall-mounted laser scanner and each of the robots' sonar sensors. The map is available online for anyone to see. While this scenario is complex, [5] states that it is fully supported by the Player software and that many of the client programs are already implemented.

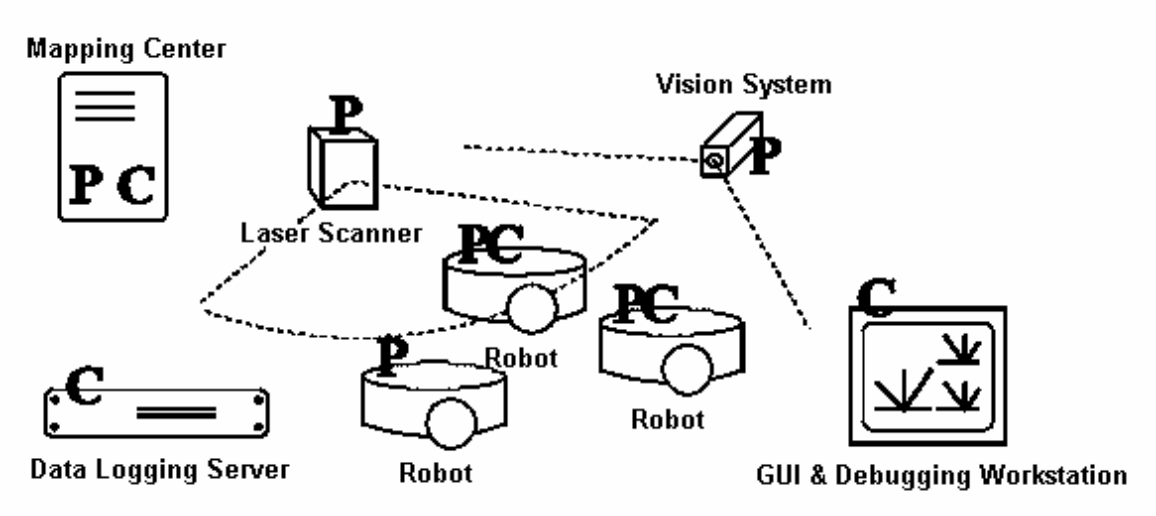


Figure 2: An example of the Player server (after [5], Fig. 1)

SARA-1 makes use of many interfaces and function calls available in the C++ client library. It does not, however, take advantage of the client-program "subscribing" ability of Player. Using this ability, each robot could subscribe to all other robots' sensors and Player would internally handle the data swapping at a predetermined rate. Alternatively, SARA-1 makes use of a simulated external wireless-communication device interface to handle all data-swapping between robots.

There are three main reasons for not choosing to take advantage of Player's internal client-program "subscribing" ability. First, SARA-1 requires single communication instances that occur at varying times—times that are initiated and coordinated by the robots. Player's predetermined, non-coordinated update rate (10 hertz by default) is not well-suited for this task. Second, the subscribing ability would only allow robots to share sonar data. A separate communication device and protocol would still be required for initializing the search and for signaling when the target is found. Finally, the CRR team currently has several sensor motes which are ideal for inter-robot communication. Therefore, SARA-1 relies on Player for its interfacing ability and function calls, but assumes the sensor motes are used for all inter-robot communication. However, the powerful "subscribing" ability of Player remains an available option to future projects of the CRR Team.

3.2 The Stage Simulator

Stage is a two-dimensional simulator that provides virtual Player robots that can move, sense, and interact in any bitmapped environment. It is compatible with the Player software and offers configurable, simulated versions of many of the devices currently available to Player (but not all). It is stated by the developers that "agents developed in simulation will work with little or no modification on the real devices and visa-versa" [5]. The design of Stage aims at simulations that are efficient and configurable rather than highly accurate [6].

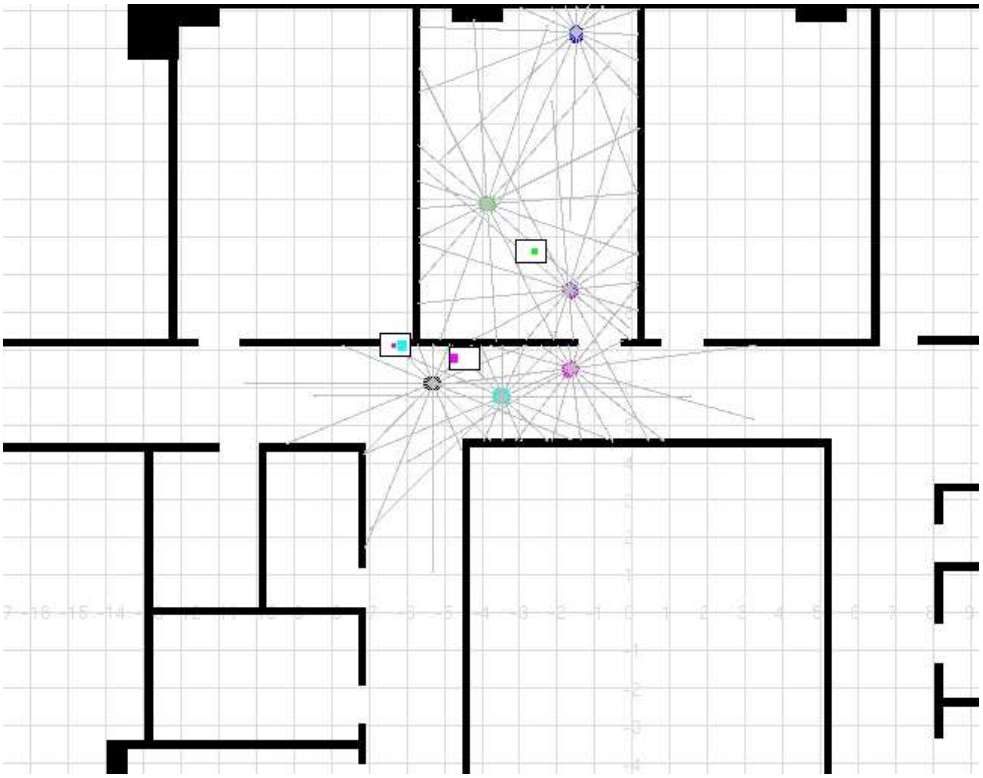


Figure 3: The multi-robot Stage simulator

A screenshot of the Stage simulator is shown in Figure 3. In this simulation, six robots (circular objects) perform a cooperative search in the CRR lab at Broun Hall, Auburn University. The underlying rectangular grid pattern seen in the figure is added by Stage, each grid cell representing one square meter ($1\,m^2$). Each of the six robots shown in the figure is equipped with an array of sixteen sonar sensors and a blob-finding camera[1]. The simulated sonar beams are shown as thin lines radiating from the robots.

[1] A blob-finding camera is a video camera with software for detecting shapes of various colors. An example is the CMUcam [13].

The blob-finding camera is visible in the simulator as a box showing the camera's field of view that appears on the screen whenever a colored blob is detected.

The experiments presented in Chapter Five are performed in two different simulated bitmapped environments. The first is shown in Figure 4 and is titled "Simple Rooms". This is one of the many "worlds" that comes with the Stage software. The second is shown in Figure 5 and is titled "Broun Hall". The drawing on the left is the floor plan of the building that the CRR Lab is in. The bitmap on the right shows the replicated Stage world—the walls have been colored in and the doors removed to make the sonar-sensing less error-prone. The simulated robots used in SARA-1 are equipped with sixteen sonar sensors, a blob-finding camera, and a wireless communication device (not visible in any of the figures). The parameters of each of these devices are given in Table 1. For the configuration files that set up this environment and these device parameters, see Appendix C.

There is also a three-dimensional simulator that is compatible with the Player software called Gazebo. This simulation tool focuses on modeling of device physics and is meant for small numbers of robots interacting with objects in the environment. This is distinct from Stage which is designed for larger populations of robots interacting with each other. While Stage is well-suited for the search-and-rescue task described in this thesis, a rescue that involves manipulation of the rescued object could be implemented using Gazebo.

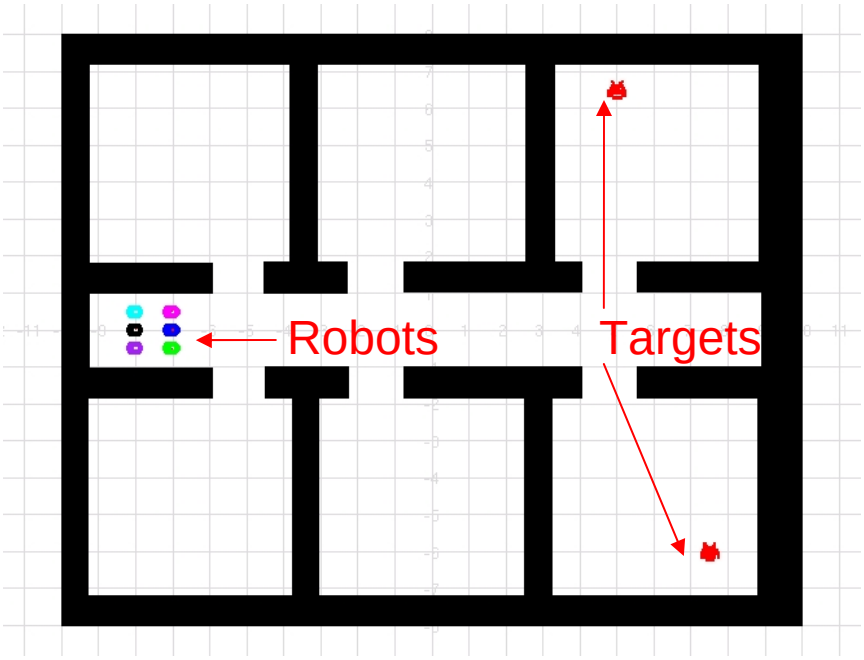
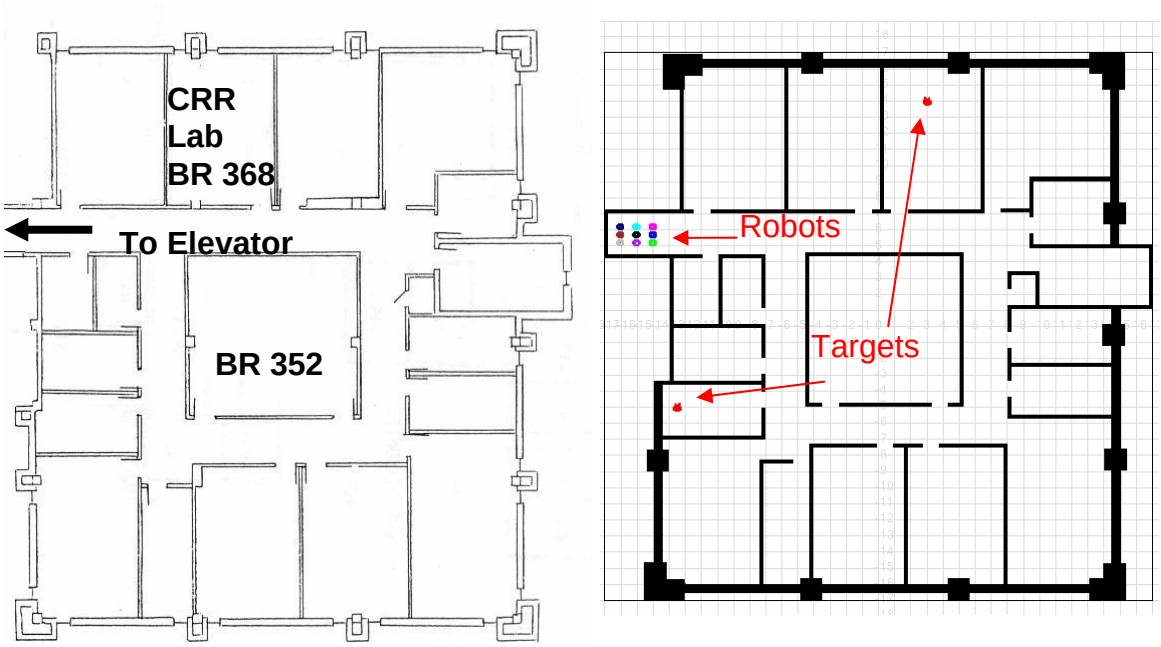


Figure 4: "Simple Rooms" Stage world



a) floor plan **b) Stage world**

Figure 5: "Broun Hall" Stage world

Sonar Sensors:	number of sensors	16
	minimum sensing distance	0 *m*
	maximum sensing distance	5 *m*
	field of view	15 degrees
Blob-finding Camera:	minimum sensing distance	0 *m*
	maximum sensing distance	4 *m*
	image dimensions	80 x 60 pixels
Communication Device:	implementation	Push/pull stack with multiple channels
	type	LIFO

Table 1: Simulated devices used on each robot in SARA-1

CHAPTER FOUR: THE SEARCH-AND-RESCUE ALGORITHM

SARA-1 enables multiple mobile robots to cooperatively search a floor plan for a single hidden target, recognizable by its red color. Throughout the task, the robots use explicit wireless-communication to exchange data about the environment, the target, and each other. Each robot runs the same code on its own processor creating a fault-tolerant, decentralized system. While SARA-1 is designed for multiple communicating robots, it still works in the absence or breakdown of communication or in the event that there is only one robot performing the task.

This chapter is organized as follows. The scenario and assumptions are defined first, followed by a flowchart of SARA-1. Next, the program inputs that are passed to SARA-1 are presented. The remainder of the chapter is broken down into several sections that provide a detailed description of each of the behaviors used in SARA-1. Throughout the chapter, the robustness, scalability, and adaptability of the code are discussed. Alternative methods and ways of improvement are also offered.

4.1 The Scenario

We now discuss the scenario used in our simulated experiments. While SARA-1 was written with indoor experiments in mind, it is not limited to this environment—it is highly applicable to outdoor environments as well. In the scenario, all robots begin in close proximity and search the limits of their environment (defined in our experiments as outer walls) for a single target that is hidden anywhere in the environment. While in the *search state*, robots build and share local maps and incorporate them into a global map of the entire known environment. Robots use these maps to find unexplored areas while attempting to spread out from each other in a more efficient search.

When one robot finds the target, it communicates the target's location to all other robots. This terminates the *search state* and initializes the *rescue state* in all robots. While in the *rescue state*, each robot plans a path to the target's presumed location and attempts to navigate that path. If any robot is unsuccessful in this navigation, it will continue to build maps and update the planned path to the target's location. Map-building in the *rescue state* is non-cooperative; each robot merely tries to get to the target as quickly as possible. The search-and-rescue task is defined as complete when all robots are within a two meter radius of the target. In the event that all areas of the map have been thoroughly searched and no target has been found, the robots will navigate back to their original starting positions.

4.2 Assumptions

There are several assumptions built into SARA-1. Some of them may be changed by manipulating the code and some of them are unavoidable limitations of the SARA-1 algorithm itself. The reasons for many of these assumptions will become apparent in the descriptions of the robotic behaviors. These assumptions include

- Each robot begins with the knowledge of its location in a user-defined Cartesian coordinate system and is not manually moved during the entire simulation.
- All motion and sensing takes place in a two-dimensional plane.
- All motion of a robot is intentional, i.e. robots do not slide—even when they collide.
- Communication is not guaranteed to be available and communication can fail, without the robot being informed of it.
- The current version of SARA-1 supports no more than 20 robots.
- All obstacles, including other robots, are treated as boundaries (i.e. walls). Openings to other areas of the environment (i.e. doors) are assumed to be at least one meter wide. A robot is not guaranteed to find or go through an opening that is less than one meter wide.

4.3 The Algorithm

SARA-1 is written in C and C++. The actual code, in its entirety, can be found in Appendix A. SARA-1 takes advantage of behavior-based robotics, introduced in Chapter Two, and makes several different uses of the same behaviors. It consists of a single, continuous loop with a flag that tells the robot whether it is currently in a *search state* or in a *rescue state*. The flowchart for SARA-1 is given in Figure 6. In the flowchart, behaviors are shown in light gray, the two robot shutdowns are shown in dark gray, and the three instances of the communication behavior are shown in boldface. In an effort to keep the chart simple, several small features of the code are not shown.

4.4 Program Inputs

There are several optional command line arguments available when calling SARA-1. These arguments allow the user to pass different variables to SARA-1 as program inputs without having to alter the code and re-compile it. Table 2 shows each of the optional inputs along with a short description and the default value if an optional input is not supplied. An example of a script used to call SARA-1 can be found in Appendix C.

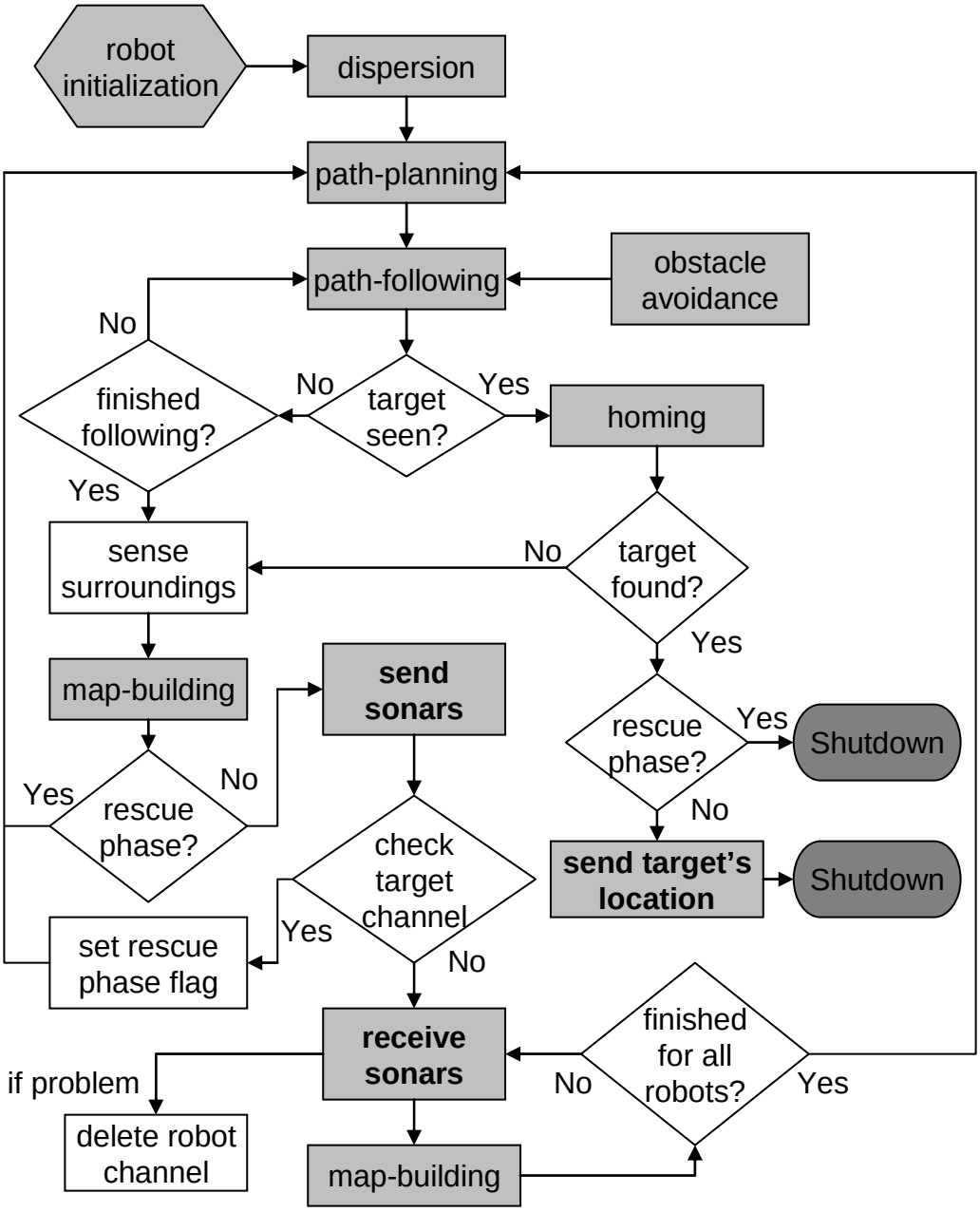


Figure 6: Flowchart for SARA-1

Argument	Definition	Default value
-h <host>	host to connect to Player	'localhost'
-p <port>	port to connect to Player	6665
-N <robots>	number of robots searching	1
-M <size>	size of area to search (meters)	$25\ m^2$
-x <dist>	starting x-position within search area (meters)	$0\ m$
-y <dist>	starting y-position within search area (meters)	$0\ m$
-r <res>	resolution of map (meters)	$0.3\ m$ $(30\ cm)$
-s <speed>	speed robot should move (meters/second)	$1\ m/s$
-d <dist>	minimum tolerated distance for obstacle-avoidance (meters)	$0.5\ m$

Table 2: Optional program inputs available to SARA-1

The -N argument supplies the code with the supposed number of robots that are participating in the search-and-rescue task. It was originally intended for the robots to communicate back and forth on a single channel to determine how many robots were assisting. However, the Player communication software driver is extremely error-prone when several robots are trying to push data onto and read data from the same channel at the same time. The -N argument was added to reduce these errors. While the robots do still perform initialization communications to determine how many robots are available to assist in the task, this argument adds order to the back-and-forth communication. See section 4.5.1 on *robot initialization* for more explanation.

The -M argument tells SARA-1 how large of a search area the robots should explore. The robot uses this parameter to allocate memory for the global map. The current version only allows square maps of less than $75 m^2$. It is intended for a human operator to guess the size of the search area and enter a number that is slightly larger to

ensure the entire search area can fit into the global map. The actions and decisions of a robot are not negatively affected by a map that is slightly larger than their outer boundaries (i.e. exterior walls). In the event that the map is smaller than their outer boundaries, the robots will not attempt to explore any areas outside of their map; thus, this parameter can also serve as an imaginary outer boundary for the robots. The target may still be found in areas just outside of the map and a robot will home in on it; however, other robots will not to be able to path-plan to these unmapped areas while in the *rescue state*, though they will attempt to.

The -x and -y arguments tell SARA-1 where each robot begins in its mapped environment. This information is relative to the upper-left corner of the robot's map. As an example of the three map-related parameters, consider the following scenario: the human operator in Figure 7 wishes to send a robot in the front door and have it build a $50\,m^2$ map from his perspective. The correct program inputs would be -M 50 -x 25 -y -50. This is because, in a $50\,m^2$ map built from the operator's point of view, the front door is at [$25\,m$,-$50\,m$]. If no inputs are supplied, the robot will begin map-building assuming it is in the top left corner of its environment.

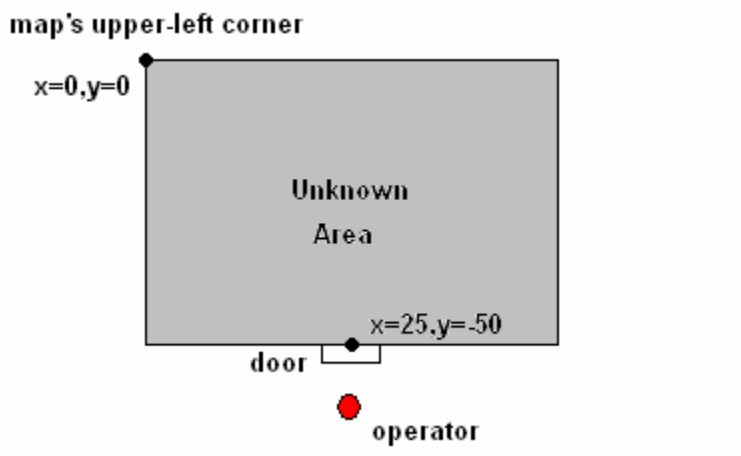


Figure 7: Example of the map-related program inputs, {M,x,y}

These -x and -y arguments are very important because they ultimately tell the robots their positions relative to each other—an important piece of information when sharing maps. While it is easy to get accurate values for these arguments in simulation, it will be difficult to determine their values in real life. A possible solution is to start the robots very close together; however, this can create dispersion problems in which robots lock together while turning. Alternatively, Matarić [31] uses triangulation beacons for location awareness. The ideal solution would be a simultaneous mapping and localization performed by each robot. This, however, is an open problem in current research.

The -r argument sets the resolution of the map built by the robot. In the current version of SARA-1, this parameter cannot go below $30\,cm$ on a $75\,m^2$ map, or $10\,cm$ on a $25\,m^2$ map. The -s argument sets the speed with which the robot is to travel. The control of the *path-following* behavior is tuned for the default speed of $1\,m/s$. Speeds

higher than this may cause runaway robots. The -d argument sets the minimum tolerated distance of the sonar sensors. It is used in the obstacle-avoidance behavior and must not exceed half the distance of any opening a robot is expected to go through; thus, assuming one meter openings, the default -d argument is $0.5\,m$.

4.5 Behaviors

There are eight main behaviors that make up the SARA-1 algorithm. Two of the behaviors, *robot initialization* and *dispersion*, are only used during the first few seconds of the search-and-rescue task. The other six behaviors occur in a continuous loop in both the *search* and *rescue states* (except for the *communication* behavior, which only occurs in the *search state*). Matarić [31] categorizes behaviors according to their associated goal: behaviors that persist in time without a terminal state are referred to as maintenance behaviors, whereas behaviors that have a defined start and finish are referred to as attainment behaviors. Of the six main behaviors used recursively in SARA-1, three are maintenance behaviors (*map-building, communication, and obstacle-avoidance*) and three are attainment behaviors (*path-planning, path-following, and target-homing*). The remainder of this chapter gives a detailed description of each of the behaviors used in SARA-1.

4.5.1 Robot Initialization

The *robot initialization* behavior is called at the beginning of the search-and-rescue task and is responsible for initializing communication, setting up channels

between robots, and passing initial data. First, a robot sends its own initialization data on its own channel to let other robots know that it is functioning and ready (for the specific communication protocol, see section 4.5.8). After a robot has sent its own data, it then checks other channels for more functioning robots. The robot uses the parameter passed to it by the -N program input to know how many channels to check and waits a certain amount of time on each channel before abandoning it. As each channel is read, the robot processes the received initialization data and takes note of the channel used to communicate with that robot.

At the end of this behavior, there should be one channel for each robot that is known by all of the functional robots that are participating in the task. While in the *search state*, each robot will push information onto its own channel and all other robots will be able to read that data. Having processed all of the received initialization data, each robot is now informed of all other robots' relative positions. This information is used in the *map-building* behavior.

4.5.2 Dispersion

Dispersion is a behavior that forces the robots to spread out. This behavior is a small but important part of many multi-robot tasks and often proves to be a limiting factor in the increase of the number of robots performing. As an example, consider a search task that normally takes a single robot two minutes to complete—if five robots take a minute and a half just to get away from each other and start the search, valuable time is lost that may undo the benefits of cooperation. *Dispersion* is a particularly difficult task when robots start in a small room, such as an elevator.

The most important part of the *dispersion* behavior in SARA-1 is the creation of the first map. If this first map is "clean" (i.e. free of robots and other non-stationary obstacles), the *path-planning* behavior will naturally disperse the robots. However, if a robot starts the task surrounded by other robots, the first map will not be clean. In a worst case, the map may appear to be "fully explored" before the robots even start searching (see the frontier cells definition of the *path-planning* behavior).

In order to get a clean first map that is free of other robots (robots are represented in the map as obstacles), the robot on port 6665[2] moves ahead of the group and builds the first map. This is somewhat of a centralized approach since one robot is given an initialization responsibility on which all other robots depend. However, if the robot on port 6665 malfunctions during this initialization responsibility, it does not cause SARA-1 to fail. The other robots will begin the search-and-rescue task after some time of waiting regardless of whether or not the robot on port 6665 successfully completes this responsibility. However, if this robot does fail, dispersion will probably take longer due to the fact that the first map will not be "clean". There is also a greater chance that the robots will falsely assess a "fully explored" area.

One other dispersion technique is used to disperse all other robots while they are waiting on robot 6665 to build the first map. Each robot uses its simulated blob-finding

---

[2] Each robot communicates with the Player software using a different port number, passed to SARA-1 by the –p program input. By default the port numbers are 6665, 6666, 6667, etc. The robot on the first port, port number 6665, is given the responsibility of moving ahead of other robots and building the first map.

camera to determine if another robot is directly in front of it. If so, the robot waits; if not, the robot moves forward while avoiding obstacles. This simple technique prevents the robots from being so close together that they lock into each other during initial turns. It works well when robots begin in a forward-facing "marching formation" as in the experiments described in Chapter Five.

4.5.3 Obstacle-Avoidance

The first implementation of the *obstacle-avoidance* behavior was long and complex and consisted of both proportional and integral control. However, the computational burden of the frequent calling of this maintenance behavior was determined to be too costly. A second writing of the behavior implements a tradeoff between accuracy and complexity. The inaccuracy of this new and simple behavior is overcome by direct combination of multiple behaviors, discussed in Chapter Two, section 2.4. Instead of having one complex and accurate behavior, two simple behaviors are called simultaneously and their results are mathematically combined. The inaccuracy of each behavior is partially compensated for by the other. See section 4.5.6 on *path-following* and section 4.5.7 on *target-homing* for examples of direct combinations with the *obstacle-avoidance* behavior.

Each time the behavior is called, it checks the front hemisphere of sonar sensors (labeled 0-8 in Figure 8) to see if any of their ranges are less than the minimum tolerated sonar distance (this parameter was passed to SARA-1 by the -d program input or was defaulted to $0.5\,m$). If none of these ranges are less than the minimum, the behavior returns the commands "full speed ahead". Otherwise, it checks to see which of the sum

of left-front ranges (sensors 0-3) or right-front ranges (sensors 5-8) is smaller, and sends commands to turn the robot in the opposite direction.

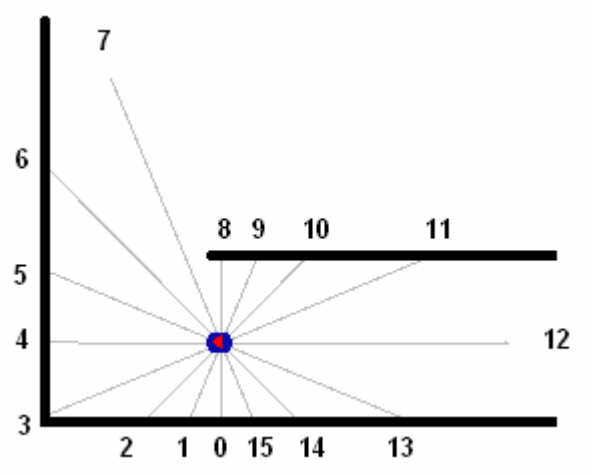


Figure 8: Robot using sonar sensors, traveling from right to left

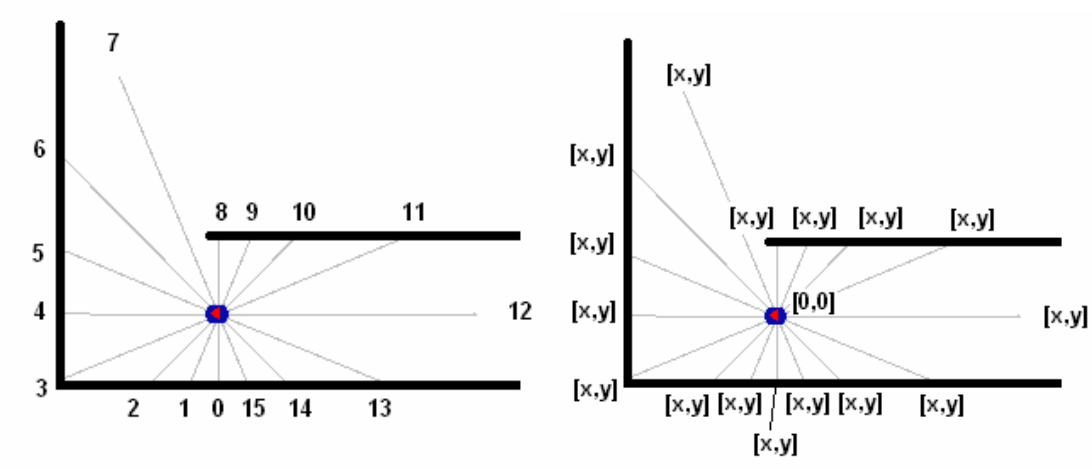
This is illustrated in Figure 8. Currently, the *obstacle-avoidance* behavior is keeping the robot traveling from right to left in the center of the hallway (or at least on a sinusoidal path down the center of the hallway if it is wide). As the robot approaches the corner, the sonar ranges will grow smaller on the robot's left-front side (sensors 0-3). Some of the sonar ranges on the robot's right-front side (sensors 5-8) will grow smaller as well, but the code compares the sums of these two sets of sensors. When any sonar range is less than the minimum tolerated sonar distance, the robot will turn right and proceed through the doorway. If there was no doorway, and the sum of the left sonar ranges matched the sum of the right sonar ranges, the robot would turn left by default. Occasionally, this will create a dead-lock situation in which the robot gets stuck in a corner turning back and forth. SARA-1 checks for this and tells the robot to back up (see section 4.5.6 on *path-following*).

4.5.4 Map-Building

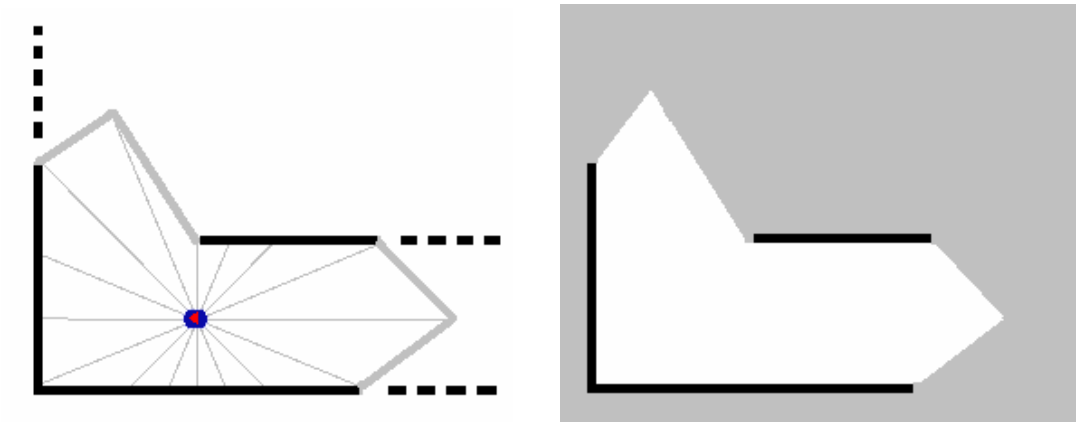
The *map-building* behavior is the backbone of SARA-1. The simulated robots use sonar readings to build local maps, communicate these sonar readings to share local maps, and incorporate all of these local maps into one large global map (the global map exists separately in each robot's memory—but all maps will be identical). Using this global map, the robots are able to cooperate in their search by spreading out and not visiting the same place twice. Also, when one robot finds the target within this map, other robots can find their way to the target's location.

SARA-1 uses robots with an array of sixteen sonar sensors, all sensing in a two dimensional plane. An alternative to using this many sonar sensors is to use one rotating sensor [22]. In real-world situations, this set-up may suffer greatly from specular reflections, a common problem in which sonar signals reflect off an object and away from the sensor instead of back to it. One simple fix to this problem is to ignore any readings that are above the maximum sensing distance of the sonar sensor. This is *not* implemented in the current version of SARA-1. Alternatively, [48] uses several arrays of forward and downward pointing sonar sensors to detect any vertical angle of incidence. This method is able to sense in a three dimensional plane but still suffers from horizontal specular reflections. In a different approach, [7] relies on both laser range-finders and sonar sensors in what is termed a "laser-limited sonar" approach to reduce the effects of specular reflections. This, however, greatly increases the cost of the hardware.

Figure 9 shows how a robot builds a local map using its sonar sensors. In Figure 9a, the robot senses the environment by reading each of the sixteen sonar range values. Trigonometric calculations are used to get the [x,y] coordinates relative to the robot's position as shown in Figure 9b. A "get points between" function is called to complete the perimeter between each of the sixteen sonar-detected points, drawing a polygon around the robot. If two points from adjacent sonar sensors are at a distance of less than one meter apart (it is stated section 4.2 that all openings and doors are assumed to be at least one meter wide), a line is drawn between those two points representing the edge of an obstacle. If the two points are more than one meter apart, or if any sonar sensors yield a maximum reading, a line is drawn between these points representing the edge of the robot's unknown environment. This is illustrated in Figure 9c, with black indicating an obstacle and grey lines indicating unknown space. With this polygon drawn around the robot, the area inside the polygon is labeled obstacle-free space, whereas the area outside the polygon remains unknown space. The robot's final local map is shown in Figure 9d.



a) sense environment **b) get sonar points**



c) define outer environment **d) fill in local map**

Figure 9: *Map-building* progression

Due to rounding errors and real-world effects, the local maps built by robots will not be perfect and will not line up perfectly when combined with other robots' local maps. To handle this, SARA-1 uses a probabilistic approach to represent obstacles in an "occupancy grid". Instead of labeling a map with black, grey, and white, each cell of an

occupancy grid is labeled with high probability, 0.5 probability, and low probability. Any cell that contains a value below 0.5 is referred to as 'open' and any cell that contains a value above 0.5 is referred to as 'occupied'. A cell with a probability of exactly 0.5 is referred to as 'unknown'. The use of probability and occupancy grids is a common occurrence in the robotics literature [7, 20, 41, 49]. It has several benefits including easy combination of multiple grids and a self-correcting nature.

The current version of SARA-1 uses a value of 0.4 for an open cell and 0.9 for an occupied cell when labeling local occupancy grids. The reason for this nonlinear relationship is that there is typically a greater chance of falsely assessing open space than occupied space, both in simulation and in real-world operation. For example, it is a common occurrence for the edges of walls or corners to fall between a robot's sensors and be falsely labeled as open space. As discussed in section 4.4, the default resolution of the occupancy grid is $30\,cm$. This is roughly the size of a robot that might be used in the search-and-rescue task.

Local occupancy grids are combined using equations 4-1 to 4-3 [49].

$$\alpha_{x,y}^{i} = \frac{P(occ_{x,y}^{i})}{1 - P(occ_{x,y}^{i})} \tag{4-1}$$

$$\alpha_{x,y} = \prod_{i=1}^{N} \alpha_{x,y}^{i} \tag{4-2}$$

$$P(occ_{x,y}) = \frac{\alpha_{x,y}}{1+\alpha_{x,y}} \tag{4-3}$$

In these equations N is the total number of robots, [x,y] are the coordinates of a grid cell, i ε [1,N] is the robot number, and $P(occ_{x,y})$ is the probability that the grid cell is occupied by an obstacle. $P(occ_{x,y})$ is a value in the range (0,1). The variables $\alpha_{x,y}$ and $\alpha^i_{x,y}$ are used as temporary variables in an effort to separate the three equations for ease of viewing. Equation 4-1 is illustrated in Figure 10. This equation expands the range (0,1) of $P(occ_{x,y})$ through a hyperbolic mapping onto $(0,\infty)$. This gives higher weight to a probability that is farther away from the center of the range of $P(occ_{x,y})$. When the values of $\alpha^i_{x,y}$ are multiplied in equation 4-2, probabilities near either extreme will have a much larger effect. Therefore, if a robot senses a grid cell as open that has been sensed multiple times by other robots as occupied, the open cell will have very little effect during occupancy grid combination. Equation 4-3 remaps the resulting product into the range (0,1). Notice in Figure 10 that when $P(occ_{x,y}) = 0.5$, $\alpha^i_{x,y} = 1$. This is so that in the multiplication of equation 4-2, cells with unknown probabilities will have no effect.

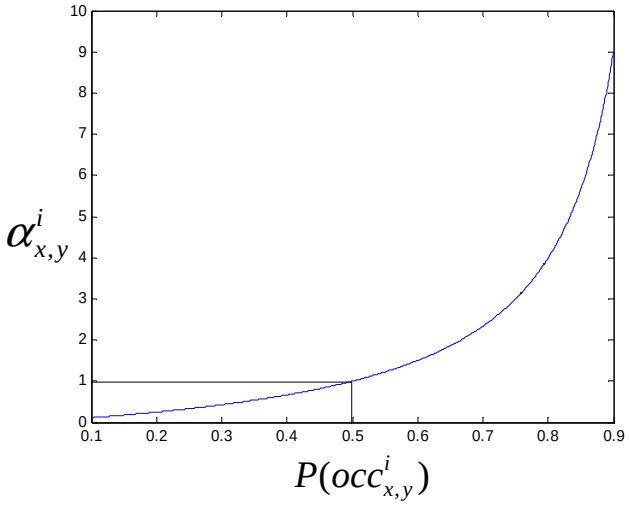
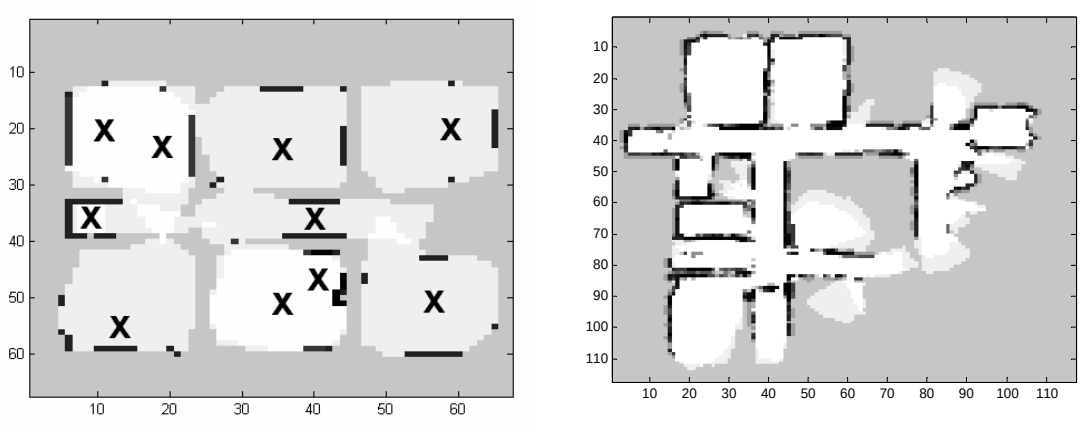


Figure 10: Plot of Pocc vs. α



a) Simple Rooms **b) Broun Hall**

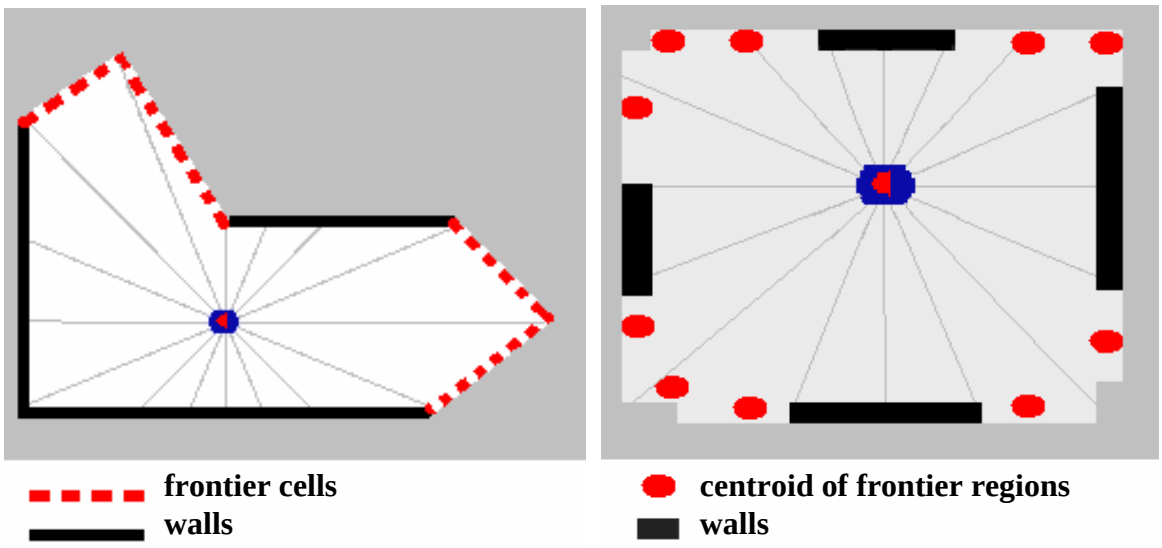
Figure 11: Examples of global occupancy grids. Robots' locations marked with X's

Two examples of global occupancy grids are given in Figure 11. In Figure 11a, ten robots were manually placed throughout the "simple rooms" world and their local maps were combined. The positions of the robots are shown by x's. Notice that where local grids overlap, such as in rooms with two robots, the probability of occupancy is less (i.e. the rooms appear whiter). Figure 11b shows a partially complete occupancy grid of

Broun Hall that three robots cooperated to build. Several methods of enhancing these occupancy grids are described in the literature [26]. However, since the map is only a means of cooperation in SARA-1 and need not be perfect, none of these methods are used here.

4.5.5 Path-Planning

Now that the robots have a shared global map to work with, they need to explore the unexplored areas of the map, pushing the outer limits of their known environment. This is accomplished through *path-planning* to one of the various "frontier cells." A frontier cell is defined as any open cell that is adjacent to an unknown cell [7]. Returning to the robot navigating down the hallway, its environment is reprinted in Figure 12a with the frontier cells shown as a dotted line.



a) frontier cells **b) frontier regions**

Figure 12: Frontier cells and frontier regions

Depending on the size and resolution of the occupancy grid, the number of frontier cells may be very large. There is no need to explore all of them; only areas that are large enough to contain new openings to go through ($1\,m$) need to be explored. For this reason, a "frontier region" is defined [7] as a group of frontier cells large enough to be of interest to a robot. Figure 12b shows the centroids of frontier regions found by a single robot in one of the rooms of Simple Rooms[3]. Limiting a robot to only explore frontier regions also prevents a robot from attempting to reach a stray open cell that is accidentally placed inside of a wall.

The remainder of the path-planning behavior is an adapted method from [49]. Once a robot has a list of frontier regions to explore, it needs to make an educated decision about which one to pursue based on how much effort it will take to get there and how far away from other robots the region is. These two ideas are mathematically modeled by two parameters called cost and utility [49]. The cost is defined as the overall cost of reaching a cell and is based on both distance to the cell and the number of obstacles that lie between the robot and the cell. The utility is defined as the amount of unexplored area that a robot can cover with its sensors upon reaching a frontier region.

[3] The current version of SARA-1 searches for frontier regions only by rows and columns. If there is a perfectly diagonal frontier, the code will not pick up on it. This problem would theoretically cause a robot not to explore corners; however, during the almost ninety experiments performed in chapter five, this problem was never encountered. It may be a bigger problem for higher-resolution occupancy grids.

A cost grid, calculated for each robot, holds the cost of reaching each cell in the global occupancy grid. It is calculated using equations 4-4 and 4-5.

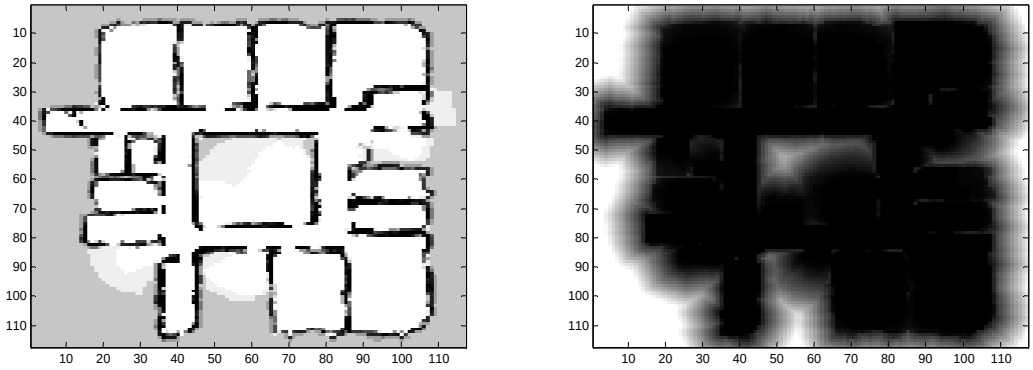
$$\text{Initialization:} \quad C_{x,y} = \begin{Bmatrix} 0, & \text{if } [x, y] \text{is the robot's position} \\ 9999, & \text{otherwise} \end{Bmatrix} \tag{4-4}$$

$$\text{Update Loop:} \quad C_{x,y} = \min_{\substack{\Delta x=(-1,0,1) \\ \Delta y=(-1,0,1)}} \left\{ C_{x+\Delta x, y+\Delta y} + \sqrt{\Delta x^2 + \Delta y^2} \cdot P(occ_{x+\Delta x, y+\Delta y}) \right\} \tag{4-5}$$

In equation 4-4, the value of ∞ is represented by 9999. After initialization, the update loop updates each cell by the value of its best neighbor plus the cost of moving to that neighbor. Cost is equivalent to the distance to a cell times the probability that it is occupied. The update loop is iterated until the entire grid converges. This resulting grid of values represents the cumulative cost for reaching each cell in the global occupancy grid.

Figure 13b shows a representation of a robot's cost grid. The actual values have been mathematically manipulated to highlight the cost due to the probability of occupancy and suppress the cost due to distance. Figure 13a displays the robot's current global occupancy grid for reference. In the cost grid, light regions represent areas of high cost and occur along the outlines of walls as well as in large areas of unknown space. This figure also shows a pictorial representation of one of the downfalls of the cost grid approach. Large areas of unknown space often cause the *path-planning* behavior to send a robot through a wall in an attempt to stay away from these large areas of high cost.

This is less of a problem in Simple Rooms, where the walls are very thick and provide much higher cost for going through them.



a) Broun Hall global map **b) Broun Hall cost grid**

Figure 13: Cost grid of a robot in Broun Hall

Using this cost grid, a robot can path-plan to any location on the map using steepest descent in $C_{x,y}$ starting from the goal location. The cost grid can be pictured as a landscape with mountains of high cost and valleys of low cost. If a ball is placed at any point, it will roll down the steepest path, through the valleys, and to the lowest point on the map. The lowest point was declared to be the robot's position in equation 4-4. An example of a planned path is shown in Figure 14. The robot has received the target's location via wireless communication and is attempting to get from its current position in the upper-left room to the target's supposed position in the lower-right room. The robot plans a path from its position to the target's position

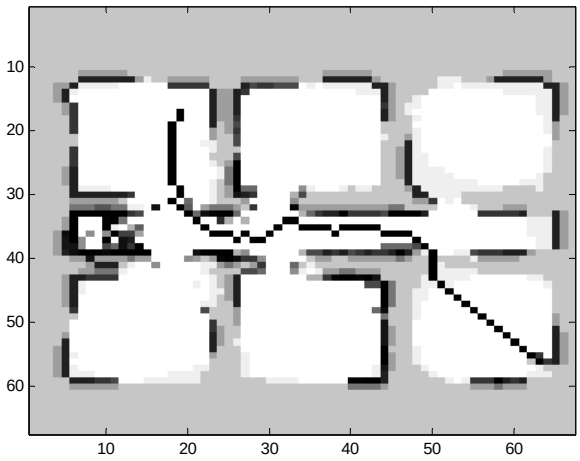


Figure 14: *Path-planning* through simple rooms

The *path-planning* behavior is not always as accurate as the path in Figure 14. It was observed during the experiments described in Chapter Five that the steepest descending path will occasionally get stuck in a spot on the map and will never make it to the robot. This problem usually occurs when robots have been mapping a single room for a long time, or when several robots are mapping the same room. As more maps build up, the valley begins to loose its slope and starts to become a plain. On a non-sloping plain, the steepest descent typically moves along the edge of obstacles instead of out in the middle of the open area (i.e. the bottom of the valley). When this happens, any stray cell in the cost grid can cause the *path-planning* behavior to get hung up. This is illustrated in Figure 15. Simplified costs of 0 and 9 are used to represent low and high cost in the cost grid. If the path is descending down the zeros on the left side of the grid, it will get stuck on the two boldfaced zero cells in row three. The *path-planning* behavior will simply go back and forth between these two cells.

9	0	0	0	0
9	0	0	0	0
9	**0**	**0**	9	0
9	9	9	9	9

Figure 15: Problem with the *path-planning* behavior

SARA-1 checks for this dilemma and halts a path that is having this problem. Since the descending path never made it to the robot, the robot will not be able to follow the path; but it will still try. After several seconds of not being able to get to the first node on the path, the robot will give up. The time is determined by the acquiescence parameter in the *path-following* behavior. This path-planning problem does not prevent SARA-1 from operating, it just reduces the efficiency. As a robot moves toward the problem spot, a different path or different frontier cell may be assigned. If not, the robot will eventually reach the problem spot. At this point, the self-correcting nature of the *map-building* behavior will typically fix the stray cell if another robot has not done so already.

So far, this section has addressed the idea of frontier regions and the methods and problems of reaching these regions. No method, however, has been introduced which would cause the robots to spread out their search in a cooperative manner (many algorithms described in the literature actually end here and do not provide for intentional cooperation). This is the reason for applying the utility function. As previously defined, utility is the amount of unexplored area that a robot can cover with its sensors upon

reaching a frontier region. With this information, robots can increase the cost of going to those frontier regions that are close to a region already assigned to another robot.

To compute the utility, a mathematical expression must be defined to represent the area that a robot can expect to cover upon reaching a frontier region. The expected visibility range, $V(d)$, is defined as the probability that a robot's sensors can cover objects at a distance d, and is given in equation 4-6. In this equation, $d_1, d_2, \ldots, d_j, \ldots$ are distances measured by robot's sonar sensors and $h(d_j)$ is the number of times that d_j was measured by any robot. This information is stored each time sonar readings are sent or received by the *communication* behavior; thus, it is a collective representation of all robots' sonar readings. The Visibility lies between zero and one and grows smaller with increasing distance. A nice characteristic of this function is that it is adaptive to the search environment. The Visibility histogram of a group of robots that is searching in a relatively confined environment will look very different compared to a group that is searching a wide-open area [49].

$$V(d) = \frac{\sum_{d_j \geq d} h(d_j)}{\sum_{d_j} h(d_j)} \qquad (4\text{-}6)$$

Get a list of all frontier regions

Calculate the cost, $C^{i}_{x,y}$, of reaching all frontier regions for all robots

Initialize Utility, $U_{x,y}$, of all frontier regions to 1

While any robot is not assigned to a frontier region:

Find a robot i and frontier (x, y) combination, where:

$$[i,(x,y)] = \underset{(i'(x',y'))}{\arg\max} \{U_{x',y'} - C^{i'}_{x',y'}\}$$

Assign robot i to the frontier (x, y) and reduce the Utility of all frontiers, (x', y'), in the Visibility range of (x, y)

$$U_{x',y'} \leftarrow U_{x',y'} \bullet [1 - V(\| (x,y) - (x',y') \|)]$$

Path-plan to the assigned frontier region

Table 3: Pseudo-code for the *path-planning* behavior

The pseudo code of the *path-planning* behavior is given in Table 3. Each robot runs its own instance of this code. As seen in Table 3, the cost grid for every robot is calculated by the processor of each robot in an effort to eliminate the need for a centralized planner. So, with a group of ten robots, each robot computes ten cost grids. The computation of the cost grid is by far the greatest processing burden on the robot. In simulation on an Intel® Pentium® 4, 2.3 GHz processor running Linux Fedora™ Core 4, the cost grid takes approximately 0.2 seconds to converge with a $20\,m^2$ map and a grid cell resolution of $30\,cm$. This means that for a group of ten robots, each *path-planning* behavior takes two seconds to perform. As shown in Chapter Five, while this behavior is

computationally costly, it still represents only a small percentage of the overall time spent during the task. The memory burden is not as intensive since the cost grid is reused for each robot after storing only the frontier region costs.

4.5.6 Path-Following

As seen in Figure 14, the *path-planning* behavior creates a path that is often jagged and difficult for a control program to follow while maintaining stability. A "path relaxation" algorithm is described in [8] which allows nodes along the path to have restricted movement and is able to turn jagged paths into smooth ones. The smooth paths are shown to be much easier for a robot to navigate. Using a slightly different approach, SARA-1 allows the robot to stray from the path up to two grid cells during *path-following*. Obstacle avoidance is relied on to keep the robot away from obstacles that might be just off the path.

The *path-following* behavior follows a path on a node-by-node basis, making calls to both *homing* and *obstacle-avoidance* behaviors (this is not the same as the *target-homing* behavior, which makes use of the camera to home in on the target). The *homing* behavior takes the next node of the path, transforms it into the robot's coordinate system using equation 4-7, and returns commands to steer the robot toward the node.

$$T(x_G, y_G) = \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix} \bullet \begin{pmatrix} x_G - x_i \\ y_G - y_i \end{pmatrix} \tag{4-7}$$

where, $[x_i, y_i, \theta_i] =$ the robot's pose

and $[x_G, y_G] =$ the goal location

The output of this homing behavior is directly combined [31] with the output of the *obstacle-avoidance* behavior, described in section 4.5.3.

A second purpose of the *path-following* behavior is to keep up with how much time it is taking the robot to navigate a path. A parameter called *acquiescence* [27] is defined that grows while a robot is not succeeding in following its path. This parameter causes the robot to give up after a certain amount of time so that other robots will not become impatient and give up on that robot (the *impatience* parameter is discussed in section 4.5.8). Currently, both an overall-acquiescence parameter and a per-node-acquiescence parameter is utilized. The second is added to keep a robot from repeatedly attempting to navigate through a wall, which will eventually cause the robot to get stuck.

This leads to the third and final purpose of the *path-following* behavior which is to continuously monitor the robot to see if it is stuck. If a robot is determined to be stuck, the behavior senses for obstacles behind and ahead that the robot might be stuck on and sends a burst of speed in the opposite direction. Occasionally, depending on the way a robot is stuck and the proximity of other nearby obstacles, a robot will determine a false direction of speed burst. SARA-1 corrects this by going into a mode in which the robot will speed burst forward and backward until it becomes unstuck. Simultaneously, the robot continues to run the same complete loop of the code, map-building and communicating, so that other robots will not become impatient and give up on it. Even with this approach a robot will occasionally, though rarely, remain stuck. In this case it

spends the rest of the time building and sending local maps while trying to get unstuck. Its cooperative effect is still useful in that no other robot will come near that robot's visibility area due to the utility parameter.

4.5.7 Target-Homing

Upon the sighting of the target, distinguishable to the robot's camera by its red color, a robot will go into a *target-homing* behavior (called camera-homing in the code of Appendix A). This behavior implements proportional control to keep the target in the middle of the camera's field-of-view while homing. Calls are also made to the obstacle-avoidance behavior whose commands are directly combined with the *target-homing* control commands. In the event that a robot looses the target, it will perform a complete revolution before giving up and going back to searching.

When the robot is sufficiently close to the target that it covers the camera's field-of-view, the robot will stop, communicate its position to the other robots, and shutdown. If a robot is *target-homing* while in the *rescue state*, communication of the target's location has already taken place; so, the robot will simply shutdown when it is within a two-meter radius of the target. This two-meter radius allows sufficient room for all robots to crowd around the target. An important note is that upon first sighting of the target, a robot builds a quick map and sends it to the other robots. This is so the robots will continue searching during the next loop of the code and not wait on this robot to finish *target-homing*. Any waiting increases the chance that other robots will become impatient and delete that robot's channel. Although, if this happens, the other robots will

still receive the target's location on the target channel, which is described in the following section.

4.5.8 Communication

The *communication* behavior is used for several different purposes and is a part of several different behaviors. There are two interfaces in the Player software designed for inter-robot communication: the fiducial and MCom interface. The fiducial interface models a coded communication beacon that can send directed messages or broadcasts, but is limited to line-of-site. The MCom interface models a communication device with several different channels that are accessible by push, pop, and read commands. After careful consideration, it was decided that the MCom interface best simulates the performance of the hardware intended for use when the code is implemented on real robots.

There are three types of communication that occur during the search-and-rescue task: initialization, sonar readings, and target's location. The protocol for each of these three types of communication is shown in Table 4. The robot channels are established during *robot initialization* and are labeled with a robot's own port number. Using the -N program input, the N channels for N robots are labeled 6665 to (6665+N). (These are the standard port numbers that Player uses, but can easily be changed.) The robot channels are used to communicate during *robot initialization* and for the sending/receiving of sonar ranges.

The target channel, which is separate from the robot channels, is used to send and receive the target's approximate location and the mapped environment around it. This channel is separate from all other communication channels and could be monitored by a human during a real life search.

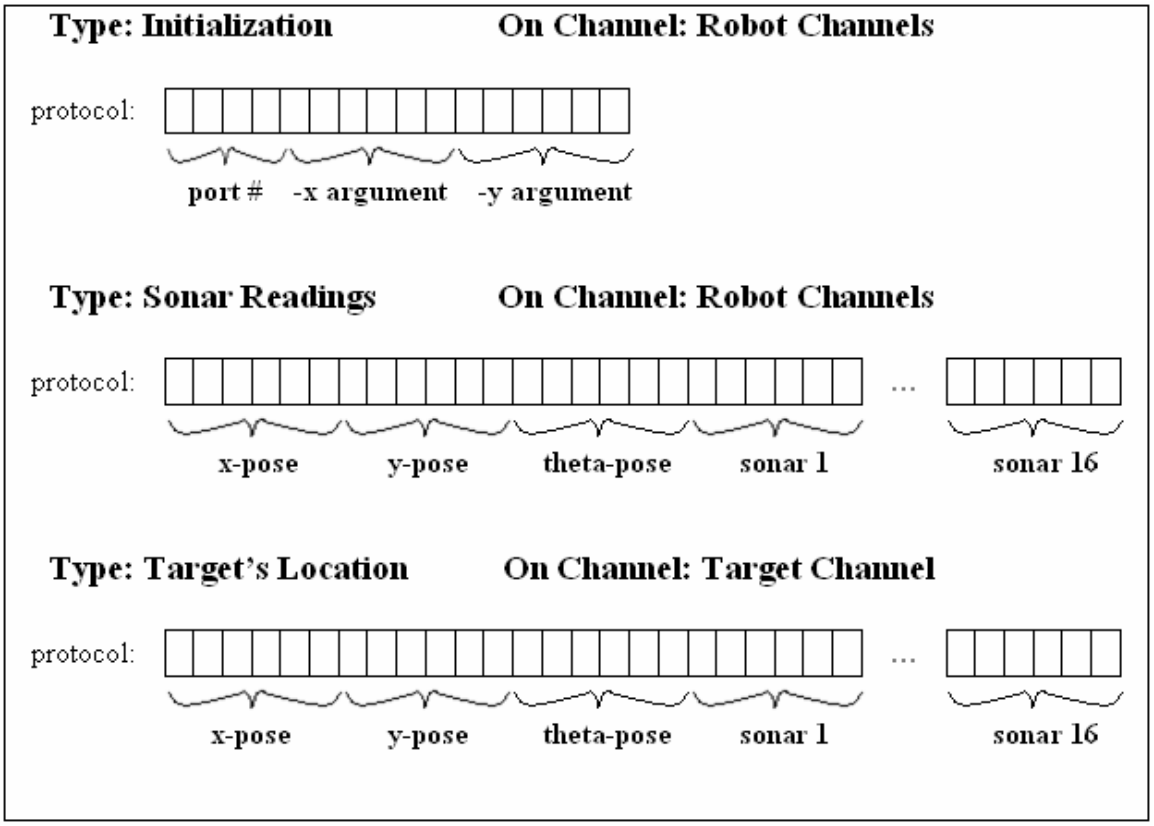


Table 4: The three *communication* protocols

The MCom interface has two limitations: the data sent can only be in ASCII format and it cannot exceed 128 bytes per message. With these two limitations, the protocol for temporary conversion of data is given in Table 5. The x's represent digits, N's are negative flags, and •'s are decimals. This protocol converts numerical data into character strings of length six and is used to create each data packet in Table 4 (except

the port number, which is transformed to a character string using the *sprintf* function). The length of six is the largest that will allow the sonar readings to fit into one message of 128 bytes. As is seen in the figure, the maximum data truncation will occur with a negative, double-digit number. When a message is received by a robot, each data packet is converted back into numerical data for processing.

	single-digit numbers	**double-digit numbers**
positive	x x x x x x	x x x x x x
negative	N x x x x x	N x x x x x

Table 5: Protocol used for converting numbers to message packets

Using the communication protocol described in this section, data is pushed onto a channel by one robot and all other robots may then read that data. The stack holds a maximum of eight messages in each channel's buffer, although only the most recent data is accessed by any robot in SARA-1[4]. Since the stack is a LIFO (last in, first out) implementation, there is no need to clear or manage old data. It is simply pushed out back end of the data buffer when new data arrives.

[4] Along with different channels, the MCom interface defines different 'types', so that a robot can look for information on a specific channel of a specific 'type'. 'Types' are labeled with integer numbers and the current loop number of SARA-1 is used for the 'type' of sonar readings being communicated. This is how robots know if data on a channel is old or new.

As previously mentioned the MCom interface is very error-prone and does not respond well to multiple robots simultaneously pushing and pulling on a single channel. Errors that occur while reading a channel appear to the control program as empty channels. For this reason, during any communication, care is taken to ensure that a robot attempts to read the channel several times before giving up. This is the reason for a lot of the waiting time that is built into the code. However, a robot will not wait forever. Following the *impatience* motivational behavior of [27], an *impatience* parameter is added to the *communication* behavior of SARA-1. This parameter grows as a robot is attempting to read a channel, and allows the robot to give up after a certain amount of time has passed. Upon giving up, the robot deletes that channel from its memory and will not attempt to read from it again[5]. From that robot's perspective, there is one less robot assisting in the search.

There is also a bug in Player that occasionally allows one robot to push while another robot is attempting to pop/read without giving errors. The bug allows half of the data to be pushed, and half of the data to be read. The rest of the data is composed of random, non-character values which often causes Player to shutdown. This bug was temporarily fixed with a patch that can be found in Appendix D.

[5] Before deleting a channel, the robot will check the target channel to ensure that the *rescue state* has not been initiated.

This ends the discussion of SARA-1. This code has been fully tested both as individual behaviors and as a complete algorithm; however, as Matarić [31] points out, when multiple, cooperative robots are sent to perform a task, no mathematical predictions can precisely determine what the code will do…and there are typically not enough robots for statistical representations.

CHAPTER FIVE: EXPERIMENTAL RESULTS

This chapter presents the results of several different experiments using simulations of SARA-1. Communication experiments are performed in the Simple Rooms world presented in Chapter Three. In this set of experiments, three types of communication intervals are compared: *continuous*, *occasional*, and *no communication*. In the *continuous* and *occasional* experiments, the number of robots is varied from one to seven. In the *no communication* experiment, the number of robots is varied from one to four. The time for task completion, number of unsuccessful runs, and distribution of behaviors is discussed and compared for each of the three types of communication.

In an effort to show the versatility of SARA-1 as well as to test it in a more realistic environment, a final set of experiments is performed in the Broun Hall world presented in Chapter Three. Due to the complexity and size of this environment and the fact that the processing of multiple robots is being performed on a single processor that is already running a graphical simulation, each experiment in Broun Hall had to be drastically slowed down to prevent timing-related errors (a thirty minute search-and-rescue task may take up to seven hours). For this reason, only the *continuous communication* experiments are performed in this room using only odd numbers of robots

from one to seven. Before presenting the results of each these experiments, several similar experiments from the cooperative robotics literature are presented along with their results.

5.1 Experiments from the literature

5.1.1 Number of Robots Experiments

Finding an appropriate number of robotic agents for a particular task and environment is a classic research scenario in the cooperative robotics literature [2, 34, 35, 40]. Typically, the positive effect of cooperation is weighed against the negative effect of cooperation byproducts such as inter-robot interference, competition for resources, or communication and computational overhead. There is usually some threshold in which the negative byproducts of cooperation begin to outweigh the positive effects—at this point the task achievement begins to deteriorate with the addition of more robots. Researching that threshold, identifying the reasons for it, and finding out what makes it change remains an open area of research in cooperative robotics.

Halme *et. al.* [2] experiment with the number of robots in a swarm approach to a stone-collecting task. In this set of experiments, it is found that, as the number of robots increases, the time per stone decreases while the number of collisions increases. This illustrates the positive effect of cooperation versus a negative effect: inter-robot interference. Much like the experiments on SARA-1, this search-like technique is measured by the total time to completion and it is found that the overall time decreases

with the addition of more robots. This shows that the positive effect of additional workers outweighs the negative effect of additional interference (at least in this experiment and environment).

Schneider-Fontán and Matarić [34] point out that the main cause of inter-robot interference is the competition for space. They explore easing this competition for space by setting up non-overlapping territories for each robot in a puck-gathering task. This spatial isolation minimizes interference; however, task performance still degrades when the number of robots is increased above a threshold.

In contrast, the results in this thesis show that increased cooperation reduces inter-robot interference; but there is still a threshold in the number of robots that causes the time for task completion to deteriorate.

5.1.2 Communication Experiments

Experiments on the amount and type of communication are also a common occurrence in the cooperative robotics literature [29, 33, 39, 40]. Experiments of this type range from differing communication protocols to comparing a single protocol with the absence of communication (this thesis attempts to do both). With the typically large computation, time, and hardware overheads, there are many differing opinions on whether or not communication is worth it. However, researchers do agree that it cannot be completely relied on, and that worst-case scenarios of total communication breakdown must be understood and dealt with.

Tyler *et. al.* perform experiments on a task scenario similar to the one used in this thesis [29] in which two Rug Warrior Pro robots search for a lighted target in a series of two rectangular areas with obstacles. Experiments are performed with a solo robot, a cooperative group, and a non-cooperative group. Surprisingly, time for task completion significantly degrades in the cooperative approach due to communication overheads. The experiment is redefined so that robots can visit the target multiple times per simulation, and a *following* behavior is added to both the robots' control systems. In this new set of experiments, communication has a strong positive effect; however, unlike the results presented in this thesis, communication has little effect on the distribution of time spent in each behavior. Several other experiments from the literature with the amount and type of communication were discussed in Chapter Two.

5.2 Search-and-Rescue Experiments

5.2.1 Simple Rooms, *Continuous Communication*

The *continuous communication* experiments utilize SARA-1 as shown in the flowchart of Figure 6 . In this set of experiments, communication occurs every loop of the code between each robot. This type of communication produces the maximum amount of cooperation as well as the maximum amount of time spent waiting on other robots. Twenty-eight simulations were performed, four each while changing the number of robots from one to seven. In each simulation, the target was alternately placed in two

different rooms of comparable difficulty (the two positions are shown in Figure 4 of Chapter Three).

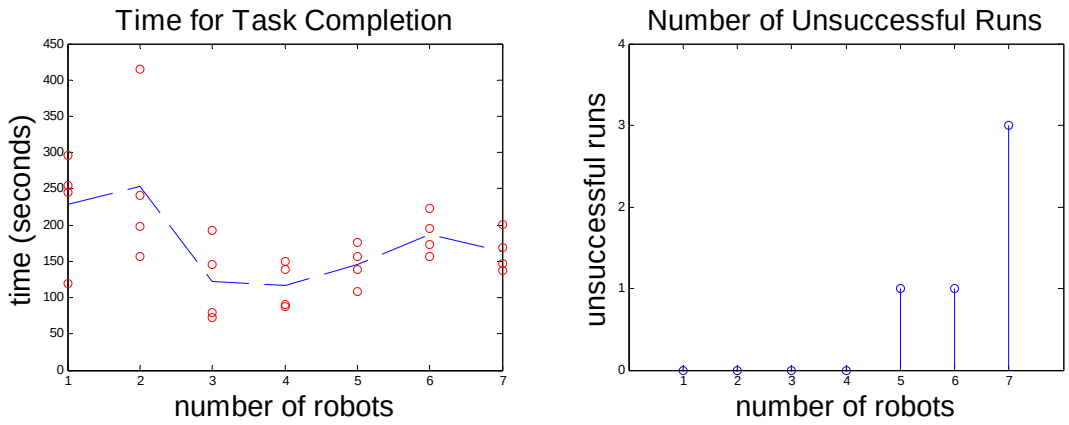


Figure 16: Graphs for Simple Rooms, *continuous communication*

The results from the *continuous communication* experiments are shown in Figure 16. The 'time for task completion' graph represents the time it takes for the first robot to find the target, which includes only the *search state*. The *rescue state* involves no direct cooperation and is therefore not included in this graph. Also, since there is no *rescue state* in the single robot approach, it would be impossible to compare it to multiple robot approaches. In this graph, each group of four points is averaged and connected to form a continuous 'average' line.

As the number of robots increases, the time for task completion decreases due to the positive effects of cooperation; however, once the number of robots passes a threshold, four in this set of experiments, negative byproducts of cooperation cause the time for task completion to rise. The two-robot approach produces a slight increase in the time for task completion over the single robot approach. This is possibly due to the

sudden introduction of communication overhead that is absent in the single-robot approach.

The ‘number of unsuccessful runs’ graph shows how many simulations were unsuccessful in each set of four experiments. “Unsuccessful” is defined as any of the robots not making it to the target. This definition includes both the *search* and *rescue states*. As seen in the graph, this number increases with the addition of more robots assisting in the task, often due to inter-robot interference. Two of the unsuccessful runs that occurred in the seven-robot experiments were due to a robot getting pinned between a wall and another robot and getting stuck on the wall. This shows the negative effect of inter-robot interference and gives a possible explanation for the slight decrease in the ‘time for task completion’ graph with seven robots. After a robot is stuck, it continues to assist in cooperative map-building, but does not produce any interference, giving a slight advantage over the six-robot experiments.

The other three unsuccessful runs were due to a single robot attempting to go through a wall to get to the target. This was explained in Chapter Four and occasionally occurs when a robot is just on the other side of a wall from the target when it is found. The robot calculates less cost to go through the wall than around it. It is possible for a robot to build up enough probability in the occupancy grid to make the cost of going through the wall more than the cost of going around it. This occurred once during this set of experiments and it only took the robot about a minute and a half to build up enough probability. It did not occur in the Broun Hall experiments, possibly due to the thin walls.

5.2.2 Simple Rooms, *Occasional Communication*

The experiments described above were performed again, but this time only allowing communication every third loop of the code. The changes in the code that made this possible are shown in Appendix B. During each *communication* behavior, three data packets are sent so that the same amount of information is swapped as in the *continuous* strategy, just at different intervals. *Path-planning* is still cooperative in each loop. Each robot simply uses the other robots' last known positions, which may be three loops old. This communication strategy makes a trade-off between cooperation and communication overhead. The robots only cooperate every third loop, but do not have to wait on each other as often. A negative byproduct of this approach is the possibility of increased inter-robot interference, since there is less frequent cooperation.

The same set of twenty-eight experiments was performed, varying the number of robots from one to seven, four experiments each[6]. The results are shown in Figure 17. Again, the 'time for task completion' has a parabolic shape due to the competing effects of cooperation with interference and communication overhead. There is also an unexplained dip in the graph at the six-robot experiments. Occasionally, in the cooperative experiments, one robot will get in front of the group and explore straight down the hallway and into the target's room (this was never observed in the single-robot

[6] The experiments for the single-robot approach were only performed once since the communication strategy does not have any effect on these experiments. Their results, however, are reprinted in all graphs for comparison.

or *no communication* approaches). The reason for the dip in the graph may be as simple as this, but more experiments would be needed to confirm this.

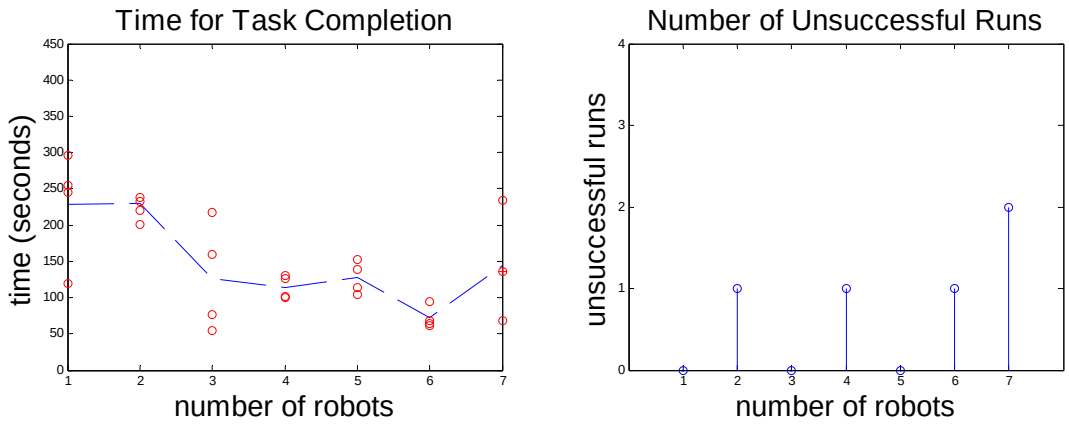


Figure 17: Graphs for Simple Rooms, *occasional communication*

The 'number of unsuccessful runs' graph shows more variation with this strategy than with the *continuous* strategy, signifying possible increased inter-robot interference. The unsuccessful runs were again a mixture of robots getting stuck and robots attempting to path-plan through walls. In one instance, two robots actually got stuck together.

The percent difference in time for task completion between the *continuous* and *occasional* strategies is shown in Figure 18. This information was calculated using equation 5-1. Except for the 3-robot case, completion times are smaller using *occasional communication* compared with using *continuous communication*. However, all differences are less than twelve percent, except for the anomalous result for 6 robots (which is related to the anomalously small completion time using *continuous communication*, shown in Figure 17). While the *occasional* strategy shows a slightly increased benefit, the inter-robot interference is much higher as will be shown in section 5.2.4.

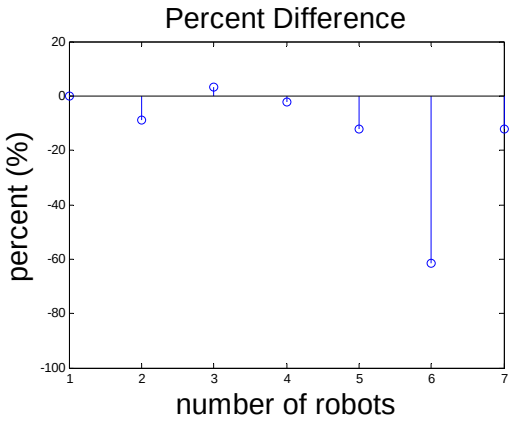


Figure 18: Percent difference of time for task completion for *occasional communication* relative to *continuous communication*

$$\%Diff = \frac{[time(occasional) - time(continuous)]}{time(continuous)} \bullet 100\% \qquad (5\text{-}1)$$

After observation of several different simulations, it was seen that robots enter the same rooms much more frequently with the *occasional communication* strategy. While robots can explore more quickly with *occasional communication*, they waste time by covering the same area multiple times. In one three-robot simulation, all robots were in the same room at the same time. This produced a large amount of interference, especially when they finally communicated, and two of them tried to leave the room simultaneously.

5.2.3 Simple Rooms, *No Communication*

In a third set of experiments, communication was not allowed and robots searched Simple Rooms as if they were searching alone. This produced a maximum amount of interference and proved to be only slightly better than the single-robot approach. The

results are shown in Figure 19. With the three- and four-robot experiments, there was a lot of variance in the data. In the experiments that fall below the 'average' line, interference actually helped by pushing one robot out in front of the others so that it almost immediately went to the room the target was in. It is extremely doubtful that this would occur in a more complex room. Regardless, there is never a major decrease in the average time for task completion. For this set of experiments, there is no 'number of unsuccessful runs' graph because, surprisingly, there were none. This wasn't because of a lack of inter-robot interference, as is seen in the next section.

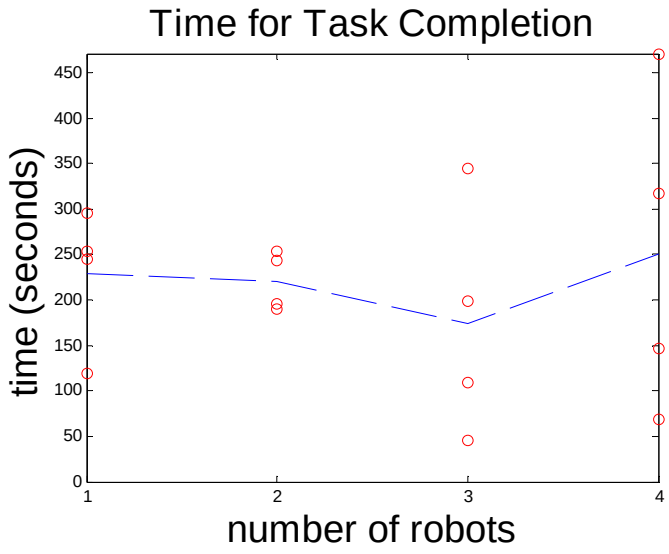


Figure 19: Graphs for Simple Rooms, *no communication*

5.2.4 Behavioral Comparison of the Three Communication Strategies

Inspired by [29], the time spent in each behavior was documented for each robot during all performed experiments. With this information it is possible to compare the three communication strategies to determine where time is being spent and how each different strategy affects the actions and interactions of robots. Each of the charts found

in this section presents averaged data from all robots that participated successfully in the task. So, for seven robots and four experiments, there is a maximum of twenty-eight values averaged together; for six robots, a maximum of twenty-four values, and so on. The only robots whose data were not used are those that were unsuccessful[7].

The overall distribution of behaviors for each of the three communication strategies is shown in Figure 20. The 'other' category includes the *robot initialization* and *dispersion* behaviors as well as data-logging, variable initialization, and other code overheads. The amount of time spent in the *map-building* behavior was recorded but was found to be negligible. It was discussed in Chapter Four that the *path-planning* behavior creates the most computational burden for the robot; however, as seen in each of the pie charts, this behavior accounts for a very small percentage of the overall task.

[7] The data of unsuccessful robots was not used in any graphs or charts except the 'number of unsuccessful runs' graphs. Using parameters such as the overall time and percent time spent in each behavior of an unsuccessful robot has little significance and would only bias averaged information away from its true value.

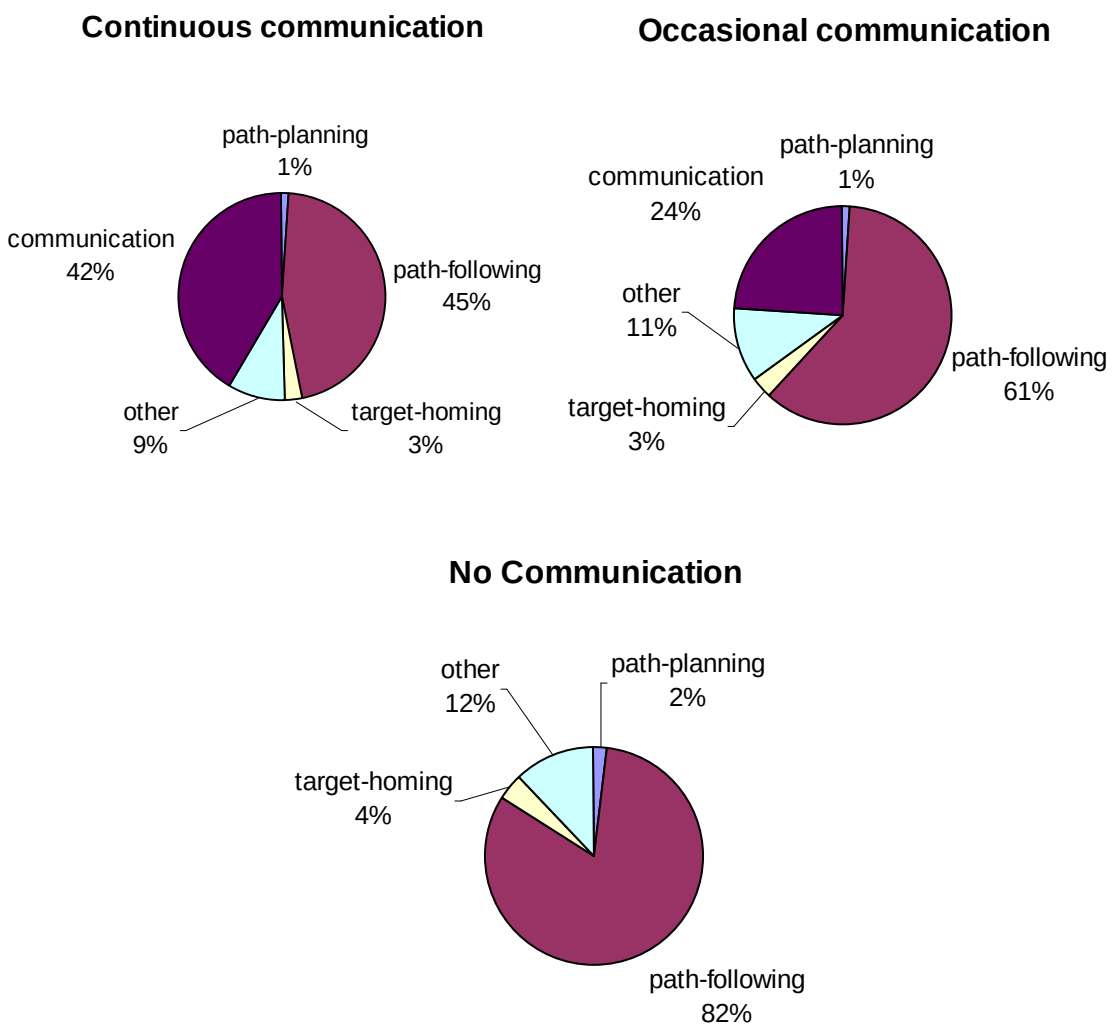


Figure 20: Distribution of behaviors in each communication approach

As expected, the percent of time spent in the *communication* behavior is significantly less in the *occasional communication* strategy compared to the *continuous communication* strategy. The *path-planning* and 'other' categories are very large for the *occasional* and *no communication* strategies due to the higher number of loops in the code. While the robots are not waiting for each other they are able to perform many loops of the code very quickly. The positive effect of this, however, is undone by the

lack of cooperation. It is important to note that, in general, a higher percentage of time spent in navigation without an evident increase in overall task performance is undesirable, especially in hazardous or unpredictable environments.

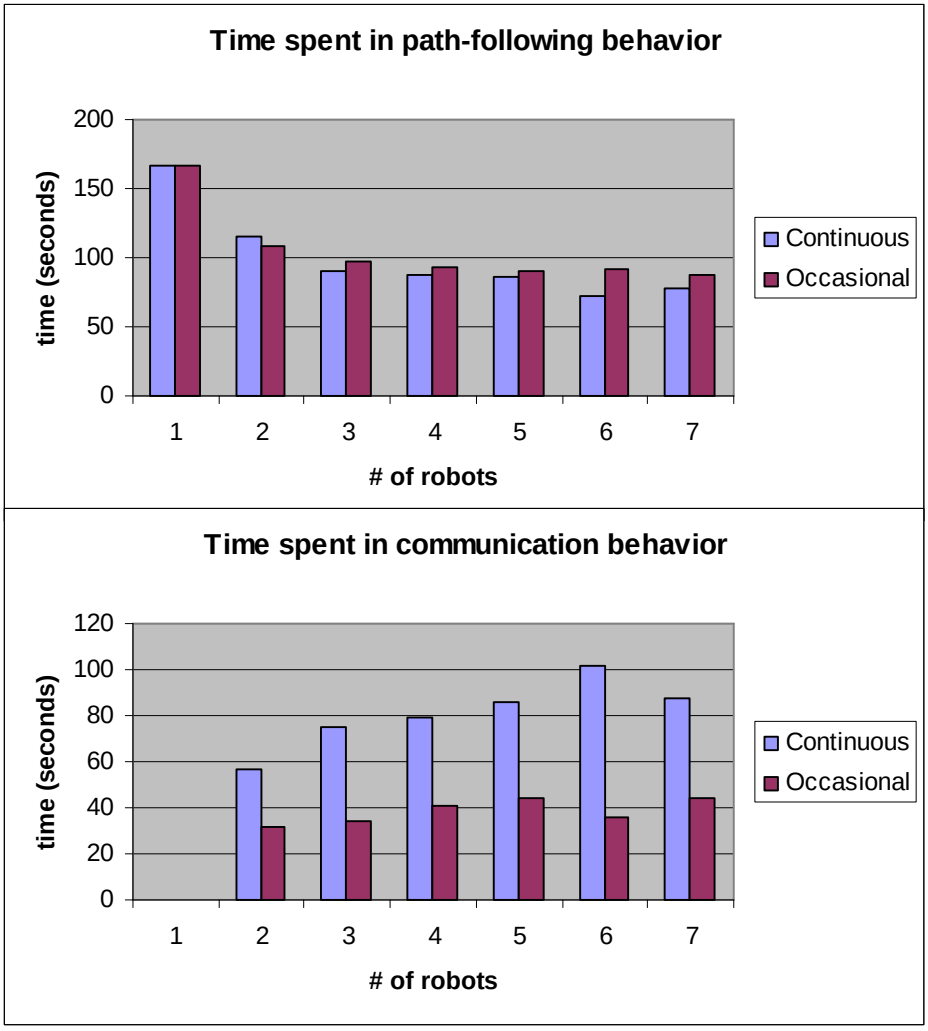


Figure 21: A closer look at the *path-following* and *communication* behaviors, comparing the *continuous* and *occasional communication* strategies

In an effort to get a better idea of what is going on in the experiments, the two largest categories from Figure 20, *path-following* and *communication,* are re-examined in Figure 21. In these charts, the distribution of each behavior is shown for each set of

experiments with differing numbers of robots. Only the *continuous* and *occasional* strategies are compared. As seen from the charts, time spent in the *path-following* behavior generally decreases as the number of robots is increased, while the time spent in *communication* generally increases (recall from sections 5.2.1 and 5.2.2 that there are unusual circumstances that occurred with the six- and seven-robot experiments). These are two of the opposing effects that bring out the parabolic shape in the 'time for task completion' graphs. As the number of robots increases, distance traveled is shared between robots, but the amount of time spent in *communication* grows.

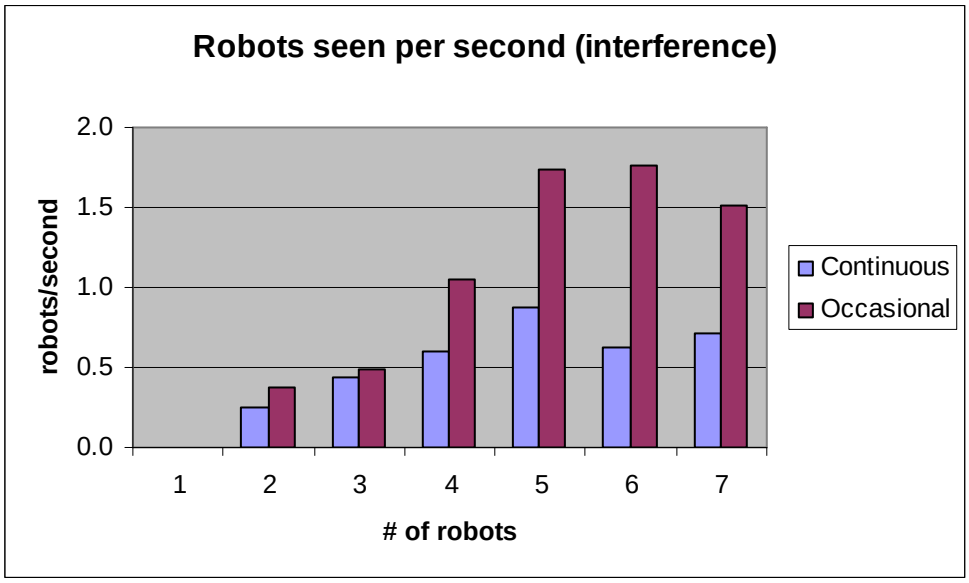


Figure 22: Comparison of inter-robot interference using the *continuous* and *occasional communication* strategies

Along with the time spent in each behavior, the number of robots seen per second was also recorded in each simulation. This parameter is updated each time SARA-1 uses the camera to count the number of blobs while looking for a blob of red color (the target).

This occurs several times per second throughout the *path-following* behavior. While this parameter is not significant by itself, using it for comparison provides a useful representation of inter-robot interference. A good algorithm and communication strategy will spread the robots out and keep them spread out, yielding a comparably low number of robots seen per second. As seen in Figure 22, inter-robot interference is a much bigger issue when using the *occasional communication* strategy compared to the *continuous communication* strategy—in many instances, the number is more than doubled. This is to be expected since there is less communication and therefore less cooperation.

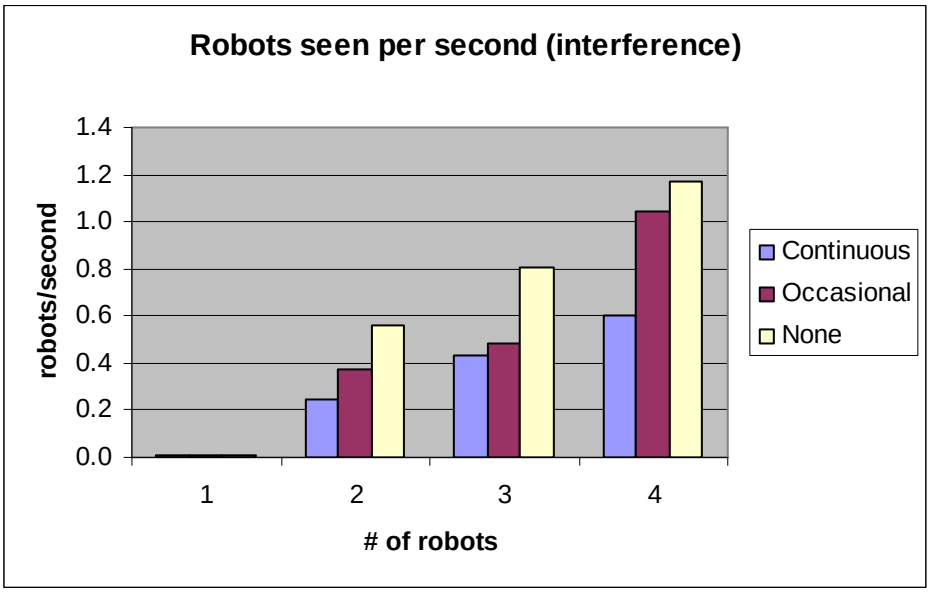


Figure 23: Comparison of inter-robot interference using all three communication strategies

As a final comparison, a chart showing the number of robots seen per second in all three communication strategies is shown in Figure 23, this time only comparing one to four robots. As expected, the most inter-robot interference occurs in the *no communication* strategy, almost doubling the *continuous* strategy in every set of

experiments. Another interesting aspect is the linearity of increase in the *no communication* strategy. An interesting experiment would be to see if this linearity continues beyond four robots.

5.2.5 Broun Hall, *Continuous Communication*

The experiments performed in Broun Hall tested the limits of SARA-1 in a realistic search environment. This large environment includes variable-sized rooms, thin walls, tight and uneven corners, and narrow openings. Before presenting the results from this set of experiments, a few comments are given about the performance of SARA-1 in this tough environment.

Figure 24 shows a partially completed global occupancy grid of Broun Hall built by five robots. The bitmapped environment is reprinted for comparison. Several corners are missing in the global occupancy grid. This is a common occurrence in the Broun Hall simulations and is due to the thin walls compared with the wide distances between sonar sensors. It has a negative effect on *path-planning* and *path-following* behaviors which ultimately causes robots to have trouble finding the exact locations of doors and navigating around corners. While the self-correcting nature of the *map-building* behavior will eventually fix these problems, robots often spend a lot of time hovering around these trouble spots, increasing the risk of getting stuck. The large room in the center of Broun Hall also gives the robots a lot of trouble. *Path-planning* into or out of this room typically requires a long detour around a wall and the behavior often attempts to send a robot through the wall instead.

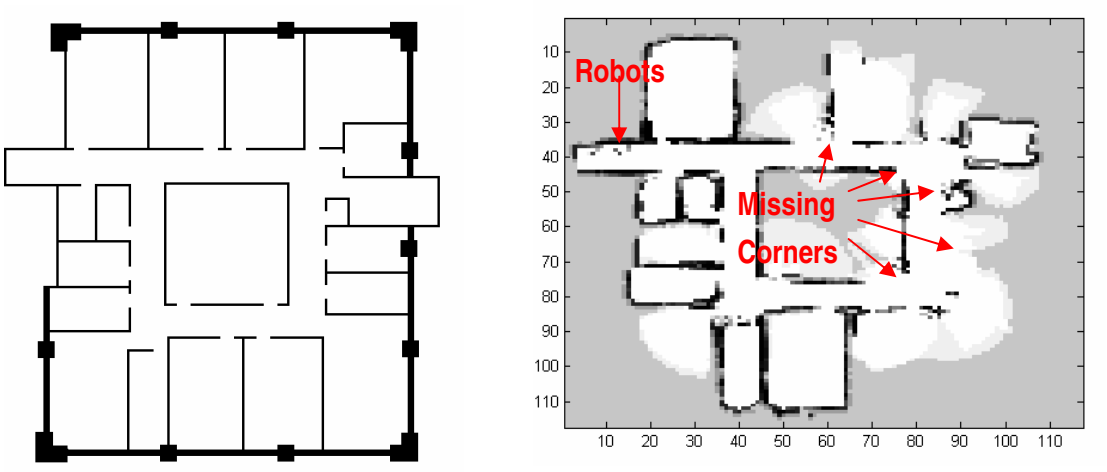


Figure 24: Broun Hall global occupancy grid showing missing corners

With these issues discussed, the Broun Hall experiment results are presented. Sixteen simulations were performed, four each while changing the number of robots from one to seven—odd numbers only. In each simulation, the target was alternately placed in two different rooms of comparable difficulty (the two positions are shown in Figure 5 of Chapter Three). As expected, it takes much more time to perform a search of Broun Hall than Simple Rooms. The time for searching in Simple Rooms ranged from fifty-four seconds to seven minutes, whereas the time for searching in Broun Hall ranged from three minutes to almost fifty.

The results from the Broun Hall experiments are shown in Figure 25. Due to the small number of experiments, very little information can be obtained from the graphs except that all cooperative experiments well out-perform the single-robot approach in time for task completion. Given the large search area and number of rooms, it is probable that the time for task completion will decrease even more as higher populations of robots

are used (provided *dispersion* goes well). The graph also shows a much higher variance in the data points when compared with any of the Simple Rooms experiments. This is another indication that more experiments need to be performed. The variance does not seem to be related to the placement of the target. The number of unsuccessful runs still tends to grow with the addition of more robots. Once again, the unsuccessful runs are due to robots getting stuck and robots attempting to path-plan through walls.

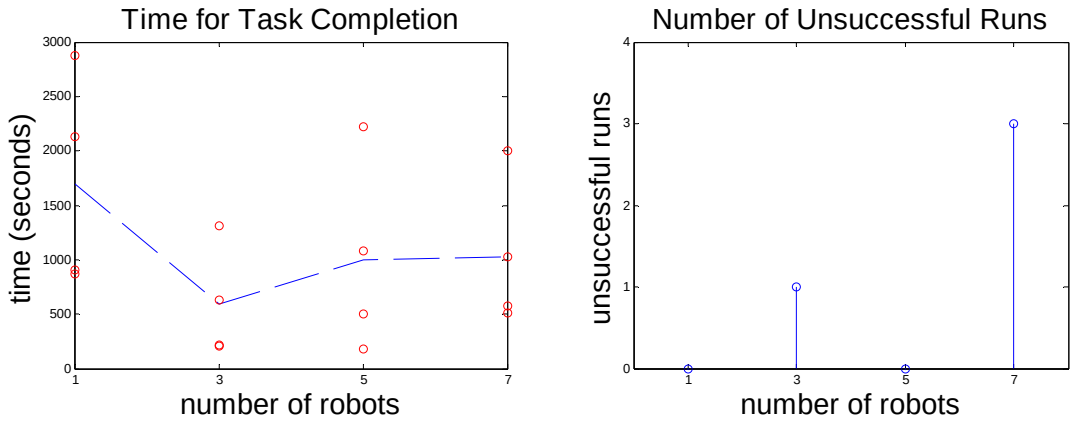


Figure 25: Graphs for Broun Hall, *continuous communication*

The time spent in each behavior may be of slightly more statistical significance since the data from every robot is averaged together. The behavioral distribution for the Broun Hall experiments is shown in Figure 26. This is the first set of experiments in which communication took up more than half of the overall task time. The reason for this is that there is a much higher potential for robots to have a long path to follow in Broun Hall than in Simple Rooms. These long paths potentially cause several robots to sit and wait for one robot that is trying to navigate a long path (recall that waiting is a major part of the *communication* behavior). *Path-planning* still takes up less than one percent of the overall task time, even in this large environment.

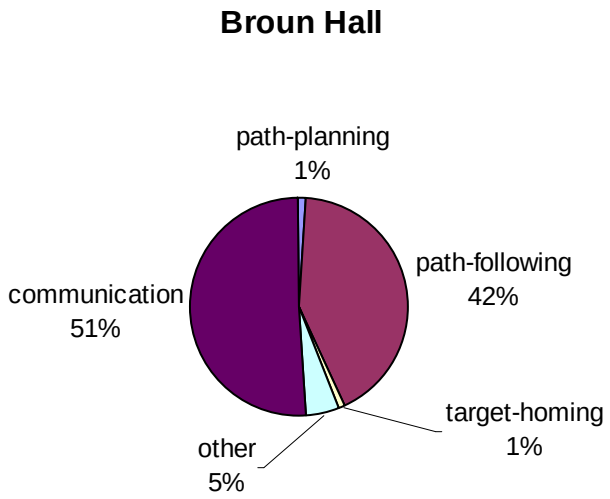


Figure 26: Distribution of behaviors in Broun Hall, *continuous communication*

Time spent in the *path-following* and *communication* behaviors is compared with the Simple Rooms (*continuous communication*) experiments and is shown in Figure 27. The Broun Hall results are very similar to those of the Simple Rooms. The time spent in *path-following* decreases while the time spent in *communication* increases with the addition of more robots. Of course, the actual amount of time spent in each behavior is much larger for the Broun Hall experiments.

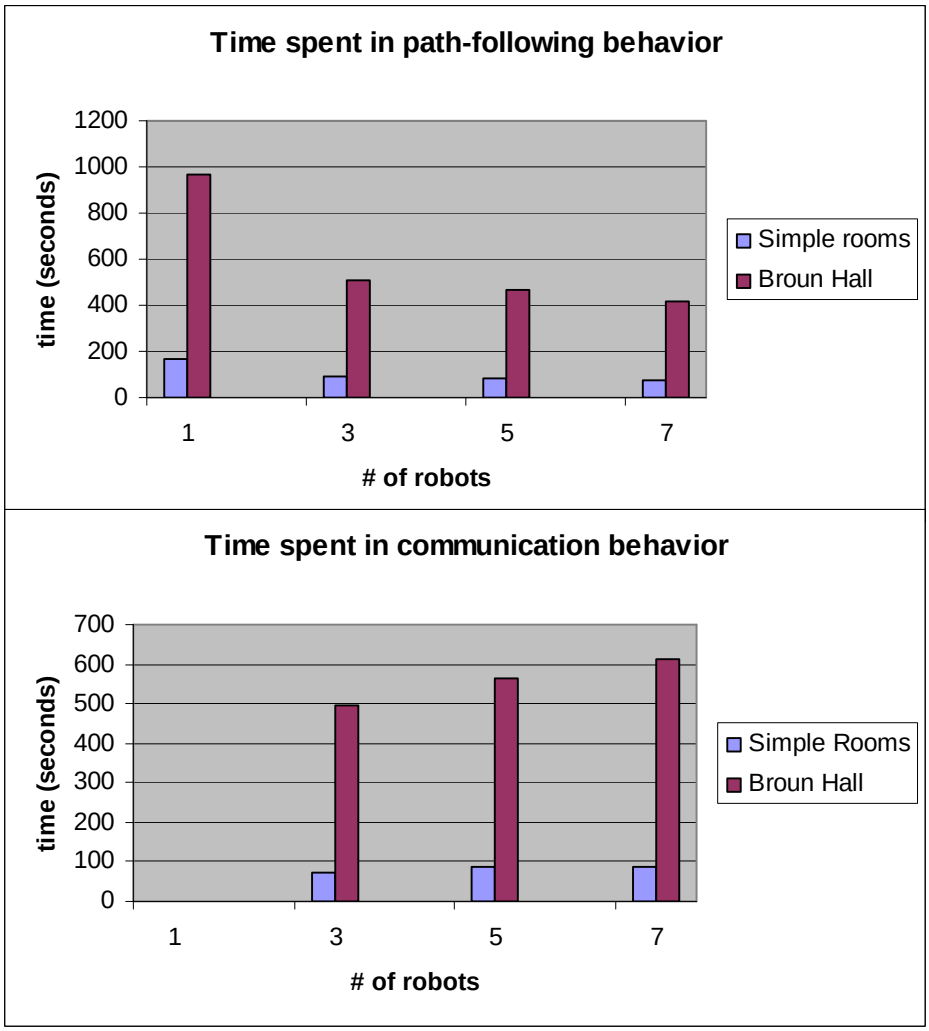


Figure 27: A closer look at the *path-following* and *communication* behaviors, comparing Broun Hall and Simple Rooms

Inter-robot interference in the Broun Hall experiments is shown in Figure 28 and is again compared with the interference of the Simple Rooms (*continuous communication*) experiments. Interference increases with the addition of more robots as expected, but the values are much less than those of the Simple Rooms experiments. The reason for this is that the larger area of Broun Hall brings out the full potential of SARA-1 to spread robots out while in the *search state*. For example, seven robots could search

Broun Hall for several minutes without ever running into each other. However, in the six rooms of Simple Rooms, there are at least two out of seven robots in the same room at all times. Once again, a linear increase in the Broun Hall interference is seen, much like in the *no communication* experiments of Simple Rooms. Whether or not there is any significance in this linearity remains unknown.

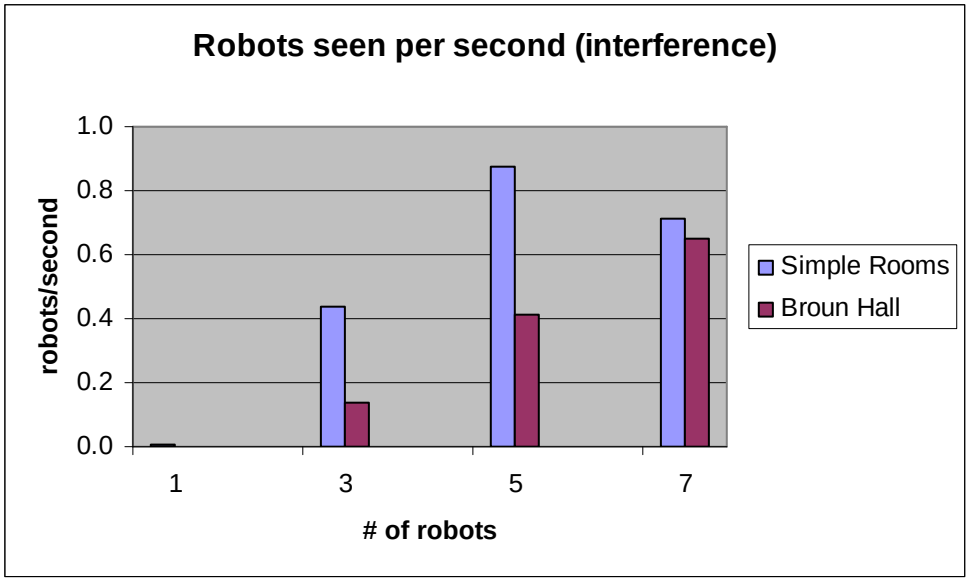
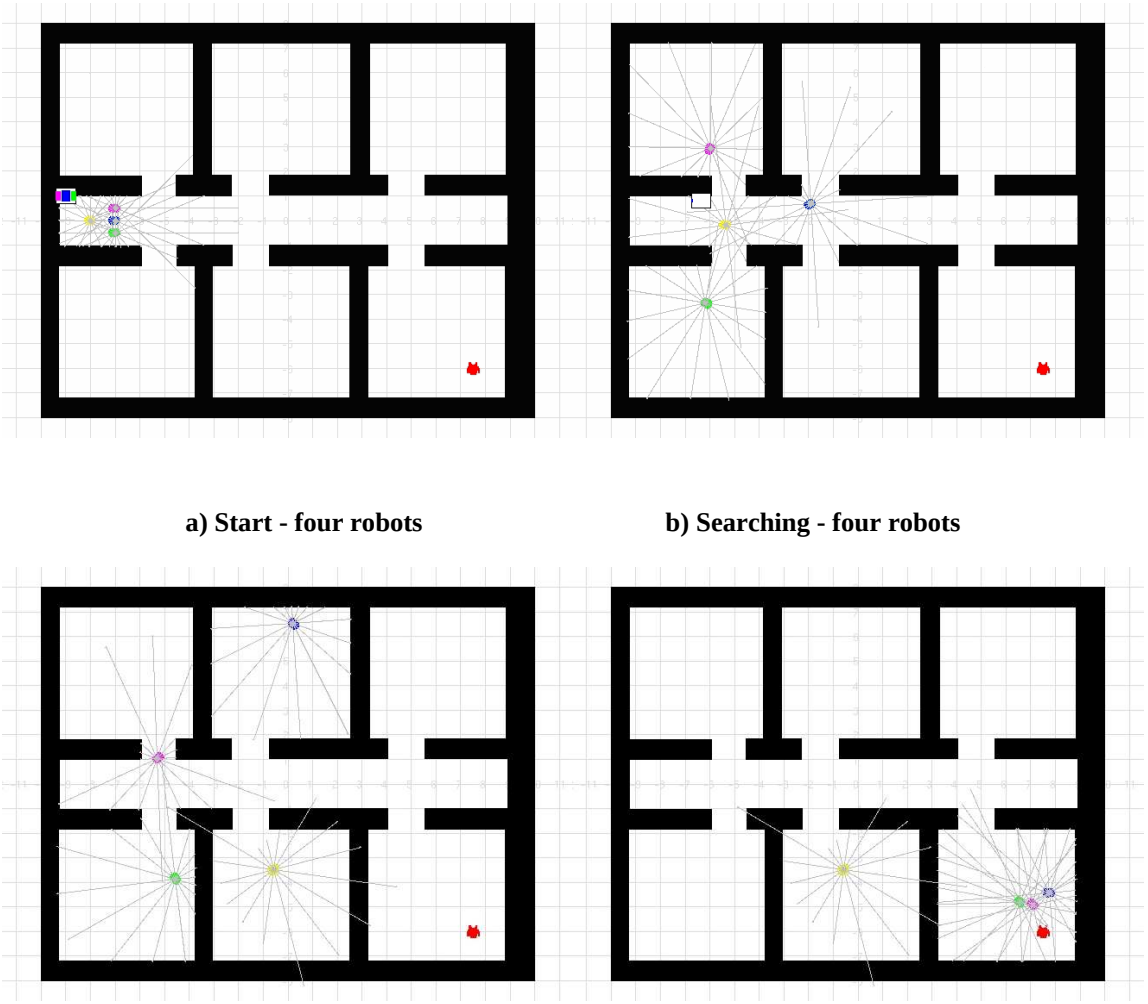


Figure 28: Comparison of inter-robot interference in Broun Hall and Simple Rooms

5.3 Example of a Malfunctioning Robot

Of the almost ninety experiments that were performed, there was not a single instance of a robot's communication malfunctioning (communication does occasionally malfunction even in simulation, and was observed in several trial runs). In order to show what would happen if communication were to malfunction during a search-and-rescue

task, an experiment is created which purposefully causes one robot to shutdown while in the *search state*. Screenshots are taken throughout this experiment and are shown in Figure 29.



a) Start - four robots **b) Searching - four robots**

c) Waiting - three robots **d) Complete - three robots**

Figure 29: Screenshots of a malfunctioning robot during a search-and-rescue task. Target is located in lower-right room

In Figure 29a, four robots begin a search in Simple Rooms for a target in the lower-right room. By Figure 29b, the robots have dispersed and spread themselves out

over half of the environment. In Figure 29c, when the robots are well into the search, the robot in the lower middle room shuts down. The other robots sit and wait, their impatience parameters growing. When the robots have been waiting for a certain amount of time (about thirty seconds in the current version of SARA-1), they delete that robot's channel and continue with the search—now with only three robots. Figure 29d shows the three functioning robots successfully at the target's position.

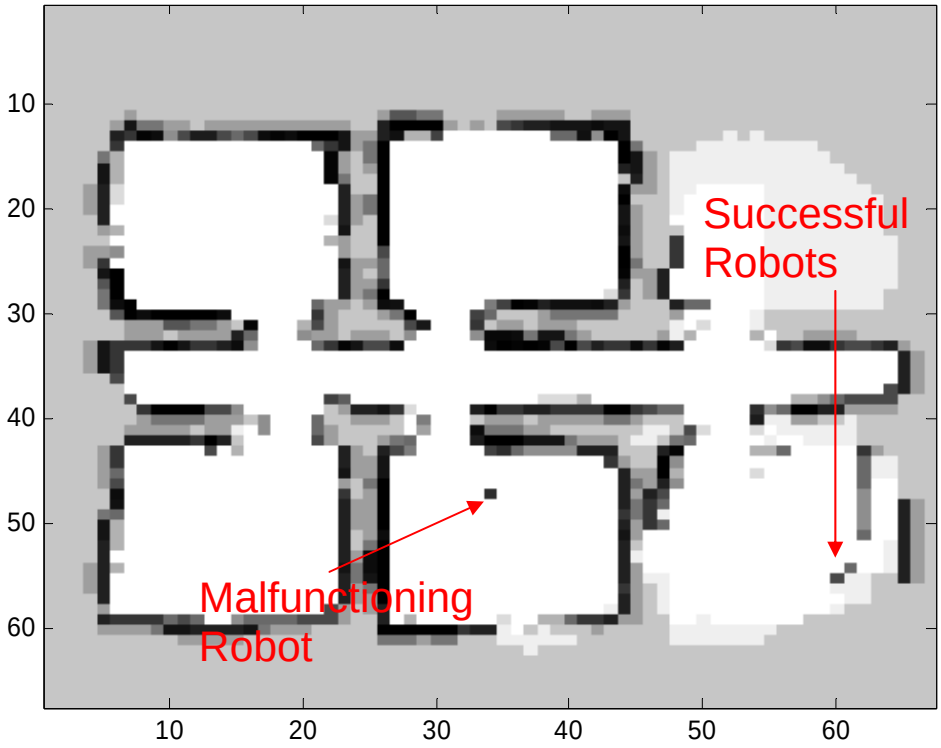


Figure 30: Final global occupancy grid showing the malfunctioning robot

Figure 30 shows the final global occupancy grid from the malfunctioning robot experiment. Since all robots are sensed as obstacles, and the malfunctioning robot did

not move for quite some time, this robot can be seen in the occupancy grid. The target can also be seen since it, too, is treated by the *map-building* behavior as an obstacle.

5.4 A Note about the Randomness of Simulations

In each set of four simulations, two simulations had an identical setup using one target, and two simulations had an identical setup using the other target. It may be suggested that two identical experimental setups in a simulation environment should produce exactly the same results. This, however, was not the case. It is likely that the inherent randomness in identical experiments is due to the internal multi-thread timing of Player and the allocation of the operating system. When Player starts up, it begins communicating with the control program every ten Hertz. If either the code or Player runs slightly faster in one simulation, or in one section of a simulation, randomness is introduced into the experiment. For example, if a robot is given a large turn-rate for just a millisecond too long, the robot will be oriented slightly differently and will draw a slightly different local occupancy grid. Once introduced, these slight changes are propagated throughout the simulation due to repeated robot interactions.

While the exact same simulation never occurred in any multi-robot experiment, there were a few instances in which the single-robot approach yielded two sets of identical simulation results (identical simulations are easy to determine by comparing the final global occupancy grids). Many researchers avoid this problem by starting the robots from several random locations [29]. This, however, is an undesirable fix for this research

since four experiments is not nearly enough for a statistical sample. Using random starting points would only introduce another variable that might falsely suppress or magnify the effects of cooperation.

CHAPTER SIX: CONCLUSIONS

A search-and-rescue algorithm for multiple robots cooperating through wireless communication has been presented. The algorithm, referred to as SARA-1, was shown to be robust, adaptable, and scalable both in theory and during experimental runs. It was also shown to respond well to the breakdown of communication and to the malfunctioning of other robots. It is applicable to both indoor and outdoor environments.

Several experiments were run in simulation using SARA-1. Three parameters were tested: the number of robots, the communication interval, and the complexity of the environment. In each set of experiments, the number of robots was varied and it was seen that there is typically an optimal number of robots which will minimize the time for task completion. This is due to a tradeoff between the benefits of cooperation and the burdens of communication overhead and inter-robot interference.

In the communication experiments, the communication interval was varied from *continuous*, to *occasional*, to *none*. Both sets of experiments using communication, and therefore cooperation, outperformed the *no communication* runs in terms of time for task completion. The *continuous* and *occasional communication* strategies implemented trade-offs between cooperation and time spent waiting. The *occasional* strategy only

slightly out-performed the *continuous* strategy, while creating much more inter-robot interference.

SARA-1 was tested in the more realistic environment of Broun Hall. The problems of the algorithm were magnified in this new environment, but did not have any major effects on overall results. The time for task completion values in the Simple Rooms communication experiments were relatively consistent. This was not the case for the Broun Hall experiments. In a search area of this size, a larger number of experiments still needs to be performed to have any statistical significance. The behavioral data collected from the Broun Hall experiments had slightly more statistical significance, and closely matched the results of the Simple Rooms experiments.

6.1 Contribution to the Field of Cooperative Robotics

This thesis has presented, to the author's knowledge, the first algorithm for multi-robotic search-and-rescue using wireless communication in the Player/Stage environment. There have been few experiments presented in the literature involving a cooperative search-and-rescue task and even fewer that involve inter-robot wireless communication.

The research presented in this thesis has provided significant groundwork for the Cooperative Robotics Research team at Auburn University. Robotic platforms that are capable of running the SARA-1 algorithm are currently in the design phase. This

research and the research of several other team members will pave the way for Auburn University to become a major contributor to the field of Cooperative Robotics.

6.2 Suggestions for Future Work

- Install Player/Stage on multiple machines to enable parallel simulation runs.
- Conduct further experiments in more complex environments.
- Use Player's "subscribing" ability instead of the MCom push/pop/read stack for inter-robot communication. This would require some alterations of SARA-1 but would eliminate the need for the sensor motes. It would also have an effect on the time for task completion.
- Modify the code to enable the handling of "newcomers"—robots added in during simulation.
- Fix the code to avoid the *path-planning* problem described in section 4.5.5.

REFERENCES

[1] A. Davids, “Urban search and rescue robots: from tragedy to technology,” in IEEE Intelligent Systems and their Applications, vol. 17, no. 2, 2002, pp. 81-83.

[2] A. Halme, et. al, “The concept of robot society and its utilization,” in Proceedings of IEEE/Tsukuba International Workshop on Advanced Robotics, 1993, pp 29-35.

[3] A. Jacoff, et. al, “Test arenas and performance metrics for urban search and rescue robots,” in Proceedings of the IEEE International Conference on Intelligent Robots and systems, vol. 4, 2003, pp. 3396-3403.

[4] A. M. Flynn and R. A. Brooks, “MIT mobile robots – What’s next?,” in Proceedings of IEEE International Conference on Robotics and Automation, vol. 1, 1988, pp.611-617.

[5] B. P. Gerkey, et al, “Most valuable player: a robot device server for distributed control,” in Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems, 2001, pp. 1226-1231.

[6] B. P. Gerkey, R. T. Vaughan, and A. Howard. “The player/stage project: tools for multi-robot and distributed sensor systems,” in Proceedings of the International Conference on Advanced Robotics, 2003, pp. 317-323.

[7] B. Yamauchi, “Frontier-based exploration using multiple robots,” in Proceedings of the Second International Conference on Autonomous Agents, 1998, pp. 47-53.

[8] C. E. Thorpe and L. H. Matthies, “Path Relaxation: path planning for a mobile robot,” in OCEANS, vol. 16, 1984, pp. 576-581.

[9] C. R. Kube and H. Zhang, “Collective Robotic Intelligence,” in Second International conference on Simulation of Adaptive Behavior, MIT Press, 1992, pp. 460-468.

[10] C. R. Kube, Collective Robotic Intelligence Project (CRIP) Home Page: http://www.cs.ualberta.ca/~kube/research.html

[11] C. R. Kube, E. Bonabeau, “Cooperative transport by ants and robots,” in Robotics and Autonomous Systems, vol. 30, 2000, pp. 85-101.

[12] C. R. Kube, H. Zhang, and X. Wang, “Controlling collective tasks with an ALN,” in Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems, vol. 1, 1993, pp 26-30.

[13] CMUcam Vision Sensors Home Page: http://www.cs.cmu.edu/~cmucam/

[14] Cooperative Robotics Research Team’s Home Page: http://crr.eng.auburn.edu

[15] D. J. Turnell and M. D. F. Q. V. Turnell, “SimBot—a simulation tool for autonomous robots,” in IEEE Conference on Systems, Man, and Cybernetics, vol. 5, 2001, pp. 2986-2990.

[16] E. P. Dadios and O. A. Maravillas Jr., “Cooperative mobile robots with obstacle and collision avoidance using fuzzy logic,” in Proceedings of IEEE International Symposium on Intelligent Control, 2002, pp 75-80.

[17] Gumstix Home Page: http://www.gumstix.com/

[18] H. Asama, A. Matsumoto, and Y. Ishida, “Design of an autonomous and distributed robot system – ACTRESS,” in Proceedings of the IEEE/RSJ International Workshop on Intelligent Robots and Systems, 1989, pp. 283-290.

[19] H. Asama, et. al, “Collision avoidance among multiple mobile robots based on rules and communication,” in Intelligent Robots and Systems, vol. 3, 1991, pp 1215-1220.

[20] H. P. Moravec and A. Elfes, “High resolution maps from wide angle sonar,” in Proceedings of the IEEE International Conference on Robotics and Automation, vol. 2, 1985, pp. 116-121.

[21] J. Casper and R. R. Murphy, “Human-robot interactions during the robot-assisted urban search and rescue response at the world trade center,” in IEEE Transactions on Systems, Man, and Cybernetics – Part B: Cybernetics, vol. 33, no. 3, 2003, pp. 367-385.

[22] J. L. Crowley, “Navigation for an intelligent mobile robot,” in IEEE Journal of Robotics and Automation, vol. RA-1, no. 1, 1985, pp. 31-41.

[23] J. Liu and J. Wu. Multi-Agent Robotic Systems. CRC Press: London, 2001.

[24] J. S. Jennings, G. Whelan, and W. F. Evans, “Cooperative search and rescue with a team of mobile robots,” in Proceedings of ICAR Internatonal Conference on Advanced Robotics, 1997, pp. 193-200.

[25] K. T. Simsarian and M. J. Matarić, “Learning to Cooperate using two six-legged mobile robots,” in Proceedings of the 8th European Conference on Machine Learning, 1995.

[26] L. Cahut, K. P. Valavanis, and H. Delic, “Sonar resolution-based environment mapping,” in Proceedings of the IEEE International Conference on Robotics and Automation, 1998, pp. 2541-2547.

[27] L. E. Parker, “ALLIANCE: An architecture for fault tolerant multi-robot cooperation,” in IEEE Transactions on Robotics and Automation, vol. 14, no. 2, 1998, pp 220-240.

[28] L. E. Parker, “Current state of the art in distributed autonomous mobile robotics,” in Distributed Autonomous Robotic Systems 4, Springer-Verlag Tokyo, 2000, pp 3-12.

[29] L. Tyler, P. Innocent, and R. John, “Co-operation and interference in mobile robot groups,” in International Conference on Mechatronics, 2003, pp. 549-554.

[30] M. J. Matarić, “Behavior-based control: main properties and implications,” in Proceedings of Workshop for Architectures for Intelligent Control Systems, IEEE International Conference on Robotics and Automation, 1992, pp. 46-54.

[31] M. J. Matarić, “Issues and approaches in the design of collective autonomous agents,” in Robotics and Autonomous Systems, vol. 16, 1995, pp 321-331.

[32] M. J. Matarić, “Minimizing complexity in controlling a mobile robot population,” in Proceedings of IEEE International Conference on Robotics and Automation, vol. 1, 1992, pp 830-835.

[33] M. J. Matarić, M. Nilsson, and K. T. Simsarian, “Cooperative multi-robot box-pushing,” in Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems, vol. 3, 1995, pp 556-561.

[34] M. Schneider-Fontán and M. J. Matarić, “Territorial multi-robot task division,” in IEEE Transactions on Robotics and Automation, vol. 14, no. 5, 1998, pp. 815-822.

[35] M. Vainio, P. Appelqvist, and A. Halme, “Generic control architecture for a cooperative robot system,” in Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems, vol. 2, 1998, pp 1119-1125.

[36] N. Hutin, C. Pégard, and E. Brassart, “A communication strategy for cooperative robots,” in Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems, vol. 1, 1998, pp 114-119.

[37] Player/Stage Project Home Page: http://playerstage.sourceforge.net

[38] R. A. Brooks, “A robust layered control system for a mobile robot,” in IEEE Journal of Robotics and Automation, 1986, pp 14-23.

[39] R. C. Arkin and J. Diaz, “Line-of-sight constrained exploration for reactive multi-agent robotic teams,” in7th International Workshop on Advanced Motion Control, 2002, pp 455-461.

[40] R. C. Arkin, T. Balch, and E. Nitz, "Communication of behavioral state in multi-agent retrieval tasks," in Proceedings of IEEE International Conference on Robotics and Automation, vol. 3, 1993, pp 588-594.

[41] R. Simmons, et. al, “Coordination for multi-robot exploration and mapping,” in Proceedings of the National conference on Artificial Intelligence, 2000, pp. 852-858.

[42] R. Trieb and E. von Puttkamer, “3d7—simulation environment a tool for autonomous mobile robot development,” in Proceedings of the IEEE International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 1994, pp. 358-361.

[43] S. A. Redfield, “Land finding using homogeneous groups of cooperating autonomous vehicles,” in IEEE/OES Autonomous Underwater Vehicles, 2004, pp. 26-31.

[44] S. Bergbreiter and K. S. J. Pister, “CotsBots: An off-the-shelf platform for distributed robotics,” in IEEE International Conference on Intelligent Robots and Systems, vol. 2, 2003, pp. 1632-1637.

[45] T. Fukuda and S. Nakagawa, “A dynamically reconfigurable robotic system,” in Proceedings of the International Conference on Industrial Electronics, Controls, and Instrumentation, vol. 2, 1987, pp. 588-595.

[46] T. Fukuda, S. Nakagawa, Y. Kawauchi, and M. Buss, “Structure decision method for self organizing robots based on cell structures – CEBOT,” in Proceedings of the IEEE International Conference on Robotics and Automation, 1989, pp. 695-700.

[47] Tmote Sky Home Page: http://www.moteiv.com/products.php

[48] V. P. Burhanpurkar, “Real world application of a low-cost high-performance sensor system for autonomous mobile robots,” in IEEE International Conference on Intelligent Robots and Systems, vol. 3, 1994, pp. 1840-1844.

[49] W. Burgard, et. al, “Collaborative multi-robot exploration,” in Proceedings of the IEEE International Conference on Robotics and Automation, vol. 1, 2000, pp. 476-481.

APPENDICES

APPENDIX A: SARA-1 CODE

This appendix includes the SARA-1 code in its entirety. The code is written in C and C++. It uses classes and function calls available to the Player-1.6.4 software server. More information can be obtained about this software and its functions at [37]. All Player-related files that are necessary for running this code in the Stage simulator can be found in Appendix C.

```
/*  Algorithm:  Search and Rescue for multiple cooperative robots
    Date:  October 1, 2005
    Author:  Adam Ray <surge20_99@yahoo.com>
*/

#include <stdio.h>
#include <stdlib.h>           //for atoi()
#include <playerclient.h>     //for player client stuff
#include <string.h>           //for strcpy()
#include <cmath>              //for trig functions
#include <math.h>             //for multiple functions
#include <time.h>             //for clock()
#include <iostream>           //for C++ functions, such as cerr
using namespace std;          //also for C++ functions

//Define Arguments used when calling search program
#define USAGE \
  "USAGE: search [-h <host>] [-p <port>] [-N <robots>] [-M <size>] [-x
<dist>] [-y <dist>] [-s <speed>] [-r <res>] [-d <dist>]\n" \
  "          -h <host>   : connect to Player on this host, default =
'localhost'\n" \
  "         -p <port>  : connect to Player on this TCP port, default =
6665\n" \
  "        -N <robots>: Number of robots (integer)--these robots should
be on ports { 6665 , 6665+N }, default = 1 robot\n" \
  "                   : can not be more than 20 robots\n" \
  "       -M <size>  : search area size (float) (meters)--creates a M x
M map, default = 25 meters\n" \
  "                    : cannot be greater than 75 meters for a default
resolution of 30cm (see below)\n" \
```

```
  "       -x <dist>  : approximate starting x distance (float) (meters)
from top left corner of map, default = 0\n" \
  "       -y <dist>  : approximate starting y distance (float) (meters)
from top left corner of map, default = 0\n" \
  "          -s <speed> : move at this speed (float) (meters/second),
default = 1 meter/second\n" \
  "                     : Be Careful! Path_following is tuned for default
speed, speeds too high may cause runaway robots\n" \
  "       -r <res>   : resolution of occupancy grids (float) (meters),
default = 30centimeters\n" \
  "                     : cannot go below 10cm (this minimum resolution
can only be used in a 25m search area or smaller)\n" \
  "           -d <dist>   : minimum tolerated sonar distance (float)
(meters), default = 0.5 meters\n"

//------------------Declare Program Variables-------------------------
//type defines
typedef struct coordinates {//to store [x,y] coordinates
  float x;
  float y;
} point;
typedef struct polar_coord {//to store [x,y,theta] coordinates
  float x;
  float y;
  float theta;
} odometry;
typedef struct cell_indexes {//to access occupancy grid cells
  int i;
  int j;
} cell_index;
typedef struct position_commands {// to send commands to robot
  float speed;
  float turnrate;
} command;

//VARIABLES ACCESSIBLE TO USER (via parse_args)
char host[256] = "localhost"; //host for Player to use
int port = PLAYER_PORTNUM;    //port number for Player to use
                              //each robot must be on a separate port!
                  int NUM_OF_ROBOTS = 1;        //the number of robots
                  searching //initialized to 1 for the robot running
                  this code
float SEARCH_AREA_SIZE_IN_METERS = 25;    //defines size of search area
for map building, try to overestimate a little.  Search area can be
very large, but change GRID_CELL_SIZE if more than 75 meters
point MY_SHIFT={0,0};         //initial {x,y} distance from top-left
corner in Global Map
float SPEED = 1.000;          //set the normal speed of the robot
float GRID_CELL_SIZE = 0.3;   //RESOLUTION:  size of grid cells in
occupancy grid (30 centimeters~=size of robot).  This must not go below
10cm, or program will attempt to write outside of maps
float MINFRONTDISTANCE = 0.50;//set sonar threshold to avoid obstacles

//VARIABLES ACCESSIBLE TO PROGRAMMER ONLY
//program statistics variables—used with time-per-behavior calculations
```

```
int wait_count=0;
int num_blobs_seen=0;
float communication_time = 0;
float path_planning_time = 0;
long path_following_time = 0;
long camera_homing_time = 0;
long search_and_rescue_total_time = 0;
float  Position_tracker[256][256];//To plot out the path of the robot throughout simulation (for show only)

//robot variables
odometry robot_positions[20];     //to store positions (x,y,theta) of other robots, assumed < 20 robots
point ROBOT_SHIFTS[20];           //to store she SHIFT (x,y) passed to each robot in the program via parse_args.  Variables are converted to grid cells in robot_initialization.  This is how robots "know" their relative positions to one another.   Each of the above variables correspond to the robots in MCOM_CHANNEL_ROBOTS below
int ME=0;         //to keep up with my number in MCOM_CHANNEL_ROBOTS

//sonar variables
int NUM_OF_SONARS = 16;       //number of sonar sensors
float MAX_SENSING_DIST = 5;   //maximum sensing distance of sonar sensors (5 meters)
float SONAR_ANGLES[16];       //angular pose of the sonar sensors
float sonar_readings[16];     //holds sonar readings to pass to map_building behavior...assumes 16 sonar sensors

//map variables
float SIZE_OF_ROBOT = 0.3;    //size of robot--default=30 centimeters
int GLOBAL_MAXX,GLOBAL_MAXY;  //to access Global_Map
float Global_Map[256][256];   //Global Map of entire search are, 256 is maximum, although rarely used.    This maximum allows 10cm (30cm) GRID_CELL_SIZE for a 25m (75m) search area
point temp_points[100];       //used to return arrays from the "get_points_between" function.  Array size=100 to ensure big enough if MAX_SENSING_DIST or GRID_CELL_SIZE changes.  With default settings, the largest array size is only 17
float OPEN=0.4;               //to label cells in Global_Map as being occupied by an obstacle or open (obstacle-free)
float OCCUPIED=0.9;

//Cost and path_planning variables
float Cost_Grid[256][256];    //to store cost of traveling to a frontier cell
int Visibility[201];            //to store the number of times a robot measures 0, 0.1, 0.2, 0.3, ... with sonar sensors.  Variable is changed each time sonars are read and is used in the path_planning behavior.  '201' allows for a MAX_SENSING_DIST of 20 meters
point  the_path_to_follow[512];//to store the path for a robot to follow, assuming no path will be longer than 2*256

//target related variables
int robot_that_found_target;  //to store the number of the robot that found the target
```

```
odometry target_location;       //to store the position of the target
when found
bool target_homing=false;     //FLAG to signal if robot currently knows
where robot is and is homing toward it (rescue phas)
int red = 16711680;           //colors defined for "blob" detection
int blue = 255;
int green = 65280;
int THE_TARGET_COLOR = red;    //target is recognized by its red color—
may be changed

//communication variables
char    MCOM_CHANNEL_TARGET_FOUND[MCOM_CHANNEL_LEN]    =    "target";//to
communicate target's location when found
char  MCOM_CHANNEL_ROBOTS[20][MCOM_CHANNEL_LEN];//to  send  and  receive
robots' sonar readings, assumed <20 robots
int MCOM_TYPE_INITIALIZATION = 0;   //used   at   start   of   program   on
initialization channel
int MCOM_TYPE_SONAR_READINGS = 1;   //used    on    robots'    channels
(incremented each communication to ensure data is new)
int MCOM_TYPE_TARGET_FOUND = -1;    //used on target found channel
char message_packet[6];       //character  string  to  temporarily  store
message packets

//------------------Declare Program Functions-------------------------
void parse_args(int argc, char** argv);
void robot_initialization(PlayerClient &robot,PlayerClient &mcomPlayer,
MComProxy &mcp);
command obstacle_avoidance(SonarProxy &sp);
void   send_sonars(PlayerClient   &robot,   PlayerClient   &mcomPlayer,
MComProxy &mcp, odometry Location, bool target_channel);
odometry receive_sonars(PlayerClient &robot, PlayerClient &mcomPlayer,
MComProxy &mcp, char channel[MCOM_CHANNEL_LEN], bool target_channel);
void map_building(odometry robot_odometry, int robot_number);
int path_planning(PlayerClient &robot, PlayerClient &mcomPlayer, bool
go_to_target);
bool path_following(PlayerClient &robot, PlayerClient &mcomPlayer, int
the_path_length, BlobfinderProxy &bfp, PositionProxy &pp, SonarProxy
&sp);
bool   camera_homing(PlayerClient   &robot,   PlayerClient   &mcomPlayer,
BlobfinderProxy   &bfp,   PositionProxy   &pp,   SonarProxy   &sp,   bool
go_to_target);
command homing(odometry the_robot, point the_goal);
int get_points_between(point point1,point point2,float spacing);
bool check_for_target(BlobfinderProxy &bfp,PositionProxy &pp);
void log_information(int log_type,char filename[]);
void ftoa(int num_digits, float float_number);
float inv_ftoa(int num_digits);
void robot_shutdown();
void pause_four_seconds(PlayerClient &robot, PlayerClient &mcomPlayer);

//---------------------------MAIN-------------------------------------

int main(int argc, char **argv)
{
  parse_args(argc,argv);      //get program input arguments
```

```
  //variables
  int ii,jj;                  //used in for loops
  int loop_count;             //to initialize search state
  bool wait;                  //FLAG to let the robot know to wait on
other robots that are in the way
  int path_length;            //to store the returned path_length from
the path_planning behavior
  command go_to_obs={SPEED,0};//to send position commands to robot
  odometry my_last_position={0,0,0};//to detect if robot is stuck, we
keep up with the last position in each loop
  bool  target_seen=false,target_found=false;//FLAGS  to  signal  the
finding of the target
  bool Stuck = false;         //FLAG to singal that robot is unable to
move

  //initialize variables
  GLOBAL_MAXX = (int)ceil(SEARCH_AREA_SIZE_IN_METERS / GRID_CELL_SIZE);
      //initialize variables with new info from parse_args
  GLOBAL_MAXY = GLOBAL_MAXX;
  SONAR_ANGLES[0]  =  DTOR(90);      //initialize  sonar  pose  for  each
sensor...these will not change
  SONAR_ANGLES[1] = DTOR(67.5);
  SONAR_ANGLES[2] = DTOR(45);
  SONAR_ANGLES[3] = DTOR(22.5);
  SONAR_ANGLES[4] = DTOR(0);
  SONAR_ANGLES[5] = DTOR(-22.5);
  SONAR_ANGLES[6] = DTOR(-45);
  SONAR_ANGLES[7] = DTOR(-67.5);
  SONAR_ANGLES[8] = DTOR(-90);
  SONAR_ANGLES[9] = DTOR(-112.5);
  SONAR_ANGLES[10] = DTOR(-135);
  SONAR_ANGLES[11] = DTOR(-157.5);
  SONAR_ANGLES[12] = DTOR(180);
  SONAR_ANGLES[13] = DTOR(157.5);
  SONAR_ANGLES[14] = DTOR(135);
  SONAR_ANGLES[15] = DTOR(112.5);

  for(jj=0;jj<GLOBAL_MAXY;jj++)//initialize Global_Map to all 0.5's
  {
    for(ii=0;ii<GLOBAL_MAXX;ii++)
    {
      Global_Map[ii][jj] = 0.5;
    }
  }
  for(jj=0;jj<GLOBAL_MAXY;jj++)//initialize Position_tracker to all 0's
  {
    for(ii=0;ii<GLOBAL_MAXX;ii++)
    {
      Position_tracker[ii][jj] = 0;
    }
  }
  for(ii=0;ii<201;ii++)        //initialize Visibility to all 0's
    Visibility[ii] = 0;
  for(ii=0;ii<NUM_OF_ROBOTS;ii++)//initialize robots' positions to 0's
```

```
  {
     robot_positions[ii].x = 0;
     robot_positions[ii].y = 0;
     robot_positions[ii].theta = 0;
  }

  //initialize robot...these commands start the Player server reading
from the simulated hardware
  PlayerClient robot(host,port);    //initialize robot client
  PositionProxy pp(&robot,0,'a');   //initialize position
  SonarProxy sp(&robot,0,'r');      //initialize sonar sensors
  BlobfinderProxy bfp(&robot, 0, 'r');//initialize blob-finder camera
  PlayerClient mcomPlayer(host,6664);//initialize communication client
  MComProxy mcp(&mcomPlayer,0,'a'); //initialize communication device

  //initialize search state
  robot_initialization(robot,mcomPlayer,mcp);//check communication link
with all robots and initialize channels

  //Next ~60 lines of code starts the robots moving.   If there are
multiple robots, the robot on port #6665 begins the task by moving
forward a certain distance to get away from other robots and build the
first map.   While this isn't necessary, it makes for a much better
initial search if the first map is clean and robot-free
  loop_count = 10 * (int) (3/SPEED);      //get time it will take a
robot to go three meters--multiplied by 10 because this code waits on
Player, which communicates at a rate of 10Hz.
  if(NUM_OF_ROBOTS > 1 && port == 6665)   //if I'm robot # 6665 and not
searching alone, initialize search state
  {
    for(ii=0;ii<loop_count;ii++)
    {
      //wait for new robot data (10Hz by default)
      if(robot.Read()){cerr <<endl <<"HELP"; exit(1);}
      if(mcomPlayer.Read()){cerr <<endl <<"HELP"; exit(1);}
      if((sp[3] + sp[4] + sp[5])/3 < MINFRONTDISTANCE)//only avoid
obstacles in front (don't avoid robots on side)
        go_to_obs = obstacle_avoidance(sp);
      else
      {
        go_to_obs.speed = SPEED;          //else, full speed ahead
        go_to_obs.turnrate = 0;
      }
      pp.SetSpeed(go_to_obs.speed,go_to_obs.turnrate);//send commands
    }
    pp.SetSpeed(0,0);                     //stop robot
    //we perform Read() commands twice to make sure robot is stopped,
and sonar data is accurate before map_building
    if(robot.Read()){cerr <<endl <<"HELP"; exit(1);}
    if(mcomPlayer.Read()){cerr <<endl <<"HELP"; exit(1);}
    if(robot.Read()){cerr <<endl <<"HELP"; exit(1);}
    if(mcomPlayer.Read()){cerr <<endl <<"HELP"; exit(1);}
    for(ii=0;ii<NUM_OF_SONARS;ii++)       //get sonar data
      sonar_readings[ii]=sp.ranges[ii];
    robot_positions[ME].x=pp.xpos;        //get robot's own location
```

```
    robot_positions[ME].y=pp.ypos;
    robot_positions[ME].theta=pp.theta;
    send_sonars(robot,mcomPlayer,mcp,robot_positions[ME],false);
    map_building(robot_positions[ME],ME);       //build Global map
  }
  else if(NUM_OF_ROBOTS > 1 && port != 6665)    //if not robot # 6665,
wait for robot 6665 to communicate and disperse
  {
    wait = true;                                //initialize wait FLAG
    for(ii=0;ii<(loop_count+10);ii++)           //wait about the amount
of time it will take robot 6665 to initialize
    {
      //wait for new robot data (10Hz by default)
      if(robot.Read()){cerr <<endl <<"HELP"; exit(1);}
      if(mcomPlayer.Read()){cerr <<endl <<"HELP"; exit(1);}
      if(wait || ii < 15)           //wait if other robots are in front
or if robot 6665 is just starting
      {
        wait = false;
        for (jj = 0;jj<bfp.blob_count;jj++)//check all blobs (robots)
to see if I need to wait
        {
          if (bfp.blobs[jj].area > 300)   //if blob is too close,
            wait = true;                  //keep waiting
        }
      }
      else
      {
        if((sp[3] + sp[4] + sp[5])/3 < MINFRONTDISTANCE)//only avoid
obstacles in front (don't avoid robots on side)
          go_to_obs = obstacle_avoidance(sp);//If something in front,
get commands to avoid obstacles
        else
        {
          go_to_obs.speed = SPEED;        //else, full speed ahead
          go_to_obs.turnrate = 0;
        }
        pp.SetSpeed(go_to_obs.speed,go_to_obs.turnrate);//send commands
      }
    }
    pp.SetSpeed(0,0);                       //stop robot
    if(robot.Read()){cerr <<endl <<"HELP"; exit(1);}
    if(mcomPlayer.Read()){cerr <<endl <<"HELP"; exit(1);}
    char channel_6665[MCOM_CHANNEL_LEN] = "6665";
    if(strncmp(MCOM_CHANNEL_ROBOTS[0],channel_6665,4) == 0) //make sure
that this is robot 6665's channel, and get his data
      robot_positions[0]                                                =
receive_sonars(robot,mcomPlayer,mcp,MCOM_CHANNEL_ROBOTS[0],false);
    else
      cerr <<endl <<"ERROR: Robot " <<port <<" did not find channel of
initilization robot-first map will NOT be accurate";
    map_building(robot_positions[0],0);   //build Global Map out of
Robot 6665's sonar data only
  }
```

```
  else                              //if  neither  of  the  first  two
statements are true, there's only one robot
  {
    pp.SetSpeed(0,0);               //make sure robot is not moving
    if(robot.Read()){cerr <<endl <<"HELP"; exit(1);}
    if(mcomPlayer.Read()){cerr <<endl <<"HELP"; exit(1);}
    for(ii=0;ii<NUM_OF_SONARS;ii++) //get sonar data
      sonar_readings[ii]=sp.ranges[ii];
    robot_positions[ME].x=pp.xpos;        //get robot's own location
    robot_positions[ME].y=pp.ypos;
    robot_positions[ME].theta=pp.theta;
    map_building(robot_positions[ME],ME); //build Global map
  }
  //Now,  if  there  are  robots  in  front,  try  to  wait  on  them  to  move
first, this helps in initial dispersion
  wait = true;                      //FLAG  that  there  are  other
robots in front, so wait on them to move
  loop_count = 0;                   //but  don't  wait  too  long,
keep up with how long robot has waited
  pp.SetSpeed(0,0);                 //stop all robots
  while(wait && loop_count < 20)    //wait  for  up  to  2  seconds
for other robots to move
  {
    wait = false;
    if(robot.Read()){cerr <<endl <<"HELP"; exit(1);}
    if(mcomPlayer.Read()){cerr <<endl <<"HELP"; exit(1);}
    for (ii = 0;ii<bfp.blob_count;ii++)   //check all blobs (robots) to
see if I need to wait
    {
      if (bfp.blobs[ii].area > 300)       //if blob is too close,
        wait = true;                      //keep waiting
    }
    loop_count++;
  }

  //---------BEGIN SEARCH-----------------
  for(;;)
  {
    //temporarily log data
    if(port == 6665)
    {
      cerr <<endl <<"Still " <<NUM_OF_ROBOTS <<" robots searching";
      char temp_for_logging_map[30] = "/home/globalmap";
//      char temp_for_logging_cost[30] = "/home/costgrid";
      char temp_port[6];
      sprintf(temp_port, "%d", MCOM_TYPE_SONAR_READINGS);
      strcat(temp_for_logging_map,temp_port);
//      strcat(temp_for_logging_cost,temp_port);
      log_information(1,temp_for_logging_map);
//      log_information(2,temp_for_logging_cost);
    }
    path_length  =  path_planning(robot,mcomPlayer,target_homing);//plan
a path to the next frontier cell
```

```
target_seen=path_following(robot,mcomPlayer,path_length,bfp,pp,sp);//fo
llow that path
    pp.SetSpeed(0,0);
    //allow three data cycles to make sure robot is stopped before
continuing with map_building, etc.
    for(ii=0;ii<3;ii++)
    {
      if(robot.Read()){cerr <<endl <<"HELP"; exit(1);}
      if(mcomPlayer.Read()){cerr <<endl <<"HELP"; exit(1);}
    }
    for(ii=0;ii<NUM_OF_SONARS;ii++)       //get sonar data
      sonar_readings[ii]=sp.ranges[ii];
    robot_positions[ME].x=pp.xpos;        //get robot's own location
    robot_positions[ME].y=pp.ypos;
    robot_positions[ME].theta=pp.theta;
    //if robot is stuck, try to go forward or backward
    if(robot_positions[ME].x       ==       my_last_position.x       &&
robot_positions[ME].y == my_last_position.y)
    {
      if(Stuck == true)
      {
        cerr <<endl <<"Robot " <<port <<" trying to get unstuck with
speed: " <<powf(-1.0,(float)(MCOM_TYPE_SONAR_READINGS))*2;
        pp.SetSpeed(powf(-1.0,(float)(MCOM_TYPE_SONAR_READINGS))*2,0);
      }
      else
      {
        Stuck = true;
  //move in opposite direction of obstruction
        if((sp[3] + sp[4] + sp[5]) > (sp[11] + sp[12] + sp[13]))
          pp.SetSpeed(2,0);         //go forward one meter
        else
          pp.SetSpeed(-2,0);        //go back one meter
      }
      for(jj=0;jj<5;jj++)           //allow time to move one meter
      {
        if(robot.Read()){cerr <<endl <<"HELP"; exit(1);}
        if(mcomPlayer.Read()){cerr <<endl <<"HELP"; exit(1);}
      }
      pp.SetSpeed(0,0);             //stop robot
      if(robot.Read()){cerr <<endl <<"HELP"; exit(1);}
      if(mcomPlayer.Read()){cerr <<endl <<"HELP"; exit(1);}
      cerr <<endl <<"Robot " <<port <<" was stuck!";
    }
    else
      Stuck = false;
    my_last_position.x=pp.xpos;my_last_position.y=pp.ypos;  //to  detect
when robot is stuck
    MCOM_TYPE_SONAR_READINGS++;     //increment  this  mcom  message  type
so robots can know if data is new or old
    if(NUM_OF_ROBOTS > 1 && !target_homing)
      send_sonars(robot,mcomPlayer,mcp,robot_positions[ME],false);
    map_building(robot_positions[ME],ME); //use  robot's  own  data  to
build Global map
```

```
    if(target_seen)                 //if   target   was   seen   while
path_following, try to home in on it
      target_found                                                    =
camera_homing(robot,mcomPlayer,bfp,pp,sp,target_homing);
    if(target_found)                //if  able  to  home  in  on  target,
send on TARGET_FOUND channel
    {
      search_and_rescue_total_time  =  robot.timestamp.tv_sec;//keep  up
with total S&R time
      if(target_homing)             //if  not  the  first  robot  to  find
it, then simply shut robot down
        robot_shutdown();           //end search-and-rescue task
      pp.SetSpeed(0,0);
      if(robot.Read()){cerr <<endl <<"HELP"; exit(1);}
      if(mcomPlayer.Read()){cerr <<endl <<"HELP"; exit(1);}
      for(ii=0;ii<NUM_OF_SONARS;ii++)//get sonar data
        sonar_readings[ii]=sp.ranges[ii];
      robot_positions[ME].x=pp.xpos;//get robot's own location
      robot_positions[ME].y=pp.ypos;
      robot_positions[ME].theta=pp.theta;
      cerr  <<endl  <<"Target  is  at  ["  <<robot_positions[ME].x  <<","
<<robot_positions[ME].y <<"]";
      MCOM_TYPE_SONAR_READINGS++;   //increment  this  mcom  message  type
so robots can know if data is new or old
      if(NUM_OF_ROBOTS > 1)
      {
        //send sonar data and target's location
        send_sonars(robot,mcomPlayer,mcp,robot_positions[ME],true);
        //also send on MY channel, so robots won't delete ME
        send_sonars(robot,mcomPlayer,mcp,robot_positions[ME],false);
        MCOM_TYPE_SONAR_READINGS++; //send  on  next  loop  too,  just  to
make sure robots won't delete ME
        send_sonars(robot,mcomPlayer,mcp,robot_positions[ME],false);
      }
      map_building(robot_positions[ME],ME);//use  robot's  own  data  to
build Global map
      robot_shutdown();             //end search-and-rescue task
    }
    if(target_seen  &&  !target_found)//if  we  spent  time  homing  in  on
target, but did not find the target, we need to update MY positions so
path_planning will be accurate
    {
      robot_positions[ME].x=pp.xpos;//get robot's own location
      robot_positions[ME].y=pp.ypos;
      robot_positions[ME].theta=pp.theta;
      my_last_position.x=pp.xpos;my_last_position.y=pp.ypos;//to  detect
when robot is stuck
    }
    //check  to  see  if  any  other  robots  found  target  by  checking  the
TARGET_FOUND channel
    if(!target_homing)              //only  communicate  during  search
state, not during rescue (homing) state
    {
      if(NUM_OF_ROBOTS > 1)
      {
```

```
        target_location                                                 =
receive_sonars(robot,mcomPlayer,mcp,MCOM_CHANNEL_TARGET_FOUND,true);
//check TARGET_FOUND channel
        if(target_location.x != 9999)     //if   something   other   than
9999 is returned, target was found
        {
          target_homing = true;           //from   here   on   out,   robot
will be homing in on target instead of searching
          map_building(target_location,robot_that_found_target);//build
map from robot that found target
          cerr  <<endl  <<"Robot  "  <<port  <<"  says  that  robot  #  "
<<robot_that_found_target  <<"  found  the  target.    I'm  going  to  ["
<<target_location.x <<"," <<target_location.y <<"] NOW!!!!";
        }
      }
    }
    if(!target_homing)
    {
      for(ii=0;ii<NUM_OF_ROBOTS;ii++)
      {
        if(ii != ME)                      //ignore my channel
        {
          robot_positions[ii]                                           =
receive_sonars(robot,mcomPlayer,mcp,MCOM_CHANNEL_ROBOTS[ii],false);//ge
t robot's sonar data
          if(robot_positions[ii].theta   ==   9999)//this   FLAGS   that
something went wrong
          {
            if(robot_positions[ii].x  ==  9999)//if  x-FLAG  is  9999,  we
had to delete a channel
            {
              ii--;                       //the   next   robot   will   now
take the place of the deleted robot
              //get   current   robot's   location   again--receive_sonars
occasionally puts [9999,9999] into MY position
              robot_positions[ME].x=pp.xpos;//get robot's own location
              robot_positions[ME].y=pp.ypos;
              robot_positions[ME].theta=pp.theta;
            }
            else                    //else, target was found!!!
            {
              target_location = robot_positions[ii];  //store        the
target's location returned from receive_sonars
              target_location.theta = 0;  //target's   theta   is   never
used, but set to zero in case (not 9999)
              target_homing = true;       //from   here   on   out,   robot
will be homing in on target instead of searching
              cerr <<endl <<"Robot " <<port <<" says that robot # "
<<robot_that_found_target  <<"  found  the  target.    I'm  going  to  ["
<<target_location.x <<"," <<target_location.y <<"] NOW!!!!";
              break;                      //get out of this for loop!
            }
          }
          else
```

```
            map_building(robot_positions[ii],ii);     //use      robot's
data to build Global map
          }
        }
      }
  }
  return 0; //code will never make it here, but put a return 0 anyway:)
}



//-------------------------------------------------------------------------
//-------------------------------BEHAVIORS---------------------------------
//-------------------------------------------------------------------------
//-----------------------------Obstacle Avoidance--------------------
/*This Behavior checks to see if there is an obstacle in front of the
robot and decides which way to turn to avoid it
--It takes in the SonarProxy and robot client and returns the commands
to send to the PositionProxy (speed,turnrate)
--These Global Variables must be updated prior to calling: SPEED,
MINFRONTDISTANCE
--These Global Variables may be changed during this program:
*/

command obstacle_avoidance(SonarProxy &sp)
{
  //variables
  float fullspeed=1;
  float slowspeed=0.1;              //a speed for avoiding obstacles
  float min_side_dist = MINFRONTDISTANCE;//minimum side distance
  command commands_to_send = {fullspeed,0};//to hold commands to send

  //if object in front, slow down and turn away
  if((sp[0] < min_side_dist) ||     //sonars 2-5 are front four sensors
     (sp[1] < MINFRONTDISTANCE) ||
     (sp[2] < MINFRONTDISTANCE) ||
     (sp[3] < MINFRONTDISTANCE) ||
     (sp[5] < MINFRONTDISTANCE) ||
     (sp[6] < MINFRONTDISTANCE) ||
     (sp[7] < MINFRONTDISTANCE) ||
     (sp[8] < min_side_dist))
  {
    commands_to_send.speed = slowspeed;
    if((sp[0] + sp[1] + sp[2] + sp[3]) < (sp[5] + sp[6] + sp[7] +
sp[8]))
        commands_to_send.turnrate = DTOR(-500); //turn right
    else
        commands_to_send.turnrate = DTOR(500);  //turn left
  }
  return commands_to_send;
}

//-----------------------------Send Sonars---------------------------
/*This behavior sends data via MCom interface.  It takes in the two
Player Clients, MComProxy, the location of the robot, and a flag for
```

```
sending on the target channel.   It returns nothing; only Global
Variables are updated
--These Global Variables must be updated prior to calling:
MCOM_TYPE_SONAR_READINGS, NUM_OF_SONARS, sonar_readings,
--These Global Variables may be changed during this program:
MCOM_TYPE_SONAR_READINGS,          message_packet,          Visibility,
robot_that_found_target
--It sends messages in this format: {x-location,y-location,theta-
location,sonar_reading1,sonar_reading2,...,sonar_reading16}
--Each parameter is converted to a string of length 6 (total message
length = 114) to send--this is a limitation of the MComProxy
--See function ftoa for protocol for converting to strings
*/

void   send_sonars(PlayerClient   &robot,   PlayerClient   &mcomPlayer,
MComProxy &mcp, odometry Location, bool target_channel)
{
  timeval start_time=robot.timestamp;//get timestamp to keep up with
total communication time
  //variables
  int ii;                     //used in for loops
  bool pushed=false;          //"pushed" FLAG
  int loop_count=0;           //to keep up with number of iterations
  char data[MCOM_DATA_LEN];   //to store data to send
  char channel[MCOM_CHANNEL_LEN];//to store channel to send data on
  int data_type;              //to store data type
  int index;                  //used with Visibility global variable

  //prepare data to send
  memset(data,'\0',MCOM_DATA_LEN);  //clear data string
  ftoa(2,Location.x);               //get data in correct format
  strncpy(data,message_packet,6);
  ftoa(2,Location.y);
  strncpy(data+6,message_packet,6);
  ftoa(1,Location.theta);
  strncpy(data+12,message_packet,6);
  for(ii=0;ii<NUM_OF_SONARS;ii++)
  {
    index = (int)rint(10 * sonar_readings[ii]);
    Visibility[index]++;            //keep up with how many of each
reading is received (used in path_planning)
    ftoa(1,sonar_readings[ii]);
    strncpy(data+18+6*ii,message_packet,6);
  }
  if(target_channel)                //if target_channel FLAG is set
  {
    strcpy(channel,MCOM_CHANNEL_TARGET_FOUND);//write   to   TARGET_FOUND
channel
    data_type = MCOM_TYPE_TARGET_FOUND;//with TARGET_FOUND data type
    strncpy(data+114,MCOM_CHANNEL_ROBOTS[ME],MCOM_CHANNEL_LEN);//also,
add robot's port number to end of message
    robot_that_found_target = ME;   //set   myself   as   the   robot   that
found the target
  }
  else
```

```
  {
    strcpy(channel,MCOM_CHANNEL_ROBOTS[ME]);    //otherwise,  robot  is
writing to his own channel
    data_type = MCOM_TYPE_SONAR_READINGS;       //with     data     type
SONAR_READINGS
  }
  //Attempt to send for 1 second (10 tries)
  while(!pushed && loop_count < 10)  {
    if(robot.Read()){cerr <<endl <<"HELP"; exit(1);}
    if(mcomPlayer.Read()){cerr <<endl <<"HELP"; exit(1);}
    if(mcp.Push(data_type,channel,data) == 0)   //try  to  push  info  on
robot's own channel
      pushed = true;          //if no error, set push FLAG
    else                      //Report errors and update variables
      cerr <<endl <<"ERROR: Robot " <<port <<" couldn't send data...try
" <<loop_count <<" out of 9";
    loop_count++;
  }
  if(!pushed)
    cerr <<endl <<"ERROR: Robot " <<port <<" could not send sonar data
on type: " <<data_type;
  communication_time     +=     ((float)(robot.timestamp.tv_sec)     +
(float)(robot.timestamp.tv_usec)/1000000)  -  ((float)(start_time.tv_sec)
+ (float)(start_time.tv_usec)/1000000);         //keep  up  with  total
communication time (may be < 1 second)
  return;
}


//-----------------------------Receive Sonars-------------------------
/*This behavior receives data via MCom interface.  It takes in the two
player clients, MComProxy, the channel to receive data on, and a flag
for checking the target channel.  It returns the odometry received: {x-
location,y-location,theta-location}
--These   Global   Variables   must   be   updated   prior   to   calling:
MCOM_TYPE_SONAR_READINGS,         NUM_OF_SONARS,         NUM_OF_ROBOTS,
MCOM_CHANNEL_ROBOTS
--These   Global   Variables   may   be   changed   during   this   program:
NUM_OF_ROBOTS,   MCOM_CHANNEL_ROBOTS,   sonar_readings,   message_packet,
Visibility, robot_that_found_target
--See function inv_ftoa for protocol for converting from strings back
to float variables
*/

odometry receive_sonars(PlayerClient &robot, PlayerClient &mcomPlayer,
MComProxy &mcp, char channel[MCOM_CHANNEL_LEN], bool target_channel)
{
  timeval start_time=robot.timestamp;     //get  timestamp  to  keep  up
with total communication time
  //variables
  int ii,jj;                  //used in for loops
  bool channel_read=false;    //"pushed" FLAG
  int impatience=0;           //to keep up with number of iterations
  char *ptr;                  //pointer to the data last read
  int data_type;              //to store data type
```

```
  odometry Location;          //to store the robot's
  odometry  the_target_location;//used  to  check  the  target  channel
before deleting the robot
  int index;                  //used with Visibility global variable

  if(target_channel)          //if   target_channel   FLAG   set,   we're
checking the TARGET_FOUND channel
  {
    data_type = MCOM_TYPE_TARGET_FOUND;   //so set to this type of data
    if(mcp.Read(data_type,channel) == 0)  //if data there
    {
      if(robot.Read()){cerr <<endl <<"HELP"; exit(1);}
      if(mcomPlayer.Read()){cerr <<endl <<"HELP"; exit(1);}
      if(mcp.Read(data_type,channel) == 0)//make sure data is there
        channel_read = true;  //if  data  is  read,  set  channel_read  FLAG
in order to process information
    }
    if(!channel_read)
    {
      Location.x = 9999;      //else,  return  a  location  that  will  not
affect the Global Map
      Location.y = 9999;
      Location.theta = 9999;
      communication_time    +=    ((float)(robot.timestamp.tv_sec)    +
(float)(robot.timestamp.tv_usec)/1000000)  -  ((float)(start_time.tv_sec)
+ (float)(start_time.tv_usec)/1000000);   //keep    up    with    total
communication time (may be < 1 second)
      return Location;        //return,  and  do  not  proceed  if  nothing
read on this channel
    }
  }
  else
    data_type = MCOM_TYPE_SONAR_READINGS; //if  not  "target_channel"  get
ready to read a SONAR_READINGS type of data
  //Attempt to read data for about 10 seconds
  while(!channel_read && impatience < 14) //try  to  read  for  about  40
seconds until giving up
  {
    if(robot.Read()){cerr <<endl <<"HELP"; exit(1);}
    if(mcomPlayer.Read()){cerr <<endl <<"HELP"; exit(1);}
    if(mcp.Read(data_type,channel) == 0)
      channel_read = true;    //if no error, set channel_read FLAG
    else
    {
      wait_count++;   //keep up with number of times robot had to wait
      for(ii=0;ii<30;ii++)    //we  don't  want  robots  to  continuously
try to read...so wait three seconds
      {
        if(robot.Read()){cerr <<endl <<"HELP"; exit(1);}
        if(mcomPlayer.Read()){cerr <<endl <<"HELP"; exit(1);}
      }
    }
    impatience++;
  }
  //If able to read, process data
```

```
  if(channel_read)
  {
    if(robot.Read()){cerr <<endl <<"HELP"; exit(1);}
    if(mcomPlayer.Read()){cerr <<endl <<"HELP"; exit(1);}
    ptr = mcp.LastData();                  //point to last data read
    strncpy(message_packet,ptr,6);         //store data
    Location.x=inv_ftoa(2);
    strncpy(message_packet,ptr+6,6);
    Location.y=inv_ftoa(2);
    strncpy(message_packet,ptr+12,6);
    Location.theta=inv_ftoa(1);
    for(ii=0;ii<NUM_OF_SONARS;ii++)
    {
      strncpy(message_packet,ptr+18+6*ii,6);
      sonar_readings[ii] = inv_ftoa(1);
      index = (int)rint(10 * sonar_readings[ii]);
      Visibility[index]++;                 //keep up with how many of
each reading is received (used in path_planning)
    }
    if(target_channel)
    {
      strncpy(message_packet,ptr+114,MCOM_CHANNEL_LEN);    //temporarily
store robot's port number in message packet
      for(ii=0;ii<NUM_OF_ROBOTS;ii++)     //compare with all robots to
see which one found the target
      {
        if(strncmp(message_packet,MCOM_CHANNEL_ROBOTS[ii],
MCOM_CHANNEL_LEN) == 0)
          robot_that_found_target = ii;   //store the number of the
robot that found the target
      }
    }
  }
  //if could not read waiting, give up and delete this channel--it can
no longer be trusted
  else
  {
    for(ii=0;ii<3;ii++)       //But first, try to read target channel
three times to make sure target has not been found
    {
      the_target_location     =     receive_sonars(robot,mcomPlayer,mcp,
MCOM_CHANNEL_TARGET_FOUND,true);    //check TARGET_FOUND channel
      if(the_target_location.x != 9999)        //if something other
than 9999 is returned, target was found
      {
        map_building(the_target_location,
robot_that_found_target);//build map from robot that found target
        the_target_location.theta = 9999; //return a bogus theta as a
FLAG (target's theta is not used in program)
        communication_time    +=    ((float)(robot.timestamp.tv_sec)    +
(float)(robot.timestamp.tv_usec)/1000000)  -  ((float)(start_time.tv_sec)
+ (float)(start_time.tv_usec)/1000000);        //keep up with total
communication time (may be < 1 second)
        return the_target_location;
      }
```

```
    }
    cerr <<endl <<"ERROR: Robot " <<port <<" could not read sonar data
on channel " <<channel <<" type " <<data_type <<"--DELETING THIS
CHANNEL";
    for(ii=0;ii<NUM_OF_ROBOTS;ii++)
    {
      if(strcmp(MCOM_CHANNEL_ROBOTS[ii],channel) == 0)      //find   the
robot's channel # in MCOM_CHANNEL_ROBOTS, it won't be ME
      {
        NUM_OF_ROBOTS--;      //decrement the number of robots
        for(jj=ii;jj<NUM_OF_ROBOTS;jj++)
        {
          strncpy(MCOM_CHANNEL_ROBOTS[jj],MCOM_CHANNEL_ROBOTS[jj+1],
MCOM_CHANNEL_LEN);//shift all channels, positions, and SHIFTS back one
          robot_positions[jj].x = robot_positions[jj+1].x;
          robot_positions[jj].y = robot_positions[jj+1].y;
          robot_positions[jj].theta = robot_positions[jj+1].theta;
          ROBOT_SHIFTS[jj].x = ROBOT_SHIFTS[jj+1].x;
          ROBOT_SHIFTS[jj].y = ROBOT_SHIFTS[jj+1].y;
          if((jj+1) == ME)    //if  I'm  one  of  the  ones  that  shifted
back, then decrement ME
            ME--;
        }
        Location.x = 9999;    //return a bogus location as a FLAG
        Location.y = 9999;
        Location.theta = 9999;
      }
    }
  }
  communication_time      +=      ((float)(robot.timestamp.tv_sec)      +
(float)(robot.timestamp.tv_usec)/1000000)  -  ((float)(start_time.tv_sec)
+ (float)(start_time.tv_usec)/1000000);   //keep    up    with    total
communication time (may be < 1 second)
  return Location;
}


//------------------------------Map Building--------------------------
/*This behavior uses an array of 16 sonar sensors to build a local grid
of the environment.  This local grid is integrated into the global grid
--It takes in a robot's odometry {x-location,y-location,theta-location}
(theta is in radians) and a robot_number.  This robot_number is the one
corresponding to the Global Variable MCOM_CHANNEL_ROBOTS and is used in
integrating the Local Grid into the Global Map.
--It returns nothing; only Global Variables are changed
--It builds a square map representing the probability of an obstacle
                  | 0    open space |      |0.4  probably open space|
Probability =     | 0.5  unknown    | =>   |0.5  unknown            |
                  | 1    occupied   |      |0.9  probably occupied  |
--A probability of 0 or 1 is never used since these are absolutes and
sensors are not perfect
--These Global Variables must be updated prior to calling: GLOBAL_MAXX,
GLOBAL_MAXY,      ROBOT_SHIFTS,      SIZE_OF_ROBOT,      GRID_CELL_SIZE,
NUM_OF_SONARS, SONAR_ANGLES, MAX_SENSING_DIST, sonar_readings
```

```
--These Global Variables may be changed during this program:
Global_Grid, temp_points
--Default GRID_CELL_SIZE = 30cm (roughly the size of a robot)
--The map represents approximately a 10.3m^2 environment (10 meters of
sonar sensing + 0.3 meter robot) at this default GRID_CELL_SIZE
--This local map is then integrated into the Global Map
--The map is oriented in the direction that the robot began, assuming
his original angular pose represents 0 degrees, and increases counter
clock-wise, and his original position was {0,0} (unless specified
otherwise by the -x and -y parameters passed to the program)
*/

void map_building(odometry robot_odometry, int robot_number)
{
  //variables used in building a Local Grid
    //define grid size-->we need space for sonar sensor, robot, then
sonar sensor--the "/2" and "*2+1" are to ensure the result is odd, to
make sure robot is in MIDDLE
  int             MAXX=(int)ceil(((2*MAX_SENSING_DIST+SIZE_OF_ROBOT)/2)/
GRID_CELL_SIZE - 1) * 2 + 1;
  int MAXY=MAXX;
  float Local_Grid[MAXX][MAXY];//Local Occupancy Grid
  int shift=(MAXX-1)/2;       //to    shift    the    robot's    sensors'
origination to the middle of the grid
  int ii,jj;                  //used in for loops
  point a,b;                  //temporary storage for sonar readings
  point all_outer_points[8*MAXX];   //to store outer points of robot's
sensed environment
  point sonar_points[NUM_OF_SONARS];//to store the coordinates from the
sonar readings
  int num_points=0;           //to hold the number of points returned
from the get_points_between function
  cell_index outer_grid_indexes[2*MAXX];  //to  hold  outtermost  sonar
points (in grid cells, not meters)
  cell_index inner_grid_indexes[2*MAXX];  //to hold points inside sonar
points (in grid cells, not meters)
  int tot_num_points=0;       //to access values in "all_outer_points"
  int delta_x,delta_y;        //to access eight cells arround one cell
  bool proceed;               //used  to  omit  some  endpoints  (due  to
sonar uncertainty)
  float temp1,temp2;          //to temporarily hold sonar readings
  float distance;       //to store distance between two sonar points

  //variables used in Global Map integration
  float Pocc1,Pocc2,Pocc,odds1,odds2,odds;//for probability (see below)
  int robot_x,robot_y;  //to  store  robot's  location  in  grid  cell
coordinates
  int idx,idy;          //to  store  upper-left  corner  of  Local_Grid  in
global indexes

  //initialize
  for(jj=0;jj<MAXY;jj++)//initialize Local_Grid to all 0.5's
  {
    for(ii=0;ii<MAXX;ii++)
    {
```

```
      Local_Grid[ii][jj] = 0.5;
    }
  }
  //get [x,y] coordinates of local area from sonar sensors
  for(ii=0;ii<NUM_OF_SONARS;ii++)
  {
    sonar_points[ii].x= sonar_readings[ii] * cos(robot_odometry.theta +
SONAR_ANGLES[ii]);
    sonar_points[ii].y= sonar_readings[ii] * sin(robot_odometry.theta +
SONAR_ANGLES[ii]);
  }
  //Build map from sonar points.  This for loop builds a line around
the robot of his surroundings, assigning "OCCUPIED" where necessary
  for(ii=0;ii<NUM_OF_SONARS;ii++)
  {
    a=sonar_points[ii];       //a and b are easier to keep up with:)
    if(ii==(NUM_OF_SONARS-1)) //if this is the last sonar on the ring,
the next sonar is sonar 0
      b=sonar_points[0];
    else                      //else just get the next sonar
      b=sonar_points[ii+1];
    if(a.x > b.x)             //since slope formula does not give a
vector, we must keep up with direction
    {
      a=b;
      b=sonar_points[ii];
    }
    distance = sqrt( (b.x-a.x)*(b.x-a.x) + (b.y-a.y)*(b.y-a.y) );
    //call function to get all points between a and b...function
returns number of points and puts points in "temp_points"
    num_points = get_points_between(a,b,GRID_CELL_SIZE);
    temp1 = sonar_readings[ii];     //temporarily store sonar readings
    if(ii==(NUM_OF_SONARS-1))
      temp2 = sonar_readings[0];
    else
      temp2 = sonar_readings[ii+1];
    //transfer points (type: float) to grid indexes (type: int) and
save points for future use
    for(jj=0;jj<num_points;jj++)
    {
      all_outer_points[tot_num_points+jj].x = temp_points[jj].x;
      all_outer_points[tot_num_points+jj].y = temp_points[jj].y;
      //if sonar readings are not max (5 meters) or too spread out (> 1
meter--roughly size of doorway), then assign "OCCUPIED" to the
occupancy grid on that line, otherwise leave "UNKNOWN"
      if(temp1<5 && temp2<5 && distance<1)
      {
        outer_grid_indexes[jj].i   =   (int)truncf(temp_points[jj].x   /
GRID_CELL_SIZE);
        outer_grid_indexes[jj].j   =   (int)truncf(temp_points[jj].y   /
GRID_CELL_SIZE);
        Local_Grid[outer_grid_indexes[jj].i     +     shift][shift     -
outer_grid_indexes[jj].j] = OCCUPIED;
//In an effort to get rid of any stray cells inside walls, make a
probabilistic distribution of the sensed cell AND the one beyond it
```

```
        if(fabsf(outer_grid_indexes[jj].i)                              >
fabsf(outer_grid_indexes[jj].j))
          Local_Grid[outer_grid_indexes[jj].i+shift+(int)(copysignf(1,
outer_grid_indexes[jj].i))][shift-outer_grid_indexes[jj].j]=0.6;
        else
          Local_Grid[outer_grid_indexes[jj].i+shift][shift-
outer_grid_indexes[jj].j-(int)(copysignf(1,outer_grid_indexes[jj].j))]
= 0.6; //don't use 0.9 here, just anything small and more than 0.5
      }
    }
    tot_num_points = tot_num_points + num_points;
  }
  //We now have a polygon around the robot's environment...the points
are stored in "all_outer_points".  From here, we can shrink in toward
robot filling in each cell with "OPEN"
  //there is a large amount redundancy in this part of the code
  for(ii=0;ii<tot_num_points;ii++)
  {
    if(all_outer_points[ii].x < 0)
    {
      a=all_outer_points[ii];
      b.x=0; b.y=0;
    }
    else
    {
      a.x=0; a.y=0;
      b=all_outer_points[ii];
    }
    //call  function  to  get  all  points  between  a  and  b...function
returns number of points and puts points in "temp_points"
    num_points = get_points_between(a,b,GRID_CELL_SIZE);
    //transfer  points  (type:  float)  to  grid  indexes  (type:  int)  and
lable all grid cells "OPEN"
    for(jj=0;jj<num_points;jj++)
    {
      proceed = true;               //initialize to true
      inner_grid_indexes[jj].i    =    (int)truncf(temp_points[jj].x    /
GRID_CELL_SIZE);
      inner_grid_indexes[jj].j    =    (int)truncf(temp_points[jj].y    /
GRID_CELL_SIZE);
      if(jj == 0 || jj == (num_points-1))
        proceed = false;            //don't  assign  the  first  or  last
cell (due to sensor uncertainties)
      else  if(inner_grid_indexes[jj].i==inner_grid_indexes[jj-1].i  &&
inner_grid_indexes[jj].j==inner_grid_indexes[jj-1].j)
        proceed = false;            //don't  assign  any  cells  twice
(this may occur due to rounding)
      if(proceed)
      {
        if(Local_Grid[inner_grid_indexes[jj].i    +    shift][shift    -
inner_grid_indexes[jj].j] == 0.5)   //only change if not already
          Local_Grid[inner_grid_indexes[jj].i    +    shift][shift    -
inner_grid_indexes[jj].j] = OPEN;
      }
    }
```

```
    for(jj=num_points-4;jj<num_points;jj++)     //reset   any   of   the
outer ones to 0.5
    {
      if(inner_grid_indexes[jj].i==inner_grid_indexes[num_points-1].i)
       {
         if(inner_grid_indexes[jj].j==inner_grid_indexes[num_points-
1].j)
         {
           if(Local_Grid[inner_grid_indexes[jj].i   +   shift][shift   -
inner_grid_indexes[jj].j] == OPEN)
             Local_Grid[inner_grid_indexes[jj].i   +   shift][shift   -
inner_grid_indexes[jj].j] = 0.5;
         }
       }
    }
    //last but not least, label the cell the robot is currently in as
"OPEN"—still don't use "0" since robot's location is not perfect
    delta_x = 0; delta_y = 0;
    Local_Grid[shift+delta_x][shift+delta_y] = 0.1;
  }

  //integrate into Global Map
  robot_x      =      (int)rint((robot_odometry.x/GRID_CELL_SIZE)      +
ROBOT_SHIFTS[robot_number].x);
  robot_y      =      (int)rint((-robot_odometry.y/GRID_CELL_SIZE)      -
ROBOT_SHIFTS[robot_number].y);
  idx=robot_x-((MAXX-1)/2);         //upper-left  corner  of  Local_Grid
in global indexes
  idy=robot_y-((MAXY-1)/2);         //upper-left  corner  of  Local_Grid
in global indexes

  for(jj=0;jj<MAXY;jj++)
  {
    for(ii=0;ii<MAXX;ii++)
    {
      if((ii+idx)>=0  &&  (ii+idx)<GLOBAL_MAXX    &&  (jj+idy)>=0    &&
(jj+idy)<GLOBAL_MAXY)
      {
        Pocc1 = Local_Grid[ii][jj];
        Pocc2 = Global_Map[ii+idx][jj+idy];
        odds = (Pocc1 / (1 - Pocc1)) * (Pocc2 / (1 - Pocc2));
        if(isnan(odds) || isinf(odds))//don't allow infinity
          odds = 100000;
        Global_Map[ii+idx][jj+idy] = odds / (1 + odds);
      }
    }
  }
  return;
}


//---------------------------------Path Planning-----------------------
/*This behavior gets the next frontier cell for a robot to explore and
plans a path to this frontier cell
```

```
--It takes in the two Player Clients and a flag to path-plan to the target's location.
--It returns the length of the path to follow.  The path is stored in the Global Variable, the_path_to_follow
--These Global Variables must be updated prior to calling: GRID_CELL_SIZE, SEARCH_AREA_SIZE_IN_METERS, MAX_SENSING_DIST, NUM_OF_ROBOTS, Global_Map, GLOBAL_MAXX, GLOBAL_MAXY, Visibility, robot_positions, ROBOT_SHIFTS, MY_SHIFTS, ME
--These Global Variables may be changed during this program: the_path_to_follow
--This behavior first finds all of the frontier regions in the Global_Map (a frontier cell is an open cell next to an unexplored cell; a frontier region is defined as anywhere there is a meter's worth of frontier cells--roughly the size of a door)
--Next the behavior calculates a cost function for each robot based on the Global Map and the robot's position that represents the cost of migrating to any point; the cost of migrating to each stored frontier region is saved
--Care is taken so that MY (the robot currently running this code) Cost_Grid is the last one updated; this is used later for path-planning.
--Next, the robots are assigned frontier regions to go to based on minimizing a utility/cost function
--The cost and utility method is adapted from the following reference: W. Burgard et. al., "Collaborative Multi-Robot Exploration," in IEEE Int. Conf. on Robot. & Automat., 2000, pp.476-481.
--Finally, a path is planned for the current robot (ME) using the stored Cost_Grid and the robot's assigned frontier.  Path-planning is accomplished through steepest decent in cost from the goal to the origination (robot's current position)
*/

int path_planning(PlayerClient &robot, PlayerClient &mcomPlayer, bool go_to_target)
{
  if(robot.Read()){cerr <<endl <<"HELP"; exit(1);}
  if(mcomPlayer.Read()){cerr <<endl <<"HELP"; exit(1);}
  timeval start_time=robot.timestamp;     //get timestamp to keep up with total path_planning time

  //variables
  int ii,jj,kk,delta_x,delta_y,Delta_X,Delta_Y; //for accessing grids
  int loop_count=0;//keep up with number of loops until convergence
  cell_index robot_grid_location;   //to transfer a robot's position into grid indexes

  //variables used in Cost Formulas
  int current_robot;          //to keep up with each robot in the loop
  float distance,Pocc,Cost,convergence,temp_Cost,temp_convergence;
  float the_max_cost,max_cost[NUM_OF_ROBOTS];   //to keep up with a normalizing factor (we need 0<Cost<1)

  //Variables used in Frontier Calculations
  int unknown_count; //to check for unknown cells around an open cell
```

```
  int region_criteria = (int)rint(1/GRID_CELL_SIZE);  //a  region  =  1
meter of grid cells
  int  region_boundary  =  (int)floor(region_criteria/2);//define  delta
x,y boundary for searching for more frontier cells
  int num_in_region;          //to keep up with the number of frontier
cells in a possible region
  int num_frontier_cells=0;   //to  keep  up  with  the  total  number  of
final frontier cells that pass the region test
  int temp1,temp2;            //temporary   variables   used   to   make
calculations easier
  bool open,frontier_cell;    //to FLAG when a cell is an open cell or
a frontier cell
  cell_index frontier_cells[256];   //holds  final  frontier  cells  that
pass the region test--hopefully no more than 256:)
  //variables used in utility formula
  float total_sum=0;          //used in calculating Visibility
  float partial_sum=0;        //used in calculating Visibility
  int recent_assignment;      //to  keep  up  with  the  robot  #  that  is
assigned to a frontier cell in Utility loop
  cell_index  frontier_assignment[NUM_OF_ROBOTS];//to  store  a  frontier
cell assignment to a robot
  float Utility[256];         //holds Utility of all frontier cells
  float  Prob_of_coverage[(int)rint(10*MAX_SENSING_DIST)]; //Probability
that a robot can see a dist. from his cell
  float robot_frontier_costs[NUM_OF_ROBOTS][256];//holds costs for each
robot of reaching each frontier cell
  float maximum_value,temp_value;   //used to find the maximum value of
the utility function

  //variables used in path planning
  int delta_i,delta_j,tempi,tempj,dist;         //to access cells
  int length_of_path;               //variable    that    is    returned
representing the length of the Path_to_follow in cells
  cell_index Path_to_follow[512];   //to store the cell indexes of the
path to follow
  cell_index next_cell;             //temporarily  store  the  next  cell
to follow (potentially becomes part of the path)
  float least_cost;                 //stores the least cost for finding
the next cell to follow in the path

  //if  not  path_planning  to  target's  location,  find  all  frontier
regions, searching by rows first, then by columns
  if(!go_to_target)
  {
    for(jj=0;jj<GLOBAL_MAXY;jj++)
    {
      for(ii=0;ii<GLOBAL_MAXX;ii++)
      {
        if(Global_Map[ii][jj] < 0.5)//if open cell, explore further
        {
          frontier_cell = false;    //initialize to false
          unknown_count = 0;
          for(delta_y=-1;delta_y<=1;delta_y++)  //check  surrounding  8
cells to see if they are unknown
          {
```

```
            for(delta_x=-1;delta_x<=1;delta_x++)
            {
              if((ii+delta_x)>=0    &&    (ii+delta_x)<GLOBAL_MAXX    &&
(jj+delta_y)>=0 && (jj+delta_y)<GLOBAL_MAXY)
              {
                if(Global_Map[ii+delta_x][jj+delta_y]  ==  0.5)   //p.s.
don't worry about current cell, we know it's !=0.5
                  unknown_count++;
                if(unknown_count >= 2)    //this is to stop robot from
finding "frontier cells" beyond jagged walls
                  frontier_cell = true;   //we only need three of the 8
cells = unknown for current cell to be a frontier cell
              }
            }
          }
          if(frontier_cell)
          {
            for(kk=0;kk<num_frontier_cells;kk++)      //check      other
frontier cells to see if too close
            {
              temp1           =           (ii-frontier_cells[kk].i)*(ii-
frontier_cells[kk].i);              //get i distance
              temp2           =           (jj-frontier_cells[kk].j)*(jj-
frontier_cells[kk].j);              //get j distance
              if(((int)rint(sqrt(temp1        +        temp2)))       <=
(2*region_boundary))                //check distance formula
                frontier_cell = false;//if it is too close, forget it
            }
          }
          if(frontier_cell)         //if it is a frontier cell, see if
there are enough surrounding frontier cells to make a region
          {
            num_in_region = 0;      //initialize to zero
            for(Delta_X=-region_boundary;Delta_X                        <=
region_boundary;Delta_X++)          //check surrounding region
            {                       //region will be a 3x3 (11x11) area
for a GRID_CELL_SIZE of 30cm (10cm)
              for(Delta_Y=-region_boundary;Delta_Y                      <=
region_boundary;Delta_Y++)
              {
                if(Global_Map[ii+Delta_X][jj+Delta_Y] < 0.5)  //if cell
is open, explore further to see if it is a frontier
                {
                  frontier_cell = false;  //initialize to false
                  unknown_count = 0;      //initialize to zero
                  for(delta_y=-1;delta_y<=1;delta_y++)        //check
surrounding 8 cells to see if they are unknown
                  {
                    for(delta_x=-1;delta_x<=1;delta_x++)
                    {
                      temp1 = ii+Delta_X+delta_x;             //these
two variables just make the next if statement less complicated
                      temp2 = jj+Delta_Y+delta_y;
                      if(temp1>=0  &&  temp1<GLOBAL_MAXX  &&  temp2>=0  &&
temp2<GLOBAL_MAXY && Global_Map[temp1][temp2] == 0.5)
```

```
                        unknown_count++;
                      if(unknown_count >= 2)    //this is to stop robot
from finding "frontier cells" beyond jagged walls
                        frontier_cell = true;
                    }
                  }
                  if(frontier_cell)
                    num_in_region++;      //if it is a frontier cell,
increment the number of frontier cells in the region
                  if(num_in_region >= region_criteria)      //if      we
already have enough, quit looking
                    break;
                }
              }
              if(num_in_region >= region_criteria)          //if      we
already have enough, quit looking
                break;
            }
            if(num_in_region >= region_criteria)            //if    this
is a frontier region, add it to frontier_cells
            {
              //These next Read statements will rarely happen (i.e.
num_frontier_cells rarely reaches as high as 75); however, for an
unknown reason, the Player server will skip this section of code if a
robot.Read() statement is not somewhere in the code (maybe because it's
such a long loop??)
              if(num_frontier_cells == 75)
              {
                if(robot.Read()){cerr <<endl <<"HELP"; exit(1);}
                if(mcomPlayer.Read()){cerr <<endl <<"HELP"; exit(1);}
              }
              frontier_cells[num_frontier_cells].i = ii;
              frontier_cells[num_frontier_cells].j = jj;
              num_frontier_cells++;       //increment   the   number   of
frontier cells
              ii=ii+(2*region_boundary);  //no need to look within the
same region boundary
            }
          }
        }
      }
    }
    if(num_frontier_cells == 0)           //if   there   aren't   anymore
frontier cells to explore, search state will end
    {
      cerr <<endl <<"ERROR: Robot " <<port <<" did not find any more
frontier cells to explore!";
      target_location.x = 0;  //use rescue state to get robot back home
      target_location.y = 0;
      go_to_target = true;     //set path-planning flags to perform a
"rescue"
      if(MCOM_TYPE_SONAR_READINGS > 7)    //if   robot   is   not   still
dispersing, end search state -- if still dispersing, robot may simply
be surrounded by robots with no place to go, so ignore
      {
```

```
        cerr <<endl <<"Robot " <<port <<" giving up on search and going
back home";
        target_homing = true;
      }
    }
    if(num_frontier_cells > 256)   //we only allocated 256 spaces for
frontier cells, should be enough
      cerr <<endl <<"ERROR: More than 256 frontier cells!! Must change
size of several variables in path_planning function";
  }
  //calculate the cost of reaching all frontier cells for all robots.
The reason for decrementing in the next for loop instead of
incrementing is so that the current robot of "-1", which represents ME,
will be the last to change the Cost Grid.  This last Cost Grid will be
used for MY path planning after all other costs are calculated for all
other robots
 for(current_robot=(NUM_OF_ROBOTS-1);current_robot>=-1;current_robot--)
  {
    if(go_to_target)          //if we're going to target, no need to
calculate other robots' costs
      current_robot = -1;     //only calculate MINE
    if(current_robot != ME)   //otherwise, skip me and wait until the
end, when current_robot = -1
    {
      //initialize
      if(current_robot == -1) //this means calculate a robot's own cost
instead of others'
      {
        robot_grid_location.i     =     (int)rint((robot_positions[ME].x/
GRID_CELL_SIZE) + MY_SHIFT.x);
        robot_grid_location.j     =     (int)rint((-robot_positions[ME].y/
GRID_CELL_SIZE) - MY_SHIFT.y);
      }
      else                    //else, calculate the other robot's cost
      {
        robot_grid_location.i                                          =
(int)rint((robot_positions[current_robot].x/GRID_CELL_SIZE)+ROBOT_SHIFT
S[current_robot].x);
        robot_grid_location.j       =       (int)rint((-robot_positions
[current_robot].y/GRID_CELL_SIZE)-ROBOT_SHIFTS[current_robot].y);
      }
      for(jj=0;jj<GLOBAL_MAXY;jj++)//initialize Cost_Grid to 'infinity'
      {
        for(ii=0;ii<GLOBAL_MAXX;ii++)
        {
          Cost_Grid[ii][jj] = 9999; //"infinity"  =  more  than  longest
distance robot could travel
        }
      }
      //zero cost where robot is (i.e. zero cost to stand still)
      Cost_Grid[robot_grid_location.i][robot_grid_location.j] = 0;
      loop_count = 0;
      convergence = 100;      //holds  how  much  change  there  is  in  the
cost grid each iteration
      the_max_cost = 0;       //initialize to zero to find the max
```

```
      //Calculate Cost of going to each cell
      while(convergence > 0.0005)         //Each   cell   in   Grid   must
converge to some number
      {
        convergence = 0;                  //initialize   convergence   to
zero and find the maximum (see below)
        for(jj=0;jj<GLOBAL_MAXY;jj++)     //Compute Costs
        {
          for(ii=0;ii<GLOBAL_MAXX;ii++)
          {
            Cost = 4*SEARCH_AREA_SIZE_IN_METERS;      //initialize  Cost
to infinity and find the minimum (see below)
            for(delta_y=-1;delta_y<=1;delta_y++)
            {
              for(delta_x=-1;delta_x<=1;delta_x++)
              {
                if((ii+delta_x)>=0   &&   (ii+delta_x)<GLOBAL_MAXX     &&
(jj+delta_y)>=0  && (jj+delta_y)<GLOBAL_MAXY)
                {
                  distance  =  (1/GRID_CELL_SIZE)  *  (float)sqrt((delta_x
*delta_x) + (delta_y*delta_y));
                  Pocc = Global_Map[ii+delta_x][jj+delta_y];
                  temp_Cost    =    Cost_Grid[ii+delta_x][jj+delta_y]    +
(distance * Pocc);
                  if(temp_Cost < Cost)    //get  minimum  Cost  of  all  8
surrounding grid cells
                    Cost = temp_Cost;
                }
              }
            }
            temp_convergence = sqrt((Cost - Cost_Grid[ii][jj]) * (Cost
- Cost_Grid[ii][jj]));                    //get positive difference
            if(temp_convergence > convergence)  //keep  up  with  maximum
difference for convergence criteria
              convergence = temp_convergence;
            if(Cost > the_max_cost)       //keep  up  with  maximum  cost
for normalization factor
              the_max_cost = Cost;
            Cost_Grid[ii][jj] = Cost;//update cost of current grid cell
          }
        }
        loop_count++;
        if(loop_count>200)                //don't  let  it  take  too  long!
If too many loops, send error
        {
          cerr   <<endl   <<"Robot   "   <<port   <<"   surpassed   maximum
loop_count  in  Cost  Equation  for  robot  #  "  <<current_robot    <<".
Convergence = " <<convergence;
          convergence = 0;    //this will cause the while loop to end
        }
      }
      //store each robot's frontier costs
      if(current_robot == -1)
      {
        max_cost[ME] = the_max_cost;
```

```
        if(go_to_target)      //if going to target's location, skip all
the frontier stuff
          num_frontier_cells = 0;
        for(ii=0;ii<num_frontier_cells;ii++)
        {
          robot_frontier_costs[ME][ii]    =    Cost_Grid[frontier_cells
[ii].i][frontier_cells[ii].j];
        }
      }
      else
      {
        max_cost[current_robot] = the_max_cost;
        for(ii=0;ii<num_frontier_cells;ii++)
        {
          robot_frontier_costs[current_robot][ii]                    =
Cost_Grid[frontier_cells [ii].i][frontier_cells[ii].j];
        }
      }
    }
  }
  //Now that we have the frontier cells in frontier_cells and the cost
of reaching any of these frontier cells in robot_frontier_costs, we
need to assign robots to frontier cells based on Cost and Utility
  if(!go_to_target) //only assign frontier cells if not going to target
  {
    if(NUM_OF_ROBOTS > 1)     //Visibility only has effect when there
are multiple robots
    {
      for(jj=0;jj<=(int)rint(10*MAX_SENSING_DIST);jj++)
        total_sum += Visibility[jj];
      if(total_sum == 0)      //This will cause a divide-by-zero error
in next calculation; but it should NEVER happen
        cerr <<endl <<"ERROR: Robot " <<port <<"received an infinite
Prob_of_Coverage in Visibility Calculations";
      for(jj=(int)rint(10*MAX_SENSING_DIST);jj>=0;jj--)
      {
        partial_sum += Visibility[jj];
        Prob_of_coverage[jj] = partial_sum / total_sum;
      }
    }
    for(ii=0;ii<num_frontier_cells;ii++)  //initialize Utility of each
frontier cell to 1
      Utility[ii] = 1;
    for(ii=0;ii<NUM_OF_ROBOTS;ii++)       //initialize              all
frontier_assignments to -1
      frontier_assignment[ii].i = -1;     //this   will   serve   as   an
"unassigned" FLAG
    //assign each robot the best frontier cell based on maximizing
(utility-cost)
    for(loop_count=0;loop_count<NUM_OF_ROBOTS;loop_count++)
    {
      maximum_value = -10;          //initialize this to some low value
      recent_assignment = -1;       //this lets code know that no robot
has been assigned in this loop
      for(ii=0;ii<NUM_OF_ROBOTS;ii++)     //for every robot, ii. . .
```

```
      {
        if(frontier_assignment[ii].i == -1) //(only attempt to assign
this robot if not yet assigned)
        {
          for(jj=0;jj<num_frontier_cells;jj++)  //and     for     every
frontier cell, jj=[i,j]
          {
            temp_value  =  Utility[jj]  -  (robot_frontier_costs[ii][jj]
/max_cost[ii]);         //maximize this function for ii and jj
            if(temp_value > maximum_value)
            {
              maximum_value = temp_value;       //store    the    maximum
value of temp_value
              if(recent_assignment >=0)         //if    a    robot    has
already been assigned
                frontier_assignment[recent_assignment].i  =  -1;  //reset
unassigned FLAG for that robot
              recent_assignment = ii;           //store  the  robot  that
is currently being assigned to cell
              frontier_assignment[ii].i = frontier_cells[jj].i; //store
the frontier cell as that robot's assignment
              frontier_assignment[ii].j = frontier_cells[jj].j;
            }
          }
        }
      }
      //when a robot is assigned to a frontier cell, reduce the Utility
of all other cells in the Visibility Area
      for(jj=0;jj<num_frontier_cells;jj++)      //for   every   frontier
cell, jj=[x,y]
      {
        temp1      =      frontier_assignment[recent_assignment].i      -
frontier_cells[jj].i;   //find distance from assigned frontier cell
        temp2      =      frontier_assignment[recent_assignment].j      -
frontier_cells[jj].j;
        dist                                                            =
(int)rint(10*GRID_CELL_SIZE*(sqrt(temp1*temp1+temp2*temp2)));
        if(dist <= (int)rint(10*MAX_SENSING_DIST))    //if cells are in
Visibility area, Utility will be affected
          //recalculate Utility of this cell based on new info.
          Utility[jj] = Utility[jj] * (1-Prob_of_coverage[dist]);
      }
    }
  }

  //Now that we have our robot's assigned cell, use steepest decent to
plan a path from assigned frontier cell back to the robot's position.
The robot will follow this path backwards ("uphill")
  jj=512;                     //the length of Path_to_follow
  if(go_to_target)
  {
    frontier_assignment[ME].i       =       (int)rint((target_location.x/
GRID_CELL_SIZE) + MY_SHIFT.x);
    frontier_assignment[ME].j       =      (int)rint((-target_location.y/
GRID_CELL_SIZE) - MY_SHIFT.y);
```

```
  }
  Path_to_follow[jj-1].i = frontier_assignment[ME].i; //store         MY
assigned cell in the last Path_to_follow node
  Path_to_follow[jj-1].j = frontier_assignment[ME].j;
  for(ii=(jj-2);ii>=0;ii--)
  {
    least_cost = 9999;
    for(delta_j=-1;delta_j<=1;delta_j++)  //for the 8 surrounding cells
    {
      for(delta_i=-1;delta_i<=1;delta_i++)
      {
        if(delta_i != 0 || delta_j != 0)  //don't go to same cell again
        {
          tempi = Path_to_follow[ii+1].i + delta_i;
          tempj = Path_to_follow[ii+1].j + delta_j;
          if(tempi>=0    &&    tempi<GLOBAL_MAXX    &&    tempj>=0    &&
tempj<GLOBAL_MAXY)
          {
            if(Cost_Grid[tempi][tempj] < least_cost)  //find  cell  with
the least cost
            {
              least_cost = Cost_Grid[tempi][tempj];
              next_cell.i = tempi;                    //and store it as
the next cell to travel to
              next_cell.j = tempj;
            }
          }
        }
      }
    }
    if(next_cell.i   ==   robot_grid_location.i   &&   next_cell.j   ==
robot_grid_location.j)  //if back to robot location, we're done
      break;
    Path_to_follow[ii].i = next_cell.i;   //else,  store  the  next  cell
in the Path_to_follow and continue
    Path_to_follow[ii].j = next_cell.j;
  }
  length_of_path = jj-1-ii;
  if(length_of_path >= 512)               //send  error  if  path  is  too
long to fit in variable
    cerr <<endl <<"ERROR: Robot " <<port <<" is trying to follow a path
that is too long.  Must increase the size of 'Path_to_follow' because
its  current  length,  "  <<jj  <<",  is  too  short.  Path  length:  "
<<length_of_path  <<"  going  from  ["  <<robot_grid_location.i  <<","
<<robot_grid_location.j  <<"]  to  ["  <<Path_to_follow[511].i  <<","
<<Path_to_follow[511].j <<"]";
  for(ii=(512-length_of_path);ii<512;ii++)//change  from  grid  cells  to
meters and put in the_path_to_follow
  {
    the_path_to_follow[ii].x  =  (Path_to_follow[ii].i  -  MY_SHIFT.x)  *
GRID_CELL_SIZE;
    the_path_to_follow[ii].y  =  -(Path_to_follow[ii].j  +  MY_SHIFT.y)  *
GRID_CELL_SIZE;
  }
  //wait for new time data and keep up with time-per-behavior
```

```
  if(robot.Read()){cerr <<endl <<"HELP"; exit(1);}
  if(mcomPlayer.Read()){cerr <<endl <<"HELP"; exit(1);}
  path_planning_time     +=     ((float)(robot.timestamp.tv_sec)     +
(float)(robot.timestamp.tv_usec)/1000000)  -  ((float)(start_time.tv_sec)
+ (float)(start_time.tv_usec)/1000000);                  //keep up with
total path_planning time (may be < 1 second)
  return length_of_path;
}


//-------------------------------Homing-----------------------------
/*This behavior homes in on a point by sending commands to give to a
robot to direct him toward a goal location.  It is called multiple
times to follow a path with multiple nodes.    It returns the
PositioinProxy::SetSpeed commands that are to be sent to the robot.
--The speed command can be used directly or as a multiplier in a
function.  If direct, the speeds sent are 1m/s and 0.1m/s.  If as a
multiplier, the commands are full speed (*1) or 1/10th speed (*0.1)
--These Global Variables must be updated prior to calling:
--These Global Variables may be changed during this program:
--If the distance between the robot and goal are less than 60cm, the
command {0,0} is sent, meaning "stop"
--Else, the goal location is transformed into the robot's coordinate
system, and the angular heading from the robot to the goal is returned
as the angular speed command.
--The function finally checks to see if the goal is directly behind the
robot, in which case the command is to reduce speed and turn at 90
degrees per second
--If the goal is directly in front of the robot, the default command of
"full speed ahead" is sent
*/

command homing(odometry the_robot, point the_goal)
{
  //variables
  float fullspeed=1;                //multiplier for full speed
  float slowspeed=0.1;  //speed to turn around if node is behind robot
  float Transformx,Transformy;      //to hold transformed coordinates
  command command_to_send = {fullspeed,0};      //initialize command to
full speed ahead
  float Distance;       //to store distance from robot to goal location

  Distance  =  sqrt((the_goal.x-the_robot.x)*(the_goal.x-the_robot.x)  +
(the_goal.y-the_robot.y)*(the_goal.y-the_robot.y));
  if(Distance < 0.6)    //if  robot  is  less  than  60  centimeters  from
goal, robot can stop
  {
    command_to_send.speed = 0;      //send command to stop
    return command_to_send;
  }
  //otherwise, perform a transformation to get the goal's location into
the coordinate system of the robot
  the_robot.theta  =  DTOR(90)  -  the_robot.theta;  //get  robot's  angle
with respect to the y-axis
```

```
  Transformx = (the_goal.x - the_robot.x)*cos(the_robot.theta) -
(the_goal.y - the_robot.y)*sin(the_robot.theta);
  Transformy = (the_goal.x - the_robot.x)*sin(the_robot.theta) +
(the_goal.y - the_robot.y)*cos(the_robot.theta);
  command_to_send.turnrate = atan2(Transformy,Transformx) - DTOR(90);
  if(command_to_send.turnrate < DTOR(-179.9))    //if turnrate is too
negative, make it positive (this happens because of the -90 shift)
    command_to_send.turnrate = command_to_send.turnrate + DTOR(360);
  if(fabsf(command_to_send.turnrate) > DTOR(90))
    command_to_send.speed = slowspeed;
  return command_to_send;                       //otherwise, goal is
directly in front of robot, so full speed ahead!
}


//------------------------Path Following-------------------------------
/*This behavior follows the path in the variable "the_path_to_follow".
It does this by making multiple calls to the homing and obstacle
avoidance behaviors and summing their returns
--It takes in two clients, three proxies, and the path_length
representing how much of "the_path_to_follow" to use
--It returns a flag representing if the target was seen while
path_following.   Three things may cause this program to stop and
return.  1) If the target is seen, the robot is stopped, and "true" is
returned  2) If the robot successfully navigates its entire path, the
robot is stopped and "false" is returned  3) If the robot is not
successful in following the path, its acquiescence causes it to give up
after a certain amount of time; the robot is stopped and "false" is
returned
--These Global Variables must be updated prior to calling:
the_path_to_follow, path_length, GRID_CELL_SIZE, MY_SHIFT, SPEED,
MINFRONTDIST (for obstacle_avoidance), THE_TARGET_COLOR (for
check_for_target)
--These Global Variables may be changed during this program:
robot_positions[ME]
--These functions are called from this behavior: check_for_target,
homing, obstacle_avoidance
*/

bool path_following(PlayerClient &robot, PlayerClient &mcomPlayer, int
the_path_length, BlobfinderProxy &bfp, PositionProxy &pp, SonarProxy
&sp)
{
  timeval start_time=robot.timestamp;     //get timestamp to keep up
with total path_following time
  //variables
  int ii,jj;                              //to access for loops
  int per_node_acquiescence=0;            //variable that grows with
time and allows an unsuccessful robot to give up
  int overall_acquiescence=0;             //same as above
  bool flag_target_seen;                  //flag if target is seen
  cell_index my_position;                 //for use in Position_tracker
  command go_to_path,go_to_obs,commands_to_send;//to store position
commands to send to the robot
```

```
  for(ii=(512-the_path_length);ii<512;) //follow each node of the path
  {
    if(robot.Read()){cerr <<endl <<"HELP"; exit(1);}
    if(mcomPlayer.Read()){cerr <<endl <<"HELP"; exit(1);}
    flag_target_seen = check_for_target(bfp,pp);      //look for target
    if(flag_target_seen)
    {       //wait for data to make sure robot is stopped immediately
      if(robot.Read()){cerr <<endl <<"HELP"; exit(1);}
      if(mcomPlayer.Read()){cerr <<endl <<"HELP"; exit(1);}
      //keep up with total path_following time
      path_following_time+=robot.timestamp.tv_sec - start_time.tv_sec;
      return true;      //if seen, return TRUE
    }
    robot_positions[ME].x=pp.xpos;              //robot's own location
    robot_positions[ME].y=pp.ypos;
    robot_positions[ME].theta=pp.theta;
    my_position.i  =  (int)rint((robot_positions[ME].x/GRID_CELL_SIZE)  +
MY_SHIFT.x);            //change to grid cell coordinates
    my_position.j  =  (int)rint((-robot_positions[ME].y/GRID_CELL_SIZE)  -
MY_SHIFT.y);
    Position_tracker[my_position.i][my_position.j] = 1; //track robot's
position (only used for debugging/show)
    //get commands for homing and obstacle avoidance
    go_to_path = homing(robot_positions[ME],the_path_to_follow[ii]);
    go_to_obs = obstacle_avoidance(sp);
    if(go_to_path.speed == 0)                   //if  we  made  it  to  the
goal, increment ii (go to next node on path)
    {
      ii++;
      per_node_acquiescence = 0;//reset acquiescence for each node
      if(ii == 512)           //if last node on the path, stop
      {
        pp.SetSpeed(0,0);     //stop robot
        path_following_time       +=       robot.timestamp.tv_sec       -
start_time.tv_sec;            //keep up with total path_following time
        return false;         //return false
      }
    }
    else                      //Multiply  the  two  speeds  and  add  the
turnrates (multipliers may need to be adjusted)
    {
      commands_to_send.speed = go_to_path.speed * go_to_obs.speed;
      commands_to_send.turnrate      =      2*go_to_path.turnrate      +
go_to_obs.turnrate;
      pp.SetSpeed(commands_to_send.speed,commands_to_send.turnrate);
    }
    per_node_acquiescence++;  //increase  per  node  acquiescence  each
time we send a command
    if(per_node_acquiescence == 30) //if  robot  has  tried  a  single  node
unsuccessfully for 3 seconds, give up
    {
      pp.SetSpeed(0,0);       //stop robot, and return
      path_following_time       +=       robot.timestamp.tv_sec       -
start_time.tv_sec;            //keep up with total path_following time
      return false;           //return false
```

```
    }
    overall_acquiescence++;
    if(overall_acquiescence == 250) //give up after 25 seconds (robots
will delete my channel after 30 seconds of waiting)
    {
      cerr <<endl <<"Robot " <<port <<" gave up path following after 25
seconds; trying to get to node " <<ii;
      pp.SetSpeed(0,0);       //stop robot
      path_following_time       +=       robot.timestamp.tv_sec       -
start_time.tv_sec;            //keep up with total path_following time
      return false;           //return false
    }
  }
  //if code ever gets here, path length was zero, and something went
wrong.  So print a message and return.  Robot will not move anywhere
this loop of the code.
  cerr   <<endl   <<endl   <<"ERROR--robot   got   past   for   loop   in
path_following  behavior--path_length  must  have  been  zero"  <<endl
<<endl;
  return false;
}


//------------------------------Camera Homing-------------------------
/*This behavior uses the blob-finding camera to home in on the target
once it is found.  It takes in two clients, three proxies, and a flag
to let the behavior know if robot is already homing on the target, or
if it is the first to find the target—code will behave differently.
The behavior returns true or false if the robot does (does not) make it
to the target
--These  Global  Variables  must  be  updated  prior  to  calling:
THE_TARGET_COLOR
--These Global Variables may be changed during this program:
--Each time during homing, if the robot temporarily looses the target,
the robot will turn around counter-clock-wise one time looking for the
target before its acquiescence parameter causes the robot to give up
and returning “false”
--If the robot is searching, it will get as close to the target as
possible before returning “true”.  If the robot is rescuing, it will
get within a two meter radius of the target, surge forward as a simple
aggregation technique, and return “true”
*/

bool  camera_homing(PlayerClient  &robot,  PlayerClient  &mcomPlayer,
BlobfinderProxy  &bfp,  PositionProxy  &pp,  SonarProxy  &sp,  bool
go_to_target)
{
  timeval start_time=robot.timestamp; //get timestamp to keep up with
total target_homing time
  //variables
  int ii;                     //to access loops
  int acquiescence=0;         //a parameter that grows if robot cannot
find target, so robot will quit
  bool currently_homing;      //a  FLAG  to  let  robot  know  if  it  is
already homing or still searching for target
```

```
  float approach_speed=1.000; //speed to approach target
  float homing_turnrate;      //turnrate for approaching target
  float Distance;             //to store distance from robot to target
if not the first to find target
  command go_to_obs={0,0};  //commands received from obstacle avoidance
  while(acquiescence < 45)    //this will allow acquiescence to grow
for a little more than four seconds
  {
    if(robot.Read()){cerr <<endl <<"HELP"; exit(1);}
    if(mcomPlayer.Read()){cerr <<endl <<"HELP"; exit(1);}
    currently_homing = false;       //initialize to false
    for(ii=0;ii<bfp.blob_count;ii++)//find a blob of THE_TARET_COLOR
    {
      if(bfp.blobs[ii].color == THE_TARGET_COLOR)
      {
        if(bfp.blobs[ii].top == 0)  //if target is so close to blob, it
covers the camera, that's close enough
        {
          pp.SetSpeed(0,0);         //stop the robot and return true
          if(robot.Read()){cerr <<endl <<"HELP"; exit(1);}
          if(mcomPlayer.Read()){cerr <<endl <<"HELP"; exit(1);}
          cerr <<endl <<endl <<"Robot " <<port <<" FOUND THE TARGET!!!"
<<endl;
          camera_homing_time      +=      robot.timestamp.tv_sec      -
start_time.tv_sec;            //keep up with total camera_homing time
          return true;
        }
        if(go_to_target)//if already homing in on target (rescue state)
        {
          Distance         =         sqrt(         (target_location.x-
pp.xpos)*(target_location.x-pp.xpos)        +        (target_location.y-
pp.ypos)*(target_location.y-pp.ypos) );   //get distance to target
          if(Distance < 2)    //if less than 2 meters, close enough!
          {
            pp.SetSpeed(1,copysign(DTOR(15),go_to_obs.turnrate)); //go
forward 2 meters and to the side to allow room for more robots
            for(ii=0;ii<20;ii++)
            {
              if(robot.Read()){cerr <<endl <<"HELP"; exit(1);}
              if(mcomPlayer.Read()){cerr <<endl <<"HELP"; exit(1);}
            }
            pp.SetSpeed(0,0);       //stop the robot and return true
            if(robot.Read()){cerr <<endl <<"HELP"; exit(1);}
            if(mcomPlayer.Read()){cerr <<endl <<"HELP"; exit(1);}
            cerr  <<endl  <<endl  <<"Robot  "  <<port  <<"  FOUND  THE
TARGET!!!" <<endl;
            camera_homing_time     +=     robot.timestamp.tv_sec     -
start_time.tv_sec;            //keep up with total camer_homing time
            return true;
          }
        }
        currently_homing = true;    //set currently homing flag
        acquiescence=0; //if robot sees the target, reset acquiescence
        if(bfp.blobs[ii].left > 40) //if  blob  is  completely  in  right
half of camera,
```

```
          homing_turnrate = -DTOR(30);          //turn right
        else
          homing_turnrate = DTOR(30);           //turn left
        go_to_obs = obstacle_avoidance(sp);     //avoid obstacles also
        pp.SetSpeed(approach_speed*go_to_obs.speed,homing_turnrate     +
0.25*go_to_obs.turnrate);
      }
    }
    if(!currently_homing)     //if robot temporarily lost target, try
to find it, increasing acquiescence
    {
      acquiescence++;         //increment acquiescence
      pp.SetSpeed(0,DTOR(-copysign(90,go_to_obs.turnrate)));//turn at a
rate of 90 degrees per second (one revolution per 4 seconds) in the
opposite direction of last obstacle avoidance (which made us turn away
from target)
    }
  }
  pp.SetSpeed(0,0);           //if robot's acquiescence gets too high,
robot lost target--so stop robot and return false
  if(robot.Read()){cerr <<endl <<"HELP"; exit(1);}
  if(mcomPlayer.Read()){cerr <<endl <<"HELP"; exit(1);}
  cerr <<endl <<endl <<"Robot " <<port <<" LOST THE TARGET!!!" <<endl;
  //keep up with total path_homing time
  camera_homing_time += robot.timestamp.tv_sec - start_time.tv_sec;
  return false;
}



//------------------------------------------------------------------------
//-------------------------------FUNCTIONS--------------------------------
//------------------------------------------------------------------------
//---------------------------Robot Initialization-------------------------
/*This function attempts to communicate with all robots and creates
channels for all robots that communicate successfully.
--The number of robots cooperating is passed to the search program when
called.
--The robots should be on ports 6665,6666,6667...and so on.
--These Global Variables must be updated prior to calling:
MCOM_TYPE_INITIALIZATION, NUM_OF_ROBOTS, MY_SHIFT.x, MY_SHIFT.y
--These Global Variables may be changed during this program:
NUM_OF_ROBOTS, MCOM_CHANNEL_ROBOTS, ROBOT_SHIFTS
--The function takes in the two player client programs and the
MComProxy.  It passes back nothing, only changing Global Variables
--Each robot first pushes his own port and shift data onto his channel
and creates a channel for himself (a robot's channel is his port #)
--After this, the robot waits a length of time to allow other robots to
catch up, then tries to read all other channels that are supposed to
exist a maximum of 10 times (about 1 second) before giving up
--If a robot cannot access this channel after 10 tries for any reason,
the channel is not created and will not be used for the remainder of
the search
--Errors are sent each time a robot has trouble pushing his own data or
reading another channel that is supposed to exist
```

```
--If a robot finds no other robots, an error is send "I'm searching
ALONE" and no communication is performed throughout the remainder of
the search
*/

void     robot_initialization(PlayerClient     &robot,     PlayerClient
&mcomPlayer, MComProxy &mcp)
{
  //variables
  int ii;               //to use in for loops
  bool pushed=false;    //"pushed" FLAG
  bool channel_read;    //"channel_read" FLAG
  bool ignore;          //a FLAG to ignore a robot's own port
  int loop_count=1; //keep up with how many times a loop has iterated
  int robot_count=0;    //keep up with how many robots have been found
  char my_init_data[MCOM_DATA_LEN];       //to hold my data to send
  char my_channel[MCOM_CHANNEL_LEN];      //to hold my channel name
  char other_robots[MCOM_CHANNEL_LEN];    //to   hold   other   robots'
channel names
  char *ptr;                  //pointer to the data read last

  //initialize variables
  memset(my_init_data,'\0',MCOM_DATA_LEN);  //clear strings
  memset(other_robots,'\0',MCOM_CHANNEL_LEN);
  if(NUM_OF_ROBOTS > 6)       //more  robots  need  more  time  to  start—
many errors if robots start asynchronously
    pause_four_seconds(robot,mcomPlayer);
  sprintf(my_init_data, "%d", port);  //convert port # to a string
  strncpy(my_channel,my_init_data,MCOM_CHANNEL_LEN);//make   a   channel
for myself
  ftoa(2,MY_SHIFT.x);               //convert MY_SHIFT.x to a string
  strncpy(my_init_data+MCOM_CHANNEL_LEN,message_packet,6);//append    it
to the init data after the port #
  ftoa(2,MY_SHIFT.y);               //convert MY_SHIFT.y to a string
  strncpy(my_init_data+MCOM_CHANNEL_LEN+6,message_packet,6);//append  it
to the init data after MY_SHIFT.x
  MY_SHIFT.x = MY_SHIFT.x/GRID_CELL_SIZE; //convert  SHIFTS  from  meters
to grid cells
  MY_SHIFT.y = MY_SHIFT.y/GRID_CELL_SIZE;

  //attempt to push robot's own data on his own channel
  while(!pushed && loop_count < 11) //try to push 10 times
  {
    if(robot.Read()){cerr <<endl <<"HELP"; exit(1);}
    if(mcomPlayer.Read()){cerr <<endl <<"HELP"; exit(1);}
    if(mcp.Push(MCOM_TYPE_INITIALIZATION,my_channel,my_init_data) == 0)
      pushed = true;                //if no error, set push FLAG
    else                            //else, report error and try again
      cerr  <<endl  <<"ERROR  pushing  initialization  data  for  Robot:  "
<<port <<" Retry " <<loop_count <<" out of 10";
    loop_count++;
  }
  pause_four_seconds(robot,mcomPlayer);   //wait for other robots
  //check for other robots on other channels
```

```
  for(ii=6665;ii<=(6664+NUM_OF_ROBOTS);ii++)     //check channels 6665-
>(6665 + #_of_robots)
  {
    ignore = false;
    channel_read = false;
    loop_count = 1;
    sprintf(other_robots,"%d",ii);              //int to char
    if(strncmp(other_robots,my_channel,4) == 0) //if  this  is  robot's
own channel, then set ignore FLAG
    {
      ignore = true;
      ME = robot_count;
      strncpy(MCOM_CHANNEL_ROBOTS[robot_count],my_channel,
MCOM_CHANNEL_LEN);      //copy channel into MCOM_CHANNEL_ROBOTS
      ROBOT_SHIFTS[robot_count].x = MY_SHIFT.x; //copy MY_SHIFTS
      ROBOT_SHIFTS[robot_count].y = MY_SHIFT.y;
      robot_count++;    //increment the number of robots actually found
    }
    while(!ignore && !channel_read && loop_count < 11)      //try     to
read channel ten times
    {
      if(robot.Read()){cerr <<endl <<"HELP"; exit(1);}
      if(mcomPlayer.Read()){cerr <<endl <<"HELP"; exit(1);}
      if(mcp.Read(MCOM_TYPE_INITIALIZATION,other_robots) == 0)
        channel_read = true;  //if no error, set channel_read FLAG
      else                    //else, report error and try again
        cerr <<endl <<"ERROR reading channel: " <<ii <<" for Robot: "
<<port <<" Retry " <<loop_count <<" out of 10";
      loop_count++;
    }
    if(channel_read)          //process data
    {
      if(robot.Read()){cerr <<endl <<"HELP"; exit(1);}
      if(mcomPlayer.Read()){cerr <<endl <<"HELP"; exit(1);}
      ptr = mcp.LastData();   //point to last data read
      //copy channel into MCOM_CHANNEL_ROBOTS
      strncpy(MCOM_CHANNEL_ROBOTS[robot_count],ptr,MCOM_CHANNEL_LEN);
      strncpy(message_packet,ptr+MCOM_CHANNEL_LEN,6);
      //convert SHIFTS back to float variables and to grid cells
      ROBOT_SHIFTS[robot_count].x = inv_ftoa(2)/GRID_CELL_SIZE;
      strncpy(message_packet,ptr+MCOM_CHANNEL_LEN+6,6);
      ROBOT_SHIFTS[robot_count].y = inv_ftoa(2)/GRID_CELL_SIZE;
      robot_count++;    //increment the number of robots actually found
    }
  }
  //Update new information, report, and return
  NUM_OF_ROBOTS = robot_count;      //update  #_of_robots  =  number  of
robots found plus one for this robot
  cerr  <<endl  <<"Robot  "  <<port  <<"  found  "  <<NUM_OF_ROBOTS  <<"
robot(s).  They are: ";
  for(ii=0;ii<NUM_OF_ROBOTS;ii++)
    cerr <<MCOM_CHANNEL_ROBOTS[ii] <<" , ";
  cerr <<"and I'm robot # " <<ME;
  if(NUM_OF_ROBOTS == 1)
    cerr <<endl <<"ERROR: Robot " <<port <<" is searching ALONE";
```

```
  return;
}


//-----------------------------Check for Target-----------------------
/*This function checks for the target which is recognized by
"THE_TARGET_COLOR".  It is called ANY time the robot is moving
--It takes in the BlobFinderProxy and returns true/false if it does
(does not) see the target
--These Global Variables must be updated prior to calling:
THE_TARGET_COLOR
--These Global Variables may be changed during this program:
--By default, THE_TARGET_COLOR is red
*/

bool check_for_target(BlobfinderProxy &bfp, PositionProxy &pp)
{
  int ii;
  num_blobs_seen += bfp.blob_count;
  for (ii = 0;ii<bfp.blob_count;ii++)     //check    all    blobs    for
THE_TARGET_COLOR
  {
    if (bfp.blobs[ii].color == THE_TARGET_COLOR) //if target is seen
    {
      pp.SetSpeed(0,0);                   //stop the robot
      cerr  <<endl  <<endl  <<"Robot  "  <<port  <<"  SAW  THE  TARGET!!!"
<<endl;
      return true;                        //and return true
    }
  }
  return false;
}


//-------------------------------Get Points Between--------------------
/*This function calculates all points on a line between any two points.
It is used in the map_building behavior
--It takes in the two endpoints and the desired spacing between points
to be calculated and returns the number of points found on the line
--The calculated points are stored in the global variable temp_points
--These Global Variables must be updated prior to calling:
--These  Global  Variables  may  be  changed  during  this  program:
temp_points
*/

int get_points_between(point point1, point point2, float spacing)
{
  //variables
  int kk,the_num_points;      //to access loops; holds # of points
  float Distance,Slope;       //holds length and slope of the line

  spacing *= 0.75;            //this gives a little more resolution to
this process, so that no grid cells are skipped
  Distance = sqrt( (point2.x-point1.x)*(point2.x-point1.x) + (point2.y-
point1.y)*(point2.y-point1.y) );    //get distance between endpoints
```

```
  Slope = (point2.y - point1.y) / ((point2.x - point1.x) + 0.00001);
  //the 0.0000001 is to make sure not to divide by ZERO!!!
  if(Slope > 100000 || isinf(Slope) || isnan(Slope))
  {
    if(isinf(Slope) || isnan(Slope))//send error if divide by zero
      cerr <<endl <<"ERROR: Robot " <<port <<" received an infinite
Slope in get_points_between function";
    Slope = 100000;                //keep the slope*slope below from
being too large...this will not affect accuracy
  }
  the_num_points = (int)ceil(Distance/spacing) + 1;   //actual   number
of points, including point1 and point2
  temp_points[0] = point1;          //make point1 and point2 endpoints
of "temp_points"
  temp_points[the_num_points-1] = point2;
  //get outer points between two readings
  for(kk=1;kk<(the_num_points-1);kk++)
  {
    //formulas below get every point between point1 and point2 with a
distance of "spacing" apart; formulas are derived from distance and
slope formulas
    temp_points[kk].x = temp_points[kk-1].x + sqrt( (spacing*spacing) /
(Slope*Slope +1) );
    temp_points[kk].y = temp_points[kk-1].y + Slope*(temp_points[kk].x
- temp_points[kk-1].x);
  }
  return the_num_points;
}


//------------------------------Float to String-----------------------
/*This function converts floating point numbers to strings for
communicating data between robots
--it takes in a float number and an integer representing the number of
digits before the decimal
--It updates a global string in the following format (N=negative,
decimal is left out):
--if positive, one digit:   x.xxxxx -->xxxxxx
--if positive, two digits:  xx.xxxx -->xxxxxx
--if negative, one digit:   -x.xxxx -->Nxxxxx
--if negative, two digits:  -xx.xxx -->Nxxxxx
--These Global Variables must be updated prior to calling:
--These Global Variables may be changed during this program:
message_packet
--This function is used to send messages...mcom can only send strings
and we need to send float variables
--Since the decimal place is lost, care must be taken in the message
passing protocol to know where the decimal goes in each number on the
receiving end
*/

void ftoa(int num_digits, float float_number)
{
  int N=0;              //a negative FLAG
  int int_number;       //to store integer created from "float_number"
```

```
  float multiplier;     //a multiplier to change from float to int

  switch(num_digits)
  {
    case 1:
      multiplier = 1e5;
      break;
    case 2:
      multiplier = 1e4;
      break;
    default:
      cerr <<endl <<"ERROR: in ftoa, Robot " <<port <<": num_digits = "
<<num_digits <<" is not recognisable";
      return;
  }
  if(float_number < 0)
  {
    message_packet[0] = 'N';    //set negative FLAG
    N = 1;
    multiplier /= 10;           //change multiplier
    float_number = sqrt((float_number * float_number)); //make positive
  }
  int_number = (int)rint(float_number * multiplier);  //convert     from
float to int
  sprintf(message_packet+N, "%d", int_number);        //convert     from
int to string
  return;
}


//-----------------------------------String to Float-------------------
/*This function provides the inverse of the ftoa function.  It converts
strings back into floating point numbers
--it takes in an integer representing the number of digits before the
decimal and returns a float variable representing the message_packet
string
--These  Global  Variables  must  be  updated  prior  to  calling:
message_packet
--These Global Variables may be changed during this program:
--it is used when receiving messages from other robots using MCom Proxy
to get the information back into the correct format
*/

float inv_ftoa(int num_digits)
{
  int N=0;              //a negative FLAG
  int int_number;       //to store integer created from atoi()
  float float_number;   //float variable to return
  float multiplier;     //multiplier to change from int to float

  switch(num_digits)
  { case 1:
      multiplier = 1e-5;
      break;
    case 2:
```

```
      multiplier = 1e-4;
      break;
    default:
      cerr <<endl <<"ERROR: in ftoa, Robot " <<port <<": num_digits = "
<<num_digits <<" is not recognisable";
      return 0;
  }
  if(message_packet[0] == 'N')
  {
    N=1;                //set negative FLAG
    multiplier *= 10;   //change multiplier
  }
  int_number = atoi(message_packet + N);  //convert from string to int
  float_number = int_number * multiplier; //convert from int to float
  if(N == 1)
    float_number = -float_number;   //make negative if N FLAG is set
  return float_number;
}


//-----------------------------Robot Shutdown-------------------------
/*This program shuts down the robot at the end of the search-and-rescue
task.  Several files of final information are logged and the time it
took the robot to find the target is printed onscreen.
*/

void robot_shutdown()
{
  //variables
  char temp_for_logging_position_tracker[36] ="/home/position_tracker";
  char temp_for_logging_statistics[36] = "/home/statistics";
  char temp_for_logging_global_map[36] = "/home/globalmap.final";
  char temp_port[5];

  sprintf(temp_port, "%d",port); //prepare filenames for logging info.
  strcat(temp_for_logging_position_tracker,temp_port);
  strcat(temp_for_logging_statistics,temp_port);
  strcat(temp_for_logging_global_map,temp_port);
  log_information(0,temp_for_logging_statistics); //log final info.
  log_information(1,temp_for_logging_global_map);
  log_information(3,temp_for_logging_position_tracker);

  cerr <<endl <<"The search and rescue task took a total of "
<<search_and_rescue_total_time <<" seconds for Robot " <<port <<endl
<<"Robot " <<port <<" shutting down" <<endl <<endl;
  exit(0);                          //exit the program
}

//-----------------------------Log Information------------------------
/*This function is used for Sending information to the operator in a
file, which is used for debugging purposes
--The function takes in a log_type and a filename
--The types are:  type=0.........log Search-and-Rescue Statistics
                  type=1.........log a Global Map
                  type=2.........log a Cost Function
```

```
                type=3.........log the Position_tracker
*/

void log_information(int log_type,char filename[])
{
  int ii,jj;                    //for accessing loops
  int temp1,temp2;              //used in temporary calculations
  FILE *fp;
  fp = fopen(filename,"w");     //open file for writing

  switch(log_type)              //switch to the correct type of logging
  {
    case 0:                     //log search-and-rescue statistics
      fprintf(fp,"Search and Rescue Statistics for Robot #%d with %d
Robots\n\n",port,NUM_OF_ROBOTS);
      fprintf(fp,"Percent of time spent in communication behavior: %2f
%\n",(communication_time/(float)(search_and_rescue_total_time))*100);
      fprintf(fp,"     --Amount of time spent waiting on other robots:
%d seconds\n",(wait_count*3));
      fprintf(fp,"Percent of time spent in map building behavior:
Negligible\n");
      fprintf(fp,"Percent of time spent in path_planning behavior: %2f
%\n",(path_planning_time/(float)(search_and_rescue_total_time))*100);
      fprintf(fp,"Percent of time spent in path_following behavior: %2f
%\n",(float(path_following_time)/(float)(search_and_rescue_total_time))
*100);
      fprintf(fp,"Percent of time spent in target_homing behavior: %2f
%\n",(float(camera_homing_time)/(float)(search_and_rescue_total_time))*
100);
      fprintf(fp,"Number of robots seen per second: %2f robots (for
comparison
ONLY)\n",((float)(num_blobs_seen))/((float)(search_and_rescue_total_tim
e)));
      fprintf(fp,"Number of sense/map/communicate/move loops during
search state: %d\n\n",MCOM_TYPE_SONAR_READINGS);
      temp1 = (int)(floor((float)(search_and_rescue_total_time)/60));
      temp2 = search_and_rescue_total_time % 60;
      if(temp2 <10)
        fprintf(fp,"Total time spent for Search-and-Rescue task =
%d:0%d\n",temp1,temp2);
      else
        fprintf(fp,"Total time spent for Search-and-Rescue task =
%d:%d\n",temp1,temp2);
      if(fprintf(fp,"\n") < 0)      //just for error printing
      {
        cerr <<endl <<"ERROR logging " <<filename <<" for Robot "
<<port;
        break;
      }
      cerr <<endl <<"Robot " <<port <<" logged " <<filename <<"
SUCCESSFULLY!!";
      break;

    case 1:             //log the current global map
      if(fprintf(fp,"%d %d ",GLOBAL_MAXX,GLOBAL_MAXY) < 0)
```

```
      {
        cerr <<endl <<"ERROR logging " <<filename <<" for Robot "
<<port;
        break;
      }
      for(jj=0;jj<GLOBAL_MAXY;jj++)
      {
        for(ii=0;ii<GLOBAL_MAXX;ii++)
        {
          if(fprintf(fp,"%f ",Global_Map[ii][jj]) < 0)
          {
            cerr <<endl <<"ERROR logging " <<filename <<" for Robot "
<<port;
          break;
          }
        }
        fprintf(fp,"\n");
      }
      cerr <<endl <<"Robot " <<port <<" logged " <<filename <<"
SUCCESSFULLY!!";
      break;
    case 2:
      if(fprintf(fp,"%d %d ",GLOBAL_MAXX,GLOBAL_MAXY) < 0)
      {
        cerr <<endl <<"ERROR logging " <<filename <<" for Robot "
<<port;
        break;
      }
      for(jj=0;jj<GLOBAL_MAXY;jj++)
      {
        for(ii=0;ii<GLOBAL_MAXX;ii++)
        {
          if(fprintf(fp,"%f ",Cost_Grid[ii][jj]) < 0)
          {
            cerr <<endl <<"ERROR logging " <<filename <<" for Robot "
<<port;
            break;
          }
        }
        fprintf(fp,"\n");
      }
      cerr <<endl <<"Robot " <<port <<" logged " <<filename <<"
SUCCESSFULLY!!";
      break;

    case 3:                    //log the current Positioin_tracker
      if(fprintf(fp,"%d %d ",GLOBAL_MAXX,GLOBAL_MAXY) < 0)
      {
        cerr <<endl <<"ERROR logging " <<filename <<" for Robot "
<<port;
        break;
      }
      for(jj=0;jj<GLOBAL_MAXY;jj++)
      {
        for(ii=0;ii<GLOBAL_MAXX;ii++)
```

```
        {
          if(fprintf(fp,"%f ",Position_tracker[ii][jj]) < 0)
          {
            cerr <<endl <<"ERROR logging " <<filename <<" for Robot "
<<port;
            break;
          }
        }
        fprintf(fp,"\n");
      }
      cerr <<endl <<"Robot " <<port <<" logged " <<filename <<"
SUCCESSFULLY!!";
      break;

    default:
      cerr <<endl <<"Robot " <<port <<", ERROR logging data!!! "
<<log_type <<" is not a recognisable log type";
  }
  fclose(fp);
}


//--------------------------------Pause 4 seconds----------------------
/*This function pauses for 4 seconds.  It is used for robots to wait on
each other during initialization
--It is also useful in diagnosis and debugging to prevent from runaway
screen printing
--It takes in the two client programs running the robot and uses them
to wait on their data to come in at 10Hz by default
*/

void pause_four_seconds(PlayerClient &robot, PlayerClient &mcomPlayer)
{
  int ii;
  for(ii=0;ii<40;ii++)
  {
    //wait for new robot data (10Hz by default)
    if(robot.Read()){cerr <<endl <<"HELP"; exit(1);}
    if(mcomPlayer.Read()){cerr <<endl <<"HELP"; exit(1);}
  }
}


//--------------------------------Parse Arguments----------------------
/*This function interprets the arguments passed to this program when
"search" is called
--See USAGE defined at the beginning of this program
*/

void
parse_args(int argc, char** argv)
{
  int i=1;

  while(i<argc)
```

```
{
  if(!strcmp(argv[i],"-h"))
  {
    if(++i<argc)
      strcpy(host,argv[i]);
    else
    {
      puts(USAGE);
      exit(1);
    }
  }
  else if(!strcmp(argv[i],"-p"))
  {
    if(++i<argc)
      port = atoi(argv[i]);
    else
    {
      puts(USAGE);
      exit(1);
    }
  }
  else if(!strcmp(argv[i],"-N"))
  {
    if(++i<argc)
      NUM_OF_ROBOTS = atoi(argv[i]);
    else
    {
      puts(USAGE);
      exit(1);
    }
  }
  else if(!strcmp(argv[i], "-M"))
  {
    if(++i<argc)
      SEARCH_AREA_SIZE_IN_METERS = atof(argv[i]);
    else
    {
      puts(USAGE);
      exit(1);
    }
  }
  else if(!strcmp(argv[i], "-x"))
  {
    if(++i<argc)
      MY_SHIFT.x = atof(argv[i]);
    else
    {
      puts(USAGE);
      exit(1);
    }
  }
  else if(!strcmp(argv[i], "-y"))
  {
    if(++i<argc)
      MY_SHIFT.y = atof(argv[i]);
```

```
      else
      {
        puts(USAGE);
        exit(1);
      }
    }
    else if(!strcmp(argv[i], "-s"))
    {
      if(++i<argc)
        SPEED = atof(argv[i]);
      else
      {
        puts(USAGE);
        exit(1);
      }
    }
    else if(!strcmp(argv[i], "-r"))
    {
      if(++i<argc)
        GRID_CELL_SIZE = atof(argv[i]);
      else
      {
        puts(USAGE);
        exit(1);
      }
    }
    else if(!strcmp(argv[i], "-d"))
    {
      if(++i<argc)
        MINFRONTDISTANCE = atof(argv[i]);
      else
      {
        puts(USAGE);
        exit(1);
      }
    }
    else
    {
      puts(USAGE);
      exit(1);
    }
    i++;
  }
}
```

APPENDIX B: *OCCASIONAL COMMUNICATION* CODE

This appendix contains the changes to the SARA-1 code in Appendix A that limit communication to every third loop. It was used in the *occasional communication* experiments. All of the rest of the code is identical to that found in Appendix A. In the *no communication* experiments, the code was not changed from the code found in Appendix A. Instead, a –N program input of '1' was sent to each robot so that they did not look for any other robots and, therefore, assumed they were searching alone.

```
/*  In  'communication  variables'  section  of  global  variable
declarations, add: */

MCOM_TYPE_INITIALIZATION = -2;
MCOM_TYPE_SONAR_READINGS = 0;

/* In Main program, after checking target_found channel but before
checking robot channels, add: */

if(!target_homing && MCOM_TYPE_SONAR_READINGS%3 = -2)   //this allows
robot to enter communication loop only every third loop
{
  for(jj=2;jj>=0;jj--)             //robot  needs  to  perform  three
communications with each other robot
  {
    MCOM_TYPE_SONAR_READINGS -= jj; //update    variable    used    for
communication type
    .
    .
    .
/* code here is same: receive_sonars and map_building for all robots */
    .
    .
    .
  }
  MCOM_TYPE_SONAR_READINGS += jj;   //return    MCOM_TYPE_SONAR_READINGS
to original value
```

```
  if(target_homing)                 //if,    at    any    point    during
communication, target was found by another robot, then break from loop
  {
    break;
  }

/* this is the end of the changes; the rest of the code is identical */
```

APPENDIX C: PLAYER/STAGE FILES

This appendix contains all of the files associated with the Player/Stage software that were designed to use with SARA-1. The versions of the software are Player-1.6.4 and Stage-1.6.2. The first file contains an example of calling SARA-1 for five robots. The search.world and search.cfg files set up the simulation environment. The two .inc files are called in the search.world file. These are the only files that are needed to run the code[8].

Calling the program

```
#!/bin/sh
cd /stage-1.6.2/worlds
player search.cfg&
sleep 3
#echo -e "\n\n\n\n\nPress ENTER to continue"
#read nothing
/player-1.6.4/examples/c++/search -p 6665 -M 20 -N 5 -x 3.5 -y -10.5 &
/player-1.6.4/examples/c++/search -p 6666 -M 20 -N 5 -x 3.5 -y -10 &
/player-1.6.4/examples/c++/search -p 6667 -M 20 -N 5 -x 3.5 -y -11 &
/player-1.6.4/examples/c++/search -p 6668 -M 20 -N 5 -x 2.5 -y -10.5 &
/player-1.6.4/examples/c++/search -p 6669 -M 20 -N 5 -x 2.5 -y -10 &
```

search.world file

```
# Desc: Multiple "search" robots and one "target" to find
```

[8] To make the appendix simpler, only the files used with Simple Rooms are given. However, any room can be used by changing a few commands in the search.world file and one line of the search.cfg file.

```
# This is where the simulation time is changed
interval_sim 100  # milliseconds per update step
interval_real 500 # real-time milliseconds per update step

# defines Pioneer-like robots
include "pioneer.inc"
# defines 'map' object used for floorplans
include "map.inc"


# set the size of a pixel in meters
resolution 0.03 #3cm

# configure the GUI window--just the way it looks on screen
window
(
  size [662 550]
  center [0 0]
  scale 0.035
)

# load an environment bitmap
map
(
  bitmap "bitmaps/simple_rooms.png"  #this room comes with software
  size [20 16] #meters
  boundary 1
)

# create the robots
pioneer2dx
(
  name "robot1"
  color "blue"
  pose [-7 0 0] #meters
)
pioneer2dx
(
  name "robot2"
  color "magenta"
  pose [-7 0.5 0] #meters
)
pioneer2dx
(
  name "robot3"
  color "green"
  pose [-7 -0.5 0] #meters
)
pioneer2dx
(
  name "robot4"
  color "yellow"
  pose [-8 0 0] #meters
)
```

```
pioneer2dx
(
  name "robot5"
  color "cyan"
  pose [-8 0.5 0] #meters
)

#TARGETS
position(
 name "target"
 size [0.5 0.5]
 pose [5 6.5 0] #meters
 color "red"
 # loads a bitmap for the model's body
 bitmap "bitmaps/space_invader.png"
```

search.cfg file

```
# Desc: Stage configuration file for controlling a search and rescue
task
# Author:  Adam Ray
# Date: October 1, 2005

driver
(
  name "lifomcom"
  provides ["6664:mcom:0"]
)

# load the Stage plugin simulation driver
driver
(
  name "stage"
  provides ["simulation:0"]
  plugin "libstage"
  # load the named file into the simulator
  worldfile "search.world"
)

# robot 1
driver
(
  name "stage"
  provides ["6665:position:0" "6665:sonar:0" "6665:blobfinder:0"]
  model "robot1"
)
# robot 2
driver
(
  name "stage"
  provides ["6666:position:0" "6666:sonar:0" "6666:blobfinder:0"]
  model "robot2"
)
# robot 3
```

```
driver
(
  name "stage"
  provides ["6667:position:0" "6667:sonar:0" "6667:blobfinder:0"]
  model "robot3"
)
# robot 4
driver
(
  name "stage"
  provides ["6668:position:0" "6668:sonar:0" "6668:blobfinder:0"]
  model "robot4"
)
# robot 5
driver
(
  name "stage"
  provides ["6669:position:0" "6669:sonar:0" "6669:blobfinder:0"]
  model "robot5"
)
```

map.inc file

```
define map model
(
  color "black"
  gui_nose 0
  gui_boundary 1
  gui_grid 0
  gui_movemask 0
)
```

pioneer.inc file

```
# Desc: Device definitions for Activemedia robots.
# Author: Andrew Howard, Richard Vaughan
# Date: 10 Jun 2002
# Modified by: Adam Ray
# Date: 01 Oct 2005


# The Pioneer2DX sonar array
define p2dx_sonar ranger
(
  scount 16

  # define the pose of each transducer [xpos ypos heading]
  spose[0] [ 0 0 90 ]
  spose[1] [ 0 0 67.5 ]
  spose[2] [ 0 0 45 ]
  spose[3] [ 0 0 22.5 ]
  spose[4] [ 0 0 0 ]
  spose[5] [ 0 0 -22.5 ]
```

```
  spose[6] [ 0 0 -45 ]
  spose[7] [ 0 0 -67.5 ]
  spose[8] [ 0 0 -90 ]
  spose[9] [ 0 0 -112.5 ]
  spose[10] [ 0 0 -135 ]
  spose[11] [ 0 0 -157.5 ]
  spose[12] [ 0 0 180 ]
  spose[13] [ 0 0 157.5 ]
  spose[14] [ 0 0  135]
  spose[15] [ 0 0  112.5]

  # define the field of view of each transducer [range_min range_max
view_angle]
  sview[0] [0 5.0 15]
  sview[1] [0 5.0 15]
  sview[2] [0 5.0 15]
  sview[3] [0 5.0 15]
  sview[4] [0 5.0 15]
  sview[5] [0 5.0 15]
  sview[6] [0 5.0 15]
  sview[7] [0 5.0 15]
  sview[8] [0 5.0 15]
  sview[9] [0 5.0 15]
  sview[10] [0 5.0 15]
  sview[11] [0 5.0 15]
  sview[12] [0 5.0 15]
  sview[13] [0 5.0 15]
  sview[14] [0 5.0 15]
  sview[15] [0 5.0 15]

  # define the size of each transducer [xsize ysize] in meters
  ssize[0] [0.01 0.05]
  ssize[1] [0.01 0.05]
  ssize[2] [0.01 0.05]
  ssize[3] [0.01 0.05]
  ssize[4] [0.01 0.05]
  ssize[5] [0.01 0.05]
  ssize[6] [0.01 0.05]
  ssize[7] [0.01 0.05]
  ssize[8] [0.01 0.05]
  ssize[9] [0.01 0.05]
  ssize[10] [0.01 0.05]
  ssize[11] [0.01 0.05]
  ssize[12] [0.01 0.05]
  ssize[13] [0.01 0.05]
  ssize[14] [0.01 0.05]
  ssize[15] [0.01 0.05]
)

# The Pioneer2DX blobvinder
define p2dx_blob blobfinder
(
  channel_count 6
  channels [ “red” “green” “blue” “cyan” “yellow” “magenta” ]
  range_max 4.0
```

```
  ptz [0 0 60.0]
  image [80 60]
)

# a Pioneer 2 or 3 in standard configuration
define pioneer2dx position
(
  # actual size
  size [0.44 0.33]

  # the pioneer's center of rotation is offset from its center of area
  origin [-0.04 0.0 0]

  # draw a nose on the robot so we can see which way it points
  gui_nose 1
  gui_boundary 0

  # estimated mass in KG
  mass 15.0

  # this polygon approximates the shape of a pioneer
  polygons 1
  polygon[0].points 8
  polygon[0].point[0] [  0.23  0.05 ]
  polygon[0].point[1] [  0.15  0.15 ]
  polygon[0].point[2] [ -0.15  0.15 ]
  polygon[0].point[3] [ -0.23  0.05 ]
  polygon[0].point[4] [ -0.23 -0.05 ]
  polygon[0].point[5] [ -0.15 -0.15 ]
  polygon[0].point[6] [  0.15 -0.15 ]
  polygon[0].point[7] [  0.23 -0.05 ]
  polygon[0].filled 1

  # omni-directional steering model
  drive "omni"

  # use the sonar array defined above
  p2dx_sonar()

  # use the blobfinder defined above
  p2dx_blob()
)
```

APPENDIX D: PATCHES

This appendix contains two patches for the Player-1.6.4 software. The first patch fixes an error in the blobfinder camera interface, in which the blob_count would not reset to zero. The second patch is a quick and task-dependent fix to the MCom interface in which erroneous data would occasionally get through the device to the robots. Both patches are only guaranteed to work with the Player-1.6.4 version of the software.

Blobfinder patch

The blobfinder patch is available at <http://playerstage.sourceforge.net>. It is called "Blobfinder doesn't zero blob count part 2a" and was written by Mike Gauthier. To install the patch, follow these instructions:

1. Get the actual patch, described above (the file name is hdrSize2.diff)
2. Save the file in the player-1.6.4 directory and navigate to that directory. With the hdrSize2.diff file in the current directory, type:

   patch -p0 --verbose -i hdrSize2.diff

   The patch takes about a minute.
3. After this, perform a make, make install of Player. This will give an error.
4. In order to get rid of this error, comment out the entire if statement in the mapproxy.cc file, line 188-191 (to find this file, go to a command prompt, and type "locate mapproxy.cc")

MCom patch

```
/* Add these three lines to the beginning of the file, mcomproxy.cc (to
find this file, go to a command prompt, and type "locate mcomproxy.cc")
*/
#include <iostream>
```

```
using namespace std;
int internal_ii;

/* Add a few lines to the Read function that will only allow 0-9, N
(for the negative flag), and '\0' (NULLS) to come through the device to
the robot.  This fix will only work with the search-and-rescue code, or
any other code that only sends these twelve characters.  The entire
Read function (with changes) is reprinted: */

int internal_ii;                                    //ADDED
int MComProxy::Read(int type, char * channelQ){
    player_msghdr_t hdr;
    if(!client)
        return(-1);
    player_mcom_config_t cfg;
    cfg.command = PLAYER_MCOM_READ_REQ;
    cfg.type=htons(type);
    strcpy(cfg.channel,channelQ);
    player_mcom_return_t reply;
    int r = client->Request(m_device_id,
            (const char*)&cfg, sizeof(cfg), &hdr,
            (char*)&reply, sizeof(reply));
    if(r < 0)
        return r;
    if(hdr.type != PLAYER_MSGTYPE_RESP_ACK) {
        memset(&data, 0, sizeof(data));
        type = 0;
        memset(channel, 0, sizeof(channel));
        return -1;
    }
    data=reply.data;
                                            //NEXT FEW LINES ADDED
    for(internal_ii=0;internal_ii<MCOM_DATA_LEN;internal_ii++)
    {
      if (data.data[internal_ii] != 'N' && data.data[internal_ii] !=
'0' && data.data[internal_ii] != '1' && data.data[internal_ii] != '2'
&& data.data[internal_ii] != '3' && data.data[internal_ii] != '4' &&
data.data[internal_ii] != '5' && data.data[internal_ii] != '6' &&
data.data[internal_ii] != '7' && data.data[internal_ii] != '8' &&
data.data[internal_ii] != '9' && data.data[internal_ii] != '\0')
      {
        cerr <<endl <<"**MComProxy internally ignored non-numerical
data...ERROR SAVED**";
        return -1;
      }
    }
    type=htons(reply.type);
    strcpy(channel,reply.channel);
    return 0;
}
```

APPENDIX E: DATA VALUES FROM FIGURES

This appendix gives the data values that make up each of the four 'time for task completion' graphs in Chapter Five. These data values are shown in the graphs as red circles. All time values are given in seconds.

'Time for task completion' graph of Figure 16, *continuous communication*

		Number of Robots						
		1	2	3	4	5	6	7
Run #	1	254	415	73	91	176	196	169
	2	119	241	193	150	157	157	137
	3	296	198	79	88	139	173	147
	4	245	157	145	139	108	223	201
	Avg.	228.5	252.75	122.5	117	145	187.25	163.5

'Time for task completion' graph of Figure 17, *occasional communication*

		Number of Robots						
		1*	2	3	4	5	6	7
Run #	1	254	228	54	100	104	94	69
	2	119	241	76	130	139	61	234
	3	296	211	160	102	153	68	136
	4	245	242	217	126	114	64	136
	Avg.	228.5	230.5	126.75	114.5	127.5	71.75	143.75

* for the 1-robot column, values were re-used from a previous experiment.

'Time for task completion' graph of Figure 19, *no communication*

Run #	Number of Robots			
	1*	2	3	4
1	254	254	46	317
2	119	243	198	146
3	296	190	109	68
4	245	196	344	471
Avg.	228.5	220.75	174.25	250.5

* for the 1-robot column, values were re-used from a previous experiment.

'Time for task completion graph' of Figure 25, Broun Hall, *continuous communication*

Run #	Number of Robots			
	1	3	5	7
1	2132	212	1080	507
2	867	1308	2226	1999
3	2874	209	176	576
4	905	627	504	1029
Avg.	1694.5	589	996.5	1027.75