AN ALGORITHM FOR NETWORK FORMATION AND AN IMPLEMENTATION OF A MOBILE ROBOTIC ROUTER SYSTEM

By

Wei Yang

A Thesis Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE

Major Subject: COMPUTER SCIENCE

Approved:

_____

Dr. I. Volkan Isler, Thesis Adviser

Rensselaer Polytechnic Institute
Troy, New York

June 2008
(For Graduation August 2008)

© Copyright 2008
by
Wei Yang
All Rights Reserved

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENT

I would like to express my gratitude to all those who gave me the possibility of completing this thesis.

My utmost gratitude goes to my thesis advisor, Dr. I. Volkan Isler, for his guidance during my research and study at RPI. Without his help, stimulating suggestions, and encouragement, none of this would have been possible. His passion and enthusiasm for this field has greatly motivated me throughout my graduate and undergraduate years and has helped to inspire a lifelong interest in mobile robotics.

I am forever indebted to Eric Meisner and Onur Tekdas for their significant contributions that my thesis is based on. Your work has inspired me to take on new challenges and that no problem is too big to solve.

I would like to thank everyone from the Robotics Lab for supporting me in my work and providing a tremendous research environment. I would also like to thank the Computer Science department for allowing me to use their resources and for providing me with a wonderful opportunity to teach and inspire other students.

I am forever grateful for all the friends that I have made while studying at RPI and for the ones that I have kept from Guilderland. Your support has helped me throughout the years, both inside and outside the academic field, and especially during those times when I needed to take a break from my studies and unwind over a few (or several) beers.

I cannot end without thanking my family for their unwavering support, despite the hardships in their lives, so that I could grow up and live in the best possible environment. I am ever grateful for my mother, Yan Situ, whose boundless love and constant encouragement I have relied throughout my life. I would like to thank my father, Bin Yang, for providing the role model and guidance that every boy needs to become a successful man in life.

ABSTRACT

Evidence of the prevalence of wireless networking devices can be seen everywhere. These days, more and more consumer electronics are being released with the ability to communicate with each other, free from the limitations of wires and restricted only by distance. These advances have also greatly helped with the mobile robotics field as well. Many algorithms can now move from computer simulations to real robotic systems, helping to develop practical applications that solve important problems involving multiple robot.

This thesis explores two problems in networked robotics. Its contributions are organized in two parts.

The first part of this thesis addresses the following problem: imagine a number of robots, with unknown locations, are scattered in an environment. How can a network be formed as quickly as possible? The thesis takes a theoretical approach to this problem of network formation by presenting a novel network formation algorithm and analyzing its performance. The algorithm developed is then contrasted and compared to another algorithm that has a more intuitive, but not necessarily more efficient, approach. The performance bounds of these two algorithms are analyzed and compared from both a mathematical standpoint and in computer simulations. A proof-of-concept implementation on a real system is also presented.

In the second part of the thesis, a networked mobile robot system that provides connectivity services to mobile users is developed. This part of the thesis takes a systems approach and presents the details of a full implementation of two algorithms for connectivity maintanence.

From a high level, it explores the challenges faced when implementing a system to test and run simulated algorithms. It also covers different routing techniques that can be used to control a set of distributed robots. From a low level, it looks at the different networking and development technologies that are needed to develop a working system.

These two levels, when combined, cover every stage of a networking algorithm's development process: from analysis to simulation and even implementation. This thesis also covers the entire application life-cycle of a networked robotics system from the initial network formation to maintaining network connections in order to collectivity accomplish a given task.

CHAPTER 1
Introduction

1.1 Overview

Evidence of the prevalence of wireless networking devices can be seen everywhere. It has become a necessary standard on every laptop computer released in the past few years. They have even started to appear on devices often thought of as requiring connection wires such as printers, scanners and digital cameras. These days, more and more consumer electronics are being released with the ability to communicate with each other, free from the limitations of wires and restricted only by distance. Wireless hotspots, where anyone with a laptop can just sit down and start using the Internet, have exploded in urban environments. They are now appearing everywhere all over the world from major corporate coffee shops such as Starbucks to locally owned cafes to even fast food restaurants such as McDonald's.

These advances have greatly helped the mobile robotics field as well. The mobile robotics field has been around for many years with extensive research on topics such as multi-robot communication and motion planning of distributed robots. Many algorithms rely on teams of robots to communicate with each other to collectivity accomplish a given task. With the advances of wireless and robotics technologies, it has become much more feasible to conduct research and create systems using large groups of real mobile robots. This has helped to further push areas of multi-robot systems into solving practical problems with real world applications such as floor cleaning, lawn mowing, mine hunting, search and rescue. Along with these area coverage problems, teams of robots can also work together to map and explore unknown or even hostile remote environments. This will in turn, lead to more robust consumer electronics where groups of robots might one day help accomplish mundane tasks we reluctantly work on now. The iRobot Roomba, released in 2002, is a small mobile robot that is used to autonomously vacuum a house and at the time of this writing, has sold over three million units. This demonstrates that small mobile robots are slowly but surely being incorporated into our normal lifestyles.

1.2 Motivation and Contributions

One of the most fundamental aspects of having teams of robots work together is their ability to network and communicate. Often, this can become a challenge as there are many times when the robots' communication range is much smaller than the area of their environment. This is

especially true when considering the limited range of wireless radios and the vast environments mobile robots might one day be assigned to explore. Robots distributed in this environment will face the difficult problem of setting up an initial communication network, even if they are scattered randomly outside of each other's communication range. This situation is explored in the first of two contributions of this thesis.

The first part of this thesis takes a theoretical look at the problem of network formation by developing an algorithm to solve this fundamental challenge to mobile robotics. This problem is unavoidable for many situations but is often overlooked by many mobile robot algorithms. This thesis presents a new algorithm, Stripes, and analyzed its performance. The Stripes algorithm is then contrasted and compared to another algorithm that has a more intuitive, but not necessarily more efficient, approach. The performance bounds of these two algorithms are analyzed and compared from both a mathematical standpoint and in computer simulations. A proof-of-concept implementation on a real system is also presented.

While a limited communication range is seen as a problem in many cases, there are also other scenarios where its properties can be used to solve problems and provide practical applications. One of these applications is presented in [14]. In this paper, several mobile robots use their wireless radios to help maintain a constant multi-hop network connection between another mobile device and base station, both outside of each others' communication range. The properties and limitations of wireless communication play a large role in the development of the resulting motion planning algorithms and an even larger role in the second contribution of this thesis: the implementation of these algorithms on a physical mobile robot system.

The second part of this thesis focuses on the system component in the context of another application. Its contribution is the development of a networked mobile robot system from the ground up in order to implement the motion planning algorithms from [14]. From a high level, it explores the challenges faced when implementing a system to test and run simulated algorithms. It also covers different routing techniques that can be used to control a set of distributed robots. From a low level, it looks at the different networking and development technologies that are needed to develop a working system. This includes everything from the methods and formats that are used to transfer data between robots to the programming techniques that allow for better and more scalable applications to be created.

These two parts explore different aspects of wireless communications in robotics systems. When they are combined, their contributions cover every stage of the development of a networked robotic system. These stages include the creation, analysis and simulation of an algorithm, and

the development and implementation of a robotic system. This thesis also covers the entire application life-cycle of mobile robotics, starting with a common problem faced when they are initially deployed into an environment. Afterwards, this thesis demonstrates how the robots could use their network to effectively work together and collectivity accomplish a given task.

1.3 Outline

This thesis is separated into three main chapters with the first chapter dedicated to the introduction, motivation and contributions.

Chapter 2 studies the problem of network formation and proposes an algorithm which efficiently passes information from one robot to all of the other robots in the environment. It first introduces the problem, some background information and the problem formulation. Afterwards, Section 2.3 explains the new network formation algorithm, which is analyzed and contrasted to another possible solution to the problem. These two strategies are simulated and their results are compared in Section 2.4. Section 2.5 demonstrates the feasibly of the proposed algorithm in a proof-of-concept experiment. The chapter ends with a discussion on possible extensions to the algorithm and future work.

Chapter 3 is dedicated to the mobile robotic system used to implement the motion planning algorithms from [14]. This chapter explores the many different networking technologies and routing techniques used at all levels of the system. It starts off with an introduction of the problem and then gives an overview of the system and how it will use the motion planning algorithms. Section 3.2 explains the routing strategies and techniques that the system uses to control the robots from across the network. Afterwards, Section 3.3 dives down into the actual implementation and describes the specific technologies it uses. An overview of the different custom applications and some of their programming designs is given in Section 3.4. Following this, Section 3.5 describes the experiments that were carried out using the mobile robotic system. The last section, Section 3.6, describes the scalability of the system and improvements that can be made to further increase its benefits and contributions.

CHAPTER 2
Network Formation Through Freeze Tag and Coverage

2.1 Introduction

Consider the following scenario: a number of robots are performing independent tasks autonomously in an environment where there is no communication infrastructure. Suppose, at a certain point, mission priorities change and a piece of information must be propagated to all nodes. For example, robots could be initially stationed to perform monitoring or surveillance in a large environment. Upon detection of an event, they may have to form a network or perform a collaborative task. What is a good strategy to get all robots involved as quickly as possible?

The contribution from this part of the thesis studies this process of propagating information as quickly as possible. Specifically, it studies the case where the process is initiated by a single robot. This robot could, for example, be sent out from the command and control center. Alternatively, it could be the robot that detects an intruder. The primary difficulty in solving the problem arises from the fact that the robots do not know each others' positions. The first robot must therefore start a search. Once discovered, other robots can participate in propagating the information. Since the primary motivation for studying this problem is to form a connected network, throughout this thesis this problem will be referred to it as the *Network Formation Problem*.

Contributions This chapter of the thesis studies a probabilistic scenario where the locations of robots are chosen uniformly at random in a rectangular environment. For this scenario, a network formation strategy (Stripes) is presented and prove that its expected performance (i.e. network formation time) is within a logarithmic factor of the optimal performance. To obtain this result, a lower bound on the expected performance of *any* network formation strategy was obtained. Stripes is also compared with a natural, intuitive "Split and Cover" strategy and shown that in large environments, Split and Cover has inferior performance to Stripes. In addition to formal performance bounds, this work demonstrates the utility of Stripes with simulations and its feasibility with a proof-of-concept implementation.

The next section starts with an overview of related work where connections are established between the network formation problem and other fundamental problems such as rendezvous, coverage and freeze-tag.

2.1.1 Related Work

Considerable work has been done in designing decentralized protocols for propagating information in networks (also known as gossip protocols) [7]. Gossip protocols are mainly designed for stationary networks. In contrast, this work focuses on information propagation among mobile robots. The network formation problem is closely related to the Freeze-Tag problem [4]. In freeze-tag, a number of players are "frozen". A single unfrozen player must visit each frozen player in order to unfreeze it, at which point it can aid in unfreezing other players. In freeze tag, it is assumed that the players know each others' positions. In network formation, this focus is on the case where the node locations are unknown to each other.

Recently, Poduri and Sukhatme explicitly addressed the problem of forming a connected network under the name *coalescence* [11, 12]. In their model, all of the nodes (other than the base station) are performing a random walk on a torus. The authors obtain bounds on the network formation time. The advantage of the random-walk strategy is that it does not require localization with respect to a global reference frame. However, the network formation is rather slow because nodes visit most locations many times. This may not be acceptable in time-critical applications. This work addresses the problem of explicitly designing motion strategies for network formation with guaranteed performance.

Since the focus is on the case where the robot locations are unknown, the network formation problem is related to rendezvous and coverage.

The rendezvous search problem [2] asks how two players with unit speed can locate each other when placed at arbitrary locations in a known environment. Typically, each player has a radius of detection within which the players can establish communication. The goal of the players is to find each other as quickly as possible. The rendezvous problem has been studied extensively for two players on a line. Optimal or provably good strategies for various versions of this problem have been established [1].

The problem of multi-player rendezvous search on a complete graph is studied in [15]. This work addresses the question of whether players that meet should stick together or split and meet again later. The result of this work shows that, if the players have no memory, then the optimal strategy is to stick together. Also, simulations show that as the number of players increases, the split and meet strategy becomes less effective. In general, this has limited applications because the environment is a complete graph. In other words, players can teleport to arbitrary locations at every time step.

Work in [3] describes a time optimal strategy for two-player limited visibility rendezvous

in the plane with known and unknown distances. The optimal solution in this case is for one of the players to follow a semi-circular spiral trajectory. This work also relates rendezvous in the continuous domain to coverage problems. Deterministic and randomized multi-robot rendezvous strategies were proposed in [13]. More recently, results on rendezvous in simply-connected polygons have been obtained [8]. In [9], the authors provide upper and lower bounds on the time complexity of two geometric laws that result in rendezvous.

The coverage or lawn mowing problem [5, 6] has been extensively studied and is known to be closely related to Traveling Salesperson Problem. The primary difference between network formation and standard multi-robot coverage is that in coverage, all robots participate in the coverage process from the beginning. In network formation, the process starts with one robot, who must "recruit" others to participate in coverage.

In short, the algorithm presented in this thesis can be considered a novel algorithm for (i) online freeze-tag, (ii) probabilistic multi-robot coverage, and (iii) network formation.

2.2 Problem Formulation

Since the robot locations are unknown, network formation is an online problem. Online problems are typically analyzed using competitive analysis where the input is chosen by an adversary. The competitive ratio of an online algorithm $\mathcal{O}$ is the worst case ratio of the performance of $\mathcal{O}$ to the optimal offline performance. In other words, this work compares $\mathcal{O}$ with the optimal algorithm that has access to all of $\mathcal{O}$'s choices in advance and consider the worst case deviation from this optimal behavior.

It is easy to see that there is no online algorithm for the network formation problem with bounded competitive ratio: no matter which path the first robot chooses, the adversary can place all other robots at the last location the first robot will visit. In the case of a square environment with area a^2, the online algorithm would take time proportional to a^2 whereas the optimal offline cost would be at most in the order of a. Therefore the competitive ratio would be a and it would grow unboundedly with the size of the environment.

Since there are no competitive online algorithms for network formation, in this thesis the focus is on the probabilistic case where the locations of the robots are sampled from a distribution. In the lack of any information, it is reasonable to assume that the locations are chosen uniformly at random from the environment. *The goal is to minimize the expected time to discover all robots.* In this thesis, the main focus is on rectangular environments and uniform distributions. It is believed that the rectangular environment case has practical relevance for robots operating in an open field

as well as for Unmanned Aerial Vehicles. Extensions to general convex environments are discusses in Section 2.6.

2.2.1 Robot Model

In network formation, several identical mobile robots are distributed uniformly at random within the bounded rectangle A with one of these robots possessing information that needs to be propagated to all of the other robots. The rectangle A has a width w and height h where $w \geq h$. It is assumed that the robots are initially stationary. Once discovered, they can move anywhere within the rectangle A at a constant speed and can communicate with each other within a limited range. Once a robot is within the communication range of another robot, they can exchange any information that either robot might have, including the information that needs to be propagated. In this thesis, it is assumed that the robots can localize themselves within A. Energy limitations are also ignored and it is assumed that the robots will always move at their maximum speed and can utilize their wireless radio anytime.

Notation and Conventions: The units are normalized so that the robots move at unit speed per time-unit. The rest of the paper abuses the notation and use A to denote both the rectangle environment and its area. This convention implies that a single robot can cover the entire environment in A time units. Hence, A is a trivial upper bound on network formation since inactive nodes are stationary. The number of robots in the environment is k. It is assumed that the first robot enters the environment at a corner of A.

2.3 Stripes: An Efficient Network Formation Strategy

This section presents the main result: an efficient network formation strategy referred to as "Stripes". This strategy relies on dividing the rectangular environment into n equal sized vertical stripes $S_1, \ldots, S_n$ (Figure 2.1). For now, n will be treated as a variable whose value will be fixed later. Let S denote the area of a single stripe which is equal to the time to cover a single stripe with one robot.

In the beginning, a single robot is active and proceeds to cover S_1 with a simple back and forth sweeping motion. That is, the robot follows the well-known boustrophedon[1] path. When an inactive robot is encountered and activated, this newly active robot does not join in the coverage of S_1 right away [2]. Instead, it heads to a designated location on the line that separates S_1 and S_2

[1] boustrophedon = "the way of the ox" [6].

[2] In practice, this robot can participate in covering the stripe. However, this does not improve the analysis. Hence, this benefit is ignored for analysis purposes.

and waits for the first active robot to finish its coverage of S_1 and arrive at the designated location. When the first robot is finished, it meets all newly active robots at the same designated location. Let k_1 denote the number of active robots. The robots evenly divide S_2 among themselves so that it can be covered in parallel in time S_2/k_1.

When these k_1 robots encounter other inactive robots in S_2, the same procedure is repeated and all active robots meet at a designated meeting location on the line which separates S_2 and S_3. This process repeats for the remaining stripes until all of the stripes are covered, at which point, the entire bounded area will be covered and all of the robots will be activated.

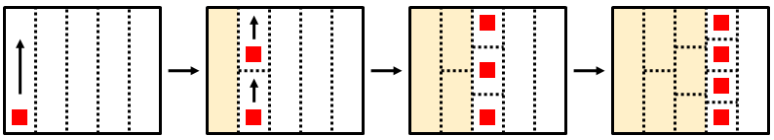


Figure 2.1: Stripes strategy: The environment is divided into equal vertical stripes which are covered sequentially. Active robots split the current stripe equally. Once a stripe is covered, all active robots (including newly discovered robots) meet at the boundary of the stripe.

The environment that will be used to highlight the main ideas of the contribution is a rectangular bounded area where its dimensions are much larger than the number of robots available, $w \geq h \gg k$.

Saturation occurs when the number of robots exceeds the size of the shorter side of the rectangular h, $k \geq h$. When this happens, the remaing $m \times h$ area can be covered in m steps by k or fewer robots. Further discussion of saturation can be found in Section 2.3.4. The analysis will refer to the case where $w \geq h \gg k$ as an *unsaturated environment* and $k \geq h$ as a *saturated environment*.

2.3.1 Network Formation with Stripes

This section will establish an upper bound on the network formation time of Stripes for an unsaturated environment. The following lemma from discrete probability will be used.

Lemma 1 *If k balls are assigned to n bins randomly, with high probability, the number of empty bins is at most $ne^{-\frac{k}{n}}$.*

See, for example, [10, pp. 94], for a proof of Lemma 1.

To obtain an upper bound on the network formation time with Stripes, the robots are treated as balls and stripes as bins. The number of empty stripes, x is then $x = ne^{-\frac{k}{n}}$. In the worst case,

all of these empty stripes occur at the beginning. When this happens, these stripes will be covered with a single robot. The resulting coverage time is $\frac{A}{n}x = Ae^{-\frac{k}{n}}$ for these empty stripes.

There are $n - x$ remaining non-empty stripes. In the worst case, each stripe contains only one robot with the remaining robots all residing in the last stripe. This results in a coverage time of $\frac{A}{n}$ for the first non-empty stripe, $\frac{A}{n}\left(\frac{1}{2}\right)$ for the second, $\frac{A}{n}\left(\frac{1}{3}\right)$ and so on for $n - x$ stripes. Therefore, the coverage time for the non-empty stripes will be:

$$\sum_{i=1}^{n-x} \frac{A}{n}\left(\frac{1}{i}\right) = \frac{A}{n}\sum_{i=1}^{n-x}\frac{1}{i} \approx \frac{A}{n}log(n-x) = \frac{A}{n}log\left(n - ne^{-\frac{k}{n}}\right) \tag{2.1}$$

Let $T(A, k)$ denote the expected time to cover area A with n stripes and k robots. From the equations above:

$$T(A,k) = Ae^{-\frac{k}{n}} + \frac{A}{n}log\left(n - ne^{-\frac{k}{n}}\right) + c \tag{2.2}$$

Since robots group together before and after covering a stripe, there is some overhead associated with the different strategies that can be used to cover an individual stripe. This is defined as c in the equation and is discussed in the following section.

Finding the best upper bound for Equation 2.2 involves finding the number of stripes that minimizes $T(A, k)$ for a given number of robots.

Figure 2.2 shows $T(A, k)$ as a function of n and k from two different viewpoints.

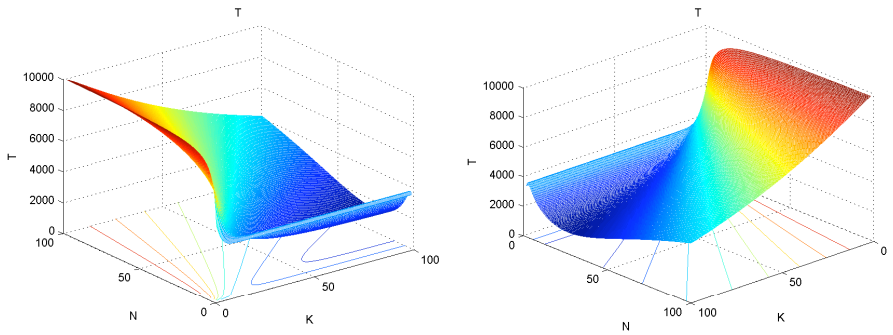


Figure 2.2: A 3D plot of the Stripes upper bound coverage time.

An analytical upper bound can be obtained by choosing the number of stripes to be $n = \frac{k}{log(k)}$, which gives us the following upper bound on $T(A, k)$.

$$
\begin{aligned}
T(A,k) &= Ae^{-\frac{k}{\frac{k}{log(k)}}} + \frac{A}{\frac{k}{log(k)}} log\left(\frac{k}{log(k)} - \frac{k}{log(k)} e^{-\frac{k}{\frac{k}{log(k)}}}\right) + c \\
&= Ae^{-log(k)} + \frac{A}{k} log(k) log\left(\frac{k}{log(k)} - \frac{k}{log(k)} e^{-log(k)}\right) + c \\
&= \frac{A}{k} + \frac{A}{k} log(k) log\left(\frac{k}{log(k)} - \frac{1}{log(k)}\right) + c \\
&< \frac{A}{k} + \frac{A}{k} log(k) log(k) + c \\
&= \frac{A}{k} + \frac{A}{k} log^2(k) + c
\end{aligned}
$$

Hence, the following result:

Lemma 2 *The network formation time for k robots in a rectangular unsaturated environment with area A is $O(\frac{A}{k} log^2(k))$ when robots execute the Stripes algorithm with $n = \frac{k}{log(k)}$ stripes.*

In establishing Lemma 2, the overhead c is ignored. This is justified in the next section.

2.3.2 Covering a Single Stripe

This section presents a strategy for multiple robots to cover a single stripe in a way that minimizes the constant c in Equation 2.2. Overhead occurs when the robots are regrouping and redistributing themselves to equally cover the next stripe. After meeting, the robots must travel to their newly assigned coverage areas and will most often have to travel through areas that have already been covered or will be covered by another robot. Since this overhead hurts the overall coverage time, and a strategy is presented that will practically eliminate the overhead for large rectangular environments.

The Stripes algorithm calls for all of the active robots to meet and regroup between covering stripes. The proposed single stripe strategy first sets the meeting locations to always be located on the same side of the environment as the previous meeting location. Essentially, at the end of the algorithm, all of the meeting locations will be lined up along the same side of the environment.

In order to minimize the coverage time of a single stripe, its area must be split equally between all of the active robots. This can be simply done by dividing the strip along the long side to create equal sized rectangular areas, one for each of the active robots. However, this presents a problem where the robot that has to cover the area furthest away from the meeting location has to travel the entire length of the environment. Since this area will be eventually covered by the other robots, this produces an overhead of $2h$ per stripe, adding up to $2nh$ for the entire bounded

area A. To eliminate this overhead, the single stripe strategy removes the area that a robot covers while traveling to its assigned coverage area from the assigned areas that belong to the other active robots. This prevents areas from being covered multiple times and thus, eliminating the overhead. However, in doing this, the coverage area for the robots are now uneven since the robot that is closest to the meeting locations will cover the least amount of area because it is gradually chipped way and assigned to the other active robots. To once again achieve minimal coverage time, each robots' assigned rectangular coverage area must now be resized to take into account the areas gained while traveling to and from the meeting locations. In the end, the paths of the robots resemble archways with different sized horizontal areas that inversely proportional to the distance a robot needs to travel to the meeting location (Figure 2.3). The "legs" of the archways represent the areas covered by each robot as they travel up to their assigned rectangular areas.

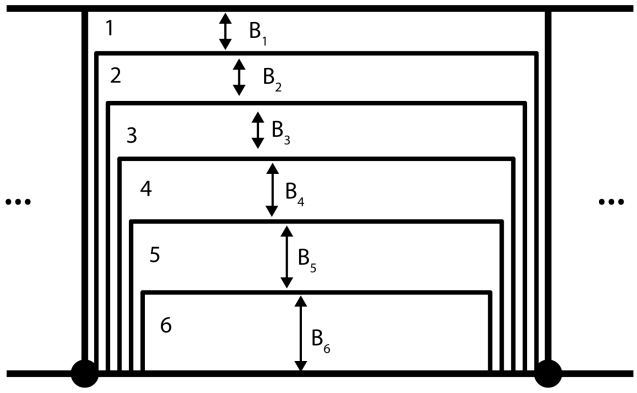


Figure 2.3: This single stripe coverage strategy divides a vertical stripe into equally sized areas that take into account any additional area covered by a robot as it travels away from the meeting locations (large black dots). This figure shows the resulting divisions for six active robots and the designated B_i values used to reference the height of the horizontal rectangular areas.

Given a particular vertical stripe to cover with r active robots, let $A_1, A_2, A_3 ... A_r$ designate the total assigned coverage area for each robot, with robot i covering A_i and robot $i = 1$ covering the area that is furthest away from the meeting location. Since each stripe is vertical, its area is $h\frac{w}{n} = hw'$. The height of the main rectangular part from each area A_i is assigned as B_i (see Figure 2.3). This is the parameter that changes to offset the additional area covered from traveling. Each robot's wireless radio is assumed to cover one unit area per time step so the additional travel area covered by robot 1 is simply twice the distance from the meeting point to its assigned coverage area or $2(h - B_1)$. This produces the following equations for calculating the total area that each robot must cover, with the generic equation as Equation 2.6:

$$\begin{aligned} A_1 &= B_1 w' + 2(h - B_1) \\ &= B_1(w' - 2) + 2h \qquad (2.3) \\ A_2 &= B_2(w' - 2) + 2(h - B_1 - B_2) \\ &= B_2(w' - 4) + 2h - 2B_1 \qquad (2.4) \\ A_3 &= B_3(w' - 4) + 2(h - B_1 - B_2 - B_3) \\ &= B_3(w' - 6) + 2h - 2B_1 = 2B_2 \qquad (2.5) \end{aligned}$$

$$A_i = B_i(w' - 2(i-1)) + 2(h - \sum_{j=1}^{i} B_j) \qquad (2.6)$$

Since each robot must cover the same area, the generic area formula at i and $i-1$ can be set equal to each other, producing a relationship between B_i and B_{i-1} (Equation 2.9). From here, all of the necessary heights for the rest of the areas can be calculated using a single given B_i value. To find this, it is a simple matter of setting one of the area equations to the area of a single stripe divide by the number of active robots. Therefore, in Equation 2.10, A_1 is equated to $\frac{hw'}{r}$, producing Equation 2.11 where B_1 is expressed using the number of active robots, r, the width of a stripe $w' = \frac{w}{n}$, and the height of the overall environment, h.

$$\begin{aligned} A_i &= A_{i-1} \qquad (2.7) \\ B_i(w' - 2(i-1)) + 2h - 2\sum_{j=1}^{i} B_j &= B_{i-i}(w' - 2(i-1-1)) \\ &\quad +2h - 2\sum_{j=1}^{i-1} B_j \qquad (2.8) \\ B_i w' - 2(i-1)B_i - 2B_i &= B_{i-1}w' - 2B_{i-1}(i-2) \\ \frac{B_i}{B_{i-1}} &= 1 + \frac{4}{w' - 2i} \qquad (2.9) \end{aligned}$$

$$A_1 = \frac{hw'}{r} = B_1(w'-2) + 2h \tag{2.10}$$

$$B_1 = \frac{\frac{hw'}{r} - 2h}{w'-2} = \frac{h(w'-2r)}{r(w'-2)} \tag{2.11}$$

The limitation of the single stripe strategy is that the number of active robots must be less than half of a single stripe width, $r < \frac{w'}{2}$. This does not present a problem for the analyzed environment where it is assumed that $w = h >> k$. Therefore, using this single stripe strategy, each stripe can be divided into r equally sized areas with minimal repeated coverage and thus, eliminating the overhead coverage time from the Stripes equation.

2.3.3 Lower Bound

This section establishes a lower bound on the expected network formation time that can be achieved by any algorithm. It is easy to see that the best coverage time for any area occurs when all of the robots are active at the beginning and the area is evenly split between all of them. Therefore, the absolute lower bound for total coverage time, regardless of algorithm, is $T(A,k) = \frac{A}{k}$.

For the case when the robots are uniformly distributed, a better bound is obtained using the following result.

Lemma 3 ([10], pp.45) *When n balls are assigned uniformly at random to n bins, with probability at least $1 - \frac{1}{n}$, no bin has more than $\alpha = \frac{\log n}{\log \log n}$ balls in it.*

Now consider any network formation strategy for k robots in area A. During the execution of this strategy, the coverage process is divided into epochs where i^{th} epoch ends when all active robots cover a (previously uncovered) total area of A/k. Let S_i denote the subset of A covered during epoch i. Let E_1 be the event that no S_i has more that α balls. By Lemma 3, E_1 happens with probability $(1 - 1/k)$.

When E_1 happens, the maximum number of robots in S_1 is α. Therefore, the minimum time it takes to cover S_1 is $T_1 = \frac{A}{k\alpha}$. There will be α new robots in S_2, therefore its coverage time T_2 is at least $\frac{A}{k2\alpha}$. Similarly, the i^{th} epoch will last at least $T_i = \frac{A}{ki\alpha}$ steps.

Since there are k epochs, the total time for all epochs to finish will be:

$$\sum_{i=1}^{k} \frac{A}{ki\alpha} \approx \frac{A}{\alpha k} \log k = \frac{A}{k} \log \log k \tag{2.12}$$

When E_1 does not happen (with probability $\frac{1}{k}$), the coverage time is at least A/k as discussed earlier. This gives us the lower bound on the expected coverage time of any algorithm.

Lemma 4 *The expected time for k robots to cover area A is at least* $(1-\frac{1}{k})\frac{A}{k}\log logk + (\frac{1}{k})\frac{A}{k}$.

Ignoring $o(\frac{1}{k^2})$ terms, a lower bound of $\Omega(\frac{A}{k}\log\log k)$ is established. Using Lemmata 2 and 4, the main result is established:

Theorem 1 *The performance of the Stripes strategy is within a factor $O(\frac{\log^2 k}{\log\log k})$ of the optimal performance for a rectangular environment where $w \geq h \gg k$.*

2.3.4 Saturated Environments

In a saturated environment where $k \geq h$, the analysis is a little different as the number of robots can exceed the height. This section will prove that even though the number of robots exceeds the height of the bounded area, the Stripes algorithm will still run with a ratio of $\mathrm{O}\left(\frac{log^2 k}{loglogk}\right)$. The balls and bins analogy is still used with the balls analogous to robots and the bins to stripes.

2.3.4.1 Upper Bound

The main difference between the coverage time for a saturated environment and an unsaturated environment how the $k \geq h$ constraint affects the coverage of a single stripe. The single stripe strategy section proposed a coverage strategy where a stripe is divided into horizontal areas, one for each active robot. In a unsaturated environment the height of one of these horizontal area will always be greater than one, even with the number of active robots increase in each stripe, because $h \gg k$. However, in a saturated environment, eventually the number of active robots in a stripe, r, will be equal to the height of a stripe because $k \geq h$. When this occurs, each horizontal area will have a height of only one unit and can be covered by having an active robot by simply move from one end of the stripe to the opposite end. If overhead is not taken into account, this produces a coverage time of $\frac{w}{n}$ or the width of a single stripe. Once this situation occurs, having additional active robots would not be able to decrease the coverage time because all of the robots must already travel across the width of a stripe just to move from one meeting point to another. Therefore, when $r \geq h$, the coverage time for a stripe becomes $\frac{w}{n}$. Any stripe where this occurs is called *saturated*.

This changes the original upper bound equation to contain an additional term. In the unsaturated environment, the coverage time was:

$$Ae^{-\frac{k}{n}} + \frac{A}{n}log\Big(n - ne^{-\frac{k}{n}}\Big) \tag{2.13}$$

This coverage time is a summation of the coverage time for the empty stripes and the coverage time for stripes with a single robot in them. The term in the logarithm represents the total number of non-empty stripes. For the saturated environment, this term will become the number of non-empty stripes before before saturation occurs. Since saturation occurs when the number of active robots exceeds the height and it is assumed that non-empty stripes only contain one active robot, this term becomes h. The additional term is the coverage time for the saturated stripes, which is the width of a single stripe multiplied by the number of remaining uncovered stripes. This equates to $\frac{w}{n}(n - ne^{-\frac{k}{n}} - h) = w - we^{-\frac{k}{n}} - \frac{A}{n}$. The overall coverage time becomes:

$$Ae^{-\frac{k}{n}} + \frac{A}{n}log(h) + w - we^{-\frac{k}{n}} - \frac{A}{n} \tag{2.14}$$

One benefit of saturation is that when it occurs, there is no longer a need for the active robots to meet after covering stripes.

2.3.4.2 Lower Bound

In the upper bound, the coverage time was different because $k \geq h$ had an effect on the coverage time of a single stripe. However, in the lower bound, the epochs can be in any configuration in the environment so the shape of the area does not affect the coverage time. Therefore, the lower bound remains at $\Omega(\frac{A}{k}\log logk)$, producing the result:

Theorem 2 *The performance of the Stripes strategy where the number of robots is greater than the shorter side of a rectangular bounded area, $k \geq h$, is within a factor $O(\frac{\log^2 k}{\log\log k})$ of the optimal performance.*

2.3.5 Split-and-Cover

This section discusses the performance of the following intuitive and natural network formation strategy: The first robot moves up and down the environment following a boustrophedon path as before. When it meets an undiscovered node, the two nodes split the undiscovered parts of the environment evenly and recursively cover their assigned partitions (see Figure 2.4).

The justification for this natural "Split-and-Cover" strategy is that since the nodes are scattered uniformly in the environment, each split should divide the load equally between discovered robots, therefore balancing (and intuitively minimizing) the network formation time.

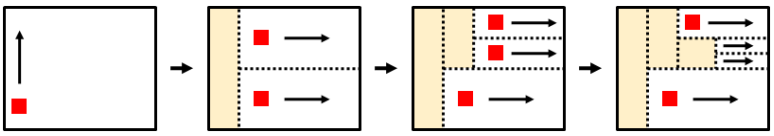


Figure 2.4: The Split-and-Cover strategy

It turns out that Split-and-Cover is not an effective strategy for large environments for the following reason: suppose that when the split happens, one of the partitions has a very small number of robots. Even though this event has a small probability for a single split, as the number of robots (and hence the number of splits) increases, it becomes probable. When such an imbalance occurs, the algorithm can not recover from it. A small number of robots must cover a large environment making Split-and-Cover inefficient. This argument is further justified with simulations and show that the Stripes algorithm is much more efficient than Split-and-Cover in unsaturated environments.

2.4 Simulations

Two simulation programs were developed to help compare the Stripes and Split and Cover algorithms. These simulations were designed in different ways and as a result, focused on different aspects of the algorithm performance.

The first simulation program focused on just the running time of the algorithms and uses a discrete grid environment where each cell represented one unit area. It also assumes that areas can be split evenly regardless of their shape and it ignores the robot traveling time and overhead associated with both of the algorithms. Section 2.4.1 is dedicated to this first simulation program and the preliminary comparison results it produced.

The second program was focused on creating more accurate simulations of the overall environment and coverage algorithms. This program was written by Eric Meisner and featured a continuous polygon area instead of a grid. It also took into account any overhead associated with covering the environment and produced a more accurate calculation of the total coverage time. Another improvement was that it demonstrated the single stripe algorithm with practical realistic paths that robots would travel. Section 2.4.2 focuses on this program and its results.

2.4.1 Discrete Simulations

To better analyze and compare the two different strategies, as well as the other variables that are present in the network formation problem, several simulations were developed to mimic

a real world problem. These simulations provide a better understanding of how the coverage time might be affected by different sized areas and the number of robots, and how each of the strategies would run.

The simulations were conducted on a Java application (Figure 2.5) and the results were analyzed using Matlab. In a real world situation, each robot would have a disc-like wireless communication region and would be able to move continuously within the bounded area. For simplicity and feasibility, the bounded area was discretized into a two dimensional grid with each grid cell representing the amount of area that a robot's wireless radio can reach. If two robots appear in the same grid cell, then they are within each other's wireless range and can communicate with one another. The maximum speed of the robots allows them to cover only one grid cell per time unit. Hence, the coverage time in the simulation is based on the number of grid cells a robot has to cover. The simulation does not take into account any overhead that is associated with splitting an area into equal sections for the split and cover strategy or with dividing stripes evenly between all active robots in the stripes strategy. The robots were distributed uniformly at random by starting them at random grid locations. It also allows the different covering strategies to be compared more accurately by having the algorithms run on the same set of distributed robots as inputs.

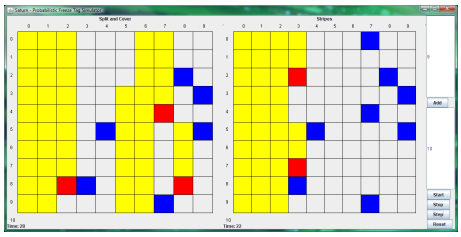


Figure 2.5: Java Simulator. Each side represents the bounded area after it is divided into a discrete two dimensional grid with the left side running the Split-and-Cover strategy and the right side running the Stripes strategy. The red squares represent cells with active robots, the blue squares represent cells with inactive robots. Grid cells that have been covered already are colored yellow.

2.4.1.1 Split-and-Cover

The Split-and-Cover simulation runs the algorithm exactly as it is stated in Section 2.3.5. It tracks the grid cell coverage by assigning each cell to be covered by an active robot. Initially, all of the grid cells are assigned to the first active robot, which starts covering them from one of the corner grid cells. When an inactive robot is discovered, the simulation simply assigns half of original active robot's remaining uncovered grid cells to the newly active robot. The robots then

separate and proceed with the rest of the Split-and-Cover strategy, with this process repeating for any other robots that they might find. Since each grid cell is always assigned to an active robot and meetings only change the active robot that a cell is assigned to, the entire bounded area is covered and the simulation ends when all of the robots have finished covering their assigned cells.

2.4.1.2 Stripes

In the Stripes algorithm, a single active robot starts out again in a corner grid cell and moves across the bounded area, covering one stripe or group of grid cells at a time. When an inactive robot is found, it becomes active but does not move. Although in the algorithm, this newly active robot would move towards the meeting point, it is not necessary to demonstrate this in the simulation as this robot will always be able to move to the meeting point before the active robot covering the stripe will be able to. After the original active robot finishes covering the stripe, the grid cells of the next stripe are divided evenly and assigned to each of the active robots. These active robots then start covering their assigned grid cells and this process repeats for the remaining stripes until all of the stripes are covered. This effectively mimics the single stripe algorithm proposed in Section 2.3.2 without creating the complex polygon coverage areas.

2.4.1.3 Results

The first set of simulations was run on a bounded area of 100x100 grid cells with 99 randomly distributed inactive robots and one active robot. The goal was to compare the Split-and-Cover strategy with the Stripes strategy and to determine if the number of stripes had an effect on the coverage time. The simulation results support the claim that the stripes strategy performs better than the split and cover strategy. See Tables 2.1 and 2.2.

Split-and-Cover	n=number of stripes	Stripes
935.53	21	843.93
930.32	30	739.92
942.81	40	686.59
925.94	57	639.87
940.26	80	629.56
933.45	100	627.13
936.05	150	622.00
931.74	200	625.64

Table 2.1: Comparison of Split-and-Cover with Stripes for varying number of stripes.

The simulation in Table 2.1 also revealed that as the number of stripes increases, the coverage time decreases. This makes sense intuitively as with more stripes, the size of each stripe

decreases so the overall stripes distribution becomes more granular and any newly active robot would be able to start covering area sooner. This decrease is not reflected in the theoretical analysis because in simulation, empty stripes are distributed throughout the bounded area. The analysis focused on the worst case and assumed that they all appear at the beginning.

The results also show that even if the number of stripes used is not optimal, the Stripes strategy still outperforms the Split-and-Cover algorithm and yields efficient network formation times.

A second set of simulations was run on various sized areas with varying number of robots (Table 2.2). The goal of this experiment was to determine if there exists a relationship between the size of the bounded area and the number of robots. This set of simulations also provided additional comparisons between the efficiency of the Stripes strategy and the Split-and-Cover strategy. The results show that when the area is held constant, having more robots will always decrease the coverage time, regardless of the strategy used.

$\sqrt{A}$	k	Split-and-Cover	Stripes ($n = k$)
50	100	235.58	199.07
100	100	933.34	627.14
100	150	736.12	493.97
100	200	569.49	425.34
150	100	2125.06	1349.94
150	150	1580.96	1006.13
150	200	1281.36	808.71
200	100	3740.22	2333.19
200	150	2810.29	1709.20
200	200	2277.60	1395.47

Table 2.2: Simulation results for different areas and number of robots. The first column represents the square root of a square bounded area or the length of one of the sides. The second column represents the number of robots distributed within the bounded area. The third and forth columns represent the network formation time with the Stripes strategy using the same number of stripes as robots distributed within the area.

The third set of simulations was aimed at determining how the number of robots affects the running time for a given area size (Table 2.3). The results show a logarithmic decay as adding more robots when there are only a few of them greatly reduces the amount of area that each robot needs to cover (Figure 2.6). When there are a lot of robots already, adding more would only be able to reduce the coverage time by a little as the bounded area is already widely spread between many robots.

k	Split-and-Cover	Stripes ($n = k$)
10	3904.85	3474.61
20	2648.97	2071.42
30	2082.37	1527.8
40	1741.61	1223.96
60	1325.24	891.41
80	1082.98	716.012
90	1020.73	663.022
100	933.34	627.14

Table 2.3: Simulation results for different number of robots in a 100×100 area. The first column represents the number of robots distributed within the area. The second and third columns represent the running time of the Stripes strategy using the same number of stripes as robots distributed within the area.

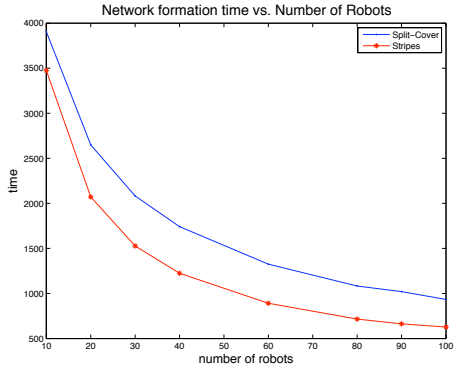


Figure 2.6: A plot of the results from Table 2.3.

2.4.2 Continuous Simulations

This section compares Stripes and Split-and-Cover in simulations with an continuous area. It also presents simulation results that demonstrate the effect of the number of Stripes on the performance of the Stripes algorithm.

The simulations were performed by representing each of the individual robots as a point within the rectangular world. Each robot can be assigned to sweep along a continuous piecewise linear path. As described in the Section 2.3, an active robot can detect an inactive robot when it is within communication range. Each robot moves one unit distance per unit time and maintains an internal clock, to represent the time with respect to the start of the first robot. Robots that meet can synchronize clocks. The robots were placed uniformly at random within the environment at the start of each trial. The simulation runs the algorithm exactly as it is stated in Section 2.3.5.

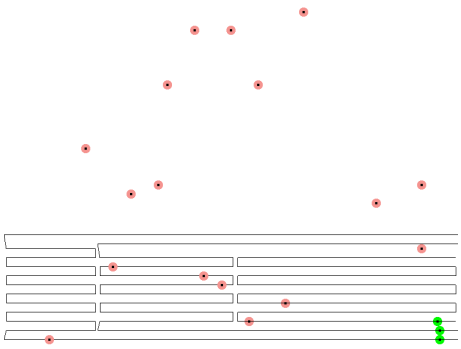


Figure 2.7: This figure shows the simulation of the Stripes algorithm. Green circles are active robots, and red circles are inactive. The black lines represent the paths of active robots within the current stripe.

2.4.2.1 Split-and-Cover

The Split-and-Cover simulation runs the algorithm exactly as it is stated in Section 2.3.5: It starts by decomposing the environment into a single piecewise linear (Boustrophedon) path, starting at a corner of A, and assigning it to the first robot. When an an active robot discovers an inactive robot, it splits the remainder of its assigned path, and gives half of it to the discovered robot. The newly active robot travels a straight line to the start of its assigned trajectory. In doing so, it counts the cost of travel by incrementing its internal clock accordingly. The simulation ends when the last robot has completed its assigned coverage task.

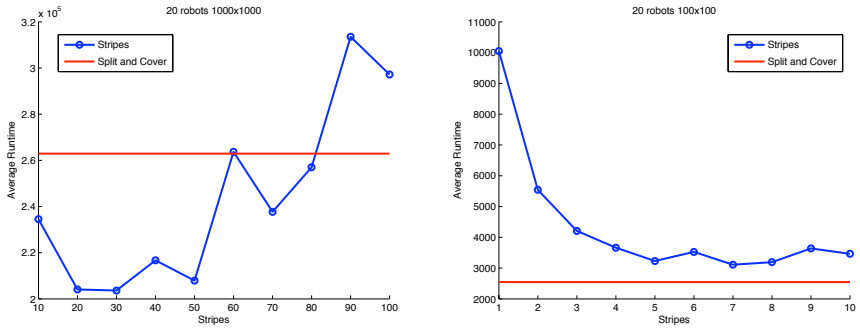
2.4.2.2 Stripes

The Stripes algorithm divides the environment into n stripes. The first stripe is decomposed into a continuous piecewise coverage path, as described in Section 2.3.2, and assign it to a single active robot. The robot begins moving along this path and any robots that are discovered move to the starting location of the next stripe. When all active robots have reached the meeting point, the active robots synchronize their internal clocks. The next stripe is decomposed according to Equation 2.7 and the number of active robots. These active robots then start covering their assigned regions. This process repeats for the remaining stripes until all of the stripes are covered, or all of the robots are active.

2.4.2.3 Results

The first simulation compares Stripes and Split-and-Cover in environments that are sparse and dense with respect to the number of robots per area.

Figure 2.8(a) shows the results of the Split-and-Cover and Stripes algorithms in an environment where the concentration of robots is low. The simulations use a 1000×1000 environment and 20 robots. The plotted values are the average time to completion of 1000 trials, and the value for each number of stripes uses the same set of initial robot distributions. These simulations conclude that (i) the performance of the Stripes algorithm is sensitive to the number of stripes, and (ii) in sparse environments (with a "good" stripe-number selection) Stripes outperforms Split-and-Cover. This is also justified by Figures 2.9(a) and 2.9(b), where a histogram plot of running times of the Stripes algorithm (with 25 Stripes) and Split-and-Cover. Stripes not only outperforms Split-and-Cover on the average but also its worst case performance is considerably better.

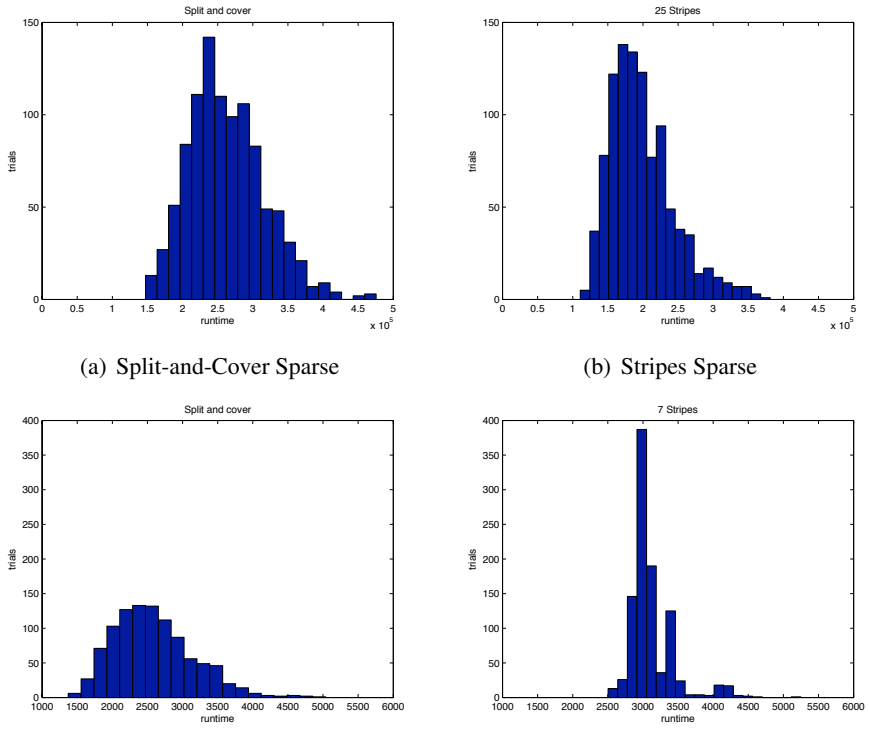


(a) Sparse environment (b) Dense environment

Figure 2.8: Comparison of Split-and-Cover strategy with Stripes algorithm for varying number of stripes. The plotted values are the average time to completion of 1000 trials. Left: 1000×1000 unit world with 20 robots. Right: 100 ×100 unit world with 20 robots.

On the other hand, when the concentration of robots is high, the Split-and-Cover outperforms Stripes (Figures 2.8(b),2.9(c),2.9(d)). There are two reasons that this is true. First, for Split-and-Cover, the unbalanced split described in the previous section, occurs with extremely low probability when the concentration of robots is high. Second, for Stripes the overhead cost of meeting up to evenly distribute the coverage of a stripe, becomes very large compare to the discovery time.

In conclusion, simulation results suggest that Stripes should be used in sparse environments with the number of stripes equal to the number of robots. In dense environments, split-and-cover algorithm is expected to yield better performance.



(a) Split-and-Cover Sparse

(b) Stripes Sparse

(c) Split-and-Cover Dense

(d) Stripes Dense

Figure 2.9: Top:Distribution of time to completion for Split-and-Cover(2.9(a)), and for Stripes(2.9(b)) in a sparse environment. Bottom: Distribution of time to completion for Split-and-Cover(2.9(c)), and for Stripes(2.9(d)) in a dense environment.

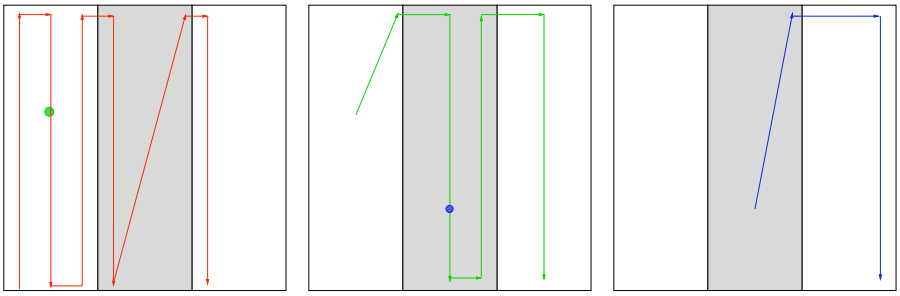
2.5 Experiments

This section demonstrates the feasibility of the Stripes algorithm using a small team of three Acroname Garcia robots shown in Figure **??**. The robots are each equipped with ARM/risk PCs, and wireless network adapters. When configured to work in ad-hoc mode, the robots form a wireless sensor network, each capable of determining its own set of neighbors in the network graph.

Currently, the robots do not have visual localization capabilities. Therefore, motor encoders and dead reckoning were relied on for position information. An external stereo camera was also utilized as a means of determining the robot positions off-line (to obtain ground truth). The external camera allows us to compute robot positions, but also limits the size of the work area to the field of view of the camera. The experiments take place in a square workspace of 7.5 meters by 7.5 meters. These experiments where conducted inside of the institute gymnasium.

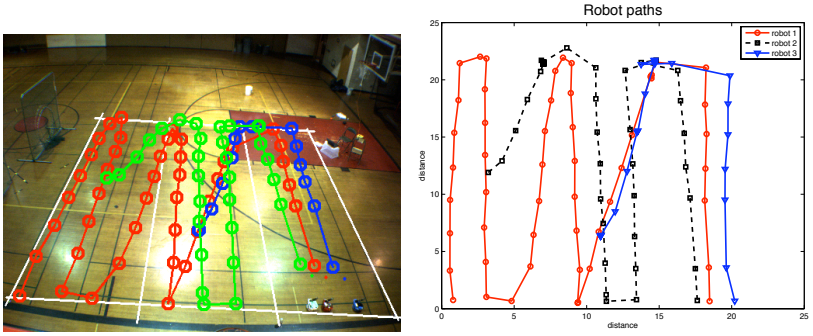
In practice, it would be useful to use wireless connectivity and connection strength to determine when an inactive node has been detected. However, in this setup the nodes would have to be

separated by very large distances before observing a noticeable decline in signal strength. Therefore, in order to demonstrate the algorithm in the restricted space, restrictions in communication were simulated by allowing inactive nodes to be discovered only when they are within a short range. Figure 2.10 shows the placement of the robots and their ideal strategies when executing Stripes. Figure 2.11(a) shows an image of the experimental setup from the external camera. The white lines superimposed on the image outline the workspace and stripe boundaries.Figure 2.11(b) shows actual robot trajectories during the experiment computed from the image to ground plane homography.



(a) Robot 1 (b) Robot 2 (c) Robot 3

Figure 2.10: The ideal trajectories for robots 1, 2 and 3 (2.10(a), 2.10(b), and 2.10(c) respectively). The stripes are denoted by alternating white and gray. The first robot starts at the lower left corner. The circle in Figure 2.10(a) is the meeting location with robot 2. The third robot is discovered by robot 2. The meeting location is shown in Figure 2.10(b)



(a) paths (b) trajectory

Figure 2.11: 2.11(a): This image of the experimental setup shows the environment divided into three stripes. The paths traversed by each robot are shown in red, green and blue (lines superimposed). 2.11(b): This image shows the actual trajectory of each robot computed using the homography between the image and ground planes.

2.5.1 Results

In the experiment, each stripe is selected so that it can be covered by a single robot in 24 time units. This corresponds to three up and down motions (Figure 2.10). The initial placement of the robots is shown by the start of each of the three paths in Figure 2.11(a). Robots move approximately 1 meter per time unit. The total area of the workspace is approximately 56 m^2. The stripes are completed in 24, 16, and 8 time units respectively. Hence, the total completion time is 48 units. Note that the total coverage time for a single robot is 72 time units. The measured path lengths of each robot are 41.08, 25.848, and 13.26 meters respectively. In this setup, even though odometry errors resulted in deviations from the ideal trajectories in Figure 2.10, they were not significant enough to prevent proper execution of Stripes.

2.6 Extension to General Convex Environments

As mentioned earlier, the time to cover the environment is a trivial upper-bound on the network formation time. This is because the undiscovered nodes are stationary, and by covering the environment, the first robot can guarantee that all nodes are discovered. In a convex environment, this coverage time is proportional to the area of the environment A. In case of a rectangular environment, it was shown that the Stripes algorithm performs much better than this upper bound.

In general environments, even when the environment is convex, the upper bound would be achieved. To see this, consider a long $1 \times A$ environment. In this environment, even when the robot locations are chosen randomly, the network formation time would be roughly A because the information is propagated sequentially and one of the robots is expected to be close to the "other" end of the environment.

Therefore, the upper bound of A can not be beaten in some general convex environments. The optimality of Stripes in the general convex case is currently being investigated.

2.7 Conclusion and Future Work

This chapter of the thesis introduced a novel network formation problem with ties to freeze-tag and coverage problems. In the network formation problem, a robot tries to propagate a piece of information to other robots with unknown locations as quickly as possible. Once a robot is discovered, it joins in the information propagation process by searching for other robots.

An algorithm for rectangular environments was presented and analytically proved that its performance is within a logarithmic factor of the optimal performance. The utility of the algorithm was further demonstrated with simulations and a proof-of-concept implementation.

In future work, network formation will be addressed in general environments, starting with general convex environments. It is believed that the Stripes algorithm can be modified to achieve good performance in such scenarios (compared to the optimal performance). Arbitrary environments, represented as polygons with holes, seem to be more challenging due to the lack of a natural way of partitioning the environment. This plan of study will be both an interesting and equally challenging problem in future work.

CHAPTER 3
Distributed Mobile Robotic System

3.1 Introduction

The systems component of my thesis is based on the algorithms developed in [14]. This paper presents and simulates motion planning algorithms that run on several mobile robots in a rectangular environment with obstacles. While these motion planning algorithms have many benefits and applications, their performance and feasibility have only been tested in a simulated environment. As with all algorithms and applications, their contributions and benefits are only fully realized if they can be implemented in an actual usable system. In addition, many assumptions are often made during the analysis which are also overlooked in a computer simulation, such as controlling a robot. In a simulation, it is very easy to control a robot since it is a simple matter of changing values within the application. This becomes non-trivial in a real mobile robotic system where these values are stored on separate robots, which are often times outside of direct communication range.

My contribution was to develop physical mobile robotic system and implement the motion planning algorithms from [14]. Using this system, several experiments were ran to test the practically and performance of these algorithms, as well as provide real world experimental data and results. The resulting robotic system was then extended to provide a more generic framework for future algorithm implementations and distributed mobile robot experiments.

3.1.1 Robotic Routers Overview

The situation described in [14] is one where a wireless network connection must be maintained between a single mobile robot, referred to as the *user*, and an immobile base station. If the user and base station are outside of each others' wireless communication range, then additional mobile robots can be deployed so that their wireless radios can be used as a makeshift network bridge between the base and user. These mobile robots are call *mobile routers* since their network bridge acts as a chain to relay all of the messages between the user and base, effectively connecting them to each other. If the user decides to move, then these mobile routers must adapt and be able to redistribute themselves to maintain the user-base network connection. The contribution of [14] are algorithms to automatically and dynamically redeploy the mobile robots so that the user-base network connection can be kept for as long as possible. The ultimate goal of the application is shown in Figure 3.1.

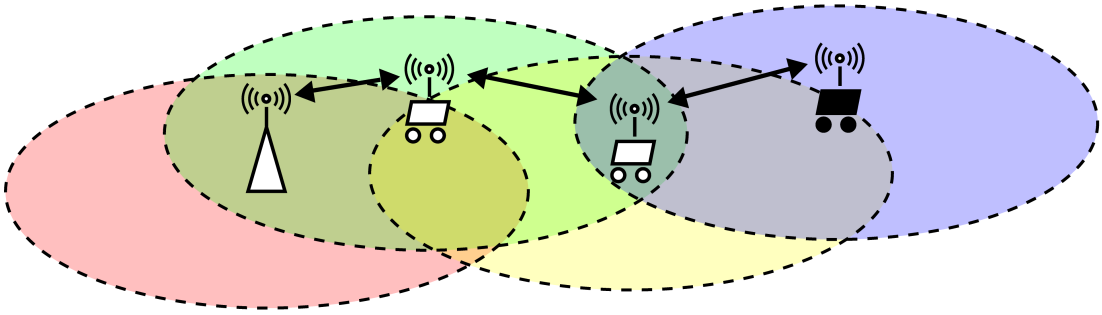


Figure 3.1: This shows the ultimate goal of the algorithms from [14] and the implementation from this thesis. The base station (triangle) needs to connect to the user (black car) but they are outside of each other's communication range (dashed ellipses). However, through the use of mobile routers (white cars), a multi-hop wireless path (arrows) can be formed between them, allowing the user and base station to communicate.

3.1.2 User Motion Models

In the robotic routers paper, there are two different motion models to describes how the user behaves. The first model is a *known user trajectory model* in which the user trajectory in the environment is known beforehand. In this case, the locations of the mobile routers can be calculated for each step of the user before the user even starts moving. The second model is a *adversarial user trajectory model* in which the user tries to break the connectivity as quickly as possible. The reasoning is that if the routers can maintain connectivity in this model, then they can maintain a connection for any possible user trajectory. However, for the actual experimentation, a human is used to control the robot and the overall goal is to simply maintain connectivity without knowing the user trajectory beforehand. Therefore, the second motion model is replaced with a more generic *unknown user trajectory model*.

3.1.3 Experiments

Several experiments were conducted, both in simulation and in the real world implementation, using the two motion models to verify and measure the performance of the motion planning.

For the known trajectory model, before the user even starts moving, the mobile router locations for each time step are generated for the given user trajectory. In the simulation experiment, the user and mobile routers moved to their respective locations at the same time and connectivity is checked at all times to ensure a user to base connection is maintained. The real world experiment

is identical to the simulated experiment only using physical robots in the actual environment.

In the unknown trajectory model, the mobile routers must respond to the user's movements at each time step. Their goal is to find the move that has the potential of maintaining the user to base connectivity the longest. In the simulation experiment, the user moves in an adversarial fashion to represent the worst case scenario for the mobile routers. For the actual real world experiment, the user is a human with a laptop computer who can move to wherever he chooses. In both cases, the test is to see whether the mobile robots can dynamically and correctly respond to the user's movements.

All of these simulations and experiments used an identical rectangular environment with two rectangle obstacles as shown in Figure 3.2.

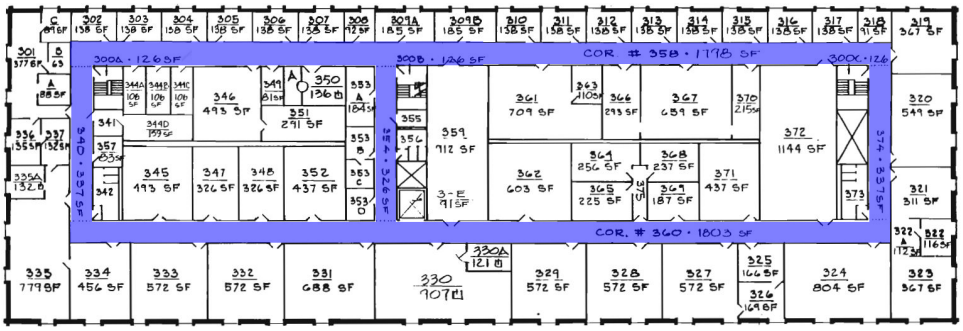


Figure 3.2: An overhead view of the environment for both the simulated experiments and real world implementation experiments. The shaded area represents the possible areas the robots can move in.

3.1.3.1 Simulation

In simulation, the user, mobile routers and base station are all modeled as identical robots with a limited communication range. The continuous environment is discretized so that all of the robots are restricted to a finite set of point locations as seen in Figure 3.3. This allows analysis to be feasible without sacrificing accuracy since these locations are dense enough to represent the physical properties of the surrounding areas. From these locations, a connectivity graph is set up to represent whether robots at each pair of distinct locations are able to communicate with each other. Movement is allowed only between locations that are adjacent to each other and two robots may reside at the same location at the same time.

The simulated experiment runs on a discrete time step system that synchronizes the robots' movements and motion planning calculations. All of the robots are only allowed to move from one location to an adjacent possible location in one time step and their next moves are calculated

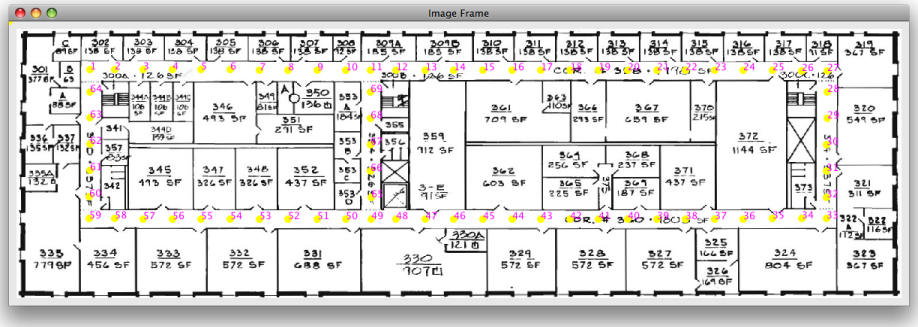


Figure 3.3: An overhead view of the environment for both the simulated experiments and real world implementation experiments showing the discrete possible robot locations as numbered circles.

at the end of each time step. In the known trajectory model, all of the robots move at the same time since the mobile router locations have already been calculated before the start of the experiment. For the adversarial model, the user movement, motion planning calculations for the mobile routers and final mobile router movements are all contained in one time step.

3.1.3.2 Real World

In the real world implementation, the same environment is used with the user and mobile routers all implemented on identical mobile robots as seen in Figure 3.4 for the known user trajectory. In the unknown user trajectory experiment, a human using a laptop computer acts as the user. A personal computer served as the base station where the algorithm processing and robot control takes place. All of these devices contain wireless radios which allows them to communicate with each other using the same connectivity model as the simulation.

While the computer simulation of the robotic routers algorithm is fairly discrete, the physical implementation is in a continuous environment where the robots can move to an infinite number of possible locations in a various amounts of time. In order to still utilize the discrete motion planning algorithm, the original finite set of possible locations are still used but now they are treated as target locations that the robots need to move towards, as opposed to being the only places a robot can reside. This simply means that in each time step, a robot has to move to within a certain distance of one of these locations for it to count as being there. This gives the robots the freedom of moving between the points in any means necessary, taking as long as needed, with the algorithm only proceeding when they are close enough to their target locations. In the unknown user trajectory case, the human user was restrict to only moving between the target locations. The

time steps can be thought of as steps of the experiment used to synchronized the calculations and not necessary as the time it takes for each robot to move. This whole process is aimed at providing a way of using a discrete algorithm in a continuous environment.

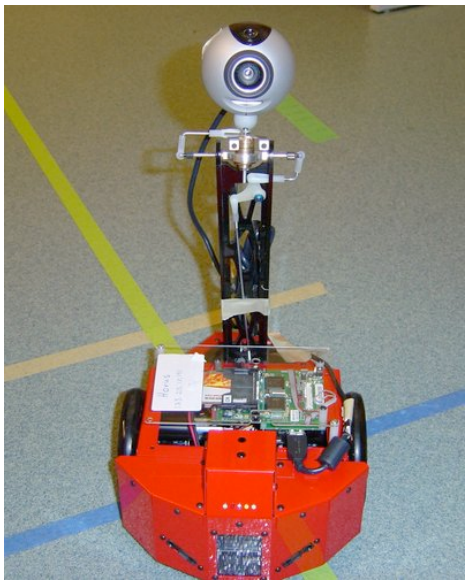


Figure 3.4: One of the three Acroname Garcia robots used in the implementation

3.1.4 Challenges

When an algorithm is developed in a computer simulated environment, a lot of assumptions are made to simplify the overall process, especially if they do not pertain to the problem at hand. An excellent example of this is how to control the robots.

3.1.4.1 Route Finding

The algorithms presented in this paper are centralized, in the sense that, a single algorithm decides on the motion of all robots and is run on a single computer (which also acts as the base station). However, due to the dynamic nature of the system (especially in the adversarial case), the base-station must send motion commands to the robots. The base station has direct access to only the robots that are within its communication range. Therefore, to control a robot outside of this range, the base station must send a message through several intermediate robots to the desired robot since there is no global wireless coverage. The problem then becomes a matter of determining which robots to route the message through. This can be increasingly complex as the robots are constantly moving, causing one route to work at the current time step but not work at the next.

Two solutions to this problem were implemented: one which uses broadcasting and Dijkstra's algorithm and one which uses the properties of the physical environment. The first solution involves using a broadcasting or flooding mechanism to retrieve all of the robots' locations, then generating a connectivity graph with this information. With this graph and a shortest path algorithm, such as Dijkstra's algorithm, valid paths can be calculated to all of the nodes starting from the base station. This solution is explained in detail in Section 3.2.4. The second solution is to use the broadcasted messages themselves as a way of finding paths. This is done by having each message track the nodes that it passes through on its way to the final destination. When the final node receives this message, it will know exactly which path the message took and can send a reply back along the same path. This solution is explained in Section 3.2.7.

3.1.4.2 Localization

Another problem that comes up is how to accurately maintain the positions of the robots. In the simulation, the location information for all robots is stored on a single computer so it can be easily and accurately accessed by the motion planning algorithm. In the real world, each robot has its own odometer and this information has to be propagated to the base station at each time step, regardless of how the robots are organized and who can communicate with each other.

Two techniques were developed to solve this challenge. The first technique is to still store the robots' positions on the base station and only send the actual move commands to the robots. The second technique is to store the positions on the robots but use a broadcasting technique to retrieve this information when needed. A detailed discussion of this problem and its solutions are presented in Section 3.2.5.

3.1.4.3 Connectivity

In the simulation, the environment is idealized in that there is a clear cutoff between when two robots can communicate with each other given a particular connectivity model. However, this is rarely the case in a physical environment where communication channels can routinely work one minute but fail the next, especially if the robots are at the limits of their communication range. Obstacles in the simulated environment always have a predictable effect on communication devices but in reality, they often have an unpredictable dampening effect on wireless signals. The connectivity issues that are faced by the system are explained in detail in Section 3.2.3. Section 3.2.8 explains how intermittent network connections can cause problems on the overall system.

3.2 Systems

3.2.1 Overview

The physical components used in this robotic routers system comprise of several identical mobile robots (Figure 3.4) used for the user and mobile routers, and a personal laptop computer as the base station. All of these devices, which will also be referred to as *nodes* since they reside in the same network, use range limited wireless radios with identical capabilities to communicate with one another. The robots are allowed to move in any direction at a constant speed with the laptop computer always remaining stationary. The robotic router algorithm from [14] and all of the necessary processing is centralized at the base station where the robot tracking and movement commands are also conducted. Each of the robots, as well as the base station, run custom networking applications so that messages can be routed throughout the network in both a centralized and distributed manner.

Although this system can run in any environment, the one used is the rectangular environment with the two rectangle obstacles as shown in Figure 3.9. The overall system can support both of the real world user motion models: the *known trajectory model* and the *adversarial trajectory model*. In the known trajectory model, the system moves all of the robots at the same time in accordance with the pre-computed trajectories based on the known user trajectory. In the adversarial trajectory model, the user is a human walking around with a laptop computer. The user's trajectory is not known to the system (but user reports its next location before each movement). The system is responsible for using the algorithm from [14] to move the mobile robots to the best location in order to maintain the longest connectivity duration.

The implementation presented in this paper can be used as a generic framework to test other algorithms in the future. The various applications created can easily be extended to expand on existing functionality.

3.2.2 Ad-hoc Network

All of the nodes are equipped with wireless radios that allow for communication and data transfer. When two nodes are in communication range of each other, a data connection can be established between them. This allows the robots to transfer routing data, commands and other information. Since this system is designed to provide and maintain its own wireless coverage, all of the robots are only allowed to communicate in an ad-hoc manner between pairs of nodes. However, each node can maintain multiple connections to several different nodes at the same time.

The IEEE 802.11b wireless standard used only allows nodes to exchange data if they are

directly connected. It does not provide any built-in functionality for establishing multi-hop connections between disconnected nodes by routing data through intermediate nodes. This means that in order to establish a connection between the user and base station through the mobile routers, there needs to be a routing system capable of routing data throughout the system.

Connections, Packets and Messages: The framework of the distributed routing system was created by having all of the robots run identical programs that contain server/client networking components. These programs allow networking connections to be made between pairs of nodes and *data packets* to be transferred between them. If the user and base station are not within communication range of each other, then using this mechanism, data can be sent between them by creating connections to the mobile routers. The mobile routers can then create connections between each other as necessary in order to form a multi-hop path, thereby allowing data to be sent between the user and base station.

When transferring a data packet between two nodes, an underlying transmission protocol, TCP, is used. TCP is one of the standard internet protocols for transmitting data and is used to transfer everything from web pages to e-mails and even files. This protocol transfers data by first creating a connection between two nodes, transmitting the data, then closing this connection. UDP, an alternate method of sending data using segmented connectionless packets, is described in Section 3.3.2.1 where these two protocols are contrasted. However, regardless of the transmission protocol used, The data packets themselves are formatted as short unidirectional messages sent one at a time containing commands for a robot to execute. More details on the actual format of these data packets can be found in Section 3.3.3

As mentioned earlier, when two nodes communicate with each other, the total data being sent between them is referred to as a *data packet*. The actual command inside of a packet, such as the encoding for "turn left", is referred to as a *message*. Essentially, a packet is the combined data from a command along with any administrative data that is required in order to successfully complete a data transfer from one node to the next. This administrative data usually consists routing data, characters to signal the end of a data packet or characters to separate different command parameters.

3.2.3 Connectivity

The connectivity model used in the simulations is more conservative than the actual physical connectivity. The 802.11b wireless radio used has an estimated range of 38 meters or 124 feet but obstacles, such as the rectangular rooms in the middle of our environment, can reduce this

range or even completely absorb the signal. To model connectivity, an analytical mathematical model was created in [14] where connectivity was based on distance with penalties given to connections made through obstacles. This connectivity model is used as the underlying structure in both the simulation and implemented systems in order to determine if two nodes can talk to each other from their current positions. Using the same connectivity model maximizes the accuracy of demonstrating an implementation of the complete robotic routers algorithm.

With this connectivity model, the real world system will use a mathematical look up table to determine if two robots are allowed to communicate with each other instead of basing it on the physical environment. The implication is that this method creates a discrepancy between the network topology the system thinks it has and the real physical network topology. In a scenario where the mathematical connectivity table is more conservative, this does not present a problem as it can be justified by allowing robots to only communicate when the wireless signal quality is above a very high threshold. The downside is that existing physical connections that could be used are not utilized and this could lead to inefficient networks. However, this does not cause major connectivity problems.

If the mathematical model does not take into account some part of the environment and ends up being more relaxed than the physical environment, then major problems can occur since the system expects two robots to be able to talk to each other when in reality, they cannot.

While it may seem that this problem can easily be solved by simply basing the connectivity matrix on the physical environment, the reality is that this is not a complete solution. The problem is the standard assumption of a binary model where nodes either can or cannot talk to each other. However, in a real world environment, there are often various shares of grey where some messages periodically may get through while other messages are lost. Sometimes, only parts of a message will be transmitted while other times, they can arrive in a different order with long delays. As a result, it can often be very hard to build an accurate network topology because one network link might exist one second and disappear the next. These unstable network connections and their consequences are explained more in detail in Section 3.2.8. *The implemented robotic router system uses the mathematical model to determine connectivity but also gives users the option of determining the physical network topology themselves using a broadcast based path finding technique (Section 3.2.7).*

3.2.4 Routing

In order to route packets between the user and base, a path through the network must first be established for the given network topology. Once the base knows the locations of all of the nodes, it can create a network topology or graph by using the connectivity model to determine which nodes are close enough to communicate with each other. A shortest path algorithm, such as Dijkstra's algorithm, can be run on this graph to create the shortest routes from the base station to all of the nodes. From here, any information that needs to be transmitted, such as movement commands, can be sent through these paths to the desired nodes. If any of the nodes move, the shortest paths must be recalculated as the network topology could have changed. All of these calculations are done on the base node and when sending packets to a particular node, the transmitted packets carry a header specifying each node in its path to the destination. The mobile routers are only responsible for routing the packets they receive and occasionally sending a list of its visible neighbor nodes, making this implementation a centralized routing system.

3.2.5 Network Localization: Centralized vs. Decentralized

One of the assumptions of the system is that the base station knows the location of all of the nodes at all times. This can be justified since the mobile robots are controlled by the base, their locations must be known at all times. For the user, since it is using services provided by the system, it is reasonable to assume that the user must provide its location to the base station in return. However, in implementation, this proves to be more difficult as there must be a way of maintaining each nodes' location at each time step. This system assumes that each node is able to keep track of its own location, either with perfect odometer from a given starting position or a global positioning type system, but does not make any assumptions in where and how this information is stored or delivered to the base station for route discovery and path finding.

The implementation of the system allows for two different approaches to maintaining each nodes' location. The first approach simply stores the position and orientation of each of the nodes on the base station itself. Initially, the starting locations of the nodes are set in the base station and with each mobile node movement, the system updates the local position values of that node. Ideally, with a perfect network and no outside interferences, the nodes location kept on the base station would be identical to its position as kept on the node itself and thus, its real location. This is the simpler and more efficient of the two approaches as only move commands are passed between the base station and the mobile robots through the network.

The second approach can be thought of as a more distributed system where each node is

responsible for keeping track of its own location and orientation and the base station must poll the nodes for their latest positions before each time step. Storing the location of each node on itself has several advantages over storing this information on the base station:

1. It is more accurate because all of the location data is updated at each time step right before calculations are done.

2. It is more flexible because the robots do not have to update the base station whenever minor adjustments are made to its position between time steps.

3. It is more scalable because it allows other parties to control robots as necessary and prevent errors from out of sync data. One example of this is that multiple bases can now control the same set of robots.

4. It is less error prone because there is a much tighter connection between the robot's location information, which is stored on the robot, and its physical movement mechanism, also located on the robot. This causes less problems then if this information is kept separate given the greater likelihood of having unstable network connections.

Keeping the location data stored on the robots removes the discrepancy that might occur between the robot position values on the base station and the actual robot location. However, the advantage of this method also contributes to its disadvantage as well. Updating each node's location at every time step allows for a more accurate system but this also increases the amount of network traffic on the network, especially if broadcasting (Section 3.2.6) is used.

One simple way of combining the advantages of these two approaches is to use both of them at the same time. By keeping a copy of each nodes' position on the base station and periodically polling each node, a compensation can be easily made between the efficiency of the first method and the accuracy and flexibility of the second one.

3.2.6 Broadcasting: Decentralized Network Localization

In the second localization approach, the base station must poll all of the nodes for their locations without knowing the network topology and which nodes can communicate with each other. The approach used to solve this problem is to broadcast a message throughout the network, asking each robot to report its location to the base station. This can also be thought of as flooding the entire network with a single message. This is implemented in the intuitive way by enclosing this message in a packet and having each node forward this packet to all of its neighbors. Each

broadcast message is also assigned a unique ID and all packets carrying this message would use the same ID in their headers. Each node would only process and transmit the first packet it receives for a particular ID and would proceed to ignore any further packets of the same ID, thereby preventing a situation where the same packet is transmitted throughout the network indefinitely. The broadcast message ID must be a globally unique ID. One simple approach is to use a combination of the initial broadcasting node's ID and local time.

3.2.7 Broadcast Based Path Finding

After a particular node receives a broadcasted message, such as "send your location to the base station", it must be able to figure out how to reply back if necessary. Since the node also does not know the topology of the network, it faces the problem of determining how to get a message to a particular node, in this case back to the base station. Depending on the stability of the network, this can be accomplished in two ways. The first way is to simply reply back using the same broadcast mechanism as discussed in Section 3.2.6. This is a fairly inefficient method since it creates a lot of network traffic with sending even a single message but there are times when this is the only choice, as explained in Section 3.2.8. Another method is to send the message through the same exact path the broadcasted message took to arrive at the node. While the broadcasted message may not have taken the shortest path, sending a message along a single path is far more efficient than flooding a network with it.

The broadcast packets can be modified so that each node adds its unique ID to the footer of any packet that it will retransmit to its neighbors. Through this, whenever a node receives a broadcast packet, it can extract the exact path the packet took to reach that particular node. An application of this is when the base station must retrieve the location of all of the nodes in the network. First, the base station sends out a broadcast message throughout the network, which will be received by every node, telling them to reply back with their locations. Each node can then use the path encoded in each of the broadcasted packet to send its location back to the base station. Even though neither the base station nor any of the nodes know the overall network topology, all of the necessary information is transmitted to the desired recipients.

3.2.8 Unstable Network Environments

The methods described in Sections 3.2.4, 3.2.5, 3.2.6 and 3.2.7 primarily deal with an environment where the network topology is ideal and stable. This means that physical connectivity can be accurately predicted based on a nodes' location with respect to the obstacles and distance to neighboring nodes. However, in reality, there are often times when this does not hold true.

This is one of the reasons why a more conservative connectivity model was created to use for path and route calculations (Section 3.2.3) even though the system is capable of using properties of the physical environment (Sections 3.2.6 and 3.2.7). In these cases, sometimes nodes can talk to each other to pass a message but at the next second, the connection breaks even when the environment does not change. Other times, a message might reach a node after a long delay, well after shortest path calculations are done. These intermittent network connections can cause major problems for broadcasting and path finding functions which must depend on a reliable and consistent distinction between whether two nodes can or cannot talk to each other.

In broadcasting and path finding, each node only processes the first broadcast message it receives and uses that message's path to reply back to the initial broadcasting node as needed. However, if the network connections are unstable, a situation can occur where a message makes it through to a node over a weak connection. Since the node will discard any other packets with the same message ID, it will ignore a packet which arrived through a longer but stronger path through the network. If the initial weak connection can no longer support data transfer, then the node is stuck in a situation where its only reply path does not work and it is too late to extract paths from any of the duplicate packets. One potential method of solving this problem is to have every single communication sent on the network be sent through the broadcasting method. While this method is highly inefficient, it is more reliable in an environment where physical network connectivity is very unstable. Another potential method is to store all of the paths from all of the duplicate broadcast messages that arrive at each node and try them one at a time if any fail. However, this introduces the problem of how to determine if a path is broken and when to try the next path.

These problems are some of the ones which still need to be resolved and accounted for by the overall system. Our current solution essentially relies on creating a stable network within a network containing unstable network connections. While these particular problems and potential solutions are not specifically addressed in this implementation, the current system does provide the underlying mechanism to develop advanced techniques to overcome these obstacles.

3.3 Implementation

3.3.1 Overview

The system, as described in Section 3.2, was successfully implemented using mobile robots and personal laptop computers. All of these devices used standard wireless network adapters running in ad-hoc mode to form a wireless network. The same environment that [14] used in simulation was used to test the implemented system and run the experiments. Custom applications

were developed and deployed onto all of the devices to add the network routing and message broadcasting functionalities (Sections 3.2.4 and 3.2.6) since they were not provided by the inherent underlying wireless systems.

As defined previously, the term *node* is used to refer to the individual devices that make up the overall network, whether it is the user, a mobile router or the base station. If a node runs an instance of the custom routing application, then they can be labeled as a *host* because they are now hosting the networking components essential to the functionality of the system. Since all of the nodes in the network must also run this networking application, the terms *node* and *host* can be used interchangeably when referring to devices in the network.

3.3.1.1 Hardware

For the implementation and experiments, Acroname Garcia mobile robots equipped with ARM/risk PCs running Linux (Figure 3.4) were used to represent the two mobile routers and the user in the known user trajectory case. In the unknown user trajectory case, the user had a personal laptop running Ubuntu. An Apple Macbook personal laptop running Mac OS X was used as the base station, which also held the centralize network control application. All of these devices used 802.11b wireless adapters to establish TCP network connections in order to transmit data back and forth. The robots were programmed to accept move commands using primitives and can move forwards and backwards at a constant velocity, as well as rotate in place. The user was allowed to make any possible moves but must update his new location on his computer. The laptop was kept stationary at the designated base station location.

3.3.1.2 Software

A custom C/C++ application named Mercury (Section 3.4.2) was created to run on the robots and user laptop to add routing and broadcasting functionality while a separate Java application called Jupiter (Section 3.4.3) ran on the base station to control the overall network. Jupiter provides the base station with the same capabilities as Mercury in addition to the abilities of route calculations and path finding functions. Since it is used to control the network, it also has a graphical user interface to visualize the location of the nodes within the environment (Figure 3.7). Another Java module, which was later integrated into Jupiter, was created to step through the experiments from [14] and direct each robot to their target locations. This module is titled Neptune and is explained in Section 3.4.4. Although these applications were written in different languages, they are able to communicate with each other by transmitting data using standard TCP/IP sockets in a client and server programming model.

3.3.1.3 Client and Server

The client and server programming model is simply a nomenclature used to assign names based on the roles that devices take when sending information. As used in this thesis, the term *client* refers to the device that first initiates a connection and the *server* is the device that waits for an incoming connection. In cases where information is sent without first establishing a connection, the client is the sender and the server is the receiver. If a device can accept an incoming connections as well as initiate outgoing connections with other devices, then its individual networking components are named based on their functionalities. Hence, that device would have a server component to handle connections and a client component to initiate connections.

3.3.2 Client, Server and TCP Connections

The two main programs, Mercury and Jupiter, contain both the client and server components. This allows them to initiate connections with neighboring nodes as well as listen for any neighbors who would want to connect with them. The server components of programs are also multi-threaded so they can open multiple connections at the same time if necessary.

3.3.2.1 TCP/IP and Sockets

Once two nodes are close enough and there is a strong wireless connection between them, they can start establishing network connections and transmitting data to each other. The Mercury and Jupiter applications run on top of the operating system's network stack in the user space of their respective hosts. They can be thought of as user applications, such as a web browser, which uses the underlying operating system to take care of the actual networking functions. Currently, both of these applications use the TCP internet protocol where a connection between the two hosts must be established first before data can be transfered. The alternative would be to use the UDP internet protocol where data can be sent in segmented packets without the need to first create connections. A comparison between the two protocols is provided in Section 3.3.2.1 and future work with implementing UDP is explained in Section 3.6.2.

TCP vs UDP For transmitting data between two network devices using the standard TCP/IP data model, there are two widely used protocols, TCP and UDP.

TCP is a stream oriented connection protocol where two devices transfer data by forming and maintaining a direct network connection between them. This connection model favors reliability over timely delivery and is the standard transmission method for the applications such as the World Wide Web, E-mail, File Transfer Protocol and Secure Shell. The TCP network connection

is reliable in that any missing or timed out data packets will be resent as needed and any successfully received data is acknowledged. From the view of an application, data is sent and received as a continuous stream and is guaranteed to arrive exactly as it is sent. Once a TCP connection is established, data can be transfered in either direction from both sides, regardless of which party first initiated the connection. The basic concept of TCP connections can be roughly thought of as calling another person and speaking to them on a telephone. Anything that one party says into the phone will come out in the same exact way on the other end and both parties can speak at the same time without breaks in the service.

UDP is a packet or message based protocol and focuses on a lightweight timely delivery mechanism instead of the heavier but reliable TCP connection. It's applications include the Domain Name System (DNS), streaming media such as Voice over IP (VoIP) and online games. Instead of forming direct connections to stream data as TCP does, data is segmented and sent as individual packets in one direction from one host to another. There is no ordering or reliability mechanism so the data packets can arrive in any order or possibly not arrive at all. There is no acknowledgment or retransmission of packets so once a host sends out data, it has no way of knowing if it ever arrived at the destination. UDP is a more lightweight protocol compared to TCP because there is no overhead for the reliability and ordering mechanisms. Whereas TCP is analogous to a telephone call, the basic UDP concept can be thought of as mailing several letters using the postal service. The information sent in the letters travel only in one direction and there is no guarantee to which letters will actually arrive at the destination and their arrival order.

Choice of TCP TCP was first used instead of UDP as a way of easily determining which hosts can talk to each other. TCP connections are only established if hosts can communicate with each other, otherwise the operating system will alert the user application of a failed connection. Since all of the nodes are assumed to be on and running the necessary routing programs, if a connection attempt failed, then that means the specified node is not reachable and outside of communication range. However, if a connection can be made, then that node is reachable and thus, within communication distance. Therefore, an easy way of determining which nodes are reachable is to simply go through a list containing the addresses of all of the nodes in the network and attempt to connect to every single one of them. Since a TCP connection attempt will either succeed or fail almost immediately, when the node reaches the end of the list, it will know precisely which nodes in the network are in communication distance. In contrast, if UDP was used, then the procedure would be more complicated and prone to error. A node would have to send a UDP packet to each of the nodes and any node in range would have to reply back with another UDP

packet or through a TCP connection. Since UDP packets are sent without acknowledgments, the initial node would have to send out all the UDP packets at the same time and wait for each node's reply packet to come in or time out. Also, because there is no reliability mechanism, if a UDP packet is lost then a node could be mistakenly marked as out of range when it is actually reachable.

As more functionalities and features were added to the applications, they were built on top of the TCP connection framework. However, some of these features, such as the broadcasting messages, could see a benefit if they were sent using UDP instead. Also, an UDP server could run along side of a TCP server at the same time using the same port without complications. These features and extensions are explained more in depth in Section 3.6.2.

3.3.2.2 TCP Client

The TCP client Application Programming Interface (API) for connecting to another host's server is very simple and requires only the network address and port number of the destination. In C/C++, a socket is first opened to specifically support TCP connections. This socket is then passed into a `connect` function, along with the network address and port number of the remote server. If the return value of the `connect` function indicates a successful connection, then the socket becomes valid and is open for reading and writing data. Otherwise, the socket is closed and it is concluded that the destination node cannot be reached from the current node. In Java, the same procedure occurs only objects are used to represent the sockets, input and output streams to follow Java's Object Oriented Programming language model. Code for the client interfaces can be found in Appendix B.1 in C and Appendix B.2 in Java.

3.3.2.3 TCP Server

A similar procedure is used to set up a TCP server and like the TCP client, the overall process is almost identical between C/C++ and Java. In C/C++, a socket is created and a `bind` function is called to bind this socket to a particular port number on the server host. Then, a function is called which places this socket in a listening mode and the current server thread sleeps while waiting for a client to connect. Once a client has connected, an `accept` function returns a socket that represents the server's end of the new TCP connection with the client. This socket can then be used to send or receive any data from the client. In Java, the same process occurs only using Java objects instead of procedural C method calls. Code for the server interfaces can be found in Appendix B.3 in C and Appendix B.4 in Java.

Since the thread that runs the TCP server sleeps while waiting for a client to connect, this essentially puts the application in a state where it can only perform tasks that are in response to

commands from remote clients. To allow the application to conduct other tasks, the server component must run in its own thread. This way, an application can wait to accept a client connection while still commanding a robot to move or even connecting to another node simultaneously. Additionally, a server can also start its own thread to further improve performance. Whenever a client connects to a server, instead of having the original server thread handle the connection, the server can create a child thread and hand off the task of processing the client connection to this new thread. This allows each application to handle multiple connections at the same time, greatly increasing its efficiency and capacity for multi-tasking and expansion. This also allows the human controller to enter commands into Mercury through the terminal or the graphical user interface of Jupiter without disrupting their background network routing and broadcasting responsibilities. Both of the C/C++ and Java servers follow this approach.

3.3.3 Packet Format

Whenever information is transfered, it must be encoded in a format that is standard throughout the entire system. In this implementation, commands are called *messages* and encoded into *packets* before they are sent to hosts.

Each *message* is defined as a string of characters, numbers and symbols that commands a host to perform a particular action. Messages can range from telling the robot to move one step forward, have it broadcast its own location or even print a list of received message ID's. In order to transmit these messages, headers and footers must be added to include information such as which host to route the message through or the broadcast ID. After this information has been added to a particular message, the combined data is sent through an established network connection in the case of using the TCP protocol. This data that will be ultimately transmitted, including headers and footers, will be referred to as a *packet*.

3.3.3.1 Layering

The structure of the packets and the workflow for how to create and format packets from messages is based on the multi-layered philosophy used in the TCP/IP and OSI network models. A figure of the overall message can be seen in Figure 3.5.

The first and innermost layer consists of the actual message itself. The core of this message is comprised of a command string and any necessary parameters, all separated by a special character (`#`). An example of a command would be `crc` with the parameters of `2` and `5.0` which would tell a robot to turn five degrees counter-clockwise. This command would be encoded into a message represented as `crc#2#5.0`. A full table of commands is given in Appendix A.

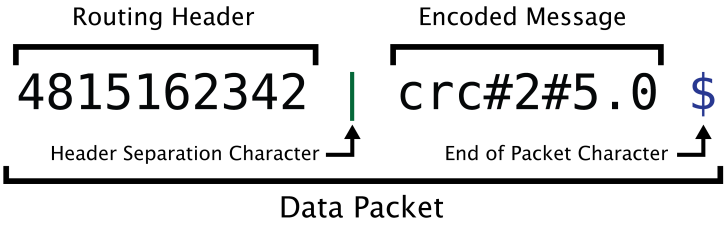


Figure 3.5: The format for a data packet. The encoded message (a move command) with separation characters is in the middle of the data packet. Before the message is the routing header, which in this case is a broadcast ID. At the very end is an end of data packet character.

The second layer of processing involves adding routing information as a message header. For messages that are routed, the header is a series of alphabetical characters is used to designate the ID's of all of the nodes along the desired path through the network. Continuing with the previous example, if the control robot command has to be routed through nodes with ID's of 3, 2 and 5, the second layer would add `CBE` to the message, making the packet: `CBE|crc#2#5.0`. To keep track of how far a packet has progressed in its path, the characters are converted from upper case to lower case to mark off the nodes that have processed the packet. This also marks the node the message needs to be routed to. When the example message arrives at node 2, it will look like `cbE|crc#2#5.0`. The `E` is extracted and converted to lowercase as node 2 will try to send the packet (`cbe|crc#2#5.0`) to the final destination of node 5. For messages to be broadcasted, the header becomes a globally unique numeric ID (e.g. `4815162342|crc#2#5.0`). This ID can be a random number with a large enough range to prevent repeating or a number created using the source node's ID and time stamp. This allows nodes to discard messages with ID's that have been processed already, ensuring that the same message will not be sent around the network forever. This header is separated from the command by another separation character (`|`) that is different from the one used between commands and parameters. This approach greatly simplifies the task of routing, broadcasting and processing packets because a node only needs to process the routing header to determine the next step it should take.

The final and outermost layer of the overall data sending and receiving process involves adding a special character at the end of each packet to signify the end of the data. This character (`$`) is also used as a signal to tell the operating system when to send the transmission buffer through the network and when to return the receiving data buffer to the user application. Using the previous example, the entire data that is received at node 5 would be: `cbe|crc#2#5.0$`. Without this character, any data that is sent or received could be kept in the system buffers for long periods

of time before being processed. These characters are added and removed at the lowest level of processing, which is completely transparent to the layers above.

3.3.3.2 Sending a Message

Sending a message is done very simply by first creating a message in the first layer, passing it down to the second layer where the routing information is added and finally, sending it with the end of packet character added in the last layer. If the message is to be broadcasted, a new broadcast ID is generated and added as the routing information in the second layer. If the message is designated to be sent to a single node and requires routing, then the process is a little more complicated. Whenever the path finding (Section 3.2.7) or the route discovery algorithm (Sections 3.2.4) runs, the resulting paths are stored for each node after they are calculated. If the final destination node has one of these paths then it is used as the routing information in the second layer. However, if there is no path stored, then application attempts to send the message directly to the final destination node without any routing information.

3.3.3.3 Receiving a Packet

When a packet arrives at a node, the first job is to determine whether this packet should be broadcasted to the node's neighbors, routed to the next node in the packet path or if it should be processed. Since routing information always use alphabetical characters and broadcast ID's only contain numeric characters, a simple test of the routing layer substring will easily determine what to do with the packet. If it is a broadcast packet and that particular broadcast ID hasn't been received yet, then everything is kept the same and the packet is sent to all of the node's neighbors. If the broadcast ID has been received before, then the packet is simply ignored. Should the packet contain routing information, then the next node in the packet's path is extracted and the packet is forwarded to that node. For any packets where there are no remaining nodes in the path or contain no routing information, then the receiving node assumes that the packet has reached its final destination and the message is extracted and processed.

3.4 Code

3.4.1 Overview

Two different applications were created to add the routing and broadcasting functionality to the networked robot system. The first application, Mercury, is written in C and C++ and runs on each of the Garcia robots and the user laptop in a Linux environment. It is the distributed workhorse of the system and is responsible for routing and broadcasting messages throughout the

network as well as actually moving and controlling the robots. The second application, Jupiter, was created as the framework for controlling and interacting with the entire robotic network system. It is written in Java and runs on the personal laptop that acts as the base station. Although for this experiment it ran on Apple's Mac OS X operating system, Java is cross platform compatible so any platform can serve as a host for the Jupiter application. This application is where a human user would control the robots through its graphical user interface and also where the shortest path routes and network topology are calcaulated. A separate standalone Java module, Neptune, was later developed to implement the strategy used in both the known and unknown user trajectory cases by reading files and determining the user and mobile router locations at each time step. Later on, this module would be integrated into Jupiter so that it can take advantage of the already created network connection framework as detailed in Section 3.4.4.1.

3.4.1.1 Input Files

All of these applications rely on a plain text file that contains a list of every single node in the network and their respective network addresses. This file is read in the initialization portion of the Jupiter and Mercury applications and each node is assigned an unique integer ID to be used for routing and node identification. There are additional files which are used by each application and are explained in their respective sections.

3.4.2 Mercury

Mercury is a C/C++ application that runs on all of the mobile robots as the actual distributed networking framework responsible for routing and broadcasting messages throughout the network. It is also responsible for accepting and executing any robot move or turn commands. It has a command line interface where users can enter text commands to be executed sequentially one at a time, depicted in Figure 3.6 and Appendix A. A separate TCP server thread runs in the background, listening for any incoming network connections to be processed in newly created child threads.

Mercury is capable of storing the coordinates and heading of its host as well as retrieve this information from any other nodes. Currently, each node is able to obtain the status and location of all of the nodes that are up to two hops away and store this information in a visibility graph. Mercury also has the abilities to create, route and broadcast any messages using the techniques presented in Sections 3.2.4 and 3.2.6. Lastly, Mercury can receive robot control messages, such as moving a certain distance forward or turning a specific amount, and command the physical hardware to perform these actions. Any robot move actions would also change the internally

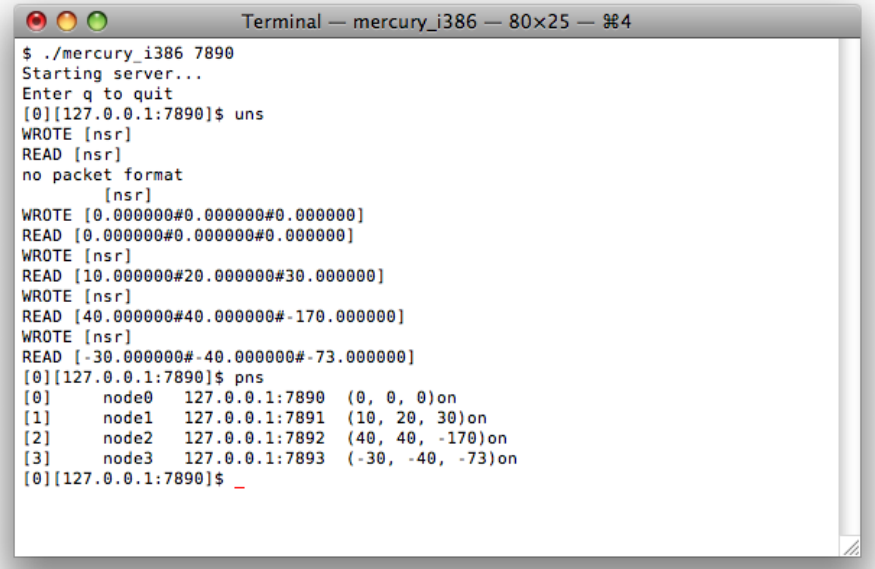


Figure 3.6: Mercury a terminal application that runs on the user laptop and on all of the mobile robots. It accepts commands from both its server component and its command line interface. For testing, several instances can ran at the same time on a single machine using different port numbers, as seen in the figure (Four instances running on ports 7890 to 7893).

stored location and heading variables. Additionally, these locally stored variables can be changed manually through the command line interface, as used in the unknown user trajectory experiment by the human user.

3.4.2.1 Classes and Cross Compiling

Mercury is primarily written in C++ with the code divided into several classes of distinct functionalities in an object oriented design. The multi-threading code and network code are C procedural functions, part of the POSIX standards.

Mercury is cross-compiled on the development platform using a compiler that produces machine code specific to the Garcia robot's ARM processor and runtime environment. When compiling this executable, a `RobotARM` class is used as an interface between the Mercury application and the already compiled libraries that control the physical movements of the robot. However, it is often easier to debug and test Mercury on the development platform so a class called `Roboti386` was created. This class contains all of the same function calls as the `RobotARM` class but does not contain any of the references to the robot control libraries since they are only compatible with the

ARM compiler. By using the `Roboti386` class in place of the `RobotARM` class and changing the server's binding port number, several instances of the Mercury application can be run at the same time on a single development platform, as seen in Figure 3.6. This also allows the Mercury application to run on nodes that are simply standard laptop computers.

3.4.3 Jupiter

This application is used to control the robots, develop the shortest paths in the network and display the locations, status and paths of all of nodes. In the experiments, it was used to visualize each node's location as well a visibility graph that shows which nodes could talk to each other as seen in the top image of Figure 3.7. While the Mercury application runs on each of the nodes and essentially forms the actual robotic network system, this application is the central control interface for the entire distributed system.

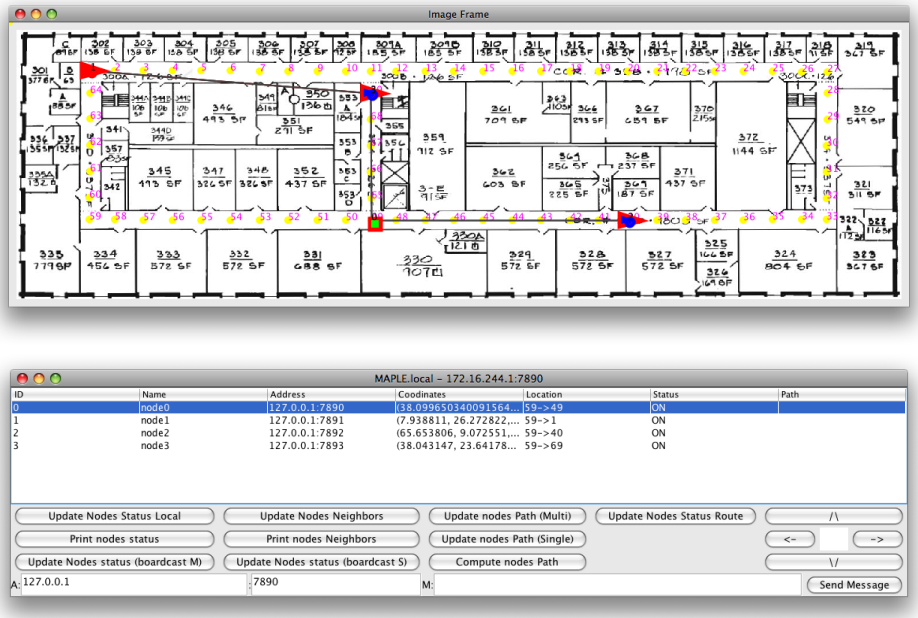


Figure 3.7: The Jupiter control application. It contains a frame with the floor plan of the environment that shows the current status and locations of the nodes. The control frame lists the nodes and their properties in a table as well as the various buttons to issue commands.

The Jupiter application is capable of retrieving the location and status of any node that it can talk to by connecting to their Mercury instance. Its networking component is also capable of creating, routing and broadcasting messages just like its C/C++ counterpart in Mercury. Jupiter is a multi-threaded application, with separate threads for the main server instance that creates new

child threads for incoming network connections.

3.4.3.1 Model-View-Controller Design (MVC)

To improve compatibility and future expansions, the Jupiter and Neptune applications were created following a Model-View-Controller (MVC) software design pattern. A design pattern is a development template or guideline that is used to help create better programs, both in functionality and developmental design. By following some of the commonly used software design patterns in the computer science industry, programs can be written more cleanly and structured so that different parts and modules can be reused, even in other applications. This saves precious time in the development and maintenance phase of the software life cycle. Since design patterns typically call for modular components and reusable classes and objects, they are used mostly for object oriented languages such as C++ and Java.

The Model-View-Controller is a design pattern that is used as an overall template for structuring an entire application. The pattern calls for a separation of an application into three components: the *model*, *view* and *controller*. These components are created independent of each other and interact with one another through common interfaces. One of the benefits of this is that it allows several different implementations of the same component to be used at the same time since they all must share the same interface. Another benefit is that it allows developers to create components in parallel once all of the common interfaces have been planned. An example of this is the Neptune module integration, which is explained in Section 3.4.4.1.

Model The model component consists of the classes that encapsulate and interact with the data. This component is responsible for retrieving data from the environment or a data storage location and formatting it so that it can be used in the rest of the application. Essentially, the model would provide a front-end for interacting with the application's data source. It would take care of any data mapping or conversions between the data representation the application uses, such as Java objects, and the format of the data storage, such as text files or database entries. All of these functions are transparent to the rest of the application as they only have access to the shared common interface.

In the case of the robotic router system, the data for the overall system is the location and status of the robots. A class called `GraphModel` was created to interact with the robotic network. This class is responsible for sending update broadcasts to the robots and storing their responses in an object array. The rest of the application would have access to this array and interact with the robot network by calling generic functions such as `UpdateStatus` or `MoveRobot`. The actual code for sending and processing the actual messages to and from the robots is transparent

and hidden from the rest of the application.

View The view component is responsible for displaying the data in the model. This can mean displaying the data as a graphical format for a human user, displaying it for another computer system or even just writing it to a text file for processing later. The view component usually has a reference to the models of an application and will retrieved the latest data before formating and displaying it.

The view component has two different implementations in the robotic routers system. The first implementation is a class called `ImageView`. This class takes a robot's information and status from the `GraphModel` class and displays it in a graphical format over the floorplan of the environment, as seen in the top figure of Figure 3.7. The robots are drawn as triangles in the environment representing their physical locations and in colors representing their operational status. If any of the robots are visible to each other, a dark gray line is drawn connecting them.

The second view implementation is a class called `TableView`. Whereas `ImageView` displays the robots in a graphical fashion, `TableView` displays the robots in a table that contains a column for each of the robot's property, such as name, ID, location, and operational status. This view is show in the bottom figure of Figure 3.7.

Controller The controller is a class that holds the application together because it contains the actual application work flow and controls the various views and models that a program might have. Its responsibilities can range from running the business logic to manipulating data by calling functions in the model class or even changing display options in the view classes. This is the class that will determine the set of actions to perform when it receives a command from either a human user or a remote client. If the application needs to add or remove data, this component will call the necessary methods in the model component then proceed to tell the view component to update its interface to display the newest data.

In the Jupiter application, the main class `Jupiter` is the controller component of the system. It contains logic which processes each command that arrives and determines which model or view functions to call. It also interacts with the network classes responsible for running the TCP server and the classes that provide a graphical user interface to the human users.

3.4.3.2 Listeners

Listeners is a design pattern which provides an easy way of having several classes update themselves if their common dependent class changes. An example of how this works in the Model-

View-Controller design pattern is the relationship between the model component and the view component. The view component is designed to retrieve and display the latest data from the model component but retrieving and formating the data for display can be time consuming. Therefore, the view component only retrieves data after the model's data has been changed since the last display. The listener pattern provides an easy way of notifying the view component anytime the model component has been changed.

Listeners work by having the model class keep an array of references to all of the view classes that would like to be notified when the model changes. An example of this array can be seen in Listing 3.1 on Line 4. The view class all inherit an common interface of functions and if they want to listen to a model they pass a reference of themselves to the model's add listener function (Listing 3.1, Line 7). When the model is updated, it goes through its list of views and calls a function common to all of the views referenced in this array (Listing 3.1, Line 19). In each view class, this common interface function is written to act as a flag, so whenever it is call, the view class knows that the data in the model has changed so it must retrieve the latest data set (Listing 3.1, Line 29).

```
1  // A Model class which other View classes listen to for the latest →
       data
2  public class GraphModel {
3      // Array of objects that want to listen to this model
4      private List<GraphModelListener> graphListeners = new ArrayList<→
           GraphModelListener>();
5      ...
6      //add an object to the list of listeners
7      public void addGraphModelListener(GraphModelListener l) {
8          graphListeners.add(l);
9      }

11     //remove a listener from the list of listeners
12     public void removeGraphGraphModelListener(GraphModelListener l) {
13         graphListeners.remove(l);
14     }
15     // go through and run the notifyGraphChange function
16     public void notifyListeners() {
17         for (GraphModelListener listener : graphListeners) {
```

```
18              // do something to each listeners
19              listener.notifyGraphChange();
20          }
21      }
22      ...
23  }

25  // A View class which listens to the Model class
26  public class ImageView implements GraphModelListener {
27      ...
28      // called when graph model is changed
29      public void notifyGraphChange() {
30          // get the newest data and repaint the image
31          getDataFromModel();
32          repaint();
33      }
34      ...
35  }
```

Listing 3.1: Snips of source code that shows the Listener design pattern. The `ImageView` class is a listener of the `GraphModel` class.

3.4.3.3 Integration

Integration of these classes together in Jupiter is straightforward. The `Jupiter` class is the controller and has references to the model and view instances. These instances are created when the program starts up and all of the view classes are added as listeners to the model class. The model class is populated with the names and addresses of all of the nodes in the network from a local file and a separate thread is started for a server class for accepting incoming TCP connections. Lastly, the graphical user interface is laid out and displayed to the user, marking the end of the initialization process, and making Jupiter ready for user and network command input.

3.4.4 Neptune

While Jupiter is the central command for the robotic network and Mercury is the distributed application that makes up the framework, Neptune is a Java component that runs the actual algorithm and movement strategies presented in [14]. Its job is to determine where each of the robots should move to depending on their current location and experiment scenario. During initialization, it reads a file containing the locations of the robots at each time step for the known user trajectory

case and a table to calculate the optimal mobile routers locations in the unknown user trajectory case.

3.4.4.1 Integration into Jupiter

Although the Neptune module can run as a standalone application, it is designed to be integrated into the Jupiter application to take advantage of the already developed network framework. Therefore, Neptune also follows the Model-View-Controller design pattern so that integration can be as easy and seemless as possible. Because of this, the Neptune module only contains code essential to implementing the strategies and algorithms from [14], with its own graphical user interface (Figure 3.8). The networking and robot manipulation functionalities can all be shared from the Jupiter application.

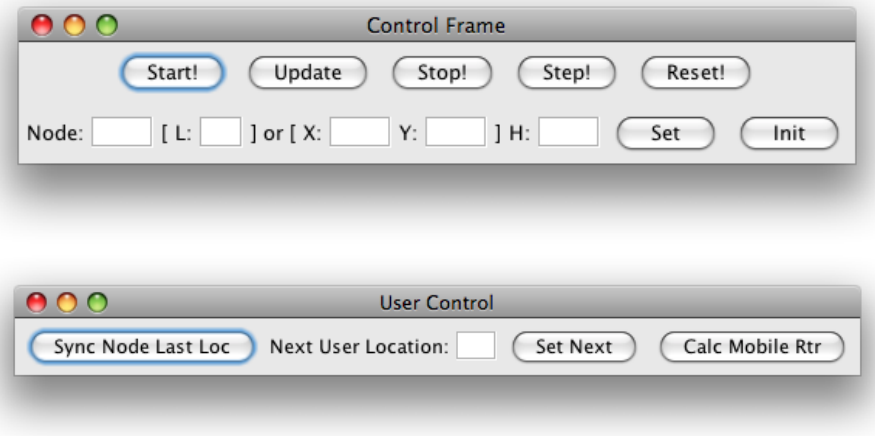


Figure 3.8: The Neptune control frame. It is used to move the robots or increment the time step in the experiment. The frame on the bottom is used for the unknown user trajectory experiment.

One of the benefits of using the Model-View-Controller design pattern is that it organizes applications into separate components that can be reused or used more than one, even at the same time. The Jupiter application has its own model, view and controller components while the Neptune module only has view and controller components. Since the MVC design pattern calls for common interfaces between all of the different components regardless of implementation, it is very easy to develop the functionality in Jupiter and Neptune separately and then integrate them together to use the same robotic network. By using the same `GraphModel` instance for both the Jupiter application and Neptune module, their controllers can control and interact with the same robotic network at the same time without conflicts. This also allows all of the different views to

display the same data simultaneously, regardless of which application they belong to.

3.4.4.2 Known User Trajectory Case

The known user trajectory case is the scenario where the user trajectory is known before the experiment starts and thus, the paths that the mobile routers need to move can be calculated in advance. Here, a text file with the locations of all of the robots at each time step is given. Neptune will read this file and at each step, determine how much to turn and move in order to move each of the robots to their assigned locations in a particular time step. Once all the of robots are within an acceptable margin from their target locations, then Neptune will increment the time step and repeat the procedure again. When a move or turn command needs to be sent, it is passed on to the `GraphModel` class, which takes care of the actual message transmission. Once integrated, both Jupiter or Neptune can be used to retrieve the latest robot locations and this will update all of the views across both applications simultaneously. The actual algorithm for creating the mobile robots given a known user trajectory is explained in [14]. Section 3.5.3 details the system's role in the known user trajectory experiment.

3.4.4.3 Unknown User Trajectory Case

The unknown user trajectory case is where the user trajectory is not known before the experiment so the mobile routers must react dynamically at each time step based on the user's movements. This is done with a large look up table containing every single possible state of the experiment. This table can be used to determine the optimal move for the mobile routers give the current robot locations and next user movement. Once these locations are determined, the rest of the process would proceed exactly the same as the known trajectory case. The algorithm for calculating the look up table is presenting in [14]. Section **??** explains the system's role in the unknown user trajectory experiment.

3.5 Experiments

3.5.1 Overview

To fully test the algorithm presented in [14], experiments were ran in the real world using the implemented mobile robotic system. Both the known and unknown user trajectory cases were tested to see whether the actual implementation would be able to perform as well as the simulated robots.

As stated in Section 3.1.3.2, the robots were allowed to move anywhere in the continuous environment. The actual robotic routers algorithm simply produces target locations to designate

where the robots need to move to in order for the discrete motion planning to work. A figure showing the starting positions of the robots as seen on the graphical user interface can be seen in Figure 3.9.

3.5.2 Known User Trajectory

In the experiment, there is a single user robot, two mobile routers and a single base station (Figure 3.9). The user starts off at location 1 with the mobile routers at locations 40 and 69, and the base station at location 49, as seen in Figure 3.9. Initially, the user is connected to the base station through the third mobile router. A dark line highlights the calculated network topology.

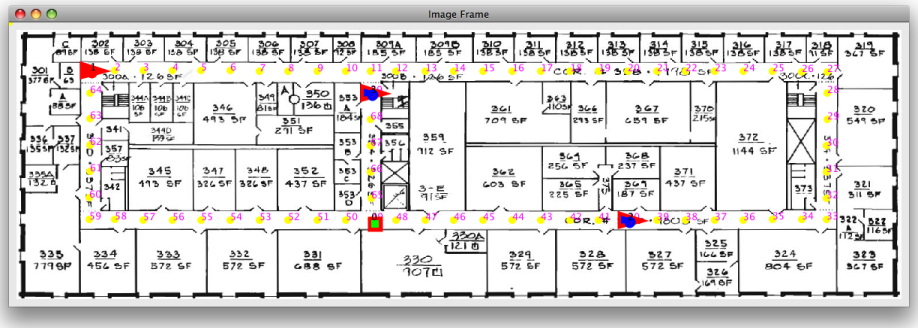


Figure 3.9: An overhead view of the environment for the real world implementation experiments with the initial starting positions of all of the nodes. The triangles represent the actual location of the mobile robots, with the robots containing a blue center. The circles are their target locations that the motion planning algorithm produces. The square represents the base station.

The overall experiment proceeded for half an hour which corresponds to 10 steps in which the user (solid red triangle) moved down towards location 59 (lower left corner) then right to end on location 55. The second mobile router stayed at location 40 for the duration of the experiment since the user is never close enough to utilizes its services. However, the third mobile router moved up to location 11 (top of middle hallway), then to the left as the user moved down in order to help maintain a user to base connection (second image in Figure 3.11). When the user reached a location where it can talk to the base directly, then the second mobile router moved to the right in preparation for when the user will enter that part of the environment (bottom two images in Figure 3.11). Several steps of the experiment can also be seen in Figure 3.10. The entire user and mobile router trajectories for the known user trajectory are given in [14].

The responsibility of the implemented mobile robot system is to use the algorithm developed from [14] and move the user and routers to their respective target locations at each time

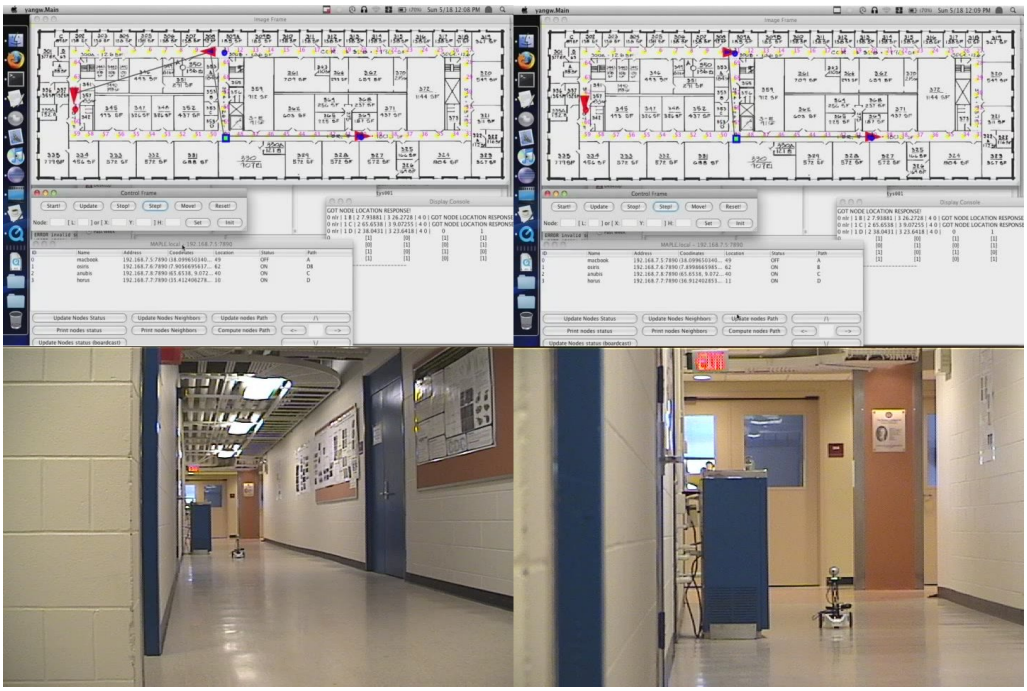


Figure 3.10: These pictures show the time when the connection path of the user is changed as it moves down the middle of the left hallway. The top row shows the configuration of mobile router network on the Jupiter user interface while the bottom row shows the corresponding user location at that time step. The left column shows the final time step when user is connected to base station through the mobile router. The right column shows the time step right after the direct connection of user to base station is satisfied.

step. Since the locations and connectivity of the robots could potentially change at any time, the implemented system must also calculate the shortest path to every node at each time step as well. Initially, the user starts off connected to the mobile routers with the base station aware of the the starting locations of the robots. The known user trajectory algorithm was then used to calculate the new user and mobile robot target locations, and the necessary movement commands to move the robots there from their current locations. A network topology graph was created using the base station copy of the robots' positions and the connectivity matrix for the environment. From this network topology graph, Dijkstra's algorithm was ran to find the shortest path to all of the nodes from the base station. Using these paths, the base station sent out move commands to all of the robots and adjusts their local position variables accordingly. If the robots were at their target locations after moving, the known user trajectory algorithm incremented the time step and the entire process repeats with new target locations. However, if the robots were not at their target locations, then the process repeats continuously to move the robots as much as necessary until they have

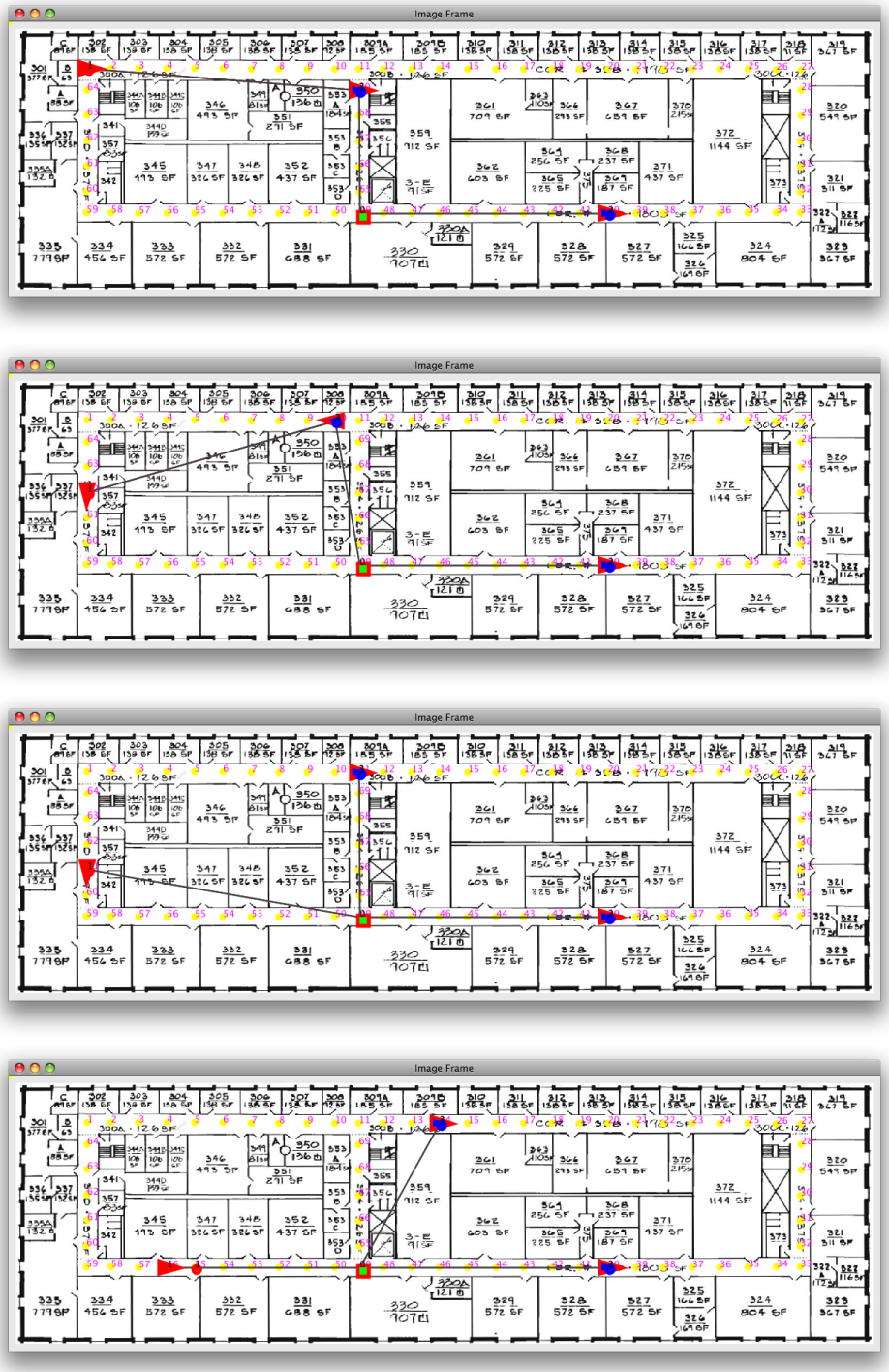


Figure 3.11: This figure shows various stages of the known user trajectory experiment being conducted using the real world robotic system implementation. The dark lines show the connectivity paths between the nodes. Initially, the user is connected through a mobile router but the path changes as the user moves closer to the base station.

reached their target locations.

Although the experiment was relatively short, it was enough to demonstrate that the overall robotic system is capable of moving three robots simultaneously so that they will follow a known trajectory. It also showed that the system can calculate and successfully hand off the user's network connection from a path through a mobile router to being directly connected to the base station. Overall, this experiment was able to demonstrate that the implemented robotic routers system can successfully implement the known user trajectory motion planning algorithm.

3.5.3 Unknown User Trajectory

In the adversarial user trajectory experiment, the same environment was used. The mobile routers were once again represented by the robots, with a personal laptop serving as the base station (bottom left of Figure 3.12). However, *the user was a human holding a laptop* instead of another mobile robot, as in the known user trajectory experiment (bottom right of Figure 3.12).

This new user was allowed to move at the same velocity as the robots but its trajectory was not known ahead of time to the base station or the mobile routers. For this case, the robotic system had to retrieve the user's location at each and every time step (this information was provided by the user) as well as its next location. The table constructed by unknown user trajectory algorithm was used to determine the motion strategies of robotic router network. By treating the user's steps as adversarial steps and picking motion strategies based on this assumption, it ensures a guaranteed optimal performance.

In the experiment, the user first started off at location 16 (in the top hallway) with the mobile routers and base station all at location 49 (Figure 3.12 and top image of Figure 3.15). From here, the user proceeded to move right to the corner of hallway, ending at location 27 (top right corner). Initially, the mobile routers did not move because the user was still in range of the base station. However, once the user moved to the limits of its communication range, the first mobile router started to move up from location 49 to 65 (bottom of middle hallway) as seen in Figure 3.13. This movement ensured that the user can still talk to the base station by using a robot to relay messages. As the user continued to move right, the first mobile router also continued to move up, using itself to maintain the user to base network connection. Once this mobile router reached location 11 (top of middle hallway), it stopped moving because the user was now at location 22 (right side of top hallway) and will remain in communication range regardless of its next movement (third image of Figure 3.15). As the user continued to move left while at the same time, the mobile router remained stationary at location 11. However, once the user reached location 27 (top right corner),

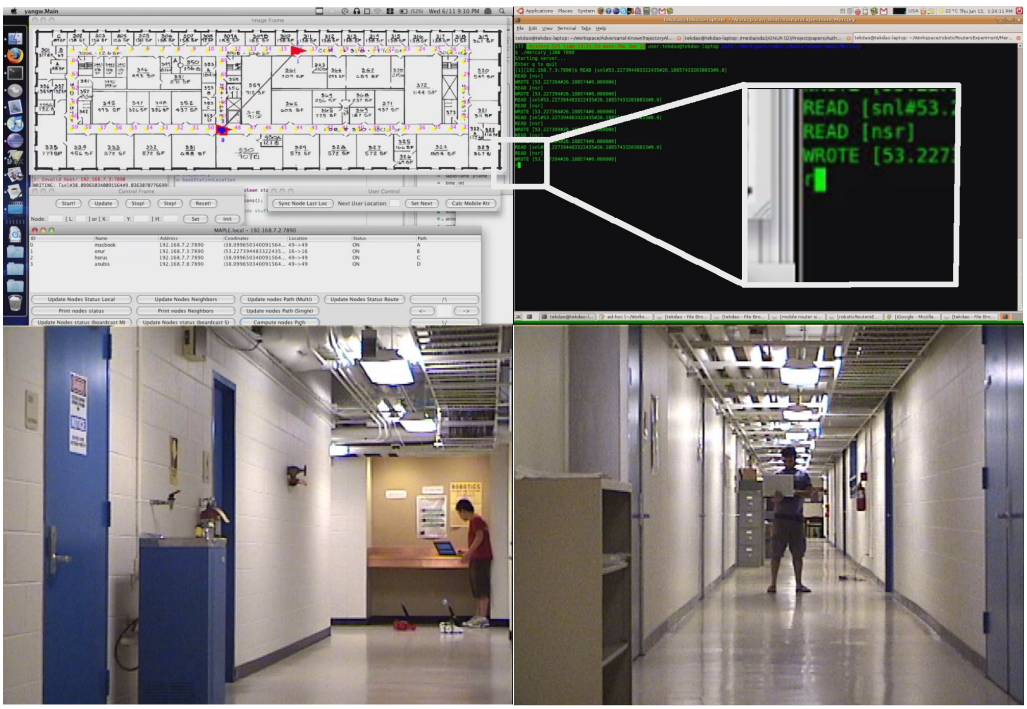


Figure 3.12: These pictures show the initial configuration of the mobile router network. In this experiment, the mobile router network keeps the connectivity of an unknown user (laptop) who requests wireless connectivity and sends acknowledgment of its initial location and moving direction in each time step. The top left figure shows the Jupiter interface, the top right figure shows the message sent from user, the bottom left figure shows the robots and base station, and the bottom right figure shows the user.

the mobile router started to move to the left from location 11 to 12. This was in response to the user reaching the end of the hallway where it will start to lose connectivity if it started to move down towards location 33 (bottom right corner). As a result, the first mobile router started to move left to help maintain the user to base station connectivity (last image of Figure 3.15). The final configuration of the experiment can be seen in Figure 3.14.

The mobile router system's role in the adversarial user trajectory experiment is similar to its role in the known user trajectory experiment. Once again, at each time step, the base station must use its locally stored robot position to calculate the best paths to use to communicate with the robots. However, the major difference is that previously, the base station knew the movements of all of the robots and could rely on using and incrementing the locally stored position variables to calculate the necessary network topology for the next step. In the adversarial user trajectory case, the base station did not know the user's movements and thus, must be able to retrieve this information before it can determine the network topology for the next step. This is a case where

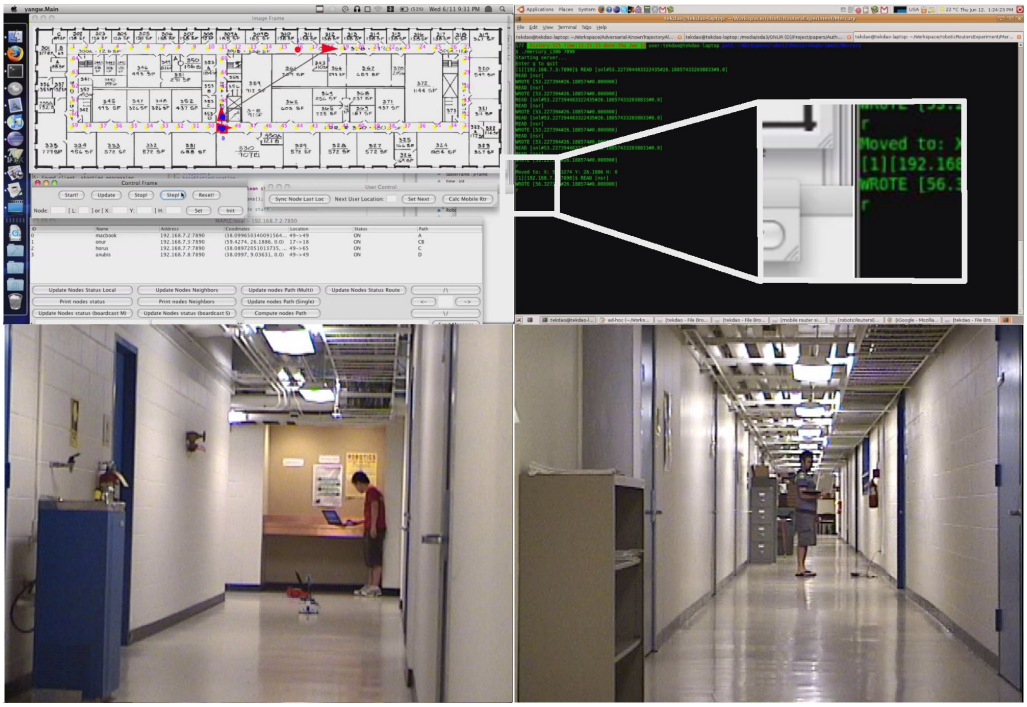


Figure 3.13: These pictures show the second step of the user. The user continues his movement in the right direction (the input `r` shown in top right figure). To keep him connected, the mobile router moves one step forward. The mobile router network maximizes the connection time by choosing the configuration where the connection time of an adversarial user is maximized.

the system needs to be able to retrieve position location variables stored on the robots as explained in Section 3.2.5. In the experiment, the base station used the current paths that were generated to ask the user for its next location. The user then responded by sending a message back through the same path to the base station and this information was used to generate the network topology for the next step. This procedure can also be used to retrieve the latest positions from the mobile routers as well.

From this second experiment, the system clearly demonstrated the ability to react dynamically to an unknown user trajectory. It showed that it is able to retrieve the user's current and next location and using this information, generate the network topology and shortest routes in order to move the mobile routers to their optimal locations. The success of this experiment represents the accomplishment of the initial goal of this problem: using mobile routers to help maintain a multi-hop wireless network connection between a mobile user and base station.

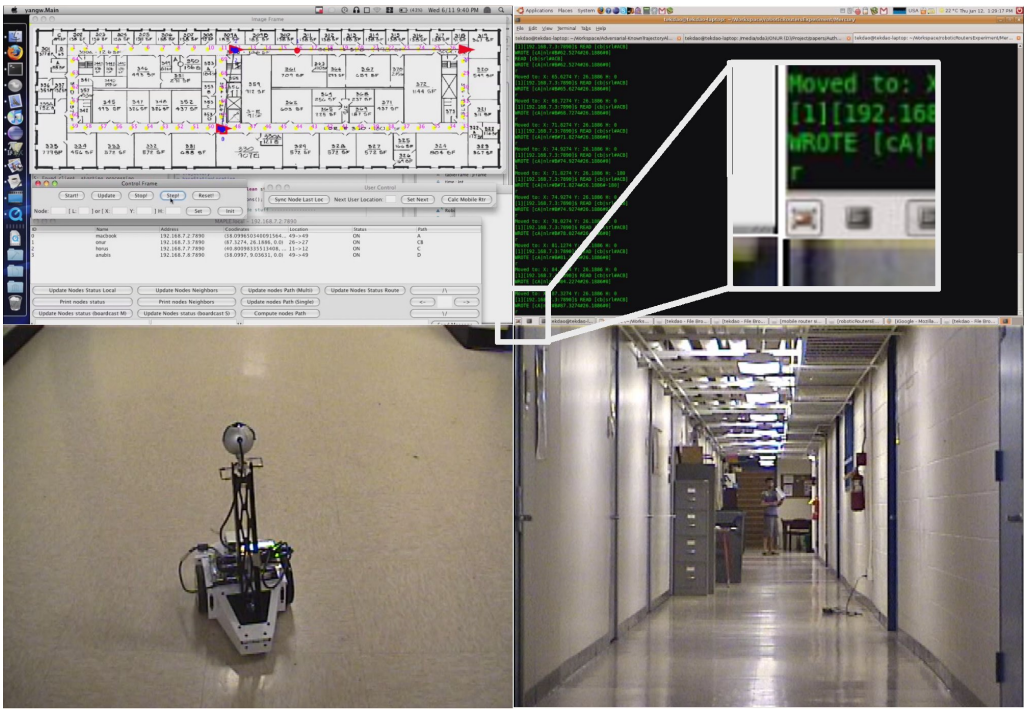


Figure 3.14: This is the final configuration of the mobile router network. The user is at the right end of the top hallway and the router had moved to the middle of the top hallway to satisfy the connectivity. In this experiment, user did not lose any sent data packets.

3.6 System Scalability and Future Work

3.6.1 Scalability

When the system was being designed, the main goal was to build a functioning framework and more time was focused on implementing networking features instead of creating a highly scaleable system. Although, the current system has only been tested with networks containing a few nodes, there are no major obstacles that would prevent the system from being scaled to networks containing dozens or even possibly hundreds of nodes. However, there are issues within the system that would need to be addressed before this is possible.

Since the simulations and experiments only contained a few nodes within the network, several functions within the system were developed in methods that favor easier implementation instead of more computationally efficient techniques. Typically, these problems are simple issues such as using an array to store variables when a hash map would be faster and more beneficial. Another example is in routing, where alphabetical characters were used to designate nodes, producing a hard limit of 26 nodes in the network. However, this can be extended to always use two characters to designate a node or add separation characters between node names. These issues can

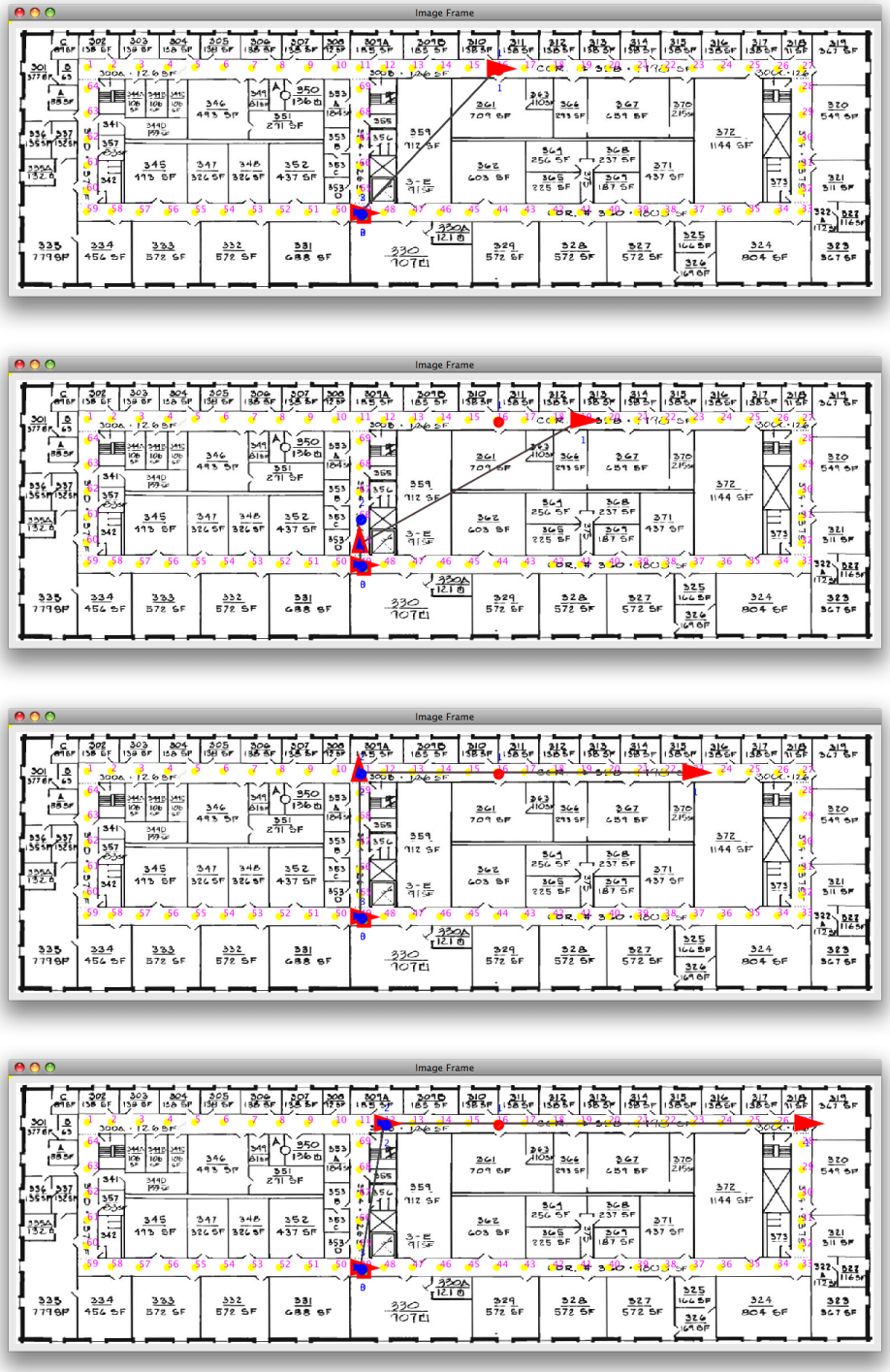


Figure 3.15: This figure shows various stages of the adversarial user trajectory experiment being conducted using the real world robotic system implementation. The dark lines show the connectivity paths between the nodes. The first screen shows a mobile router moving up to maintain connectivity as the user moves to the right. After reaching the top, the mobile router remains stationary until the user reaches the end of the hallway. When this occurs, the router starts to move right in an attempt to maintain connectivity between the user and base station.

be easily fixed to improve scalability and at the time time, increase the stability and performance of the system.

The other limiting factor with scalability is the use of bandwidth within the network. When the transmission packet format and networking algorithms were developed, they also focused on easy to implement techniques over performance and scalability. Since the experiments only utilized four nodes and energy conservation was ignored, the networking algorithms were designed with the assumption of unlimited bandwidth. However, this is never the case in the real world where often, bandwidth and energy are both severally limited. One area that is affected by this is the broadcasting algorithm. The current method simply broadcasts messages blindly and the only limiting mechanism is to throw away messages that have been received before. More efficient algorithms could be developed to selectively broadcast messages to only certain nodes but still achieve the same results as before. One example is to have nodes not broadcast a message to its neighbor if that neighbor has already received the same message before. This is a improvement that is simple to implement since all broadcast messages already track the nodes they have passed through. The transmission packet formats themselves could also be compressed or encoded differently to save bandwidth.

The system right now is tested and performs well in the environment where there are only a few nodes in the network. It is certainly possible to extend this system to use dozens of nodes but some improvements need to be made before this is possible.

3.6.2 Future Work

Although this system was designed to implement the algorithm and experiments presented in [14], it can easily be extended to a more generic robotic network system capable of supporting other algorithms and implementations. Along with the scalability techniques as mentioned in the previous section, other improvements could be made to create a better, more robust, robotic system.

The applications right now has the capability of transmitting, routing and broadcasting any message using the TCP internet protocol. The biggest feature that would improve and expand the overall system would be to add support for the UDP protocol. As explained in Section 3.3.2.1, these two protocols function in different ways to serve different purposes. By supporting UDP, developers would have the flexibility of picking the best protocol for the function they are trying to implement instead of being restricted to only one standard.

Additional features that would be helpful are acknowledgment or status messages. Cur-

rently, if any application sends a robot move command, there is no indication of whether that robot actually received the command and if the command was successfully executed. This is especially problematic when move commands are routed across the network through several intermediate routers. Since TCP connections are already made when commands are transfered, they can be kept open until a robot finishes its move action and a status message could be returned just as connections are closed. If the control messages were sent using UDP, the acknowledge messages could be sent in the same way, using either the same route or broadcasted throughout the network.

One of the initial problems with TCP is that sometimes intermittent connections can be made but not kept and this can cause many accuracy problems with calculating network topology, as described in Section 3.2.8. One way of solving this is to develop a mechanism, similar to the `ping` command, where individual links between nodes can be tested for quality. This mechanism could either attempt several TCP connections and measure their bandwidth or transmission time to determine their strength. If UDP was used, several packets could be sent back and forth to measure the round trip packet time and percentage of lost packets. All of this information could be used to determine be best possible links to use and would greatly increase the reliability and stability of the overall robotic network system.

One feature that would help achieve the ultimate goal of maintaining a user to base station network connection is to keep TCP connections open between nodes and forward data streams directly. Currently, network connections are only kept long enough to transmit individual packets and are closed after the packets have been received. The current implementation essentially only propagates individual message from the user to the base station. However, TCP connection functions could be modified so that connections are kept open even after the data has been transmitted. Therefore, instead of passing single messages, each node would forward streams of data between its connections. These chains of network connections would allow the user to send streams of data directly to the base station and vice versa. This would have enormous potential as sensory data, such as live webcam video, could now be streamed directly to the base station without interruption.

CHAPTER 4
Discussion and Conclusion

The contributions from the two parts of this thesis detailed many different aspects and areas associated with wireless mobile robotic networking. It looked at a theoretical approach to the fundamental problem of network formation using mobile robots that have small communication ranges compared to the size of their environment. The developed algorithm was analyzed and contrasted to another approach in several computer simulations. From a systems point of view, this thesis detailed a mobile robotic system that was implemented to test motion planning algorithms designed to maintain multi-hop network connections. This system was further expanded to provide a robust framework that can be used to support future networking algorithm implementations and experiments.

These contributions demonstrate some of the recent trends in mobile robotics research. Mobile robots have already started to become an acceptable, and even welcomed, part of many peoples' lives and this will only increase with time. With the prevalence of wireless technology, it has helped to push this field into new and exciting areas, particularly when it comes to practical real world applications. More research is constantly being implemented and tested on large teams of robots and this will only expand as the field of mobile robotics matures. As wireless and computing technologies become more powerful and affordable, it might not be a too distant future where tasks assigned to a single robot (or several humans) will be finished in a fraction of the time by several groups of networked mobile robots.

REFERENCES

[1] S. Alpern and S. Gal. *The Theory of Search Games and Rendezvous*. Springer, 2002.

[2] Steve Alpern. The rendezvous search problem. *SIAM J. Control Optim.*, 33(3):673–683, 1995.

[3] Edward J. Anderson and Sandor P. Fekete. Asymmetric rendezvous on the plane. In *SCG '98: Proceedings of the fourteenth annual symposium on Computational geometry*, pages 365–373, New York, NY, USA, 1998. ACM Press.

[4] E. Arkin, M. Bender, S. Fekete, J. Mitchell, and M. Skutella. The freeze-tag problem: How to wake up a swarm of robots, 2002.

[5] Esther M. Arkin, Sandor P. Fekete, and Joseph S. B. Mitchell. Approximation algorithms for lawn mowing and milling. *Computational Geometry*, 17(1-2):25–50, 2000.

[6] Howie Choset. Coverage of known spaces: The boustrophedon cellular decomposition. *Auton. Robots*, 9(3):247–253, 2000.

[7] Supratim Deb, Muriel Médard, and Clifford Choute. Algebraic gossip: a network coding approach to optimal multiple rumor mongering. *IEEE/ACM Trans. Netw.*, 14(SI):2486–2507, 2006.

[8] A. Ganguli, J. Cortes, and F. Bullo. Multirobot rendezvous with visibility sensors in nonconvex environments. *IEEE Transactions on Robotics*, November 2006. To appear.

[9] S. Martinez, F. Bullo, J. Cortes, and E. Frazzoli. On synchronous robotic networks - Part II: Time complexity of rendezvous and deployment algorithms. *"IEEE Transactions on Automatic Control"*, 2007. To appear.

[10] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge Press, 2000.

[11] Sameera Poduri and Gaurav S. Sukhatme. Achieving connectivity through coalescence in mobile robot networks. In *International Conference on Robot Communication and Coordination*, Oct 2007.

[12] Sameera Poduri and Gaurav S. Sukhatme. Latency analysis of coalescence in robot groups. In *IEEE International Conference on Robotics and Automation*, 2007.

[13] Nicholas Roy and Gregory Dudek. Collaborative robot exploration and rendezvous: Algorithms, performance bounds and observations. *Auton. Robots*, 11(2):117–136, 2001.

[14] O. Tekdas and V. Isler. Robotic routers. In *Proc. IEEE Int. Conf. on Robotics and Automation*, 2008. to appear.

[15] L.C. Thomas and M. Pikounis. Many-player rendevous search: stick together or split and meet? *Naval Research Logistics*, 48:710–721, 2001.

APPENDIX A
Commands

User Commands		
Command Name	Encoded Message	Purpose
Update Nodes Status (Local)	uns	Updates the status of all nodes locally
Update Nodes Status (Route)	unsr	Updates the status of all nodes using their paths
Update Nodes Multi Broadcast	unbm	Updates the status of all nodes using multiple broadcasts
Update Nodes Single Broadcast	unbs	Updates the status of all nodes using a single broadcast
Update Nodes Path Multi Broadcast	unpm	Updates the paths of all nodes using multiple broadcasts
Update Nodes Path Single Broadcast	unps	Updates the paths of all nodes using a single broadcast
Update Nodes Neighbors	unn	Updates the status neighboring nodes and their neighbors
Send Message Prompt	smp	Prompts the user to send a message
Send Discovery Broadcast Prompt	sdp	Prompts the user to send a status update broadcast
Set Node Location	snl	Prompts the user to set the node's current location
Print Nodes Status	pns	Prints the status of all nodes, obtained from the current node
Print Nodes Neighbors	pnn	Prints the status of all nodes and their neighbors, obtained from the current node
Print Recieved Id	pid	Prints all of the message IDs received so far
Control Robot Prompt	crp	Prompts the user to control the robot
Compute Nodes Path	cnp	Computes the shortest paths from the base station to all nodes
Left, Right, Up, Down	l, r, u, d	Sets the node's location to one unit left, right, up or down respectively
Quit	q	Quits the application

Table A.1: A table containing all of the message encoding and user commands for the mobile robotic system.

Node Commands and Responses		
Command Name	Encoded Message	Purpose
Control Robot Command	crc	Moves the robot
Send Message Command	smc	Sends a message
Path Request Broadcast (Multiple)	rdb	Requests a node's path using a single broadcast message
Path Request Broadcast (Single)	rdbs	Requests a node's path using multiple broadcast messages
Status Request Broadcast (Multiple)	nlb	Requests a node's status using a single broadcast message
Status Request Broadcast (Single)	srb	Requests a node's status using multiple broadcast messages
Status Request Local	nsr	Requests a node's status locally
Status Request Route	srl	Requests a node's status using its path
Route Discovery Response	rdr	Response to a node path request
Node Location Response	nlr	Response to a node status request
Node Neighbor Request	nnr	Response to a node neighbor status request

Table A.2: A table containing all of the messages that a node can receive and respond with.

Separation Symbols		
Separator Name	Character	Purpose
Header Message Separator	|	Seperates the routing header from the message
Send Message Separator	!	Seperates the send message command from the actual message to send
Message Parm Separator	#	Seperates the parameters of a command
Node Neighbor Separator	%	Seperates the status of different neighbors
End Of Packet Char	$	Designates the end of a data packet

Table A.3: A table containing all of the separation symbols used for encoding messages.

APPENDIX B
TCP Client and Server Code

B.1 C Client

```
1  // C version
2
3  // address and port to connect to
4  const char * sendAddr;
5  int sendPort;
6
7  // socket and tcp connection variables
8  int sockfd, portno, n;
9  struct sockaddr_in serv_addr;
10 struct hostent *server;
11
12 // open socket for connection
13 sockfd = socket(AF_INET, SOCK_STREAM, 0);
14
15 // get host name
16 server = gethostbyname(sendAddr);
17
18 // set server address
19 bzero((char *) &serv_addr, sizeof(serv_addr));
20 serv_addr.sin_family = AF_INET;
21 bcopy((char *)server->h_addr,
22 (char *)&serv_addr.sin_addr.s_addr,
23 server->h_length);
24 serv_addr.sin_port = htons(sendPort);
25
26 // connect
27 connect(sockfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr))
28
29 // reading and writing is now possible with a char[] buffer
30 read(sockfd, buffer, bufferSize);
31 write(sockfd, buffer, bufferSize);
```

```
// close socket when finished
close(sockfd);
```

Listing B.1: Source code that is used for the C client interface

B.2 Java Client

```
// Java Client

// create a socket to the server's address and port
Socket socket = new Socket(address, port);

// set up readers and writers
InputStreamReader cin = new InputStreamReader(socket.getInputStream()→
    );
BufferedReader in = new BufferedReader(cin);
PrintWriter out = new PrintWriter(socket.getOutputStream(), true);

// reading and writing is now possible with a char[] buffer
in.read( buffer );
out.write( buffer );
out.flush();

// close when done
out.close; in.close(); cin.close();
socket.close();
```

Listing B.2: Source code that is used for the Java client interface

B.3 C Server

```
// C Server

// server variables
int clntSock, portno, clilen;
pthread_t threadID;
struct ServerThreadArgs *threadArgs;
struct sockaddr_in serv_addr, cli_addr;

bool subThread = false;
```

```
11 portno = _port;

13 // open a server socket
14 servSock = socket(AF_INET, SOCK_STREAM, 0);

16 // set server address info
17 bzero((char *) &serv_addr, sizeof(serv_addr));
18 serv_addr.sin_family = AF_INET;
19 serv_addr.sin_port=htons(portno);
20 serv_addr.sin_addr.s_addr = INADDR_ANY;

22 // bind the server to a socket
23 bind(servSock, (struct sockaddr *) &serv_addr, sizeof(serv_addr))

25 // listen for an incoming connection on the server socket
26 listen(servSock, 5);
27 clilen = sizeof(cli_addr);

29   // loop forever
30 for (;;) {

32     // accept the new client's connection
33     clntSock = accept(servSock, (struct sockaddr *) &cli_addr, (→
           socklen_t *) &clilen);

35     // use threads to handle each client

37     // create a struct to store the client's info for the new thread
38     threadArgs = (struct ServerThreadArgs *) malloc(sizeof(struct →
           ServerThreadArgs));
39     threadArgs->clientSocket = clntSock;
40     threadArgs->mercuryObject = _mercury;

42     // create and start a new thread to handle the connection
43     pthread_create(&threadID, NULL, ServerHandlerThreadFunction, (→
           void *) threadArgs);
```

```
45 }
```

Listing B.3: Source code that is used for the C server interface

B.4 Java Server

```
1 // Java Multi-threaded Server

3 // create a new socket for the server at a specific port
4 ServerSocket serverSocket = new ServerSocket(port);

6 // loop until listening ends
7 while (listening) {

9     // block until a client connects to the server
10    Socket clientSocket = serverSocket.accept();

12    // create and start a new thread to take care of the new client
13    ServerThread  st = new ServerThread(clientSocket);
14    Thread t = new Thread(st);
15    t.start();
16 }
```

Listing B.4: Source code that is used for the Java server interface