Distributed-in/Distributed-out Sensor Networks: A New Framework to Analyze Distributed Phenomena

by

Constantine Kleomenis Christakos

Submitted to the Program in Media Arts and Sciences, School of Architecture and Planning in partial fulfillment of the requirements for the degree of Doctor Of Philosophy at the MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2006

© Massachusetts Institute of Technology 2006. All rights reserved.
Distributed-in/Distributed-out Sensor Networks: A New Framework to Analyze Distributed Phenomena

by

Constantine Kleomenis Christakos

Submitted to the Program in Media Arts and Sciences, School of Architecture and Planning on January 10, 2006, in partial fulfillment of the requirements for the degree of Doctor Of Philosophy

Abstract

With a new way of thinking about organizing sensor networks, we demonstrate that we can more easily deploy and program these networks to solve a variety of different problems. We describe sensor networks that can analyze and actuate distributed phenomena without a central coordinator. Previous implementations of sensor networks have approached the problem from the perspective of centralized reporting of distributed events. By contrast, we create a system that allows users to infer the global state from within the sensor network itself, rather than by accessing an outside, central middleware layer. This is accomplished via dynamic creation of clusters of nodes based on application or intent, rather than proximity. The data collected and returned by these clusters is returned directly to the inquirer at his current location. By creating this Distributed-in/Distributed-out (DiDo) system that bypasses a middleware layer, our networks have the principal advantage of being easily configurable and deployable. We show that a system with this structure can solve path problems in a random graph. These graph problems are directly applicable to real-life applications such as discovering escape routes for people in a building with changing pathways. We show that the system is scalable, as reconfiguration requires only local communication. To test our assumptions, we build a suite of applications to create different deployment scenarios that model the physical world and set up simulations that allow us to measure performance. Finally, we create a set of simple primitives that serve as a high-level organizing protocol. These primitives can be used to solve different problems with distributed sensors, regardless of the underlying network protocols. The instructions provided by the sensors result in tangible performance improvements when the sensors’ instructions are directed to agents within a simulated physical world.

Thesis Supervisor: Andrew Lippman
Title: Principal Research Scientist, MIT Media Laboratory
Distributed-in/Distributed-out Sensor Networks: A New Framework to Analyze Distributed Phenomena

The following people served as readers for this thesis

Accepted by
Joseph Paradiso
Associate Professor of Media Arts and Sciences
Massachusetts Institute of Technology

Accepted by
Samuel Madden
Assistant Professor of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Acknowledgments

No thesis happens without a large number of people who help to make it possible. This thesis is no exception. I want to thank all of my fellow students with whom I worked closely at the Media Lab. Their input and insight were invaluable while I was developing research plans and research ideas. Special thanks go out to Bill Butera and Jacky Mallett. Josh Lifton provided research insight, vision in helping “sell” sensor networks to the Media Lab, and engineering prowess in his design of the Pushpin-computing platform which provided much inspiration for this research as well as providing a platform with which I could explore preliminary research directions. The ultimate result became this thesis. In addition, my officemates Aggelos Bletsas and Dimitris Vyzovitis provided advice and good humor while this thesis was in progress.

Among the Media Lab faculty, special thanks goes out to Ted Selker for providing me with research space and facilities during the final stages of my work, support which was crucial under rather difficult circumstances. Also, V. Michael Bove, Jr., starting with his work as a thesis reader for my Master’s thesis, was always willing to provide research tips and pointers while this Ph.D. research was being developed. Finally, I would like to thank my thesis committee, composed of Joseph Paradiso, Samuel Madden, and my advisor, Andrew Lippman, who provided constant goading to ensure my research was as rigorous as was needed.

Other faculty who provided me with research insights were Matt Welsh of Harvard’s Computer Science Department and Daniela Rus of MIT’s Computer Science and Artificial Intelligence Laboratory.

The Media Lab’s support staff has been an invaluable resource, and the day-to-day support from Deborah Widener, Polly Guggenheim, Kevin Davis, Michalis Bletsas, and Will Glesnes was extraordinarily helpful.

Other thanks go out to Orbital, The Ramones, The Donnas, Queen, and Neil Young for helping provide the soundtrack to my graduate school career.

Finally, my family was the source of much patience during the long and difficult
process of finishing this thesis, particularly given all of my missed predictions regarding the completion date. They provided incredible amounts of support without which this would probably not be possible. My parents’ support would not have been possible were it not for my grandparents who came to the United States in search of a better life for their children. Those grandparents who saw me grow up over the past 30 years all passed away during my tenure at the Media Lab, and they are deeply missed, particularly as I now find myself having finally come to the end of my stay here at MIT.
Contents

1 Introduction 21
  1.1 Purpose ................................................................. 21
  1.2 Sample Problem ....................................................... 23
  1.3 Approach ................................................................. 27

2 Background 31
  2.1 Introduction ............................................................... 31
  2.2 The Taxonomy of Monitoring Sensors ................................ 33
    2.2.1 Basic Detection ..................................................... 35
    2.2.2 Environmental Sensing ............................................. 37
  2.3 Battlefield Sensors ..................................................... 37
    2.3.1 Sensors in Distributed Tracking .................................. 39
  2.4 Distributed Control ................................................... 42
    2.4.1 Basic Control Systems .............................................. 42
    2.4.2 Distributed Control Systems ..................................... 44
  2.5 Viral Networks ............................................................ 49
  2.6 Architecture of Sensor Networks ..................................... 52
  2.7 Simulation of Sensor Networks ........................................ 54

3 Experiences with Communication Protocols for DiDo 57
  3.1 Introduction ............................................................... 57
  3.2 Pathfinding in a Graph ................................................ 58
    3.2.1 Representation of Physical Layouts as a Graph ............... 58
3.2.2 Analysis of Paths to a Sink in a Graph . . . . . . . . . . . . . 62
3.3 Using Ad Hoc Network Protocols to Route People, Not Packets . . 64
3.4 Building DiDo Applications with High-Level Primitives . . . . . . . 67
3.5 Simple Communication Models . . . . . . . . . . . . . . . . . . . . 70
   3.5.1 Pathfinding Using Simplified Assumptions . . . . . . . . . . . 70
   3.5.2 Evaluation of Implementations and Performance . . . . . . . . 72
3.6 Adaptive Communication Models . . . . . . . . . . . . . . . . . . . 74
   3.6.1 New Thinking About Protocols . . . . . . . . . . . . . . . . . 74
   3.6.2 Creating a Protocol . . . . . . . . . . . . . . . . . . . . . . . 75
   3.6.3 Routing Protocol . . . . . . . . . . . . . . . . . . . . . . . . 76
3.7 How the Sensors Cooperate . . . . . . . . . . . . . . . . . . . . . . 78
3.8 Designing the Protocol . . . . . . . . . . . . . . . . . . . . . . . . . 78
   3.8.1 Simple Rules for Choosing Paths . . . . . . . . . . . . . . . . 79
   3.8.2 Working with Different Sensor Densities and Network Models 80
   3.8.3 Sample Protocol Walkthrough . . . . . . . . . . . . . . . . . . 82
   3.8.4 Physical Blockage vs. Network Blockage . . . . . . . . . . . . 87
   3.8.5 Implementation . . . . . . . . . . . . . . . . . . . . . . . . . 88
3.9 Simulating New Protocols for DiDo . . . . . . . . . . . . . . . . . . 90

4 Building Simulation Tools . . . . . . . . . . . . . . . . . . . . . . . . 91
   4.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 91
   4.2 Designing Scenarios with the Map Generator Tool . . . . . . . . 93
   4.3 Specifying Environment Changes . . . . . . . . . . . . . . . . . . . 96
   4.4 Code Generation . . . . . . . . . . . . . . . . . . . . . . . . . . . . 98
   4.5 Other Control Output . . . . . . . . . . . . . . . . . . . . . . . . . 101
   4.6 Automating the Map Generation Process . . . . . . . . . . . . . . 101
   4.7 Integration with TOSSIM . . . . . . . . . . . . . . . . . . . . . . . 102
   4.8 Evaluation and Results . . . . . . . . . . . . . . . . . . . . . . . . . 104
   4.9 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 109
# Pedestrian Simulation

5.1 Introduction .............................................. 111
5.2 Previous Work in Pedestrian Simulation ................. 112
5.3 The Movement Simulator .................................. 115
  5.3.1 Architecture ........................................ 115
  5.3.2 Representing Floorplan Layouts as a Graph .......... 116
  5.3.3 Pedestrian Behavior ................................ 117
5.4 Pedestrian Escape Performance ............................ 118
  5.4.1 Evaluation Tools and Scenarios ..................... 118
  5.4.2 Results ............................................. 118
5.5 Integration with TOSSIM .................................. 122
5.6 Conclusion ............................................ 123

# Results and Evaluation

6.1 Introduction ............................................ 125
6.2 Integration of the Movement Simulator and TOSSIM ....... 126
6.3 Performance Comparison .................................. 128
6.4 Stability ................................................. 134
6.5 The Case of Two Exits ................................... 141
6.6 Conclusion ............................................ 145

# Conclusion and Further Work

7.1 Managing Sensor/Transducers ............................. 147
7.2 Wireless Communication .................................. 148
7.3 Scalability ............................................... 149
7.4 Actuation Models ........................................ 150
7.5 Simulation Features ...................................... 151
7.6 Additional Further Work .................................. 152
7.7 Final Issues ........................................... 153
List of Figures

1-1 Clockwise from upper left: Traffic sign directs westbound drivers to turn left at the corner of Lafayette St. and Kenmare St. The next sign directs drivers to make a right onto Broome St. Third, drivers are directed to bear left at the Broome St./Watts St. split. Finally, at Watts St. and Varick St., drivers see the entrance to the Holland Tunnel. Note the electronic sign below that can be changed, depending on conditions. ................................................................. 24

1-2 The traffic signs form a path from point A to the Holland Tunnel entrance at point D. ................................................................. 25

1-3 A basic architecture of intelligent signs with peer-to-peer communication 26

2-1 Left: When aggregating to a base station on the same plane as the sensors, sensors may use multihop protocols to aggregate the data. Right: When the base station is on a plane above the sensors, sensors can take advantage of their numbers to aggregate the data by using cooperative diversity. ................................................................. 39

2-2 A toaster as an example of an open-loop control system ............. 43

2-3 A closed-loop control system ..................................................... 43

2-4 A Full Multi-Input, Multi-Output (MIMO) Control System ........ 44

2-5 A model of a decentralized distributed control system, also known as a networked control system ........................................... 45

2-6 The decentralized control system of the Naiades Project .............. 46
2-7 The architecture of typical sensor systems. All inquiries are mediated through the aggregating base station. 

3-1 Left: The paths to which users will be directed in order to reach the exit the fastest. Right: That layout as it corresponds to a graph, with individual rooms maintaining their same label. 

3-2 A sample floorplan with noted dimensions with individual rooms maintaining their same label. 

3-3 The sample floorplan represented more formally. 

3-4 The floorplan represented as a combination of current flow and potential. 

3-5 The bridges of Königsburg represented as a graph. 

3-6 The hedge maze at Versailles. 

3-7 The hedge maze at Versailles represented as a graph. 

3-8 A graph with an associated spanning tree rooted at the sink of that graph. 

3-9 In the case of a blocked path, the source node has to pick the shortest route by picking the smallest valid hop value passed to it from the nodes adjacent to it. 

3-10 Left: A shortest path in an ad hoc network is created between source, \( s \), and sink, \( t \) for routing packets. Right: A shortest path in a floorplan is created to route a person at \( s \) to the exit at \( t \). 

3-11 A comparison of the active and passive protocols in the discovery of paths to the exit and recovery from blockage in a simple ring topology of different sizes. 

3-12 Top: When a graph is not very dense and an edge is blocked, messages need to travel distantly to find an alternate path. Bottom: In a very dense graph, a blocked path can be corrected much more quickly because an alternate path is more easily available.
3-13 The sensor model. Transducers are in the center of the room with sensing elements, marked by an “X”, monitoring the state of the passageways of the room’s exits.

3-14 **Left:** The network communication between the two transducers. **Right:** The physical pathway known by the top transducer between it and the exit transducer.

3-15 **Left:** The network communication between the two transducers. **Right:** The network communication after an intermediate transducer is added.

3-16 **Left:** The network communication before a blockage is detected. **Right:** The network communication after a blockage is detected.

3-17 **Left:** The physical paths maintained by the transducers before a blockage is detected. **Right:** The physical paths maintained after a blockage is detected.

4-1 The Output of the Map Generator Application

4-2 The Map Generator Application GUI

4-3 **Left:** A small floorplan. **Right:** That floorplan as represented using the map generator.

4-4 **Left:** A layout created in the Map Generator. **Right:** A blocked passageway is induced.

4-5 The architecture of the automated map generation system.

4-6 A flowchart of the Tython scripts used to run the simulation

4-7 Power consumed by motes for different functions.

4-8 Number of messages per node for recovery in the event of a physically blocked passageway. The horizontal line indicates the average network connectivity at which the system breaks down.

4-9 Number of messages per node for setup and recovery in the event of a blocked network connection versus a physically blocked passageway.

5-1 A simulation of pedestrians escaping through a room in the Panic simulator.
5-2 An examination of escape times out of a room, given the starting number of pedestrians for that simulation. ........................................ 113

5-3 The average escape time for a single pedestrian to leave a room, given the number of pedestrians in the room at the time, averaged over all simulations, with moving average ........................................ 115

5-4 The Stand-Alone Movement Simulator Application .......................... 116

5-5 The Architecture of the Movement Simulator ................................. 117

5-6 A comparison of mean escape time vs. number of pedestrians for graphs of 15 Nodes ................................................................. 119

5-7 A comparison of the variance of uniformly distributed pedestrians and power law distributed pedestrians with linear interpolation for graphs of 15 nodes ................................................................. 120

5-8 A comparison of escape times vs. nodes in the graph layout for graphs with 50 pedestrians ................................................................. 120

5-9 A comparison of escape times vs. nodes in the graph layout for graphs with 250 pedestrians ................................................................. 121

5-10 A comparison of escape time variance vs. nodes in the graph layout for graphs with 250 pedestrians ...................................................... 121

5-11 A comparison of escape times vs. nodes in the graph layout for graphs with 500 pedestrians ................................................................. 122

6-1 Interactions Between the TOSSIM simulator and DirectPedestriansPlugin ................................................................. 126

6-2 The Movement Simulator plugin integrated with TOSSIM ............... 127

6-3 Performance improvement of DiDo using increasing bandwidth settings 129

6-4 A comparison of the mean escape times for the Movement Simulator in the stand-alone case and DiDo-augmented case for uniformly distributed pedestrians ................................................................. 130
6-5 A comparison of the variance of escape times for the Movement Simulator in the stand-alone case and DiDo-augmented case for uniform distributed pedestrians ........................................ 130
6-6 A comparison of mean escape times for the Movement Simulator in the standalone case and DiDo-augmented case for power law-distributed pedestrians ........................................ 131
6-7 A comparison of mean escape time variance for the Movement Simulator in the standalone case and DiDo-augmented case for power law-distributed pedestrians ........................................ 132
6-8 A Comparison of mean escape times by graph size, with 250 power law-distributed pedestrians, with mean values. .......................... 132
6-9 A Comparison of escape time for the DiDo-augmented Movement Simulator both the uniform-distributed and power law-distributed pedestrian cases. ........................................ 133
6-10 A comparison of the number of active pedestrians remaining in the system over time in a standalone pedestrian simulation ............... 135
6-11 A comparison of the number of active pedestrians remaining in the system over time in a DiDo-augmented pedestrian simulation ........ 135
6-12 Queue evolution of the standalone pedestrian simulation over time with uniform distributed pedestrians ........................................ 136
6-13 Queue evolution of the standalone pedestrian simulation over time with power law-distributed pedestrians ........................................ 137
6-14 Queue evolution of the DiDo-augmented pedestrian simulation over time with power law-distributed pedestrians ....................... 138
6-15 Queue evolution of the DiDo-augmented pedestrian simulation over time with uniform distributed pedestrians ....................... 138
6-16 Average distance from exit in the standalone case .......................... 140
6-17 Average distance from exit in the DiDo-augmented case .................. 140
6-18 Mean exit time of uniform-distributed pedestrians out of a 15 node graph with one exit versus two exits ............................... 142
6-19 Mean exit time of power law-distributed pedestrians out of a 15 node graph with one exit versus two exits .......................... 142
6-20 Mean exit time of power law-distributed pedestrians out of a 15 node graph with one exit versus two exits, using DiDo .................. 143
6-21 Performance in the two-exit scenario in a 15 node graph .............. 144
# List of Tables

2.1 Features of basic sensing systems ........................................ 35  
2.2 A Comparison of Features in Distributed Control and DiDo Systems . 49  
3.1 Code Complexity of Protocols Implemented in nesC ................. 72
Chapter 1

Introduction

1.1 Purpose

I propose an architecture termed “Distributed-in/Distributed-out” that comprises an array of devices that both gather data and locally report results. The data is obtained in situ, and the processed output is used in the same area. Meanwhile, the devices communicate with their local neighbors and coordinate with devices within a scope limited to the immediate area. This allows such a network to scale in size while maintaining simple, local patterns of communication. Finally, because the number of sensors may not even be known beforehand, I want the system to work with networks of increasing size within a physical region.

This system consists of geographically dispersed and networked sensors and effectors. The sensors automatically establish both local and distant communications channels, and they perform local data processing to report results. I do this in the context of a sample application that is illustrative: building evacuation. I have chosen this because it is an example case where the sensing is distributed and the results of that information are needed everywhere in the building. For example, if one wishes to direct people out of a building during a fire, one needs to know which pathways are blocked, which are open, which are clogged, which are being used by emergency personnel, which are handicap-accessible, and when these states change and new instructions need to be provided. This example stresses the system architecture and I
will show that it is typical of a set of applications that are broadly useful, for example, in traffic control, resource allocation, amusement selection in a theme park, or even package delivery.

I seek to create new way of thinking about and designing sensor networks to solve problems. By understanding what problems the sensor network is trying to solve, a designer can designate certain places or phenomena within the network as “resources” and sensors that control “actuators.” At times, such as the example of building escape, the actuators are signs that indicate directions to pedestrians in the building. The sensors gather data about the sensed region, and the sensors then tell the signs to change the directions given to the pedestrians regarding where to travel. This redirection of pedestrian flow causes changes in the crowding conditions in the building; the sensors then read in this new state, and the pedestrians receive an updated set of instructions from the signs. I create a high-level protocol based on simple primitives that organize sensor networks around these principles so that designers can more easily build sensor networks with this model of resources and actuators in mind. Designers can then quickly use these primitives to organize the coordination of the sensors to solve the problem at hand.

Evaluating the effectiveness of these solutions is challenging. Every new scenario in which the sensors are deployed presents a new problem to be solved. Those wishing to measure performance must rebuild, reevaluate, and re-measure the network performance manually. Physical implementations can measure physical performance (such as how well actuators and plant respond to commands), but learning about network behavior in simulation requires that one understands what kind of communication relationships are formed between communication elements. These relationships will be different for each new configuration, and measuring their performance quickly will present difficulties as proposed physical deployment becomes more complex.
1.2 Sample Problem

An application of these ideas of directing and reacting to changes in the environment is pathfinding and directing pedestrian or auto traffic. First, let us look at an example of how traffic is directed at present, and then one can see how this could be solved with sensor networks, using our “Distributed-in/Distributed-out” model.

Traffic signs direct drivers to a certain goal by providing the next immediate local instruction rather than providing all of the instructions at once. For example, a driver in lower Manhattan who wishes to go through the Holland Tunnel into New Jersey will see a sign directing him to drive south towards Broome Street. Once the driver reaches Broome Street, he will be directed by the sign to make a right in order to head westbound until reaching the on-ramp for the Holland Tunnel. (Figure 1-1) Notice that this solution provides limited, simple instructions about next steps to the driver’s goal until he reaches his destination. The signs make up a path from one part of Manhattan to the Holland Tunnel. (Figure 1-2) The traffic signs are static and have no intelligence, but their design demonstrates that the use of limited, next-step instructions is effective in directing users to resources. Other driving instructions can be provided dynamically, through electronic signs that provide information about detours due to changing conditions, though the changes are programmed in by traffic engineers when a route needs to be redirected.

Note, however, that coordinating these signs is generally done centrally. Even the changing of electronic signs to reflect changing traffic conditions is managed by a central office. Whether changed remotely or on-site, the decision to make changes would be based on data taken throughout the city and aggregated in a central office before being relayed to those responsible for updating any electronic signs or changing the arrangement of the standard signs to indicate detours. Such a centralized system may not be a bad idea for managing traffic in city streets. After all, the streets themselves do not shift often, and the traffic management authorities may wish to optimize the system for consistency, which requires some centralized decision-making, rather than immediacy, which would be helped by highly localized decision-making.
Figure 1-1: Clockwise from upper left: Traffic sign directs westbound drivers to turn left at the corner of Lafayette St. and Kenmare St. The next sign directs drivers to make a right onto Broome St. Third, drivers are directed to bear left at the Broome St./Watts St. split. Finally, at Watts St. and Varick St., drivers see the entrance to the Holland Tunnel. Note the electronic sign below that can be changed, depending on conditions.
Sensor nodes can provide a similar function. A decentralized application of these ideas can be found by using sensor nodes to provide next-step instructions to a new point in a path which contains another sensor node. A user following the instructions from these sensor nodes will ultimately reach his or her goal. As the sensors detect changes in local conditions, the nodes change their instructions to provide the path for the user. In the event of a failure of a node, surrounding nodes can simply redirect their instructions to the next-nearest node on that path.

Take the example architecture in figure 1-3. Each intersection is marked A, B, and C. Intersection C is adjacent to A and B, but A and B are not adjacent to each other. At each intersection, there is a transducer that monitors traffic in the intersection and receives data from adjacent transducers about traffic conditions. The transducers take their local data and data received from adjacent intersections and adaptively change their signs indicating the suggested route for drivers.

In contrast to the scenario of unintelligent street signs, data regarding changes in traffic conditions is not aggregated centrally, nor are decisions about how to redirect
Figure 1-3: A basic architecture of intelligent signs with peer-to-peer communication
traffic made by a central coordinator. Instead, data about conditions is gathered locally, and instructions to other local sensor nodes are sent locally. The system of next-step instructions is reconfigured locally, based on the decisions of the sensor nodes at the site.

Looked at in terms of DiDo networks, the “input” of the traffic conditions is distributed throughout the area as the input data is read in from the sensors distributed throughout. Meanwhile, that data is distributed outwards through the network to the other sensor nodes which adjust their signs accordingly. The drivers will presumably follow the signs, changing the flow of traffic.

Use of sensor networks to provide local instructions mimics a well-known method that is used to provide geographical instructions. The idea of distributing instructions over a wide area in order to simplify the delivery of those instructions is not new. However, my proposal for sensor networks that distribute the output based on the data gathered is naturally applied to this application. Instead of applying this architecture to cars driving through streets attempting to escape New York, I will apply it to pedestrians walking around a building’s hallways seeking to evacuate.

1.3 Approach

Problems can be solved in a distributed fashion with local data augmented by neighboring data sent by intelligent sensors and actuators providing “next step” instructions, dividing up responsibility for reaching resources between those intelligent sensors, in the same way that traffic signs provide the “next step” for drivers on their way to a destination. I perform a set of tests to measure performance of the system to see how well the system handles reading input over a distributed area and changing its output accordingly. I measure how the system reacts to changes in the environment (changes in the “ Distributed input”) and how quickly it can redirect instructions (changing its “ Distributed output”) and measure the recovery time and performance of the protocols used to solve various network issues that may arise.

Again, consider the case of building escape. One can consider the exits as “re-
sources” that the people in the building are trying to reach. The pedestrians are given instructions about which direction to travel in by actuators that indicate the best path to the exit. Other problems can be solved in a similar fashion by understanding which components of the problem correspond to the resources, actuators, and paths. As another example, consider the sprinkler system for fighting fires in a building. Different types of fires require different types of methods and chemicals, and using data from sensors to analyze the type and extent of fires is starting to catch hold in order to provide a more detailed fire scenario to firefighters.[22] Rather counterintuitively, I designate the fires to be the “resources.” The sprinklers are the “actuators” directing the extinguishing agent to reach those resources, in this case, the fire. The sensors detect the fire, their distance from the fire, and instruct the nearby sprinklers to act in the appropriate manner, depending on the location of those sprinklers relative to the fire.

The general framework is one in which the sensor network must also account for changes to the environment caused by actuators as they interact with the network. Within this framework, problems of redistributing data through the network to those who need it can be solved without a central coordinator. The high-level protocols and architecture are more than a specialized means of solving a specific problem; they are part of a reusable infrastructure for locating resources and distributing data where and when it is needed.

I will present some background on other implementations of sensor networks and discuss possible shortcomings and where DiDo networks can help. Next, I provide evidence that solutions to path-finding problems can be solved efficiently by distributed sensors. I then present three novel contributions. I present a tool that allows users to design different problems related to pathfinding in DiDo networks and automatically generates the code to be programmed into the sensors themselves. This tool allows a user to design the layout of the physical world in which the sensors are to be deployed. This tool also creates the scripts necessary to analyze the performance under different scenarios using existing network simulators. I use this tool to create a model for building evacuation using a set of primitives appropriate for that solution.
In evaluating network communication patterns, I show that using sensor networks, ad hoc communication can scale. I discuss issues that need to be addressed in terms of physical deployment of the system. While most of the work here is done in simulation, mention must be made of the practicalities involved in actual implementation, and I show the form that a hypothetical physical deployment would take.

Finally, I develop a new pedestrian simulator that allows me to evaluate how well the DiDo system augments users’ decision-making abilities in the physical world. This integrates with existing network simulators, and “completes the loop” between the physical actuators and the sensors that monitor and send instructions to the physical world. The physical simulation influences behavior within the network simulation in order to test all aspects of the DiDo system’s performance. I show that the network protocols are intelligent and adaptive and reconfigure themselves to changing physical conditions and effectively augment the experience of the pedestrians in the physical world, providing greater utility. I show that performance of the pedestrians improves when decisions made by the pedestrians is augmented by instructions from the sensors and actuators.
Chapter 2

Background

2.1 Introduction

In this chapter, I consider various existing sensor nodes and sensing systems and the communications techniques that connect them. The purpose of this exposition is two-fold: to familiarize the reader with the diverse ways by which sensor systems have been designed and deployed and to demonstrate the utility of extending their change of application and capability with “Distributed-in, Distributed-out” (DiDo) techniques, the core work of this thesis. Furthermore, a review of sensing technologies provides a model to the user regarding how a real-life implementation of a DiDo system can be realized when one moves beyond simulation. I proceed by categorizing sensors and their associated networks into a taxonomy, at least some of the components of which are useful and used in the world today. Then, I show which parts of that classification could be improved by the techniques proposed in this thesis, and finally, I add a note about the underlying “viral” networking techniques that enable those networks both to scale and to evolve at reasonable cost.

The uses and technologies of diverse sensor networks are important motivators of the work because, as we will see, there are a great many barriers to widespread use of some of the intuitively beneficial sensing, reporting, and control systems. They are often costly, either in terms of installation or maintenance, they often are considered in a vacuum, as if no other network or sensing system was nearby, and they are often
difficult to program, manage, and evolve to suit changing circumstances. The devil is truly in the details.

To date, peer-to-peer sensor networks are relatively uncommon in day-to-day life, and use of autonomous sensor networks to solve day-to-day problems has not taken hold. However, when monitoring distributed phenomena, the logical choice would be to deploy monitors that could intelligently accumulate and share information about the area of interest. Ideally, these sensors would act in autonomous, independent ways, yet function as a coherent system. I argue that peer-to-peer sensor networks will become more common for monitoring distributed phenomena and problem-solving when the process of programming and deploying them is eased.

Interest in sensor networks flourished when it became possible to place more intelligence in the sensors themselves and give them their own displays. Developers of Smart Dust [41] pointed to the trends of “complete systems on a chip, integrated low-power communication, and integrated low-power transducers,” as forces that made intelligent networks of sensors possible. Increased processing power allows more complicated, fully-featured operating systems to be implemented on a small hardware platform. Low power-consumption allows the nodes to be deployed in remote areas without access to an external power source and without the need for their internal power source to be replaced very often. In addition, complex communication protocols and more full-featured programming methods have allowed other researchers to explore ways of coordinating communications protocols in order to exploit more power efficiency, as this is a crucial component that allows such networks to be realized. [38], [39]

In essence, the sensor network I develop in this thesis combines scalable radio networking with both sensing and actuation. I hypothesize a fully distributed system where much of the computing associated with sensing some environmental and personal attributes is done locally, and where the result of that computation is likewise used, or presented, in a distributed way. Archetypal cases are, for example, a traffic system that senses vehicle density and suggests alternate routes for cars already on the road, or a building evacuation system that both detects an impending disaster,
such as a fire, and presents escape routes to the occupants. Later, I will show how a DiDo network can improve both existing and proposed sensing systems, and how it can lead to novel new ones that were not considered either feasible or “easy.”

2.2 The Taxonomy of Monitoring Sensors

One can consider existing sensing and monitoring systems along several independent dimensions defined by their design and application parameters. For example, there are sensors that are optimized for low-power and long life, those that include such sophisticated computing that one might classify them more closely to a distributed computer system with a sensor attached than a sensor at all, and those that are either fixed, movable or autonomous.

There are other issues of physical implementation and deployment that are relevant to our inquiry:

- Mobile or Fixed: Some sensors are deployed to fixed locations; others are movable, occasionally under their own power.

- Large area or local: Are the sensors within a navigable area such as building, or are they spread along a battlefield or seacoast?

- Wired or Radio: Some sensing arrays are hardwired, others use radio. The latter have different interconnection constraints and features

- Low cost or High Capability: Much attention today is focused on large scale, extremely low-power sensing systems, but there are more sophisticated applications where the sensors can cost quite a bit more and have access to essentially limitless power.

In addition to physical considerations of sensors, there are issues relevant to communication and data monitoring that are of some relevance:

- Scalable or Pre-configured - Can the system be seamlessly expanded, or is it pre-configured with a size limit?
- Serial or Integrated: Are the sensors used to produce a discrete set of sample points that represent an environmental variable, or are they used in an ensemble to create a “gestalt” or “feel” for the space as a whole?

- Distributed Output or Centralized Output: Is the output from the data aggregated to a central point (Centralized Output) or are the readings distributed throughout the network to interested inquirers (Distributed Output)?

- In-Situ Signalling or Centralized Data: does the system provide signals to inquirers in their immediate location, or do inquirers consult the central aggregator?

I examine these options with the following illustrations that take various slices through this space to show where sensing systems have been deployed. Unlike other taxonomies, the features of various sensing systems reflect much overlap. These sensors cannot be categorized as a simple tree, as there is much overlap in features. Sensors with distributed output, for example, can be both mobile or stationary, use both low-cost and high-capability sensors and provide can be found providing both serial and integrated data. What is more interesting is finding which features are tightly coupled together in order to determine if this coupling is a seminal one or whether the boundaries and couplings of sensor system features are necessary or if new technologies allow these features to be treated in new ways, providing areas for which DiDo can be developed.

I present here some examples of the various sensor systems and research efforts, both to show how diverse the applications and technologies are and to highlight work that either informed DiDo systems or would benefit from them. First I examine some basic sensor technologies and what kind of sensors were used to attack certain problems, such as tracking and detection, which are specifically relevant to DiDo applications. I then go on to discuss implemented systems of sensors and sensor networks and describe the challenges faced within changing dimensions of the sample space.
### Table 2.1: Features of basic sensing systems

<table>
<thead>
<tr>
<th>Sensor Feature</th>
<th>Intruder Detection</th>
<th>Traffic</th>
<th>Environmental</th>
<th>Tracking</th>
<th>Battlefield</th>
<th>Distributed Control Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wired</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wireless</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>In-Situ Signalling</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Centralized Signalling</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Local Area</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Large Area</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Serial Data</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Integrated Data</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Centralized Decisions</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Distributed Decisions</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Viral</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Actuation</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

#### 2.2.1 Basic Detection

By “basic sensing,” I refer to early sensing systems that perform a single, simple function. I divide these into categories of intruder detection and basic traffic sensors whose properties are noted in table 2.1. One early example of the use of sensors is railroads. Sensors that detect the presence of an oncoming train nearing a railroad crossing activate signals at the railroad crossing to stop oncoming road traffic. These sensors are not intelligent: they detect a phenomenon and react accordingly. Simple alarm systems fall into this category as well: when a connection is broken, such as when a window or door is opened, an alarm sounds. Neither of these examples use particularly advanced sensor technologies—microphones are commonly used, along with inductive loop sensors.[74] Similarly, these sensor systems that are deployed over an area limited in scope find themselves applied to tracking applications and routing of traffic or autonomous robots.

Microphone arrays [26] are examples of attempts to monitor moving objects. When these systems analyze the inputs from the microphones, one can determine the
location and velocity of those objects that pass in front of the microphone arrays.[37] Extracting the data out of the sensors (i.e., the microphones) is easy: there are wires strung from the microphones to the analyzer. The raw acoustical analog signals travel from the sensors to the analyzer, and details about the monitored phenomenon are derived from the data by the central intelligent analyzer which makes some complex calculations on the data to extract an understanding of what is happening.

Simple acoustical detection designs have also been found in traffic sensing[15], the detection of intruders[87], and object tracking.[13] Other forms of intruder detection can be accomplished via magnetic sensors which are designed as gradiometers, using 3-axis sensor readings to measure magnetic gradients and a reference sensor to measure the background field.[5] Differences between the 3-axis sensors measure the gradients. The magnetic sensors are a highly preferred sensing technology compared to others in that they can be deployed without special packaging, operate regardless of orientation and have a reasonable signal processing cost associated with them. However, because magnetic sensors can only detect targets that contain or carry ferrous material, intruder detection systems generally also require some form of radar-based sensing or microphones, in the event that the intruder is an unarmed person, for example. However, similar magnetic sensors placed in a roadbed have also been used to count vehicles passing through that roadway in order to predict traffic patterns.[49]

Other forms of detection are vision-based. Detecting incidents and accidents in traffic, for example, has been explored using a video camera connected to a video analyzer to detect the existence of accidents and differentiate normal traffic patterns in intersections[47] or highway on-ramps [30] from unusual incidents. Vision-based detection appears to select for systems that depend much more on centralized decision-making. In the case of [30] and [47], the data from the cameras is analyzed at a central base station, and decisions about which signals to send to drivers are to be made at that central point, rather than having the decision-making process distributed.
2.2.2 Environmental Sensing

Some sensor network applications focus on environmental sensing, embedding ad hoc communication nodes with temperature sensors, light sensors, and wind speed sensors [86] and others including barometric pressure sensors and humidity sensors.[58] In many cases, the observers wish to find specific information about the environment, such as the temperature of a certain region or whether intruders have entered a restricted area. These are examples of sensor networks with a centralized architecture being used to deliver “serial data” and one sees them applied in the above-mentioned environmental systems such as TephraNet[86], the Great Duck Island sensors [58], and ArborNet.[66]

The Naiades project [52] creates an entire framework for environmental sensing and analysis. The Naiades sensors also incorporate the sense/actuate model, monitoring the environment in a lake and reporting their data back to a central system which then redirects the actuators based on those results; I will return to this project when I discuss sensor systems that fall under the category of distributed control. All of these sensor systems fall under the category of environmental monitoring systems. Monitoring conditions near a volcano presents a challenge because of its remoteness. Habitat monitoring is a daunting problem facing researchers because of the need to avoid disturbing the habitat. When sensors can be rapidly deployed in a small form-factor with a minimum of maintenance of infrastructure, suddenly many of the stumbling blocks to environmental monitoring are overcome. Furthermore, all of these cases depend on using the multihop capabilities of the sensor networks to aggregate data back to a central organizer.

2.3 Battlefield Sensors

Battlefield sensors are generally concerned with the issue of monitoring a large area. Battlefield commanders are interested in the movements and locations of enemy forces in order to have a better idea about how to deploy their own assets. Before the advent of ground-based, small computational sensors that could be easily deployed in a
battlefield and have their data relayed wirelessly back to commanders, the military used—and still uses—unmanned aerial vehicles (UAVs) to monitor battlefield conditions. Multiple UAVs can monitor hundreds of thousands of square kilometers per day[25] and report their data back as visual images to field commanders. However, the sheer amount of raw data makes it unwieldy to digest by any one person, and thus a large amount of intelligence needs to be embedded in these aerial, mobile sensors. Rather, the pre-processing and analysis of data gathered by these sensors serves to accomplish the specific intelligence goals.

The sensing technologies behind these UAV-based battlefield sensors are much more advanced than one sees in applications such as intruder detection. In one example application, one set of UAVs uses a Synthetic Aperture Radar (SAR) to analyze the terrain and detect stationary vehicles.[25] When one of these UAVs realizes that a previously detected vehicle is gone, responsibility is handed off to another set of UAVs using Moving Target Indicator (MTI) radar. The data is analyzed and the responsibilities are coordinated by the command and control center. The process of tracking moving targets is handled by these central analyzers which use algorithms to identify crucial routes around which traffic is moving.[45] The mix of different radar-based sensing capabilities in [25] is argued to improve the ability to identify and classify important targets and thus makes a compelling argument for the use of a diversity of sensing technologies for battlefield applications. This bias in favor of advanced sensing technologies means that using mobile UAVs is going to be more effective for economic reasons. The sunk costs of developing a UAV already means that the additional costs of adding complex sensors, rather than simple sensors, is going to be economically worthwhile.

The battlefield also provides new methods of sensor aggregation not normally available in other situations. While environmental sensor networks often aggregate data and send it to a central base station and deal with the means of routing data there[39], [5], the battlefield provides other means of routing data to a central aggregator. Specifically, in a battlefield situation, there may be constant patrolling of the region by UAVs, and UAVs can act not only as sensors, as described above, but also
Figure 2-1: **Left:** When aggregating to a base station on the same plane as the sensors, sensors may use multihop protocols to aggregate the data. **Right:** When the base station is on a plane above the sensors, sensors can take advantage of their numbers to aggregate the data by using cooperative diversity.

as aggregators, as data from the ground is transmitted into the air. When the central aggregator is on the ground, the other ground sensors facilitate aggregating the data by using geographically closer transmitters to forward on packets back to the aggregator. By contrast, when a large number of sensors are deployed on the ground, but the aggregator is above them, there is no straightforward method of geographically routing data packet-by-packet from the ground to the air. In this case, the authors in take advantage of cooperative-diversity methods (Figure 2-1).

Note that both the sheer amount of data being gathered over such a large area and the complexity of that data biases these systems in favor of centralized aggregation of data both to provide information to battlefield commanders as well as centralized analysis and decision-making at the base-station. The next step is to see whether the responsibility can be distributed among sensors for applications such as tracking.

### 2.3.1 Sensors in Distributed Tracking

Tracking is a specific example of detection, and while tracking has been use with battlefield UAVs, tracking can also be accomplished in a more distributed fashion, and the data can be disseminated in a more distributed fashion, as well. These
previous examples should not indicate that all sensor networks serve the main purpose of distributing their local data back to a base station. In fact, many sensor network applications rely on a high level of distributed cooperation among sensors at the peer-to-peer level. One example of this is the use of sensors for distributed tracking. As an object passes through an area saturated with sensors, such as cameras, the sensors must cooperatively determine the location of the object and then hand off responsibility for tracking the object as it moves through the area.

Tracking systems that depend on very simple sensor capabilities do, in fact, lend themselves to centralized architectures, such as when sensors only supply one bit of information.[6] As the same time, with more complicated data sets, gains in processing power now allow sensors with cameras to perform their own image processing and make independent decisions about tracking and surveillance. Collins [20] uses a centralized model in which cameras perform their own image processing before sending their data to a central organizer which returns the relevant data to the user, freeing him from having to monitor each camera himself. Foresti and Snidaro [29] demonstrate that an object-tracking system can be built in which low-level nodes perform low-level processing in order to improve tracking performance and do the base-level recognition. This data is passed to higher-level nodes which aggregate received data in an area to perform object-tracking analysis after all of the video data has already been processed by the lower-level nodes. This system is not completely decentralized, but rather acts as a hierarchy with explicit clusters that communicate with higher-level leaders but not with other clusters. Similarly, Horling, et. al.[42] describe a system in which a region divided into sectors cooperatively manages the sector tasks related to tracking an object. The sector manager assigns tasks to sensors who have the relevant capabilities required to detect the moving objects, and the sensors report their data to the sector manager. Another agent uses this data to estimate the track being traced by the object and the sector manager uses the calculated track to assign other sensors to the task of tracking the object as it moves to other parts of the network. Rather than depending on a central organizer, this latter system uses a more decentralized, cooperative for assigning tasks and reporting results. Yang
and Sikdar[89] also present a system in which responsibility for tracking is diffused throughout the network. However, in this case the sensors organize themselves into clusters, and the cluster heads manage which sensors are responsible for tracking objects and which new cluster head is given responsibility for tracking that object as it moves through the network.

A more decentralized model for tracking, which also takes advantage of the computational power of the sensors, was designed by Chong [17]. This model concentrates less on forming an explicit hierarchy, as seen in [29], [89], and [42] and instead focuses on how sensors can collaborate to track an object in order to predict and designate which sensors need to be turned on at a given moment in order to continue tracking.

However, when sensors are more intelligent, sensing systems can organize themselves to distribute responsibility for tracking an object as it passes through the network. For example, giving users access to tracking data is also an important issue. Zhao, Shin, and Reich[91] describe an architecture that is much more decentralized than all of the above-mentioned tracking examples. In fact, like DiDo, the authors foresee an architecture in which users make queries of sensors within the network in order to find information about any objects being tracked. Rather than aggregating data to a central base station, data is dynamically routed to the point of query. Furthermore, instead of an architecture in which groups of sensors are managed by intermediate organizers, such as cluster leaders, tracking is instead managed by sensors which make decentralized, intelligent decisions about whether to track an object moving through the network by exchanging information with their neighbors using that data to calculate individual “beliefs” about whether to continue tracking an object in their vicinity.

In tracking applications, once again we see cameras being combined with image processing, such as the centralized model of Collins [20] in which cameras perform their own image processing before sending their data to a central organizer which returns the relevant data to the user, freeing him from having to monitor each camera himself. Foresti and Snidaro [29] demonstrate that an object-tracking system can be built in which low-level nodes perform low-level processing in order to im-
prove tracking performance and do the base-level recognition. These above mentioned vision-based tracking applications find themselves used in relatively limited areas. For large-scale tracking and detection over hundreds of thousands of square miles, UAVs can take visual images from the air and report their data back to field commanders and use separate types of radar-based technologies in order to differentiate stationary vehicles from moving ones. [25]

These above examples exhibit a pattern of increasingly decentralized behavior. [91], in fact, reflects the most decentralized model in which the process of tracking is completely decentralized and cluster formation is handled dynamically. However, lacking in their system is the concept of physical manipulation within the sensor environment. While the sensors organize themselves dynamically and can route results to inquirers at any other place in the network, there is no support for affecting and changing the physical environment. As we will see, I seek to combine the ability both to detect changes in the physical world and to make changes in the physical world.

Distributed sensors that are augmented with actuators as well as well as monitoring and making decisions in a distributed fashion can solve complicated communications problems. DiDo systems combine these two features, and below I will explore both the idea of distributed control in sensor networks and the idea of viral communications.

2.4 Distributed Control

Compared to the earlier examples of sensor networks, use of sensor networks in control systems adheres closer to the DiDo model in that sensors are being used to direct the movements of agents within the system and affect the environment. I examine their architectural models compared them to the model envisioned for DiDo networks.

2.4.1 Basic Control Systems

Basic control systems work in a straightforward manner— an input is placed into the system, resulting in a corresponding output. The result of the output may change
One of the key distinctions to make in control systems is the difference between open-loop and closed-loop systems. Open-loop systems do not have their inputs depend on their outputs. In an example given by [84], a toaster is an example of an open-loop control system (Figure 2-2). The heating element in a toaster is controlled entirely by the timer, regardless of the state of the output, which in this case is a piece of toasted bread. A similar argument could be made that basic alarm/escape systems are open-loop control systems. For example, in the event of a fire, an alarm goes off, and escape signs may be illuminated, directing escapees to the exits. However, the signals that control the signs are not affected by the output signals of the signs or the flow of escapees out of the building.

What if, for example, a toaster’s energy to heating elements was reduced when the
system detected that temperature of the heating elements became too high? What if the signals to the escape signs changed due to the movement of the escapees? These would be a so-called “closed-loop” control system (Figure 2-3) in which feedback is applied to the system. The system output is fed back into a function that changes the reference input, and this can be used to regulate the input to bring the output closer to the desired result. Examples of these closed-loop control systems in everyday life include home heating systems which regulate themselves according to the current temperature in the home or a toilet tank which stops filling once the water level has reached a certain point, ensuring that the tank does not overflow.

2.4.2 Distributed Control Systems

While a basic control system as an input and an output, a distributed control systems has many sensors reporting data and many actuators being affected. The term “distributed control,” however, has frequently been used to refer to a centralized control strategy that aggregates these inputs and directs the outputs (Figure 2-4), also known as a multi-input, multi-output (MIMO) control system. One example is a robotic system[72] with many separate camera sensors whose video data is sent to a control system that coordinates the movement of a robot’s actuators. The cameras then detect the results of the actuators’ movements, and the control system reacts accordingly.

By contrast, a DiDo-based system, like decentralized control systems, is truly
Figure 2-5: A model of a decentralized distributed control system, also known as a networked control system

distributed in that each node is embedded with its own sense/actuate system. Information about the state of the system is fed back into the sensors, and each individual control system readjusts its own actuators (Figure 2-5). This sort of decentralized MIMO control system can also be approximated as a group of single-input single-output (SISO) control systems tied together. However, this data needs to be communicated to the other individual control systems. In effect, the nodes are taking sensed data and transforming it into communication signals to be sent to other nodes. One could thus liberally refer to one of these sense/actuate/compute/communicate systems as a *transducer*, which is normally defined as something that converts one form of signal energy into another, in this case sensor signals are being converted to communication signals. I will refer to intelligent in these distributed systems as transducers in the remainder of this thesis, with the understanding that the term refers to intelligent nodes containing sensing, actuation, computation, and communication elements.

The issue of communication is important in these fully-distributed control systems which is why they are differentiated from the above-mentioned traditional MIMO “distributed control” systems depicted in figure 2-4 by referring to them as “networked control systems” [82], depicted in figure 2-5. Introducing this other additional issue of communication between transducers and the delays involved in getting information from one transducer to another creates additional effects that need to be accounted for in the architecture.[90]

A DiDo-application, such as one that might control the movement of traffic or
people would fall under the category of a “hybrid control system,” as it has both continuous-time events being managed by a discrete-time controller. While DiDo is unique in that it demonstrates the efficacy of a control system in directing pedestrians, much work has been done to rigorously model hybrid control systems.[11],[77] Specifically, work as been done in designing controllers for hybrid systems specifically for applications related to danger-avoidance[81], an issue directly applicable to some sample applications of DiDo.

Issues apparent in “network control systems” are going to be the ones that affect wireless sensor networks systems used for distributed control. An example of such a networked control system is the Naiades project, mentioned previously, which covers environmental applications.[52] The reason such systems are referred to as networked control systems is apparent in figure 2-6. As one can see, the embedded sensor/actuators are connected by a common communications network. This could be wired or wireless, though for almost all purposes explored here, the communications network will be wireless. Such a system has certain physical actuation goals that need to be met, and the ability to meet those goals is going to be affected by network performance. Lemmon, Ling, and Sun[53] acknowledge this fact and analyze the consequences of dropping packets when a network becomes overloaded. The consequences revolve around how changes in network behavior will affect the actuators’
goals. In fact, I also will analyze the effect of bandwidth changes on DiDo systems.

Some examples of control systems applied to sensor networks can be found in the field of mobile robotics. To provide a contrast between two different forms of control, one may consider the RoboCup Organization[2], which sponsors a soccer-like contest to be performed between autonomous robots. Within the organization, there is a Small Size League and a Middle Size League. Competitors in the Small Size league may use a global vision system whose data can be provided to the individual robots, while competitors in the Middle Size league are limited to only the data that they themselves sense and that which is exchanged with other competitors. The physical architecture of the Small Size league is thus very centralized, with a control workstation that issues commands to individual robots based on the result of the globally sensed vision.[21] The problem may be solved by the set of robots by using a central task assigner which calculates which tasks need to be performed in order to accomplish a specific goal.[23] This is closer to a MIMO-based model in that multiple inputs are being aggregated by a central controller and directing the behavior of several robots, the multiple outputs. By contrast, the system architecture of the Middle Size league depends more on multi-agent systems in which robots exchange sensor data with each other in order to agree on which roll they will play at a given moment in the RoboCup contest.[61],[14] The work of [14] provides an interesting solution to the sharing of sensed data by having each robot exchange not raw sensed data but rather a utility function based on the sensor readings such as the robot’s proximity to the ball and whether obstacles are sensed between the robot and the ball. This is an example of a control system exchanging “integrated data” between the sensors as a means of coordinating tasks.

Another of sensors used in distributed control is the pursuit-evasion game (PEG), in which an autonomous pursuer attempts to catch an agent attempting to actively evade the pursuer. The solutions to the PEG using sensor networks at UC Berkeley depend on their centralized systems. One implementation [48] uses sensors attached to the pursuers themselves, and the pursuers report back to a central map builder and strategy planner which coordinates the pursuers. Another implementation [76] uses
sensors spread throughout the field of pursuit to detect the evader. These sensors report back to a central base station which actuates a camera to the location of the evader. However, the camera’s ability to find the evader is dependent on communication with the base station. The work in [76] deals with the issue of distributed control head-on, observing that a robust implementation of a PEG-system requires that the system be rigorously modelled with control-system principles in mind. Specifically, such a system must content with communication delays, time skew between clocks, and independent, discrete decision-making on the part of the agents within the system. These systems, like DiDo, reflect a hybrid automaton model [81], as the system will be controlled both with discrete-time and continuous-time controllers, reflecting both discrete events communicated between the transducers and continuous data captured by the transducers themselves.

Finally, the pathfinding application of Li, De Rose, and Rus [56] uses a sensor network to direct a robot along the best path to a final destination point. This system depends on direct interaction with the sensor network to create a safe path to the destination by having sensors in “danger” areas send out a signal that blocks potential pathways that pass through those areas. The distance vector indicated at each node will account for paths that go around “danger” areas, and the robot will follow those paths.

What is notable is that none of these projects attack large issues that are addressed by robotic path-planning researchers. The challenge of many robotic path-planning projects is how an autonomous robot can avoid falling into “local optima” when seeking an optimal path. Simple greedy algorithms may trap the robot in a disaster area. Extracting itself out of that area may result in a longer total route than if the robot had not used such a greedy pathfinding algorithm in the first place.

The reason that so many traditional pathfinding algorithms need to depend on heuristic methods such as genetic algorithms[80] is because the robots doing such path planning are blind to the global environment; the robots depend on information from their local set of sensors. In these circumscribed areas, direct interaction with sensors which issue commands based on “serial” data representation – specific location data
### Table 2.2: A Comparison of Features in Distributed Control and DiDo Systems

<table>
<thead>
<tr>
<th>Sensor Feature</th>
<th>Distributed Control Systems</th>
<th>DiDo</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wired Communication</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Wireless Communication</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>In-Situ Signalling</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Centralized Signalling</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Local Area</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Large Area</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Serial Data</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Integrated Data</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Centralized Decisions</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Distributed Decisions</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Viral</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Actuation</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

related to the tracking of a specific object and returning data about its location.

### 2.5 Viral Networks

Decentralized distributed control systems would seem to be the systems most similar to DiDo systems, in terms of architecture and application. Using similar criteria to the earlier-outlined sensor taxonomy (Table 2.1), it is useful to compare the distributed control systems and DiDo properties side-by-side. As can be seen in table 2.2, one of the significant features that separates DiDo systems from other distributed control systems is that DiDo is intended to be a *viral* system.

Scaling and controlling a system is as important for a communications network as it is for a collection of sensors and effectors. While that has underlying technologies that are different (radio and networking protocols) from the ones I dwell on for sen-
sors (distributed programming, interactions with physical elements), there is much in common in the two domains. Indeed, it is work on viral communications system that inspired and guided the thesis.

By “viral networks,” we mean networks in which the total throughput is increased as more members join and in which the members of the network participate and cooperate in the process of routing network content from the source to the destination. These are networks in which the users carry their own infrastructure with them. The ability of these so-called viral networks to operate depends on a large amount of intelligence being placed at the ends of the network. The guiding principles of viral networks are that the network be scalable, incremental, and contributory. Some early inspiration for the idea of viral networks came from the area of distributed multicasting. In this application, the problem of multicasting is that as more listeners join, the additional listeners produce additional request for error corrections and additional requests for other missing data that they may need. Reliability schemes that work best push the error-correcting intelligence out the the ends of the network, rather than concentrating it at the server. Requests to fix lost data within a multicast community can be contained within a small group at the end of the network.[27],[28] In the case of listeners who wish to tune in to a video stream late and pick up what they missed, a multicast system can support the caching of video data locally. As a consequence, the video soon becomes distributed to many listeners. New listeners who tune in later can receive recently broadcast data in a “chain” from the sequence of listeners who have tuned in beforehand[75] or can receive fragments of the video they requested in a not only from the original source, but also from many other listeners that retain pieces of the data.[18] As more listeners join, they act as additional sources that new listeners who have tuned in can draw upon. New listeners can pick up not only the limited data available from the next most recent member of the multicast network, but also can receive missing data from any number of other members of the multicast network that can fulfill the request, in whole or in part. Many individual fragments of data from many different sources all of the network can combine to re-create the entire video stream for the new member of the network. The cases of [27]
and provide examples of clients that bring their own infrastructure with them—infrastructure that can be used by other clients to support the improved performance of the overall system.

In the wireless domain, both Gupta[36] and Li[55] have shown that the total capacity of ad hoc networks is $O(\sqrt{n})$. However, this assumes that all nodes are involved in communicating with each other—either randomly or in a scale-free fashion. With $n$ nodes in the network this means that the average bandwidth per node will be $\frac{\sqrt{n}}{n}$—a term that goes to zero as the number of members of the network increase. This is a problem that designers of sensor networks need to contend with—the networks may seem attractive because of their decentralized nature, but a naive design results in a system that does not scale. We seek to design networks where the utility for the individual sensor does not decline as more sensors join. In any design, the sensor’s available bandwidth should at least stay constant. I argue that this is possible for certain applications with sensors organizing themselves via an intelligently designed network communication protocol.

Greater intelligence on the part of autonomous sensors in an ad hoc network can also be leveraged to ensure more efficient radio communication. As argued by Chong [17] and demonstrated by Anastasi, et. al. [4], transmission and reception is one of the most significant factors in power consumption in sensor networks. By minimizing the power necessary to transmit, overall power consumption will fall. If the network protocols themselves can make decisions about which sensors they wish to communicate with, then decisions about power transmission radius can be made on the network communication level. By contrast, a case for variable-range transmission power control by Gomez [35] and Ramanathan [69] both argue for the advantages of transmission power control from the perspective of network connectivity. While [35] demonstrates greater available bandwidth capacity with smaller transmission radius, the goal of Ramanathan[69] is to maintain network connectivity while minimizing power consumption.

In additional to variable-range transmission power control, other network efficiencies can be gained by exploiting coordinated transmission, allowing receivers to com-
bine the signals of directly received messages from transmitters as well as repeated signals. Reconstructing messages based on the combination of these two low-power signals has been demonstrated using standard 802.11a transmission technologies and has been shown to provide substantial energy saving. The “best” relay paths can be chosen dynamically in order to exploit these benefits. In addition, Xie and Kumar go so far as to argue that in cases of RF signal absorption of high signal attenuation, bandwidth consumption will be bounded and transport capacity will scale as $O(n)$, particularly when traffic can be load-balanced using multi-path routing.

I argue that these methods are also applicable when the primary metric of connectivity is not simply network connectivity, but a network connectivity that reflects paths in the physical topology. The matter of primary importance in supporting this is the network-level protocols and support for the intersection of the physical world and the network world.

DiDo systems are those which are designed to take advantage of these viral features. In the next chapter, I will show the design of a protocol which can take advantage of adaptive RF protocols. This ability to take advantage of viral technologies and features is what differentiates DiDo from other distributed control systems.

### 2.6 Architecture of Sensor Networks

Most of the above-mentioned sensor systems depend on the existence of central base stations (Figure 2-7). The previous-mentioned pathfinding systems do not account for multiple users or dynamic changes in the topology of the network, nor does the action of the robots affect the results returned by the sensors when determining paths. Why are sensor networks designed in this centralized, hierarchical fashion in the first place? One could argue that one of the reasons is that the first instinct of designers is to approach a problem from the perspective of central coordinators. After all, so many systems in the world appear, at least at first glance, to have a top-down structure. A CEO at the head of a company appears to direct the operations from above to the employees down below. A flock of seagulls seemingly follows a leader at
the tip of the V-formation. Early computer systems were organized around a set of
dumb terminals which were doled out slots of processing time by a central mainframe.
Certainly the architecture is effective – the central coordinator can issue orders to all
of the clients below it in the hierarchy which is, at first glance, a rather efficient means
of implementation.

Perhaps the intuition of designers is wrong, and it would help to examine how
systems actually form, rather than rely on intuition. In fact, birds do not get into
formation by coordinating with a leader. Rather, their local interactions with each

Figure 2-7: The architecture of typical sensor systems. All inquiries are mediated
through the aggregating base station.
other are responsible for organizing into coordinated flight patterns [70]. This sort of flocking motion has been simulated using simple rules based on interactions with their environment to create patterns of order that only appear to be coordinated by a leader [71]. Some researchers who have studied the dynamics of large corporations have argued that the most successful companies do not depend on CEOs who are merely good at making decisions and predictions for the rest of the company by themselves, but rather depend on leaders who create a structure and a culture in which all separate units of the company share the same goals and vision. These autonomous units are designed to coordinate well with other units and are better able to form larger and more complicated structures that can operate more efficiently than companies that depend on the individual direction of the CEO [19]. Furthermore, in creating this sort of strict hierarchy, designers create a central point of failure and force communication with the top of the hierarchy in order for lower-level clients to operate and/or make new decisions.

Economics may also play a role in the design of centrally-coordinated sensor networks. The limited memory and processing power of small sensors meant that serious processing needed to be done by a central base station. Furthermore, such simple machines could be programmed only with rudimentary, low-level programming languages, meaning that only simple behaviors were allowed by the sensors at the ends of the network. With widespread application of the above-mentioned viral principles, this bias towards centrally-coordinated systems could change.

2.7 Simulation of Sensor Networks

Evaluating the performance of DiDo protocols in simulation requires simulation systems that acknowledge the effects of the physical environment. Network protocols can be evaluated without large-scale deployment through the use of simulation. Using simulation, the efficacy of protocols deployed on a large scale can be analyzed. Network simulators were originally developed and popularized for wired networks, where creative new protocols could be tested and understood. However, with the
increased popularity of sensor network development, simulation of sensor networks has also become a lively field. Furthermore, because of the unique role played by sensor networks in the monitoring of the physical environment, some sensor network simulators have begun to incorporate models of physical aspects of the environment. The ability to design physical environments in simulation will be an integral piece of any advanced sensor-simulation system.

A common network simulator used for analyzing terrestrial networks is ns.[7] Early work implementing new ideas about peer-to-peer multicast protocols was evaluated using ns. Recent applications of ns in the wireless domain have focused on proving the validity of algorithms to support time synchronization, for example. ns uses a Tcl-based scripting language to build networks and implement protocols. It has support for ad hoc wireless networking, though it only supports routing via the Ad-hoc On-demand Distance Vector (AODV) protocol[63], which is a pro-active routing protocol for ad-hoc networks. Some basic sensor-network algorithms can be implemented, but it does not represent an environment for deploying sensors.

As the issue of sensor network simulation became more important, other simulation environments were created specifically for the purpose of simulating sensor networks. The component-based network simulator J-Sim [1] incorporates the sensorm[62] system that accounts for wireless propagation models, network-level protocols, and different CPU architectures.

SENS, the “Sensor, Environment, and Network Simulator,”[78] incorporates the idea of accounting for the makeup of the physical environment when designing sensor network simulations. These differences in the physical environment can affect radio propagation. Examples of these physical aspects of the environment include concrete, grass, and walls that will affect characteristics of the radio transmission.

At the same time, the TinyOS [41] simulator, TOSSIM [54], was designed specifically to simulate the behavior of the Berkeley motes [83]. Code written in TOSSIM is directly exportable to the Berkeley motes. While it supports a variety of network propagation models, there is little support for work that accounts for the way the motes are deployed in the physical world. However, there is support for simulator
“plugins” that allow users to implement changes to the network propagation models and provide additional methods of visualizing activity in the network. Plugins may also be implemented that account for the physical world, and tests can be scripted under a variety of scenarios. However, as we will see, designing these scenarios and test scripts, even with the benefits of plugins, can be difficult and time-consuming.

Because TOSSIM does not support the ability to simulate a set of motes that run different code and cannot simulate heterogeneous networks of sensors and servers, Girod, et. al. [33] developed a system to support and evaluate heterogeneous systems based on the Emstar [32]. This allowed the researchers to evaluate how topologies formed and how the different forms of simulation and emulation compared to real-world deployment.

The common thread in all of these simulation systems is how to provide support for the evaluation of network models. We argue that for the evaluation of Distributed-in/Distributed-out systems, new methods of validation and scenario design will be required. While the field of sensor network development has been a lively one, the space of applications is constantly changing, and the testbeds for implementation of these applications are not well developed. As a consequence, developing new applications will require the development of more full-featured simulators to evaluate the efficacy of the application design.
Chapter 3

Experiences with Communication Protocols for DiDo

3.1 Introduction

In simulation, I implement various applications for DiDo networks. First, I provide theoretical justification for why using sensor networks to solve adaptive pathfinding problems in a graph should be scalable in terms of communication overhead. Next, I show that using actuators to direct the plant in the physical environment is similar to established methods known to route packets in the network environment. With this knowledge in hand, I look at some high-level primitives that can be used to implement the DiDo networks.

Given theoretical justification, a model for implementation, and a set of programming primitives, I present both a simple and an adaptive implementation of pathfinding. As a test case, the simple model is compelling. While it is not applicable to real-life deployments, the simple model gives us an idea of how DiDo networks can be built. The limitations of the simple model allow us to understand what new functions need to be designed to provide a more full-featured system that lines up more with real-life implementations. I describe this new design and discuss how it would be implemented and evaluated.
3.2 Pathfinding in a Graph

3.2.1 Representation of Physical Layouts as a Graph

Our system explores the problem of efficiently directing people in a building to the shortest escape routes. This problem is actually very complicated because the sensor network, like all large sensor networks, needs to analyze a large amount of data. For the physical pathfinding case, the data bottleneck is the large number of possible paths to the exit to choose from. Solving this escape problem can be mapped to a pathfinding problem in a graph (Figure 3-1). The sink in the graph can be represented as the exit. Each node represents a point at which a user can make a choice of which direction to go, aided by a sensor indicating the direction of the shortest path to the exit. The goal is to find the shortest path in a graph and to find a new path when the current one becomes unavailable.

Representing floorplans as graphs allows others to more keenly analyze constraints of a floorplan. As a consequence, tools of graph theory have been applied to the geometry of architecture. Applications of graph theory to architectural floorplans fall into two categories: constraint analysis and path finding. These two applications require differences in representation, which I will explore below.
Figure 3-2: A sample floorplan with noted dimensions with individual rooms maintaining their same label.

The first application related to representing floorplans as graphs, constraint analysis, presents some counter-intuitive representations. Take the case in which an architect is designing the layout of a floorplan but is constrained by a set of fixed dimensions of that floorplan. Taking an example from March and Steadman’s *The Geometry of the Environment* [59], the floorplan could look like the one shown in figure 3-2, whose dimensions are 20-by-21 meters. Representing this more formally would look something like the representation in figure 3-3. Taking the labels from the rooms represented in figure 3-3, the constraints on the room dimensions can be represented as a combination of potential and current flow, as depicted in figure 3-4. Here, the rules of circuits are applied to maintaining constraints on a floorplan, as the y-dimension is represented as potential and the x-dimension is represented as current. By representing the room dimensions of a floorplan as a graph that adheres to the constrains of current flow and potentials, an architect can be assured that his proposed layout is a valid one.

A more intuitive representation of physical layouts can be found in a problem in graph theory known as the “Bridges of Königsburg” problem from the 18th century. As can be seen in figure 3-5, the bridges are represented by an edge, and each destination is represented as a node.\(^1\) This formalized representation allowed Euler

\(^1\)Once again, this example and depiction is taken from March and Steadman’s *The Geometry of the Environment* [59]
Figure 3-3: The sample floorplan represented more formally

Figure 3-4: The floorplan represented as a combination of current flow and potential
to determine that one could not traverse each bridge only once and return to one’s starting point, because at most two vertices of odd degree are needed to do so, and such a constraint does not exist for the bridges of Königsburg.

Similarly, in an exploration of computer applications for architectural problems[60], graph representation similar to that used in the Bridges of Königsburg problem can be used to trace a path out of a maze. Taking a famous maze such as the hedge maze at Versailles (Figure 3-6), one can represent the junctions as nodes and the paths to the next junctions as edges (Figure 3-7). By representing the maze as a graph, one can more easily subject the maze to rigorous analysis and standard pathfinding.
For my purposes, the most applicable models of representation of floorplans is the latter cases of the maze at Versailles and the Bridges of Königsburg problem in which the destinations are nodes and the travel conduits are the edges.

### 3.2.2 Analysis of Paths to a Sink in a Graph

Let us first start with the most reduced form of an escape scenario – finding a path between a source and sink in a graph. If the graph, $G$, is fully connected and has $n$ nodes, path lengths may vary from two nodes (directly from the source to the sink) to $n$ nodes (a Hamiltonian path containing all nodes). In a graph of $n$ nodes, the longest-length path will be $n - 1$ steps. A unique path here is defined as a path that does not repeat any nodes and whose entire length has never been counted before. However, a unique path may contain an already known path within it. Thus, if the nodes are numbered and mark node 1 as the sink, I count paths 2–1, 3–2–1, and 4–3–2–1 as unique paths, but not 4–3–4–3–2–1.

Some simple analysis provides a starting point to count the number of paths that exist to the sink. A graph $G$ can be represented as a spanning tree $T$ rooted at the sink (Figure 3-8). In this tree, there are $(n - 1)$ unique paths to the sink from any other node. So one can say that the number of unique paths to the sink is definitely $\Omega(n)$. 

![Figure 3-8: A graph with an associated spanning tree rooted at the sink of that graph.](image)
Figure 3-9: In the case of a blocked path, the source node has to pick the shortest route by picking the smallest valid hop value passed to it from the nodes adjacent to it.

All of the possible paths can be enumerated by creating a spanning tree of nested spanning trees. If there are \((n - 1)\) unique paths to the sink using the spanning tree rooted at the sink, then each of those \((n - 1)\) nodes in the spanning tree itself as a spanning tree rooted at itself with \((n - 2)\) unique paths to that node, and so on. This gives an upper bound of \(O(n!)\) on the number of unique paths to the sink in the graph. Clearly, exhaustively searching through such a large number of possible paths for an optimal one is an intractable problem.

Finding a single shortest path in a graph is not difficult, as one can just use Dijkstra’s algorithm. However, quickly rerouting and finding a new path in the event that an edge is removed presents a challenge due to the large number of different possible paths in the graph; one wants to avoid the need to re-calculate an algorithm such as all-pairs shortest path, which has a cost of \(O(n^3)\). Even just rerunning Dijkstra’s algorithm has a cost of \(O(n^2)\) and, even in the best case, a runtime of \(O(m + n \log n)\) using a Fibonacci heap[43]. Our goal is to see if one can recover from a failure in a graph.

As an example of how these nodes will pick the optimal route, assume that each node wishes to know the shortest number of hops to the exit. A sensor informs its neighbors about its distance to the exit. A sensor receiving that information about the exit picks the link to the node that tells it of the shortest distance. In figure 3-9, nodes pass their knowledge up towards the source. Nodes adjacent to the exit (nodes m and n) know how many hops away they are from the exit node. In the event of a blocked passageway, such as between node q and the exit, the node at the source will
direct users to an alternate route, such as through nodes n or p. Note that in the
case of node p, there are many different links leading to the exit, but no node knows
the entire path between it and the exit, only the number of hops from there to the
exit and which link will lead that way. Knowledge of the entire path is distributed.
This is the same principle in networks as distance vector routing, which leads one to
suspect that similar principles may apply when trying to manage physical routing in
a changing environment.

Some comments must be made about how a network can understand the underly-
ing topology and treat a floorplan as a graph. In the building scenario, the destination
nodes at the ends of the paths are already known. In a non-emergency situation, the
network can determine optimal paths when the nodes in the network distribute data
about the exits to all of their neighbors.

3.3 Using Ad Hoc Network Protocols to Route
People, Not Packets

I implement this system using well-known protocols to organize ad hoc, multihop
networks. First, I argue that routing people through a physical network is analagous
to routing data in an ad hoc network, and thus the protocols for maintaining the
physical paths will be similar. As a simplified example case, I make the assumption
that packet hops are comparable to distance [56] when determining the distance to the
exit from a node. Next, I argue that when a path is blocked, the closest alternate path
will be sufficiently close that the amount of communication required to reconfigure
will be local. While assuming fixed-radius communication is not very realistic, using
that model serves as a initial “first pass” in exploring the use of ad hoc networks to
solve physical pathfinding problems.

An ad hoc connection formed between a source and destination has two kinds
of traffic – the actual data traffic between the two nodes, and the communication
overhead necessary to establish and maintain the connection between those two nodes.
Discovering and maintaining routes to destination nodes is accomplished either by discovering routes on demand when a node initiates communication or by having all nodes periodically announce their presence to their neighbors who then update their routing data.

In the on-demand, or “proactive,” routing scheme, an ad hoc route can be constructed when a source node sends out a search message to the destination node that floods the network until the destination node is found. Each intermediate sensor that forwards messages on records the message origin and hop distance to create a routing table. Using the routing tables maintained by the intermediate nodes, the rest of the messages in that communication can then proceed using only the necessary relay sensors.

Sending out periodic maintenance messages to indicate that a connection still exists is called the Destination-Sequenced Distance Vector (DSDV) [64] protocol. I refer to this as a “reactive” protocol, since nodes receive routing data passively rather than requesting it. The rules are simple – a sensor receives a message indicating the distance of the sender to the resource along with the sender’s x,y location. If the receiving sensor has no information of distance and direction to the resource, then it accepts the message and chooses the sender as the “next step” on the way to the resource. If the receiving sensor already has a known distance to the resource, then the message is accepted if the indicated distance is less than or equal to the known distance. If the message indicates a distance longer that the known distance, the message is ignored. Finally, the known distance is forgotten if no messages are accepted within a certain interval of time (usually equal to the time in which a sensor would send 4 messages). Compared to proactive routing schemes, a reactive scheme will require less overhead traffic when the network topology rapidly changes. However, a reactive scheme may impose unnecessary overhead by maintaining routes that are not used and constantly updating routes that are already stable [12].

In the scenario of building escape, I seek to maintain accurate paths to exits, just as an ad hoc network maintains shortest communication paths to a destination node (Figure 3-10). In my example, those accurate physical paths may not correspond to
the network topology. The sensors must thus remain aware of the physical topology and not confuse this with the network communication paths. All nodes need to be aware of the fastest pathways to the exits. However, as described above, keeping track of pathways and routes is part of communication overhead. The system is maintaining the paths for the specific purpose of routing people to those exits. Thus, there is no “data traffic” in this model. All wireless communication between nodes is path discovery and path maintenance. By framing the problem this way, one can get a good idea about how much network traffic will be involved in maintaining such a system.

Simple analysis shows why any network repairs can be done locally. In a graph of \( n \) nodes with average degree \( \delta \), then the average path length is \( \frac{n}{\delta} \). With \( \delta \) choices at each node, there are at most \( \delta^{\frac{n}{\delta}} \) paths from the source to the sink. With \( \delta^{\frac{n}{\delta}} \) paths, there are a total of \( \frac{n}{\delta}(\delta^{\frac{n}{\delta}}) \) nodes in all of these paths.

If \((\frac{\delta}{n} - 1) \geq 1\), then there are enough paths that a person at a node can get to the exit within one step from that node. Otherwise, there are an expected \((\frac{\delta}{n} - 1)^{-1}\) steps to the next path to the sink.

This is scalable because changing conditions in the network can be accounted for by making local changes. In the event of a blocked path, a user must go to another path. The expected difference in distance between a node and the the exit when an edge is removed will provide an estimate of how much reconfiguration will be required.

Figure 3-10: **Left:** A shortest path in an ad hoc network is created between source, \( s \), and sink, \( t \) for routing packets. **Right:** A shortest path in a floorplan is created to route a person at \( s \) to the exit at \( t \).
as the network increases in size. When an edge on a path is removed, the number of edges in the network falls from $M$ to $M - 1$. Thus, average degree $\delta$ falls from $\frac{M}{n}$ to $\frac{M-1}{n}$. How many steps must we travel before one can find another path? If any random node is $\frac{n}{\delta}$ steps away and $\delta = \frac{2M}{n}$, then $\frac{n}{\delta} = \frac{n^2}{2M}$. In a graph with $M - 1$ edges, the average distance to the sink will rise to $\frac{n^2}{2M - 2}$. The expected difference in path lengths, $P$, to the exit with the missing edge is given by $\frac{n^2}{2M - 2} - \frac{n^2}{2M}$. This reduces to

$$P = \frac{n^2}{2M(M - 1)}$$

(3.1)

and

$$P = \frac{2n^2}{n^2\delta^2 - 2n\delta}$$

(3.2)

Since $M \sim n$, one can see that the difference is constant as $n \to \infty$. Thus, all reconfiguration is local for networks of constant density as size increases. Whereas Li, et. al. [55] claim that network bandwidth will increase as $\sqrt{n}$, I argue that for our application, network bandwidth will increase as $n\sqrt{k}$, where $k$ is constant – based on the maximum radius in which the sensors will need to communicate with each other in order to make their repairs. The goal is to see if my predictions hold when I implement such a system and simulate failures and repairs.

### 3.4 Building DiDo Applications with High-Level Primitives

I seek to create a set of primitives designed to help sensor networks coordinate to provide information on resources and resolve the interests of those that interact with the network. Most sensor applications depend on low-level programming to handle the communications protocols and react to sensor data. I propose primitives that are intended to serve as a language for solving the resource discovery and pathfinding issues described earlier.

The goal is to create an extensible protocol that can incorporate the ability to find
paths to many different types of resources that can be added to the network. Intermediate sensors themselves act as resources (after all, a sensor acts as a passageway that leads a user to a resource). This provides a framework to build extensible and scalable systems.

Each resource has a built-in set of features based on what it can sense (whether the resource is occupied, the amount of congestion present, etc.). A resource propagates knowledge of its existence to its local neighborhood. The underlying network protocol responsible for this is left to the discretion of the implementer. The important matter is that the primitives used by the programmer to build systems to find resources are independent of the underlying network protocol used.

Some sensors, such as destination resources, play the role of informing their neighbors about their existence. Other sensors, such as “path” sensors, play the role of passing on information about resources.

Now that I have described some features of the data that will be sent, I will describe some primitives that implement this protocol in order to build these systems.

First, I assume a set of results that a sensor can detect with respect to the ability to reach a resource:

**Available**

**Unavailable**

Given these sensor readings, a sensor, depending on its function, can convey different pieces messages to a neighbor, regardless of the specific communication protocol used to convey the data. I reduce the data exchanged to a few specific directives issued by a sensor:

**AnnounceResource** If a resource is available, it announces its presence to surrounding nodes, along with information about the nature of the resource.

**AnnounceResourceUnavailable** If a resource is unavailable, then the node follows the network protocol which invalidates its existence to its neighbors. This may be done through explicit invalidation or simply by no longer periodically announcing
its existence, thus causing neighbors’ knowledge of that resource to “time out.”

**AnnounceDistanceToResource**  If a path exists between a node and a resource, then it announces its distance to that resource to its neighbors.

**AnnouncePathUnavailable**  If a path to a resource becomes unavailable because of changing conditions, then the node notifies its neighbors that the path is invalid.

The nodes on the receiving end of this information execute their own functions based on what they receive:

**ReceiveDistanceToResource**  A node receives information about a resource, including the nature of the resource, the distance away from it, and information about the path to that resource.

**ReceivePathInvalid**  A node receives information that a path to a resource is invalid. The precise nature of this message is a function of the protocol. It may be an explicit invalidation or a decision made because that validity of that path has “timed out.”

**AddPath**  A node adds information about a path to the list it maintains.

**RemovePath**  A node removes a path from its list because it is no longer valid.

**PickBestPath**  Out of all the paths that a node maintains, it will pick the best path, based on criteria specified by the programmer or other conditions. This can involve path length, dependability of the path, or flexibility (the number of alternate paths available in that direction), for example. This returns local information about the next step in the path, as well as the nature of the resource.

**DirectActuator**  Given the path, the sensor instructs actuators or gives other actors within the network the appropriate instructions to get to the resource or how to use the resource.

I have created a basic set of primitives that define paths to resources and a protocol to pick the best paths to a resource, irrespective of the exact sensing and display methods involved. In addition, the precise communication and routing protocols used to maintain these paths is left undefined. The primitives and high-level protocol presented here provide a structure that allows us to think about how to solve these pathfinding problems under different network and sensing circumstances.
Exit Sensor code:
if (Available) then
    AnnounceResource
else if (Unavailable) then
    AnnounceResourceUnavailable

Path Sensor code:
if (p = ReceiveDistanceToResource) then
    /* add 1 to the path length */
    AddPath(p+1)

if (Available) then {
    /* display the direction the user should go in */
    DirectActuator(PickBestPath)
    AnnounceDistanceToResource
} else if (Unavailable) then
    PathUnavailable

if (p = ReceivePathInvalid) then
    RemovePath(p)

The underlying functions of these primitives may change. For example, DirectActuator could result in any number of different reactions, depending on the application being used. DirectActuator could specify different behavior depending on whether the designer wishes actuators to react by traveling towards or away from the resource, or simply direct different behavior depending on distance. Announcing the existence or non-existence of available paths will differ depending on the underlying communication protocol being used.

3.5 Simple Communication Models

3.5.1 Pathfinding Using Simplified Assumptions

To construct a system that would support this pathfinding and resource discovery, I used the TinyOS simulator, TOSSIM [54], described in Chapter 2. First, I made some simplifying assumptions about the network model, designed a communication protocol that could understand the physical layout given that network model, and ran multiple tests to measure system performance. I will examine some simulation results and discuss experiences with solving the pathfinding problem.

TinyOS sensors are programmed using a programming language called nesC [31], a programming language designed specifically for the TinyOS to build applications.
by connecting together separate individual components. This allows us to separate responsibility for receiving data and sending data into different components. Since all nodes in a TOSSIM simulator must contain the same program, I feed nodes specific sensing data to indicate to them whether they are located at a resource node or a path node. In order to easily map message hops to distance, I assume a network model in which data sent by sensors travels in a fixed 10-unit radius around each sensor node. All sensors are aware of their location in x,y space, and they were placed in a topology in which distance could be inferred from hop count.

Both the “proactive” and “reactive” network protocols were implemented with these network model and placement assumptions in mind. The reactive protocol was a modified version of DSDV in which the sensor at the exit node periodically announces its presence and location to surrounding sensors. The nearby path sensors wait until they receive a message from the sensor at the exit node and infer the direction to the exit node using knowledge of their own location and data received from the sensor at the exit about the exit node’s location. Once the path sensor receives information about the hop distance and direction to the exit node, it periodically sends out that data to its neighbors. Sensors may receive multiple messages like this from several of their neighbors. The path sensors pick the message from a sender whose message is the lowest path distance to the exit. When a path becomes broken, the node on the path stops sending. When a path is not updated, the knowledge about the path held by neighboring nodes “times out.” Those nodes may then switch to a new path based on data they receive from another neighbor, perhaps data that was previously discarded because the path was longer than the original one.

By contrast, in the active protocol, based on the Ad Hoc On-Demand Distance Vector (AODV) protocol [63], sensors send out requests for path information from their surrounding sensors. Surrounding nodes either send back replies to the original senders with path information or forward on the request in the hopes of finding path information from another sensor or the resource itself. Messages containing path information are intended for a specific recipient and are forwarded on until they reach that recipient, though intermediate sensors forwarding them may update their own
Table 3.1: Code Complexity of Protocols Implemented in nesC

<table>
<thead>
<tr>
<th>Code Module</th>
<th>Lines of Code</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Passive Messaging</td>
<td>Active Messaging</td>
<td></td>
</tr>
<tr>
<td>Message Reception/Route Maintenance</td>
<td>379</td>
<td>547</td>
<td></td>
</tr>
<tr>
<td>Message Sending</td>
<td>133</td>
<td>282</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>512</td>
<td>829</td>
<td></td>
</tr>
</tbody>
</table>

path information. When a path becomes blocked, a sensor sends out an invalidation message which is forwarded to nodes using that path. Finally, because path updates are made solely through explicit invalidation and explicit requests for new paths, after the path data propagates throughout the network, no more messages are needed to maintain the paths.

To direct an actual user to the exit in this scenario, use of a simple display or visual cues [79] can be used. In simulation, I use the node’s LEDs to indicate the direction of the next step to the exit sensor. This behavior is executed with the DirectActuator function described above. In this case, DirectActuator specifies that a different LED (or combination of LEDs) is to be turned on depending on the direct the sensor instructs the user to travel.

### 3.5.2 Evaluation of Implementations and Performance

I now evaluate the success in implementing these protocols and then evaluate their performance under sample conditions.

As described in the previous section, the active protocol requires more maintenance of routing tables and handles a greater variety of messages than the passive protocol. As shown in Table 3.1, the added complexity of implementing the active protocol is significant. It required more than 60% additional lines of code to implement the modules responsible for processing incoming messages and managing outgoing messages in the active pathfinding protocol.
Figure 3-11: A comparison of the active and passive protocols in the discovery of paths to the exit and recovery from blockage in a simple ring topology of different sizes.

Next, I examine whether this tradeoff in code complexity is worthwhile. Tests occurred in two stages. The first is the setup stage. Using simple ring-shaped maps of 8, 16, 24, and 32 nodes with the exit designated at the “top” of the ring, I first measured how many messages per node were required for the nodes to trace out the shortest paths to the exit. Next, I induced a blockage in the ring one step away from the node where the alternate path begins – the point where the ring switches from a counterclockwise path to the exit to a clockwise path to the exit. I then observed how long it took the network to recover and discover the alternate path. This process was automated using TOSSIM’s built-in support for scripting via Python. With it, I can determine when the system had found the correct paths at setup time and when the nodes had recovered from a blocked pathway.

As can be seen in figure 3-11, the number of messages per node required under the active protocol is much higher, owing to the increased amount of routing overhead required. By contrast, the simple nature of the passive protocol allows for fewer messages per node for setup, though the number steadily increases as the size increases. That figure will steadily increase with network size because each node with data about a path to the exit will send out its known data at regular intervals. The
more time required for setup to finish, the more messages each node will send, on average.

During the recovery period, also shown in figure 3-11, I see a similar number of messages per node required in the active protocol. Again, there is little overall rise in the number of messages sent per node, and there is some decline between 24 and 32 nodes under the active protocol – just as during the setup time. In comparison, there is a small but steady and perceptible rise in the passive protocol which mirrors the rise during the setup stage.

There is a larger fixed cost associated with using the active protocol for setup, and one does not see a corresponding advantage in terms of ability to recover from a blocked passageway.

This test case is an imperfect and limited one. A failure of the sensors is interpreted as a blocked passageway. The system cannot tolerate a failure of the sensor at the resource, which causes the entire system to fail. The system depends on the premise that the sensors will be aligned such that their communications patterns correspond with the physical geometry. The system depends on carefully calibrating the layout of the sensors so that one communications hop reaches a maximum of one other sensor in each direction – communication cannot reach over multiple sensors on a path. This simplified system is presented only so that basic ideas behind the use of sensors for pathfinding can be explored.

3.6 Adaptive Communication Models

3.6.1 New Thinking About Protocols

The previous example depended on very simple assumptions about the radio transmission model. In reality, radio transmission is a much more complex phenomenon which experiences fading, differences in signal strength based on present obstacles, and asymmetric channel behavior between two points. Furthermore, a connection via radio transmission between two sensors does not mean that there is a physical con-
nection between two locations of those sensors. Therefore, inferring topology based merely on network connectivity is not a valid idea in the physical world. To solve the problems of deploying sensor networks to interact with the physical world, one needs to think about protocols that do not depend on the RF communication model in order to infer physical topology.

Protocols for DiDo systems that do not depend on a specific communication model can more easily adapt to changes in the network topology. Adding more sensors and dealing with network failures can be accounted for. Furthermore, a network protocol in software can support the integration of adaptive RF technologies. I describe such a protocol that attempts to solve a pathfinding problem while remaining fault tolerant, adapts to additional sensors, trades information about the the physical topology, and effectively partitions the network into local clusters to aid in scalability.

### 3.6.2 Creating a Protocol

The goal of our model is to create a sensor network that is fault-tolerant, scalable, and easy to deploy. Early assumptions about multihop networks were made for simplification purposes, but I argue that these assumptions actually make the deployment of applications less useful and work against fault-tolerance instead of in favor of it. I discuss why these simplifications are invalid, present a model in which sensors have certain necessary knowledge and capabilities, and design a protocol that works within these parameters.

Earlier in this chapter, I simplified the simulation model of ad hoc networks by assuming that wireless communication occurred in a fixed-radius fashion. In fact, for a pathfinding application, this assumption is a very convenient model. If one aligns sensors along a set of pathways such that they are all one communication-hop away from each other, the sensors can infer the geometry of the network very simply, assuming that each node is aware of its location in x,y space. Unfortunately, this model is neither real nor effective. A node failure will imply a lack of connectivity. If, in a set of sensors along a physical path, one of the sensors fails, network communication will be broken, and the network will infer that no physical path exists. Furthermore,
even in simulation, a network that does this sort of geographical inference is difficult to “deploy,” as time is spent trying to ensure that all of the sensors line up correctly and their communication regions do not overlap.

Our solution is to provide knowledge of the geographical layout to each sensor and allow the sensor to infer its own position within that map. The sensor can then make decisions accordingly based on any additional data it receives from other sensors, which may or may not be present. In the event of a sensor failure, the original sensor can rely simply on the static data it has stored or on another local sensor that can supply similar information. I assume that sensors are able to sense conditions such as crowding in a corridor or a blocked passageway and can adjust their calculations about distance and routes based on this information.

Based on empirical experiences with mote deployments in buildings [3], motes not physically connected to each other are still likely to have network connectivity. In fact, the network topology of that mote deployment shows a highly connected graph. The communication network is a much more highly connected network than the network of the physical floorplan. I thus assume that in the event that a passageway is physically blocked, network communication between two sensors connected by that blocked passageway will still remain active.

While each sensor has knowledge of the entire static map, as conditions change, each sensor is only aware of changes in its immediate area and those of its neighbors. When it comes to maintaining information about dynamic changes in the network with respect to paths to resources, no sensor knows the entire path, only the distance from there to the exit and which link will lead that way. Knowledge of the best path is distributed. The more sensors that exist, the finer-grain and more distributed the dynamic path-data becomes.

### 3.6.3 Routing Protocol

I again use a reactive protocol based on the Destination-sequenced Distance Vector (DSDV) [64] protocol. This protocol was chosen for its simplicity. Also, the specifics of this DiDo system, as applied to a single-resource/exit building evacuation scenario,
mean that DSDV is a more efficient method of communication and coordination. However, this is not a pure DSDV protocol, as DSDV concerns itself primarily with network routing. I am concerned with geographical routing, so what I have created is a modified protocol based on DSDV that determines communication routes based on the physical layout of the sensors, in this case a Geographical-DSDV.

The alternative is a proactive protocol, such as AODV [63], as described earlier. A proactive protocol may have advantages in other, more general, resource-discovery scenarios such as Directed Diffusion [44]. Directed Diffusion supposes a plurality of resource nodes and a plurality of nodes with interests in resources within the network. Some nodes may have interests in one type of resource. Some nodes may have interests in another type of resource. Finally, some nodes may have no interests. In this case, a proactive protocol may be warranted, as the nodes in search of specific resources should explicitly make their interests known, and resource nodes should only respond if they receive explicit requests for their resources. This model would likely result in better use of bandwidth than if all resources flooded the network to announce their presence to any interest nodes that may be available.

By contrast, in the DiDo model presented here, there is one resource – the exit – all sensors have the same interest, and the resource knows that it is needed. Thus, it makes sense to have the sensor at the resource announce its presence and have the other sensors forward on their routing knowledge to their neighbors, rather than having all of the other sensors initiate search requests for the resource. Given the structure of the system as presented, DSDV makes sense, as the requests of the sensors for the resources are implicit. One may deploy DiDo systems where a proactive protocol is the more appropriate choice, but in this case, DSDV is the better option.
3.7 How the Sensors Cooperate

3.8 Designing the Protocol

Even without any network communication at all, a sensor can determine the best path to the exit by sensing the state of paths to adjacent nodes and then examining the map it holds to provide path information to a user. In doing so, it calculates the distance to the exit, or resource, \(d_r\).

Let us start with the case in which nothing about the conditions in the map changes while network communication occurs. Each sensor in the network periodically sends out its location and distance from the exit, based on its knowledge of the map. Receiving sensors maintain three things: a calculation of the distance and path to the resource, without any other information from the network; the “next step” to the next closest next sensor on the way to the resource, \(s_n\); and the distance to all other parts of the map, based on its stored map data and the sensor information it receives. A sensor receives messages about location and distance from other nearby nodes. With its knowledge of the map, the receiving sensor can determine which sending sensor is closest to the receiver and has the best path to the exit. In this case, a sensor receives a location from the sender \(l_s\) and the sender’s distance to the resource, \(d_{sr}\). The receiver looks up its own calculated distance \(d_r\) and compares it to the sum of \(d'_r\) and the calculated distance to the sender, based on the receiver’s map data, \(d_s\). If \(d_{sr} + d_s \leq d_r\) and \(d_s \leq d_n\), the distance to the “next sensor,” \(s_n\) that it maintains, then the receiver sets \(d_r = d_{sr} + d_s\) and sets the “next sensor,” \(s_n\) to be the the ID and location of the sender. Users will be directed to that next node, where they will receive the next step in their path to the resource. That “next sensor” is always the closest sensor on the best path to the resource. Messages from sensors on the best path to the resource, but far away from the receiver, will be discarded in favor of sensors on that same path that are closer to the receiving sensor.

This protocol satisfies the intended goals of our system. First, it is fault-tolerant. In the event that the “next sensor,” \(s_n\), on a path to the resource fails, the sensor
that had previously selected that one as its “next” will automatically detect the next closest sensor on the path, $s'_n$. The sensor will only remain aware of the physical state of the regions covered by the active sensors, but it will still mark a valid path.

3.8.1 Simple Rules for Choosing Paths

The protocol is expressed as a set of simple rules, broken into three different scenarios. In a simple case, dealing with only sensors at adjacent nodes, messages have three fields, distance, senderId, and originId.

**Scenario One: Sensor Does Not Know of a Path**

- Accept a message indicating the distance from a sensorId on a path to the resource which does not have an originId equal to the ID of the receiver
  
  If the message is accepted:
  
  - Forward on the known distance with senderId equal to the ID of the receiver and originId equal to the ID of the senderId of the message

- If the message is not accepted:
  
  - do nothing

**Scenario Two: Sensor Knows of a Path**

- Accept a message on a path to the resource if the distance that is less than or equal to the known distance which does not have an originId equal to the ID of the receiver

  If the message is accepted:

  - Forward on the known distance with senderId equal to the ID of the receiver and originId equal to the ID of the senderId of the message

- If the message is not accepted:
  
  - Forward on the already-known distance already saved

  - If the known distance is not renewed or replaced by a message after a given interval, delete known distance

**Scenario Three: A Passageway Becomes Blocked**

- If a passageway becomes blocked, then the sensor will no longer accept messages from the sensor on the other side of the passageway.

  If the sensor’s known distance and destination are connected to the sensor on the other side of that passageway, this knowledge will be discarded
- The protocol continues from the state in which the sensor does not know of a path

Keep in mind that for each sensor, there are three types of available passageways, represented as edges in the graph: a passageway that the sensor directs pedestrians or other agents in the system towards, passageways that other sensors direct pedestrians or other agents into the node where the sensor in question resides, and passageways that are unused. These unused passageways are represented by edges in the graph that are not part of the minimum spanning tree rooted at the resource. When a passageway is blocked, the sensors will realign their connections with other sensors in search of an unused passageway connected to another path to the resource. In graph terms, a node with an unused edge connected to a node on another path will remove the edge associated with the old path, and take the unused edge to connect to the node on the other path. The system will face a reconfiguration of all nodes between the blocked passageway and an available unused passageway, backing up our earlier assertion that reconfiguration time will be a function of graph degree, not the number of nodes in the graph (Figure 3-12).

### 3.8.2 Working with Different Sensor Densities and Network Models

The protocol operates independent of the radio model used. As long as a sensor can communicate with its neighbors, the protocol will function correctly. In the event of overlapping communications from multiple sensors on the same path, the receiver will pick the closest valid one on the path, because of its already pre-existing knowledge of the map. This is contrasted with attempts to infer map layout from hop-count, which can depend on the assumption of fixed-radius communication and that signals will not pass through walls.

Finally, the protocol allows for improvement of performance with a greater number of sensors. For example, take the case of a map represented as a graph of nodes and edges, where the edges represent passageways and the nodes represent rooms. Assume
Figure 3-12: **Top:** When a graph is not very dense and an edge is blocked, messages need to travel distantly to find an alternate path. **Bottom:** In a very dense graph, a blocked path can be corrected much more quickly because an alternate path is more easily available.
that sensors are placed on \(\frac{1}{4}\)th of the map nodes. Sensors would direct users to map nodes an average of 4 steps away, and the network connection graph created by the sensors would cover only \(\frac{1}{4}\)th the number of edges in the actual map. Furthermore, given the assumption that sensors can detect edge integrity in the edges adjacent to them, one is faced with the prospect of half of the edges being unscanned. If one doubles the density of sensors to cover \(\frac{1}{2}\) the nodes on the map instead of \(\frac{1}{4}\)th, then almost all of the map edges will be covered, and the network connection graph formed by the sensors will have twice as many nodes and form a clearer map connection path to the exit.

### 3.8.3 Sample Protocol Walkthrough

The intent of this protocol’s design is to support automatic recognition of new sensors as the density increases. At first, the protocol should give some course-grain knowledge of the physical region with two sensors and then provide finer-gain knowledge as more sensors are added. In addition, communication between sensors will be as local as possible. Ideally, the only other neighbors which with a sensor will communicate will be its predecessors and successors on a path to the exit.

The sensor model is shown in figure 3-13. I propose a high-complexity communications/computation platform with several different sensing elements. This platform is capable of processing incoming data from its sensors and transmitting the results over RF to its neighbors as well as changing a display which directs pedestrians to the best passageway. As mentioned in chapter 2, because of the multifunction nature of these individual units, it is most descriptive to refer to them as transducers. Each DiDo transducer resides in a room and has sensing elements that monitor the state of the room’s adjacent passageways. Any illustration of this process will show two sets of connections. The first will be the network connections that a sensor has with neighboring sensors and the connections that represent the pathway to an exit. In the graph representation in figure 3-14, the nodes correspond to the rooms and the edges correspond to the passageways. In the case of a map fully covered with transducers, the network connections between the transducers will look the same as a shortest
Figure 3-13: The sensor model. Transducers are in the center of the room with sensing elements, marked by an “X”, monitoring the state of the passageways of the room’s exits.
path spanning tree and be identical to the path to the exits. At the same time, a transducer is only responsible for knowing just enough path information to direct an actuator to the next transducer. It is not responsible for maintaining knowledge about all the paths or which specific nodes will take it from the map node where the transducer is located to the exit.

Take the base case of two transducers in a map of arbitrary size, with one at the exit (Figure 3-14). Assume that the exit transducers and the top transducers (which I will call the source transducers) can communicate with each other, regardless of their relative placement. Each transducer understands its relative place in the map and detects the state of the pathways from the node on the map on which it is placed. The exit transducer sends a message to the top transducer indicating its location at the exit and the fact that both of its pathways are clear. The top transducer realizes that its own pathways are clear and picks the best path to the exit node based on its knowledge of the map layout, even though the state of the intermediate pathways is unknown. Using the best knowledge it has, the top transducer picks the best path to the next source of information – in this case, the transducer at the exit.

As more transducers are added, the transducers will partition themselves into separate communication regions. In figure 3-15, when an intermediate transducer is added at a map node on the path between the exit and the source transducer, the top
transducer will ignore any messages from the transducer at the exit in favor of the intermediate transducer. At the same time, a transducer will also ignore any other messages from transducers that are more distant from the exit than it is. Whereas the top transducer previously communicated with the transducer at the exit, now it will only communicate with the intermediate transducer, and the top transducer will trace out a path to that intermediate transducer, rather than to the transducer at the exit. The intermediate transducer will take responsibility for receiving information conditions at the exit transducer and trace out a path between it and the exit. As more transducers are added, communication becomes more local and responsibility for handling the overall solution becomes more diffused.

This implicit partitioning of communication between the transducers allows decisions to be made locally. In fact, decision-making about which path to take can be done only by the transducers for which it is necessary. All transducers only forward on the distance they know to the exit, based on sensed data about their immediate area as well as known data about the map layout. In the event that changes in the map layout occur, transducers will update their knowledge about estimated distance to the exit and forward on that new knowledge. Under DSDV, old data expires if not consistently renewed, so the new data sent by the transducers will replace the original data kept by the receivers. In the event that an intermediate transducer on a
Figure 3-16: **Left:** The network communication before a blockage is detected. **Right:** The network communication after a blockage is detected.

path changes its estimated distance to the exit, any receiving transducers may reevaluate their connection to this intermediate transducer. A receiver may choose another transducer that indicates a closer path. Alternately, the receiving transducers may do nothing at all, but the intermediate transducer may choose a different sensor between it and the exit to communicate with and trace a path to.

An example of this process of reconfiguration is in Figure 3-16. When the sensor of an intermediate transducer at node A detects that a path is blocked, the transducer will ignore messages from the transducer at at node B in favor of messages from the transducer at node C. This is because the transducer at location C is the closest transducer on the new best path to the exit. The transducer at location A communicates with a nearer neighbor instead of the more distant transducer at the exit. This will also change the physical paths that the transducer at location A maintains (Figure 3-17). In the event that there is no transducer at location C, then the transducer at location A would still trace out the same physical path to the exit that went through location C without the detailed knowledge of the state of the passageways. Note that the source transducer (at the top) will not make any changes, since the transducer at node A is still the closest transducer on the path. However, because the path to the exit has gone from 6 steps to 7 steps from the top transducer, the top transducer would be equally likely to choose an intermediate transducer at node D which is also on a path 7 steps away from the exit node and an equal number of steps away from
Figure 3-17: **Left:** The physical paths maintained by the transducers before a blockage is detected. **Right:** The physical paths maintained after a blockage is detected.

the top transducer as the transducer at node A.

If I can show that these path reconfigurations will always be handled by local communication, then I can argue that network communication is scalable as a map layout becomes larger. Proving the argument will require a set of tests under changing conditions to see if reconfiguration is scalable.

### 3.8.4 Physical Blockage vs. Network Blockage

The above protocol is concerned with how to transmit information about a physical passageway that becomes blocked and how sensors can direct pedestrians around such a blocked passageway. As I discussed above, the assumption in the case of a blocked passageway was that sensors connected by that passageway would still be able to communicate with each other. In fact, transducers that were never physically connected are assumed to be likely to be able to communicate with each other, which is why embedding the layout of the floorplan into the sensors was integrated into the system. However, such a scenario forces us to consider the reverse situation—what if a passageway to the resource node was available, but network communication between two active transducers at each end of the passageway was blocked? Here one would observe a passageway that is physically open but network communication is impossible. I can resolve this problem by allowing sensors to make active requests for data in the event that they cannot communicate with a sensor on an adjacent
passageway.

The protocol works as follows: a transducer receives messages from other nearby sensors and chooses the nearest transducer on the shortest path to the resource node, as before. However, if the transducer does not receive a message from the transducer at a neighboring node with a shorter path to the resource node, then the transducer makes an explicit request for data from this node. When other transducers receive this request, they place that request in a queue and wait to see if they receive any messages from the requested transducer. If they do not receive any messages from the requested transducer, they forward on the request, and the process repeats. When a transducer receives a message from a requested transducer that it has in its queue, instead of ignoring that transducer, as it would for messages from any other transducers not on the best path, the receiving sensor forwards on the message from that requested transducer, essentially acting as a repeater for the otherwise out-of-range requested transducer. This ensures that previous transducers with queued requests will receive this repeated message, eventually reaching the original requester, which then can update its own data. This feature can be integrated into a DiDo system if the risk of facing a blocked network connection between adjacent transducers is considered severe enough to warrant it.

3.8.5 Implementation

This protocol was implemented on the TinyOS simulator in conjunction with its visualization tool, TinyViz, which allowed one to observe graphically the evolution of the protocol. With the TinyOS simulator, I took my DiDo transducer model described above and exported that model to the Berkeley motes platform. Using this platform presented many challenges unique to the architecture of this simulator as well as allowing me to think about the design at a higher level than simple embedded implementations of sensor systems.

The nesC programming language for TinyOS allows programmers to build applications for the Berkeley motes sensor platform by connecting together separate individual components. nesC allows me to separate responsibility for receiving data
and sending data into different components. Since all motes in a TOSSIM simulator must contain the same program, I feed the motes specific sensing data to indicate to them whether they are an exit transducer or a path transducer. In addition, the ability to send in data readings to the sensors with the simulator allows one to simulate the presence and absence of available pathways. These changes can be made using a Python-based scripting language called Tython that integrates with TOSSIM in order to change the data readings received by the sensing elements on the motes which then react to those changes in data readings in their code.

Knowledge of the initial map layout was hardcoded into a separate module of the mote’s code. On initialization, each mote calls that module to load in the hardcoded layout. This separate module was important because later I will explore the use of automatically generated code to aid in this. This separate module is called by the mote upon initialization with the assumption that the module will be replaced for different map layouts. The map layout code is intended to be transparent and separable from the rest of the code, which will remain unchanged, regardless of the map layout. Because map layouts are of a maximum of 20 nodes, I used a simple two-dimensional connection matrix to represent the layout. More efficient data structures should be used as the layout becomes larger. The idea behind hardcoding the map layout is that this will represent static information that the mote has available and that it will update that static data with new information it receives via its own sensing elements as well as data it receives from other motes.

To aid in visualization, I added a plugin to the TinyViz simulator which displays the map of nodes and connections in the simulation area. As the simulation runs, a directed graph is traced out when the motes choose which neighbors from which to receive data. When every node in the map is covered with motes, the directed graph will form a shortest-path spanning tree that lies on top of the map layout. However, under other circumstances, the user will be able to view the changing granularity of the network connections as motes are turned on and off.
3.9 Simulating New Protocols for DiDo

Evaluation of DiDo systems is more complicated than simply evaluating the effectiveness of the network protocols. Evaluation of ad hoc network protocols is generally rather straightforward. A source and a sink can be designated, and the number of packets sent from the source can be checked against the number of packets received at the sink. Meanwhile, experimenters can evaluate how changes in network density and mobility affect network throughput.

By contrast, evaluating the efficacy of a DiDo system forces the implementer to think about the changes that will occur in the physical world and how the network will react to these changes. Also, because I wish to evaluate the DiDo protocols under a large number of different conditions, simulations need to be built and evaluated quickly. Furthermore, success conditions need to be set, which are going to be different for every physical scenario. Finally, the TinyOS simulator is not designed to handle the issue of physical conditions within the network.

Initial testing using very simple communication models and simple topologies was done quickly but provides little insight with respect to the ability to implement these new systems. Gaining more insight requires one to evaluate more complicated scenarios and demands that I create a means of testing these new scenarios quickly.
Chapter 4

Building Simulation Tools

4.1 Introduction

To solve the problem of evaluation, new tools are required. One issue is that manually encoding the physical layouts in the sensors is difficult and time consuming. Evaluating the performance of this system means that it will have to be tested on several different layouts and scenarios. This requires a means of creating scenarios very quickly and programming the simulator to terminate when the sensors have solved the shortest path and recovery problems. Furthermore, since the model was designed to support the quick implementation of different DiDo problems, such a simulation-building application should be able to create those scenarios very quickly and generate the code necessary for implementation.

Here, I present a tool that helps simulate and evaluate the proposed system. The goals are both to quickly design and deploy new scenarios and evaluate the effectiveness of the chosen protocols.

This tool is called the Map Generator, and it allows designers to create custom scenarios in which the sensors can be deployed in simulation. The tool determines the solution to the initial problem that the sensors need to solve so that the program can generate scripts that tell the simulation to terminate when those conditions are met. Users can also create new problems, such as blocked passageways, for the sensors to recover from. The performance of the network when it comes to accounting for those
changes can be evaluated there as well.

The tool automatically generates necessary knowledge of the physical topology to be programmed into the sensors. Furthermore, the tool automatically generates scripts for the TinyOS simulator (TOSSIM) in order to set success conditions for the simulation (Figure 4-1). In addition, one can create changes in the physical environment and evaluate the ability of the system to recover from those changes. Using this knowledge about performance, I integrate it into a simulation of the physical world to see how well these systems affect the physical world.

The next important tool in this system is a random graph generator that integrates with the Map Generator. This random graph generator allows one to automate the process of evaluating network protocols. In an automated fashion, one can generate a graph layout, pick an edge to block, and then evaluate the time it takes for the system to set itself up and recover from a failure. While a user can design and evaluate a single scenario manually, the system also allows a protocol to be evaluated on the basis of its performance over many random scenarios in series, which is a new application for running and evaluating TOSSIM-based applications.

Other programming environments for sensor networks focus on creating a means of making sensor network development easier. The TinyOS development language, nesC [31], sought to support ease the development process by allowing developers to create interchangeable modules that can be connected together in different ways for different applications. On a higher level, others have moved to create better middleware support to allow programmers to use advanced primitives in order to express more advanced concepts.[85], [10] Other projects, such as Viptos, explicitly expand on both of these ideas to create a graphical development environment.[16] Viptos uses the nesC programming model as a jumping-off point and allows users to graphically arrange and connect those interchangeable nesC modules when building TinyOS applications. The Map Generator tool is reminiscent of Viptos in that is supports methods of reconfiguration and programming through a visual interface.

The model of scenario design is about creating problems that are to be solved by the sensor network and evaluating the performance when solving those problems in...
terms of messages per sensor. Furthermore, I evaluate the ability of a sensor network to solve a specific problem, rather than evaluating its performance as it implements a protocol over a given time. With the appropriate tools, one can evaluate the efficacy of the solutions quickly and under a variety of circumstances.

4.2 Designing Scenarios with the Map Generator Tool

The first part of the Map Generator is allows users to design scenarios akin to floor layouts and specify what changes in the physical world will occur. The Map Generator provides an interface to support that.

These tools can also integrate themselves with the TinyOS simulator to make other changes in the physical environment. In short, this tool allows the users to design physical scenarios, direct the simulation scenario, and, finally, evaluate the results. Those results can then be applied to systems such as pedestrian simulators to evaluate the efficacy of the system.

The Map Generator allows users to create custom layouts and designate resources.
Figure 4-2: The Map Generator Application GUI
A user adds nodes representing locations in the physical environment, designates the location of resources, and sets “success” conditions with respect to the problem being solved. The Map Generator provides a simple visual interface in order to allow the user to make these specifications.

First, a user creates a layout by marking the placement of physical nodes in the map area and specifying their connections to one another. A floorplan translates to a node-and-edge layout in a fairly straightforward manner, as seen in Figure 4-3. Next, the user switches modes from building a map to selecting resources. Finally, the user sets success conditions to specify how the sensor network should react to the layout with respect to the location of the resources. For example, one success condition specifies that the sensors are to trace out the shortest path to the resource. Another success condition specifies that the actuators are to direct users away from

Figure 4-3: Left: A small floorplan. Right: That floorplan as represented using the map generator.
the resource. Others indicate that the sensors should indicate their distance or absolute direction from the resource. Setting these conditions becomes relevant both for measuring performance by understanding how long it takes the simulation to reach these states, and for code generation to implement the specified output of the sensors. Code generation will be discussed below.

Encoding these physical scenarios as snippets of code or numerical representations in data structures is much more time-consuming than simply visually placing the conditions on a desktop. Thus far, the application allows users to specify nodes, paths, and resources, corresponding to the DiDo model.

4.3 Specifying Environment Changes

At times the physical world may change, and I seek to understand how quickly the system can reconfigure and recover in response to these events. To do so, users of the Map Generator must be able to specify physical changes that occur. The application must recalculate the new “correct” conditions and pass this information on to new scripts in the TinyOS simulator as success conditions which will cause the simulation to terminate and record its result. The ability to set these conditions, recalculate, and generate scripts for the simulator automatically is very powerful because the simulations will now no longer merely measure the results of a given protocol in a given scenario (such as network bandwidth or pedestrian throughput). More importantly, the simulations measure the time it takes to recover from blocked passageways and determine how much traffic is required under various circumstances. Since the goal is to confirm my speculation that problems such as error recovery are local and that bandwidth per node will not decline as graph size increases, I explore how the network performs when subjected to changes and stresses in the physical world. Using this knowledge, I can integrate the system with pedestrian or other flow simulations.

When a user finishes designing his initial layout of the system and location of resources, he “freezes” the layout, preserving its state and preventing any more modification with respect to the number, location, and connections of the nodes. However,
a user can then specify changes to the environment, such as whether to block a passageway, or changing the location of a resource, requiring a recalculation of the layout.

To designate the occurrence of a blocked passageway, a user enters into the “block passage” mode and selects two connected nodes. The connection between the two nodes is then marked as “blocked” (Figure 4-4). When the Map Generator outputs scripts, one script will change the sensor readings to indicate that one of the passages here is blocked.

Users may also specify which sensors are on and which are off when the simulation begins and may even change the location and availability of the resources. Normally, this will take the form of changing the sensor readings by the motes in simulation, changing their behavior in the network.

Furthermore, the user may choose to specify different re-routing conditions when changes occur. For example, let us say a user wishes to indicate that a hazardblocked passageway occurs at a certain point in the model he is designing but that pedestrians should be routed a radius of 2 steps away from that hazard. This is an added feature which will change the protocol itself. In short, users are able to make modifications to the communication protocol by designing new scenarios in a physical world. Users can specify the behavior of a sensor system when describing the physical world in which
it is deployed and then immediately test their system in simulation, without the need for rewriting the underlying sensor code. Instead, that code can be regenerated.

The simplification of the code into a set of specific primitives makes quick reconfiguration and respecification possible. Users can specify different reactions to different phenomena based on distance from the resource, direction to the resource (regardless of distance), and reactions to hazards or blockages. Users can then experiment with new reactions and new applications and evaluate performance. Those primitives can be rearranged to form different applications as well, though the design favors a situation in which rearrangement of primitives is implicit in the design of the new scenario, rather than explicit via programming.

A user then exports the simulation scripts that check for this success condition. This allows the user to measure performance under that changing condition by measuring the number of messages that are sent and received before the success conditions are met.

By specifying changes to the environment using a new application, users can change their physical models more quickly and easily before testing them out in simulation. By integrating the sensor network simulator with these “scenario building tools,” users can gain more flexibility in designing and testing sensor network applications. The system automates the process of setting success conditions for the simulation and allows the user to reprogram and respecify aspects of the network protocol and algorithms used by the sensors to make the “Distributed-out” decisions. The application does this by translating specification instructions by users into code.

### 4.4 Code Generation

The physical layout of the “world” created by the users needs to be encoded in the sensors themselves. That layout will change for each new system designed by a user. Also, the user may wish, as described above, to customize the behavior of the system and the protocol depending on the specific scenario. Fortunately, the code used to program the sensors is simple and designed around a few basic organizing primitives
which can be modified and rearranged to implement different applications. Finally, the ability to automate tests, confirm the correctness of the communication protocols, and evaluate their performance depends on scripts that are custom-tailored for the specific physical scenario that has been designed. Each of these components – the physical layout, the behavior of the sensors themselves, and the scripts used in testing and evaluation of the simulator – require automatic generation of code based on the user’s specifications and desires while using the Map Generator.

Using a code-generation scheme for the physical layout is straightforward in my model. The sensors load in their layout information through a separate module. The code generator outputs a new module, based on a template, that contains the map layout in the form of an adjacency matrix. In the Map Generator, the map layouts are represented as an undirected graph using the OpenJGraph [46] library for Java. The Map Generator then translates the graph designed by the user into an adjacency matrix, represented as a 2-dimensional array in nesC using an existing template. The mote loads in the new layout data when it calls the layout module, which can be replaced without changing any of the other existing code that controls the rest of the mote. The module does not require any preexisting knowledge of the names of the variables used by the mote’s controlling code outside of the map layout module. The graphical layout described by the user with the Map Generator is translated into nesC code used by the mote.

Next, code-generation is used to design the scripts for the testing and evaluation of the system’s performance. In part, the result of these scripts will depend on the success conditions chosen by the user. For example, if the user designs a scenario where the success condition is to determine the shortest paths to the resource and specifies that the layout will be fully covered with sensors, then the Map Generator will output a script specifying a termination condition when a shortest-path spanning tree is formed by the sensors. The Map Generator calculates the shortest path spanning tree and then generates a script that accesses the state of all of the sensors to check if their “next step” data is in line with the correct solution that forms this shortest-path spanning tree. Other solutions, based on the specified behavior of the
“Distributed-out” functions or differing behavior that would result in a different connection graph (such as, say, creating paths away from the resource), would result in different scripts generated to check for the success of the protocol.

First, “Distributed-out” functions may be specified by the user. This is the DirectActuator function described in the set of primitives mentioned above. The user can specify a set of “Distributed-out” reactions, such as LEDs that point to the resource, away from the resource, or represent the distance from the resource. Next, as mentioned above, modifications can be made to the protocol itself to specify communication relationships between sensors, such as specifying that sensors remain a radius of two or more steps away from an obstruction.

Furthermore, in keeping with the nesC model, one can model many of the primitives mentioned in Chapter 3 as separate TinyOS modules. In my example, the modules which handle the types of messages accepted and the types of messages which are sent can be interchanged using the Map Generator by switching out modules. I have designed the system so that DirectActuator, AnnounceDistanceToResource, ReceiveDistanceToResource are separate modules which can be switched in and out or modified by the Map Generator depending on the protocol used or type of actuation methods.

Changing the “Distributed-out” will change the visualization results in simulation and, if integrated with a simulation of the physical world or deployed in the physical world, will allow the users to measure the effects of the actuation. For example, the code can be modified to change the distance from a blocked passageway that sensors will route around. The underlying code itself is based on the outlined set of primitives, and the communications protocols are modified by the the code generator.

These three code-generation capabilities of the Map Generator all aid in the implementation and customization of user simulations. Because the model demands that each sensor possess knowledge of the layout, the Map Generator programs in that knowledge automatically without forcing the user to do the translation from graphical layout to nesC code. Furthermore, because any change in physical layout changes the ultimate solution to the problem, confirming the system in simulation requires
one to automatically generate scripts, because custom-writing a script for every new scenario is very time-consuming. The process is automated based on the scenario designed by the user. Finally, because a user may wish to evaluate completely different applications, I provide an interface that allows the user to modify the code of the protocol and Distributed-output itself.

4.5 Other Control Output

In addition to generated code, the Map Generator also generates configuration files used by TOSSIM and the visualization plugin that displays the physical layout. These configuration files are responsible for the layout of the motes, specification of initialization data before the simulation starts, and the visual display of the physical nodes.

These files are “housekeeping” for TOSSIM and necessary in order to make the sensor layout match the layout of the physical world that I have created with the Map Generator. These files carry a common naming convention for those running tests after they have created the scenario using the Map Generator. The templates can also be customized to incorporate different radio models, for example.

4.6 Automating the Map Generation Process

A user can design his own layouts and specify which scripts he wishes to output using the Map Generator, but this process can also be automated. I provide an automatic layout generator and automate the process of picking an edge to block, generating the scripts, and compiling the system for TOSSIM.

The automating process is rather straightforward. Instead of relying on an independent user to manually design the layout, one can construct a set of connections randomly. This will not take the form of a traditional random graph [24] because a realistic graph layout limits one to rooms that are directly connected only by reasonable proximity. Therefore, the random graph generator works by creating an additional node at a location near at least one other node and connects itself to any of the nearby
Figure 4-5: The architecture of the automated map generation system.

nodes (and only those nearby nodes). The resource node is chosen randomly from any one of those nodes, giving a set of different graphs with different connections and travel routes that will vary widely, given that the resource node will be in a different location.

Therefore, I script the process of automatically generating geographical layouts and feed that layout into the Map Generator, which then picks a random edge to be blocked and generates the necessary scripts and mote code, and is finally compiled and run by TOSSIM. (Figure 4-5)

4.7 Integration with TOSSIM

I have described a system that allows one to generate code and scripts that can be used for the TinyOS Simulator (TOSSIM). However, to evaluate the efficacy of the system, I must construct a framework that allows one to measure the performance and measure it automatically. This requires creative use of the TOSSIM scripting features.

Take the example of evaluating the speed at which the protocol can understand the topology of the network and recover from failures. In the case of finding shortest paths to the resource node, the system forms a shortest-path spanning tree for the physical topology of the sensors. Therefore, a simulation must terminate when this spanning tree has been calculated. After a failure, the simulation should terminate
again when the sensors organize themselves into the new shortest-path spanning tree. That means that an implementer would need to design a physical layout, program that topology into the sensors, determine the correct solution for the sensors to calculate, direct the simulator to terminate when it reaches the “correct” solution, and run the simulation. This is useful when testing a few simple topologies (such as the “ring” topology in the early tests of pathfinding performance), but to evaluate the system on a large variety of topologies, the problem becomes much more difficult.

The main support for batch processing of multiple simulations in TOSSIM is via the “autorun” files. The autorun file allows the user to specify which code to run on the motes, how many motes to use in the simulation, how long the simulation should run, and how many times this simulation should be repeated. These specifications are helpful for determining data such as how many messages are sent within a given time interval, how many are received, how many are lost, and other metrics that allow a programmer to determine how well a given network protocol works at tasks generally considered important in early ad hoc network research.[66]

By contrast, my main concern is determining how long it takes the sensor network to reach a certain state, modifying the sensor readings at specific times, and seeing how quickly the sensors can reconfigure based on those new sensor readings. Accomplishing these tasks requires the use of Python scripting, which is supported by TOSSIM via its “Tython” scripting language. After exposing some of the Java classes defined in TOSSIM as public, I created Tython scripts that automatically run the functions of the “autorun” files and allow the simulator to set up the appropriate scenarios (Figure 4-6). In addition, the scripts can directly manipulate parts of the simulator that would normally only be accessible to a user via manipulation of the GUI. Since the goal is to take as much data as possible as quickly as possible, I designed scripts and modified the structure of TOSSIM to allow these previously user-specified functions to be handled automatically. I now have an integrated system with which one can generate scenarios, manipulate the simulator to implement those scenarios and measure how quickly the sensor network can reach a desired state. Furthermore, one can run this repetitively and take data automatically.
Figure 4-6: A flowchart of the Tython scripts used to run the simulation

4.8 Evaluation and Results

Using the Map Generator and the integrated scripts running on TOSSIM, I measure some significant metrics related to network performance. The first goal is to see how quickly a network can reconfigure its pathways after a passageway becomes physically blocked. I examine how this performance is affected by graph size and network connectivity. I then present the performance of other protocol add-ons.

I examine how DiDo networks perform and behave. Furthermore, I wish to ensure that these systems are scalable and efficient. What does this mean? First, I seek to ensure that communication is local. No matter how large the network gets, the motes should communicate with a limited set of other nearby motes. Reconfiguration should require a constant number of messages to perform with respect to the number of messages per node.

Measuring the number of messages per node sent and received is an important metric because it touches on significant issues facing sensor network deployment. The first is power. According to [17], network communication is the dominating factor in power consumption, rather than power consumed by the CPU. In fact, both transmission and reception are the dominating power consumers, compared to CPU processing, in a study done of mote performance (Figure 4-7).[4]

Furthermore, I also stated in Chapter 3 that a system designed to recover from
failure should be scalable by ensuring that the recovery process occurs with only local communication. One argument against this design is that in an emergency situation, one may consider it more important to use whatever means necessary to implement the recovery and not worry about the power consumption or bandwidth necessary to do so. I believe that this assumption is wrong-headed. First, in a truly DiDo system, one faces not merely the possibility of a single changing condition, but the prospect of multiple changing conditions in multiple places. It is thus necessary to ensure that the process of recovery is based on local communication and that data exchanged during these incidents does not propagate, interfering with changes that may be occurring elsewhere in the network. When I mentioned in chapter 3 that the transducers need to find the next nearest “unused” physical passageway to the resource, I hypothesized that such a physical passageway would be available within a limited number of steps, requiring reconfiguration by and communication with a limited set of other transducers.

Because the system uses DSDV as its routing protocol, the number of messages sent per node is inextricably linked to the amount of time it takes for the system to recover from any changes. Thus, the higher the number of messages per node that occur in any instance is a reflection of the speed with which the system reconfigures and recovers. Variations on this protocol would divorce the direct relationship between the number of messages per node and the time elapsed, and if one wishes to
minimize the amount of time required for recovery, then this is a separate protocol avenue that could be pursued. However, for now, I seek to ensure that the number of messages required for any recovery remains constant.

Thus, by measuring the number of messages per node required to set up and readjust, I measure three important issues:

- Power consumption

- Speed

- Data Traffic

Since DSDV is a fixed-bandwidth protocol, the bandwidth required to recover from a failure is implicit in the time and number of messages required, and there is an indication of which kind of bandwidth would be required in an active protocol, such as AODV, if one chose to use that method. If one requires that changes be made faster, one has indication about how much bandwidth will be required, given that the time and number of messages required to make the adjustments necessary is already known. Therefore, these metrics provide all the data necessary for a network designer building a DiDo network – speed, power consumption, and bandwidth consumed. Even if the designer chooses to use different network protocols or modify them to account for different specifications, the performance can be extrapolated from these metrics derived from the simpler protocol.

After the system is done setting itself up for its initial state, the Tython script created by the Map Generator pauses the simulation and changes the sensor readings that describe the environment “seen” by the sensors. The simulation then starts up again and waits for the system to reconfigure itself to the new pre-calculated recovery state, which has been already determined by the Map Generator. Sometimes, however, the blocked passageway, which is chosen at random among the passageways on the shortest path to the exit, will be the only passageway that exists on the path between a given node and the resource node. To get a better idea of how the network performs, I take these “failure” cases and separate them from the rest of
the data for later analysis, keeping the data related specifically to recovery. The results of the recovery experiments are apparent in Figure 4-8. It can be seen that repairs are fairly steady until the number of nodes increases past a given threshold. However, this threshold is not the number of nodes itself, but rather the network density. Under TOSSIM’s “empirical” network model, the motes in the network are highly connected, and adding more motes causes increasing network density, and the messages flood the network. The problem is with the network connectivity, not the physical size of the network itself. Before the network “hits the wall,” repair performance is steady even as the number of physical nodes in the floorplan doubles. The line at which this occurs is when the weighted network connectivity (the number of other motes that a single mote can expect to hear from if all motes send a message), is about 10.5. Since motes in TOSSIM are configured to receive at 20 Kbits/sec\(^1\), this indicates that each mote can receive up to about 1700 bits/sec from each other mote before flooding, which occurs when network connectivity is 10.5, since there are messages from 11.5 motes (including the sender) being sent simultaneously. The

\(^1\)In the TOSSIM file tinyos-1.x/tos/platform/pc/hpl.c, the procedure TOSHrfm_set_bit_rate(uint8_t level) is sent a parameter value of 0 when initialized, indicating a receive bandwidth of 20 Kbits/sec.
maximum bandwidth before flooding found under slotted ALOHA conditions was \( \frac{1}{e} \) of the maximum bandwidth\[73\], which would imply a maximum send bandwidth per mote of 700 bits/sec. The actual protocol used in the above experiment sent two messages per second of 600 bits each, for a total of 1200 bits/sec, which is 70% of the maximum. Note that this would not be a problem if one were to dynamically adjust the radio communication radius to avoid transmitting to distance motes, as is suggested in work I mentioned in Chapter 2.\[35\],\[69\] This result showing that the system can handle 70% of the maximum bandwidth before failing should not imply that the work in \[73\] is somehow superseded, as no analysis was performed regarding how many messages successfully arrived at each mote. Rather, the protocol itself works under conditions in which the network is within 70% of its stated maximum, even if the actually bandwidth is much lower.

I next consider the case in which the physical path between two sensors is still available, but no network communication is possible. I used similar methods as with the physical blockage case, but instead created a new network routing protocol which acts as a hybrid “proactive/re-active” protocol based on my Geographical DSDV. As described in the previous chapter, if a sensor does not receive any messages from an adjacent sensor – which, based on its physical knowledge of the layout, it expects to receive – the sensor makes an explicit request from nearby sensors for any information they receive about that missing sensor. I measure how many messages are required between the time the network communication is blocked between those two sensors and when the original sensor received its missing data and placed the previous missing sensor back on its path. As seen in figure 4-9, the costs of recovering from the case of a blocked network connection is high, compared to the cost of recovering from a physically blocked passageway. I assume that part of the reason for this is because the rerouted data sent to the receiving sensor indirectly is competing with all of the DSDV data already being repeatedly sent in the background. However, from this expanded perspective, the cost of recovering from a physically blocked passageway does not seem as noisy as it does in figure 4-8. In fact, both recovery schemes seem to scale well with network size, and perhaps the fluctuations can be attributed to noise.
and changing conditions within the simulations.

\section*{4.9 Conclusion}

This section has demonstrated the need for and described the design of new tools that allow users to create new scenarios with which to design and test applications for sensor networks. Users need to be able to design test scenarios quickly without building new sensor network systems from scratch. The class of “Distributed-in/Distributed-out” applications provides this structure, which allows users to make these changes and redesigns quickly and easily.

Different DiDo applications will require different locations of resources and their layouts, different reactions to those resources, and different protocol actions. These different scenarios will all cause the changes in sensor actions, and the simulator needs to receive new instructions for each new scenario. Since it is burdensome to expect users to do this manually, it is best to create an application that interfaces directly with the simulator itself.

While in Chapter 3, I described the framework for DiDo, here I have shown how tools designed to work within this framework allow users to customize their
applications and measure the performance of these DiDo systems. Because current simulators are not designed around DiDo principles, the application integrates with TOSSIM to enable users to test their own DiDo-based applications and measure their performance.

The ability to quickly redesign systems and scenarios and evaluate them allows one to properly quantify the efficacy of the system and the “Distributed-in/Distributed-out” architecture in general.

Results are steady when it comes to the recovery performance until the network becomes flooded due to increased density. Keeping density constant could be accomplished by dynamically adjusting the communication radius. Since the ultimate goal is to integrate this with a pedestrian simulation, it is important to keep the performance data in mind – performance data that appeared scalable even when the network size was doubled before network effects took over – when motes are providing instructions to a simulation of the physical world.
Chapter 5

Pedestrian Simulation

5.1 Introduction

The next important piece of the evaluation is the effectiveness of the network as it applies to the real world. Without real-world deployment with actuators, I need to examine some other way of demonstrating the effectiveness of directing actuators with a sensor/transducer system. To this end, I built another major simulation tool which can simulate actuators being directed by the sensors. In this case, I created a pedestrian simulator in which the pedestrians accept instructions from the transducers in the simulator, and the presence of the actuators changes the sensor readings in the sensor simulator, which in turn changes the instructions that are output by the transducers.

The development of the Movement Simulator came from a desire to evaluate the showcase DiDo application of building escape. The model I have designed has pedestrians being directed by the sensors towards the exit. Meanwhile, I wish to evaluate the efficacy of the transducers’ directions to see how well they redirect pedestrians and improve the throughput of pedestrians to the exit. To this end, I need an accurate – or at least believable – model for pedestrian movement. Furthermore, this model needs to be integrated directly to the sensor simulations. Here I describe the models of the Movement Simulator and the architecture of the system and how it integrates with the sensor and network simulation.
5.2 Previous Work in Pedestrian Simulation

Simulating the behavior of pedestrians is an active research area, whose diverse applications include areas from park management [34] to room escape [40]. Pedestrian behavior is affected by the speed and direction at which the pedestrian wishes to travel, and the sum of all forces exerted upon him by other pedestrians and the walls. This force model is given by

\[
\frac{d\mathbf{v}_i}{dt} = \frac{m_i}{\tau_i} (v_0^i(t) e_0^i(t) - v_i(t)) + \sum_{j=\neq i} f_{ij} + \sum_w f_{iW} \quad (5.1)
\]

where a pedestrian \( i \) has a mass \( m_i \) and a desired velocity \( v_0^i \) in the direction \( e_0^i \). \( v_i \) is the pedestrian’s actual velocity, and \( f_{ij} \) and \( f_{iW} \) are the forces of each other adjacent pedestrian, \( j \), and the walls adjacent to pedestrian \( i \), respectively. In some cases, the sum of the forces exerted on the pedestrian is sufficient to cause injury or death, and some pedestrian simulations [40] take these possibilities into account in order to predict how dangerous an environment is.

Modeling the precise physical interactions between pedestrians is beyond the scope of this work. Rather, I seek to come up with a simple means of modeling pedestrian traffic through a floorplan and to gain an understanding about how crowds of pedestrians affects their ability to travel. To this end, I examined two different pedestrian scenarios – pedestrian travel through hallways (in my model, the edges of the graph) and pedestrian travel through rooms (in my model, the nodes of the graph). The initial hypothesis was that pedestrians traveled with a velocity \( v_0 \) in isolation, but their velocity would change according to a recognizable function of the number of other pedestrians in that hallway or room. Using the pedestrian simulator developed by [40], I ran a number of simulations of pedestrian escape times out of a simulated room to see how pedestrian throughput was affected by the number of other pedestrians in the room. (Figure 5-1) Looking at the overall throughput of pedestrians given different initial starting values for the number of pedestrians in the room, one sees that the time it takes for a room to completely empty is linear, and the average
Figure 5-1: A simulation of pedestrians escaping through a room in the Panic simulator.

Figure 5-2: An examination of escape times out of a room, given the starting number of pedestrians for that simulation.
throughput of pedestrians in all cases is relatively constant (Figure 5-2). This provided an indication that there was no real difference in velocity as a function of the number of pedestrians. Rather, pedestrians only took a longer time to escape out of a crowded room because there were so many other pedestrians in front of them. The more pedestrians in the room, the longer a pedestrian would have to wait to leave, similar to if he was standing in a queue. To confirm my suspicions, I re-analyzed the data to examine the average amount of time it took a single pedestrian to escape a room given that there were X pedestrians in the room at that time. For example, in the case of how long it took a single pedestrian to leave when there were 50 pedestrians in the room, I looked at all simulations of between 50 and 100 starting pedestrians in the room and took the average of the amount of time it took for the number of pedestrians in the room to go from 50 to 49. The results are similar (Figure 5-3) in the case of these “marginal” escape times – the time to escape is unaffected by the number of pedestrians in the room at that time. The only difference is at the ends of the graph – for 1 pedestrian and 99-100 pedestrians. In the case of 1 pedestrian, the simulation ran longer because the single pedestrian starts at the middle of the room, adding simulation time while the pedestrian travels to the exit. In the case of 99-100 pedestrians, there were only a few cases in which there were this many pedestrians in the room, and there were few data points to average. Once again, the time it took for the initial pedestrian to leave dominated the average, distorting the results, whereas in all other cases, the average escape time was relatively constant, regardless of how many pedestrians were in the room.

As a result, I concluded that a simple yet effective way to model pedestrian escape from a room was as a queue. The pedestrian at the head of the queue waits a certain interval before escaping, removing himself from the queue and entering the hallway. The next pedestrian in the queue then begins to wait a certain interval before he, too, is removed, and so on, until the queue is empty.

By contrast, I did not find many crowding effects in the hallways. In addition, the behavior of the pedestrians, squeezing out of a room one by one in a panic situation, means that the pedestrians will have minimal physical interactions once they are in
the hallway.

5.3 The Movement Simulator

5.3.1 Architecture

To get some initial ideas of how well a Movement Simulator performed during an evacuation task, I constructed a Movement Simulator whose pedestrians performed in a manner similar to what I measured in more the physics-based panic simulator. Since the movement simulator was intended ultimately to be integrated with the network simulator, similar components were used (or reused) when building it. I first describe the Movement Simulator as a stand-alone component (Figure 5-4) and explain the decisions that were made when designing it and justify the Movement Models that I chose. Next I discuss how the Movement Simulator is integrated into TOSSIM and compare the different methods of taking and comparing data. Because the Movement Simulator will eventually be integrated with the TOSSIM simulator, the data structures used to build the geographical layouts were the same as those used for the Map Generator. Ultimately, when both the sensor simulator and the
Movement Simulator are integrated, the Map Generator will create layouts that are shared by both modules. The Movement Simulator was originally constructed as a multi-threaded system with each of the pedestrians acting independently, while all of their movements were tied to a global clock, but more consistent results were gathered when the global clock controlled each individual pedestrians (Figure 5-5).

The distance between any two connected nodes is always considered to be equal within the graph. Support could be added to model distances of different lengths and how this would affect the calculation of the “best path” between a node and the exit, but this is tangential to the evaluation. The current model’s results remain valid without loss of generality; the floorplan would simply be modeled as a weighted graph, and shortest paths to the resource node would be calculated using a minimum-weight spanning tree rather than a shortest-path spanning tree.

5.3.2 Representing Floorplan Layouts as a Graph

The layout model is as follows: each node is a room. Of all of the node’s edges, one edge is designated as the exit edge, based on which edge is part of the shortest path to the resource node. A pedestrian in a room wishes to reach the resource node as
quickly as possible, so the pedestrian chooses the exit edge assigned to that node.

5.3.3 Pedestrian Behavior

As described above, the pedestrian-escape simulations showed that most of the delay in room escape due to people in the room occurs because each pedestrian needs to squeeze through the doorway. Instead of modelling the precise physical interactions between pedestrians, I simply place each new, arriving pedestrian into a queue for a certain edge. The arriving pedestrians must get into a queue behind the other pedestrians and wait until they leave the node over that edge before that pedestrian can exit. This queue models the crowding around an exit, as shown in the Panic simulator.[40] Regardless of how long the queue is, pedestrians leave the node through the exit edge at the same rate.

However, if too many pedestrians are waiting in a queue over an exit edge, then the doorway to an edge is considered “blocked”; the pedestrian will then decide to “flee” and seek out an alternate route to the exit. The pedestrian does so by choosing random edges from each node for 5 steps from one node to the next. At this point, the pedestrian will either have reached the exit, or the pedestrian will return to always choosing the designated “exit edge” each time he enters a node.

The consequences of crowding to the problem of escape are clear – if no edges
are ever blocked, then pedestrians proceed normally through the graph, taking the shortest path from their start nodes to the resource node. On the other hand, if edges are consistently blocked, some pedestrians will flee throughout the graph at random, over and over again, stymied by both the blocked passageways and their inability to find an alternate path to the resource node.

## 5.4 Pedestrian Escape Performance

### 5.4.1 Evaluation Tools and Scenarios

To get an idea of how quickly pedestrians could escape to the resource node under various circumstances, I used many of the same components from the automated scripts for TOSSIM evaluation. Specifically, I used the same random graph generator with which the Movement Simulator calculates the shortest paths from each node to the resource node. The Movement Simulator then takes a set number of pedestrians and seeds those pedestrians into the nodes at random. I then measure how long it takes all of the pedestrians to escape.

In the evaluation, I examine how quickly all of the pedestrians can evacuate when pedestrians are distributed randomly through the graph according to a uniform distribution and a power-law distribution. The latter distribution will result in initial conditions in which a large number of pedestrians are gathered in a single node in the graph. I took performance measurements for initial pedestrian distributions of 50 to 500 pedestrians in graphs of between 10 and 20 nodes.

### 5.4.2 Results

To get an idea of how the pedestrian simulation performed as the number of pedestrians rose, I ran several trials with randomly generated graphs of a fixed size, and seeded the graph with an increasing number of pedestrians. One of the possible scenarios considered was how building escape would be affected in the event that pedestrians within the building were holding a meeting, and thus crowded in a specific area. To
Figure 5-6: A comparison of mean escape time vs. number of pedestrians for graphs of 15 Nodes

simulate this scenario, I seeded the pedestrians in the graph according to a power-law distribution and compared that escape time to a uniform distribution of pedestrians in the graph (Figure 5-6). As one can see, the power-law distribution of pedestrian results in consistently higher escape times than a uniform distribution of pedestrians through the graph.

Variance is also higher in the case of power-law distributed pedestrians (Figure 5-7). This is perhaps due to the fact that crowding of pedestrians can occur in a location close to or very far from the their destination of the resource node and also because the act of crowding is disruptive; thus crowding pedestrians in one place, as power law distribution of pedestrians will do, creates much more unpredictable behavior, increasing the variance.

Holding the number of pedestrians constant and evaluating the performance over a changing number of nodes in the layout, there are more ambiguous results. When crowding is low, such as when there are only 50 nodes in the graph (Figure 5-8), the graph size predominates in determining escape time for the pedestrians. However, as the number of pedestrians expands to 250 pedestrians (Figure 5-9) or 500 pedestrians (Figure 5-11), the effects of crowding predominate, and the size of the graph clearly plays less of a role in determining the mean escape time than the number of pedestrians present. In all cases, however, there are consistently higher escape times in the
Figure 5-7: A comparison of the variance of uniformly distributed pedestrians and power law distributed pedestrians with linear interpolation for graphs of 15 nodes.

Figure 5-8: A comparison of escape times vs. nodes in the graph layout for graphs with 50 pedestrians.
Figure 5-9: A comparison of escape times vs. nodes in the graph layout for graphs with 250 pedestrians.

Figure 5-10: A comparison of escape time variance vs. nodes in the graph layout for graphs with 250 pedestrians.
Figure 5-11: A comparison of escape times vs. nodes in the graph layout for graphs with 500 pedestrians.

case of power-law distributed pedestrians. Variance is consistently lower when the graph size expands while the pedestrians remain constant (Figure 5-10), as a result of lower pedestrian density.

5.5 Integration with TOSSIM

To integrate with the TinyOS Simulator, I created a separate plugin that can be controlled directly by Tython scripts. I refer to this as the “DirectPedestriansPlugin.” Most of its functions are similar to the Movement Simulator. However, the plugin specifically integrates with the TinyOS simulator to interact with the motes. The integration of the Movement Simulator with TOSSIM then completes the loop of DiDo networks – the events in the Movement Simulator affect the simulated physical world, which changes the sensor readings and causes the sensors change the directions they provide. The Movement Simulator takes the directions provided by the motes and uses them to redirect the pedestrians in the appropriate direction. I discuss the structure of this integration and how performance is affected in the next chapter.
5.6 Conclusion

While the more full-featured applications that model pedestrian interactions provide a useful guide, the behavior of pedestrians can also be modeled with simple expressions of pedestrian behavior to get similar results. Pedestrian simulation is a large and open field, but for my purposes, I must model the relevant issues that are going to be confronted in a DiDo network in order to show how effective the system is – namely, I need a model of the physical world in which agents there react to changing conditions. In this stand-alone model, pedestrians make decisions based on static knowledge given to them at each node in the graph and the state of the node they visit due to crowding.

As one can see, crowding has a significant effect on the ability of pedestrians to reach the resource node. While a larger number of pedestrians results in longer escape times, the specific effect of creating crowds of pedestrians in a single area of the layout due to power-law distribution of pedestrians shows consistently higher escape times.

If intelligent sensors can influence the reactions of the agents – in this case, the pedestrians – then one can measure whether the those influenced reactions result in better escape times than the decisions that the pedestrians make on their own. If intelligent sensors can relieve the crowding effects, then one can expect to see an improvement in performance.
Chapter 6

Results and Evaluation

6.1 Introduction

In previous sections, I developed a protocol that determines shortest paths to a resource node in a geographical network. Next, I developed a system for pedestrian simulation that relies on the same structure for geographical representation. I now seek to demonstrate the efficacy of DiDo when working in tandem with the pedestrian simulator. It has been stressed that the principles of DiDo are that the physical world is detected by the sensors, which provide instructions based on their data to actuate and direct objects in the physical world, whose reactions are detected by the sensors.

Here, I realize that final merger of the physical and the network worlds. As the network simulation was done using TOSSIM, I integrated the Movement Simulator into a TOSSIM plugin and then developed a system to control the plugin automatically. The physical state of the Movement Simulator directly changes the readings of the sensors, and the pedestrians in the Movement Simulator take their instructions from those sensors which determine their directions based on their sensor readings and information they receive from other sensors.

The core of the evaluation is the comparison of the performance of the original Movement Simulator with the performance of the Movement Simulator when the pedestrians’ movement decisions are augmented with instructions from the sensors. I then compare the performance of escape times for the pedestrians with their escape
6.2 Integration of the Movement Simulator and TOSSIM

Combining the Movement Simulator with a network simulator is eased by TOSSIM’s support for plugins, and the Movement Simulator was designed so that it could take commands from a plugin. This is the “DirectPedestriansPlugin” which can take commands from within the Tython shell (Figure 6-2). The plugin also can directly change the value of the sensor readings of each mote, and the “outgoing edge” listed at each node by mirroring the connections made by the motes to what they consider their best destination. The architecture of this integration is below.

When a pedestrian enters a node, the pedestrian picks an outgoing edge from which to exit the node and gets in the queue for that outgoing edge. In the event that the pedestrian is following the shortest path instructions rather than “fleeing,” the pedestrian picks the outgoing edge listed at that node as the “best edge” to use. In the case of the integrated DiDo simulation, the DirectPedestrians plugin has this “best edge” assigned to it by the motes in the TOSSIM simulator (Figure 6-
1). When the TOSSIM simulator displays the directed graph indicating the current pathways indicated by the motes, DirectPedestriansPlugin detects this data output by the TOSSIM simulator and reassigns the “best edge” settings on the indicated nodes. The DirectPedestriansPlugin then looks at all of the edges of its nodes and if the queue of pedestrians becomes too long on an edge (in this case, 10 pedestrians), the plugin sets the appropriate sensor reading on the mote at that location to indicate that the edge is blocked. A blocked passageway sensed by a mote causes the mote to reconfigure itself and find an alternate passageway.

Figure 6-2: The Movement Simulator plugin integrated with TOSSIM

It should be noted that the two simulators were developed independently. This model provided certain problems as the TOSSIM simulator used its own clock to manage the actions of the motes and their timing behavior while the DirectPedestriansPlugin uses its own internal clock to mark the passage of time and manage the movement of pedestrians. The system worked while mote communication and
reactions occurred much faster than pedestrians movements between nodes. However, when the system was simulated using certain high-performance computers, this became no longer true, upsetting the balance necessary to produce valid results. To ensure consistent performance, DirectPedestriansPlugin is tied to the TOSSIM clock so that the pedestrians move and the messages are sent in sync with each other.

When the simulation begins, the simulation script instructs DirectPedestriansPlugin to load the map and seed it with pedestrians, either according to a uniform distribution or a power law distribution. The network simulation then begins and runs until the system finishes the “setup stage” in which the motes create their initial shortest paths toward the resource node. Once the system is set up and the initial exit edges are designated at each node, the pedestrians are directed to start moving. Once all of the pedestrians arrive at the resource node, the simulation terminates, recording the number of time clicks in DirectPedestriansPlugin. In this manner, one can directly compare the performance of the DiDo-augmented Movement Simulator to the stand-alone Movement Simulator.

6.3 Performance Comparison

I ran several simulations of escape scenarios. To get an idea of how escape times with DiDo integration compared to the stand-alone case, I kept the number of nodes fixed at 15 while increasing the initial number of pedestrians, in line with earlier experiments with the stand-alone case. I ran these simulations for both the uniform distributed and power law cases, just as I did in experiments with stand-alone Movement Simulator.

At first, I attempted to see if there was any difference in performance according to bandwidth consumed by the network protocol. The metric I used here was the number of messages sent in the time it took one pedestrian to exit from the queue. The assumption is that if more messages are sent during an interval in which the queue changes length, the network may be able to react more quickly to changes and there would be a greater performance improvement because of these faster reaction.
Figure 6-3: Performance improvement of DiDo using increasing bandwidth settings.

Setting the pedestrian exit time at 1 second, I was able to measure the performance improvement in which each mote sent 1 message in the time it took one pedestrian departure, 2 messages per pedestrian departure, and 3 messages per pedestrian departure. Looking at figure 6-3, taking the stand-alone case as a baseline, the performance improvements are about the same. DiDo messages can have a size of up to 75 bytes, so this performance data shows that for motes consuming between 600 to 1800 bits per second, performance is about the same. For the following experiments, a performance setting of 2 messages per pedestrian departure was used.

The first set of experiments examines the results of pedestrian escape performance for the case of uniformly distributed pedestrians (Figure 6-4). As one can see, the performance in the DiDo-augmented case results in consistently improved performance. The DiDo-augmented case is compared to the “Global Knowledge” case in which each pedestrian, when faced with a decision, chooses the fastest route to the exit at that given moment. Note that this is not a global optimum, simply the result if every pedestrian makes what would be the “best” decision whenever the pedestrian is presented with a choice. One can see that the decisions provided by a DiDo system—facing issues such as delay, dropped packets, and decisions marking a path as “blocked” based only on the existence of more than 10 pedestrians in a queue—
Figure 6-4: A comparison of the mean escape times for the Movement Simulator in the stand-alone case and DiDo-augmented case for uniformly distributed pedestrians.

Figure 6-5: A comparison of the variance of escape times for the Movement Simulator in the stand-alone case and DiDo-augmented case for uniform distributed pedestrians.
compares quite well when compared to the “Global Knowledge” case, which faces none of the communication issues and makes more accurate estimations of delay with respect to reaching the exit. In the latter case, the pedestrian has all knowledge of the conditions in the entire layout at the moment the pedestrian makes the decision about where to turn. As a baseline, this is compared to the mean escape time when pedestrians escape at their maximum possible bandwidth. Seeing how well the system compares to the case of “Global Knowledge” provides insight into why there was not much performance improvement in the three different bandwidth scenarios shown in figure 6-3. “Global Knowledge” means that, effectively, data is propagating across the network instantaneously, providing each pedestrian with the best data he can get at the instant he makes his decision. Given that the quality of decisions made with data that must propagate across the network at rates of less than 2 kbits/sec, compares well with the “Global Knowledge” scenario, it is clear that there is not that much room for improvement. There simply isn’t much more to be gained by increasing the bandwidth further, which is why the results from figure 6-3 did not see much change even when bandwidth was tripled.

Next, I consider the case of power law-distributed pedestrians. As can be seen in figure 6-6, the improvement in escape time is much more consistent in the case
Figure 6-7: A comparison of mean escape time variance for the Movement Simulator in the standalone case and DiDo-augmented case for power law-distributed pedestrians.

Figure 6-8: A Comparison of mean escape times by graph size, with 250 power law-distributed pedestrians, with mean values.
of power law-distributed pedestrians. The cases of the pedestrians receiving static instructions vs. the DiDo-augmented instructions diverges much more quickly and much more substantially in the power law-distributed case than in the uniform distributed case. This is because the effects of crowding are felt much more immediately in the case of power law-distributed pedestrians. The comparison of variance between the two power law-distributed cases is also stark (Figure 6-7). When applied to issues such as pedestrian escape, many of the benefits of DiDo-augmentation occur due to the relief of crowding. As such, much of my focus in the evaluation of DiDo networks will be on the case of power law-distributed pedestrians. Continuing with the case of power law-distributed pedestrians, while there is an improvement from DiDo across the board for graphs of varying size (Figure 6-8), escape time appears to be more dependent on the number of pedestrians than on the graph size, though for very large graphs, the graph size will start to dominate eventually. Since crowding will be less of an issue in that case, DiDo would not have much to do.

The DiDo instructions do not completely eliminate the penalty caused by power law distribution of pedestrians, however (Figure 6-9). This penalty still exists, as it does in the case of static instructions which results for the stand-alone Movement Simulator were observed. However, the benefits of DiDo instructions demonstrate a
clear advantage over the static instructions.

6.4 Stability

DiDo transducers act by continuously updating their information about the state of the region and changing their output results to redirect the agents in the system towards their goal. This raises questions about stability. For example, it is possible that a pedestrian in the system described above could be directed down one passageway and then, when arriving at the next node, be redirected back down the passageway he came from if crowding conditions change in the interim. One argument against such oscillations becoming a problem is that the system is inherently damped– as pedestrians reach the exit, they leave the system, reducing the amount of crowding that will occur, and making these oscillations due to crowding less likely. However, this hypothesis needs to be quantitatively demonstrated in order to verify it. To this end, I have examined some important metrics that will provide indications about the relative stability of the system. These metrics are measured over the time of the simulation and are the number of remaining pedestrians in the system at a given moment in time, the number of queues occupied by waiting pedestrians, the average size of these occupied queues, and the average distance of the pedestrians from the exit node. If there are problems with oscillations, then the results should show average queue sizes that are expanding and contracting in size rather than steadily decreasing, and the pedestrians’ average distance from the exit node will also fluctuate rather than steadily falling.

First, examining the standalone case (Figure 6-10), over time one can see the number of pedestrians remaining in the system steadily declines. Normalizing for total time for the cases of power law-distributed pedestrians and uniform distributed pedestrians, one sees that the slope of the curve for uniform distributed pedestrians is steeper, while in the case of power law-distributed pedestrians, the number of active pedestrians declines more slowly. That is to be expected, as it was the case of power law-distributed pedestrians that resulted in longer escape times. In the
Figure 6-10: A comparison of the number of active pedestrians remaining in the system over time in a standalone pedestrian simulation.

Figure 6-11: A comparison of the number of active pedestrians remaining in the system over time in a DiDo-augmented pedestrian simulation.
Figure 6-12: Queue evolution of the standalone pedestrian simulation over time with uniform distributed pedestrians

DiDo case (Figure 6-11), one can see a similar dynamic, but the declines in the uniform and power law-distributed case are steeper. Furthermore, the decline is steady. A system with a large number of oscillations would find graphs of the number of active pedestrians to have steep falls followed by shallower or flat points along the curve. This pattern does not appear, indicating DiDo does not seem to cause unstable fluctuations, at least on average.

Looking at the evolution of the queues over time provides a finer-grain understanding of the evolution of the system. I examine both the number of occupied queues in which pedestrians are waiting and the average size of those occupied queues. The cases of power law-distributed pedestrians and uniform distributed pedestrians provide mirror-image results: in the power-law case, there are a few queues with a very long queue sizes, while in the case of uniform distributed pedestrians, there will be many occupied queues with smaller average sizes. One can see this clearly when comparing the evolution of the queues for the standalone simulation in figures 6-12 and 6-13. When DiDo-augmentation is applied to the pedestrian simulation, the results are stark. First, looking at the case of power law-distributed pedestrians, remember that the power-law distributed case showed more improvement when DiDo is applied, and this examination of queue evolution provides and indication as to why. Fig-
Figure 6-13: Queue evolution of the standalone pedestrian simulation over time with power law-distributed pedestrians

Figure 6-14 shows how DiDo disperses pedestrians much more rapidly than they would otherwise. The mean occupied queue size is much lower in the DiDo case, and the number of occupied queues remains larger (though steady). The effect of pedestrian dispersal to other queues can be seen more clearly in the case of uniform distributed pedestrians (Figure 6-15). One can see greater fluctuation in the number of occupied queues. The data for the number of occupied queues appears noiser. However, the average size of those queues declines more quickly than in the standalone case, as DiDo predicts—what the system is doing is distributing pedestrians throughout the graph, though this results in perturbations in the number of queues occupied. The sequence of events seems to work as congestion occurs, distribute pedestrians to more queues to reduce congestion, redistribute when congestion occurs again. Meanwhile, the pedestrians are emptying out of the graph as they steadily and inexorably fan in to the exit.

So far, indications are that DiDo does not cause large instabilities. One remaining issue to explore in this space is the average distance of pedestrians from the exit over time. One can imagine that a poorly-designed system would send pedestrians on a “wild goose chase” searching for the exit as instructions constantly change, making little progress or even being sent off in directions more distant from the exit. Keep
Figure 6-14: Queue evolution of the DiDo-augmented pedestrian simulation over time with power law-distributed pedestrians.

Figure 6-15: Queue evolution of the DiDo-augmented pedestrian simulation over time with uniform distributed pedestrians.

138
in mind, however, that some apparent additional distance from the exit may be expected. After all, the goal of DiDo in this application is to send pedestrians on a path that will lead them to the exit the fastest. In some cases, the pedestrians may take a route that is geographically more distant in terms of the number of steps to the exit, though the pedestrian will arrive there more quickly. Looking at the standalone case (Figure 6-16), the “clumping” of the pedestrians in a power law-distribution appears to, on average, result in an average distance greater than the uniform distribution of the pedestrians, though their average distances conform towards the very end of the simulation (once again, the graphs are normalized for simulation time). Average distance falls slowly and then suddenly declines precipitously. By contrast, the DiDo-Augmented case in figure 6-17 demonstrates some interesting, but ultimately predictable behavior. The simulations of power law-distributed pedestrians and uniform distributed pedestrians converge more quickly, matching behavior from about 90% through the simulation onwards. Also, there are visible fluctuations and perturbations in the uniform distributed case between 70% and 90% of the way through the simulation, indicating that pedestrians are being directed to nodes further from the resource, perhaps because most of the pedestrians have reached a location right next to the resource node, and the transducers adjacent to the resource node are sending pedestrians to other neighboring locations. Since there are few pedestrians left at this point, and they are all close to the resource node, any change in their location further from the resource node will result in a larger change to the average distance. Note that measurements of distance do not start getting taken until slightly into the beginning of the simulation, explaining why the pedestrians in the DiDo-augmented simulation appear to start off closer to the exit than in the stand-alone simulation. In fact, in both simulations, they have had a chance to move forward, but in the DiDo-augmented case, they have moved forward closer to the exit more quickly by the time measurements start getting taken. The most important aspect of these graphs when examining stability is the structure of the curves.

As predicted, DiDo operates effectively because the continuous “escape” of pedestrians means that energy is effectively being constantly removed from the system,
Figure 6-16: Average distance from exit in the standalone case

Figure 6-17: Average distance from exit in the DiDo-augmented case
so any oscillations or instabilities in this control system are transitory and show up only in very limited cases, on average. The improvements are very real. In an escape scenario, one could view the issue of “stability” as a genuine concern because there may be some psychological impact on escapees if they are being constantly forced in oscillating patterns. Fortunately, as one can see, this does not appear to be a problematic symptom of the system.

6.5 The Case of Two Exits

Realistic floorplans, of course, do not have a single exit that pedestrians are attempting to get to. In fact, floorplans have two or more separate exits. The question becomes whether DiDo systems improve the throughput of pedestrians through the separate exits. Because of the way that the protocol works, if one exit is crowded and blocked by outgoing pedestrians, in theory the DiDo transducers should direct pedestrians to the other exit if the other exit is less crowded. To evaluate the performance of DiDo when handling two exits, I designed a simple experiment. First, I created a random 15-node graph, similar to the ones used in the above examples. Next, I created a new, 16th node and designated that node to be the “exit” which was connected to two other nodes. Those two nodes are each considered an exit. For a one-exit case, I simply deleted one edge to the 16th node.

First, consider the case of a set of uniform-distributed pedestrians in a graph, first with one designated exit, and then with a second designated exit. Because the pedestrians are distributed uniformly, the creation of a second exit will direct about half the pedestrians to the new exit, significantly increasing the exit throughput of the pedestrians, as they are directed to another exit rather than crowding the original exit. As can be seen in figure 6-18, the performance when another exit is added is what one would expect. The average performance improvement when a second exit is added is 36%, with improvements as large as 52%, in the case of situations where pedestrians are more sparse and as small as 15% when there is much more crowding and “panic” effects.
Figure 6-18: Mean exit time of uniform-distributed pedestrians out of a 15 node graph with one exit versus two exits.

Figure 6-19: Mean exit time of power law-distributed pedestrians out of a 15 node graph with one exit versus two exits.
Figure 6-20: Mean exit time of power law-distributed pedestrians out of a 15 node graph with one exit versus two exits, using DiDo

In situations such as the power law-distributed pedestrians, not only will crowding be the norm, but most pedestrians will be distributed in an area which favors one particular exit. With the stand alone pedestrian simulator, only static instructions towards the exits are being given to the pedestrians. Thus, if all of the pedestrians reside next to exit 1, they will follow the instructions towards exit 1, even though exit 2 might be all clear. As a consequence, adding another exit does not result in as dramatic a performance improvement, as seen in figure 6-19. The average performance improvement with a new exit is only 14%, though results range from as much as a 46% improvement to a 24% performance penalty. Clearly, the place where DiDo can help the most is in the case of power law-distributed pedestrians, and this case presents a clear example of where the use of dynamic directions can help because the instructions will direct pedestrians to a different exit than the one that is physically closest.

I established earlier that DiDo-augmented systems can improve pedestrian throughput even when the underlying layout is the same. A one-exit floorplan will have better performance with DiDo then a one-exit floorplan without it. Likewise, a two-exit floorplan with DiDo will have better performance than a two-exit floorplan without DiDo, a one-exit floorplan without DiDo, and a one-exit floorplan with DiDo. At issue is not merely that DiDo will provide better throughput for pedestrians in a
two-exit scenario compared to the stand alone simulation; that is already known to be true. What is interesting is how DiDo accomplishes this. In the case of power law-distributed pedestrians, when another exit is added to the DiDo simulation, one can see a consistent improvement (Figure 6-20). The average improvement is 18%, better than the improvement in the stand alone case of power law-distributed pedestrians when another exit is added. However, when the DiDo-augmented simulation with two exits is compared to the stand alone simulation of one exit, there is a 35% average improvement when another exit is added with DiDo, compared to a 14% average improvement from adding another exit without DiDo. That is close to the 36% average improvement from adding another exit under the uniform distribution of pedestrians. This implies that DiDo spreads out pedestrians more evenly, allowing DiDo to take advantage of the new exit, in the same way that a uniform distribution of pedestrians would implicitly do.

To examine the mechanism by which DiDo created this performance improvement in the case of power law-distributed pedestrians, I instructed each pedestrian to record the last node they entered before arriving at the 16th exit node. The two possible “last nodes” connected to that 16th node serve as the two separate exits in the 15 node graphs. At the end of the simulation, I had a list of all of the “last nodes” taken by each pedestrian. Ideally, the pedestrians would be divided 50/50 across both

Figure 6-21: Performance in the two-exit scenario in a 15 node graph
of the exits. To evaluate the effectiveness of DiDo vs. the stand alone pedestrian simulation, I looked at how the exits taken by those pedestrians deviated from that 50/50 ideal. Examining the results in figure 6-21, if 90% of the pedestrians escaped through one exit and 10% through the other, then the deviation from the ideal would be 40%. On the other hand, if 55% of the pedestrians escaped through one exit and 45% through the other, then the deviation from the ideal would be 5%. Thus, lower is better. As can be seen in figure 6-21, DiDo-augmented pedestrians act closer to the ideal because, as predicted, when the area near one exit becomes crowded (which will occur especially in the case of power law-distributed pedestrians), the transducers direct the pedestrians to the other exit, driving the results closer to the 50/50 ideal.

6.6 Conclusion

The key to evaluating the system is integrating the physical simulation with the network simulation and controlling it from within the simulation environment. This seamless integration allows the user to run multiple trials and tests under different circumstances in the same way that one is able to test different scenarios of movement simulations and network simulations separately.

The advantages of DiDo are clearly demonstrated in the results. Consistently, one finds a decrease in escape time when crowding becomes a factor. Furthermore, credit for the decrease in variance in the DiDo-augmented case points to a decrease in “random behavior” caused by fleeing pedestrians. Under circumstances where crowding is not a problem, the results are much more ambiguous. The case of uniform-distributed pedestrians even showed a slight increase in total escape time. It is unclear whether this increase was the result of noise/random trials or a case in which the DiDo instructions were actually harmful.

Since the effect of distributing pedestrians using a power law distribution is to increase the amount of crowding, it should come as no surprise, then, that the advantages of dynamic DiDo instructions are greater in the power law case than the uniform-distribution case. The predicted crowding relief of DiDo works to allevi-
ate the consequences of crowding and panic, which would otherwise cause increased escape times.
Chapter 7

Conclusion and Further Work

7.1 Managing Sensor/Transducers

The TinyOS simulator provides enough flexibility for one to assume sufficiently intelligent transducers that are able to detect changes in the physical world. The DiDo framework allows users to decide what phenomena are being read in and how to react to those changes. In the sample case, the important phenomena to be detected are the presence of resources and whether passageways are blocked. Other features can be added such as representations of a slowed passageway.

Physical implementation of these transducers is also a concern. While one can depict the transducers as “sufficiently intelligent” in simulation, the important matter is to ensure that simulations do not assume transducers that are “arbitrarily intelligent.” All sensing assumptions made in simulation must be grounded in reality, lest the simulations lose touch with what can actually be determined. Future directions of this work involve designing transducers that can implement the basic functions that are proposed in simulation and providing a choice of various sensing functions to users working with the Map Generator application.

One example is the issue of localization. The system assumes that transducers can infer their own location in x,y space. This is actually a more difficult problem than the simulation assumes. Transducers cannot simply determine their location to a fine grain of accuracy without a separate infrastructure designed to provide this,
such as the Crickets.\[67\]

7.2 Wireless Communication

I have attempted to demonstrate that the protocol is one that functions independently of the actual wireless communication scheme used, though for the simulations I assume certain conditions based on existing physical deployments. Since the protocol makes decisions in software only to exchange data with the closest available transducers, this is obviously applicable to adaptive RF methods. I have measured the relative power consumption necessary under various non-adaptive RF models, but I have not explored how the system performs under adaptive RF conditions. Because of the increasing availability of adaptive RF communication, and because the protocol presented here is designed to take advantage of such features, it is worth exploring in the future what benefits adaptive RF methods possess. This requires simulators that can support such models – specifically, a simulator in which each sensor may well have a different radio model, which is quite different from many simulators, including TOSSIM, which assumes that each sensor sends out data using the same RF characteristics.

Furthermore, because the simulation work specifically deals with the interface of the physical world with the network world, the issue of how one handles the radio behavior of the transducers given the physical layout is an important one. I have already discussed in chapter 2 the behavior of SENS, which seeks to simulate how radio propagates in a variety of physical environments. More improved DiDo simulators will have to incorporate knowledge of the physical environment that is designed into the radio model, which will vary by sensor depending on its location. With these new radio models, I can evaluate the power advantages of adaptive RF when applied to DiDo protocols.
7.3 Scalability

My argument was that reconfiguration would be scalable, because changes made would be local. This assumption turns out to be mostly correct. While results are ambiguous and noisy when measuring recovery, any rises in messages required for physical recovery appear to be from network connectivity effects alone. When the network is not saturated from density, changes in direction specified by the transducers do not propagate far up the graph. Scalability depends on the average degree of the physical topology. In a simple ring topology, knowledge of an alternate path can take a long time to propagate after a passageway is blocked. This is because in a graph where $\delta = 2$, as in a ring graph, the removal of an edge will mean that the nearest information about an alternate path will be at the midpoint the ring, and thus the time it takes for the network to reconfigure will depend on its distance from the midpoint. This is expected, as the argument for scalability depends almost entirely on the principle that the distance to an alternate path is constant. In fact, the system appears scalable when attempting to recover from a blockage that is a fixed distance away from an alternate path.

At the same time, in the model of pathfinding in which transducers are hard-coded with knowledge of the layout, there is more flexibility and the system has the potential to scale both in terms of reconfiguration and in the introduction of new transducers for the purpose of adding detail. However, because the model depends on transducers that possess knowledge of the entire map layout, the model obviously will not scale in terms of memory required. The model depends on retaining knowledge of some geographical information. A designer can divide geographical knowledge into manageable pieces as map layouts get larger; this also assumes that each divided region will be expected to maintain enough transducers to operate. Whereas knowledge of the entire map layout allows the system to function with only two transducers in the entire map, giving the transducers only partial geographical data would mean that at least one other transducer would have to be functional within that geographical region in order to exchange data. Messages from functional transducers in the layout
that are outside of the receiver’s region of responsibility would be ignored. Improving scalability of data in the map-layout case demands that the communication protocol be changed to support communication with transducers outside the layout, as the current protocol assumes that all transducers have an associated location within the geographical layout.

Furthermore, use of the DSDV protocol, while scalable for recovery in terms of blocked passageways and failed transducers, would provide significant problems of scalability during recovery during a change in resource location– the entire map would have to be recalculated. While the active model of AODV discussed in Chapter 3 may not be the appropriate one because it also faces the same scalability problems during setup, other protocols that require low routing overhead while also not requiring the constant reinforcement necessary to cement the relationships and connections between transducers in their paths should be explored in greater detail for this application.

7.4 Actuation Models

My method of “Distributed-out” depended on the idea of using simple visual cues from the transducers. In the case of TOSSIM, these visual cues were represented by associating an LED with an instruction. The model was that these LEDs represented signals to actuators.

In the case of autonomous actuators, at issue is the problem of location estimation. Actuators such as remote-controlled cars cannot determine their closest transducer merely based on signal strength.[65] There are too many outside factors that make this difficult, particularly when transducers at different locations will be giving different actuation instructions, as the model describes. This is going to be a necessary component of any actuation model – how does an actuator choose which Distributed-output points to pick?

In addition, since other issues are involved with how DiDo networks are effective in improving the performance of physical systems, I would like to explore how much physical benefit is gained from having actuators directed by DiDo networks. I have
already demonstrated that these networks are scalable and able to adjust quickly to changing conditions. Would large-scale deployment of sensors and actuators result in improved performance over more centralized solutions? This topic needs to be explored in depth by specialists in actuation.

### 7.5 Simulation Features

The main features of the simulation system are the ability to design and make changes to the physical environment where the transducers will be deployed, generate scripts for the TOSSIM simulator to run, and generate code to change the behavior of the transducers. This is a very powerful tool in order to design and test sensor network applications.

Further extensions to these principles can be explored. Cross-layer protocols depend frequently on knowledge of the radio model. One may wish to allow the user to specify changes in the protocol depending on the physical conditions which affect the wireless protocol. As other sorts of applications that fit into the DiDo framework are defined, extensions can be added to the “Distributed-out” behavior. I have designed principles that depend either on direction or proximity to resources. Other possibilities exist that could be added when other proposed applications are formulated.

Furthermore, different principles of layout knowledge may need to be explored. I have already discussed the possibility of partitioning knowledge of the map layout among the transducers so that each one does not need to know about the entire layout, and instead designed a system in which each transducer only has knowledge of its specific region. Also, at the moment, TOSSIM only supports simulations in which all motes contain the same controlling code. One can experiment more with the ability to partition off physical knowledge of each transducer and handle heterogeneous code-generation when simulators support such features.

Script generation is a very powerful extension to TOSSIM, as redesigning a new script for each new scenario is quite time-consuming. Other extensions to this are support for tests that can deal with multiple resources. Still to be explored is whether
there are general principles of simulation script formulation that apply across DiDo simulations, rather than recalculating and designing a custom script for each and every new scenario. For now, however, the system functions well for evaluation purposes.

7.6 Additional Further Work

The system leaves several open issues available for exploration and improvement. These issues include the realism of the physical layouts, more intelligent decision-making by the transducers, and examination of the performance of other protocols when tasked with directing pedestrians. Furthermore, tighter integration of a physical and network simulator will ultimately be required.

First, while the issue of representing real floorplans as graphs was taken into consideration when designing test scenarios, the model is very simplistic. I consider connections between nodes to be of equal length and do not confront how decision-making by the transducers changes when connections between nodes are of variable length. Furthermore, the transducers do not currently make intelligent decisions regarding whether it is better for a pedestrian to wait in the queue for a crowded exit edge versus taking an alternate route. Transducers now only act to send pedestrians down uncrowded exits of a node that lead to the resource node. On the other hand, since DiDo performance comparably well when placed alongside the “Ideal Decisions” scenario, this indicates that the limited information collected by DiDo may be sufficient. That one still considers this an improvement indicates that simply reducing crowding and panic is important in and of itself.

Next, accounting for the physical environment by the network is not well-integrated into the network simulation. While I used an “empirical” model for network connectivity, and while this matched up well with what was known about actual physical deployments of motes in indoor spaces [3], the actual network connectivity itself was unconcerned with the physical layouts and connections with which I was experimenting, except insofar as they influenced the absolute distances between the transducers. This returns us to the sensor simulator model embodied by SENS.[78]
Finally, while one can make general conclusions about how pedestrian performance will be affected by different network protocols based on recovery time, it bears exploring how actual performance of pedestrian throughput is affected by network protocols. While much work has been done comparing the performance of data throughput using different routing protocols [12], additional work needs to be done in applying protocols other than the geographical DSDV protocol to the problem of directing pedestrians. Exploring the use of active protocols in this case may well be warranted.

These open issues provide several additional directions for exploration in the immediate future, now that DiDo itself has proven itself effective. Now that it works with simple physical models and simple decision-making algorithms, I feel that it is now worth the effort to explore further and build more realistic, well-integrated models.

### 7.7 Final Issues

I have presented the design of a new framework for thinking about sensor networks and shown how it could be evaluated and customized within this framework. Some parts of the evaluation and simulation are straightforward – monitoring network traffic, measuring power consumption, and determining the speed with which the network is able to adjust to changing conditions. On the other hand, more and more detailed simulations of the physical world present more complicated challenges. While a major open issue that has attracted strong interest is the ability to simulate the communication behavior of deployed sensor networks in mapped-out floorplans, the issue of how these sensor networks integrate with the physical behavior of actuators is much more obscure. First, while it is certainly possible and desireable to make basic integrations with well-known pedestrian simulation research in order to see if DiDo networks can effectively manage traffic flow, other DiDo applications may not have straightforward applications to existing physical simulations.

The solution is not always “more simulation.” An important future advance for DiDo networks is the creation of a framework for managing the interface between
sensors and actuators so that actuation can be tested quickly and easily in real-life deployment, rather than in simulation. Since my goal is to ease the ability to deploy sensor networks, the final step is a hardware/software framework that eases the ability to use sensor networks for use in a variety of physical actuators.
Bibliography


[7] Sandeep Bajaj, Lee Breslau, Deborah Estrin, Kevin Fall, Sally Floyd, Padma Haldar, Mark Handley, Ahmed Helmy, John Heidemann, Polly Huang, Satish Kumar, Steven McCanne, Reza Rejaie, Puneet Sharma, Kannan Varadhan,


[14] Claudio Castelpietra, Luca Iocchi, Daniele Nardi, Maurizio Piaggio, Alessandro Scalzo, and Antonio Sgorbissa. Communication and coordination among


[32] Lewis Girod, Jeremy Elson, Alberto Cerpa, Thanos Stathopoulos, Nithya Ramanathan, and Deborah Estrin. Emstar: a software environment for developing


